公众号：旧时光大数据

大数据面试题 V3.0

版本	更新时间
大数据面试题 V1.0	2021 年 9 月 14 日
大数据面试题 V2.0	2022 年 1 月 10 日
大数据面试题 V3.0	2022 年 4 月 30 日

公众号

微信

整理作者： 蓦然

版本说明： 预计一季度至半年更新一次

购买后一年内免费更新版本

Hadoop面试题

Hadoop基础

介绍下Hadoop

可回答：1）Hadoop是什么；2）Hadoop了解吗？3）Hadoop原理

问过的一些公司：字节×4，美团×2，美团(2021.08)x2，京东，阿里×2，bigo，海康威视，大华，猿辅导，360，蘑菇街，小红书(2021.11)，腾讯(2021.10)，四方伟业(2021.08)

回答技巧：说下Hadoop核心组件，讲下HDFS原理、MapReduce过程等，也可以在介绍完组件部分后，看看面试官会不会继续往下问（比如让你说下HDFS读写流程、MapReduce工作原理等），给面试官留点空间，这样回答下去，面试官能发掘你的技能点，你也能面的轻松些。如果没问了，就算了。当然你一下全说完，面试官有兴趣也可以。

参考答案：

1、先说下Hadoop是什么

Hadoop是一个分布式系统基础架构，主要是为了解决海量数据的存储和海量数据的分析计算问题。

2、说下Hadoop核心组件

Hadoop自诞生以来，主要有Hadoop 1.x、2.x、3.x三个系列多个版本；

Hadoop 1.x组成：HDFS（具有高可靠性、高吞吐量的分布式文件系统，用于数据存储），MapReduce（同时处理业务逻辑运算和资源的调度），Common（辅助工具，为其它Hadoop模块提供基础设施）；

Hadoop 2.x和Hadoop 3.x组成上无变化，和Hadoop 1.x相比，增加了YARN，分担了MapReduce的工作，组件包括：HDFS（具有高可靠性、高吞吐量的分布式文件系统，用于数据存储），MapReduce（处理业务逻辑运算），YARN（负责作业调度与集群资源管理），Common（辅助工具，为其它Hadoop模块提供基础设施）。

这里也可以先说各个系列的Hadoop组成，然后再说下HDFS、MapReduce、YARN和Common的作用（见“Hadoop主要分哪几个部分？他们有什么作用？”）。

Hadoop的特点

问过的一些公司：字节

参考答案：

1）高可靠性

Hadoop底层维护多个数据副本，即使Hadoop某个计算元素或存储出现故障时，也不会大致数据的丢失

2）高扩展性

在集群间分配任务数据，可方便的扩展数以千计的节点

3）高效性

公众号：旧时光大数据

在MapReduce的思想下，Hadoop是并行工作，加快任务处理速度

4）高容错性

能够自动将失败的任务重新分配

说下Hadoop生态圈组件及其作用

回答技巧：说一些就可以了，不用全部说完，如果问还有吗，知道就说，不知道就说不知道，别画蛇添足

可回答：对Hadoop生态的理解

问过的一些公司：字节，51，作业帮，阿里，好未来，唯品会(2021.07)

参考答案：

1）Zookeeper：是一个开源的分布式应用程序协调服务,基于zookeeper可以实现同步服务，配置维护，命名服务。

2）Flume：一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统。

3）Hbase：是一个分布式的、面向列的开源数据库, 利用Hadoop HDFS作为其存储系统。

4）Hive：基于Hadoop的一个数据仓库工具，可以将结构化的数据档映射为一张数据库表，并提供简单的sql 查询功能，可以将sql语句转换为MapReduce任务进行运行。

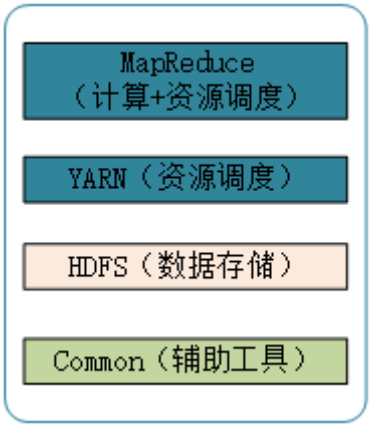
5）Sqoop：将一个关系型数据库中的数据导进到Hadoop的 HDFS中，也可以将HDFS的数据导进到关系型数据库中。

Hadoop主要分哪几个部分？他们有什么作用？

可回答：1）Hadoop的组件有哪些；2）Hadoop原理

问过的一些公司：字节x2，字节(2021.08)(2021.09)，美团x3，阿里，阿里(2021.09)，蘑菇街，好未来，携程(2021.09)，海康(2021.09)

参考答案：



Hadoop主要组件如上图，主要是HDFS、MapReduce、YARN、Common

HDFS

HDFS是一个文件系统，用于存储文件，通过目录树来定位文件。

其次，它是分布式的，由很多服务器联合起来实现其功能，集群中的服务器有各自的角色。

公众号：旧时光大数据

HDFS 的使用场景：适合一次写入，多次读出的场景。一个文件经过创建、写入和关闭之后就不需要改变。

MapReduce

MapReduce是一个分布式运算程序的编程框架，是用户开发“基于Hadoop的数据分析应用”的核心框架。

MapReduce核心功能是将用户编写的业务逻辑代码和自带默认组件整合成一个完整的分布式运算程序，并发运行在一个Hadoop集群上。

MapReduce将计算过程分为两个阶段：**Map和Reduce**

- Map阶段并行处理输入数据
- Reduce阶段对Map结果进行汇总

YARN

先来看两个问题，在Hadoop中

- 如何管理集群资源?
- 如何给任务合理分配资源?

YARN在Hadoop中的作用，就是上面两个问题的答案。Yarn 是一个资源调度平台，负责为运算程序提供服务器运算资源，相当于一个分布式的操作系统平台，而MapReduce等运算程序则相当于运行于操作系统之上的应用程序。

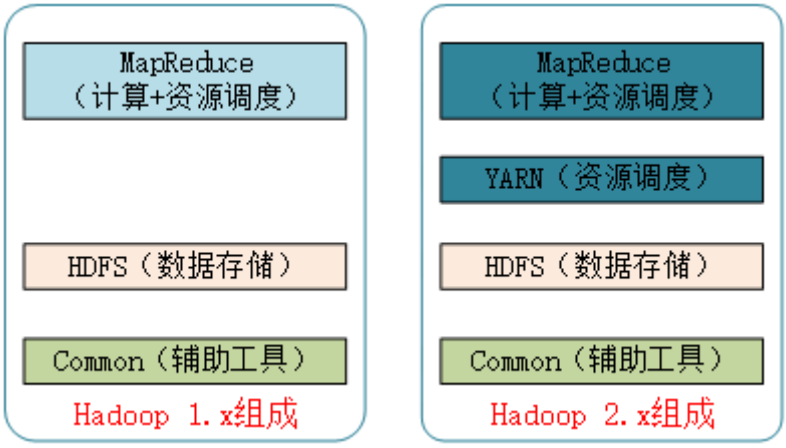
Common

Hadoop体系最底层的一个模块，为Hadoop各子项目提供各种工具，如：配置文件和日志操作等。

Hadoop 1.x，2.x，3.x的区别

问过的一些公司：深信服，小米，美团，阿里，米哈游，美团买菜(2021.09)

参考答案：



Hadoop 1.x阶段，Hadoop中的MapReduce同时处理业务逻辑运算和资源的调度，耦合性较大；

Hadoop 2.x阶段，增加了Yarn。Yarn只负责资源的调度，MapReduce只负责运算；

Hadoop 3.x相比于Hadoop 2.x阶段在组成上没有变化。

Hadoop集群工作时启动哪些进程？它们有什么作用?

可回答：Hadoop正常运行有几个进程

公众号：旧时光大数据

问过的一些公司：字节，祖龙娱乐，海康，京东(2021.07)

参考答案：

1）NameNode

就是Master，Hadoop的主管、管理者

（1）管理HDFS的名称空间；

（2）配置副本策略；

（3）管理数据块（Block）映射信息；

（4）处理客户端读写请求。

2）DataNode

就是Slave。NameNode下达命令，DataNode执行实际的操作。

（1）存储实际的数据块；

（2）执行数据块的读/写操作。

3）Secondary NameNode

Secondary NameNode并非NameNode的热备。当NameNode挂掉的时候，它并不能马上替换NameNode并提供服务。而是提供周期检查点和清理任务，帮助NN合并editslog，减少NN启动时间。

（1）辅助NameNode，分担其工作量，如定期合并Fsimage和Edits，并推送给NameNode

（2）在紧急情况下，可辅助恢复NameNode

4）ResourceManager（JobTracker）

整个集群资源（内存、CPU等）的老大。负责集群中所有资源的统一管理和分配，它接受来自各个节点（NodeManager）的资源汇报信息，并把这些信息按照一定的策略分配给各个应用程序（即ApplicationMaster）。

（1）与客户端交互，处理来自客户端的请求。

（2）启动和管理ApplicatinMaster，并在它运行失败时重新启动它。

（3）管理NodeManager，接受来自NodeManager的资源管理汇报信息，并向NodeManager下达管理命令（比如杀死Contanier）等。

（4）资源管理与调度，接收来自ApplicationMaster的资源申请请求，并为之分配资源（核心）。

5）NodeManager（TaskTracker）

NodeManager是YARN中单个节点的代理（单个节点服务器资源老大），它需要与应用程序的ApplicationMaster和集群管理者ResourceManager交互；它从ApplicationMaster上接收有关Container的命令并执行（比如启动、停止Contaner）；向ResourceManager汇报各个Container运行状态和节点健康状况，并领取有关Container的命令（比如清理Container）。

NodeManager管理的是Container而不是任务，一个Container中可能运行着各种任务，但是对NodeManager而言是透明的，它只负责Container相关操作，比如管理Container的生命周期，即启动Container、监控Container和清理Container等。

（1）管理单个节点上的资源

（2）处理来自ResourceManager的命令

（3）处理来自ApplicationMaster的命令

公众号：旧时光大数据

6）DFSZKFailoverController

高可用时它负责监控NameNode的状态，并及时的把状态信息写入Zookeeper。它通过一个独立线程周期性的调用NameNode上的一个特定接口来获取NameNode的健康状态。ZKFC也有选择谁作为Active NameNode的权利，因为最多只有两个节点，选择策略比较简单（先到先得，轮换）。

7）JournalNode

高可用情况下存放NameNode的editlog文件。

在集群计算的时候，什么是集群的主要瓶颈

问过的一些公司：美团(2021.08)

参考答案：

磁盘IO（正解），CPU，内存，网络带宽

搭建Hadoop集群的xml文件有哪些?

问过的一些公司：祖龙娱乐

参考答案：

`core-site.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>
    <!-- 指定 NameNode 的地址 -->
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://hadoop102:8020</value>
    </property>

    <!-- 指定 hadoop 数据的存储目录 -->
    <property>
        <name>hadoop.tmp.dir</name>
        <value>/opt/module/hadoop-3.1.3/data</value>
    </property>

    <!-- 配置 HDFS 网页登录使用的静态用户为 atguigu -->
    <property>
        <name>hadoop.http.staticuser.user</name>
        <value>atguigu</value>
    </property>
</configuration>
```

`hdfs-site.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>
    <!-- nn web 端访问地址-->
```

公众号：旧时光大数据

```
    <property>
        <name>dfs.namenode.http-address</name>
        <value>hadoop102:9870</value>
    </property>
    <!-- 2nn web 端访问地址-->
    <property>
        <name>dfs.namenode.secondary.http-address</name>
        <value>hadoop104:9868</value>
    </property>
</configuration>
```

yarn-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>
    <!-- 指定MR 走 shuffle -->
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>

    <!-- 指定 ResourceManager 的地址-->
    <property>
        <name>yarn.resourcemanager.hostname</name>
        <value>hadoop103</value>
    </property>

    <!-- 环境变量的继承 -->
    <property>
        <name>yarn.nodemanager.env-whitelist</name>
        <value>JAVA_HOME,HADOOP_COMMON_HOME,HADOOP_HDFS_HOME,HADOOP_CO
NF_DIR,CLASSPATH_PREPEND_DISTCACHE,HADOOP_YARN_HOME,HADOOP_MAPRED_HOME</value>
    </property>P
</configuration>
```

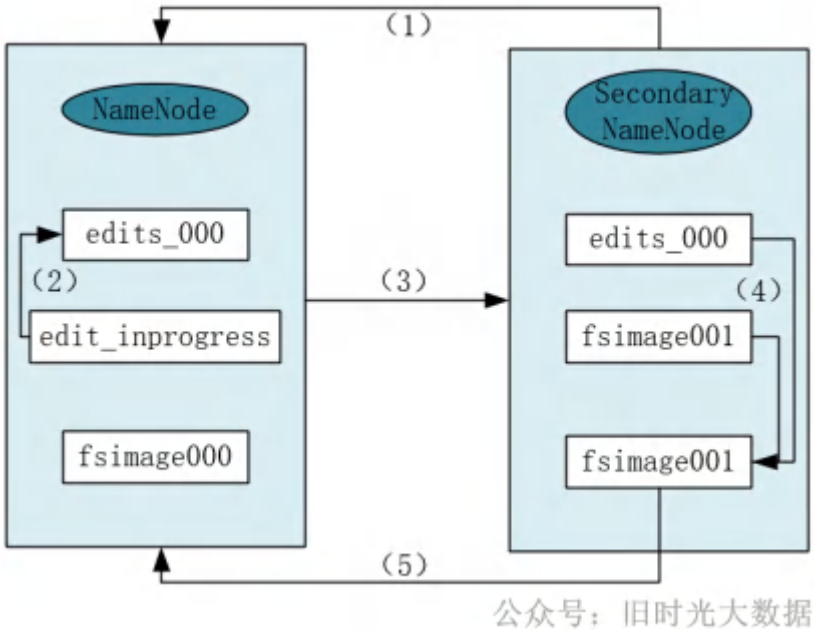
mapred-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>
    <!-- 指定 MapReduce 程序运行在 Yarn 上 -->
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

Hadoop的checkpoint流程

问过的一些公司：字节

参考答案：



在hdfs-site.xml文件指定Secondary NameNode部署在哪个机器上。

当记录条数超过100w条，或者时间超过3600s（1小时），Secondary NameNode提醒NameNode做Checkpoint。

步骤如下：

1）Secondary NameNode提醒NameNode开始做Checkpoint；

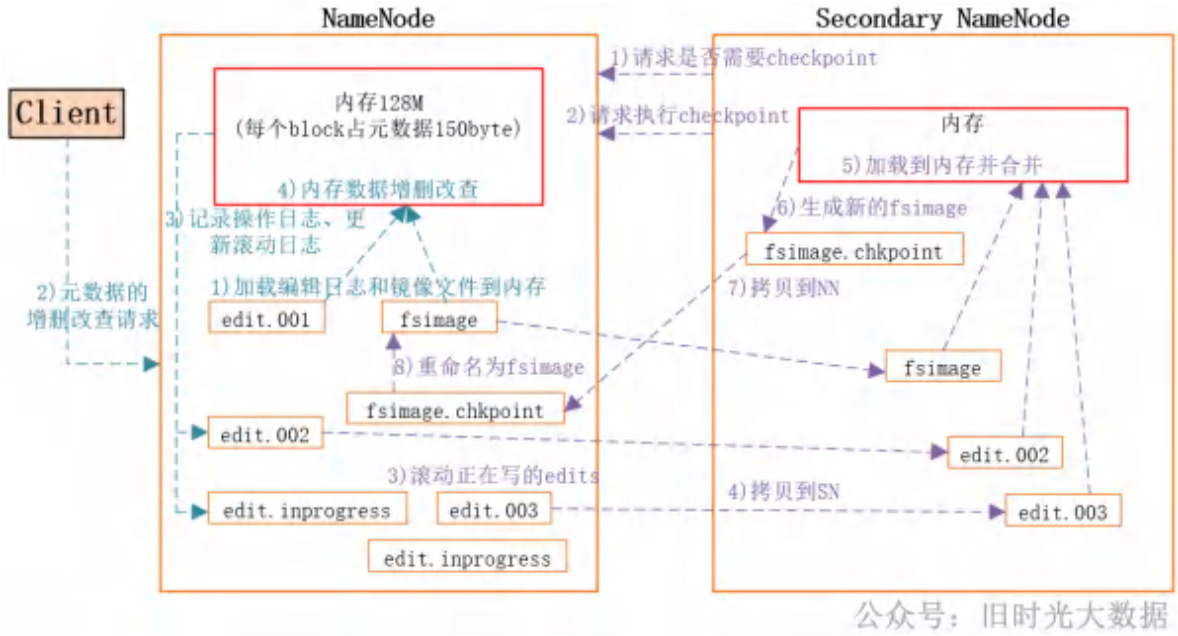
2）NameNode上的edit_inprogress文件滚动生成edit_sum文件；

3）edit_inprogress滚动生成edit文件，将滚动前的编辑日志edit和镜像文件fsimage拷贝到Secondary NameNode；

4）Secondary NameNode 加载编辑日志和镜像文件到内存并合并；

5）生成新的镜像文件fsimage.chkpoint，拷贝fsimage.chkpoint到NameNode，NameNode将fsimage.chkpoint重新命名成fsimage。

详细版本



FsImage：

元数据序列化后在磁盘存储的地方。包含HDFS文件系统的所有目录跟文件inode序列化信息。

公众号：旧时光大数据

Memory：

元数据在内存中存储的地方。

Edit文件：

1）Edit 记录客户端更新元数据信息的每一步操作（可通过Edits运算出元数据）。

2）一旦元数据有更新跟添加，元数据修改追加到Edits中然后修改内存中的元数据，这样一旦NameNode节点断电，通过 FsImage 跟 Edits 的合并生成元数据。

3）Edits文件不要过大，系统会定期的由 Secondary Namenode 完成 FsImage 和 Edits 的合并。

Hadoop的默认块大小是多少？为什么要设置这么大？

可回答：1）Hadoop的block大小调大了会造成什么影响？调大好还是调小好一点？为什么？2）对于MapReduce程序，block调大好还是调小好？为什么？3）为什么块的大小不能设置的太小，也不能设置的太大？4）HDFS的默认数据块大小是多少？（128M）为什么是128M？

回答技巧：从寻址和磁盘两个方面回答

问过的一些公司：阿里，字节，触宝(2021.07)

参考答案：

默认块大小：

Hadoop 2.7.2版本及之前默认64MB，Hadoop 2.7.3版本及之后默认128M

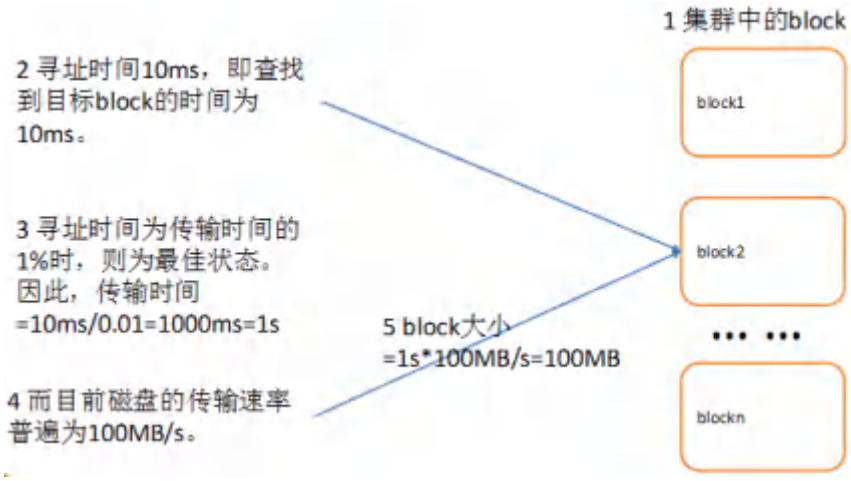
块大小：

HDFS的块设置太小，会增加寻址时间，程序一直在找块的开始位置，而且NameNode需要大量内存来存储元数据，不可取。

如果块设置的太大，从磁盘传输数据的时间会明显大于定位这个块开始位置所需的时间。导致程序在处理这块数据时，会非常慢。

如果寻址时间约为10ms，而传输速率为100MB/s，为了使寻址时间仅占传输时间的1%，我们要将块大小设置约为100MB。默认的块大小128MB。

块的大小：`10ms*100*100M/s = 100M`，如图



HDFS块的大小设置主要取决于磁盘传输速率。

公众号：旧时光大数据

Block划分的原因

问过的一些公司：Shopee

可回答：Hadoop为什么要分块存储

参考答案：

目的：减少磁盘寻道的时间。

不设置block：因为数据是分散的存放磁盘上的，读取数据时需要不停的进行磁盘寻道，开销比较大。

使用block：一次可以读取一个block中的数据，减少磁盘寻道的次数和时间。

Hadoop常见的压缩算法?

问过的一些公司：阿里

参考答案：

Hadoop中常用的压缩算法有bzip2、gzip、lzo、snappy，其中lzo、snappy需要操作系统安装native库才可以支持。

数据压缩的位置如下所示。

MapReduce数据压缩解析

1）输入端采用压缩

在有大量数据并计划重复处理的情况下，应该考虑对输入进行压缩。然而，你无须显示指定使用的编解码方式。Hadoop自动检查文件扩展名，如果扩展名能够匹配，就会用恰当的编解码方式对文件进行压缩和解压。否则，Hadoop就不会使用任何编解码器。

2）mapper输出端采用压缩

当map任务输出的中间数据量很大时，应考虑在此阶段采用压缩技术。这能显著改善内部数据Shuffle过程，而Shuffle过程在Hadoop处理过程中是资源消耗最多的环节。如果发现数据量大造成网络传输缓慢，应该考虑使用压缩技术。可用于压缩mapper输出的快速编解码器包括LZO或者Snappy。

3）reducer输出采用压缩

在此阶段启用压缩技术能够减少要存储的数据量，因此降低所需的磁盘空间。当mapreduce作业形成作业链条时，因为第二个作业的输入也已压缩，所以启用压缩同样有效。

压缩方式对比

1）Gzip压缩

优点：压缩率比较高，而且压缩/解压速度也比较快；hadoop本身支持，在应用中处理gzip格式的文件就和直接处理文本一样；大部分linux系统都自带gzip命令，使用方便。

缺点：不支持split。

应用场景：当每个文件压缩之后在130M以内的（1个块大小内），都可以考虑用gzip压缩格式。例如说一天或者一个小时的日志压缩成一个gzip文件，运行mapreduce程序的时候通过多个gzip文件达到并发。hive程序，streaming程序，和java写的mapreduce程序完全和文本处理一样，压缩之后原来的程序不需要做任何修改。

2）Bzip2压缩

优点：支持split；具有很高的压缩率，比gzip压缩率都高；hadoop本身支持，但不支持native；在linux系统下自带bzip2命令，使用方便。

缺点：压缩/解压速度慢；不支持native。

应用场景：适合对速度要求不高，但需要较高的压缩率的时候，可以作为mapreduce作业的输出格式；或者输出之后的数据比较大，处理之后的数据需要压缩存档减少磁盘空间并且以后数据用得比较少的情况；或者对单个很大的文本文件想压缩减少存储空间，同时又需要支持split，而且兼容之前的应用程序（即应用程序不需要修改）的情况 。

3）Lzo压缩

优点：压缩/解压速度也比较快，合理的压缩率；支持split，是hadoop中最流行的压缩格式；可以在linux系统下安装lzop命令，使用方便。

缺点：压缩率比gzip要低一些；hadoop本身不支持，需要安装；在应用中对lzo格式的文件需要做一些特殊处理（为了支持split需要建索引，还需要指定inputformat为lzo格式）。

应用场景：一个很大的文本文件，压缩之后还大于200M以上的可以考虑，而且单个文件越大，lzo优点越越明显。

4）Snappy压缩

优点：高速压缩速度和合理的压缩率。

缺点：不支持split；压缩率比gzip要低；hadoop本身不支持，需要安装；

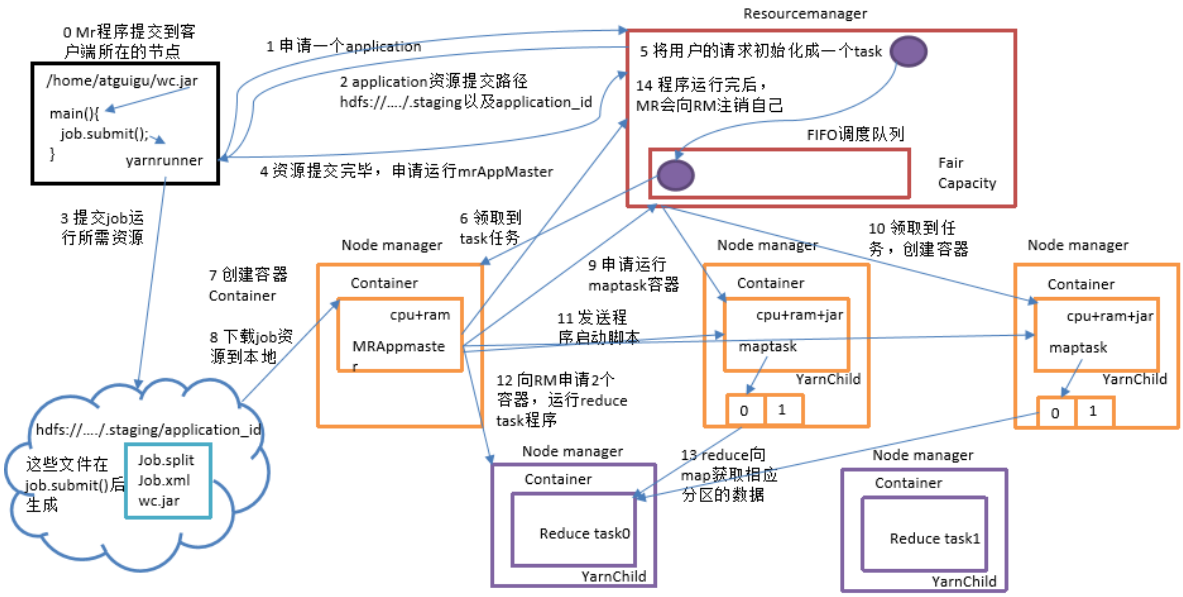
应用场景：当Mapreduce作业的Map输出的数据比较大的时候，作为Map到Reduce的中间数据的压缩格式；或者作为一个Mapreduce作业的输出和另外一个Mapreduce作业的输入。

Hadoop作业提交到YARN的流程?

问过的一些公司：字节

参考答案：

作业提交全过程详解



1）作业提交

第 1 步：Client 调用 job.waitForCompletion 方法，向整个集群提交 MapReduce 作业。

第 2 步：Client 向 RM 申请一个作业 id。

第 3 步：RM 给 Client 返回该 job 资源的提交路径和作业 id。

第 4 步：Client 提交 jar 包、切片信息和配置文件到指定的资源提交路径。

第 5 步：Client 提交完资源后，向 RM 申请运行 MrAppMaster。

2）作业初始化

第 6 步：当 RM 收到 Client 的请求后，将该 job 添加到容量调度器中。

第 7 步：某一个空闲的 NM 领取到该 Job。

第 8 步：该 NM 创建 Container，并产生 MRAppmaster。

第 9 步：下载Client 提交的资源到本地。

3）任务分配

第 10 步：MrAppMaster 向 RM 申请运行多个 MapTask 任务资源。

第 11 步：RM 将运行 MapTask 任务分配给另外两个NodeManager，另两个 NodeManager分别领取任务并创建容器。

4）任务运行

第 12 步：MR向两个接收到任务的NodeManager发送程序启动脚本，这两个NodeManager分别启动 MapTask，MapTask 对数据分区排序。

第13 步：MrAppMaster 等待所有MapTask 运行完毕后，向RM 申请容器，运行ReduceTask。

第 14 步：ReduceTask 向 MapTask 获取相应分区的数据。

第 15 步：程序运行完毕后，MR 会向 RM 申请注销自己。

5）进度和状态更新

YARN 中的任务将其进度和状态（包括counter）返回给应用管理器, 客户端每秒（通过 mapreduce.client.progressmonitor.pollinterval设置）向应用管理器请求进度更新, 展示给用户。

6）作业完成

除了向应用管理器请求作业进度外, 客户端每 5 秒都会通过调用 waitForCompletion()来检查作业是否完成。时间间隔可以通过 mapreduce.client.completion.pollinterval 来设置。作业完成之后, 应用管理器和 Container 会清理工作状态。作业的信息会被作业历史服务器存储以备之后用户核查。

Hadoop的Combiner的作用

问过的一些公司：冠群驰骋

参考答案：

作用

对每一个MapTask的输出进行局部汇总，以减小网络传输量

Combiner理解

Combiner是MapReduce程序中Mapper和Reducer之外的一种组件，它属于优化方案，由于带宽限制，应该尽量限制map和reduce之间的数据传输数量。它在Map端把同一个key的键值对合并在一起并计算，计算规则与reduce一致，所以Combiner也可以看作特殊的Reducer。

Combiner和Reducer的区别在于运行的位置：

- Combiner是在每一个MapTask所在的节点运行；
- Reducer是接收全局所有Mapper的输出结果。

在哪里使用Combiner

map输出数据根据分区排序完成后，在写入文件之前会执行一次combine操作（前提是作业中设置了这个操作）；
如果map输出比较大，溢出文件个数大于3（此值可以通过属性min.num.spills.for.combine配置）时，在merge的过程（多个spill文件合并为一个大文件）中前还会执行Combiner操作。

注意事项

不是每种作业都可以做combiner操作的，只有满足以下条件才可以：

- Combiner只应该用于那种Reduce的输入key/value与输出key/value类型完全一致，因为Combine本质上就是reduce操作。
- 计算逻辑上，Combine操作后不会影响计算结果，像求和，最大值就不会影响，求平均值就影响了。

Hadoop序列化和反序列化

问过的一些公司：字节

参考答案：

1、序列化和反序列化

序列化

把内存中的对象，转换成字节序列（或其他数据传输协议）以便于存储（持久化）和网络传输。

反序列化

将收到字节序列（或其他数据传输协议）或者是硬盘的持久化数据，转换成内存中的对象

2、为什么要序列化

公众号：旧时光大数据

一般来说，“活的”对象只生存在内存里，关机断电就没有了。而且“活的”对象只能由本地的进程使用，不能被发送到网络上的另外一台计算机。然而序列化可以存储“活的” 对象，可以将“活的”对象发送到远程计算机。

3、为什么不用 Java 的序列化

Java 的序列化是一个重量级序列化框架（Serializable），一个对象被序列化后，会附带很多额外的信息（各种校验信息，Header，继承体系等），不便于在网络中高效传输。所以， Hadoop 自己开发了一套序列化机制（Writable）。

4、Hadoop 序列化特点

- 紧凑：高效使用存储空间。
- 快速：读写数据的额外开销小。
- 互操作：支持多语言的交互

Hadoop的运行模式

问过的一些公司：央视网

参考答案：

1）单机模式

安装简单，在一台机器上运行服务，几乎不用做任何配置，但仅限于调试用途。没有分布式文件系统，直接读写本地操作系统的文件系统。

2）伪分布式模式

在单节点上同时启动namenode、datanode、jobtracker、tasktracker、secondary namenode等进程，模拟分布式运行的各个节点。配置已经很接近完全分布式。

3）完全分布式模式

正常的Hadoop集群，由多个各司其职的节点构成。

Hadoop小文件处理问题

可回到：Hadoop小文件优化

问过的一些公司：小米，端点数据(2021.07)，快手(2021.09)，百度(2021.10)，腾讯云(2021.10)

参考答案：

小文件指的是那些size比HDFS的block size小的多的文件。Hadoop适合处理少量的大文件，而不是大量的小文件。

1、小文件导致的问题

首先，在HDFS中，任何block，文件或者目录在内存中均以对象的形式存储，每个对象约占150byte，如果有1000 0000个小文件，每个文件占用一个block，则namenode大约需要2G空间。如果存储1亿个文件，则namenode需要20G空间。这样namenode内存容量严重制约了集群的扩展。

其次，访问大量小文件速度远远小于访问几个大文件。HDFS最初是为流式访问大文件开发的，如果访问大量小文件，需要不断的从一个datanode跳到另一个datanode，严重影响性能。

最后，处理大量小文件速度远远小于处理同等大小的大文件的速度。每一个小文件要占用一个slot，而task启动将耗费大量时间甚至大部分时间都耗费在启动task和释放task上。

公众号：旧时光大数据

2、Hadoop自带的解决方案

对于小文件问题，Hadoop本身也提供了几个解决方案，分别为：`Hadoop Archive`，`Sequence file` 和 `CombineFileInputFormat`。

`Hadoop Archive`

Hadoop Archive或者HAR，是一个高效地将小文件放入HDFS块中的文件存档工具，它能够将多个小文件打包成一个HAR文件，这样在减少namenode内存使用的同时，仍然允许对文件进行透明的访问。

对某个目录/foo/bar下的所有小文件存档成/outputdir/ zoo.har：

```
hadoop archive -archiveName zoo.har -p /foo/bar /outputdir
```

当然，也可以指定HAR的大小(使用-Dhar.block.size)。

HAR是在Hadoop file system之上的一个文件系统，因此所有fs shell命令对HAR文件均可用，只不过是文件路径格式不一样，HAR的访问路径可以是以下两种格式：

```
har://scheme-hostname:port/archivepath/fileinarchive
har:///archivepath/fileinarchive(本节点)
```

可以这样查看HAR文件存档中的文件：

```
hadoop dfs -ls har:///user/zoo/foo.har
```

输出：

```
har:///user/zoo/foo.har/hadoop/dir1
har:///user/zoo/foo.har/hadoop/dir2
```

使用HAR时需要两点

第一，对小文件进行存档后，原文件并不会自动被删除，需要用户自己删除；

第二，创建HAR文件的过程实际上是在运行一个mapreduce作业，因而需要有一个hadoop集群运行此命令。

此外，HAR还有一些缺陷：

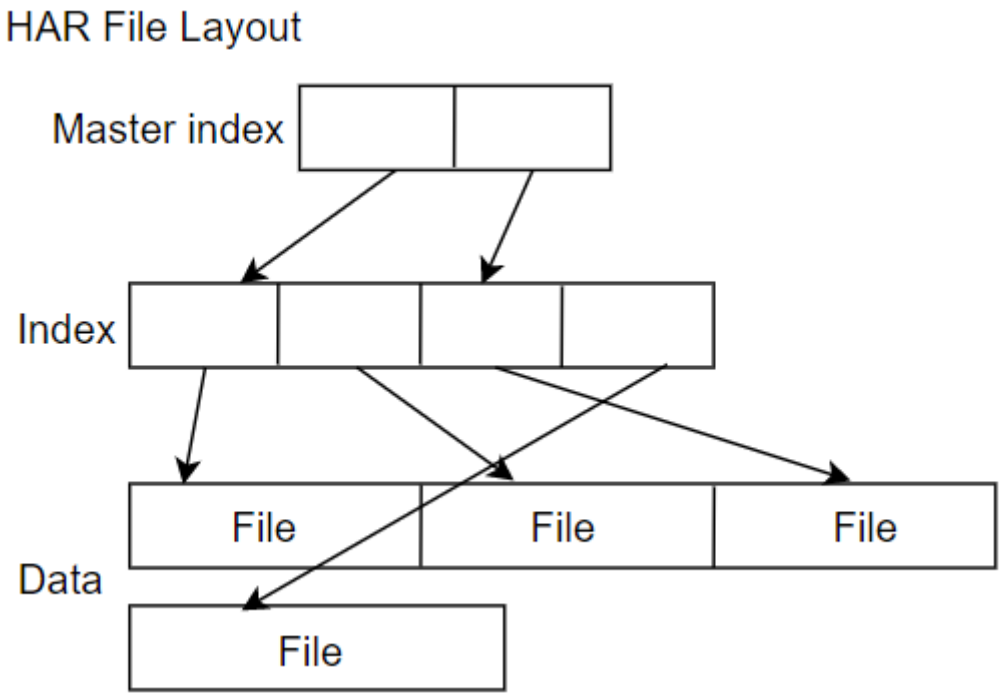
第一，一旦创建，Archives便不可改变。要增加或移除里面的文件，必须重新创建归档文件。

第二，要归档的文件名中不能有空格，否则会抛出异常，可以将空格用其他符号替换(使用-Dhar.space.replacement.enable=true 和-Dhar.space.replacement参数)。

第三，存档文件不支持压缩。

一个归档后的文件，其存储结构如下图：

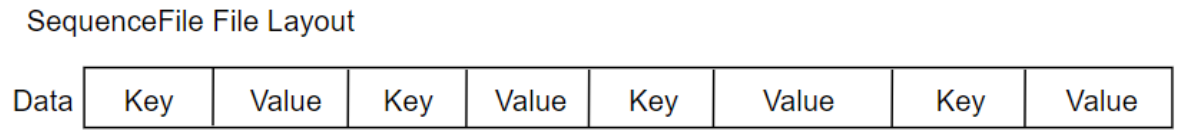
公众号：旧时光大数据

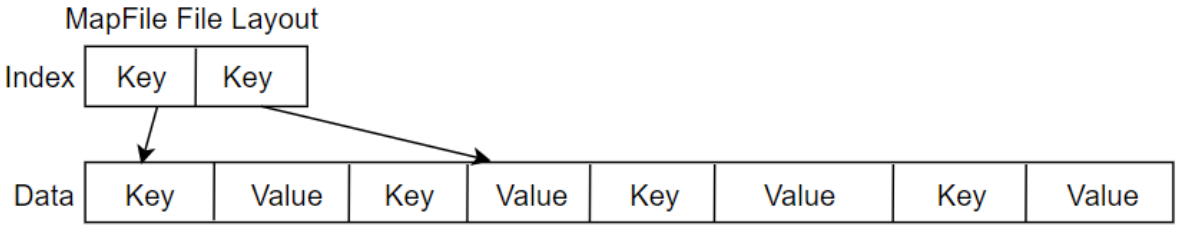


`Sequence file`

Sequence file由一系列的二进制key/value组成，如果为key小文件名，value为文件内容，则可以将大批小文件合并成一个大文件。

Hadoop-0.21.0中提供了SequenceFile，包括Writer，Reader和SequenceFileSorter类进行写，读和排序操作。





创建sequence file的过程可以使用mapreduce工作方式完成，对于index，需要改进查找算法

优缺点：对小文件的存取都比较自由，也不限制用户和文件的多少，但是该方法不能使用append方法，所以适合一次性写入大量小文件的场景

`CombineFileInputFormat`

CombineFileInputFormat是一种新的inputformat，用于将多个文件合并成一个单独的split，另外，它会考虑数据的存储位置。

Hadoop为什么要从2.x升级到3.x?

可回答：Hadoop 3.x 版本相对于 Hadoop 2.x的新特性

问过的一些公司：字节(2021.07)

参考答案：

公众号：旧时光大数据

Apache Hadoop 3.x在以前的主要发行版本（hadoop-2.x）上进行了许多重大改进。

1、最低要求的Java版本从Java 7增加到Java 8

现在，已针对Java 8的运行时版本编译了所有Hadoop JAR。仍在使用Java 7或更低版本的用户必须升级到Java 8。

2、支持HDFS中的纠删码

纠删码是一种持久存储数据的方法，可节省大量空间。与标准HDFS副本机制的3倍开销相比，像Reed-Solomon(10,4) 这样的标准编码的空间开销是1.4倍。

由于纠删码在重建期间会带来额外的开销，并且大多数情况下会执行远程读取，因此传统上已将其用于存储较冷，访问频率较低的数据。

在部署此功能时应考虑纠删码机制的网络和CPU开销。

关于HDFS中纠删码更详细的介绍，可查看我之前写的这篇文章： 深入剖析 HDFS 3.x 新特性-纠删码

3、Shell脚本重写

Hadoop Shell脚本已被重写，以修复许多长期存在的错误并包括一些新功能。Hadoop的开发人员尽管一直在寻求兼容性，但是某些更改可能会破坏现有的安装。

4、MapReduce 任务本地优化

MapReduce 增加了对 map 输出收集器的本地执行的支持，对于 shuffle 密集型工作，这可以使性能提高30%或更多。

5、支持两个以上的 NameNode

在之前的版本中，HDFS的高可用最多支持两个NameNode。在HDFS 3.x 版本中，通过将编辑复制到法定数量的三个JournalNode，该体系结构能够容忍系统中任何一个节点的故障。

但是，某些部署需要更高的容错度。这个新特性启用了这一点，该功能允许用户运行多个备用NameNode。例如，通过配置三个NameNode和五个JournalNode，群集可以忍受两个节点的故障，而不仅仅是一个节点的故障。

6、多个服务的默认端口已更改

以前，多个Hadoop服务的默认端口在Linux临时端口范围内（32768-61000）。这意味着在启动时，服务有时会由于与另一个应用程序的冲突而无法绑定到端口。

这些冲突的端口已移出临时范围，具体的端口更改如下：

- NameNode 的端口：50070 --> 9870, 8020 --> 9820, 50470 --> 9871；
- Secondary NameNode 的端口：50091 --> 9869, 50090 --> 9868；
- DataNode 的端口：50020 --> 9867, 50010 --> 9866, 50475 --> 9865, 50075 --> 9864；
- Hadoop KMS 的端口：16000 --> 9600（HBase的HMaster端口号与Hadoop KMS端口号冲突。两者都使用16000，因此 Hadoop KMS 更改为9600）。

7、支持Microsoft Azure数据湖和阿里云对象存储系统文件系统连接器

Hadoop现在支持与Microsoft Azure数据湖和Aliyun对象存储系统集成，作为与Hadoop兼容的替代文件系统。

8、数据内节点平衡器

单个DataNode可管理多个磁盘。在正常的写操作过程中，磁盘将被均匀填充。但是，添加或替换磁盘可能会导致DataNode内部出现严重偏差。原有的HDFS平衡器无法处理这种情况。新版本的HDFS中有平衡功能处理，该功能通过hdfs diskbalancer CLI调用。

9、基于HDFS路由器的联合

公众号：旧时光大数据

基于HDFS路由器的联合添加了一个RPC路由层，该层提供了多个HDFS名称空间的联合视图。这简化了现有HDFS客户端对联合群集的访问。

10、YARN资源类型

YARN资源模型已被通用化，以支持用户定义的CPU和内存以外的可计数资源类型。例如，集群管理员可以定义资源，例如GPU，软件许可证或本地连接的存储。然后可以根据这些资源的可用性来调度YARN任务。

Hadoop的优缺点

问过的一些公司：小红书(2021.11)

参考答案：

优点：

1）高可靠性

Hadoop底层维护多个数据副本，即使Hadoop某个计算元素或存储出现故障时，也不会大致数据的丢失

2）高扩展性

在集群间分配任务数据，可方便的扩展数以千计的节点

3）高效性

在MapReduce的思想下，Hadoop是并行工作，加快任务处理速度

4）高容错性

能够自动将失败的任务重新分配

缺点：

1）Hadoop不适用于低延迟数据访问

2）Hadoop不能高效存储大量小文件

3）Hadoop不支持多用户写入并任意修改文件

HDFS部分

HDFS文件写入和读取流程

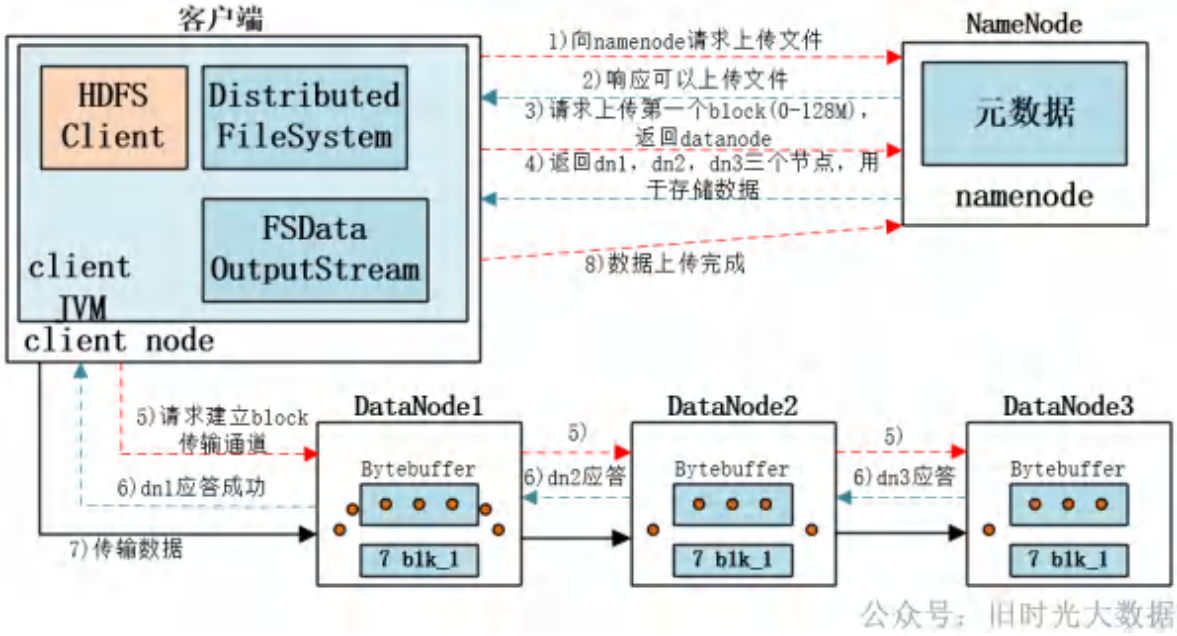
也可回答：1）读写原理（流程）；2）上传下载流程；3）Hadoop中文件put和get的详细过程；4）讲讲（介绍下）HDFS；5）介绍一下HDFS存数据原理

问过的一些公司：阿里×3，阿里社招，腾讯x2，字节x2，百度，拼多多x2，浩鲸云，小米，流利说，顺丰，网易云音乐×2，有赞×2，祖龙娱乐，360×2，商汤科技，招银网络，深信服，多益，大华，快手，电信云计算，转转，美团x5，shopee×2：回答越详细越好，猿辅导×2，科大讯飞，恒生电子，搜狐，京东，头条，富途，大华(2021.07)，远景智能(2021.08)，Shopee(2021.08)，携程(2021.09)，字节(2021.08)，四方伟业(2021.08)，海康(2021.09)，米哈游(2021.09)，欢聚(2021.09)，虎牙(2021.09)

参考答案：

HDFS存储机制，包括HDFS的写入数据过程和读取数据过程两部分

HDFS写数据过程



1）客户端通过Distributed FileSystem模块向NameNode请求上传文件，NameNode检查目标文件是否已存在，父目录是否存在。

2）NameNode返回是否可以上传。

3）客户端请求第一个 block上传到哪几个datanode服务器上。

4）NameNode返回3个datanode节点，分别为dn1、dn2、dn3。

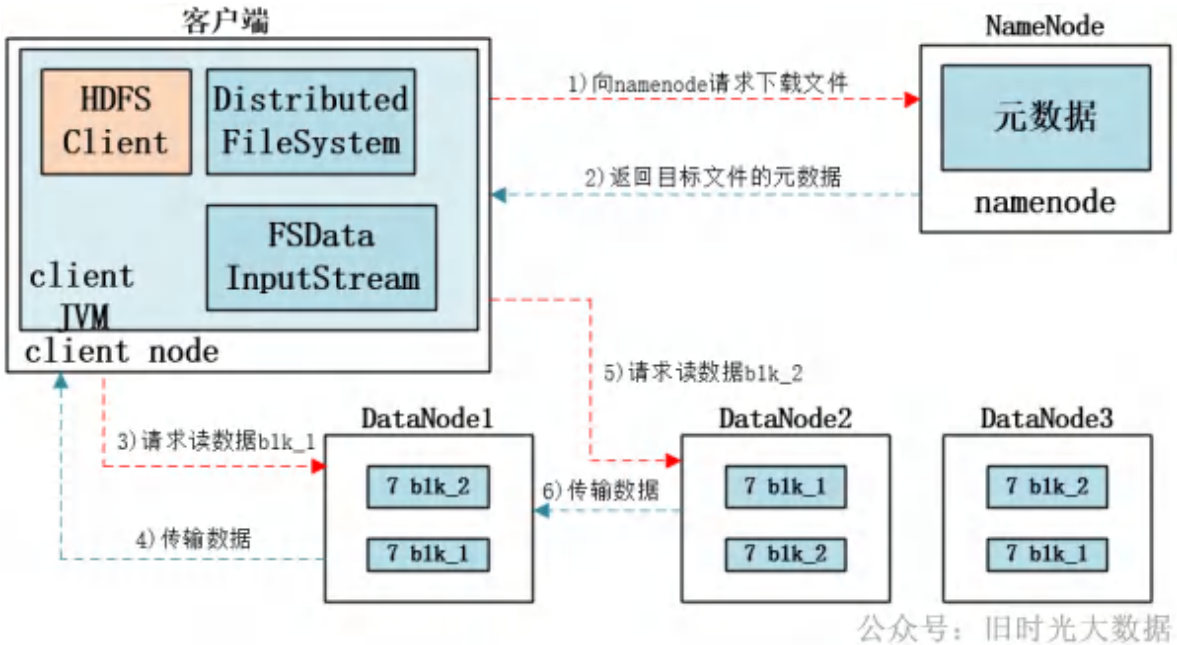
5）客户端通过FSDataOutputStream模块请求dn1上传数据，dn1收到请求会继续调用dn2，然后dn2调用dn3，将这个通信管道建立完成。

6）dn1、dn2、dn3逐级应答客户端。

7）客户端开始往dn1上传第一个block（先从磁盘读取数据放到一个本地内存缓存），以packet为单位，dn1收到一个packet就会传给dn2，dn2传给dn3；dn1每传一个packet会放入一个应答队列等待应答。

8）当一个block传输完成之后，客户端再次请求NameNode上传第二个block的服务器。（重复执行3-7步）。

HDFS读数据过程

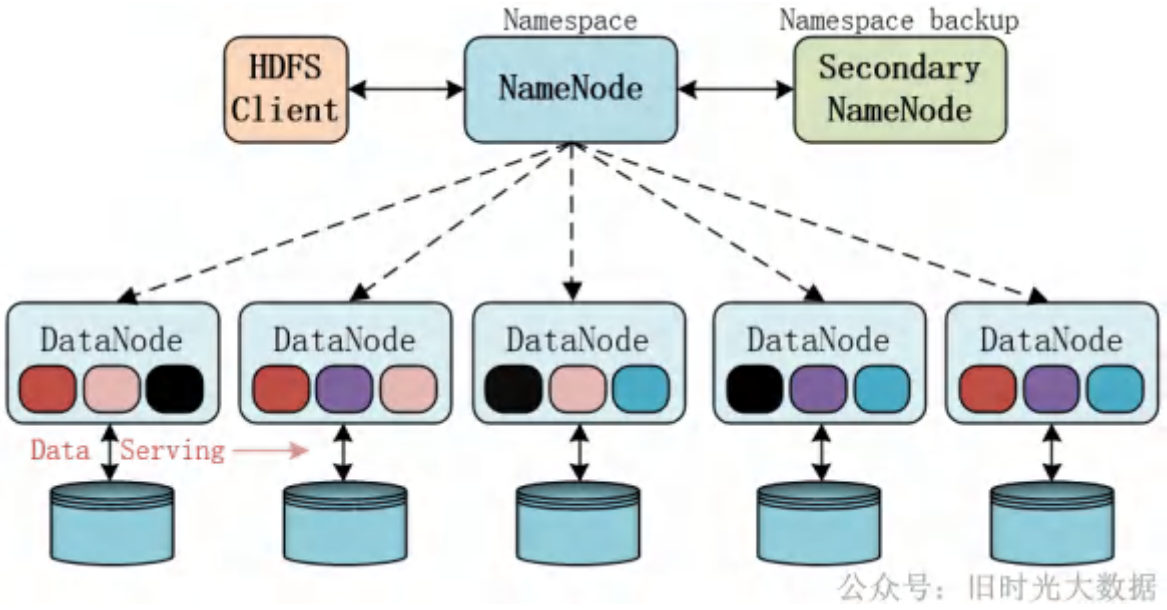


HDFS组成架构

可回答：1）HDFS组件；2）HDFS的架构，它们分别具备什么功能？3）HDFS三个进程作用；4）SeconderyNameNode的作用

问过的一些公司：京东，美团，电信云计算，猿辅导，网易，作业帮，触宝(2021.07)，京东(2021.07)，远景智能(2021.08)，字节(2021.08)，恒生(2021.09)

参考答案：



架构主要由四个部分组成，分别为`HDFS Client`、`NameNode`、`DataNode`和`Secondary NameNode`。下面我们分别介绍这四个组成部分。

1）`Client`：就是客户端

（1）文件切分。文件上传HDFS的时候，Client将文件切分成一个一个的Block，然后进行存储；

（2）与NameNode交互，获取文件的位置信息；

（3）与DataNode交互，读取或者写入数据；

（4）Client提供一些命令来管理HDFS，比如启动或者关闭HDFS；

（5）Client可以通过一些命令来访问HDFS；

2）`NameNode`：就是Master，它是一个主管、管理者

（1）管理HDFS的名称空间；

（2）管理数据块（Block）映射信息；

（3）配置副本策略；

（4）处理客户端读写请求。

3）`DataNode`：就是Slave。 NameNode下达命令，DataNode执行实际的操作

（1）存储实际的数据块；

（2）执行数据块的读/写操作。

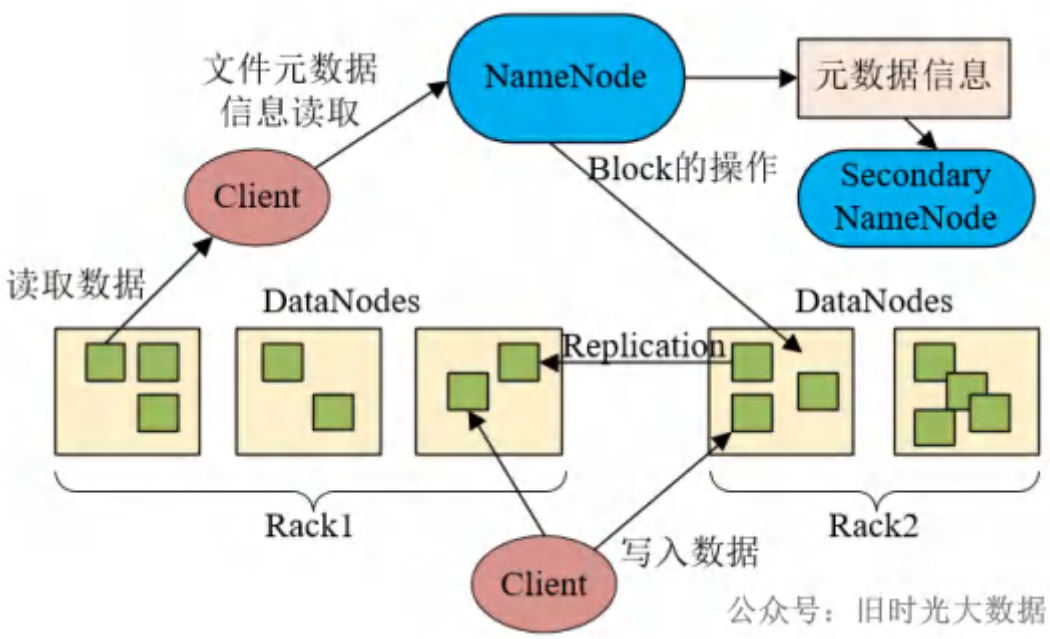
4）`Secondary NameNode`：并非NameNode的热备。 当NameNode挂掉的时候，它并不能马上替换NameNode并提供服务

（1）辅助NameNode，分担其工作量；

（2）定期合并Fsimage和Edits，并推送给NameNode；

（3）在紧急情况下，可辅助恢复NameNode。

HDFS架构另一种图



介绍下HDFS，说下HDFS优缺点，以及使用场景

可回答：1）对HDFS的了解；2）详细说一下HDFS（可结合上一题和下一题说）

问过的一些公司：小米，七牛云×2，美团x2，百度(2021.08)，海康(2021.09)，网易有道(2021.10)

参考答案：

HDFS是一个文件系统，用于存储文件，通过目录树来定位文件。其次，它是分布式的，由很多服务器联合起来实现其功能，集群中的服务器有各自的角色。

优点

1）高容错性

数据自动保存多个副本。它通过增加副本的形式，提高容错性。

某一个副本丢失以后，它可以自动恢复。

2）适合处理大数据

- 数据规模：能够处理数据规模达到GB、TB、甚至PB级别的数据；
- 文件规模：能够处理百万规模以上的文件数量，数量相当之大。

3）可构建在廉价机器上，通过多副本机制，提高可靠性。

缺点

1）不适合低延时数据访问，比如毫秒级的存储数据，是做不到的。

2）无法高效的对大量小文件进行存储

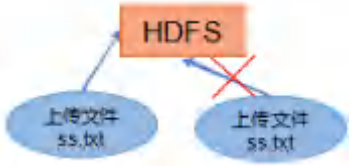
存储大量小文件的话，它会占用NameNode大量的内存来存储文件目录和块信息。这样是不可取的，因为NameNode的内存总是有限的；

小文件存储的寻址时间会超过读取时间，它违反了HDFS的设计目标。

3）不支持并发写入、文件随机修改

一个文件只能有一个写，不允许多个线程同时写；

仅支持数据append（追加），不支持文件的随机修改。



HDFS作用

问过的一些公司：美团

参考答案：

HDFS在Hadoop中的作用是为海量的数据提供了存储，能提供高吞吐量的数据访问，HDFS有高容错性的特点，并且设计用来部署在低廉的硬件上；而且它提供高吞吐量来访问应用程序的数据，适合那些有着超大数据集的应用程序。

对外部客户机而言，HDFS就像一个传统的分级文件系统。可以创建、删除、移动或重命名文件，等等。但是HDFS的架构是基于一组特定的节点构建的，这是由它自身的特点决定的。这些节点包括NameNode（仅一个，HA两个），它在HDFS内部提供元数据服务；DataNode，它为HDFS提供存储块。

HDFS的容错机制

问过的一些公司：腾讯，一点资讯

参考答案：

HDFS可以通过廉价机器搭建大规模集群，获得海量数据的分布式存储能力。对于廉价机器而言，出现网络故障、节点失效、数据损坏现象的频率并不低，所以在故障之后如何进行数据恢复和容错处理是至关重要的，HDFS提供了完善的容错机制，使得它成为一个高度容错性和高吞吐量的海量数据存储解决方案。

故障检测机制

故障的类型主要有以下三种，针对这三种故障类型，HDFS提供了不同的故障检测机制：

针对 `DataNode失效` 问题

每个DataNode以固定的周期向NameNode 发送心跳信号，通过这种方法告诉 NameNode 它们在正常工作。如果在一定的时间内 NameNode 没有收到 DataNode 心跳，就认为该 DataNode 宕机了。

针对 `网络故障` 而导致无法收发数据的问题

HDFS提供了ACK的机制，在发送端发送数据后，如果没有收到ACK并且经过多次重试后仍然如此，则认为网络故障；

针对`数据损坏(脏数据)`问题

- 在传输数据的时候，同时会发送总和检验码，当数据存储到硬盘时，总和检验码也会被存储。
- 所有的 DataNode 都会定期向 NameNode 发送数据块的存储状况。
- 在发送数据块报告前，会先检查总和校验码是否正确，如果数据存在错误就不发送该数据块的信息。

HDFS的存储机制

可回答：1）HDFS数据存储；2）HDFS存放

问过的一些公司：字节x2，美团，重庆富民银行(2021.09)

回答技巧：可以从介绍HDFS、HDFS的一些概念、读写过程、优缺点进行回答

参考答案：

HDFS（Hadoop Distributed File System）是Hadoop分布式计算中的数据存储系统，是基于流数据模式访问和处理超大文件的需求而开发的。

HDFS的基础概念

`Block`

HDFS中的存储单元是每个数据块block，HDFS默认的最基本的存储单位是64M的数据块。和普通的文件系统相同的是，HDFS中的文件也是被分成64M一块的数据块存储的。不同的是，在HDFS中，如果一个文件大小小于一个数据块的大小，它是不需要占用整个数据块的存储空间的。

`NameNode`

元数据节点。该节点用来管理文件系统中的命名空间，是master。其将所有的为了见和文件夹的元数据保存在一个文件系统树中，这些信息在硬盘上保存为了：命名空间镜像（namespace image）以及修改日志（edit log），后面还会讲到。此外，NameNode还保存了一个文件包括哪些数据块，分布在哪些数据节点上。然而，这些信息不存放在硬盘上，而是在系统启动的时候从数据节点收集而成的。

`DataNode`

数据节点。是HDFS真正存储数据的地方。客户端（client）和元数据节点（NameNode）可以向数据节点请求写入或者读出数据块。此外，DataNode需要周期性的向元数据节点回报其存储的数据块信息。

`Secondary NameNode`

从元数据节点。从元数据节点并不是NameNode出现问题时候的备用节点，它的主要功能是周期性的将NameNode中的namespace image和edit log合并，以防log文件过大。此外，合并过后的namespace image文件也会在Secondary NameNode上保存一份，以防NameNode失败的时候，可以恢复。

`Edit Log`

修改日志。当文件系统客户端client进行------写------操作的时候，我们就要把这条记录放在修改日志中。在记录了修改日志后，NameNode则修改内存中的数据结构。每次写操作成功之前，edit log都会同步到文件系统中。

`Fsimage`

命名空间镜像。它是内存中的元数据在硬盘上的checkpoint。当NameNode失败的时候，最新的checkpoint的元数据信息就会从fsimage加载到内存中，然后注意重新执行修改日志中的操作。而Secondary NameNode就是用来帮助元数据节点将内存中的元数据信息checkpoint到硬盘上的。

Checkpoint的过程如下

1）Secondary NameNode通知NameNode生成新的日志文件，以后的日志都写到新的日志文件中。

2）Secondary NameNode用http get从NameNode获得fsimage文件及旧的日志文件。

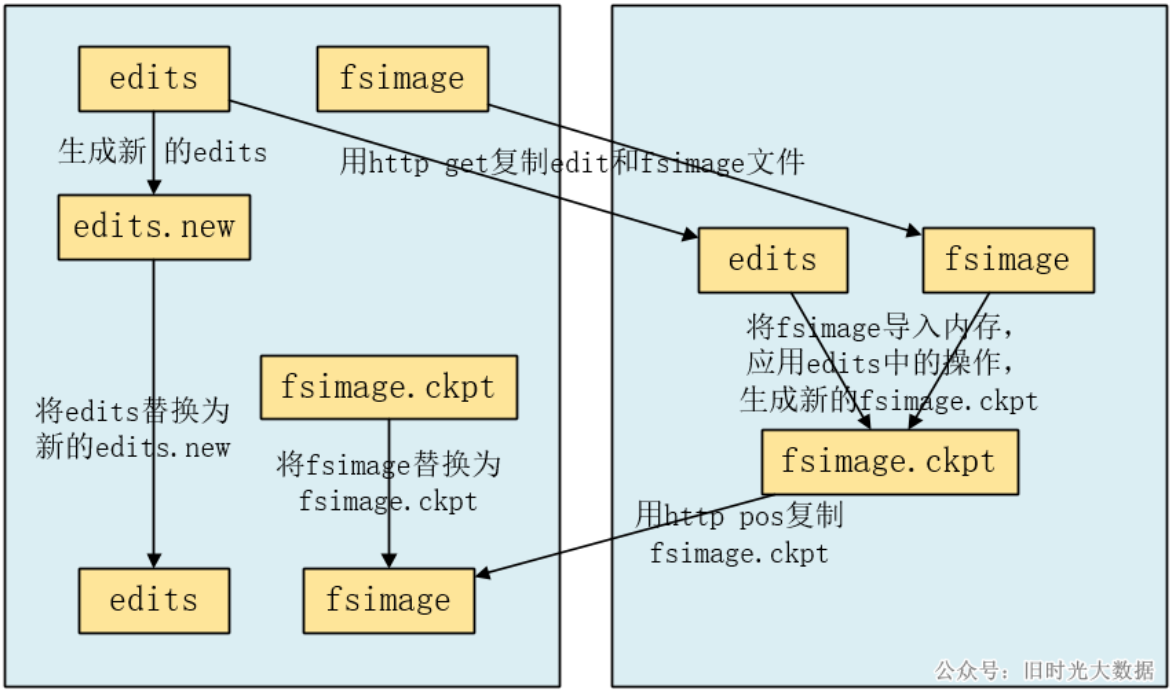
3）Secondary NameNode将fsimage文件加载到内存中，并执行日志文件中的操作，然后生成新的fsimage文件。

4）Secondary NameNode将新的fsimage文件用http post传回NameNode。

5）NameNode可以将旧的fsimage文件及旧的日志文件，换为新的fsimage文件和新的日志文件(第一步生成的)，然后更新fstime文件，写入此次checkpoint的时间。

这样NameNode中的fsimage文件保存了最新的checkpoint的元数据信息，日志文件也重新开始，不会变的很大了。

Checkpoint的过程如下图



HDFS中文件读写过程和优缺点见前面的题目

HDFS的副本机制

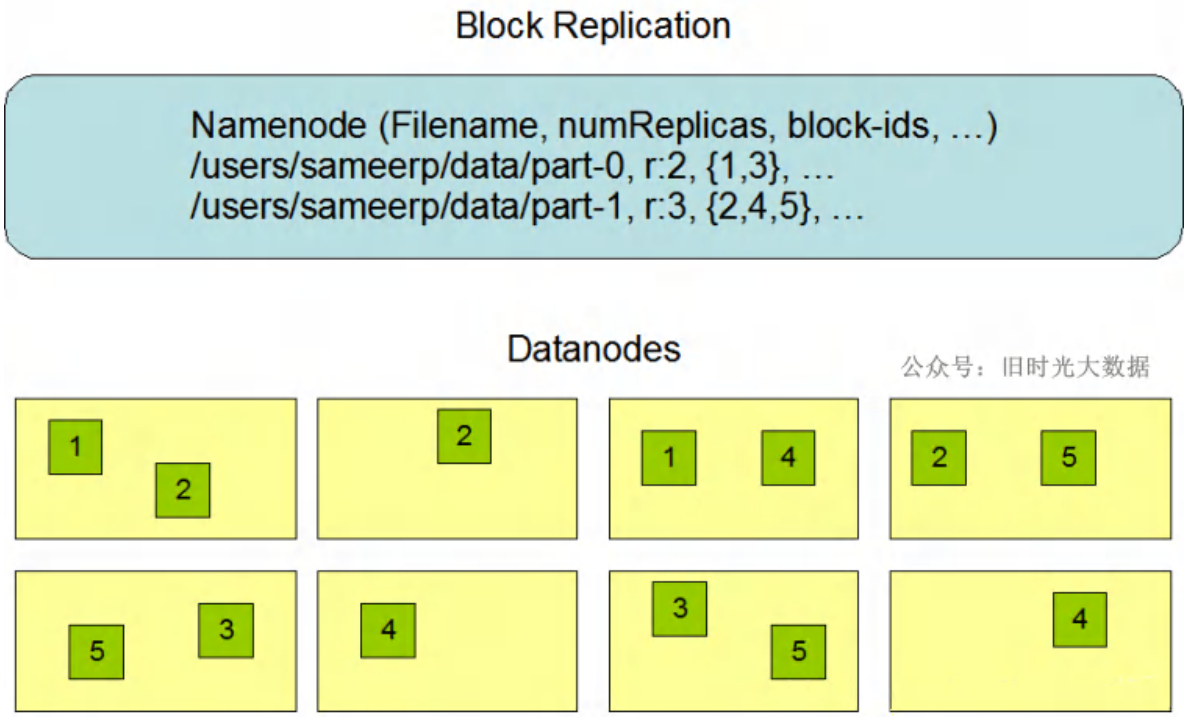
可回答：1）HDFS的副本放置策略；2）写数据时如何选择DataNode节点呢？这么选择节点的依据是什么？

问过的一些公司：多益，货拉拉(2021.07)，大华(2021.07)，远景智能(2021.08)，虎牙(2021.09)

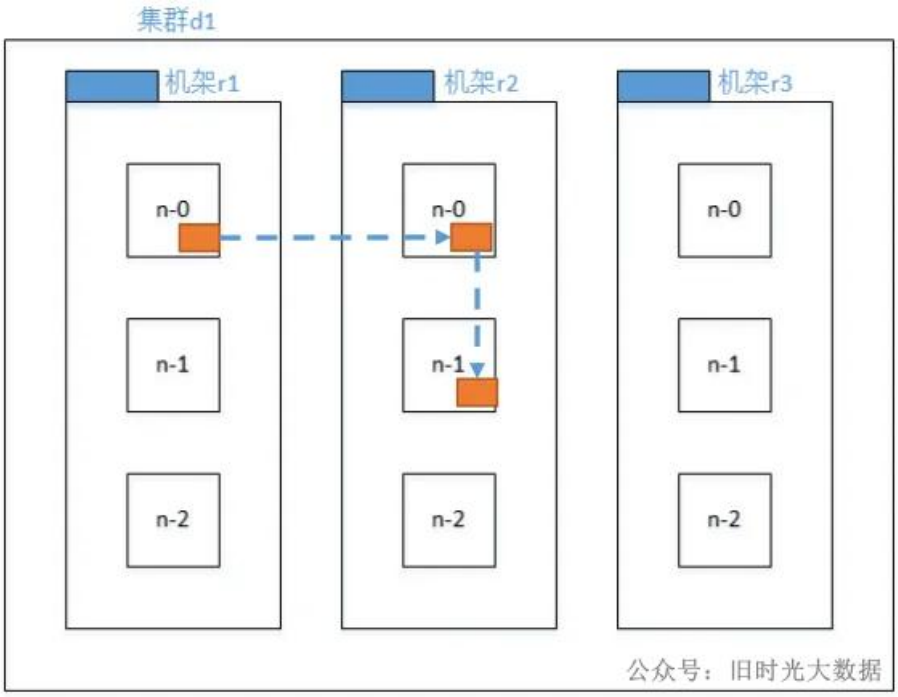
参考答案：

在HDFS中，一个文件会被拆分为一个或多个数据块。默认情况下，每个数据块都会有**3个副本**。每个副本都会被存放在不同的机器上，而且每一个副本都有自己唯一的编号。

公众号：旧时光大数据



NameNode节点选择一个DataNode节点去存储block副本的过程就叫做**副本存放**，这个过程的策略其实就是在可靠性和读写带宽间的权衡。



Hadoop3.x副本结点选择：

由上图可知，第一个副本在Client所处的节点上。如果客户端在集群外，随机选一个。

第二个副本在另一个机架的随机一个节点。

第三个副本在第二个副本所在机架的随机节点。

HDFS的常见数据格式，列式存储格式和行存储格式异同点，列式存储优点有哪些?

问过的一些公司：快手

参考答案：

1、Hadoop中的文件格式大致上分为面向行和面向列两类

行式存储

一条数据保存为一行，读取一行中的任何值都需要把整行数据都读取出来（如：Sequence Files，Map File，Avro Data Files），这种方式在磁盘读取的开销比较大，这无法避免。

列式存储

整个文件被切割为若干列数据，每一列中数据保存在一起（如：Parquet，RC Files，ORC Files，Carbon Data，IndexR）。这种方式会占用更多的内存空间，需要将行数据缓存起来。

2、列存储和行存储异同点

从以下几个方面说明

写入：

- 行存储的写入是一次完成，数据的完整性因此可以确定；
- 列存储需要把一行记录拆分成单列保存，写入次数明显比行存储多；
- 行存储在写入上占有很大的优势。

数据修改：

- 行存储是在指定位置写入一次，列存储是将磁盘定位到多个列上分别写入；
- 行存储在数据修改也是占优的。

数据读取：

- 行存储通常将一行数据完全读出，如果只需要其中几列数据，就会存在冗余列；
- 列存储每次读取的数据是集合中的一段或者全部；
- 由于列储存的数据是同质的，这种情况使数据解析变得容易。行存储则复杂的多，因为在一行记录中保存了多种类型的数据，数据解析需要在多种数据类型之间频繁转换，这个操作很消耗cpu；
- 所以列存储的解析过程中更有利于分析大数据。

3、列存储和行存储优缺点

行存储

优点：行存储的写入是一次性完成的，写入效率高，消耗的时间比列存储少，并且能够保证数据的完整性

缺点：数据读取过程中会产生冗余数据，如果只看少量数据，此影响可以忽略；数量大可能会影响数据的处理效率

列存储

优点：在读取过程中，不会产生冗余数据，这对数据完整性要求不高的大数据处理领域尤为重要

缺点：写入效率，保证数据完整性上都不如行存储

HDFS如何保证数据不丢失?

问过的一些公司：百度

参考答案：

先得出结果

1）数据在写入之后进行校验和的计算，DataNode周期性的进行校验和计算，将计算结果与第一次进行对比，若相同表示无数据丢失，若不相同表示有数据丢失，丢失后将进行数据修复；

2）数据读取之前对数据进行校验，与第一次的结果进行对比，若相同表示数据没有丢失，可以读取，若不相同表示数据有所丢失，到其他副本读取数据。

保证DataNode节点保证数据完整性的方法

1）当 DataNode 读取 Block 的时候，它会计算CheckSum；

2）如果计算后的 CheckSum，与Block 创建时值不一样，说明 Block 已经损坏；

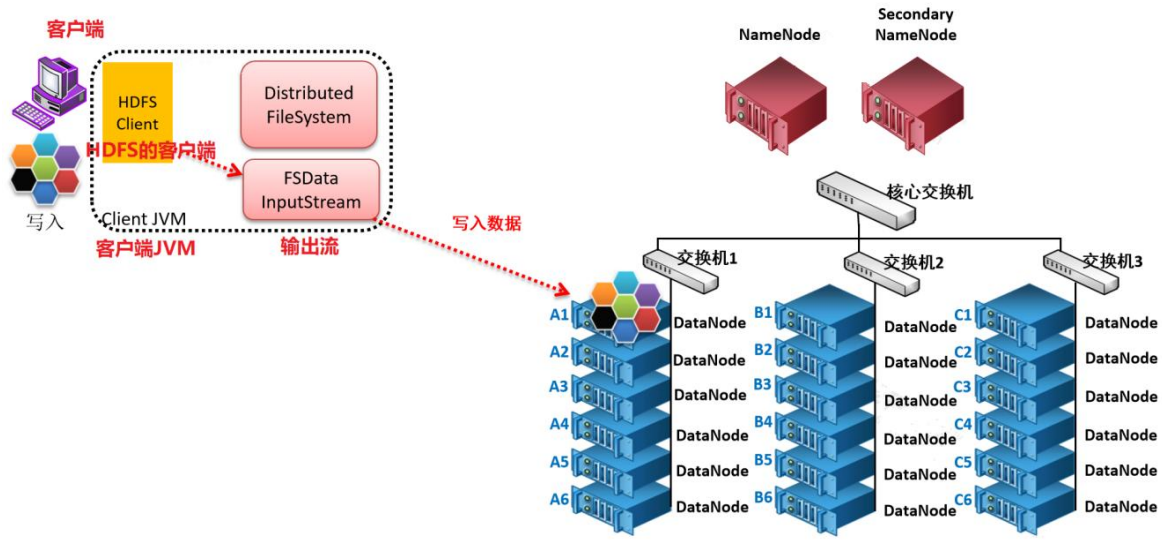
3）Client 读取其他DataNode 上的 Block；

4）常见的校验算法 crc（32），md5（128），sha1（160）；

5）DataNode 在其文件创建后周期验证CheckSum。

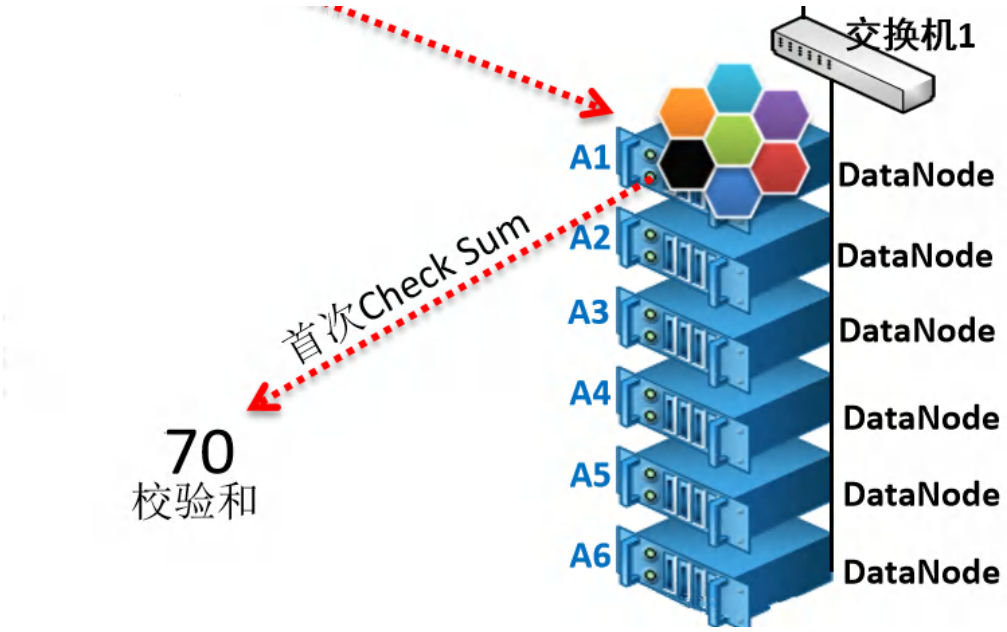
案例分析

第一步：写入数据

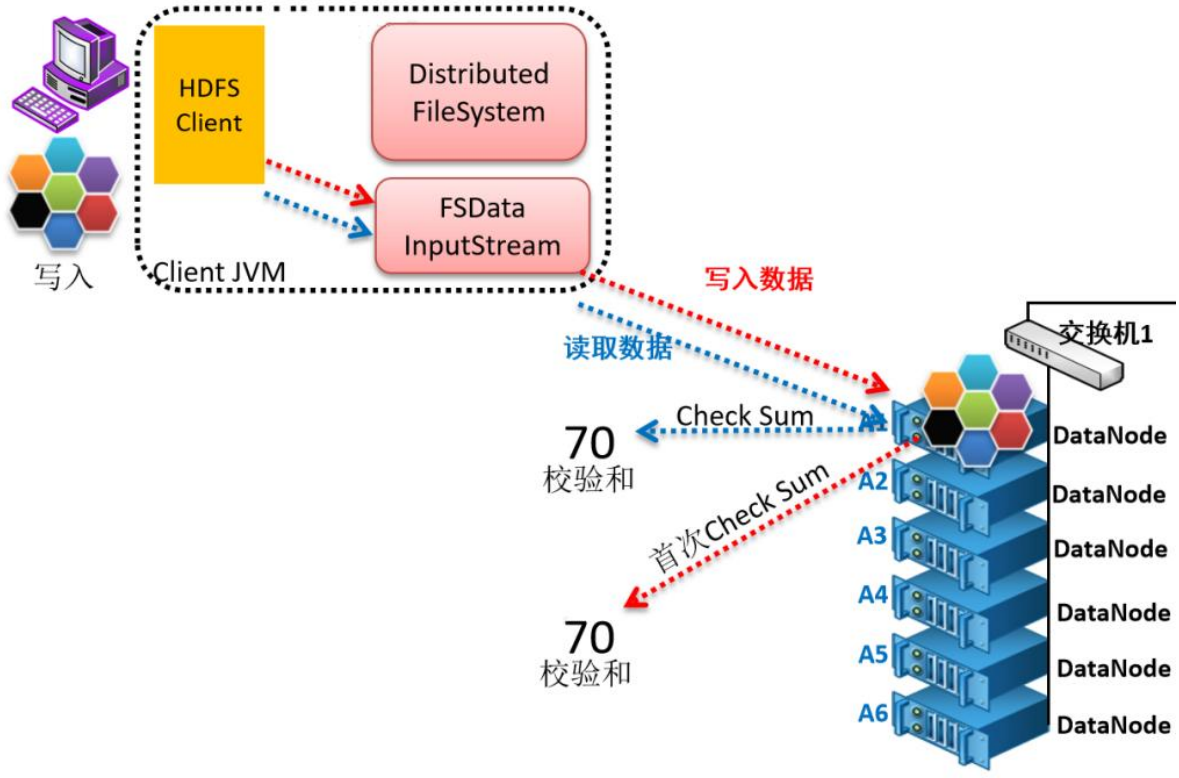


第二步：进行首次校验和，得到数值70

公众号：旧时光大数据



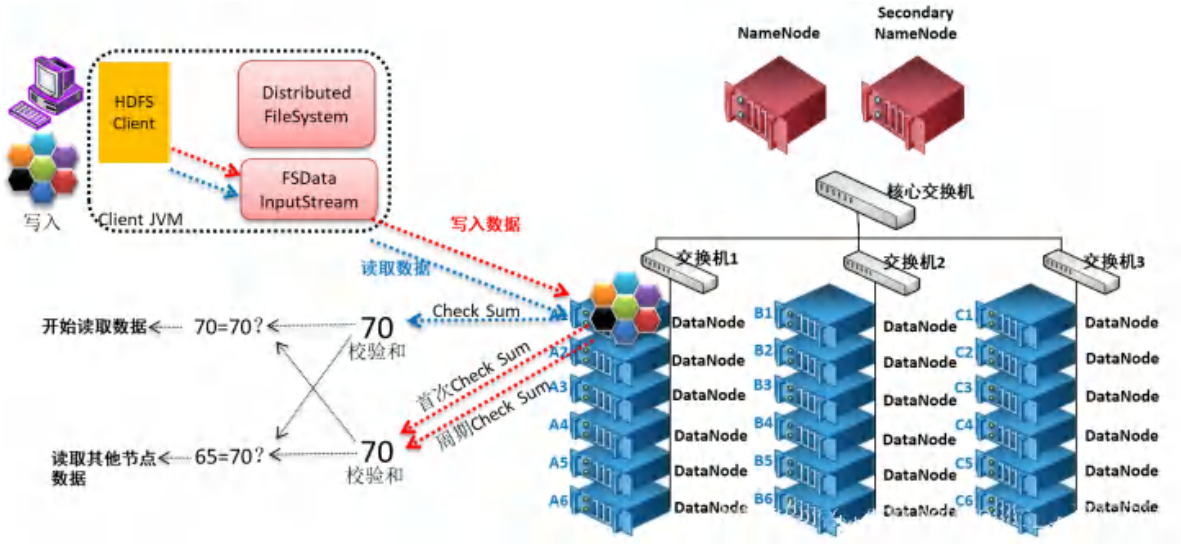
第三步：读取数据时，仍然会校验和，得到数据70



第四步：对比两次较校验和的值

如两次读取的值都是70,则开始读取数据

如两次读取的值不一致，说明此节点数据丢失，则去其他节点读取数据



HDFS NameNode高可用如何实现？需要哪些角色？

问过的一些公司：快手

参考答案：

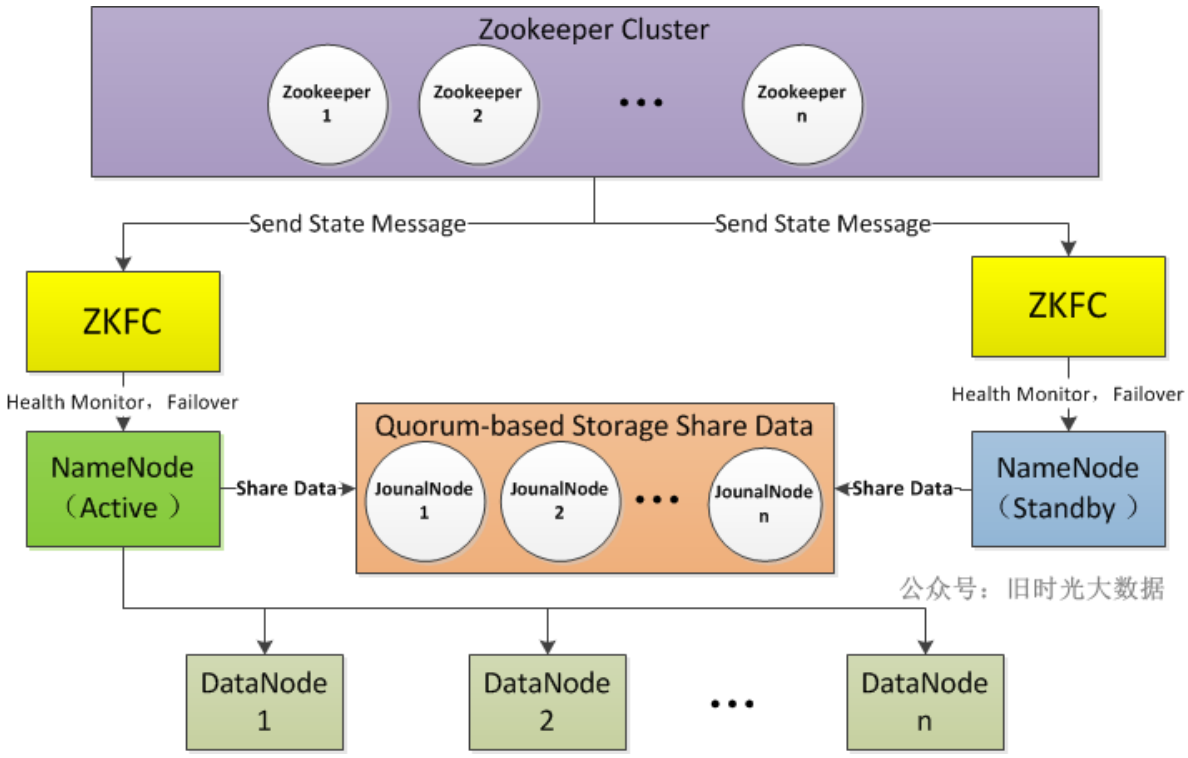
HDFS的高可用指的是HDFS持续对各类客户端提供读、写服务的能力，因为客户端对HDFS的读、写操作之前都要访问NameNode服务器，客户端只有从NameNode获取元数据之后才能继续进行读、写。所以HDFS的高可用的关键在于NameNode上的元数据持续可用。

我们知道2NN的功能是把NameNode的fsimage和edit log做定期融合，融合后传给NameNode，以确保备份到的元数据是最新的，这一点类似于做了一个元数据的快照。Hadoop官方提供了一种quorum journal manager来实现高可用，那么就没必要配置2NN了。

在高可用配置下，edit log不再存放在名称节点，而是存放在一个共享存储的地方，这个共享存储由奇数个Journal Node组成，一般是3个节点(JN小集群)， 每个Journal Node专门用于存放来自NameNode的编辑日志，编辑日志由活跃状态的NameNode写入Journal Node小集群。

那么要有两个NameNode，而且二者之中只能有一个NameNode处于活跃状态（Active），另一个是待命状态（Standby），只有Active的NameNode节点才能对外提供读写HDFS服务，也只有Active态的NameNode才能向Journal Node写入编辑日志；Standby状态的NameNode负责从Journal Node小集群中拷贝数据到本地。另外，各个DataNode也要向Active状态的NameNode报告状态(心跳信息、块信息等)。

2个NameNode与3个Journal Node构成的组保持通信，活跃的名称节点负责往Journal Node集群写入编辑日志，待命的名称节点负责观察Journal Node集群中的编辑日志，并且把日志拉取到待命节点，再加上两个NameNode各自的fsimage镜像文件，这样一来就能确保两个NameNode的元数据保持同步。一旦Active NameNode不可用，提前配置的Zookeeper会把Standby节点自动变Active状态，继续对外提供读写服务。



手动实现高可用的大概流程：

1）准备3台服务器分别用于运行JournalNode进程（也可部署在date node上），准备2台name node用于运行NameNode进程，Data Node数量不限

2）分别启动3台JN服务器上的JournalNode进程，分别在date node服务器启动DataNode进程

3）需要同步2台name node之间的元数据。具体做法：从第一台NameNode拷贝元数据到另一台NameNode，然后启动第一台的NameNode进程,再到另一台名称节点上做standby引导启动

4）把第一台名节点的edit log初始化到JournalNode节点，以供standby状态的NameNode到JournalNode拉取数据

5）启动standby状态的名称节点，这样就能同步fsimage文件

6）模拟故障，检验是否成功实现：手动把active状态的NameNode弄挂掉，正常的话会自动把standby状态的NameNode转变成active.

自动实现通过分布式协调服务者zookeeper来实现。

HDFS的文件结构?

问过的一些公司：美团，360

参考答案：

HDFS metadata以树状结构存储整个HDFS上的文件和目录，以及相应的权限、配额和副本因子（replication factor）等。HDFS Namenode本地目录的存储结构和Datanode数据块存储目录结构，也就是hdfs-site.xml中配置的dfs.namenode.name.dir和dfs.namenode.data.dir。

`NameNode`

HDFS metadata主要存储两种类型的文件

- fsimage
  - 记录某一永久性检查点（Checkpoint）时整个HDFS的元信息
- edits

公众号：旧时光大数据

- 所有对HDFS的写操作都会记录在此文件中

1、文件所在位置

文件所在位置由hdfs-site.xml中的配置项dfs.namenode.name.dir配置。这些文件都存于${dfs.namenode.name.dir}/current文件夹下，在dfs.namenode.name.dir配置项中可以配置多个目录，各个目录存储的文件结构和内容都完全一样，配置多个目录只是起到冗余备份的作用。

```
<property>
  <name>dfs.namenode.name.dir</name>
  <value>file://${hadoop.tmp.dir}/dfs/name</value>
</property>
```

dfs.namenode.name.dir在hdfs-site.xml中定义的默认值为`file://${hadoop.tmp.dir}/dfs/name，其中hadoop.tmp.dir是core-site.xml`中的配置项，hadoop.tmp.dir的默认值为`/tmp/hadoop-${user.name}`。因此文件默认都是存放在/tmp/hadoop-${user.name}/dfs/name/current路径下的。

```
<property>
  <name>hadoop.tmp.dir</name>
  <value>/tmp/hadoop-${user.name}</value>
  <description>A base for other temporary directories.</description>
</property>
```

具体到我的服务器，dfs.namenode.name.dir用了配置的/cloud/data/hadoop/dfs/nn，因此我的服务器上的文件的路径就直接为：/cloud/data/hadoop/dfs/nn/current。

```
<property>
  <name>dfs.namenode.name.dir</name>
  <value>/cloud/data/hadoop/dfs/nn/</value>
</property>
```

2、文件目录结构

```
├── current
|   ├── edits_0000000001362702459-0000000001363401818
|   ├── edits_0000000001363401819-0000000001363931603
|   ├── edits_0000000001363931604-0000000001364566627
|   ├── edits_0000000001364566628-0000000001365069009
|   ├── edits_0000000001365069010-0000000001365209404
|   ├── edits_0000000001365209405-0000000001365211445
|   ├── edits_0000000001365211446-0000000001365211447
|   ├── edits_0000000001365211448-0000000001365211451
|   ├── edits_0000000001365211452-0000000001365211453
|   ├── edits_0000000001365211454-0000000001365211625
|   ├── edits_0000000001365211626-0000000001365213635
|   ├── edits_inprogress_0000000001365213636
|   ├── fsimage_0000000001365431029
|   ├── fsimage_0000000001365431029.md5
|   ├── fsimage_0000000001365431090
|   ├── fsimage_0000000001365431090.md5
|   ├── seen_txid
|   └── VERSION
└── in_use.lock
```

3、文件目录解析

从上面的目录树中可以发现，主要有6类文件：

公众号：旧时光大数据

1）VERSION文件

VERSION是java属性文件，内容大致如下：

```
namespaceID=644097999
clusterID=CID-da027b7b-4e9f-4287-be7a-03735d895bc2
cTime=1603347772521
storageType=NAME_NODE
blockpoolID=BP-1345407316-192.168.100.148-1603347772521
layoutVersion=-63
```

namespaceID是文件系统唯一标识符，在文件系统首次格式化之后生成，在引入Federation特性后，可能会有多个namespace。

clusterID是系统生成或手动指定的集群ID，在-clusterID选项中可以使用它。

cTime表示Namenode存储的创建时间。

storageType表示这个文件存储的是什么进程的数据结构信息（如果是Datanode，则为DATA_NODE）。

blockpoolID表示每一个Namespace对应的块池id，这个id包括了其对应的Namenode节点的ip地址。

layoutVersion表示HDFS永久性数据结构的版本信息，只要数据结构变更，版本号也要递减，此时的HDFS也需要升级，否则磁盘仍旧使用旧版本的数据结构，这会导致新版本的Namenode无法使用。

2）`edits_*`文件

edits文件中存放的是客户端执行的所有更新命名空间的操作。

首先了解一下transactionId的概念。transactionId与客户端每次发起的RPC操作相关，当客户端发起一次RPC请求对Namenode的命名空间修改后，Namenode就会在editlog中发起一个新的transaction用于记录这次操作，每个transaction会用一个唯一的transactionId标识。

edits_*分为两类文件，一类是edits_startTransactionId-endTransactionId，另一类是edits_inprogress_startTransactionId。对于第一类文件，每个edits文件都包含了文件名中从startTransactionId开始到endTransactionId之间的所有事务。

第二类文件表示正在进行处理的editlog，所有从startTransactionId开始的新的修改操作都会记录在这个文件中，直到HDFS重置这个日志文件，重置操作会将inprogress文件关闭，并将inprogress文件改名为正常的editlog文件，即第一类文件，同时还会打开一个新的inprogress文件，记录正在进行的事务。

可以通过oev命令查看edits文件，基本命令格式为：

```
hdfs oev -p 文件类型（通常为xml） -i edits文件 -o 转换后文件输出路径
```

```
<?xml version="1.0" encoding="UTF-8"?>
<EDITS>
  <EDITS_VERSION>-63</EDITS_VERSION>
  <RECORD>
    <OPCODE>OP_START_LOG_SEGMENT</OPCODE>
    <DATA>
      <TXID>63949</TXID>
    </DATA>
  </RECORD>
  <RECORD>
    <OPCODE>OP_END_LOG_SEGMENT</OPCODE>
    <DATA>
      <TXID>63950</TXID>
    </DATA>
  </RECORD>
</EDITS>
```

公众号：旧时光大数据

3）`fsimage_*`文件

fsimage文件其实是hadoop文件系统元数据的一个永久性的检查点，其中包含hadoop文件系统中的所有目录和文件inode的序列化信息。

fsimage_*的具体文件命名规则是这样的：fsimage_endTransactionId，它包含hadoop文件系统中endTransactionId前的完整的HDFS命名空间元数据镜像。

可以通过oiv命令查看fsimage文件，基本命令格式为：

```
hdfs oiv -p 文件类型（通常为xml） -i fsimage文件 -o 转换后文件输出路径
```

4）fsimage_*.md5文件

md5校验文件，用于确保fsimage文件的正确性，可以作用于磁盘异常导致文件损坏的情况。

5）seen_txid文件

这个文件中保存了一个事务id，这个事务id值只会在两种情况下更新：

- 上一个检查点（即合并edits和fsimage文件）时的最新事务id
- 编辑日志重置（即生成一个新的inprogress文件）时的最新事务id

可以发现，这个事务id并不是Namenode内存中最新的事务id。这个文件的作用在于Namenode启动时，利用这个文件判断是否有edits文件丢失，Namenode启动时会检查seen_txid并确保内存中加载的事务id至少超过seen_txid，否则Namenode将终止启动操作。

6）in_use.lock文件

这是一个被Namenode线程持有的锁文件，用于防止多个Namenode线程启动并且并发修改这个存储目录。

`DataNode`

1、文件所在位置

文件所在位置由hdfs-site.xml中的配置项dfs.datanode.data.dir配置。这些文件都存于${dfs.datanode.data.dir}/current文件夹下，在dfs.namenode.name.dir配置项中可以配置多个目录，这里要注意，Datanode的多个存储目录存储的数据块并不相同，并且不同的存储目录可以是异构的，这样的设计可以提高数据块IO的吞吐率，这是与Namenode中很大不同的一个地方。

```
<property>
  <name>dfs.datanode.data.dir</name>
  <value>file://${hadoop.tmp.dir}/dfs/data</value>
</property>

```

dfs.datanode.data.dir在hdfs-site.xml中定义的默认值为`file://${hadoop.tmp.dir}/dfs/data`，其中`hadoop.tmp.dir`是`core-site.xml`中的配置项，hadoop.tmp.dir的默认值为`/tmp/hadoop-${user.name}`。因此文件默认都是存放在/tmp/hadoop-${user.name}/dfs/data/current路径下的。

```
<property>
  <name>hadoop.tmp.dir</name>
  <value>/tmp/hadoop-${user.name}</value>
  <description>A base for other temporary directories.</description>
</property>

```

具体到服务器，dfs.datanode.data.dir配置了如下值，因此服务器上的文件的路径就包括以下8个路径：/cloud/data*/hadoop/dfs/dn/current。

```
<property>
  <name>dfs.datanode.data.dir</name>
 <value>/cloud/data1/hadoop/dfs/dn,/cloud/data2/hadoop/dfs/dn,/cloud/data3/hadoop/dfs/dn,/cloud/data4/hadoop/dfs/dn,/cloud/data5/hadoop/dfs/dn,/cloud/data6/hadoop/dfs/dn,/cloud/data7/hadoop/dfs/dn,/cloud/data8/hadoop/dfs/dn
  </value>
</property>

```

2、文件目录结构

以dfs.datanode.data.dir配置项配置的其中一个目录/cloud/data1/hadoop/dfs/dn为例，它的目录结构为：

```
├── current
│   ├── BP-615261695-192.168.100.74-1602499969654
│   │   ├── current
│   │   │   ├── finalized
│   │   │   │   ├── subdir0
│   │   │   │   ├── subdir1
│   │   │   │   ├── subdir10
│   │   │   │   ├── subdir11
│   │   │   │   ├── subdir12
│   │   │   │   ├── subdir13
│   │   │   │   ├── subdir14
│   │   │   │   ├── subdir15
│   │   │   │   ├── subdir16
│   │   │   │   ├── subdir17
│   │   │   │   ├── subdir18
│   │   │   │   ├── subdir19
│   │   │   │   ├── subdir2
│   │   │   │   ├── subdir20
│   │   │   │   ├── subdir21
│   │   │   │   ├── subdir22
│   │   │   │   ├── subdir23
│   │   │   │   ├── subdir24
│   │   │   │   ├── subdir25
│   │   │   │   ├── subdir26
│   │   │   │   ├── subdir27
│   │   │   │   ├── subdir28
│   │   │   │   ├── subdir29
│   │   │   │   ├── subdir3
│   │   │   │   ├── subdir30
│   │   │   │   ├── subdir31
│   │   │   │   ├── subdir4
│   │   │   │   ├── subdir5
│   │   │   │   ├── subdir6
│   │   │   │   ├── subdir7
│   │   │   │   ├── subdir8
│   │   │   │   └── subdir9
│   │   │   ├── rbw
│   │   │   └── VERSION
│   │   ├── scanner.cursor
│   │   └── tmp
│   │   ├── blk_1080824082
│   │   ├── blk_1080824082_7111789.meta
│   │   ├── blk_1080834049
│   │   └── blk_1080834049_7120152.meta
│   └── VERSION
```

公众号：旧时光大数据

```
46    └── in_use.lock
47
```

由于篇幅所限，这里只展开了5层，在实际情况中，subdir0subdir31这32个文件夹每个文件夹下面还是subdir0subdir31这32个文件夹，这些文件夹下面才是类似于tmp文件夹下的blk_*文件。

3、文件目录解析

1）BP-*-*-*文件夹

这个目录是一个块池目录，块池目录保存了一个块池在当前存储目录下存储的所有数据块，在Federation部署方式中，Datanode的一个存储目录会包含多个以BP开头的块池目录。BP后面会紧跟一个唯一的随机块池ID。接下来的第2个*代表当前块池对应的Namenode的IP地址。最后一个*代表这个块池的创建时间。

2）外层VERSION文件

```
1   storageID=DS-fd6c86e2-c1ab-4541-9693-5844c2359a0f
2   clusterID=CID-a56f3601-ac4b-4c37-95fd-3b04c24a54c9
3   cTime=0
4   datanodeUuid=b0f4eafa-54b1-4966-acf9-3986435d5cd6
5   storageType=DATA_NODE
6   layoutVersion=-57
7
```

`storageID`：存储 id 号

`clusterID`：集群 id，全局唯一

`cTime`：标记了 datanode 存储系统的创建时间，对于刚刚格式化的存储系统，这个属性为 0；但是在文件系统升级之后，该值会更新到新的时间戳。

`datanodeUuid`：datanode 的唯一识别码

`storageType`：存储类型

`layoutVersion`：是一个负整数。通常只有 HDFS 增加新特性时才会更新这个版本号

3）内层VERSION文件

```
1   namespaceID=800713668
2   cTime=1602499969654
3   blockpoolID=BP-615261695-192.168.100.74-1602499969654
4   layoutVersion=-57
5
```

`namespaceID`：是 datanode 首次访问 namenode 的时候从 namenode 处获取的storageID， 对每个 datanode 来说是唯一的（但对于单个 datanode 中所有存储目录来说则是相同的），namenode 可用这个属性来区分不同 datanode。（但我查看了不同datanode上的都相同）

`cTime`：标记了 datanode 存储系统的创建时间，对于刚刚格式化的存储系统，这个属性为 0；但是在文件系统升级之后，该值会更新到新的时间戳。

`blockpoolID`：一个 block pool id 标识一个 block pool，并且是跨集群的全局唯一。当一个新的 Namespace 被创建的时候(format 过程的一部分)会创建并持久化一个唯一 ID。在创建过程构建全局唯一的 BlockPoolID 比人为的配置更可靠一些。NN 将 BlockPoolID 持久化到磁盘中，在后续的启动过程中，会再次 load 并使用。（查看了发现不同datanode上的都相同，BP-1008895165-10.252.87.50-1593312108051的命名就遵循BP-random integer-NameNode-IP address-creation time）

`layoutVersion`：是一个负整数。通常只有 HDFS 增加新特性时才会更新这个版本号。

4）finalized/rbw/tmp文件夹

finalized、rbw和tmp目录都是用于存储数据块的文件夹，包括数据块文件及其对应的校验和文件。他们的区别是存放不同状态下的数据块副本文件。我们知道Datanode上保存的数据块副本有5种状态：FINALIZED，RBW（replica being written），RUR（replica under recovery），RWR（replica waiting to be recovered），TEMPORATY（复制数据块或者进行集群数据块平衡操作时的状态）

finalized目录保存所有FINALIZED状态的副本，rbw目录保存RBW、RWR、RUR状态的副本，tmp目录保存TEMPORARY状态的副本。

如果对应到具体的过程，那么当客户端发起写请求创建一个新的副本时，这个副本会被放到rbw目录中；当在数据块备份和集群平衡存储过程中创建一个新的副本时，这个副本就会放到tmp目录中；一旦一个副本完成写操作并被提交，它就会被移到finalized目录中。当Datanode重启时，tmp目录中的所有副本将会被删除，rbw目录中的副本将会被加载为RWR状态，finalized目录中的副本将会被加载为FINALIZED状态。

5）finalized特殊目录结构解析

inalized目录存储了已经完成写入操作的数据块，由于这样的数据块可能非常多，所以finalized目录会以特定的目录结构存储这些数据块。

在我的机器上，finalized文件夹有32个子文件夹（分别为subdir0-31），每个子文件夹下又有32个子文件夹（同样为subdir0-31），接着才是存的真正的数据块blk文件及其校验文件。你可能会发现自己的目录下不是32个子文件夹，而是64个或者256个，不要奇怪，这是hadoop不同版本之间的差异，下面简要介绍一下不同版本下的目录结构：

- 早期版本：当存储目录中的数据块超过64个时，将创建64个子目录来存储数据块。一个父目录最多可以创建64个子目录，一个子目录最多可以存放64个数据块以及64个子目录。
- 2.6版本：finalized目录下拥有256个一级子目录，每个一级子目录下可以拥有256个二级子目录。
- 2.8.5版本：finalized目录下拥有32个一级子目录，每个一级子目录下可以拥有32个二级子目录。

以2.8.5版本为例，数据块是根据什么判断自己应放在哪个目录下的呢？答案是根据数据块的id，具体的判断规则如下代码所示(DatanodeUtil类的idToBlockDir方法)：

```
1   public static File idToBlockDir(File root, long blockId) {
2       int d1 = (int) ((blockId >> 16) & 0x1F);
3       int d2 = (int) ((blockId >> 8) & 0x1F);
4       String path = DataStorage.BLOCK_SUBDIR_PREFIX + d1 + SEP +
5           DataStorage.BLOCK_SUBDIR_PREFIX + d2;
6       return new File(root, path);
7   }
8
```

一级目录由d1指定，二级目录由d2指定，d1是数据块id右移16位，再取最后5位得到的数值，可以发现，它的范围是0-31。d2是数据块id右移8位，再取最后5位得到的数值，可以发现，它的范围也是0-31。一级目录和二级目录都对应32个文件夹，与预期相符。

2.6版本的判断规则与2.8.5完全一样，不同点是都是取最后8位，而不是5位，因此，2.6版本的一级目录和二级目录对应的文件夹个数都是256个。

6）in_use.lock文件

这是一个被Datanode线程持有的锁文件，用于防止多个Datanode线程启动并发修改这个存储目录。

7）scanner.cursor文件

这个文件记录了访问文件的游标，文件具体内容如下所示：

```
{
  "lastSavedMs" : 1604459692263,
  "iterStartMs" : 1604314428656,
  "curFinalizedDir" : null,
  "curFinalizedSubDir" : null,
  "curEntry" : null,
  "atEnd" : true
}

```

8）blk_*文件

blk_*是数据块文件，其中*代表的数据是数据块id。

blk_*_*.meta是数据块校验文件，其中第一个*是数据块id，第二个*代表数据块的版本号。

blk的meta校验文件（保存blk的checksum信息）大小大概是blk文件大小的1/128，因为每512字节做一次校验生成4字节校验，在机器上验证了一下大小，接近1/128。

HDFS的默认副本数？为什么是这个数量？如果想修改副本数怎么修改？

问过的一些公司：美团

参考答案：

hdfs的默认副本数量是3个，配置在/etc/hadoop/conf/hdfs-site.xml中（修改也是在这）

```
<property>
    <name>dfs.replication</name>
    <value>3</value>
</property>

```

关于HDFS的副本数为什么选择3，可看以下内容：

HDFS采用一种称为机架感知的策略来改进数据的可靠性、可用性和网络带宽的利用率。

在大多数情况下，HDFS的副本系数是3，HDFS的存放策略是一个副本存放在本地机架节点上，另一个副本存放在不同一机架的节点上，第三个副本存放在在与第二个节点同一机架的不同节点上。这种策略减少了机架间的数据传输，提高了写操作的效率。机架错误的概率远比节点错误的概率小，所以这种策略不会对数据的可靠性和可用性造成影响。与此同时，因为数据只存在两个机架上，这种策略减少了读数据时需要的网络传输带宽。

在这种策略下，副本并不是均匀地分布在机架上。这种策略在不损坏可靠性和读取性能的情况下，改善了写的性能。

介绍下HDFS的Block

问过的一些公司：京东

参考答案：

Block概念

磁盘有一个Block size的概念，它是磁盘读/写数据的最小单位。构建在这样的磁盘上的文件系统也是通过块来管理数据的，文件系统的块通常是磁盘块的整数倍。文件系统的块一般为几千字节(byte)，磁盘块一般为512字节(byte)。

HDFS也有Block的概念，但它的块是一个很大的单元，默认是64MB。像硬盘中的文件系统一样，在HDFS中的文件将会按块大小进行分解，并作为独立的单元进行存储。但和硬盘中的文件系统不一样的是，存储在块中的硬的一个比块小的文件并不会占据一个块大小盘物理空间（HDFS中一个块只存储一个文件的内容）。

对HDFS进行块抽象有哪些好处呢?

1、一个显而易见的好处是：一个文件的大小，可以大于网络中任意一个硬盘的大小。

文件的块并不需要存储在同一个硬盘上，一个文件的快可以分布在集群中任意一个硬盘上。事实上，虽然实际中并没有，整个集群可以只存储一个文件，该文件的块占满整个集群的硬盘空间。

2、使用抽象块而非整个文件作为存储单元，大大简化了系统的设计。

简化设计，对于故障种类繁多的分布式系统来说尤为重要。以块为单位，一方面简化存储管理，因为块大小是固定的，所以一个硬盘放多少个块是非常容易计算的；另一方面，也消除了元数据的顾虑，因为Block仅仅是存储的一块数据，其文件的元数据，例如权限等就不需要跟数据块一起存储，可以交由另外的其他系统来处理。

3、块更适合于数据备份，进而提供数据容错能力和系统可用性。

为了防止数据块损坏或者磁盘或者机器故障，每一个block都可以被分到少数几天独立的机器上(默认3台)。这样，如果一个block不能用了，就从其他的一处地方，复制过来一份。

HDFS的块默认大小，64M和128M是在哪个版本更换的？怎么修改默认块大小?

问过的一些公司：祖龙娱乐

参考答案：

Hadoop 2.7.2版本及之前默认64MB，Hadoop 2.7.3版本及之后默认128M

通过修改hdfs-site.xml文件中的dfs.blocksize对应的值修改block大小

```
<property>
    <name>dfs.blocksize</name>
    <value>134217728</value>
</property>

```

HDFS的block为什么是128M？增大或减小有什么影响?

可回答：Hadoop块大小及其原因

问过的一些公司：美团，货拉拉(2021.07)

参考答案：

1、首先先来了解几个概念

寻址时间：HDFS中找到目标文件block块所花费的时间。

原理：文件块越大，寻址时间越短，但磁盘传输时间越长；文件块越小，寻址时间越长，但磁盘传输时间越短。

2、为什么block不能设置过大，也不能设置过小

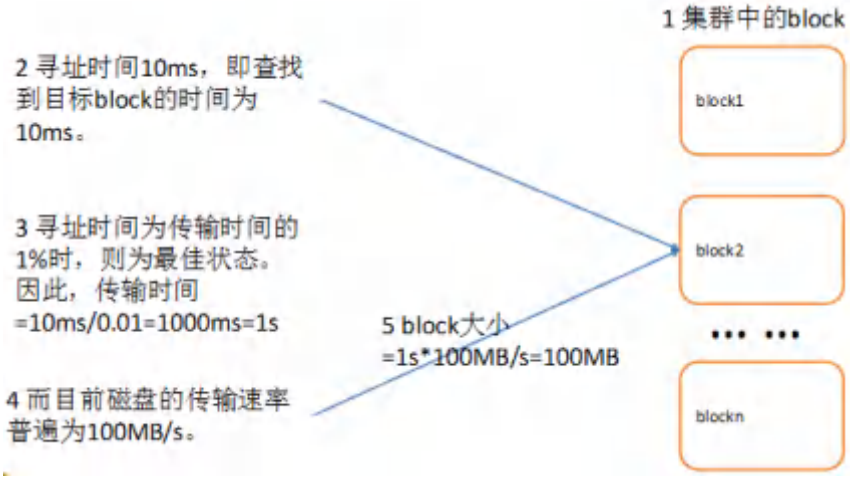
如果块设置过大，一方面从磁盘传输数据的时间会明显大于寻址时间，导致程序在处理这块数据时，变得非常慢；另一方面，MapReduce中的map任务通常一次只处理一个块中的数据，如果块过大运行速度也会很慢。

如果设置过小，一方面存放大量小文件会占用NameNode中大量内存来存储元数据，而NameNode的内存是有限的，不可取；另一方面块过小，寻址时间增长，导致程序一直在找block的开始位置。因此，块适当设置大一些，减少寻址时间，传输一个有多个块组成的文件的时间主要取决于磁盘的传输速度。

3、块大小多少合适

如果寻址时间约为10ms，而传输速率为100MB/s，为了使寻址时间仅占传输时间的1%，我们要将块大小设置约为100MB。默认的块大小128MB。

块的大小：10ms x 100 x 100M/s = 100M，如图



如果增加文件块大小，那就需要增加磁盘的传输速率。

比如，磁盘传输速率为200MB/s时，一般设定block大小为256MB；磁盘传输速率为400MB/s时，一般设定block大小为512MB。

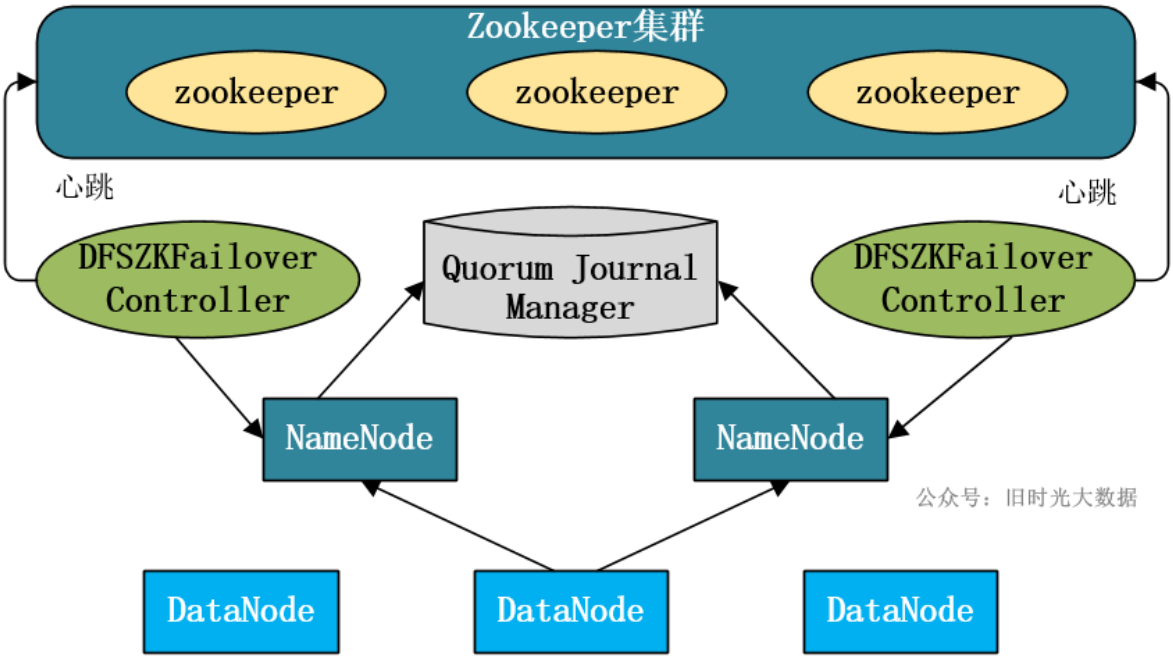
HDFS HA怎么实现？是个什么架构？

可回答：1）HDFS HA的实现原理；2）Hadoop HA中各个节点是怎么布置的

问过的一些公司：腾讯（微众）x2，头条，猿辅导，京东，蘑菇街x3，快手，顺丰，大华(2021.08)，四方伟业(2021.08)，美团买菜(2021.09)，小红书(2021.11)

在Hadoop2.X之前，NameNode是集群中可能发生单点故障的节点，每个HDFS集群只有一个NameNode，一旦这个节点不可用，则整个HDFS集群将处于不可用状态。

HDFS高可用（HA）方案就是为了解决上述问题而产生的，在HA HDFS集群中会同时运行两个NameNode，一个作为活动的NameNode（Active），一个作为备份的NameNode（Standby）。备份的NameNode的命名空间与活动的NameNode是实时同步的，所以当活动的NameNode发生故障而停止服务时，备份NameNode可以立即切换为活动状态，而不影响HDFS集群服务。



在一个HA集群中，会配置两个独立的Namenode。在任意时刻，只有一个节点作为活动的节点，另一个节点则处于备份状态。活动的Namenode负责执行所有修改命名空间以及删除备份数据块的操作，而备份的Namenode则执行同步操作，以保持与活动节点命名空间的一致性。

为了使备份节点与活动节点的状态能够同步一致，两个节点都需要同一组独立运行的节点（JournalNodes，JNS）通信。当Active Namenode执行了修改命名空间的操作时，它会定期将执行的操作记录在editlog中，并写入JNS的多数节点中。而Standby Namenode会一直监听JNS上editlog的变化，如果发现editlog有改动，Standby Namenode就会读取editlog并与当前的命名空间合并。当发生了错误切换时，Standby节点会保证已经从JNS上读取了所有editlog并与命名空间合并，然后才会从Standby状态切换为Active状态。通过这种机制，保证了Active Namenode与Standby Namenode之间命名空间状态的一致性，也就是第一关系链的一致性。

为了使错误切换能够很快的执行完毕，就要保证Standby节点也保存了实时的数据快的存储信息，也就是第二关系链。这样发生错误切换时，Standby节点就不需要等待所有的数据节点进行全量数据块汇报，而直接可以切换到Active状态。为了实现这个机制，Datanode会同时向这两个Namenode发送心跳以及块汇报信息。这样就实现了Active Namenode 和standby Namenode 的元数据就完全一致，一旦发生故障，就可以马上切换，也就是热备。
这里需要注意的是 Standby Namenode只会更新数据块的存储信息，并不会向namenode 发送复制或者删除数据块的指令，这些指令只能由Active namenode发送。

在HA架构中有一个非常重非要的问题，就是需要保证同一时刻只有一个处于Active状态的Namenode，否则机会出现两个Namenode同时修改命名空间的问，也就是脑裂（Split-brain）。脑裂的HDFS集群很可能造成数据块的丢失，以及向Datanode下发错误的指令等异常情况。

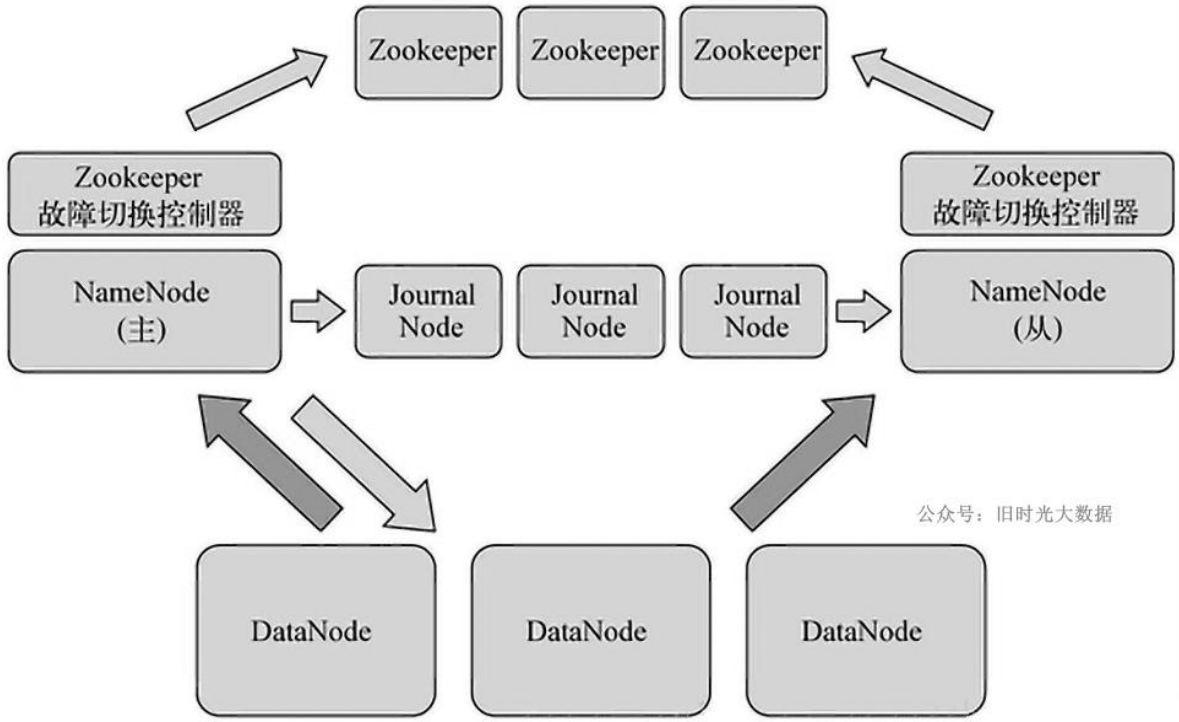
为了预防脑裂的情况，HDFS提供了三个级别的隔离机制（fencing）：

- 共享存储隔离：同一时间只允许一个Namenode向JournalNodes写入editlog数据。
- 客户端隔离：同一时间只允许一个Namenode响应客户端的请求。
- Datanode隔离：同一时间只允许一个Namenode向Datanode下发名字节点指令，李如删除、复制数据块指令等等。

在HA实现中还有一个非常重要的部分就是Active Namenode和Standby Namenode之间如何共享editlog日志文件。Active Namenode会将日志文件写到共享存储上。Standby Namenode会实时的从共享存储读取edetlog文件，然后合并到Standby Namenode的命名空间中。这样一旦Active Namenode发生错误，Standby Namenode可以立即切换到Active状态。在Hadoop2.6中，提供了QJM（Quorum Journal Manager）方案来解决HA共享存储问题。

公众号：旧时光大数据

所有的HA实现方案都依赖于一个保存editlog的共享存储，这个存储必须是高可用的，并且能够被集群中所有的Namenode同时访问。**Quorum Journa是一个基于paxos算法的HA设计方案。**



Quorum Journal方案中有两个重要的组件。

`JournalNoe（JN）`：运行在N台独立的物理机器上，它将editlog文件保存在JournalNode的本地磁盘上，同时JournalNode还对外提供RPC接口QJournalProtocol以执行远程读写editlog文件的功能。

`QuorumJournalManager(QJM)`：运行在NmaeNode上，（目前HA集群只有两个Namenode），通过调用RPC接口QJournalProtocol中的方法向JournalNode发送写入、排斥、同步editlog。

Quorum Journal方案依赖于这样一个概念：HDFS集群中有2N+1个JN存储editlog文件，这些editlog 文件是保存在JN的本地磁盘上的。每个JN对QJM暴露QJM接口QJournalProtocol，允许Namenode读写editlog文件。当Namenode向共享存储写入editlog文件时，它会通过ＱＪＭ向集群中所有的ＪＮ发送写editlog文件请求，当有一半以上的ＪＮ返回写操作成功时，即认为写成功。这个原理是基于Paxos算法的。

使用Quorum Journal实现的HA方案有一下优点：

- JN进程可以运行在普通的PC上，而无需配置专业的共享存储硬件。
- 不需要单独实现fencing机制，Quorum Journal模式中内置了fencing功能。
- Quorum Journa不存在单点故障，集群中有2N+1个Journal，可以允许有Ｎ个Journal Node死亡。
- JN不会因为其中一个机器的延迟而影响整体的延迟，而且也不会因为JN数量的增多而影响性能（因为Namenode向JournalNode发送日志是并行的）

导入大文件到HDFS时如何自定义分片?

问过的一些公司：shopee

参考答案：

通过设置block的大小（dfs.block.size）来设置分片，默认是128M

公众号：旧时光大数据

HDFS的mapper和reducer的个数如何确定？reducer的个数依据是什么？

问过的一些公司：创略科技

参考答案：

map数量

影响map个数（split个数）的主要因素有：

文件的大小。当块（dfs.block.size）为128m时，如果输入文件为128m，会被划分为1个split；当块为256m，会被划分为2个split。

文件的个数。FileInputFormat按照文件分割split，并且只会分割大文件，即那些大小超过HDFS块的大小的文件。如果HDFS中dfs.block.size设置为128m，而输入的目录中文件有100个，则划分后的split个数至少为100个。

splitSize的大小。分片是按照splitszie的大小进行分割的，一个split的大小在没有设置的情况下，默认等于hdfs block的大小。

```
splitSize=max{minSize,min{maxSize,blockSize}}

```

map数量由处理的数据分成的block数量决定default_num = total_size / split_size

reduce数量

reduce的数量job.setNumReduceTasks(x); x为reduce的数量。不设置的话默认为1

HDSF通过那个中间组件去存储数据

问过的一些公司：大华

参考答案：

DataNode

HDFS跨节点怎么进行数据迁移

问过的一些公司：ebay

参考答案：

通过DistCp来完成数据迁移

DistCp（分布式拷贝）是用于大规模集群内部和集群之间拷贝的工具。它使用Map/Reduce实现文件分发（DistCp原理是在Hadoop集群中使用MapReduce分布式拷贝数据），错误处理和恢复，以及报告生成。它把文件和目录的列表作为map任务的输入，每个任务会完成源列表中部分文件的拷贝。

迁移之前需要考虑的事情

- 迁移总数据量有多少？
- 新老集群之间的带宽有多少？能否全部用完？为了减少对线上其他业务的影响最多可使用多少带宽？
- 如何限制迁移过程中使用的带宽？
- 迁移过程中，哪些文件可能发生删除，新增数据的情况？哪些目录可能会发生新增文件的情况？
- 迁移后的数据一致性校验怎么做？

公众号：旧时光大数据

- 迁移后的HDFS文件权限如何跟老集群保持一致?

迁移方案

1、迁移数据量评估。

通过hdfs dfs -du -h /命令查看各目录总数据量。按业务划分，统计各业务数据总量。

2、制定迁移节奏。

由于数据量大，带宽有限，HDFS中的文件每天随业务不断变化，所以在文件变化之前全部迁移完成是不现实的。建议按业务、分目录、分批迁移。

3、迁移工具选择。

使用Hadoop自带数据迁移工具Distcp，只需要简单的命令即可完成数据迁移。

命令示意：hadoop distcp hdfs://nn1:8020/data hdfs://nn2:8020/

4、迁移时间评估。

由于老集群每天仍然在使用，为了减小对线上业务的影响，尽量选择老集群低负载运行的时间段来进行数据迁移。

5、对新老集群之间的网络进行硬件改造。

咨询运维同学，新老集群之间的最大传输带宽是多少，如果带宽跑满的话会不会影响线上其他业务。

能否对新老集群之间的网络进行硬件改造，例如通过新老集群之间搭网线的方式来提升网络传输的带宽并消除对其他线上业务的影响。

6、数据迁移状况评估。

在完成上面所有准备之后，先尝试进行小数据量的迁移，可以先进行100G的数据迁移、500G的数据迁移、1T的数据迁移，以评估数据迁移速率并收集迁移过程中遇到的问题。

迁移工具Distcp

工具使用很简单，只需要执行简单的命令即可开始数据迁移，可选参数如下：

hadoop distcp 源HDFS文件路径 目标HDFS文件路径

同版本集群拷贝（或者协议兼容版本之间的拷贝）使用HDFS协议

```
1   hadoop distcp hdfs://src-name-node:3333/user/src/dir hdfs://dst-name-
    node:4444/user/dst/dir
2
```

不同版本集群拷贝（比如1.x到2.x）使用hftp协议或者webhdfs协议，都是使用hdfs的HTTP端口

```
1   hadoop distcp hftp://src-name-node:80/user/src/dir hftp://dst-name-
    node:80/user/dst/dir
2   hadoop distcp webhdfs://src-name-node:80/user/src/dir webhdfs://dst-name-
    node:80/user/dst/dir
3
```

1、Distcp的原理

Distcp的本质是一个MapReduce任务，只有Map阶段，没有Reduce阶段，具备分布式执行的特性。在Map任务中从老集群读取数据，然后写入新集群，以此来完成数据迁移。

2、迁移期间新老两个集群的资源消耗是怎样的?

Distcp是一个MapReduce任务，如果在新集群上执行就向新集群的Yarn申请资源，老集群只有数据读取和网络传输的消耗。

3、如何提高数据迁移速度?

Distcp提供了 `-m <arg>` 参数来设置map任务的最大数量（默认20），以提高并发性。注意这里要结合最大网络传输速率来设置。

4、带宽如何限制?

Distcp提供了 `-bandwidth <arg>` 参数来控制单个Map任务的最大带宽，单位是MB。

5、迁移之后的数据一致性怎么校验?

Distcp负责进行CRC校验，可以通过-skipcrccheck参数来跳过校验来提供性能。

6、迁移之后的文件权限是怎样的?

Distcp提供了 `-p <arg>` 参数来在新集群里保留状态（rbugpcaxt）（复制，块大小，用户，组，权限，校验和类型，ACL，XATTR，时间戳）。如果没有指定 `-p <arg>` 参数，权限是执行MapReduce任务的用户权限，迁移完成以后需要手动执行chown命令变更。

7、迁移的过程中老集群目录新增了文件，删除了文件怎么办?

把握好迁移节奏，尽量避免这些情况的出现。Distcp在任务启动的时候就会将需要copy的文件列表从源HDFS读取出来。如果迁移期间新增了文件，新增的文件会被漏掉。删除文件会导致改文件copy失败，可以通过 -i参数忽略失败。

8、迁移中遇到文件已存在的情况怎么办?

Distcp提供了-overwrite 参数来覆盖已存在的文件。

9、迁移了一半，任务失败了怎么办?

删除掉新集群中的脏数据，重新执行迁移命令。不加-overwrite参数，来跳过已存在的文件。

10、遇到需要对一个文件增量同步怎么办?

Distcp提供-append参数将源HDFS文件的数据新增进去而不是覆盖它。

HDFS的数据一致性靠什么保证?

问过的一些公司：阿里

参考答案：

1、hdfs的namenode机制

hdfs只有一个namenode,一旦namenode出现问题，数据块信息无法寻找。namenode中的元数据信息在工作时，会将元数据信息缓存在内存中。namenode将这些内存中的数据备份到磁盘中fsimage，每当有数据进行存入时，元数据信息会被追加到editLog文件中，内存中实际是这两部分的集合。那么什么时候将editlog与fsimage进行合并呢？HDFS引入了seconderynamenode这个节点，该节点主要功能就是定期或者等到editlog达到一定数量（可在hdfs_site.xml设置）之后，就会copy namenode中的editlog和fsimage文件到seconderynamenode中，在seconderynamenode合并之后再更新namenode上的fsimage文件。如果一旦namenode中的数据丢失，seconderynamenode中的数据将作为备份，确保元数据信息不会丢失。

2、心跳机制

namenode与datanode之间通过心跳（每3秒一次可以在配置文件中设置）信息来确认并更新datanode的元数据信息。若datanode发生故障，namenode就会将取消对该datanode节点的信任（之后的读写都不会在该datanode节点上），同时，namenode将该节点上的数据进行备份处理。namenode的备份节点的选择主要依据的是拓扑距离和具体node负载情况。

3、安全模式

HDFS在初始化阶段会进入安全模式，在安全模式下namenode不允许操作。namenode会与连接的datanode进行安全检查，只有当安全的数据块比值达到设定的阈值才会退出安全模式，

4、回滚机制

在hdfs升级或者执行数据写入时，相关的数据将会被保留备份，如果成功，则更新备份，失败，则使用备份信息。

5、安全校验

为了避免网络传输造成的数据错误问题，HDFS采用了校验和机制。各个node之间数据备份和数据读取，校验通过数据备份成功，否则备份失败，重新备份

6、回收站

当数据文件从hdfs中删除时，文件并没有消失，而是转存/trash目录下。如果误删除的话，可以在该目录下找回文件，文件的存储时间可以配置fs.trash.interval。超过该时间namenode将该文件的元数据删除，同时datanode上的文件将会被删除。

HDFS怎么保证数据安全

问过的一些公司：bigo

参考答案：

1、存储在HDFS系统上的文件，会分割成128M大小的block存储在不同的节点上，block的副本数默认3份，也可配置成更多份；

2、第一个副本一般放置在与client（客户端）所在的同一节点上（若客户端无datanode，则随机放），第二个副本放置到与第一个副本同一机架的不同节点，第三个副本放到不同机架的datanode节点，当取用时遵循就近原则；

3、namonode会把其上面的数据备份到其他一个datanode节点上，保证数据的副本数量；

4、datanode会默认每小时把自己节点上的所有块状态信息报告给namenode；

5、采用safemode模式：datanode会周期性的报告block信息。Namenode会计算block的损坏率，当阀值<0.999f时系统会进入安全模式，HDFS只读不写。 HDFS元数据采用secondaryname备份或者HA备份

HDFS中向DataNode写数据失败了怎么办

问过的一些公司：阿里社招

参考答案：

1、此时，Pipeline数据流管道会被关闭，ACK queue中的packets会被添加到data queue的前面以确保不会发生packets数据包的丢失；

2、在正常的DataNode节点上的以保存好的block的ID版本会升级——这样发生故障的DataNode节点上的block数据会在节点恢复正常后被删除，失效节点也会被从Pipeline中删除；

公众号：旧时光大数据

3、剩下的数据会被写入到Pipeline数据流管道中的其他两个节点中

Hadoop2.x HDFS快照

问过的一些公司：阿里社招

参考答案：

1、介绍

快照snapshots是HDFS文件系统的只读的基于某时间点的拷贝，可以针对某个目录，或者整个文件系统做快照。快照比较常见的应用场景是数据备份，以防一些用户错误或灾难恢复。

快照的高效性实现：

- 快照可以即时创建，耗时仅为O(1)。--excluding the inode lookup time
- 只有当涉及到快照目录的修改被执行时，才会产生额外的内存消耗。而且内存消耗为O(M)，其中M是被修改的文件或目录数。
- 创建快照时，block块并不会被拷贝。快照文件中只记录了block列表和文件大小，不会做任何数据拷贝。
- 快照不会对正常的HDFS操作有任何影响：创建快照以后发生的修改操作，被按操作时间的倒序（from newer to older）记录下来。所以当前的数据能被直接获取，而快照点的数据，则通过在当前的数据基础上减去执行过的操作来获取。

快照目录

我们可以在任何被设置为snapshottable的目录上执行快照，对一个目录最多可以创建65536个快照。管理员可以把任何目录设置为snapshottable，没有限制。如果一个目录下已经存在快照，那么只有当先删除所有快照后才能对这个目录进行删除和重命名等操作。

不允许嵌套的snapshottable目录。也就是说，如果一个目录被设置为snapshottable，那么它的父目录和子目录都不允许被设置为snapshottable。

快照路径

快照被存放在一个被命名为.snapshot的目录中。比如/foo是一个snapshottable目录，/foo中有一个目录为/foo/bar，对/foo创建一个快照s0。那么

```
/foo/.snapshot/s0/bar

```

就是/foo/bar目录对应的快照。可以通过".snapshot"路径直接访问和操作快照数据。例如：

列出一个目录的所有快照：

```
hdfs dfs -ls /foo/.snapshot

```

列出快照s0中的所有文件：

```
hdfs dfs -ls /foo/.snapshot/s0

```

从快照中复制文件：

```
hdfs dfs -cp /foo/.snapshot/s0/bar /tmp

```

公众号：旧时光大数据

2、快照操作

快照管理

只有集群的管理员才有权限进行如下操作。

- **允许快照**

把一个目录设置为snapshottable，就是设置允许对一个目录创建快照。

```
hdfs dfsadmin -allowSnapshot <path>
```

对应的API为HdfsAdmin中的void allowSnapshot(Path path)。

- **禁止快照**

把原本snapshottable的目录设置为禁止快照，不允许对该目录创建快照。在对一个目录设置禁止快照之前，要先删除该目录的所有快照。

```
hdfs dfsadmin -disallowSnapshot <Path>
```

对应的API为HdfsAdmin中的void disallowSnapshot(Path path)。

用户操作

- **创建快照**

为一个目录创建快照，只有目录的所属人权限能为这个目录创建快照。

```
hdfs dfs -createSnapshot <Path> [<snapshotName>]
```

snapshotName是要创建的快照名，如果没有定义，默认取当前时间戳作为快照名。类似""s20130412-151029.033“。

对应的API为FileSystem中的Path createSnapshot(Path path)和Path createSnapshot(Path path, String snapshotName)。

- **删除快照**

删除一个snapshottable目录的一个快照。删除操作也需要目录的所属人权限。

```
hdfs dfs -deleteSnapshot <Path> <snapshotName>
```

对应的API为FileSystem中的void deleteSnapshot(Path path, String snapshotName)。

- **重命名快照**

重命名一个快照。也需要该目录的所属人权限。

```
hdfs dfs -renameSnapshot <Path> <oldName> <newName>
```

对应的API为FileSystem中的void renameSnapshot(Path path, String oldName, String newName)。

- **列出所有允许快照目录**

列出所有当前用户有权限获取的，允许快照的目录。

公众号：旧时光大数据

```
hdfs lsSnapshottableDir
```

对应的API为DistributeFileSystem中的snapshottableDirectoryStatus[] getSnapshottableDirectoryListing()。

- **对比快照**

对比两个快照。这个操作需要用户对两个快照目录同时具有读权限。

```
hdfs snapshotDiff <Path> <fromSnapshot> <toSnapshot>
```

对应API为DistributeFileSystem中的SnapshotDiffReport getSnapshotDiffReport(Path path, String fromSnapshot, String toSnapshot)。

HDFS文件存储的方式?

问过的一些公司：字节(2021.07)

参考答案：

以数据块的方式存储数据。默认一个数据块128M，该数值可以修改。

注意：这里的128仅仅是切分数据的阈值。

一个大的数据被切分成多个小的128M的数据块，分别存储在集群多个节点的不同位置

HDFS写数据过程，写的过程中有哪些故障，分别会怎么处理?

问过的一些公司：字节(2021.07)

参考答案：

写数据过程中，可能的异常模式如下所列：

- Client 在写入过程中，自己挂了
- Client 在写入过程中，有 DataNode 挂了
- Client 在写入过程中，NameNode 挂了

对于以上所列的异常模式，都有分别对应的恢复模式。

1、Client 在写入过程中，自己挂了

当 Client 在写入过程中，自己挂了。由于 Client 在写文件之前需要向 NameNode 申请该文件的租约（lease），只有持有租约才允许写入，而且租约需要定期续约。所以当 Client 挂了后租约会超时，HDFS 在超时后会释放该文件的租约并关闭该文件，避免文件一直被这个挂掉的 Client 独占导致其他人不能写入。这个过程称为 lease recovery。

在发起 lease recovery 时，若多个文件 block 副本在多个 DataNodes 上处于不一致的状态，首先需要将其恢复到一致长度的状态。这个过程称为 block recovery。 这个过程只能在 lease recovery 过程中发起。

2、Client 在写入过程中，有 DataNode 挂了

当 Client 在写入过程中，有 DataNode 挂了。写入过程不会立刻终止（如果立刻终止，易用性和可用性都太不友好），取而代之 HDFS 尝试从流水线中摘除挂了的 DataNode 并恢复写入，这个过程称为 pipeline recovery。

3、Client 在写入过程中，NameNode 挂了

公众号：旧时光大数据

当 Client 在写入过程中，NameNode 挂了。这里的前提是已经开始写入了，所以 NameNode 已经完成了对 DataNode 的分配，若一开始 NameNode 就挂了，整个 HDFS 是不可用的所以也无法开始写入。流水线写入过程中，当一个 block 写完后需向 NameNode 报告其状态，这时 NameNode 挂了，状态报告失败，但不影响 DataNode 的流线工作，数据先被保存下来，但最后一步 Client 写完向 NameNode 请求关闭文件时会出错，由于 NameNode 的单点特性，所以无法自动恢复，需人工介入恢复。

NameNode存数据吗?

问过的一些公司：触宝(2021.07)

参考答案：

存储元数据（文件名、创建时间、大小、权限、文件与block块映射关系）

使用NameNode的好处

问过的一些公司：陌陌(2021.08)

参考答案：

1、维护目录树，维护命名空间。

2、负责确定指定的文件块到具体的Datanode结点的映射关系。（在客户端与Datanode之间共享数据）

3、管理Datanode结点的状态报告

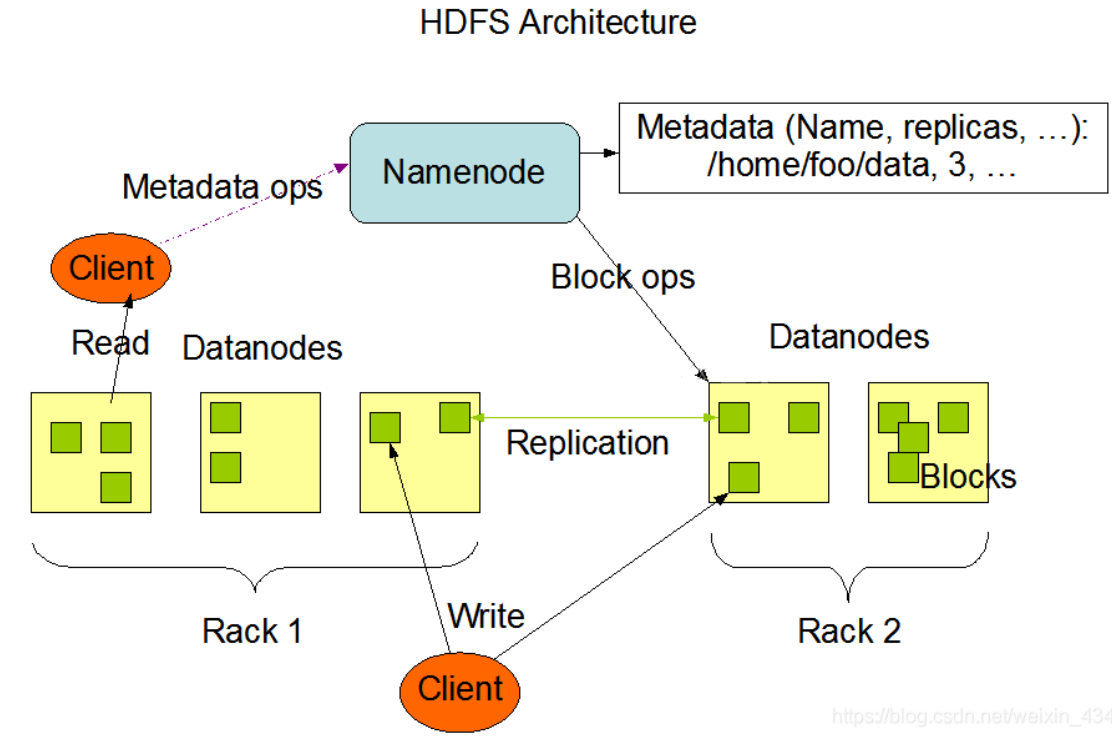
HDFS中DataNode怎么存储数据的

问过的一些公司：华为精英计划(2021.07)

参考答案：

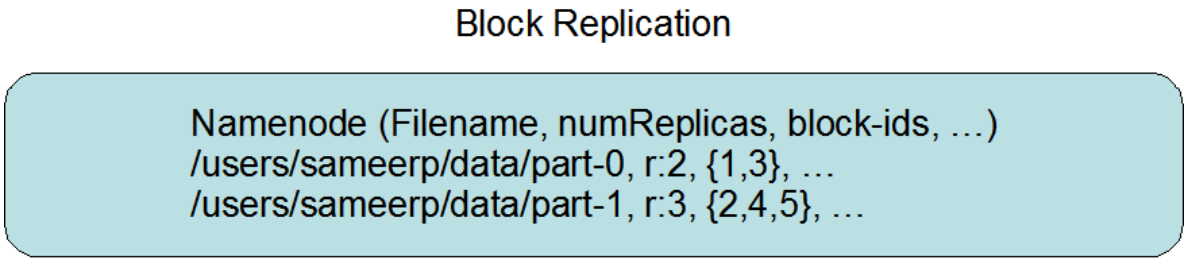
DataNode工作机制以及数据存储如下：

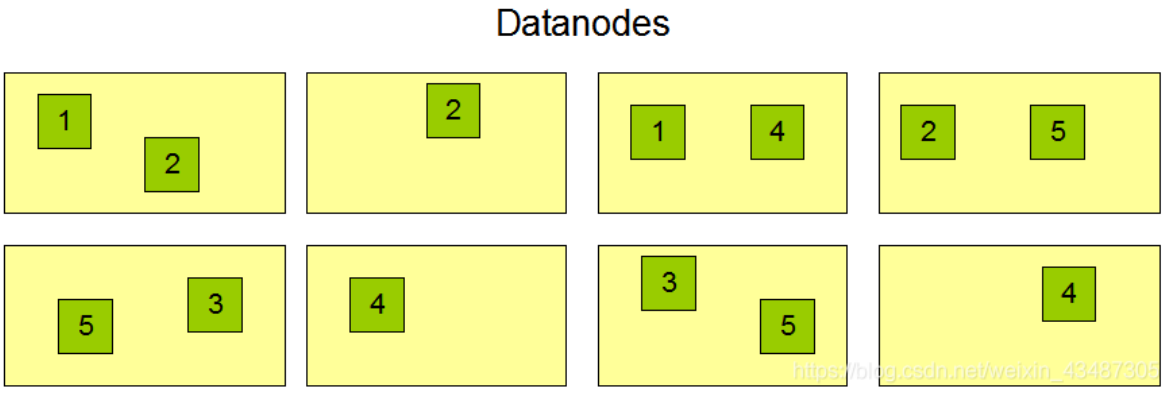
公众号：旧时光大数据



HDFS分布式文件系统也是一个主从架构，主节点是我们的NameNode，负责管理整个集群以及维护集群的元数据信息。

从节点DataNode，主要负责文件数据存储。





1、DataNode工作机制

1）**一个数据块在DataNode上以文件形式存储在磁盘上，包括两个文件，一个是数据本身，一个是元数据包括数据块的长度，块数据的校验和，以及时间戳**

2）DataNode启动后向NameNode注册，通过后，周期性（6小时）的向NameNode上报所有的块信息

3）心跳是每3秒一次，心跳返回结果带有NameNode给该DataNode的命令如复制块数据到另一台机器，或删除某个数据块。如果超过10分钟没有收到某个DataNode的心跳，则认为该节点不可用

4）集群运行中可以安全加入和退出一些机器

公众号：旧时光大数据

2、数据完整性

1）当DataNode读取block的时候，它会计算checksum

2）如果计算后的checksum，与block创建时值不一样，说明block已经损坏

3）client读取其他DataNode上的block

4）datanode在其文件创建后周期验证checksum

HDFS写流程中如果DataNode突然宕机了怎么办?

问过的一些公司：陌陌(2021.08)

参考答案：

客户端上传文件时与 DataNode 建立 pipeline 管道，管道正向是客户端向 DataNode 发送的数据包，管道反向是 DataNode 向客户端发送 ack 确认，也就是正确接收到数据包之后发送一个已确认接收到的应答，当 DataNode 突然挂掉了，客户端接收不到这个 DataNode 发送的 ack 确认，客户端会通知 NameNode，NameNode 检查该块的副本与规定的不符，NameNode 会通知 DataNode 去复制副本，并将挂掉的 DataNode 作下线处理，不再让它参与文件上传与下载。

直接将数据文件上传到HDFS的表目录中，如何在表中查询到该数据?

问过的一些公司：

参考答案：

在Hive中创建表并将HDFS中的数据导入Hive，然后再进行查询就行得到结果。

MapReduce部分

介绍下MapReduce

问过的一些公司：字节x2，字节(2021.09)，美团，美团(2021.08)，网易有道(2021.10)

回答技巧：结合MapReduce的优缺点回答（下一题）

参考答案：

MapReduce 是一个**分布式运算程序**的编程框架，它的核心功能是**将用户编写的业务逻辑代码和自带默认组件整合成一个完整的分布式运算程序**，并发运行在一个 Hadoop 集群上。

MapReduce的核心思想是将用户编写的逻辑代码和架构中的各个组件整合成一个分布式运算程序，实现一定程序的并行处理海量数据，提高效率。

海量数据难以在单机上处理，而一旦将单机版程序扩展到集群上进行分布式运行势必将大大增加程序的复杂程度。引入MapReduce架构，开发人员可以将精力集中于数据处理的核心业务逻辑上，而将分布式程序中的公共功能封装成框架,以降低开发的难度。

一个完整的mapreduce程序有三类实例进程

- MRAppMaster：负责整个程序的协调过程
- MapTask：负责map阶段的数据处理

公众号：旧时光大数据

- ReduceTask：负责reduce阶段的数据处理

MapReduce优缺点

问过的一些公司：小米

参考答案：

优点

1）MapReduce 易于编程

它简单的实现一些接口，就可以完成一个分布式程序，这个分布式程序可以分布到大量廉价的 PC 机器上运行。也就是说你写一个分布式程序，跟写一个简单的串行程序是一模一样的。就是因为这个特点使得 MapReduce 编程变得非常流行。

2）良好的扩展性

当你的计算资源不能得到满足的时候，你可以通过简单的增加机器来扩展它的计算能力。

3）高容错性

MapReduce 设计的初衷就是使程序能够部署在廉价的 PC 机器上，这就要求它具有很高的容错性。比如其中一台机器挂了，它可以把上面的计算任务转移到另外一个节点上运行， 不至于这个任务运行失败，而且这个过程不需要人工参与，而完全是由 Hadoop 内部完成的。

4）适合 PB 级以上海量数据的离线处理

可以实现上千台服务器集群并发工作，提供数据处理能力。

缺点

1）不擅长实时计算

MapReduce无法像MySQL一样，在毫秒或者秒级内返回结果。

2）不擅长流式计算

流式计算的输入数据是动态的，而MapReduce的输入数据集是静态的，不能动态变化。这是因为 MapReduce 自身的设计特点决定了数据源必须是静态的。

3）不擅长 DAG（有向无环图）计算

多个应用程序存在依赖关系，后一个应用程序的输入为前一个的输出。在这种情况下，MapReduce并不是不能做，而是使用后，每个MapReduce作业的输出结果都会写入到磁盘， 会造成大量的磁盘 IO，导致性能非常的低下。

MapReduce架构

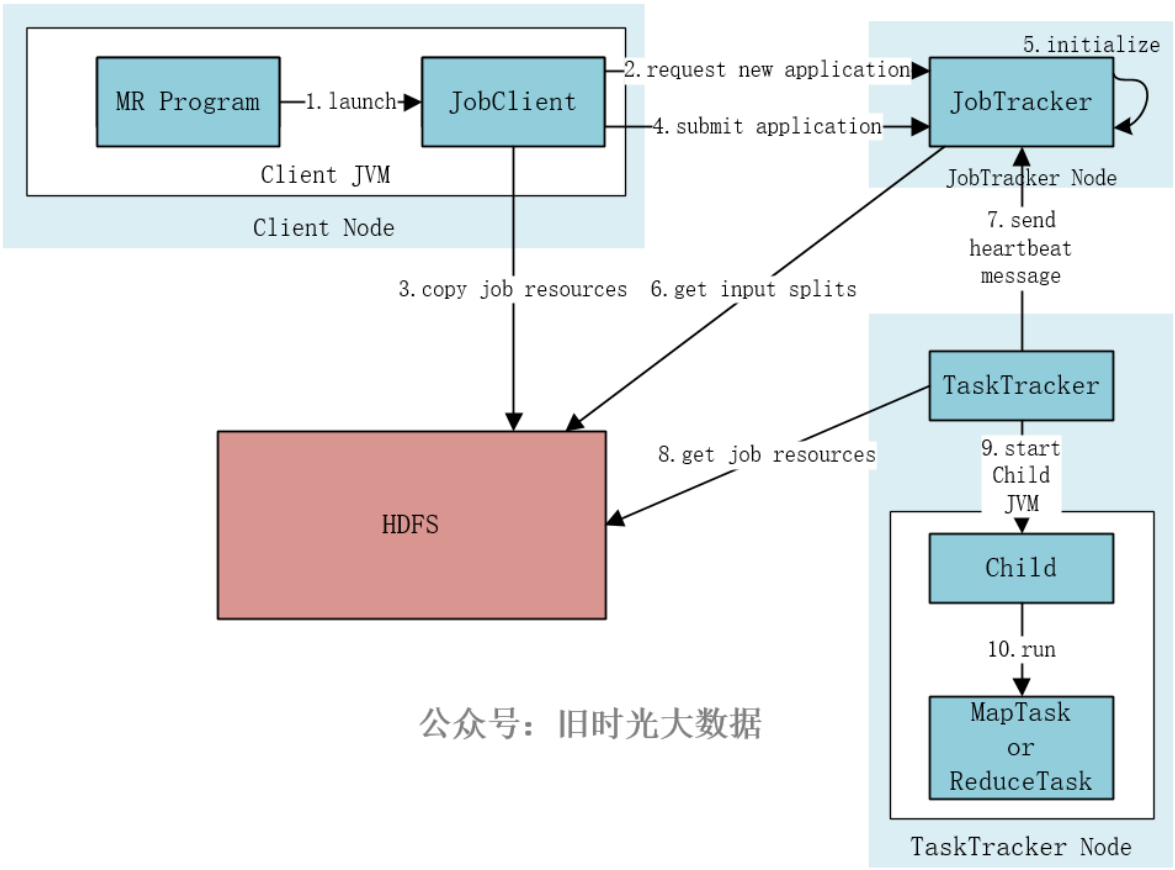
可回答：MapReduce的几部分

问过的一些公司：美团，唯品会(2021.10)

参考答案：

与HDFS一样，MapReduce也是采用Master/Slave的架构，其架构图如下所示。

公众号：旧时光大数据



MapReduce包含四个组成部分，分别为Client、JobTracker、TaskTracker和Task，下面我们详细介绍这四个组成部分。

1）`Client 客户端`

每一个 Job 都会在用户端通过 Client 类将应用程序以及配置参数 Configuration 打包成 JAR 文件存储在 HDFS，并把路径提交到 JobTracker 的 master 服务，然后由 master 创建每一个 Task（即 MapTask 和 ReduceTask） 将它们分发到各个 TaskTracker 服务中去执行。

2）`JobTracker`

JobTracke负责资源监控和作业调度。JobTracker 监控所有TaskTracker 与job的健康状况，一旦发现失败，就将相应的任务转移到其他节点；同时，JobTracker 会跟踪任务的执行进度、资源使用量等信息，并将这些信息告诉任务调度器，而调度器会在资源出现空闲时，选择合适的任务使用这些资源。在 Hadoop 中，任务调度器是一个可插拔的模块，用户可以根据自己的需要设计相应的调度器。

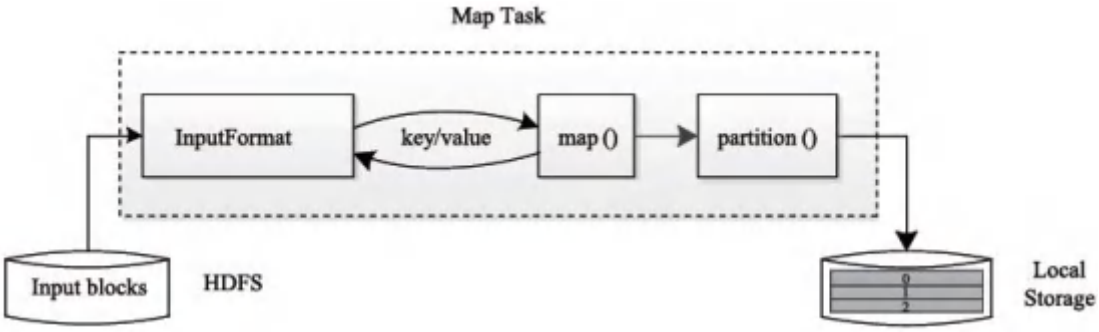
3）`TaskTracker`

TaskTracker 会周期性地通过Heartbeat 将本节点上资源的使用情况和任务的运行进度汇报给JobTracker，同时接收JobTracker 发送过来的命令并执行相应的操作（如启动新任务、杀死任务等）。TaskTracker 使用“slot”等量划分本节点上的资源量。“slot”代表计算资源（CPU、内存等）。一个Task 获取到一个slot 后才有机会运行，而Hadoop 调度器的作用就是将各个TaskTracker 上的空闲slot 分配给Task 使用。slot 分为Map slot 和Reduce slot 两种，分别供Map Task 和Reduce Task 使用。TaskTracker 通过slot 数目（可配置参数）限定Task 的并发度。

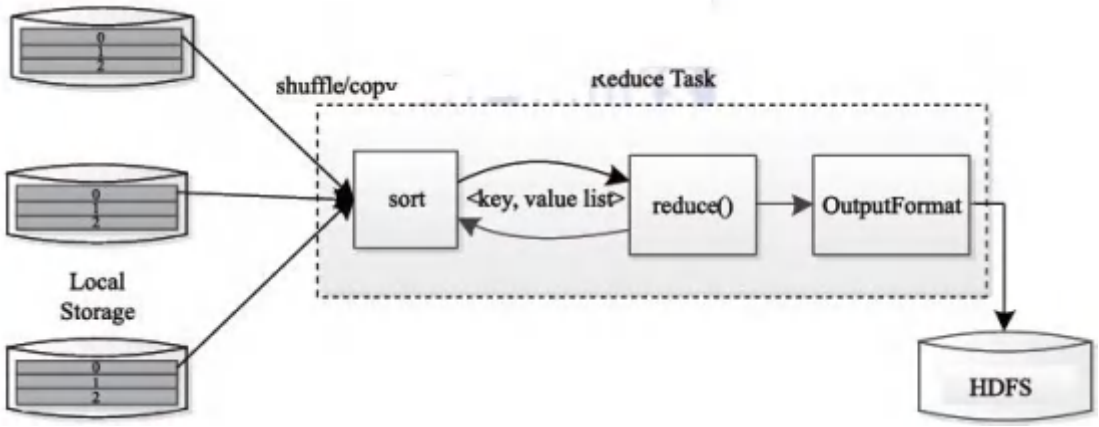
4）`Task`

Task 分为Map Task 和Reduce Task 两种，均由TaskTracker 启动。HDFS 以固定大小的block 为基本单位存储数据，而对于MapReduce 而言，其处理单位是split。

Map Task 执行过程如下图所示：由该图可知，Map Task 先将对应的split 迭代解析成一个个key/value 对，依次调用用户 自定义的map() 函数进行处理，最终将临时结果存放到本地磁盘上，其中临时数据被分成若干个partition，每个partition 将被一个Reduce Task 处理。



Reduce Task 执行过程下图所示。该过程分为三个阶段：



注意：

- 从远程节点上读取Map Task 中间结果（称为“Shuffle 阶段”）；
- 按照key对key/value 对进行排序（称为“Sort 阶段”）；
- 依次读取< key, value list>，调用用户自定义的reduce() 函数处理，并将最终结果存到HDFS 上（称为“Reduce 阶段”）。

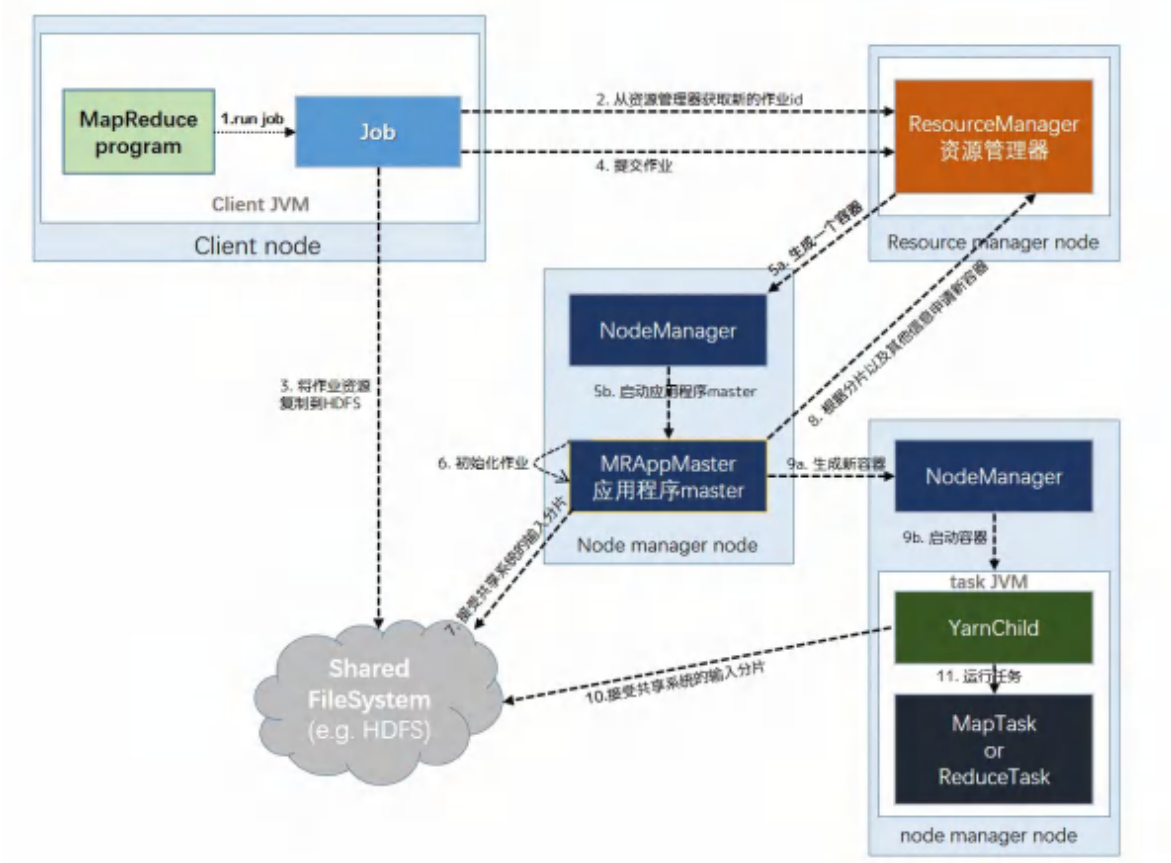
MapReduce工作原理

可回答：1）MapReduce（执行）流程；2）对MapReduce的理解；3）MapReduce过程；4）MapReduce的详细过程；5）MapReduce原理；6）MapReduce中有没有涉及到排序；7）MapReduce提交job到YARN的流程；8）MapReduce原理，map和reduce过程；9）说一下MapReduce流程

问过的一些公司：阿里×4，字节×7，字节(2021.08)x3-(2021.10)，头条，滴滴，百度，腾讯×4，Shopee，小米，爱奇艺，祖龙娱乐，360×5，商汤科技，网易×5，51×2，星环科技，招银网络，作业帮，映客直播，美团×16，美团(2021.09)x2，字节×2，有赞，58×3，华为x2，创略科技，米哈游，快手，快手(2021.09)，京东×4，趋势科技，海康威视，顺丰，好未来x3，一点资讯，冠群驰骋，中信信用卡中心，金山云，米哈游，途牛，大华(2021.07)，京东(2021.07)，Shopee(2021.08)，多益(2021.09)，荣耀(2021.09)，百度(2021.08、2021.09)，阿里蚂蚁(2021.08)，携程(2021.09)，虎牙(2021.08)，重庆富民银行(2021.09)，网易有道(2021.09)，携程(2021.09)，陌陌(2021.10)，腾讯(2022.03)

参考答案：

版本一



1、提交作业

Client提交Job：

- Client编写好Job后，调用Job实例的Sumit() 或者 waitForCompletion() 方法提交作业；
- 从RM（而不是Jobtracker）获取新的作业ID，在YARN命名法中它是一个Application ID（步骤2）。

Job提交到RM：

- Client检查作业的输出说明，计算输入分片，并将作业资源（包括作业JAR、配置和分片信息）复制到HDFS（步骤3）；
- 调用RM的 submitApplication() 方法提交作业（步骤4）。

2、作业初始化

给作业分配ApplicationMaster：

- RM收到调用它的 submitApplication() 消息后，便将请求传递给 scheduler （调度器）；
- scheduler分配一个 Container，然后 RM在该 NM的管理下在 Container中启动 ApplicationMaster（步骤5a & 5b）。

ApplicationMaster初始化作业：

- MR作业的ApplicationMaster 是一个Java应用程序，它的主类是 MRAppMaster。它对作业进行初始化：通过创建多个薄记对象以保持对作业进度的跟踪，因为它将接受来自任务的进度和完成报告（步骤6）；
- ApplicationMaster 从HDFS中获取 在Client 计算的输入分片（map、reduce任务数）（步骤7）【对每一个分片创建一个 map 任务对象以及由 mapreduce.job.reduces 属性确定的多个 reduce 任务对象】。

注意：

公众号：旧时光大数据

ApplicationMaster决定如何运行构成 MapReduce 作业的各个任务。如果作业很小，就选择在与它同一个JVM上运行。

3、任务分配

- ApplicationMaster 为该作业中的所有 map 任务和 reduce 任务向 RM 请求 Container （步骤8）；【随着心跳信息的请求包括每个map任务的数据本地化信息，特别是输入分片所在的主机和相应机架信息。理想情况下，它将任务分配到数据本地化的节点，但如果不可能这么做，就会相对于本地化的分配优先使用机架本地化的分配】

注意：

请求也为任务指定了内存需求。在默认情况下， map任务和reduce任务都分配到 1024MB 的内存，但这可以通过 mapreduce.map.memory.mb 和 mapreduce.reduce.memory.mb来设置。

4、任务执行

- 一旦 RM 的 scheduler 为任务分配了 Container， ApplicationMaster就通过与 NM通信来启动 Container （步骤9a & 9b）；
- 该任务由主类为 YardChild 的Java应用程序执行。在它运行任务之前，首先将任务需要的资源本地化（包括作业的配置、JAR文件和所有来自分布式缓存的文件）（步骤10）；
- 最后，运行 map 任务或 reduce 任务（步骤11）。

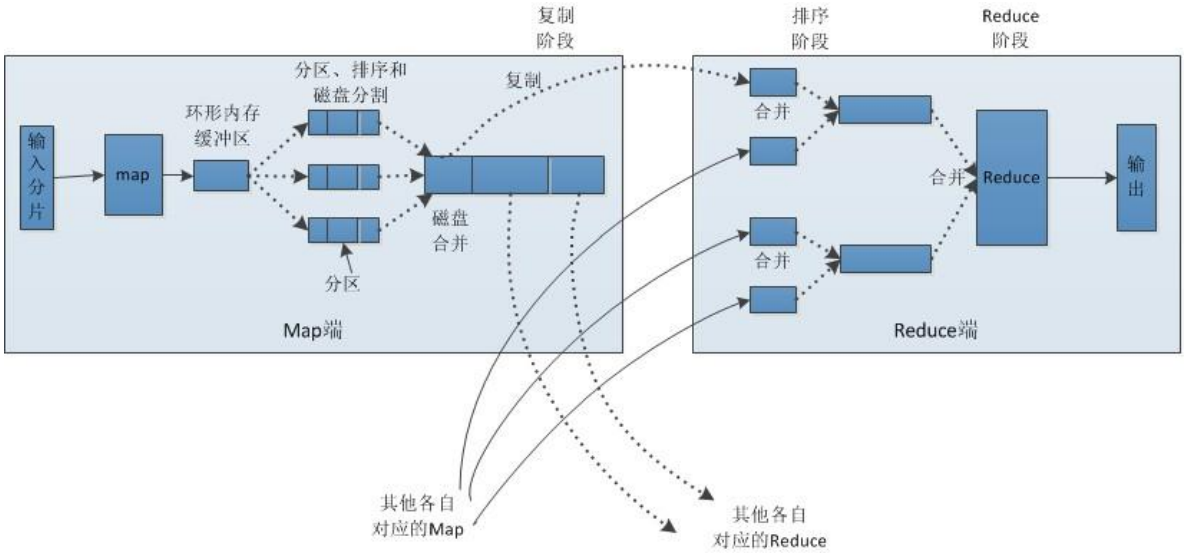
5、进度和状态的更新

- 在YARN下运行，任务每 3s通过 umbilical 接口向 ApplicationMaster 汇报进度和状态（包括计数器），作为作业的汇聚试图（aggregate view）。

6、作业完成

- 除了向 ApplicationMaster 查询进度外，Client 每 5s还通过调用 Job 的 waitForCompletion() 来检查作业是否完成【查询的间隔可以通过 mapreduce.client.completion.pollinterval 属性进行设置】。
- 作业完成后， ApplicationMaster 和任务容器清理其工作状态， OutputCommitter 的作业清理方法会被调用。作业历史服务器保存作业的信息供用户需要时查询。

版本二：Map、Reduce任务中Shuffle和排序的过程



Map端

每个输入分片会让一个map任务来处理，默认情况下，以HDFS的一个块的大小（默认为64M）为一个分片，当然我们也可以设置块的大小。map输出的结果会暂且放在一个环形内存缓冲区中（该缓冲区的大小默认为100M，由io.sort.mb属性控制），当该缓冲区快要溢出时（默认为缓冲区大小的80%，由io.sort.spill.percent属性控制），会在本地文件系统中创建一个溢出文件，将该缓冲区中的数据写入这个文件。

在写入磁盘之前，线程首先根据reduce任务的数目将数据划分为相同数目的分区，也就是一个reduce任务对应一个分区的数据。这样做是为了避免有些reduce任务分配到大量数据，而有些reduce任务却分到很少数据，甚至没有分到数据的尴尬局面。其实分区就是对数据进行hash的过程。然后对每个分区中的数据进行排序，如果此时设置了Combiner，将排序后的结果进行Combia操作，这样做的目的是让尽可能少的数据写入到磁盘。

当map任务输出最后一个记录时，可能会有很多的溢出文件，这时需要将这些文件合并。合并的过程中会不断地进行排序和combia操作，目的有两个：1.尽量减少每次写入磁盘的数据量；2.尽量减少下一复制阶段网络传输的数据量。最后合并成了一个已分区且已排序的文件。为了减少网络传输的数据量，这里可以将数据压缩，只要将mapred.compress.map.out设置为true就可以了。

将分区中的数据拷贝给相对应的reduce任务。有人可能会问：分区中的数据怎么知道它对应的reduce是哪个呢？其实map任务一直和其父TaskTracker保持联系，而TaskTracker又一直和JobTracker保持心跳。所以JobTracker中保存了整个集群中的宏观信息。只要reduce任务向JobTracker获取对应的map输出位置就ok了哦。

到这里，map端就分析完了。那到底什么是Shuffle呢？Shuffle的中文意思是“洗牌”，如果我们这样看：一个map产生的数据，结果通过hash过程分区却分配给了不同的reduce任务，是不是一个对数据洗牌的过程呢？呵呵。

Reduce端

Reduce会接收到不同map任务传来的数据，并且每个map传来的数据都是有序的。如果reduce端接受的数据量相当小，则直接存储在内存中（缓冲区大小由mapred.job.shuffle.input.buffer.percent属性控制，表示用作此用途的堆空间的百分比），如果数据量超过了该缓冲区大小的一定比例（由mapred.job.shuffle.merge.percent决定），则对数据合并后溢写到磁盘中。

随着溢写文件的增多，后台线程会将它们合并成一个更大的有序的文件，这样做是为了给后面的合并节省时间。其实不管在map端还是reduce端，MapReduce都是反复地执行排序，合并操作，现在终于明白了有些人为什么会说：排序是hadoop的灵魂。

合并的过程中会产生许多的中间文件（写入磁盘了），但MapReduce会让写入磁盘的数据尽可能地少，并且最后一次合并的结果并没有写入磁盘，而是直接输入到reduce函数。

MapReduce哪个阶段最费时间

问过的一些公司：阿里

参考答案：

Shuffle阶段

MapReduce中的Combine是干嘛的？有什么好处？

问过的一些公司：字节

参考答案：

在MapReduce中，Combine 阶段是当Map阶段所有数据处理完成后，MapTask对所有临时文件进行一次合并，以确保最终只会生成一个数据文件。

让每个MapTask最终只生成一个数据文件，可避免同时打开大量文件和同时读取大量小文件产生的随机读取带来的开销。

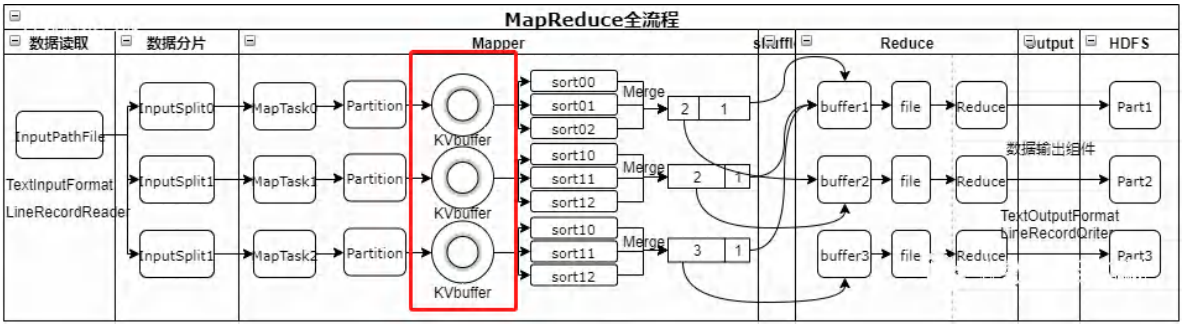
Combiner能够应用的前提是不能影响最终的业务逻辑，而且，Combiner的输出k-v应该跟reducer的输入k-v类型要对应起来。

MapReduce为什么一定要有环型缓冲区

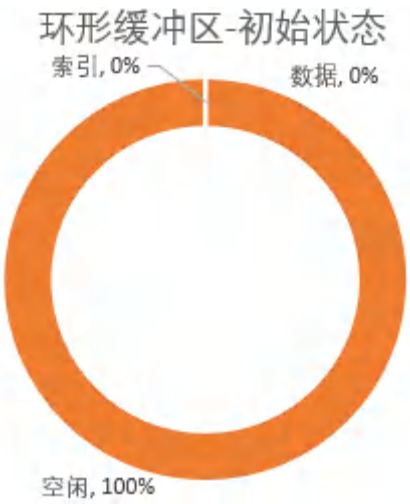
问过的一些公司：字节

参考答案：

在MapReduce的流程中，环形缓冲区是在溢写到磁盘之前的操作



我们先来剖析下环形缓冲区

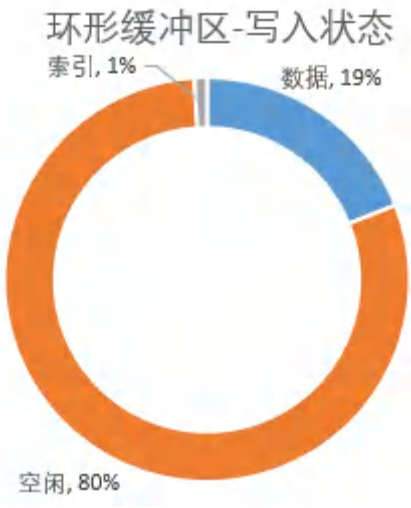


环形缓冲区分为三块，空闲区、数据区、索引区。初始位置取名叫做“赤道”，就是圆环上的白线那个位置。初始状态的时候，数据和索引都为0，所有空间都是空闲状态。

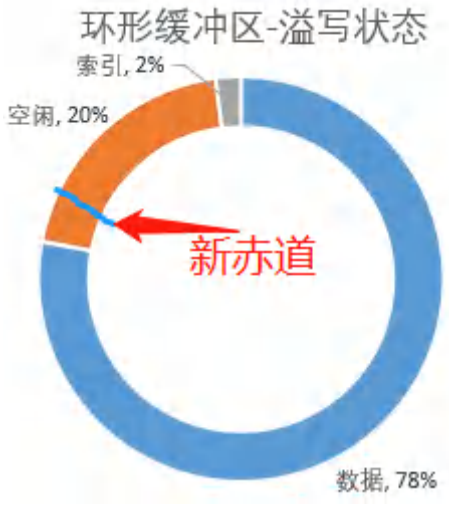
tips：这里有一个调优参数，可以设置环形缓冲区的大小：

mapreduce.task.io.sort.mb，默认100M，可以稍微设置大一些，但不要太大，因为每个spilt就128M。

公众号：旧时光大数据



环形缓冲区写入的时候，有个细节：数据是从赤道的右边开始写入，索引（每次申请4kb）是从赤道是左边开始写。这个设计很有意思，这样两个文件各是各的，互不干涉。



在数据和索引的大小到了mapreduce.map.sort.spill.percent参数设置的比例时（默认80%，这个是调优的参数），会有两个动作：

1、对写入的数据进行原地排序，并把排序好的数据和索引spill到磁盘上去；

2、在空闲的20%区域中，重新算一个新的赤道，然后在新赤道的右边写入数据，左边写入索引；

3、当20%写满了，但是上一次80%的数据还没写到磁盘的时候，程序就会panding一下，等80%空间腾出来之后再继续写。

如此循环往复，永不停歇，直到所有任务全部结束。整个操作都在内存，形状像一个环，所以才叫环形缓冲区。

为什么要有环形缓冲区？

主要有以下几点

1、使用环形缓冲区，便于写入缓冲区和写出缓冲区同时进行。

2、为了防止阻塞，所以环型缓冲区不会等缓冲区满了再spill

3、每个Map任务不断地将键值对输出到在内存中构造的一个环形数据结构中，**使用环形数据结构是为了更有效地使用内存空间，在内存中放置尽可能多的数据。**

4、环形缓冲区不需要重新申请新的内存，始终用的都是这个内存空间。大家知道MR是用java写的，而Java有一个最讨厌的机制就是Full GC。Full GC总是会出来捣乱，这个bug也非常隐蔽，发现了也不好处理。环形缓冲区从头到尾都在用那一个内存，不断重复利用，因此完美的规避了Full GC导致的各种问题，同时也规避了频繁申请内存引发的其他问题。

公众号：旧时光大数据

5、环形缓冲区同时做了两件事情：1、排序；2、索引。在这里一次排序，将无序的数据变为有序，写磁盘的时候顺序写，读数据的时候顺序读，效率高非常多！

在这里设置索引区也是为了能够持续的处理任务。每读取一段数据，就往索引文件里也写一段，这样在排序的时候能加快速度。

MapReduce为什么一定要有Shuffle过程

问过的一些公司：百度，头条，字节(2021.09)

参考答案：

MapReduce计算模型一般包括两个重要的阶段：Map是映射，负责数据的过滤分发；Reduce是规约，负责数据的计算归并。Reduce的数据来源于Map，Map的输出即是Reduce的输入，**Reduce需要通过Shuffle来获取数据。**

从Map输出到Reduce输入的整个过程可以广义地称为Shuffle。**Shuffle横跨Map端和Reduce端，在Map端包括Spill过程，在Reduce端包括copy和sort过程。**

MapReduce的Shuffle过程及其优化

可回答：1）Hadoop的Shuffle过程；2）为什么Map端输出的时候需要排序？不排序直接输出难道不好吗？3）介绍下MapReduce的Shuffle机制；3）你觉得MapReduce有哪些需要优化的地方；4）哪些操作引起shuffle；5）说一下shuffle机制；6）说一下reduce的shuffle？7）shuffle流程的细节是什么

问过的一些公司：字节×4，字节(2021.07)(2021.08)，头条，百度×2，第四范式，360，猿辅导，美团×3，美团(2021.08)(2021.09)，平安，创略科技，网易x2，多益，顺丰，转转，抖音，一点咨询，作业帮×2，东方头条，大华(2021.07)，字节(2021.08)，荣耀(2021.09)x2，贝壳(2021.08)，蔚来(2021.09)

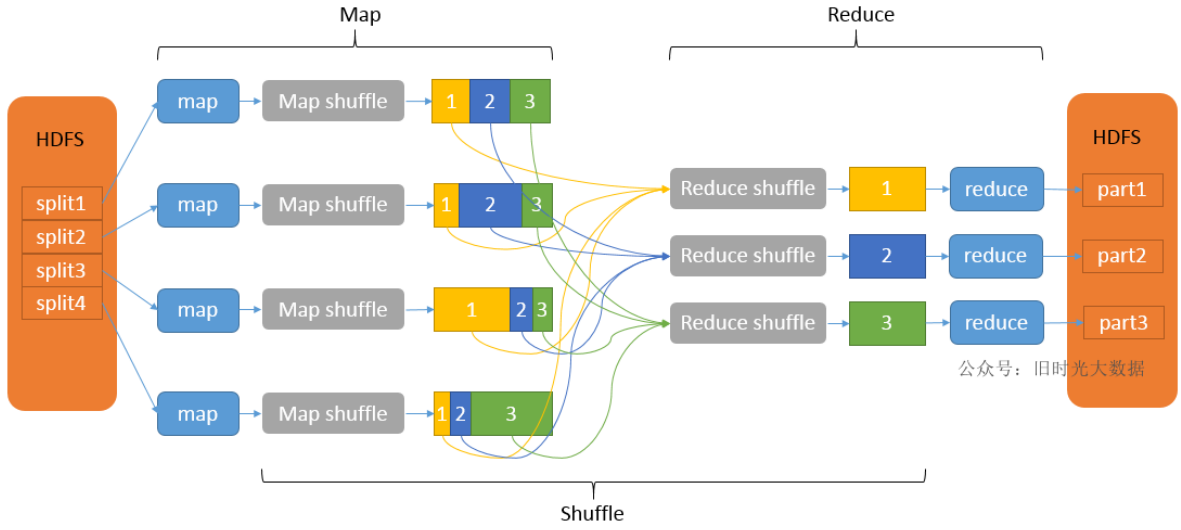
参考答案：

Shuffle的本义是洗牌、混洗，把一组有一定规则的数据尽量转换成一组无规则的数据，越随机越好。MapReduce中的Shuffle更像是洗牌的逆过程，把一组无规则的数据尽量转换成一组具有一定规则的数据。

为什么MapReduce计算模型需要Shuffle过程？我们都知道MapReduce计算模型一般包括两个重要的阶段：Map是映射，负责数据的过滤分发；Reduce是规约，负责数据的计算归并。Reduce的数据来源于Map，Map的输出即是Reduce的输入，Reduce需要通过Shuffle来获取数据。

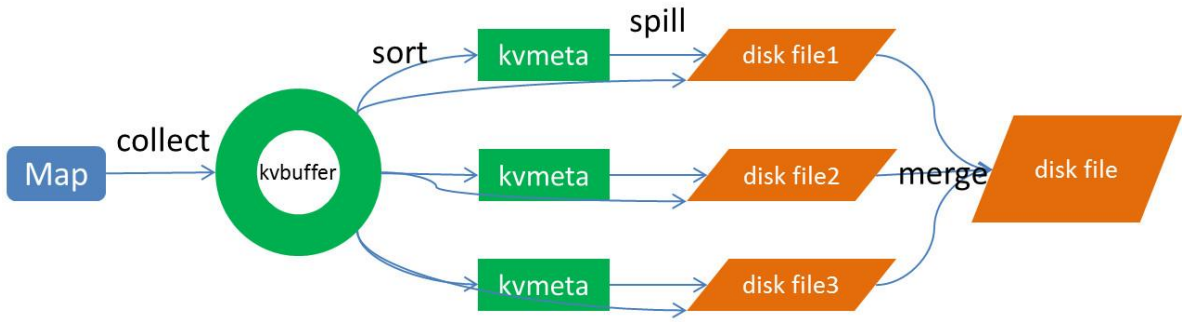
从Map输出到Reduce输入的整个过程可以广义地称为Shuffle。Shuffle横跨Map端和Reduce端，**在Map端包括Spill过程，在Reduce端包括copy和sort过程**，如图所示：

公众号：旧时光大数据



1、Spill过程

Spill过程包括输出、排序、溢写、合并等步骤，如图所示：

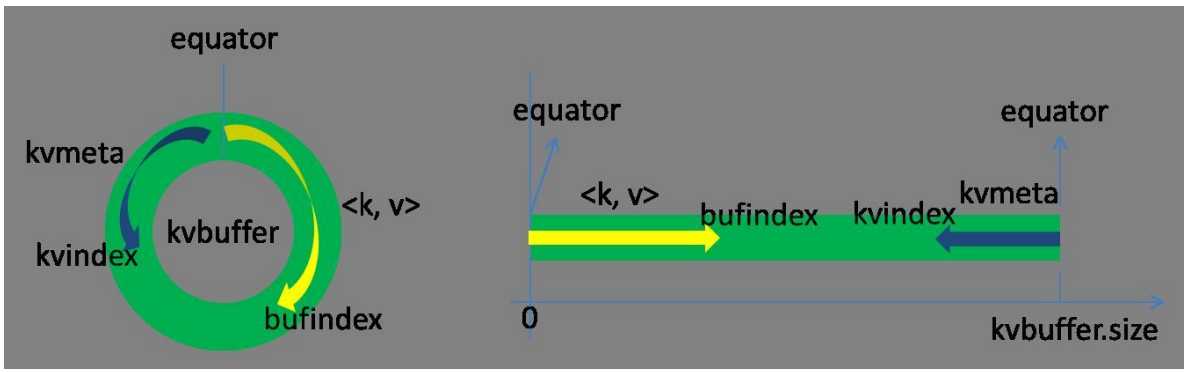


collect过程

每个Map任务不断地以对的形式把数据输出到在内存中构造的一个环形数据结构中。使用环形数据结构是为了更有效地使用内存空间，在内存中放置尽可能多的数据。

注意：关于环形缓冲区，也可以看看前面的题目解答

这个数据结构其实就是个字节数组，叫Kvbuffer，名如其义，但是这里面不光放置了数据，还放置了一些索引数据，给放置索引数据的区域起了一个Kvmeta的别名，在Kvbuffer的一块区域上穿了一个IntBuffer（字节序采用的是平台自身的字节序）的马甲。数据区域和索引数据区域在Kvbuffer中是相邻不重叠的两个区域，用一个分界点来划分两者，分界点不是亘古不变的，而是每次 Spill之后都会更新一次。初始的分界点是0，数据的存储方向是向上增长，索引数据的存储方向是向下增长，如图所示：



Kvbuffer的存放指针bufindex是一直闷着头地向上增长，比如bufindex初始值为0，一个Int型的key写完之后，bufindex增长为4，一个Int型的value写完之后，bufindex增长为8。

公众号：旧时光大数据

索引是对在kvbuffer中的索引，是个四元组，包括：value的起始位置、key的起始位置、partition值、value的长度， 占用四个Int长度，Kvmeta的存放指针Kvindex每次都是向下跳四个“格子”，然后再向上一个格子一个格子地填充四元组的数据。比如 Kvindex初始位置是-4，当第一个写完之后，(Kvindex+0)的位置存放value的起始位置、(Kvindex+1)的位置存放key的起始位置、 (Kvindex+2)的位置存放partition的值、(Kvindex+3)的位置存放value的长度，然后Kvindex跳到-8位置，等第二 个和索引写完之后，Kvindex跳到-32位置。

Kvbuffer的大小虽然可以通过参数设置，但是总共就那么大，和索引不断地增加，加着加着，Kvbuffer总有不够用的那天，那怎么办？把数据从内存刷到磁盘上再接着往内存写数据，把 Kvbuffer中的数据刷到磁盘上的过程就叫Spill，多么明了的叫法，内存中的数据满了就自动地spill到具有更大空间的磁盘。

关于Spill 触发的条件，也就是Kvbuffer用到什么程度开始Spill，还是要讲究一下的。如果把Kvbuffer用得死死得，一点缝都不剩的时候再开始 Spill，那Map任务就需要等Spill完成腾出空间之后才能继续写数据；如果Kvbuffer只是满到一定程度，比如80%的时候就开始 Spill，那在Spill的同时，Map任务还能继续写数据，如果Spill够快，Map可能都不需要为空闲空间而发愁。两利相衡取其大，一般选择后 者。

Spill这个重要的过程是由Spill线程承担，Spill线程从Map任务接到“命令”之后就开始正式干活，干的活叫SortAndSpill，原来不仅仅是Spill，在Spill之前还有个颇具争议性的Sort。

sort过程

先把Kvbuffer中的数据按照partition值和key两个关键字升序排序，移动的只是索引数据，排序结果是Kvmeta中数据按照partition为单位聚集在一起，同一partition内的按照key有序。

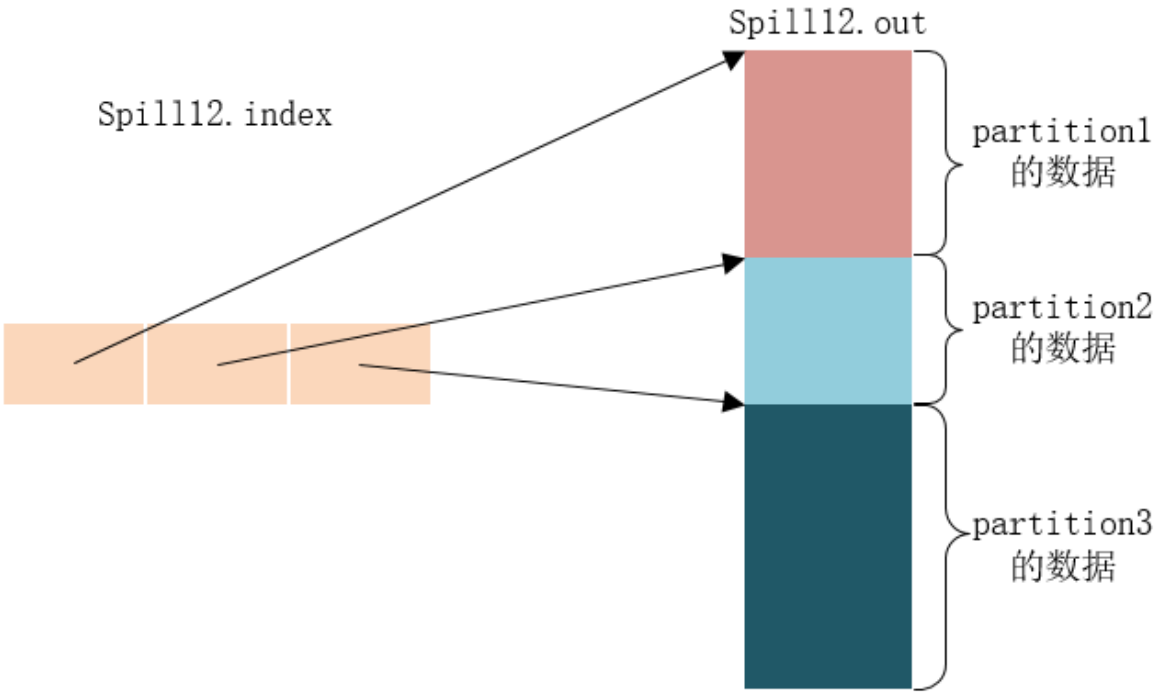
Spill过程

Spill线程为这次Spill过程创建一个磁盘文件：从所有的本地目录中轮训查找能存储这么大空间的目录，找到之后在其中创建一个类似于 “spill12.out”的文件。Spill线程根据排过序的Kvmeta挨个partition的把数据吐到这个文件中，一个partition对应的数据吐完之后顺序地吐下个partition，直到把所有的partition遍历完。一个partition在文件中对应的数据也叫段(segment)。

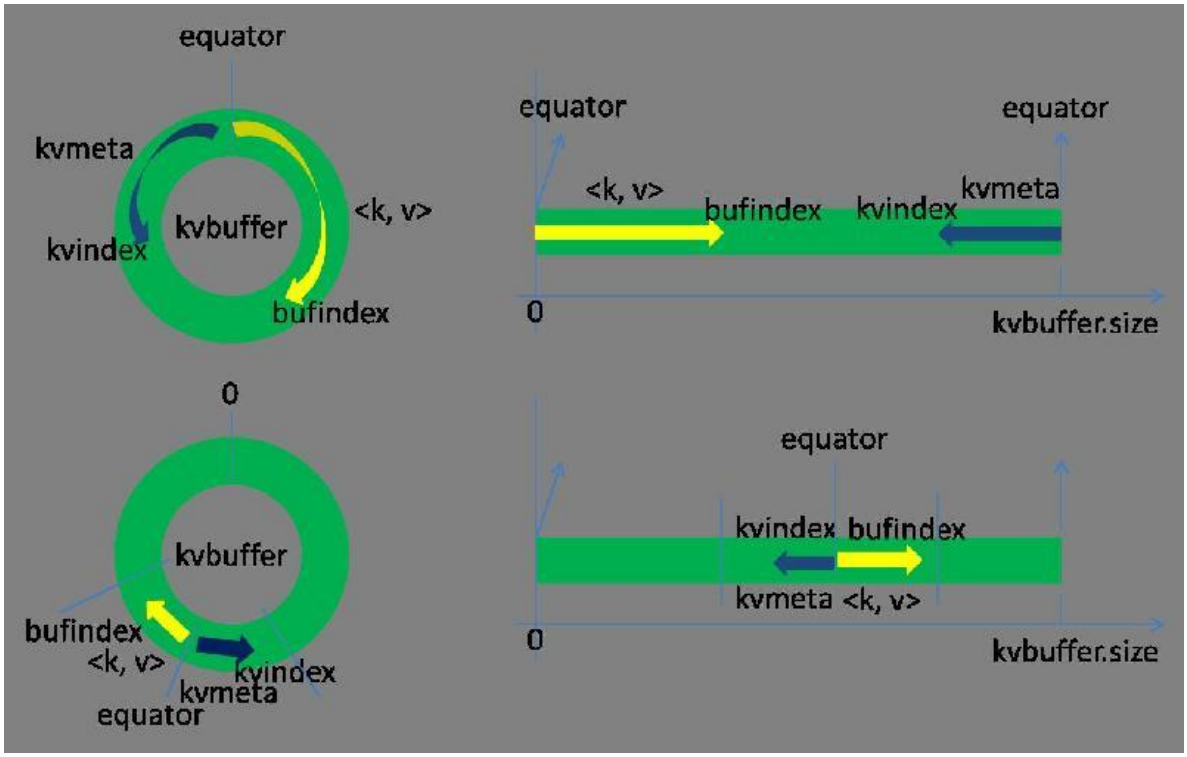
所有的partition对应的数据都放在这个文件里，虽然是顺序存放的，但是怎么直接知道某个partition在这个文件中存放的起始位置呢？强大的索引又出场了。有一个三元组记录某个partition对应的数据在这个文件中的索引：起始位置、原始数据长度、压缩之后的数据长度，一个partition对应一个三元组。然后把这些索引信息存放在内存中，如果内存中放不下了，后续的索引信息就需要写到磁盘文件中了：从所有的本地目录中轮训查找能存储这么大空间的目录，找到之后在其中创建一个类似于“spill12.out.index”的文件，文件中不光存储了索引数据，还存储了crc32的校验数据。（spill12.out.index不一定在磁盘上创建，如果内存（默认1M空间）中能放得下就放在内存中，即使在磁盘上创建了，和 spill12.out文件也不一定在同一个目录下。）

每一次Spill过程就会最少生成一个out文件，有时还会生成index文件，Spill的次数也烙印在文件名中。索引文件和数据文件的对应关系如下图所示：

公众号：旧时光大数据



在Spill线程如火如荼的进行SortAndSpill工作的同时，Map任务不会因此而停歇，而是一无既往地进行着数据输出。Map还是把数据写到kvbuffer中，那问题就来了：只顾着闷头按照bufindex指针向上增长，kvmeta只顾着按照Kvindex向下增长，是保持指针起始位置不变继续跑呢，还是另谋它路？如果保持指针起始位置不变，很快bufindex和Kvindex就碰头了，碰头之后再重新开始或者移动内存都比较麻烦，不可取。Map取 kvbuffer中剩余空间的中间位置，用这个位置设置为新的分界点，bufindex指针移动到这个分界点，Kvindex移动到这个分界点的-16位置，然后两者就可以和谐地按照自己既定的轨迹放置数据了，当Spill完成，空间腾出之后，不需要做任何改动继续前进。分界点的转换如下图所示：



Map任务总要把输出的数据写到磁盘上，即使输出数据量很小在内存中全部能装得下，在最后也会把数据刷到磁盘上。

merge过程

Map任务如果输出数据量很大，可能会进行好几次Spill，out文件和Index文件会产生很多，分布在不同的磁盘上。最后把这些文件进行合并的merge过程闪亮登场。

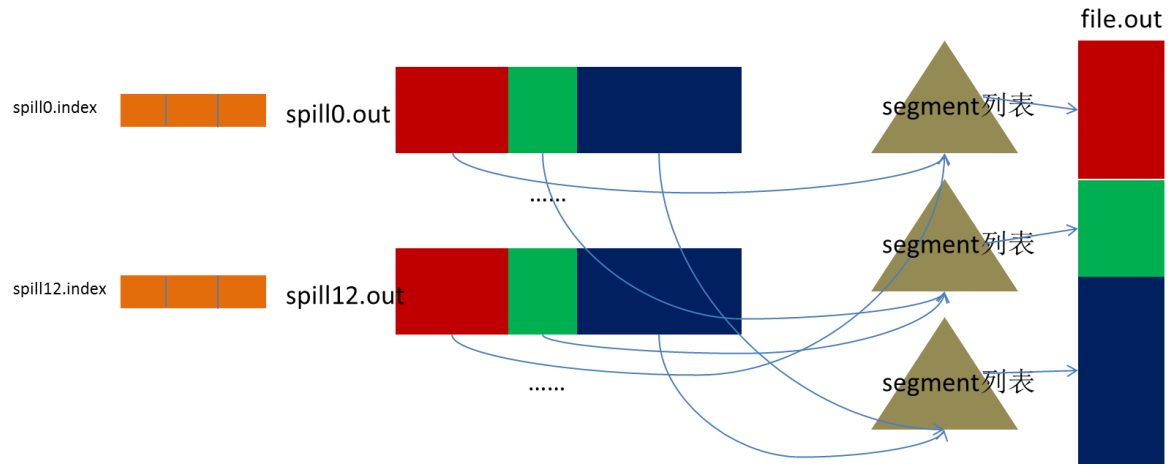
Merge过程怎么知道产生的Spill文件都在哪了呢？从所有的本地目录上扫描得到产生的Spill文件，然后把路径存储在一个数组里。Merge过程又怎么知道Spill的索引信息呢？没错，也是从所有的本地目录上扫描得到Index文件，然后把索引信息存储在一个列表里。到这里，又遇到了一个值得纳闷的地方。 在之前Spill过程中的时候为什么不直接把这些信息存储在内存中呢，何必又多了这步扫描的操作？特别是Spill的索引数据，之前当内存超限之后就把数 据写到磁盘，现在又要从磁盘把这些数据读出来，还是需要装到更多的内存中。之所以多此一举，是因为这时kvbuffer这个内存大户已经不再使用可以回 收，有内存空间来装这些数据了。（对于内存空间较大的土豪来说，用内存来省却这两个io步骤还是值得考虑的。）

然后为merge过程创建一个叫file.out的文件和一个叫file.out.Index的文件用来存储最终的输出和索引。

一 个partition一个partition的进行合并输出。对于某个partition来说，从索引列表中查询这个partition对应的所有索引信 息，每个对应一个段插入到段列表中。也就是这个partition对应一个段列表，记录所有的Spill文件中对应的这个partition那段数据的文 件名、起始位置、长度等等。

然后对这个partition对应的所有的segment进行合并，目标是合并成一个segment。当这个 partition对应很多个segment时，会分批地进行合并：先从segment列表中把第一批取出来，以key为关键字放置成最小堆，然后从最小 堆中每次取出最小的输出到一个临时文件中，这样就把这一批段合并成一个临时的段，把它加回到segment列表中；再从segment列表中把第二批取出 来合并输出到一个临时segment，把其加入到列表中；这样往复执行，直到剩下的段是一批，输出到最终的文件中。

最终的索引数据仍然输出到Index文件中。



Map端的Shuffle过程到此结束。

2、copy过程

Reduce任务通过HTTP向各个Map任务拖取它所需要的数据。每个节点都会启动一个常驻的HTTP server，其中一项服务就是响应Reduce拖取Map数据。当有MapOutput的HTTP请求过来的时候，HTTP server就读取相应的Map输出文件中对应这个Reduce部分的数据通过网络流输出给Reduce。

Reduce任务拖取某个Map对应的数据，如果在内存中能放得下这次数据的话就直接把数据写到内存中。Reduce要向每个Map去拖取数据，在内存中每个Map对应一块数据，当内存中存储的Map数据占用空间达到一定程度的时候，开始启动内存中merge，把内存中的数据merge输出到磁盘上一个文件中。

如果在内存中不能放得下这个Map的数据的话，直接把Map数据写到磁盘上，在本地目录创建一个文件，从HTTP流中读取数据然后写到磁盘，使用的缓存区大小是64K。拖一个Map数据过来就会创建一个文件，当文件数量达到一定阈值时，开始启动磁盘文件merge，把这些文件合并输出到一个文件。

有些Map的数据较小是可以放在内存中的，有些Map的数据较大需要放在磁盘上，这样最后Reduce任务拖过来的数据有些放在内存中了有些放在磁盘上，最后会对这些来一个全局合并。

3、merge sort过程

这里使用的Merge和Map端使用的Merge过程一样。Map的输出数据已经是有序的，Merge进行一次合并排序，所谓Reduce端的sort过程就是这个合并的过程。一般Reduce是一边copy一边sort，即copy和sort两个阶段是重叠而不是完全分开的。

Reduce端的Shuffle过程至此结束。

MapReduce Shuffle后续优化方向

- 压缩：对数据进行压缩，减少写读数据量；
- 减少不必要的排序：并不是所有类型的Reduce需要的数据都是需要排序的，排序这个nb的过程如果不需要最好还是不要的好；
- 内存化：Shuffle的数据不放在磁盘而是尽量放在内存中，除非逼不得已往磁盘上放；当然了如果有性能和内存相当的第三方存储系统，那放在第三方存储系统上也是很好的；这个是个大招；
- 网络框架：netty的性能据说要占优了；
- 本节点上的数据不走网络框架：对于本节点上的Map输出，Reduce直接去读吧，不需要绕道网络框架。

Reduce怎么知道去哪里拉Map结果集?

问过的一些公司：携程(2021.09)，美团(2021.09)

参考答案：

每个节点都会启动一个常驻的HTTP server，其中一项服务就是响应Reduce拖取Map数据。Reduce任务通过HTTP向各个Map任务拖取它所需要的数据。当有MapOutput的HTTP请求过来的时候，HTTP server就读取相应的Map输出文件中对应这个Reduce部分的数据通过网络流输出给Reduce。

Reduce阶段都发生了什么，有没有进行分组

问过的一些公司：陌陌(2021.10)

参考答案：

1）Copy阶段：ReduceTask从各个MapTask上远程拷贝一片数据，并针对某一片数据，如果其大小超过一定阈值，则写到磁盘上，否则直接放到内存中。

2）Merge阶段：在远程拷贝数据的同时，ReduceTask启动了两个后台线程对内存和磁盘上的文件进行合并，以防止内存使用过多或磁盘上文件过多。

3）Sort阶段：按照MapReduce语义，用户编写reduce()函数输入数据是按key进行聚集的一组数据。为了将key相同的数据聚在一起，Hadoop采用了基于排序的策略。由于各个MapTask已经实现对自己的处理结果进行了局部排序，因此，ReduceTask只需对所有数据进行一次归并排序即可。

4）Reduce阶段：reduce()函数将计算结果写到HDFS上。

在ReduceTask的Shuffle阶段，会获取到比较器，将从Map获取到的数据中key相同的分为同一组

公众号：旧时光大数据

```
public RawComparator getOutputValueGroupingComparator() {
    // 从配置中读取mapreduce.job.output.group.comparator.class，
    Class<? extends RawComparator> theClass =
getClass(JobContext.GROUP_COMPARATOR_CLASS, null, RawComparator.class);
    //如果没有设置，默认使用MapTask对key排序时，key的比较器
    if (theClass == null) {
      return getOutputKeyComparator();
    }
    // 否则用户设置了，就使用用户自定义的比较器
    return ReflectionUtils.newInstance(theClass, this);
  }

```

MapReduce Shuffle的排序算法

可回答：shuffle时会对数据进行排序吗

问过的一些公司：字节(2021.08)

参考答案：

1、排序的简单介绍

排序是MapReduce框架中最重要的操作之一。

MapTask和ReduceTask均会对数据**按照key进行排序**。该操作属于Hadoop的默认行为。**任何应用程序中的数据均会被排序，而不管逻辑上是否需要。**

默认排序是**按照字典顺序排序**，且实现该排序的方法是**快速排序。**

对于MapTask，它会将处理的结果暂时放在环形缓冲区中，**当环形缓冲区使用率达到一定阈值后，再对缓冲区中的数据进行一次快速排序**，并将这些有序数据溢写到磁盘上，而当数据处理完毕后，它会**对磁盘上所有文件进行归并排序。**

对于ReduceTask，它从每个MapTask上远程拷贝相应的数据文件，如果文件大小超过一定阈值，则溢写磁盘上，否则存储在内存中，如果磁盘上文件数目达到一定阈值，则进行一次归并排序以生成一个更大文件；如果内存中文件大小或者数目超过一定阈值，则进行一次合并后将数据溢写到磁盘上。当所有数据拷贝完毕后，**ReduceTask统一对内存和磁盘上的所有数据进行一次归并排序。**

2、排序的分类

部分排序

- MapReduce根据输入记录的键对数据集排序。保证输出的每个文件内部有序。

全排序

- 最终输出结果只有一个文件，且文件内部有序。实现方式是只设置一个ReduceTask。但该方法在处理大型文件时效率极低，因为一台机器处理所有文件，完全丧失了MapReduce所提供的并行框架。

辅助排序：(GroupingComparator分组)

- 在Reduce端对key进行分组。应用于：在接收的key为bean对象时，想让一个活几个字段相同(全部字段比较不相同)的key进入到同一个reduce方法时，可以采用分组排序。

二次排序

- 在自定义排序过程中，如果compareTo中的判断条件为两个即为二次排序。

3、自定义排序

公众号：旧时光大数据

原理分析

bean对象做为key传输，需要实现 `WritableComparable` 接口重写compareTo方法，就可以实现排序。

```
@Override
public int compareTo(FlowBean o) {

    int result;

    // 按照总流量大小，倒序排列
    if (sumFlow > bean.getSumFlow()) {
        result = -1;
    }else if (sumFlow < bean.getSumFlow()) {
        result = 1;
    }else {
        result = 0;
    }

    return result;
}

```

shuffle为什么要排序?

问过的一些公司：携程(2021.09)，网易有道(2021.09)

参考答案：

shuffle排序，按字典顺序排序的，目的是把相同的的key可以提前一步放到一起。

sort是用来shuffle的，shuffle就是把key相同的东西弄一起去，其实不一定要sort也能shuffle，那为什么要sort排序呢?

sort是为了通过外排(外部排序)降低内存的使用量：因为reduce阶段需要分组，将key相同的放在一起进行规约，使用了两种算法：hashmap和sort，如果在reduce阶段sort排序(内部排序)，太消耗内存，而map阶段的输出是要溢写到磁盘的，在磁盘中外排可以对任意数据量分组(只要磁盘够大)，所以，map端排序(shuffle阶段)，是为了减轻reduce端排序的压力。

说一下map是怎么到reduce的?

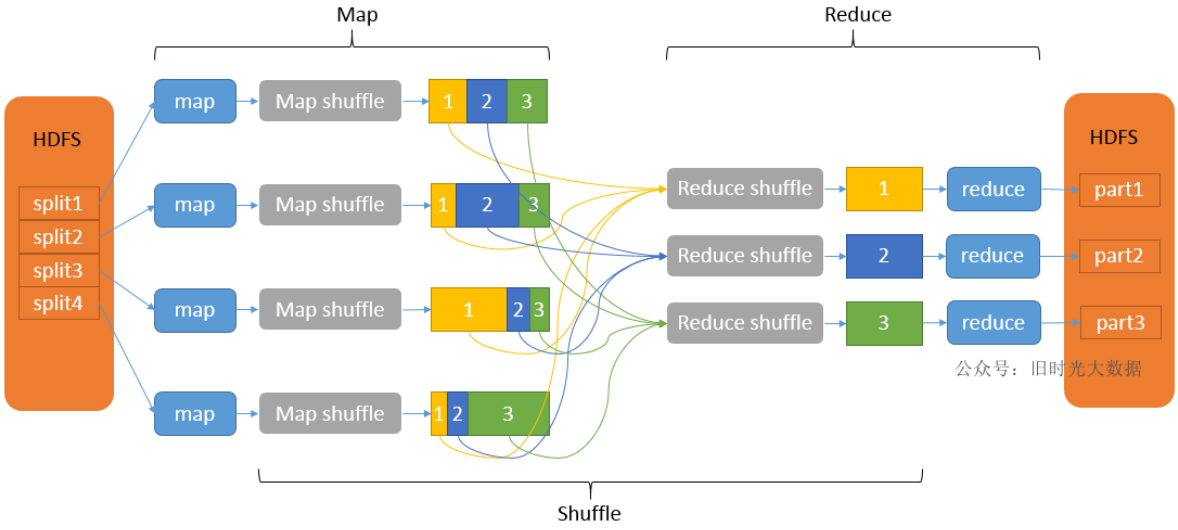
问过的一些公司：蔚来(2021.09)

参考答案：

Reduce的数据来源于Map，Map的输出即是Reduce的输入，Reduce需要通过Shuffle来获取数据。

从Map输出到Reduce输入的整个过程可以广义地称为Shuffle。Shuffle横跨Map端和Reduce端，**在Map端包括Spill过程，在Reduce端包括copy和sort过程**，如图所示：

公众号：旧时光大数据



说一下你了解的用哪几种shuffle机制?

问过的一些公司：蔚来(2021.09)

参考答案：

Map端：Spill机制，Reduce端：copy和sort机制

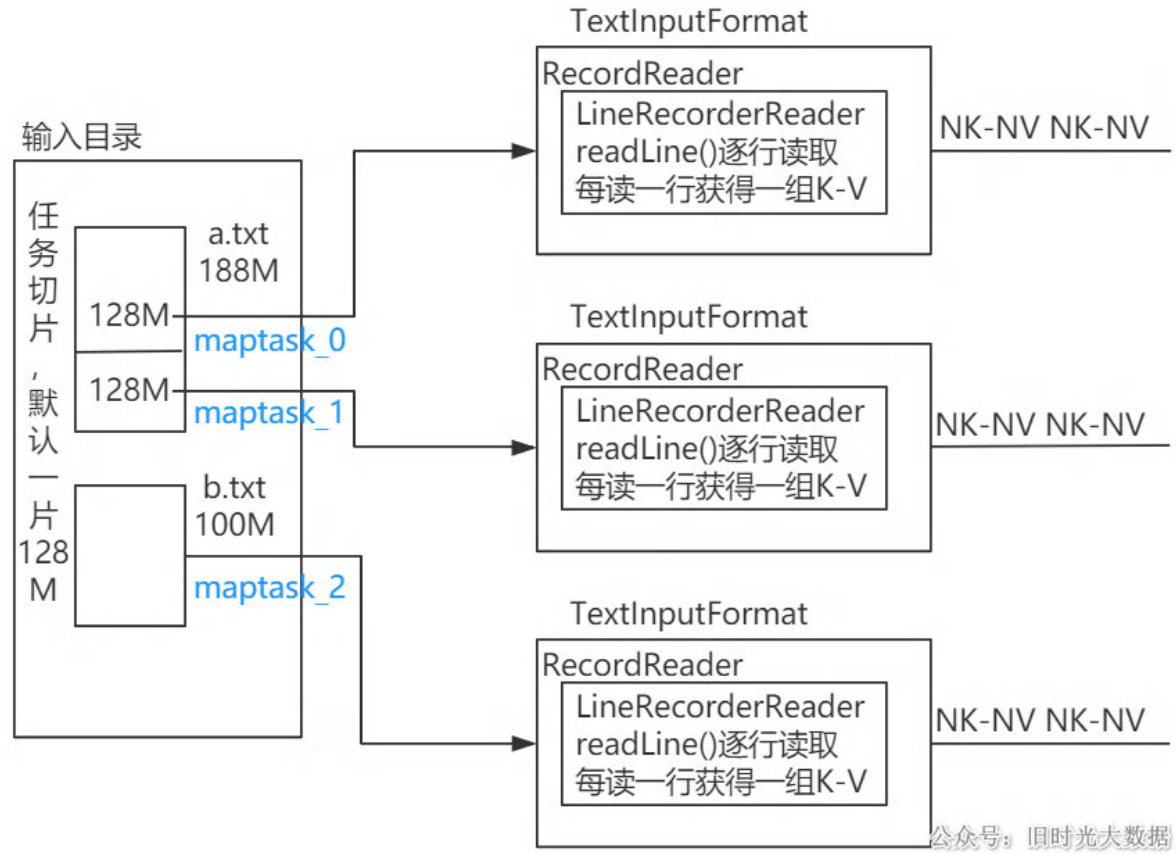
MapReduce的数据处理过程

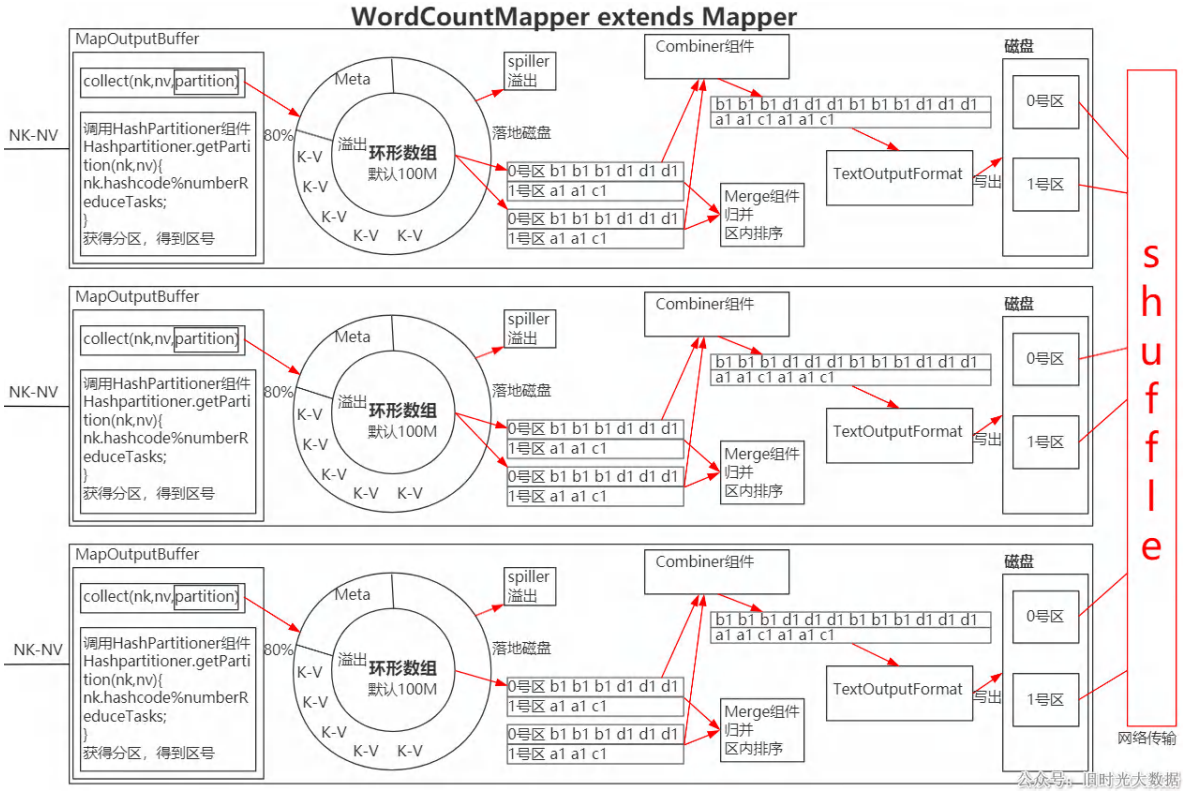
问过的一些公司：阿里，网易

回答技巧：这里也可以参考MapReduce工作原理的**版本二：Map、Reduce任务中Shuffle和排序的过程**的图

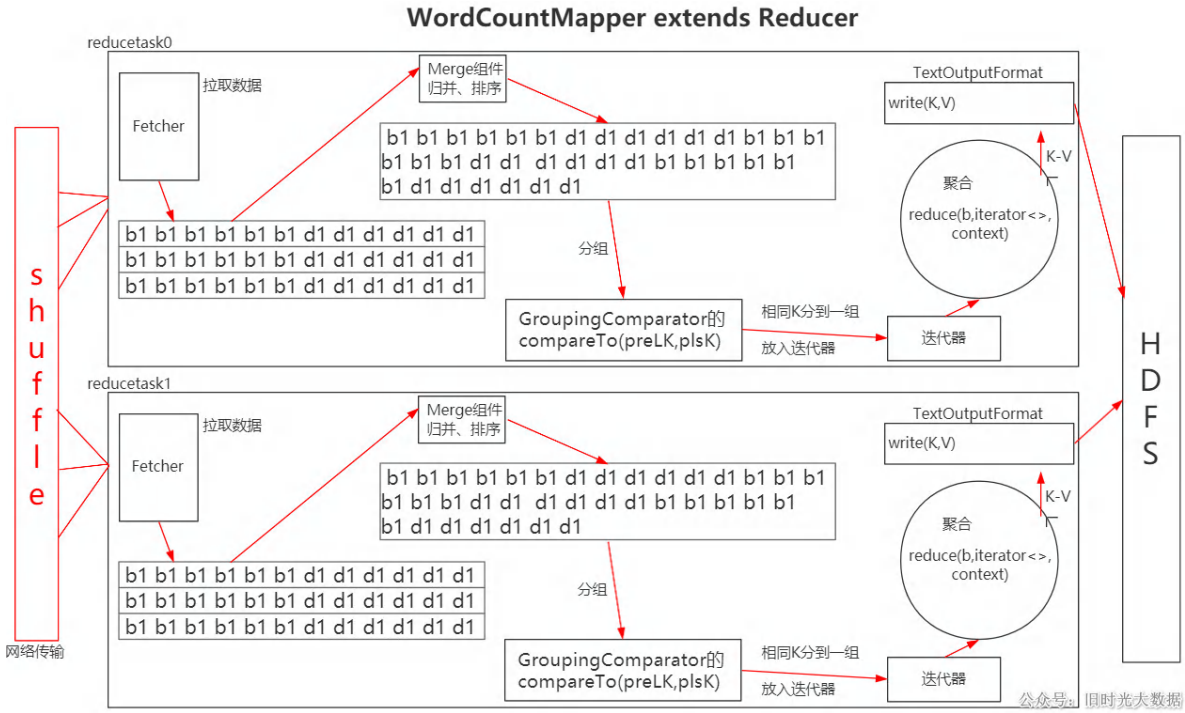
参考答案：（图太大，分成以下三幅图）

公众号：旧时光大数据


TextInputFormat
RecordReader
LineRecorderReader
readLine()逐行读取
每读一行获得一组K-V
NK-NV NK-NV
输入目录
任务切片，默认一片128M
a.txt
188M
128M
maptask_0
128M
maptask_1
b.txt
100M
maptask_2
TextInputFormat
RecordReader
LineRecorderReader
readLine()逐行读取
每读一行获得一组K-V
NK-NV NK-NV
TextInputFormat
RecordReader
LineRecorderReader
readLine()逐行读取
每读一行获得一组K-V
NK-NV NK-NV
公众号：旧时光大数据


WordCountMapper extends Mapper
MapOutputBuffer
collect(nk,nv,partition)
调用HashPartitioner组件
Hashpartitioner.getPartition(nk,nv){
nk.hashcode%numberReduceTasks;
}
获得分区，得到区号
NK-NV
80%
Meta
溢出
环形数组
默认100M
K-V
spiller
溢出
落地磁盘
0号区 b1 b1 b1 d1 d1 d1
1号区 a1 a1 c1
Combiner组件
b1 b1 b1 d1 d1 d1 b1 b1 b1 d1 d1 d1
a1 a1 c1 a1 a1 c1
Merge组件
归并
区内排序
TextOutputFormat
写出
磁盘
0号区
1号区
shuffle
网络传输
公众号：旧时光大数据

公众号：旧时光大数据



1、任务切分：对文件进行逻辑切片，切片按照范围划分，默认128M一片。

一个文件至少有一个切片，每个切片运行一个maptask，如果文件超过128M，同一个输入文件会有多个maptask运行；为减少资源浪费，如果最后一个切片大小小于1.1*128M，将不会被切分处理。

2、输入对象：FileInputFormat.setInputPaths()方法，指定数据输入路径；输入目录中可以有单个或多个文件。

读取数据、生成K-V对：由继承RecordReader的LineRecordReader类中的readLine()方法从输入的切片中读取数据；每读取一行执行一次，生成一组K-V。

3、map()方法：以单词统计为例，自定义的WordCountMapper类继承父类Mapper，接收K-V对，重写map()方法的业务逻辑。

map()的业务逻辑中，对数据进行切分，遍历数组，生成新的K-V对；由context.write(nk,nv)方法输出新的K-V。

map()方法执行时机：一对K-V执行一次。

4、map()的输出：context.write()被调用时，OutputController组件会将新的K-V输出到数组缓存区，写入数组缓存区中的还有新K-V的元数据；

5、KV分区：

MapOutputBuffer类调用collect(nk,nv,partition)方法接收新K-V；partition调用HashPartitioner组件；

```
HashPartitioner.getPartition(nk,nv){
nk.hashcode%numberReduceTasks;

}
```

获得分区，得到区号，返回给partition。

6、区内排序：①按照分区排序；②区内数据再按照K进行排序。

7、溢出：spiller，当缓冲区中的数据到达80%时，进行分区、排序，将数据溢出，当前处于阻塞状态，防止写入数据。（根据数据量大小溢出，至少一次）

8、归并、区内排序：将数组缓冲区中分区排序完的数据，用Merger组件进行归并，写入磁盘；同时进行区内排序。

9、局部聚合：调用Combiner组件，根据相同K进行数据聚合。

公众号：旧时光大数据

10、写入本地磁盘：将归并排序完成的数据写入本地磁盘，此阶段提供http下载，便于数据传输。

11、拉取数据：reducetask分别拉取属于自己的数据（本地->分区经由网络传输）

12、归并排序：调用Merger组件，按照K进行排序。

13、分组：调用GroupingComparator组件中的CompareTo(preK,postK)方法，将相同K的分到一组，放入同一迭代器中。

14、聚合：reduce(K,iterator<>,context)方法中将相同K的数据进行聚合操作，聚合一次得到一组K-V。

15、输出：由TextOutputFormat的write方法，写出到HDFS（或本地磁盘）。

map join的原理（实现）？应用场景?

问过的一些公司：美团，阿里

参考答案：

map join流程

Map side join是针对以下场景进行的优化：两个待连接表中，有一个表非常大，而另一个表非常小，以至于小表可以直接存放到内存中。这样，我们可以将小表复制多 份，让每个map task内存中存在一份（比如存放到hash table中），然后只扫描大表：对于大表中的每一条记录key/value，在hash table中查找是否有相同的key的记录，如果有，则连接后输出即可。

MapJoin简单说就是在Map阶段将小表读入内存，顺序扫描大表完成Join。减少昂贵的shuffle操作及reduce操作

MapJoin分为两个阶段：

- 通过MapReduce Local Task，将小表读入内存，生成HashTableFiles上传至Distributed Cache中，这里会HashTableFiles进行压缩。
- MapReduce Job在Map阶段，每个Mapper从Distributed Cache读取HashTableFiles到内存中，顺序扫描大表，在Map阶段直接进行Join，将数据传递给下一个MapReduce任务。

使用场景

MapJoin通常用于一个**很小的表和一个大表进行join的场景**，具体小表有多小，由参数hive.mapjoin.smalltable.filesize来决定，该参数表示小表的总大小，默认值为25000000字节，即25M。

Hive0.7之前，需要使用hint提示 /*+ mapjoin(table) */才会执行MapJoin,否则执行Common Join，但在0.7版本之后，默认自动会转换Map Join，由参数hive.auto.convert.join来控制，默认为true.

假设a表为一张大表，b为小表，并且hive.auto.convert.join=true,那么Hive在执行时候会自动转化为MapJoin。

应用场景

Map Join 实现方式一：分布式缓存

使用场景：一张表十分小、一张表很大。

用法:

在提交作业的时候先将小表文件放到该作业的DistributedCache中，然后从DistributeCache中取出该小表进行join (比如放到Hash Map等等容器中)。然后扫描大表，看大表中的每条记录的join key /value值是否能够在内存中找到相同join key的记录，如果有则直接输出结果。

DistributedCache是分布式缓存的一种实现，它在整个MapReduce框架中起着相当重要的作用，他可以支撑我们写一些相当复杂高效的分布式程序。说回到这里，JobTracker在作业启动之前会获取到DistributedCache的资源uri列表，并将对应的文件分发到各个涉及到该作业的任务的TaskTracker上。另外，关于DistributedCache和作业的关系，比如权限、存储路径区分、public和private等属性。

reduce join如何执行（原理）

问过的一些公司：美团

参考答案：

reduce side join是一种最简单的join方式，其主要思想如下：
在map阶段，map函数同时读取两个文件File1和File2，为了区分两种来源的key/value数据对，对每条数据打一个标签> （tag）,比如：tag=0表示来自文件File1，tag=2表示来自文件File2。即：map阶段的主要任务是对不同文件中的数据打标签。> 在reduce阶段，reduce函数获取key相同的来自File1和File2文件的value list，
然后对于同一个key，对File1和File2中的数据进行join（笛卡尔乘积）。即：reduce阶段进行实际的连接操作。

MapReduce为什么不能产生过多小文件

问过的一些公司：好未来x2

参考答案：

默认情况下，TextInputFormat 切片机制是对任务按文件规划切片，不管文件多小，都会是一个单独的切片，都会单独交给一个 MapTask，这样如果有大量小文件，就会产生大量的MapTask，处理效率极其低下。

MapReduce大量小文件的优化策略：

最优方案：在数据处理的最前端（预处理、采集），就将小文件合并成大文件，在上传到HDFS做后续的分析

补救措施：如果HDFS中已经存在大量的小文件了，可以使用另一种Inputformat来做切片（CombineFileInputformat），它的切片逻辑跟FileInputformat不同，它可以将多个小文件从逻辑上规划到一个切片中，这样，多个小文件就可以交给一个 MapTask 处理。

MapReduce分区及作用

可回答：Map默认是HashPartitioner如何自定义分区

问过的一些公司：百度，头条，字节(2021.08)

参考答案：

1、默认分区

系统自动调用HashPartitioner类进行分区，默认分区是根据key的hashCode对ReduceTasks个数取模得到的。用户没法控制哪个key存储到哪个分区。

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {
    public int getPartition(K key, V value, int numReduceTasks) {
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
    }
}

```

2、自定义分区

（1）自定义类继承Partitioner，重写getPartition()方法

```
public class CustomPartitioner extends Partitioner<Text, FlowBean> {
    @Override
    public int getPartition(Text key, FlowBean value, int numPartitions) {
        // 控制分区代码逻辑
        … …
        return partition;
    }
}

```

（2）在Job驱动中，设置自定义Partitioner

```
job.setPartitionerClass(CustomPartitioner.class);

```

（3）自定义Partition后，要根据自定义Partitioner的逻辑设置相应数量的ReduceTask

```
job.setNumReduceTasks(num);

```

3、全局排序

全局排序是通过将进入map端之前的数据进行随机采样，在采取的样本中设置分割点，通过分割点将数据进行分区，将设置的分割点保存在二叉树中，Map Task每输出一个数据就会去查找其对应的区间，以此来达到分区效果。

作用

- 根据业务实际需求将统计结果按照条件产生多个输出文件（分区）
- 多个reduce任务运行，提高整体job的运行效率

ReduceTask数量和分区数量关系

问过的一些公司：携程(2021.09)

参考答案：

1）如果ReduceTask的数量 > getPartition的结果数（ReduceTask > 分区数量），则会多产生几个空的输出文件part-r-000xx；

2）如果1<ReduceTask的数量 < getPartition的结果数（1 < ReduceTask < 分区数量），则有一部分分区数据无处安放，会Exception；

3）如果ReduceTask的数量=1，则不管MapTask端输出多少个分区文件，最终结果都交给这一个ReduceTask，最终也就只会产生一个结果文件part-r-00000。

案例分析

公众号：旧时光大数据

例如：假设自定义分区数为5，则

1）job.setNumReduceTasks(1); => 会正常运行，只不过会产生一个输出文件

2）job.setNumReduceTasks(2); => 会报错

3）job.setNumReduceTasks(6); => 大于5，程序会正常运行，会产生空文件

Map的分片有多大

问过的一些公司：360

参考答案：

Hadoop中在 在进行map计算之前，mapreduce会根据输入文件计算输入分片（input split），每个输入分片（input split）针对一个map任务，输入分片（input split）存储的并非数据本身，而是一个分片长度和一个记录数据的位置的数组。

源码中切片大小：

```
Math.max(minSize, Math.min(maxSize, blockSize));
mapreduce.input.fileinputformat.split.minsize=1    默认值为1
mapreduce.input.fileinputformat.split.maxsize= Long.MAXValue   默认值Long.MAXValue

```

因此，**默认情况下，切片大小=blocksize。**

Hadoop 2.7.2版本及之前默认64MB，Hadoop 2.7.3版本及之后默认128M，可以在hdfs-site.xml中设置dfs.block.size，注意单位是byte。

分片大小范围可以在mapred-site.xml中设置，mapred.min.split.size和mapred.max.split.size，minSplitSize大小默认为1B，maxSplitSize大小默认为Long.MAX_VALUE = 9223372036854775807

切片大小设置：

maxsize（切片最大值）：参数如果调得比blockSize小，则会让切片变小，而且就等于配置的这个参数的值。

minsize（切片最小值）：参数调的比blockSize大，则可以让切片变得比blockSize还大。

MapReduce join两个表的流程?

可回答：MapReduce里join怎么做

问过的一些公司：字节，阿里(2021.09)

参考答案：

假设要进行join的数据分别来自File1和File2

1、map side join

之所以存在reduce side join，是因为在map阶段不能获取所有需要的join字段，即：同一个key对应的字段可能位于不同map中。Reduce side join是非常低效的，因为shuffle阶段要进行大量的数据传输。

公众号：旧时光大数据

Map side join是针对以下场景进行的优化：两个待连接表中，有一个表非常大，而另一个表非常小，以至于小表可以直接存放到内存中。这样，我们可以将小表复制多份，让每个map task内存中存在一份（比如存放到hash table中），然后只扫描大表：对于大表中的每一条记录key/value，在hash table中查找是否有相同的key的记录，如果有，则连接后输出即可。

为了支持文件的复制，Hadoop提供了一个类DistributedCache，使用该类的方法如下：

（1）用户使用静态方法DistributedCache.addCacheFile()指定要复制的文件，它的参数是文件的URI（如果是HDFS上的文件，可以这样：hdfs://namenode:9000/home/XXX/file，其中9000是自己配置的NameNode端口号）。JobTracker在作业启动之前会获取这个URI列表，并将相应的文件拷贝到各个TaskTracker的本地磁盘上。

（2）用户使用DistributedCache.getLocalCacheFiles()方法获取文件目录，并使用标准的文件读写API读取相应的文件。

2、SemiJoin

SemiJoin，也叫半连接，是从分布式数据库中借鉴过来的方法。它的产生动机是：对于reduce side join，跨机器的数据传输量非常大，这成了join操作的一个瓶颈，如果能够在map端过滤掉不会参加join操作的数据，则可以大大节省网络IO。

实现方法很简单：选取一个小表，假设是File1，将其参与join的key抽取出来，保存到文件File3中，File3文件一般很小，可以放到内存中。在map阶段，使用DistributedCache将File3复制到各个TaskTracker上，然后将File2中不在File3中的key对应的记录过滤掉，剩下的reduce阶段的工作与reduce side join相同。

3、reduce side join + BloomFilter

在某些情况下，SemiJoin抽取出来的小表的key集合在内存中仍然存放不下，这时候可以使用BloomFiler以节省空间。

BloomFilter最常见的作用是：判断某个元素是否在一个集合里面。它最重要的两个方法是：add() 和 contains()。最大的特点是不会存在false negative，即：如果contains()返回false，则该元素一定不在集合中，但会存在一定的true negative，即：如果contains()返回true，则该元素可能在集合中。

因而可将小表中的key保存到BloomFilter中，在map阶段过滤大表，可能有一些不在小表中的记录没有过滤掉（但是在小表中的记录一定不会过滤掉），这没关系，只不过增加了少量的网络IO而已。

手撕一段简单的MapReduce程序

问过的一些公司：美团，一点资讯

参考答案：

按照 MapReduce 编程规范，分别编写 Mapper，Reducer，Driver

输入数据

```
ss ss
cls cls
jiao
banzhang
xue
hadoop

```

输出结果数据

公众号：旧时光大数据

```
banzhang 1
cls 2
hadoop 1
jiao 1
ss 2
xue 1

```

1、编写 Mapper 类

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

/**
 * KEYIN：map阶段输入的key的类型，LongWritable
 * VALUEIN：map阶段输入的value的类型，Text
 * KEYOUT：map阶段输出的key的类型，Text
 * VALUEOUT：map阶段输出的value的类型，IntWritable
 */
public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable>{

    private Text outK = new Text();
    private IntWritable outV = new IntWritable(1);

    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        // 1 获取一行
        // ss ss
        String line = value.toString();

        // 2 切割
        // ss
        // ss
        String[] words = line.split(" ");

        // 3 循环输出
        for (String word : words) {
            // 封装outK
            outK.set(word);

            // 写出
            context.write(outK, outV);
        }
    }
}

```

2、编写Reducer 类

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
```

```
5
6   /**
7    * KEYIN：reducer阶段输入的key的类型，LongWritable
8    * VALUEIN：reducer阶段输入的value的类型，Text
9    * KEYOUT：reducer阶段输出的key的类型，Text
10   * VALUEOUT：reducer阶段输出的value的类型，IntWritable
11   */
12  public class WordCountReducer extends Reducer<Text, IntWritable, Text,
    IntWritable>{
13      private IntWritable outV = new IntWritable();
14
15      @Override
16      protected void reduce(Text key, Iterable<IntWritable> values,Context
    context) throws IOException, InterruptedException {
17
18          // 1 累加求和
19          int sum = 0;
20          // ss,(1,1)
21          for (IntWritable count : values) {
22              sum += count.get();
23          }
24
25          // 2 输出
26          outV.set(sum);
27          context.write(key,outV);
28      }
29  }
30
```

3、编写Driver 驱动类

```
1   import java.io.IOException;
2   import org.apache.hadoop.conf.Configuration;
3   import org.apache.hadoop.fs.Path;
4   import org.apache.hadoop.io.IntWritable;
5   import org.apache.hadoop.io.Text;
6   import org.apache.hadoop.mapreduce.Job;
7   import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
8   import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
9
10  public class WordCountDriver {
11
12      public static void main(String[] args) throws IOException,
    ClassNotFoundException, InterruptedException {
13
14          // 1 获取配置信息以及封装任务(获取job)
15          Configuration conf = new Configuration();
16          Job job = Job.getInstance(conf);
17
18          // 2 设置jar加载路径
19          job.setJarByClass(WordCountDriver.class);
20
21          // 3 关联map和reduce类
22          job.setMapperClass(WordCountMapper.class);
23          job.setReducerClass(WordCountReducer.class);
24
25          // 4 设置map输出的kv类型
26          job.setMapOutputKeyClass(Text.class);
```

公众号：旧时光大数据

```
27          job.setMapOutputValueClass(IntWritable.class);
28
29          // 5 设置最终输出的kv类型
30          job.setOutputKeyClass(Text.class);
31          job.setOutputValueClass(IntWritable.class);
32
33          // 6 设置输入和输出路径
34          FileInputFormat.setInputPaths(job, new
    Path("D:\\BigDataTest\\06_input\\inputword"));
35          FileOutputFormat.setOutputPath(job, new
    Path("D:\\BigDataTest\\06_output\\output1"));
36          // FileInputFormat.setInputPaths(job, new Path(args[0]));
37          // FileOutputFormat.setOutputPath(job, new Path(args[1]));
38
39          // 7 提交job
40          boolean result = job.waitForCompletion(true);
41
42          System.exit(result ? 0 : 1);
43      }
44  }
45
```

4、本地测试即可

5、也可以打包在集群上测试

reduce任务什么时候开始?

问过的一些公司：作业帮

参考答案：

只要有map任务完成，就可以开始reduce任务

MapReduce的reduce使用的是什么排序?

问过的一些公司：美团

参考答案：

这里把map和reduce的都说一下

对于MapTask，它会将处理的结果暂时放到环形缓冲区中，当环形缓冲区使用率达到一定阈值后，再对缓冲区中的数据进行一次快速排序，并将这些有序数据溢写到磁盘上，而当数据处理完毕后，它会对磁盘上所有文件进行归并排序。

对于ReduceTask，它从每个MapTask上远程拷贝相应的数据文件，如果文件大小超过一定阈值，则溢写磁盘上，否则存储在内存中。如果磁盘上文件数目达到一定阈值，则进行一次归并排序以生成一个更大文件；如果内存中文件大小或者 数目超过一定阈值，则进行一次合并后将数据溢写到磁盘上。当所有数据拷贝完毕后，ReduceTask统一对内存和磁盘上的所有数据进行一次归并排序。

MapReduce怎么确定MapTask的数量?

公众号：旧时光大数据

可回答：MapReduce中的task数量确定中的MapTask数量确定可参考该题

问过的一些公司：携程，好未来，蘑菇街

参考答案：

MapTask数量影响因素

影响map个数（split个数）的**主要因素**有：

文件的大小。当块（dfs.block.size）为128m时，如果输入文件为128m，会被划分为1个split；当块为256m，会被划分为2个split。

文件的个数。FileInputFormat按照文件分割split，并且只会分割大文件，即那些大小超过HDFS块的大小的文件。如果HDFS中dfs.block.size设置为128m，而输入的目录中文件有100个，则划分后的split个数至少为100个。

splitSize的大小。分片是按照splitszie的大小进行分割的，一个split的大小在没有设置的情况下，默认等于hdfs block的大小。已知以下参数

```
input_file_num : 输入文件的个数
block_size : hdfs的文件块大小，2.7.3默认为128M，可以通过hdfs-site.xml中的dfs.block.size参数进行设置
total_size : 输入文件整体的大小，由框架循环叠加计算

```

MapTask的数量计算原则为：

（1）默认map个数

如果不进行任何设置，默认的map个数是和blcok_size相关的。

```
default_num = total_size / block_size;

```

（2）自定义设置分片的minSize、maxSize

如果在MapReduce的驱动程序(main方法)中通过以下方法设置了分片的最小或最大的大小

```
//设置最小分片大小，单位byte
FileInputFormat.setMinInputSplitSize(job,1024*1024*10L); //10MB
//设置最大分片大小，单位byte
FileInputFormat.setMaxInputSplitSize(job,1024L); //1KB

```

应用程序可以通过数据分片的最大和最小大小两个参数来对splitsize进行调节，则计算方式就变成了

```
splitSize=Math.max(minSize, Math.min(maxSize, blockSize)

```

其中maxSize即方法`setMaxInputSplitSize`设置的值，minSize即方法·setMinInputSplitSize·设置的值。

其设置原则就是

```
要增加map的个数，调整maxSize<blockSize;
要减小map的个数，调整minSize>blockSize。

```

总结：默认block为128M，当输入文件大于128M时，则会进行分块，分块后剩余文件大小大于128*1.1时，则会继续分块，否则不再分块，有多少块，就有多少MapTask。

Map数量由什么决定

问过的一些公司：远景智能(2021.08)

参考答案：

影响map个数（split个数）的**主要因素**有：

文件的大小。当块（dfs.block.size）为128m时，如果输入文件为128m，会被划分为1个split；当块为256m，会被划分为2个split。

文件的个数。FileInputFormat按照文件分割split，并且只会分割大文件，即那些大小超过HDFS块的大小的文件。如果HDFS中dfs.block.size设置为128m，而输入的目录中文件有100个，则划分后的split个数至少为100个。

splitSize的大小。分片是按照splitszie的大小进行分割的，一个split的大小在没有设置的情况下，默认等于hdfs block的大小。已知以下参数

```
input_file_num : 输入文件的个数
block_size : hdfs的文件块大小，2.7.3默认为128M，可以通过hdfs-site.xml中的dfs.block.size参数进行设置
total_size : 输入文件整体的大小，由框架循环叠加计算

```

MapReduce的map进程和reducer进程的jvm垃圾回收器怎么选择可以提高吞吐量?

问过的一些公司：网易云音乐

参考答案：

开启JVM重用

属性：mapred.job.reuse.jvm.num.tasks，默认值是1，在一个taskTracker上对于给定的作业的每个jvm上可以运行任务最大数。-1表示无限制，即同一个jvm可以被该作业的所有任务使用。

```
conf.set("mapreduce.job.jvm.numtasks", "-1"); //开启jvm重用
job.getConfiguration().setInt(job.JVM_NUMTASKS_TORUN, -1); //开启jvm重用

```

MapReduce的task数目划分

可回答：Hadoop reduce数量怎么确定

问过的一些公司：字节，美团(2021.09)，携程(2021.09)

参考答案：

在MapReduce当中，每个mapTask处理一个切片split的数据量，要注意切片与block块的概念很像，但是**block块是HDFS当中存储数据的单位，切片split是MapReduce当中每个MapTask处理数据量的单位。**

在介绍map task的数量及切片机制之前先了解这两个概念：

block块（数据块，物理划分）

公众号：旧时光大数据

block是HDFS中的基本存储单位，hadoop1.x默认大小为64M，而hadoop2.x默认块大小为128M。文件上传到HDFS，就要划分数据成块，这里的划分属于物理的划分（实现机制也就是设置一个read方法，每次限制最多读128M的数据后调用write进行写入到hdfs），块的大小可通过 dfs.block.size配置。block采用冗余机制保证数据的安全：默认为3份，可通过dfs.replication配置。

注意：当更改块大小的配置后，新上传的文件的块大小为新配置的值，以前上传的文件的块大小为以前的配置值。

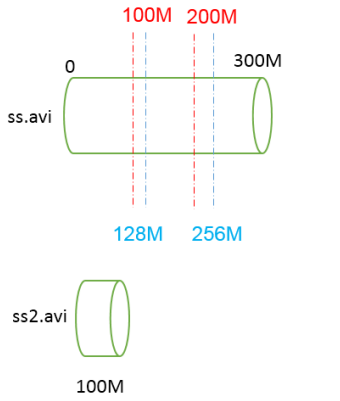
split分片（数据分片，逻辑划分）

Hadoop中split划分属于逻辑上的划分，目的只是为了让map task更好地获取数据。split是通过hadoop中的 InputFormat接口中的getSplits()方法得到的。数据切片只是在逻辑上对输入进行分片，并不会在磁盘上将其切分成片进行存储。

1、MapTask个数

1、假设切片大小设置为100M

2、假设切片大小设置为128M

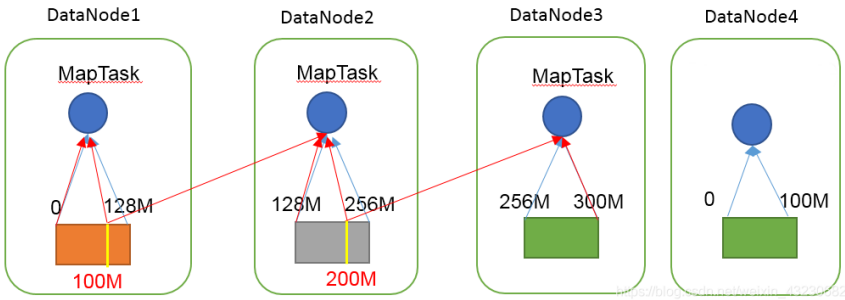


1）一个Job的Map阶段并行度由客户端在提交Job时的切片数决定

2）每一个Split切片分配一个MapTask并行实例处理

3）默认情况下，切片大小=BlockSize

4）切片时不考虑数据集整体，而是逐个针对每一个文件单独切片



综上所述

Split切片数就是我们的MapTask数量，切片的大小是可以自行设置的

切片大小的计算公式

```
Math.max(minSize, Math.min(maxSize, blockSize));
mapreduce.input.fileinputformat.split.minsize=1    // 默认值为1
mapreduce.input.fileinputformat.split.maxsize= Long.MAXValue   // 默认值Long.MAXValue
// blockSize为128M

```

2、如何控制mapTask的个数

MapTask数量设置不当带来的问题：

- Map Task数量过多的话，会产生大量的小文件，过多的Mapper创建和初始化都会消耗大量的硬件资源。
- Map Task数量过少，就会导致并发度过小，Job执行时间过长，无法充分利用分布式硬件资源。

那么，如果需要控制maptask的个数，我们只需要调整maxSize和minsize这两个值，那么切片的大小就会改变，切片大小改变之后，mapTask的个数就会改变，我们可以在MapReduce的驱动程序(main方法)中通过以下方法设置了分片的最小或最大的大小

公众号：旧时光大数据

```
//设置最小分片大小，单位byte
FileInputFormat.setMinInputSplitSize(job,1024*1024*10L); //10MB
//设置最大分片大小，单位byte
FileInputFormat.setMaxInputSplitSize(job,1024L); //1KB

```

设置原则如下

```
要增加map的个数，调整maxSize<blockSize;
要减小map的个数，调整minSize>blockSize。

```

三、 ReduceTask数量的确定

Reduce任务是一个数据聚合的步骤，数量默认为1。使用过多的Reduce任务则意味着复杂的shuffle，并使输出文件数量激增。而reduce的个数设置相比map的个数设置就要简单的多，只需要设置在驱动程序中通过 `job.setNumReduceTasks(int n)` 即可。

MapReduce作业执行的过程中，中间的数据会存在什么地方？不会存在内存中么？

问过的一些公司：小米

参考答案：

不会存在内存，存在本地磁盘

Mapper端进行combiner之后，除了速度会提升，那从Mapper端到Reduece端的数据量会怎么变？

问过的一些公司：祖龙娱乐

参考答案：

数据量会减少，因为combiner之后，会将相同的key进行一次聚合，数据量会在这时候减少一部分

map输出的数据如何超出它的小文件内存之后，是落地到磁盘还是落地到HDFS中？

问过的一些公司：祖龙娱乐

参考答案：

数据会落地到磁盘中，因为map和reduce操作，就是一次次的I/O请求

Map到Reduce默认的分区机制是什么？

问过的一些公司：祖龙娱乐

参考答案：

公众号：旧时光大数据

默认分区是根据key的hashCode对ReduceTask个数取模得到的。用户没法控制哪个key存储到哪个分区。

结合wordcount述说MapReduce，具体各个流程，map怎么做，reduce怎么做

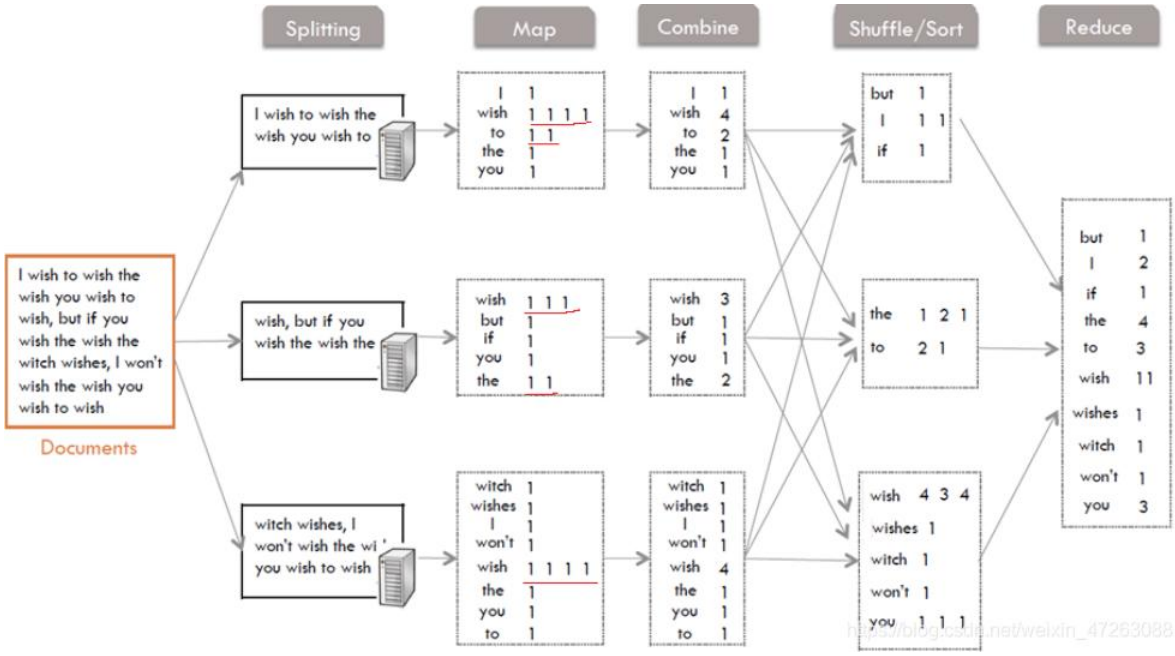
可回答：1）给一个场景，mapreduce统计词频原理；2）WordCount在MapReduce中键值对的变化；3）讲下MapReduce的具体任务过程；4）一次MapReduce过程

回答技巧：比如问结合实际场景说下MapReduce，可以说根据wordcount来讲一下MapReduce过程

问过的一些公司：字节，大华，趋势科技，一点资讯，奇安信，58同城(2021.08)，阿里(2021.09)

参考答案：

先来看一张图



具体各个阶段做了什么

spliting ： Documents会根据切割规则被切成若干块，

map阶段：然后进行Map过程，Map会并行读取文本，对读取的单词进行单词分割，并且每个词以键值对<key,value>形式生成。

```
1    例如：读取到"Hello World Hello Java"，分割单词形成Map
2    <Hello,1> <World,1><Hello,1> <Java,1>
3
```

combine阶段：接下来Combine(该阶段是可以选择的，Combine其实也是一种reduce）会对每个片相同的词进行统计。

shuffle阶段：将Map输出作为reduce的输入的过程就是shuffle,次阶段是最耗时间,也是重点需要优化的阶段。shuffle阶段会对数据进行拉取，对最后得到单词进行统计，每个单词的位置会根据Hash来确定所在的位置，

reduce阶段：对数据做最后的汇总，最后结果是存储在hdfs上。

MapReduce数据倾斜产生的原因及其解决方案

公众号：旧时光大数据

可回答：Hadoop数据倾斜问题怎么解决

问过的一些公司：腾讯，大华，多益，冠群驰骋，网易，端点数据(2021.07)，阿里蚂蚁(2021.08)，字节(2021.08)，茄子科技(2021.09)，快手(2021.09)

参考答案：

1、数据倾斜现象

数据倾斜就是数据的key的分化严重不均，造成一部分数据很多，一部分数据很少的局面。

数据频率倾斜——某一个区域的数据量要远远大于其他区域。

数据大小倾斜——部分记录的大小远远大于平均值。

2、数据倾斜产生的原因

（1）Hadoop框架的特性

- Job数多的作业运行效率会相对比较低；
- countdistinct、group by、join等操作，触发了Shuffle动作，导致全部相同key的值聚集在一个或几个节点上，很容易发生单点问题。

（2）具体原因

- key 分布不均匀，某一个key的条数比其他key多太多；
- 业务数据自带的特性；
- 建表时考虑不全面；
- 可能某些 HQL 语句自身就存在数据倾斜问题。

3、数据倾斜解决方案

从业务和数据方面解决数据倾斜

- 有损的方法：找到异常数据。
- 无损的方法：
  - 对分布不均匀的数据，进行单独计算，首先对key做一层hash，把数据打散，让它的并行度变大，之后进行汇集
- 数据预处理

Hadoop平台的解决方法

1）针对join产生的数据倾斜

场景一：大表和小表join产生的数据倾斜

① 在多表关联情况下，将小表(关联键记录少的表)依次放到前面，这样能够触发reduce端减少操作次数，从而减少运行时间。

② 同时使用Map Join让小表缓存到内存。在map端完成join过程，这样就能省掉redcue端的工作。需要注意：这一功能使用时，需要开启map-side join的设置属性：set hive.auto.convert.join=true(默认是false)

③ 还可以对使用这个优化的小表的大小进行设置：

```
1   set hive.mapjoin.smalltable.filesize=25000000(默认值25M)
2
```

场景二：大表和大表的join产生的数据倾斜

① 将异常值赋一个随机值，以此来分散key,均匀分配给多个reduce去执行

② 如果key值都是有效值的情况下，需要设置以下几个参数来解决

```
set hive.exec.reducers.bytes.per.reducer = 1000000000
```

也就是每个节点的reduce，其 默认是处理数据地大小为1G，如果join 操作也产生了数据倾斜，那么就在 hive 中设定

```
set hive.optimize.skewjoin = true;
set hive.skewjoin.key = skew_key_threshold (default = 100000)
```

2）group by 造成的数据倾斜

解决方式相对简单：

```
hive.map.aggr=true  (默认true) 这个配置项代表是否在map端进行聚合，相当于Combiner
hive.groupby.skewindata
```

3）count(distinct)或者其他参数不当造成的数据倾斜

① reduce个数太少

```
set mapred.reduce.tasks=800
```

② HiveQL中包含count(distinct)时

使用sum...group byl来替代。例如select a,sum(1) from (select a, b from t group by a,b) group by a;

Map Join为什么能解决数据倾斜

问过的一些公司：阿里蚂蚁(2021.08)

参考答案：

Map Join概念：将其中做连接的小表（全量数据）分发到所有 MapTask 端进行 Join，从而避免了 reduceTask，前提要求是内存足以装下该全量数据。

Map Join通常用于一个很小的表和一个大表进行join的场景，具体小表有多小，由参数 hive.mapjoin.smalltable.filesize来决定，该参数表示小表的总大小，默认值为25000000字节，即25M。 一般默认就够了，无须修改。

1）在多表关联情况下，将小表(关联键记录少的表)依次放到前面，**这样能够触发reduce端减少操作次数，从而减少运行时间。**

2）同时使用Map Join让小表缓存到内存。在map端完成join过程，这样就能省掉redcue端的工作。需要注意：这一功能使用时，需要开启map-side join的设置属性：set hive.auto.convert.join=true(默认是false)

注意： 大表放硬盘，小表放内存 （ 前提要求是内存足以装下该全量数据 ）。

MapReduce运行过程中会发生OOM，OOM发生的位置?

问过的一些公司：作业帮

参考答案：

背景：Hive中跑MapReduce Job出现OOM问题

公众号：旧时光大数据

该异常发生既不是map阶段，也不是reduce阶段，发现不是执行过程，而是driver提交job阶段就OOM了。

```
Hive中XMLEncoder序列化MapredWork引发OutOfMemoryError
XMLEncoder导致java.lang.OutOfMemoryError: GC overhead limit exceeded

```

先来看下，Hive中出现OOM的异常原因大致分为以下几种：

- Map阶段OOM
- Reduce阶段OOM
- Driver提交Job阶段OOM

1、Map阶段OOM

发生OOM的几率很小，除非你程序的逻辑不正常，亦或是程序写的不高效，产生垃圾太多。

2、Reduce阶段OOM

1）data skew 数据倾斜

data skew是引发这个的一个原因。

key分布不均匀，导致某一个reduce所处理的数据超过预期，导致jvm频繁GC。

2）value对象过多或者过大

某个reduce中的value堆积的对象过多，导致jvm频繁GC。

解决方案：

① 增加reduce个数，set mapred.reduce.tasks=300，。

② 在hive-site.xml中设置，或者在hive shell里设置 set mapred.child.java.opts = -Xmx512m

或者只设置reduce的最大heap为2G，并设置垃圾回收器的类型为并行标记回收器，这样可以显著减少GC停顿，但是稍微耗费CPU。

set mapred.reduce.child.java.opts=-Xmx2g -XX:+UseConcMarkSweepGC;

③ 使用map join 代替 common join. 可以set hive.auto.convert.join = true

④ 设置 hive.optimize.skewjoin = true 来解决数据倾斜问题

3、Driver提交Job阶段OOM

job产生的执行计划的条目太多，比如扫描的分区过多，上到4k-6k个分区的时候，并且是好几张表的分区都很多时，这时做join。

究其原因，是因为序列化时，会将这些分区，即hdfs文件路径，封装为Path对象，这样，如果对象太多了，而且Driver启动的时候设置的heap size太小，则会导致在Driver内序列化这些MapRedWork时，生成的对象太多，导致频繁GC，则会引发如下异常:

```
java.lang.OutOfMemoryError: GC overhead limit exceeded
at sun.nio.cs.UTF_8.newEncoder(UTF_8.java:53)
at java.beans.XMLEncoder.createString(XMLEncoder.java:572)

```

解决方案：

经过查询， 是因为扫描的表的分区太多，上到3千到6千个分区，这样在对计划进行序列化时，仅仅是路径对象Path就会耗去大半Driver，如果Driver设置的heap太小，甚至都会OOM。

解决思路

公众号：旧时光大数据

① 减少分区数量，将历史数据做成一张整合表，做成增量数据表，这样分区就很少了。

② 调大Hive CLI Driver的heap size, 默认是256MB，调节成512MB或者更大。

具体做法是在bin/hive bin/hive-config里可以找到启动CLI的JVM OPTIONS。

这里我们设置

```
export HADOOP_HEAPSIZE=512

```

双管齐下， 即做成了整合，方便使用，又调节了Hive CLI Driver的heap size，保证线上的运行稳定。

遇到这种问题：

- 一是HiveQL的写法上，尽量少的扫描同一张表，并且尽量少的扫描分区。扫太多，一是job数多，慢，二是耗费网络资源，慢。
- 二是Hive的参数调优和JVM的参数调优，尽量在每个阶段，选择合适的jvm max heap size来应对OOM的问题。

MapReduce用了几次排序，分别是什么?

可回答：MapReduce过程用到了哪些排序?

问过的一些公司：美团，米哈游，大华(2021.07)，字节(2021.08)

参考答案：

在Map任务和Reduce任务的过程中，**一共发生了3次排序**

1）当map函数产生输出时，会首先写入内存的环形缓冲区，当达到设定的阀值，在刷写磁盘之前，后台线程会将缓冲区的数据划分成相应的分区。在每个分区中，后台线程按键进行内排序

2）在Map任务完成之前，磁盘上存在多个已经分好区，并排好序的，大小和缓冲区一样的溢写文件，这时溢写文件将被合并成一个已分区且已排序的输出文件。由于溢写文件已经经过第一次排序，所有合并文件只需要再做一次排序即可使输出文件整体有序。

3）在reduce阶段，需要将多个Map任务的输出文件copy到ReduceTask中后合并，由于经过第二次排序，所以合并文件时只需再做一次排序即可使输出文件整体有序

在这3次排序中**第一次是内存缓冲区做的内排序，使用的算法使快速排序，第二次排序和第三次排序都是在文件合并阶段发生的，使用的是归并排序。**

MapReduce压缩方式

问过的一些公司：转转

参考答案：

1、MapReduce支持的压缩方式

公众号：旧时光大数据

压缩格式	hadoop自带?	算法	文件扩展名	是否可切分	换成压缩格式后，原来的程序是否需要修改
DEFLATE	是，直接使用	DEFLATE	.deflate	否	和文本处理一样，不需要修改
Gzip	是，直接使用	DEFLATE	.gz	否	和文本处理一样，不需要修改
bzip2	是，直接使用	bzip2	.bz2	是	和文本处理一样，不需要修改
LZO	否，需要安装	LZO	.lzo	是	需要建索引，还需要指定输入格式
Snappy	否，需要安装	Snappy	.snappy	否	和文本处理一样，不需要修改

2、压缩性能比较

压缩算法	原始文件大小	压缩文件大小	压缩速度	解压速度
gzip	8.3GB	1.8GB	17.5MB/s	58MB/s
bzip2	8.3GB	1.1GB	2.4MB/s	9.5MB/s
LZO	8.3GB	2.9GB	49.3MB/s	74.6MB/s

3、压缩方式选择

压缩方式选择时重点考虑：压缩/解压缩速度、压缩率（压缩后存储大小）、压缩后是否可以支持切片。

`Gzip压缩`

优点：压缩率比较高，而且压缩/解压速度也比较快；Hadoop本身支持，在应用中处理Gzip格式的文件就和直接处理文本一样；大部分Linux系统都自带Gzip命令，使用方便。

缺点：不支持Split。

应用场景：当每个文件压缩之后在130M以内的（1个块大小），都可以考虑使用Gzip压缩格式。例如一天或者一个小时的日志压缩成一个Gzip文件。

`Bzip2压缩`

优点：支持split；具有很高的压缩率，比gzip压缩率都高；hadoop本身支持，但不支持native；在linux系统下自带bzip2命令，使用方便。

缺点：压缩/解压速度慢；不支持native。

公众号：旧时光大数据

应用场景：适合对速度要求不高，但需要较高的压缩率的时候，可以作为mapreduce作业的输出格式；或者输出之后的数据比较大，处理之后的数据需要压缩存档减少磁盘空间并且以后数据用得比较少的情况；或者对单个很大的文本文件想压缩减少存储空间，同时又需要支持split，而且兼容之前的应用程序（即应用程序不需要修改）的情况。

Lzo压缩

优点：压缩/解压速度也比较快，合理的压缩率；支持split，是hadoop中最流行的压缩格式；可以在linux系统下安装lzop命令，使用方便。

缺点：压缩率比gzip要低一些；hadoop本身不支持，需要安装；在应用中对lzo格式的文件需要做一些特殊处理（为了支持split需要建索引，还需要指定inputformat为lzo格式）。

应用场景：一个很大的文本文件，压缩之后还大于200M以上的可以考虑，而且单个文件越大，lzo优点越越明显。

Snappy压缩

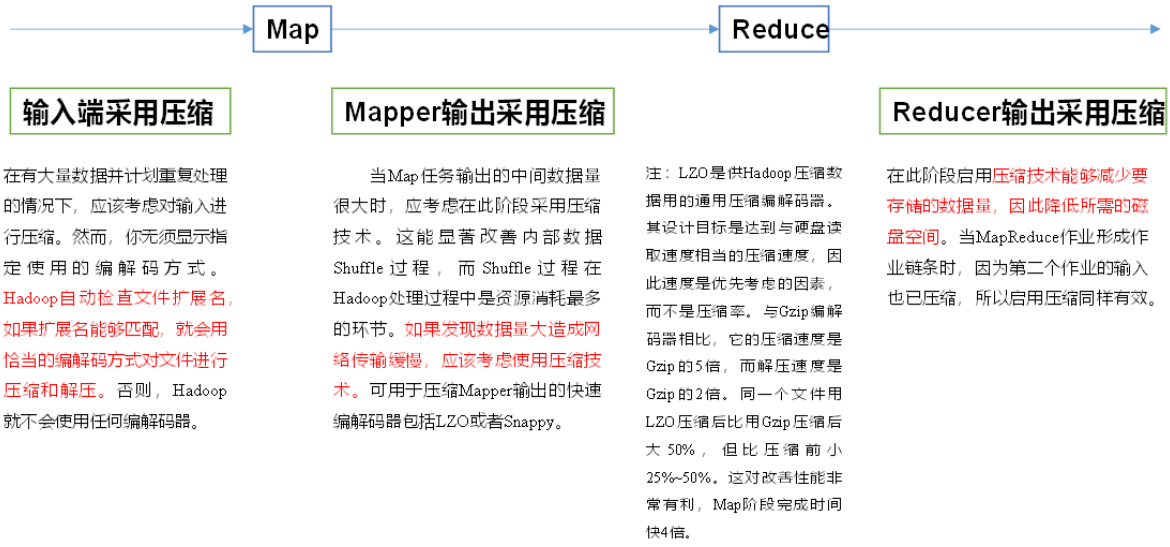
优点：高速压缩速度和合理的压缩率。

缺点：不支持split；压缩率比gzip要低；hadoop本身不支持，需要安装；

应用场景：当Mapreduce作业的Map输出的数据比较大的时候，作为Map到Reduce的中间数据的压缩格式；或者作为一个Mapreduce作业的输出和另外一个Mapreduce作业的输入。

4、压缩位置选择

压缩可以在MapReduce作用的任意阶段启用



输入端采用压缩

在有大量数据并计划重复处理的情况下，应该考虑对输入进行压缩。然而，你无须显示指定使用的编解码方式。Hadoop自动检查文件扩展名，如果扩展名能够匹配，就会用恰当的编解码方式对文件进行压缩和解压。否则，Hadoop就不会使用任何编解码器。

mapper输出端采用压缩

当map任务输出的中间数据量很大时，应考虑在此阶段采用压缩技术。这能显著改善内部数据Shuffle过程，而Shuffle过程在Hadoop处理过程中是资源消耗最多的环节。如果发现数据量大造成网络传输缓慢，应该考虑使用压缩技术。可用于压缩mapper输出的快速编解码器包括LZO或者Snappy。

reducer输出采用压缩

在此阶段启用压缩技术能够减少要存储的数据量，因此降低所需的磁盘空间。当mapreduce作业形成作业链条时，因为第二个作业的输入也已压缩，所以启用压缩同样有效。

公众号：旧时光大数据

MapReduce中怎么处理一个大文件

问过的一些公司：京东

参考答案：

1、输入大文件时

```
// 小于这个数据时进行合并
conf.setLong(FileInputFormat.SPLIT_MINSIZE,1024*1024*256L);
// 大于这个数据时进行切分
conf.setLong(FileInputFormat.SPLIT_MAXSIZE,1024*1024*1024);

```

2、输入大量小文件时

方式一：小文件先进行Merge操作再使用MapReduce

方式二：使用FileInputFormat子类CombineFileInputFormat重写RecordReader()将多个input path合并成一个InputSplit

Hadoop中用到了那些缓存机制?

问过的一些公司：大华(2021.07)

参考答案：

分布式缓存

就是在job任务执行前，将需要的文件拷贝到Task机器上进行缓存，提高mapreduce的执行效率。

YARN部分

介绍下YARN

可回答：1）YARN的RM和NM；2）YARN有哪些组件，如何分配资源；3）YARN的AM和RM的作用；

回答技巧：YARN的架构，执行流程等

问过的一些公司：字节，字节(2021.08)，网易云音乐×2，蘑菇街x2，美团，小鹏汽车，一点资讯，头条，央视网，海康x2，海康(2021.09)，恒生(2021.09)

参考答案：

介绍YARN，可以先考虑下面两个问题

1）如何管理集群资源?

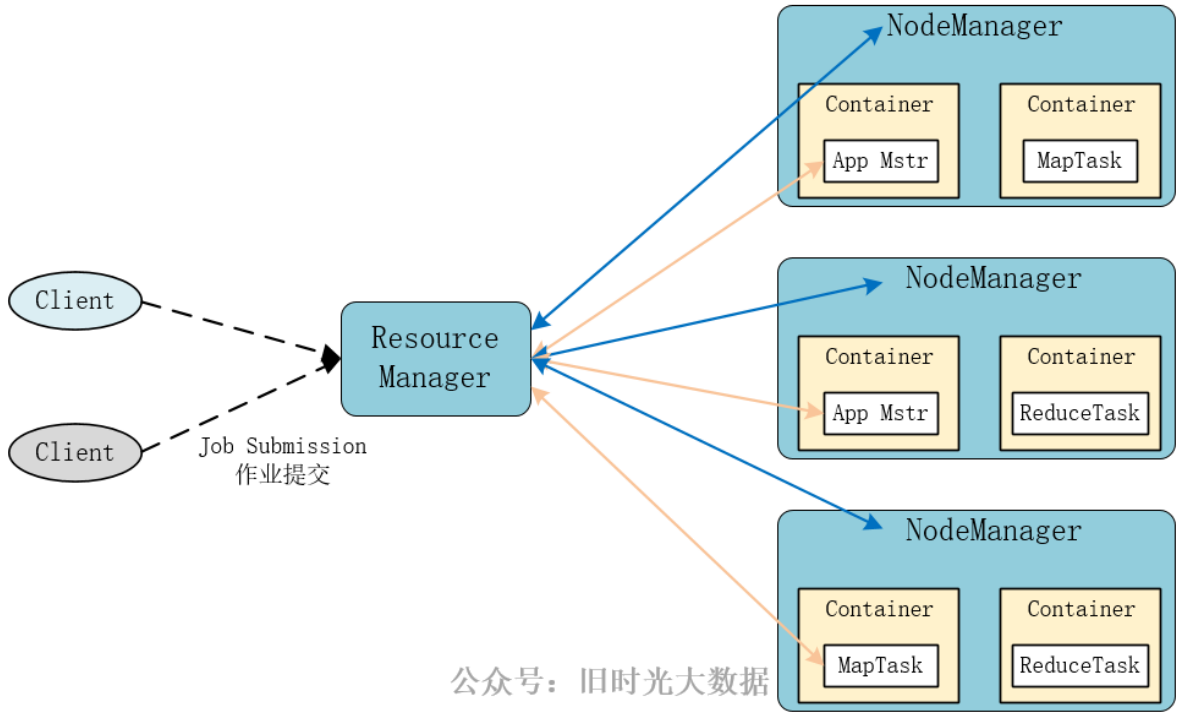
2）如何给任务合理分配资源?

YARN是一个资源调度平台，负责为运算程序提供服务器运算资源，相当于一个**分布式的操作系统平台**，而**MapReduce等运算程序则相当于运行于操作系统之上的应用程序。**

YARN 作为一个资源管理、任务调度的框架，主要包含ResourceManager、 NodeManager、ApplicationMaster和Container模块。

公众号：旧时光大数据

YARN基础架构



公众号：旧时光大数据

1）ResourceManager（RM）主要作用如下：

- 处理客户端请求
- 监控NodeManager
- 启动或监控ApplicationMaster
- 资源的分配与调度

2）NodeManager（NM）主要作用如下：

- 管理单个节点上的资源
- 处理来自ResourceManager的命令
- 处理来自ApplicationMaster的命令

3）ApplicationMaster（AM）作用如下：

- 为应用程序申请资源并分配给内部的任务
- 任务的监督与容错

4）Container

- Container是YARN中的资源抽象，它封装了某个节点上的多维度资源，如内存、CPU、磁盘、网络等。

可以结合“YARN有什么优势，能解决什么问题？”一起回答

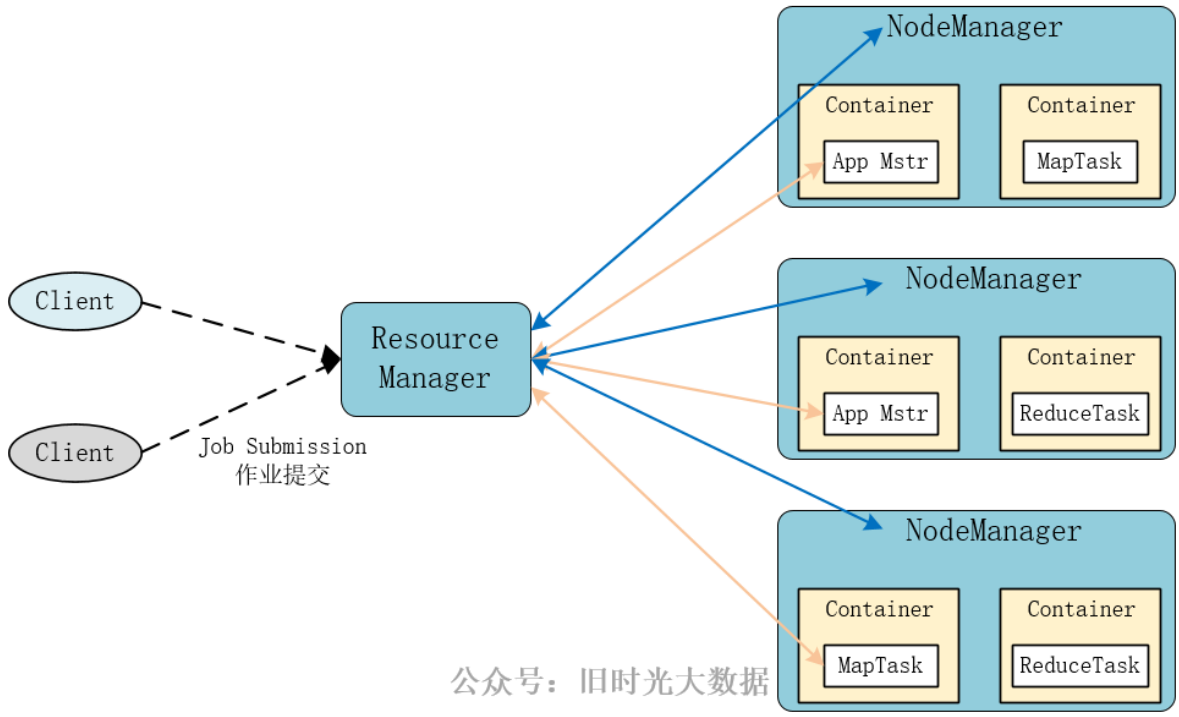
YARN有几个模块

问过的一些公司：字节(2022.03)

参考答案：

YARN基础架构

公众号：旧时光大数据



公众号：旧时光大数据

1）ResourceManager（RM）主要作用如下：

- 处理客户端请求
- 监控NodeManager
- 启动或监控ApplicationMaster
- 资源的分配与调度

2）NodeManager（NM）主要作用如下：

- 管理单个节点上的资源
- 处理来自ResourceManager的命令
- 处理来自ApplicationMaster的命令

3）ApplicationMaster（AM）作用如下：

- 为应用程序申请资源并分配给内部的任务
- 任务的监督与容错

4）Container

- Container是YARN中的资源抽象，它封装了某个节点上的多维度资源，如内存、CPU、磁盘、网络等。

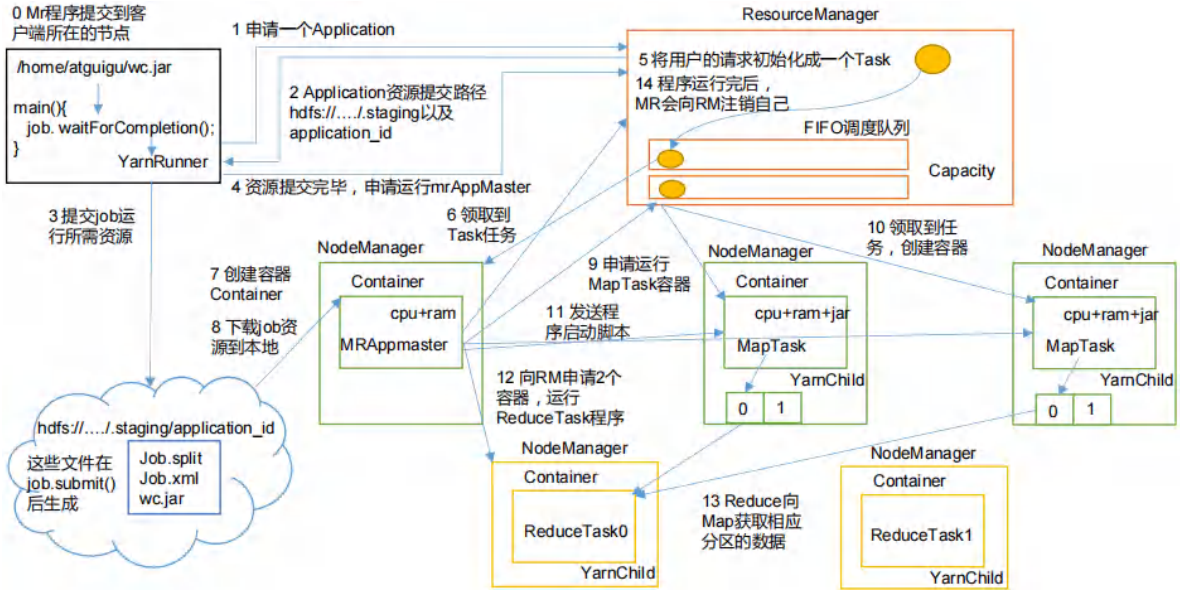
YARN工作机制

可回答：1）YARN的任务提交流程；2）YARN的通信流程或工作流程；3）YARN执行一个application的流程；4）Hadoop任务的YARN调度过程；5）YARN上提交资源（程序）的流程；6）YARN的资源调度流程；7）YARN的任务提交流程

问过的一些公司：字节x2，字节(2021.08)-(2022.03)，网易云音乐，快手，海康x2，转转，滴滴，作业帮，虎牙(2021.09)，四方伟业(2021.08)，蔚来(2021.09)

参考答案：

公众号：旧时光大数据



1）MapReduce程序提交到客户端所在的节点。

2）YarnRunner向ResourceManager申请一个Application。

3）RM 将该应用程序的资源路径以及application_id返回给YarnRunner。

4）该程序将运行所需资源提交到 HDFS 上。

5）程序资源提交完毕后，申请运行 mrAppMaster。

6）RM 将用户的请求初始化成一个 Task。

7）其中一个NodeManager 领取到 Task 任务。

8）该 NodeManager 创建容器 Container，并产生 MRAppmaster。

9）Container 从HDFS 上拷贝资源到本地。

10）MRAppmaster 向RM 申请运行 MapTask 资源。

11）RM 将运行 MapTask 任务分配给另外两个 NodeManager，另两个 NodeManager 分别领取任务并创建容器。

12）MR 向两个接收到任务的NodeManager 发送程序启动脚本，这两个 NodeManager分别启动 MapTask，MapTask 对数据分区排序。

13）MrAppMaster 等待所有 MapTask 运行完毕后，向 RM 申请容器，运行 ReduceTask。

14）ReduceTask 向 MapTask 获取相应分区的数据。

15）程序运行完毕后，MR 会向 RM 申请注销自己。

YARN有什么优势，能解决什么问题?

问过的一些公司：祖龙娱乐

参考答案：

YARN的优点

- 解决了单点故障问题，由于每一个任务由一个AppMaster进行调度，且可进行AppMaster出错重试，从而使单点故障影响到多个任务进行问题不存在。

公众号：旧时光大数据

- 解决了单点压力过大问题，每一个任务由一个AppMaster进行调度，而每一个AppMaster都是由集群中资源较为充足的结点进行启动，调度任务，起到一个负载均衡的作用。
- 完成了资源管理和任务调度的解耦，Yarn只负责对集群资源的管理，各个计算框架只要继承了AppMaster，就可以共同使用Yarn资源管理，更加充分地利用集群资源。

解决的问题

在Hadoop 1.x版本时，JobTracker和TaskTracker是常服务，资源管理和任务调度的耦合，而在Hadoop 2.x版本之后，Yarn将二者分离，只有资源管理成为了常服务，而任务调度则变成只有任务在调度时，才启用的临时服务。

YARN容错机制

问过的一些公司：美团

参考答案：

在现实情况中，用户代码错误不断，进程奔溃，机器故障等等。使用hadoop的好处之一就是可以它能处理这类故障并成功完成任务。需要考虑的实体失败任务为：任务（job），Application Master，NodeManager和ResourceManager。

`任务失败`

任务失败可能存在以下几种情况：

MapTask或者ReduceTask中由于代码原因抛出异常，jvm在关闭之前，会通知mrAppMaster这个task任务失败，在mrAppMaster中，错误报告被写入到用户日志并且任务标记为失败，并释放jvm资源，供其他任务使用。对于streaming任务，如果streaming进程以非0退出代码退出，则被标记为失败。这种行为由stream.non.zero.is.failure属性（默认值为true）控制。

jvm突然退出，可能是由于jvm缺陷而导致mr用户代码由于某种特殊原因造成jvm退出。nodeManage会将这消息通知到mrAppMaster，标记此次任务失败。

任务挂起（可能是由于资源不足造成）：一旦mrAppMaster一段时间没有接收到进度的更新，则将任务标记为失败，nodeManager会将该jvm进程杀死。任务失败时长可以由mapreduce.task.timeout来设置。如果为0，则表示关闭。如果关闭这个属性，那么可能会造成长时间运行的任务不会被标记为失败，被挂起的任务就会一直不被释放资源，长时间会造成集群效率降低，因此尽量避免这个设置。同时充分保证每个任务定期更新进度。

处理阶段：

当mrAppMaster被告知，一个任务失败的时候，会重新调度该任务。mrAppMaster会尝试避免在以前失败过的nodeManager重新调度该任务。此外，一个任务失败的次数超过4次，将不会再重新调度。这个数值由mapreduce.map.maxattempts控制。如果一个任务失败次数大于该属性设置的，则整个作业都会失败。对于一些应用程序中，不希望少部分任务失败，而导致整个作业失败，因为即使一些任务失败，作业的输出结果也是可用的，我们可用通过运行任务失败的最大比例：maptask由mapreduce.map.failures.maxpercent，reducetask由mapreduce.reduce.failures.maxpercent来设置。任务尝试也是可以用来中止（killed），因为它是一个推测副本（如果一个任务执行时间比预期的慢的时候，会启动另外一个相同的任务作为备份，这个任务为推测执行）或者它所在的nodeManager失败，导致该nodeManager所执行的任务被标记为killed，被中止的任务是不会被记录到任务运行尝试次数。

`ApplicationMaster运行失败`

在YARN中，ApplicationMaster有几次尝试次数，最多尝试次数由：mapreduce.am.max-attempts和yarn.resourcemanager.am.max-attempts确定，默认为2。

- mapreduce.am.max-attempts：表示mrAppMaster失败最大次数

公众号：旧时光大数据

- yarn.resourcemanager.am.max-attempts：表示在YARN中运行的应用程序失败最大次数。

所以如果要设置mrAppMaster最大失败次数，这两个都需要设置。

在ApplicationMaster向resourceManager定期发送心跳，当ResourceManager检查到ApplicationMaster失败的时候，ResourceManager会在新的NodeManager开启新的ApplicationMaster实例。如果是mrAppMaster，则会使用作业历史来恢复作业的运行状态，不必重新运行，由yarn.app.mapreduce.am.job.recovery.enable来控制该功能。

MapReduce客户端向mrAppMaster来轮询进度报告，如果mrAppMaster失败了，则客户端通过询问ResourceManager会定位新的mrAppMaster实例。在整个MapReduce任务作业初始化的时候，客户端会向ResourceManager询问并缓存mrAppMaster地址。

`NodeManager运行失败`

当NodeManager由于奔溃或者非常缓慢运行而失败，会停止向ResourceManager发送心跳信息。则如果10分钟内（由yarn.resourcemanager.nm.liveness-monitor.expiry-interval-ms来设置，以ms为单位），ResourceManager会停止通知发送的NodeManager，并将起从自己的节点池中移除。

在失败的NodeManager上的任务或者ApplicationMaster将由上面的机制恢复。对于曾经在失败的NodeManager运行并且成功的Map Task，如果属于为完成的作业，则ApplicationMaster则会重新分配资源重新运行，因为输出结果在失败的NodeManager的本地文件系统中，Reduce任务可能无法访问到。

如果在一个NodeManager中，任务失败次数过多，即使自己并没有失败过，则ApplicationMaster则会尽量将任务调度到其他的NodeManager上。失败次数由mapreduce.job.maxtaskfailures.per.tracker设置。

`ResourceManager运行失败`

在YARN中，ResourceManager失败是个致命的问题，如果失败，任何任务和作业都无法启动。在默认配置中，ResourceManager是单点故障。为了获得高可用（HA），我们需要配置一对ResourceManager，在主ResourceManager失败后，备份ResourceManager可以继续运行。

将所有的ApplicationMaster的运行信息保存到一个高可用的状态存储中（由ZooKeeper或者HDFS备份），这样备份ResourceManager就可以恢复出失败的ResourceManager状态。当新的ResourceManager从存储区读取ApplicationMaster，然后在集群中重启所有ApplicationMaster。这个行为不会计入到ApplicationMaster尝试。

ResourceManager在主备切换由故障转移器（failover controller）处理。默认情况，failover controller自动工作，由ZooKeeper的Leader选举，保证同一时刻只有一个主ResourceManager。不同于HDFS的HA，该failover controller不必是单独的进程，而是嵌入ResourceManager中。

YARN高可用

问过的一些公司：蘑菇街

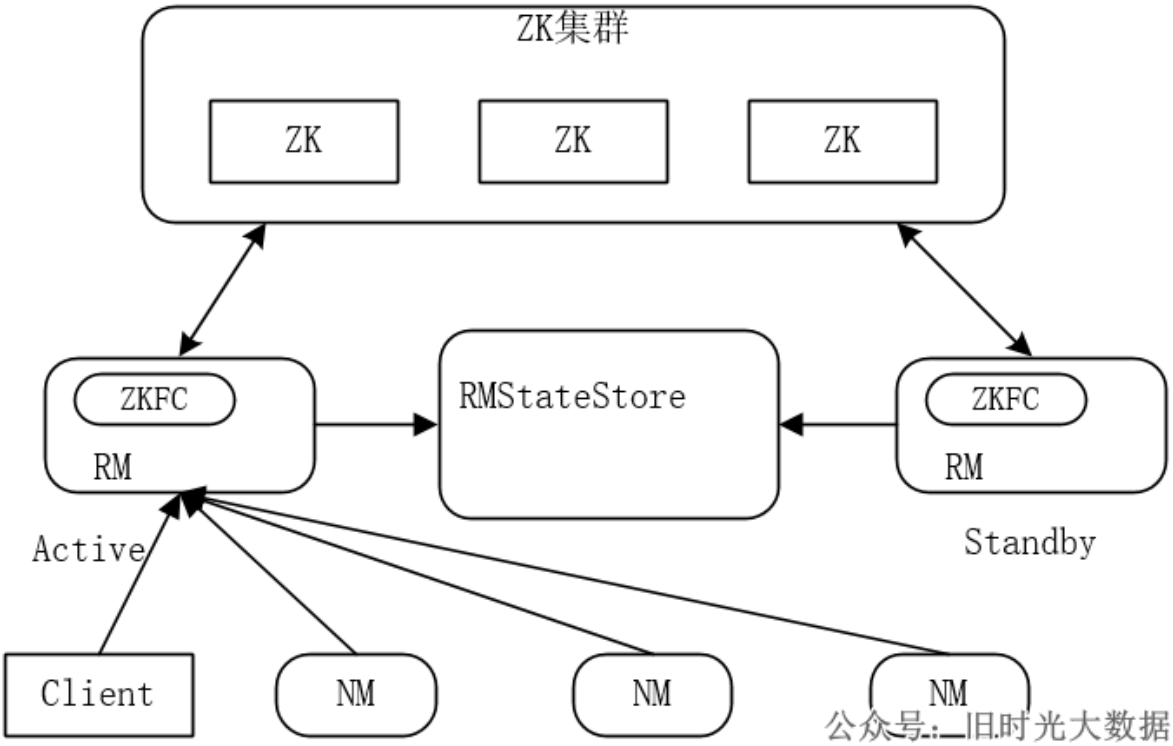
参考答案：

ResourceManager(RM)负责资源管理，并调度应用作业。在Hadop2.4之前RsourceManage是单点的，容易产生单点故障。HA 提供活动和备用的RM，解决了一个RM单点故障问题。

ResourceManager存在单点故障，基于Zookeeper实现HA，通常任务失败后，RM将失败的任务告诉AM，RM负责任务的重启，AM来决定如何处理失败的任务。RMAppMaster会保存已经运行完成的Task，启后无需重新运行。

HA架构图

公众号：旧时光大数据



公众号：旧时光大数据

集群概述

RM： ResourceManage:r一个集群只有一个active状态的,负责整个集群的管理和调度

- 处理客户端请求
- 启动监控ApplicationMaster(AM,一个作业对应一个)
- 监控NM
- 系统资源的分配和调度

NM：负责单个节点的资源管理和使用以及task运行

- 定期想RM汇报本节点的资源使用情况和container运行情况
- 接收处理RM对container的启停各种命令
- 单节点资源管理和任务管理

ZK

在ZooKeeper上会有一个/yarn-leader-election/yarn1的锁节点，所有的ResourceManager在启动的时候，都会去竞争写一个Lock子节点：/yarn-leader-election/yarn1/ActiveBreadCrumb，该节点是临时节点。ZooKeepr能够为我们保证最终只有一个ResourceManager能够创建成功。创建成功的那个ResourceManager就切换为Active状态，没有成功的那些ResourceManager则切换为Standby状态。

ZKFC

是RM里面的一个线程，在HDFS HA中，zkfc是一个独立的进程。作用 是监控RM的健康状态，并执行选举作用。

RMStateStore

RM会把job的信息存放在zookeeper的/rmstore目录下，active RM会向这个目录写app的信息。当active RM挂掉之后，standby RM会通过zkfc切换为active状态，然后从zookeeper的/rmstore目录下读取相应的作业信息。重新构建作业的内存信息，启动内部服务，开始接受NM的心跳信息，构建集群的资源信息，并且接受客户端的作业提交请求

YARN调度器

可回答：1）YARN有哪些调度策略？2）Hadoop中有哪些调度器；3）了解YARN哪些调度

公众号：旧时光大数据

问过的一些公司：转转，滴滴，作业帮，大华(2021.07)， soul(2021.09)，陌陌(2021.10)

参考答案：

目前，Hadoop 作业调度器主要有三种：FIFO、容量（Capacity Scheduler）和公平（Fair Scheduler）。Apache Hadoop3.1.3 默认的资源调度器是 Capacity Scheduler。

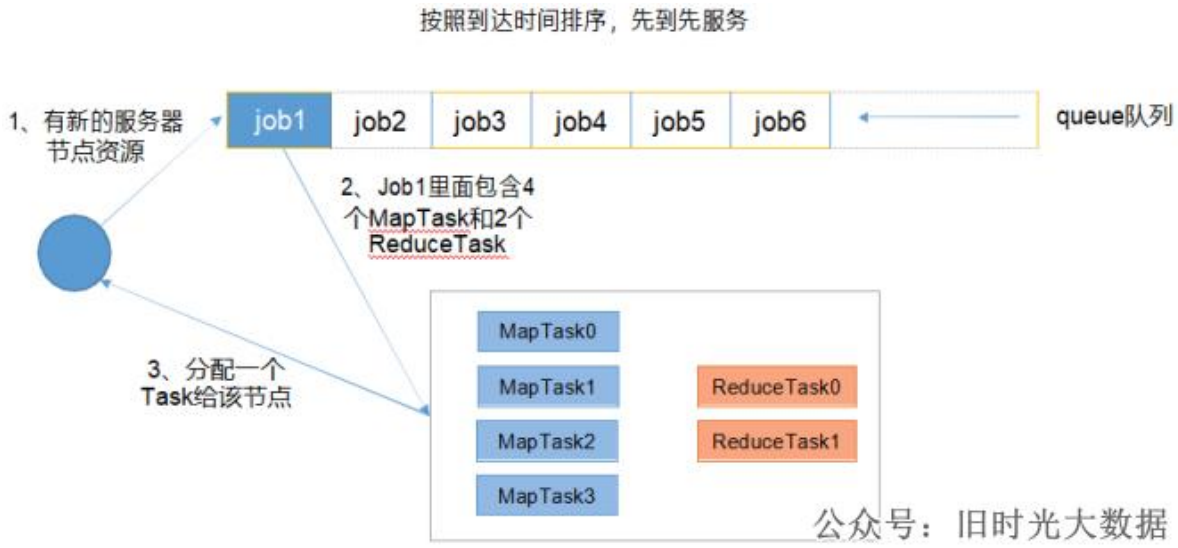
CDH 框架默认调度器是 Fair Scheduler。

具体设置详见：yarn-default.xml 文件

```
<property>
    <description>The class to use as the resource scheduler.</description>
    <name>yarn.resourcemanager.scheduler.class</name>
  <value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler</value>
</property>

```

1、先进先出调度器（FIFO）

FIFO调度器（First In First Out）：单队列，根据提交作业的先后顺序，先来先服务。



优点：简单易懂

缺点：不支持多队列，生成环境很少使用

2、容量调度器（Capacity Scheduler）

Capacity Scheduler是Yahoo开发的多用户调度器

多队列：每个队列可配置一定的资源量，每个队列采用FIFO调度策略

容量保证：管理员可为每个队列设置资源最低保证和资源使用上限

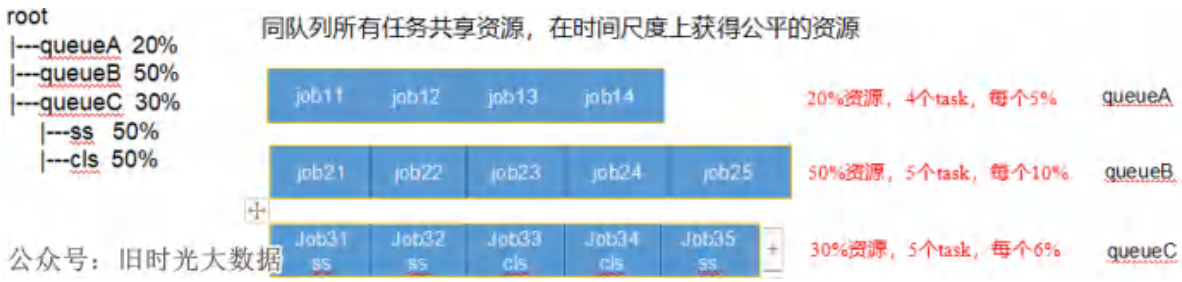
灵活性：如果一个队列中的资源有剩余，可以暂时共享给那些需要资源的队列，而一旦该队列有新的应用程序提交，则其它队列借调的资源会归还给该队列

公众号：旧时光大数据

多租户：

- 支持多用户共享集群和多应用程序同事运行
- 为了防止同一用户的作业独占队列中的资源，该调度器会对同一用户提交的作业所占资源进行限定

3、公平调度器（Fair Scheduler）



与容量调度器相同点

- 多队列：支持多队列多作业
- 容量保证：管理员可为每个队列设置资源最低保证和资源使用上线
- 灵活性：如果一个队列中的资源有剩余，可以暂时共享给那些需要资源的队列，而一旦该队列有新的应用程序中提交，则其它队列借调的资源会归还给该队列
- 多用户：支持多用户共享集群和多应用程序同事运行；为了防止同一用户的作业独占队列总的资源，该调度器会对同一用户提交的作业所占资源进行限定

与容量调度器不同点

- 核心调度策略不同
    - 容量调度器：优先选择资源利用率低的队列
    - 公平调度器：有限选择对资源的缺额比例大的

每个队列可以单独设置资源分配方式

- 容量调度器：FIFO、 DRF
- 公平调度器：FIFO、 FAIR、 DRF

YARN中Container是如何启动的?

问过的一些公司：海康威视

参考答案：

1、ApplicationMaster的主要逻辑

AM与NM通信

AM与NM们通过NMClientAsync通信，后者需要调用方提供一个回调类，NM会在合适的时机调用回调类中的方法来通知AM。回调类被AM实现为NMCallbackHandler，其中最重要的两个函数是：

- onContainerStarted()，当NM新启动了Containers时，会调用改方法，把Container列表传给它。
- onContainerStopped()，当NM停止了一些Containers时，会调用改方法，把 Container 列表传给它。

AM与RM通信

AM与RM通过AMRMClientAsync通信。

首先，通过 AMRMClientAsync.registerApplicationMaster() 向 RM 注册自己。

公众号：旧时光大数据

然后AM开始提交对Container的需求，在申请到需要数量的Container之前，先调用setupContainerAskForRM()设置对Container的具体需求（优先级、资源等），然后调用AMRMClientAsync.addContainerRequest()把需求提交给RM，最终该方法会把需求存到一个集合(AMRMClient.ask)里面。

AMRMClientAsync同样需要调用方提供一个回调类，AM实现为RMCallbackHandler。这个回调类主要实现了两个方法：

- onContainersAllocated()，获得新申请的 Container，创建一个新线程，设置ContainerLaunchContext，最终调用NMClientAsync.startContainerAsync() 来启动 Container。
- onContainersCompleted()，检查已完成的Container的数量是否达到了需求，没有的话，继续添加需求。

AM的三个主流程

总结上面说的，AM有三个主要流程与Container的创建密切相关：

- 提交需求，通过心跳，把需求发送给RM；
- 获取Container，通过心跳，拿到申请好的Container；
- 每申请到一个Container ，与 NM 通信，启动这个Container；

分析清楚了这三个主流程，也就清楚了 YARN Container 的启动逻辑。

Application 与 ResourceManager 的心跳

再看RM这边，在AM向RM注册时，RM最终会生成一个代表这个APP的实例，我们先不分析注册的具体过程，只要知道在我们的情景下，最终是生成了一个FicaSchedulerApp。

AM与RM进行心跳，发送的信息中含有：

- AM告诉RM两个信息： a) 自己对Container的要求，b) 已经用完的待回收的Container列表。
- RM给AM的回应：a) 新申请的 Container，b) 已经完成的 Container 的状态。

ApplicationMasterService是RM的一个组成部分。RM启动时，会初始化这个服务，并根据配置，把相应的调度器YarnScheduler传进来。它实现了ApplicationMasterProtocol接口，负责对来自AM的 RPC 请求进行回应。在我们的情景中， ApplicationMasterService.allocate() 方法会被调用，核心逻辑是：

- 触发 RMappAttemptStatusupdateEvent 事件。
- 调用 YarnScheduler.allocate() 方法，把执行的结果封装起来返回。YarnScheduler 是与调度器通信的接口。所以，最后调用的是具体调度器的 allocate() 方法。

我们使用的是 FIFO 调度器，FifoScheduler.allocate() 方法的主要做两件事情：

- 调用FicaSchedulerApp.updateResourceRequests()更新APP（指从调度器角度看的APP) 的资源需求。
- 通过FicaSchedulerApp.pullNewlyAllocatedContainersAndNMTokens()把FicaSchedulerApp.newlyAllocatedContainers这个 List 中的Container取出来，封装后返回。

FicaSchedulerApp.newlyAllocatedContainers 这个数据结构中存放的，正是最近申请到的 Container 。那么，这个 List 中的元素是怎么来的呢，这要从 NM 的心跳说起。

NodeManager与ResourceManager的心跳

NM 需要和 RM 进行心跳，让 RM 更新自己的信息。心跳的信息包含：

- Request(NM->RM) : NM 上所有 Container 的状态；
- Response(RM->NM) : 已待删除和待清理的 Container 列表

NM 启动时会向RM注册自己，RM生成对应的RMNode结构，代表这个NM ，存放了这个NM的资源信息以及其他一些统计信息。

公众号：旧时光大数据

负责具体心跳的，在NM这边是NodeStatusUpdater服务，在RM那边则是ResourceTrackerService服务。心跳的信息包括这个NM的状态，其中所有Container的状态等。

心跳最终通过RPC调用到了ResourceTrackerService.nodeHeartbeat() 。其核心逻辑就是触发一个RMNodeStatusEvent(RMNodeEventType.STATUS_UPDATE) 事件，这个事件由 NM 注册时生成的RMNode处理。

RMNode接收RMNodeStatusEvent(RMNodeEventType.STATUS_UPDATE) 消息，更新自己的状态机，然后调用 StatusUpdateWhenHealthyTransition.transition ，该方法从参数中获得这个NM所有的Container的信息，根据其状态分成两组：a) 刚申请到还未使用的，b) 运行完毕需要回收的，这两组 Container 的信息存放在 RMNode 的一个队列中。接着，发出一个消息：NodeUpdateSchedulerEvent（SchedulerEventType.NODE_UPDATE） 。这个消息，由调度器处理。

ResourceManager处理 `NODE_UPDATE` 消息

RM 接收到 NM 的心跳后，会发出一个 SchedulerEventType.NODE_UPDATE 的消息，改消息由调度器处理。FifoScheduler 接收到这个消息后，调用了 FifoScheduler.nodeUpdate() 方法。与 Container 申请相关的主要逻辑如下：

获取已申请到的

从 RMNode 中获取出那些「刚申请还未使用」的 Container （NM 与 RM 心跳是获得），发出消息：RMContainerEventType.LAUNCHED，该消息由 RMContainer 处理；

回收已完成的

从 RMNode 中获取出那些「已经使用完待回收」的 Container，进行回收（具体回收过程略）；

申请新的

在这个 NM 上申请新的 Container：

通过 FicaSchedulerApp.getResourceRequest() 拿到资源请求（ResourceRequest）
计算可申请的资源，调用 FicaSchedulerApp.allocate()，根据传进来的参数，封装出一个 RMContainer 添加到 newlyAllocatedContainers 中。然后触发事件 RMContainerEventType.START。该事件之后会由 RMContainer 处理。

调用 FicaSchedulerNode.allocateContainer()和RMContainer对RMContainerEventType事件进行处理处理：

RMContainerEventType.START : 状态从 NEW 变为 ALLOCATED，最终触发事件 RMAppAttemptEvent(type=CONTAINER_ALLOCATED)， 改事件由 RMAppAttemptImpl 处理。
RMContainerEventType.LAUNCHED : 状态从 ACQUIED 变为 RUNNING 。

RMAppAttemptImpl对RMAppAttemptEvent事件进行处理，该事件告诉就是告诉AppAttempt ，你这个APP有Container申请好了，AppAttempt 检查自己的状态，如果当前还没有运行AM ，就把这个Container拿来运行AM。

到此，我们已经理清楚了FicaSchedulerApp.newlyAllocatedContainers中元素的来源，也就理清楚了，AM 与 RM 心跳中获得的那些「新申请」的 Container 的来源。

ApplicationMaster 与 NodeManager 通信启动 Container

关于“AM的三个主流程”，上面已经讲过了。

基于上面的分析，第1，2两个流程已经清楚。下面我们来具体看看 NM 具体是怎么启动一个 Container 的。

AM 设置好 ContainerLaunchContext , 调用 NMClientAsync.startContainerAsync() 启动Container。

NMClientAsync 中有一个名叫 events 的事件队列，同时，NMClientAsync 还启动这一个线程，不断地从 events 中取出事件进行处理。

公众号：旧时光大数据

startContainerAsync() 方法被调用时，会生成一个 ContainerEvent(type=START_CONTAINER) 事件放入 events 队列。对于这个事件，处理逻辑是调用 NMClient.startContainer() 同步地启动 Container ，然后调用回调类中的 onContainerStarted() 方法。

NMClient 最终会调用 ContainerManagementProtocol.startContainers() ，以 Google Protocol Buffer 格式，通过 RPC 调用 NM 的对应方法。NM 处理后会返回成功启动的 Container 列表。

NodeManager 中启动 Container

ContainerManagerImpl

NM 中负责响应来自 AM 的 RPC 请求的是 ContainerManagerImpl ，它是 NodeManager 的一部分，负责 Container 的管理，在 Nodemanager 启动时，该服务被初始化。该类实现了接口 ContainerManagementProtocol ，接到 RPC 请求后，会调用 ContainerManagerImpl.startContainers() 。改函数的基本逻辑是：

- 首先进行 APP 的初始化（如果还没有的话），生成一个 ApplicationImpl 实例，然后根据请求，生成一堆 ContainerImpl 实例
- 触发一个新事件：ApplicationContainerInitEvent ，之前生成的 ApplicationImpl 收到改事件，又出发一个 ContainerEvent(type=INIT_CONTAINER) 事件，这个事件由 ContainerImpl 处理
- ContainerImpl 收到事件， 更新状态机，启动辅助服务，然后触发一个新事件 ContainersLaucherEvent(type=LAUNCH_CONTAINER) ，处理这个事件的是 ContainersLauncher 。

ContainerLauncher 是 ContainerManager 的一个子服务，收到 ContainersLaucherEvent(type=LAUNCH_CONTAINER) 事件后，组装出一个 ContainerLaunch 类并使用 ExecutorService 执行。

ContainerLaunch 类负责一个 Container 具体的 Lanuch 。基本逻辑如下：

- 设置运行环境，包括生成运行脚本，Local Resource ，环境变量，工作目录，输出目录等
- 触发新事件 ContainerEvent(type=CONTAINER_LAUNCHED)，该事件由 ContainerImpl 处理。
- 调用 ContainerExecutor.launchContainer() 执行 Container 的工作，这是一个阻塞方法。
- 执行结束后，根据执行的结果设置 Container 的状态。

ContainerExecutor

ContainerExecutor 是 NodeManager 的一部分，负责 Container 中具体工作的执行。该类是抽象类，可以有不同的实现，如 DefaultContainerExecutor ， DockerContainerExecutor ， LinuxContainerExecutor 等。根据 YARN 的配置，NodeManager 启动时，会初始化具体的 ContainerExecutor 。

ContainerExecutor 最主要的方法是 launchContainer() ，该方法阻塞，直到执行的命令结束。

DefaultContainerExecutor 是默认的 ContainerExecutor ， 支持 Windows 和 Linux 。它的 launchContainer() 的逻

- 创建 Container 需要的目录
- 拷贝 Token、运行脚本到工作目录
- 做一些脚本的封装，然后执行脚本，返回状态码

至此，Container 在 NM 中已经启动，AM 中 NMCallback 回调类中的 onContainerStarted() 方法被调用。

YARN的改进之处，Hadoop 3.x相对于Hadoop 2.x?

问过的一些公司：字节

参考答案：

YARN Timeline Service版本更新到v.2

公众号：旧时光大数据

本版本引入了Yarn时间抽服务v.2，主要用于解决2大挑战：改善时间轴服务的可伸缩性和可靠性，通过引入流和聚合增强可用性。

YARN Timeline Service v.2 alpha 1可以让用户和开发者测试以及反馈，以便使得它可以替换现在的Timeline Service v.1.x。

YARN监控

问过的一些公司：美团

参考答案：

配置 yarn-site.xml 开启日志聚合

日志聚集是YARN提供的日志中央化管理功能，它能将运行完成的Container/任务日志上传到HDFS上，从而减轻NodeManager负载，且提供一个中央化存储和分析机制。默认情况下，Container/任务日志存在在各个NodeManager上

```
<!-- Site specific YARN configuration properties -->
<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>

  <!-- 开启日志聚合 如果没有设置的话，会显示3个目录 -->
  <property>
    <name>yarn.log-aggregation-enable</name>
    <value>true</value>
  </property>
</configuration>

```

配置 mapred-site.xml

```
<property>
    <!-- 表示提交到hadoop中的任务采用yarn来运行，要是已经有该配置则无需重复配置 -->
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
</property>
<property>
  <!--日志监控服务的地址，一般填写为namenode机器地址 -->
    <name>mapreduce.jobhistroy.address</name>
    <value>master:10020</value>
</property>
<property>
    <name>mapreduce.jobhistroy.webapp.address</name>
    <value>master:19888</value>
</property>

```

重启YARN

开启日志监控服务进程

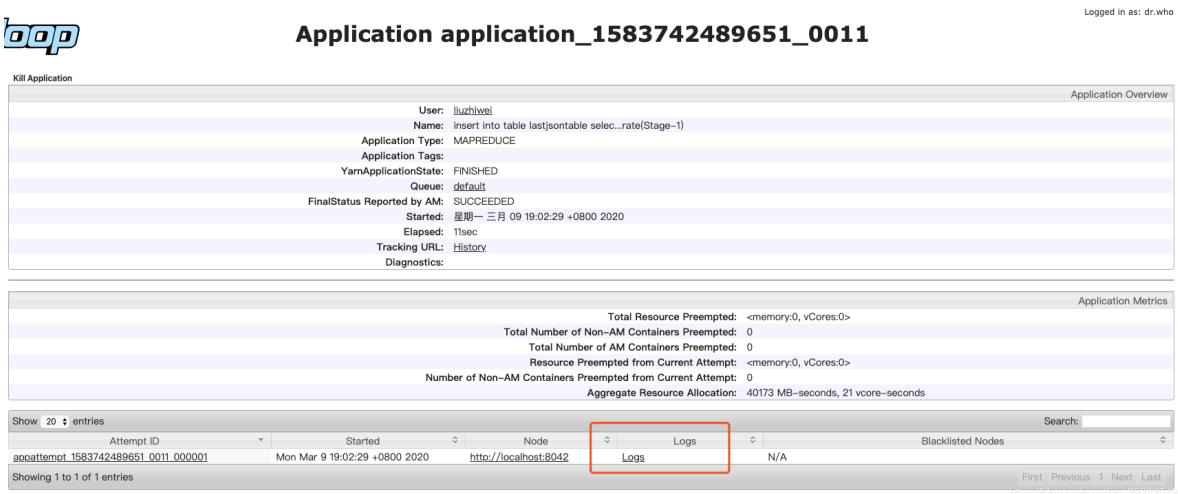
在nodenode机器上执行 `sbin/mr-jobhistory-daemon.sh start historyserver` 命令，执行完成后使用jps命令查看是否启动成功，若启动成功则会显示出JobHistoryServer服务。

公众号：旧时光大数据

最好将`yarn-site.xml 的 yarn.log.server.url`也配置上

```
<property>
   <name>yarn.log.server.url</name>
   <value>http://localhost:19888/jobhistory/logs</value>
</property>

```

不然的话这个链接跳转不到



Zookeeper面试题

介绍下Zookeeper是什么?

可回答：谈谈你对Zookeeper的理解

问过的一些公司：京东x2，字节，美团x2，蘑菇街，映客直播

参考答案：

Zookeeper是一个开源的分布式的，为分布式应用提供协调服务的Apache项目。

Zookeeper从设计模式角度来理解，是一个基于观察者模式设计的分布式服务管理框架，它负责存储和管理大家都关心的数据，然后接受观察者的注册，一旦这些数据的状态发生了变化，Zookeeper就负责通知已经在Zookeeper上注册的那些观察者做出相应的反应。

Zookeeper提供的服务包括：统一命名服务、统一配置管理、统一集群管理、服务器节点动态上下线、软负载均衡等。它致力于为那些高吞吐的大型分布式系统提供一个高性能、高可用、且具有严格顺序访问控制能力的分布式协调服务。

特性

1）顺序一致性：从一个客户端发起的事务请求，最终都会严格按照其发起顺序被应用到Zookeeper中；对于来自客户端的每个更新请求，Zookeeper都会分配一个全局唯一的递增ID(zxid)，这个ID反映了所有事务请求的先后顺序。

2）原子性：所有事务请求的处理结果在整个集群中所有机器上都是一致的

3）最终一致性：所有客户端看到的服务端数据模型都是一致的；

4）可靠性：一旦服务端成功应用了一个事务，则其引起的改变会一直保留，直到被另外一个事务所更改,如果消息被到一台服务器接受，那么它将被所有的服务器接受。

5）实时性：一旦一个事务被成功应用后，Zookeeper可以保证客户端立即可以读取到这个事务变更后的最新状态的数据。

公众号：旧时光大数据

6）等待无关（wait-free）：慢的或者失效的client不得干预快速的client的请求，使得每个client都能有效的等待。

由于Zookeeper的所有更新和删除都是基于事务的，所以其在读多写少的应用场景中有着很高的性能表现。

ZooKeeper将数据存全量储在内存中以保持高性能，并通过服务集群来实现高可用。

可结合下一题一起回答

Zookeeper有什么作用？优缺点？有什么应用场景？

可回答：你觉得Zookeeper比较重要的功能

问过的一些公司：阿里x3，京东x3，阿里，字节，字节(2021.10)，美团x3，蘑菇街x2，深信服，头条，vivo，ebay

参考答案：

作用

Zookeeper作用包括存储数据（文件系统）和监听（监听通知机制）

优点

1）分布式协调过程简单

2）同步：zk高度同步，这意味着服务器进程之间既存在互斥又存在合作，同步有助于Apache HBase进行配置管理。

3）有序消息：zk跟踪一个数字，表示每个更新的顺序，保证消息有序

4）序列化：根据具体规则，zk对数据进行编码。 另外，它还可确保我们的应用程序始终如一地运行。但是，在MapReduce中，我们使用此方法（序列化）来协调队列以执行正在运行的线程

5）速度：在读请求多的情况下，能以很快的速度运行

6）可扩展性：此外，可以通过部署更多机器来加强zk的性能

7）有序性有何优势？：众所周知，zk中的消息是有序的。所以，为了实现更高级别的抽象，需要有序性。 这就是有序性对我们有利的方式

8）快：在读多的情况下，zk会非常快

9）可靠性：zk非常可靠，因为一旦zk更新了，更新后的数据会一直保持，直到被覆盖更新

10）原子性：zk只有两种情况，要么全部成功，要么全部失败，没有中间状态的情况

11）实时性：zk保证在一定时间段内，客户端最终一定能从服务器上读到最新的数据状态

缺点

1）增加新的zk服务器时可能导致数据丢失

在现有服务器中，当新zk服务器数量超过zk服务中已存在的数量时数据会丢失。 同时，向zk服务发出Start命令，新服务器可能形成仲裁

2）不能迁移

在没有用户干预的情况下，zk服务器无法从版本3.4迁移到3.3，然后再迁移到3.4。

3）节点数（其实集群中只要存活数过半即对外可用，但是会有脑裂情况发生）

公众号：旧时光大数据

要求3或5个这样奇数个zk节点（要求奇数是为了保证选举的正常进行因为leader选举要求 可用节点数量 > 总节点数/2，防止脑裂造成集群不可用。同时在容错能力相同的情况下，奇数个节点更节省资源）

4）机架感知复制

目前，它不支持机架放置和感知

5）缩容

不支持减少pod的数量，以防止意外数据丢失

6）磁盘变更

不支持在初始部署后更改volume要求，以防止重新分配意外丢失数据

7）虚拟网络

当服务部署在虚拟网络上时，如果没有完全重新安装，服务可能无法切换到主机网络。 另外，对于尝试从主机切换到虚拟网络，它们是相同的情况

8）Kerberos

在虚拟网络上，它目前不支持启用Kerberos

9）支持有限

对跨群集方案的支持非常有限。 但是，没有CP系统会一直支持跨集群。 虽然我们可以说consul似乎在这方面做得更好

应用场景

1）数据的发布/订阅

数据的发布/订阅系统，通常也用作配置中心。在分布式系统中，你可能有成千上万个服务节点，如果想要对所有服务的某项配置进行更改，由于数据节点过多，你不可逐台进行修改，而应该在设计时采用统一的配置中心。之后发布者只需要将新的配置发送到配置中心，所有服务节点即可自动下载并进行更新，从而实现配置的集中管理和动态更新。 Zookeeper通过Watcher机制可以实现数据的发布和订阅。分布式系统的所有的服务节点可以对某个ZNode注册监听，之后只需要将新的配置写入该ZNode，所有服务节点都会收到该事件。

2）命名服务

在分布式系统中，通常需要一个全局唯一的名字，如生成全局唯一的订单号等，Zookeeper可以通过顺序节点的特性来生成全局唯一ID，从而可以对分布式系统提供命名服务。

3）Master选举

分布式系统一个重要的模式就是主从模式(Master/Salves)，Zookeeper可以用于该模式下的Matser选举。可以让所有服务节点去竞争性地创建同一个ZNode，由于Zookeeper不能有路径相同的ZNode，必然只有一个服务节点能够创建成功，这样该服务节点就可以成为Master节点。

4）分布式锁

可以通过Zookeeper的临时节点和Watcher机制来实现分布式锁，这里以排它锁为例进行说明：分布式系统的所有服务节点可以竞争性地去创建同一个临时ZNode，由于Zookeeper不能有路径相同的ZNode，必然只有一个服务节点能够创建成功，此时可以认为该节点获得了锁。其他没有获得锁的服务节点通过在该ZNode上注册监听，从而当锁释放时再去竞争获得锁。锁的释放情况有以下两种： 当正常执行完业务逻辑后，客户端主动将临时ZNode删除，此时锁被释放； 当获得锁的客户端发生宕机时，临时ZNode会被自动删除，此时认为锁已经释放。 当锁被释放后，其他服务节点则再次去竞争性地进行创建，但每次都只有一个服务节点能够获取到锁，这就是排他锁。

5）集群管理

公众号：旧时光大数据

Zookeeper还能解决大多数分布式系统中的问题： 如可以通过创建临时节点来建立心跳检测机制。如果分布式系统的某个服务节点宕机了，则其持有的会话会超时，此时该临时节点会被删除，相应的监听事件就会被触发。 分布式系统的每个服务节点还可以将自己的节点状态写入临时节点，从而完成状态报告或节点工作进度汇报。 通过数据的订阅和发布功能，Zookeeper还能对分布式系统进行模块的解耦和任务的调度。 通过监听机制，还能对分布式系统的服务节点进行动态上下线，从而实现服务的动态扩容。

6）事务操作

在ZooKeeper中，能改变ZooKeeper服务器状态的操作称为事务操作。一般包括数据节点创建与删除、数据内容更新和客户端会话创建与失效等操作。 对应每一个事务请求，ZooKeeper都会为其分配一个全局唯一的事务ID，用 ZXID 表示，通常是一个64位的数字。每一个 ZXID对应一次更新操作， 从这些 ZXID 中可以间接地识别出 ZooKeeper 处理这些事务操作请求的全局顺序。

Zookeeper的选举策略，leader和follower的区别?

可回答：1）Zookeeper的选举过程（选举机制）；2）leader的选举是如何实现的；3）说说Zookeeper的启动过程，比如现在有五台机器，ABCDE依次启动起来，那么哪台是leader？4）Zookeeper的选主策略了解过吗

问过的一些公司：阿里，字节，字节(2021.10)，去哪儿，网易，贝壳，京东，蘑菇屋，端点数据(2021.07)，华为精英计划(2021.07)，大华(2021.08)

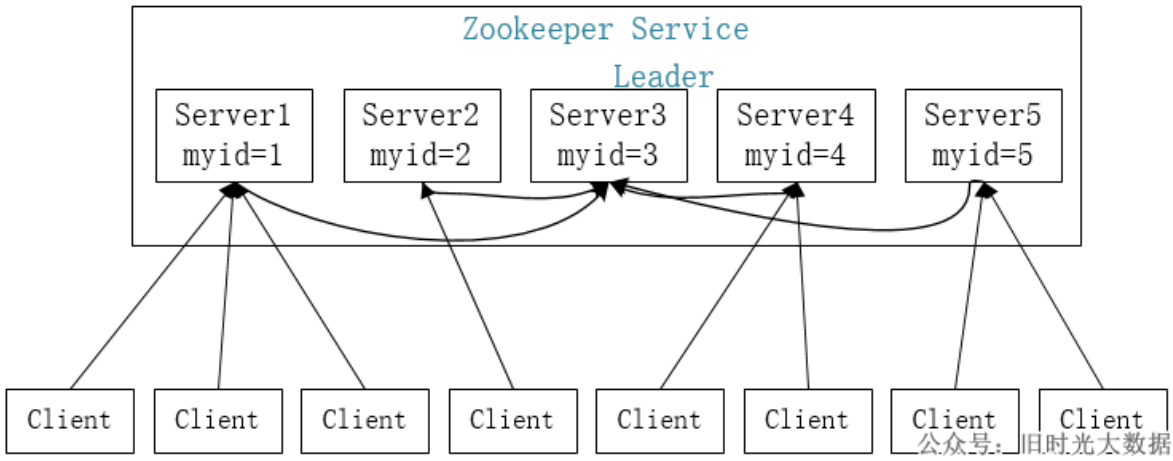
参考答案：

半数机制：集群中半数以上机器存活，集群可用。所以Zookeeper适合安装奇数台服务器。

Zookeeper虽然在配置文件中并没有指定Master和Slave。但是，Zookeeper工作时，是有一个节点为Leader，其他则为Follower，Leader是通过内部的选举机制临时产生的。

以一个简单的例子来说明整个选举的过程

假设有五台服务器组成的Zookeeper集群，它们的id从1-5，同时它们都是最新启动的，也就是没有历史数据，在存放数据量这一点上，都是一样的。假设这些服务器依序启动，来看看会发生什么。



1）服务器1启动，发起一次选举。服务器1投自己一票。此时服务器1票数一票，不够半数以上（3票），选举无法完成，服务器1状态保持为LOOKING；

2）服务器2启动，再发起一次选举。服务器1和2分别投自己一票并交换选票信息：此时服务器1发现服务器2的ID比自己目前投票推举的（服务器1）大，更改选票为推举服务器2。此时服务器1票数0票，服务器2票数2票，没有半数以上结果，选举无法完成，服务器1，2状态保持LOOKING

3）服务器3启动，发起一次选举。此时服务器1和2都会更改选票为服务器3。此次投票结果：服务器1为0票，服务器2为0票，服务器3为3票。此时服务器3的票数已经超过半数，服务器3当选Leader。服务器1，2更改状态为FOLLOWING，服务器3更改状态为LEADING；

公众号：旧时光大数据

4）服务器4启动，发起一次选举。此时服务器1，2，3已经不是LOOKING状态，不会更改选票信息。交换选票信息结果：服务器3为3票，服务器4为1票。此时服务器4服从多数，更改选票信息为服务器3，并更改状态为FOLLOWING；

5）服务器5启动，同4一样当小弟。

领导者Leader

Leader在集群中只有一个节点，可以说是老大，是zookeeper集群的中心，负责协调集群中的其他节点。从性能的角度考虑，leader可以选择不接受客户端的连接。

主要作用

1）发起与提交写请求。

所有的跟随者Follower与观察者Observer节点的写请求都会转交给领导者Leader执行。Leader接受到一个写请求后，首先会发送给所有的Follower，统计Follower写入成功的数量。当有超过半数的Follower写入成功后，Leader就会认为这个写请求提交成功，通知所有的Follower commit这个写操作，保证事后哪怕是集群崩溃恢复或者重启，这个写操作也不会丢失。

2）与**Learner（Follower与Observer）**保持心跳

3）崩溃恢复时负责恢复数据以及同步数据到Learner

跟随者Follower

Follow在集群中有多个

主要作用

1）与老大Leader保持心跳连接

2）当Leader挂了的时候，经过投票后成为新的leader。leader的重新选举是由老二Follower们内部投票决定的。

3）向leader发送消息与请求

4）处理leader发来的消息与请求

介绍下Zookeeper选举算法

问过的一些公司：海康(2021.08)

参考答案：

在ZooKeeper中，提供了三种Leader选举的算法，分别是

1）LeaderElection

2）UDP版本的FastLeaderElection

3）TCP版本的FastLeaderElection

可以通过在配置文件zoo.cfg中使用electionAlg属性来指定，分别使用数字0~3来表示。0代表LeaderElection，这是一种纯UDP实现的Leader选举算法：1代表UDP版本的FastLeaderElection，并且是非授权模式；2也代表UDP版本的FastLeaderElection，但使用授权模式；3代表TCP版本的FastLeaderElection。不过从3.4.0版本开始，ZooKeeper废弃了0、1、2这三种Leader选举算法，只保留了TCP版本的FastLeaderElection选举算法。

1、Leader选举

公众号：旧时光大数据

Leader选举是保证分布式数据一致性的关键所在。当Zookeeper集群中的一台服务器出现以下两种情况之一时，需要进入Leader选举。

1）服务器初始化启动。（集群的每个节点都没有数据 → 以SID的大小为准）

2）服务器运行期间无法和Leader保持连接。（集群的每个节点都有数据 ,或者Leader 宕机→ 以ZXID 和SID 的最大值为准）

1.1 服务器启动时期的Leader选举

若进行Leader选举，则至少需要2台机器，两台的高可用性会差一些，如果Leader 宕机，就剩下一台，自己没办法选举。一般集群也是3台机器，这里选取3台机器组成的服务器集群为例。

在集群初始化阶段，当有一台服务器Server1启动时，其单独无法进行和完成Leader选举，当第二台服务器Server2启动时，此时两台机器可以相互通信，每台机器都试图找到Leader，于是进入Leader选举过程。选举过程如下

1）每个Server发出一个投票。由于是初始情况，Server1和Server2都会将自己作为Leader服务器来进行投票，每次投票会包含所推举的服务器的myid和ZXID，使用(myid, ZXID)来表示，此时Server1的投票为(1, 0)，Server2的投票为(2, 0)，然后各自将这个投票发给集群中其他机器。

2）接受来自各个服务器的投票。集群的每个服务器收到投票后，首先判断该投票的有效性，如检查是否是本轮投票、是否来自LOOKING状态的服务器。

3）处理投票。针对每一个投票，服务器都需要将别人的投票和自己的投票进行PK，PK规则如下：

- **优先检查ZXID。**ZXID比较大的服务器优先作为Leader。（**这个很重要：是数据最新原则，保证数据的完整性）**
- **如果ZXID相同，那么就比较myid。**myid较大的服务器作为Leader服务器。 **（集群的节点标识）**

对于Server1而言，它的投票是(1, 0)，接收Server2的投票为(2, 0)，首先会比较两者的ZXID，均为0。再比较myid，此时Server2的myid最大，于是更新自己的投票为(2, 0)，然后重新投票，对于Server2而言，其无须更新自己的投票，只是再次向集群中所有机器发出上一次投票信息即可。

4）统计投票。每次投票后，服务器都会统计投票信息，判断是否已经有过半机器接受到相同的投票信息，对于Server1、Server2而言，都统计出集群中已经有两台机器接受了(2, 0)的投票信息，此时便认为已经选出了Leader。

5）改变服务器状态。一旦确定了Leader，每个服务器就会更新自己的状态，如果是Follower，那么就变更为FOLLOWING，如果是Leader，就变更为LEADING。

1.2 服务器运行时期的Leader选举

在Zookeeper运行期间，Leader与非Leader服务器各司其职，即便当有非Leader服务器宕机或新加入，此时也不会影响Leader，但是一旦Leader服务器挂了，那么整个集群将暂停对外服务，进入新一轮Leader选举，其过程和启动时期的Leader选举过程基本一致。

假设正在运行的有Server1、Server2、Server3三台服务器，当前Leader是Server2，若某一时刻Leader挂了，此时便开始Leader选举。

选举过程如下：

1）**变更状态。**Leader挂后，余下的非Observer服务器都会讲自己的服务器状态变更为LOOKING，然后开始进入Leader选举过程。

2）**每个Server会发出一个投票。**在运行期间，每个服务器上的ZXID可能不同，此时假定Server1的ZXID为123，Server3的ZXID为122；在第一轮投票中，Server1和Server3都会投自己，产生投票(1, 123)，(3, 122)，然后各自将投票发送给集群中所有机器。

3）**接收来自各个服务器的投票。**与启动时过程相同。

公众号：旧时光大数据

4）**处理投票**。与启动时过程相同，此时，Server1将会成为Leader。

5）**统计投票**。与启动时过程相同。

6）**改变服务器的状态**。与启动时过程相同。

2、Leader选举算法分析

最开始也提过，在3.4.0后的Zookeeper的版本只保留了TCP版本的FastLeaderElection选举算法。当一台机器进入Leader选举时，当前集群可能会处于以下两种状态

- 集群中已经存在Leader。
- 集群中不存在Leader。

对于集群中已经存在Leader而言，此种情况一般都是某台机器启动得较晚，在其启动之前，集群已经在正常工作，对这种情况，该机器试图去选举Leader时，会被告知当前服务器的Leader信息，对于该机器而言，仅仅需要和Leader机器建立起连接，并进行状态同步即可。

而在集群中不存在Leader情况下则会相对复杂，其步骤如下

1）**第一次投票**。无论哪种导致进行Leader选举，集群的所有机器都处于试图选举出一个Leader的状态，即LOOKING状态，LOOKING机器会向所有其他机器发送消息，该消息称为投票。投票中包含了SID（服务器的唯一标识）和ZXID（事务ID），(SID, ZXID)形式来标识一次投票信息。假定Zookeeper由5台机器组成，SID分别为1、2、3、4、5，ZXID分别为9、9、9、8、8，并且此时SID为2的机器是Leader机器，某一时刻，1、2所在机器出现故障，因此集群开始进行Leader选举。在第一次投票时，每台机器都会将自己作为投票对象，于是SID为3、4、5的机器投票情况分别为(3, 9)，(4, 8)， (5, 8)。

2）**变更投票**。每台机器发出投票后，也会收到其他机器的投票，每台机器会根据一定规则来处理收到的其他机器的投票，并以此来决定是否需要变更自己的投票，这个规则也是整个Leader选举算法的核心所在，其中术语描述如下

- **vote_sid**：接收到的投票中所推举Leader服务器的SID。
- **vote_zxid**：接收到的投票中所推举Leader服务器的ZXID。
- **self_sid**：当前服务器自己的SID。
- **self_zxid**：当前服务器自己的ZXID。

每次对收到的投票的处理，都是对(vote_sid, vote_zxid)和(self_sid, self_zxid)对比的过程。

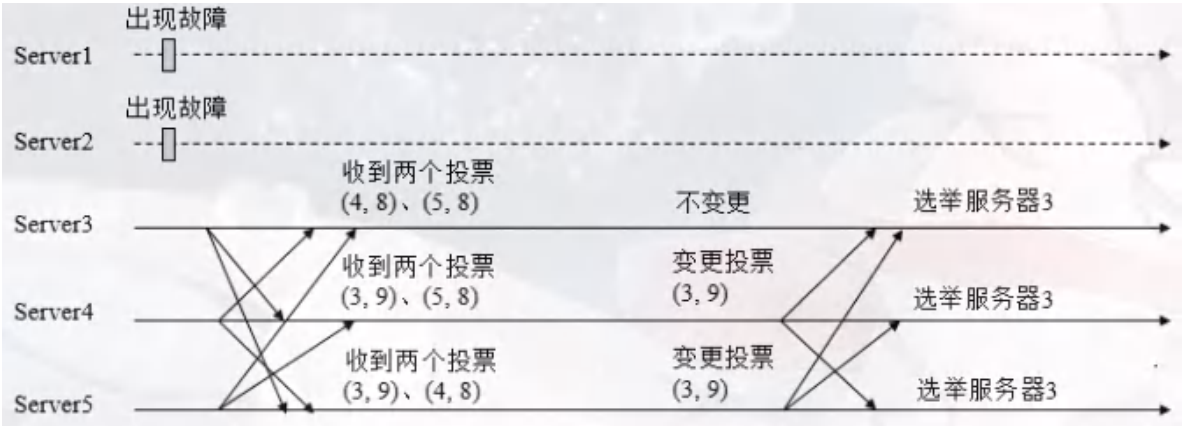
规则一：如果vote_zxid大于self_zxid，就认可当前收到的投票，并再次将该投票发送出去。

规则二：如果vote_zxid小于self_zxid，那么坚持自己的投票，不做任何变更。

规则三：如果vote_zxid等于self_zxid，那么就对比两者的SID，如果vote_sid大于self_sid，那么就认可当前收到的投票，并再次将该投票发送出去。

规则四：如果vote_zxid等于self_zxid，并且vote_sid小于self_sid，那么坚持自己的投票，不做任何变更。

结合上面规则，给出下面的集群变更过程。



3）**确定Leader**。经过第二轮投票后，集群中的每台机器都会再次接收到其他机器的投票，然后开始统计投票，如果一台机器收到了超过半数的相同投票，那么这个投票对应的SID机器即为Leader。此时Server3将成为Leader。

由上面规则可知，通常那台服务器上的数据越新（ZXID会越大），其成为Leader的可能性越大，也就越能够保证数据的恢复。如果ZXID相同，则SID越大机会越大。

3、Leader选举实现细节

3.1 服务器状态

服务器具有四种状态，分别是LOOKING、FOLLOWING、LEADING、OBSERVING。

LOOKING：寻找Leader状态。当服务器处于该状态时，它会认为当前集群中没有Leader，因此需要进入Leader选举状态。

FOLLOWING：跟随者状态。表明当前服务器角色是Follower。

LEADING：领导者状态。表明当前服务器角色是Leader。

OBSERVING：观察者状态。表明当前服务器角色是Observer。

3.2 投票数据结构

每个投票中包含了两个最基本的信息，所推举服务器的SID和ZXID，投票（Vote）在Zookeeper中包含字段如下

id：被推举的Leader的SID。

zxid：被推举的Leader事务ID。

electionEpoch：逻辑时钟，用来判断多个投票是否在同一轮选举周期中，该值在服务端是一个自增序列，每次进入新一轮的投票后，都会对该值进行加1操作。

peerEpoch：被推举的Leader的epoch。

state：当前服务器的状态。

Zookeeper的节点类型有哪些？分别作用是什么？

可回答：说下Zookeeper的临时节点和永久节点

问过的一些公司：字节，eBay，快手，蘑菇街

参考答案：

Znode有两种类型：

- 短暂（ephemeral）：客户端和服务器端断开连接后，创建的节点自己删除

公众号：旧时光大数据

- 持久（persistent）：客户端和服务器端断开连接后，创建的节点不删除

Znode有四种形式的目录节点（默认是persistent ）

1）持久化目录节点（PERSISTENT）

客户端与zookeeper断开连接后，该节点依旧存在

2）持久化顺序编号目录节点（PERSISTENT_SEQUENTIAL）

客户端与zookeeper断开连接后，该节点依旧存在，只是Zookeeper给该节点名称进行顺序编号

3）临时目录节点（EPHEMERAL）

客户端与zookeeper断开连接后，该节点被删除

4）临时顺序编号目录节点（EPHEMERAL_SEQUENTIAL）

客户端与zookeeper断开连接后，该节点被删除，只是Zookeeper给该节点名称进行顺序编号

创建znode时设置顺序标识，znode名称后会附加一个值，顺序号是一个单调递增的计数器，由父节点维护

在分布式系统中，顺序号可以被用于为所有的事件进行全局排序，这样客户端可以通过顺序号推断事件的顺序

Zookeeper的节点数怎么设置比较好?

可回答：Zookeeper集群一般设置多少个节点，允许几个节点挂

问过的一些公司：bigo，华为精英计划(2021.07)

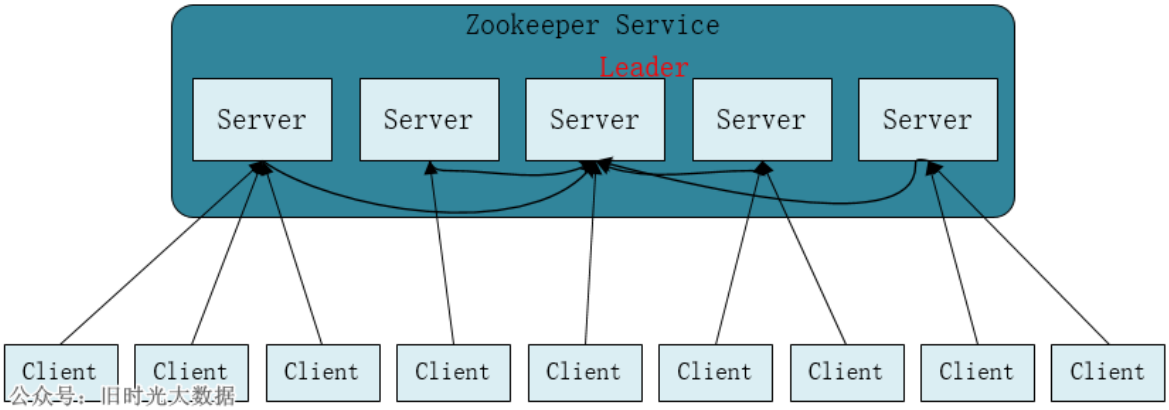
参考答案：

2n+1，半数机制。一般是3个或者3个以上节点。

Zookeeper架构

问过的一些公司：字节

参考答案：



Zookeeper本身可以是单机模式，也可以是集群模式，为了Zookeeper本身不出现单点故障，通常情况下使用集群模式，而且是master/slave模式的集群

Zookeeper集群角色

公众号：旧时光大数据

`Leader（领导者）`

Leader不直接接收client的请求，但接受由其他Follower和Observer转发过来的client请求。此外，Leader还负责投票的发起和决议，即时更新状态和数据。

`Follower（跟随者）`

Follower角色接手客户端请求并返回结果，参与Leader发起的投票和选举，但不具有写操作的权限。

`Observer（观察者）`

Observer角色接受客户端连接，将写操作转给Leader，但Observer不参与投票（即不参加一致性协议的达成），只同步Leader节点的状态，Observer角色是为集群系统扩展而生的。

`Clinet（客户端）`

连接Zookeeper集群的使用者，请求的发起者，独立于Zookeeper集群的角色。

Zookeeper的功能有哪些

可回答：Zookeeper的作用

问过的一些公司：字节，阿里，美团x2，蘑菇街

Zookeeper作用包括存储数据（文件系统）和监听（监听通知机制）

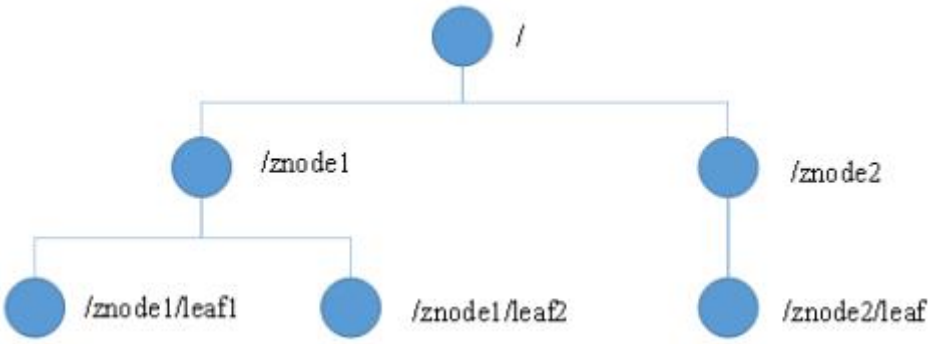
Zookeeper的数据结构（树）？基于它实现的分布式锁？基于它实现的Master选举？基于它的集群管理？Zookeeper的注册（watch）机制使用场景？

问过的一些公司：阿里，ebay

参考答案：

1、数据结构

Zookeeper数据模型的结构与Unix文件系统很类似，整体上可以看作一棵树，每个节点称作一个ZNode。每一个ZNode默认能够存储1MB的数据，每个ZNode都可以通过其路径唯一标识。



ZNode有两种类型

- 短暂（ephemeral）：客户端和服务器端断开连接后，创建的节点自己删除
- 持久（persistent）：客户端和服务器端断开连接后，创建的节点不删除

ZNode有四种形式的目录节点（默认是PERSISTENT）

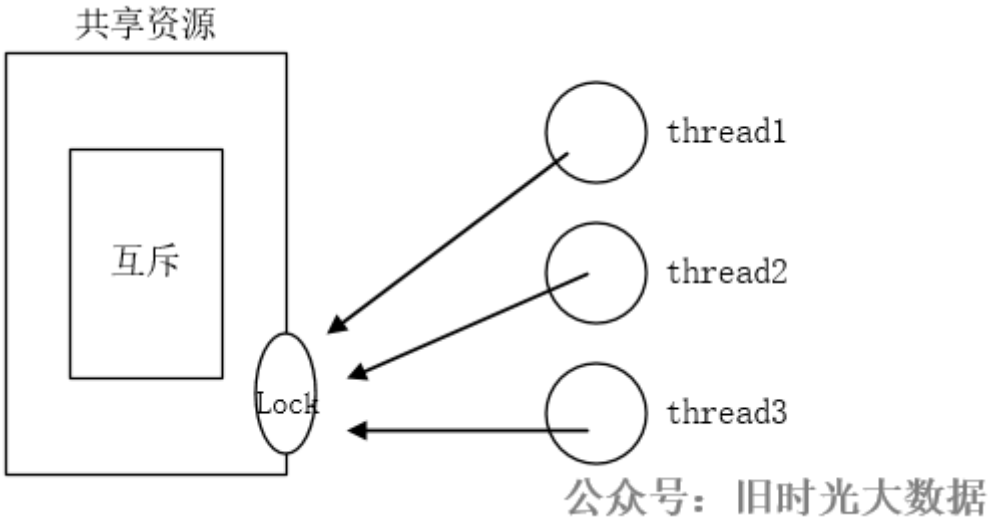
公众号：旧时光大数据

类型	描述
PERSISTENT	持久化节点
PERSISTENT_SEQUENTIAL	顺序自动编号持久化节点，这种节点会根据当前已存在的节点数自动 + 1
EPHEMERAL	临时节点，客户端session超时这类节点会被自动删除
EPHEMERAL_SEQUENTIAL	临时自动编号节点

2、分布式锁

2.1 为什么需要分布式锁

锁是多线程代码中的概念，只有当**多任务**访问同一个**互斥**的**共享资源**时才需要。如下图：



在我们进行单机应用开发，涉及并发同步的时候，我们往往采用synchronized或者Lock的方式来解决多线程间的代码同步问题，这时多线程的运行都是在同一个JVM之下。但当我们的应用是分布式集群工作的情况下，属于多JVM下的工作环境，JVM之间已经无法通过多线程的锁解决同步问题。那么就需要一种更加高级的锁机制，来处理种跨机器的进程之间的数据同步问题——这就是分布式锁。

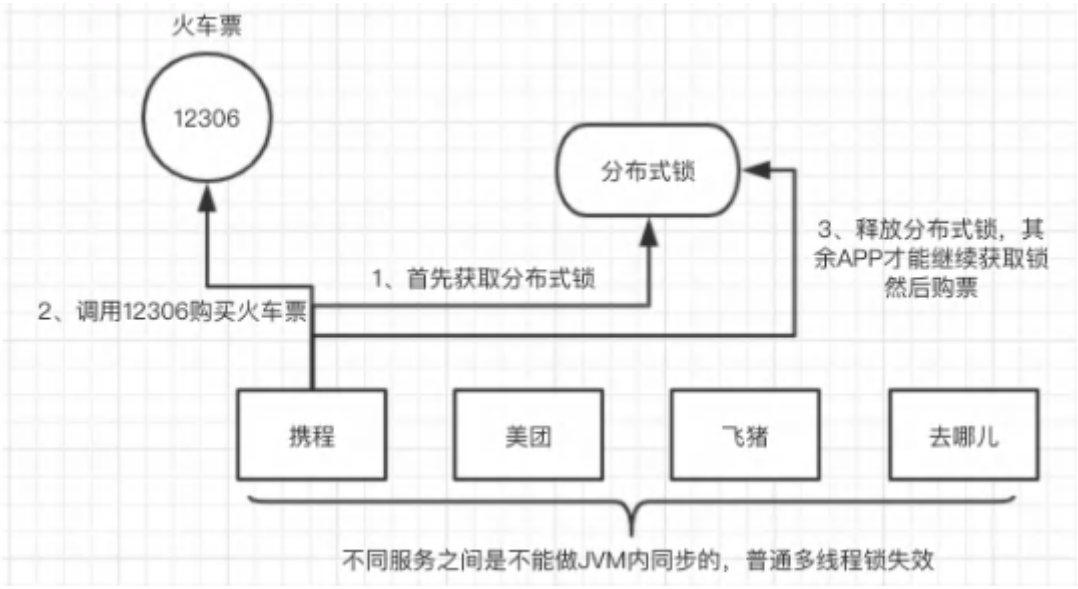
如下例：携程、美团、飞猪、去哪儿四个购票网站实际上都没有最终售票权，只有12306铁道总局有火车票，那么四个购票网站都需要购买火车票，那么四个网站必须排队进行同步，否则不同步会造成多售（类似同一进程中多线程间不同步也会造成多售）。这时他们就需要有一个公共的锁管理方案，来保证APP间的购票是同步的。要想购票：

1）首先服务获取分布式

2）服务获取分布式锁后，才能去调用12306进行购票。

3）购票成功后，释放分布式锁，这样其余APP才能获取锁并进行12306购票。

公众号：旧时光大数据



2.2 为什么要使用Zookeeper

基于mysql实现分布式锁

基于分布式锁的实现，首先肯定是想单独分离出一台mysql数据库，所有服务要想操作文件（共享资源），那么必须先在mysql数据库中插入一个标志，插入标志的服务就持有了锁，并对文件进行操作，操作完成后，主动删除标志进行锁释放，其与服务会一直查询数据库，看是否标志有被占用，直到没有标志占用时自己才能写入标志获取锁。

但是这样有这么一个问题，如果服务（jvm1）宕机或者卡顿了，会一直持有锁未释放，这样就造成了死锁，因此就需要有一个监视锁进程时刻监视锁的状态，如果超过一定时间未释放就要进行主动清理锁标记，然后供其与服务继续获取锁。

如果监视锁字段进程和jvm1同时挂掉，依旧不能解决死锁问题，于是又增加一个监视锁字段进程，这样一个进程挂掉，还有另一个监视锁字段进程可以对锁进行管理。这样又诞生一个新的问题，两个监视进程必须进行同步，否则对于过期的情况管理存在不一致问题。

因此存在以下问题，并且方案变得很复杂：

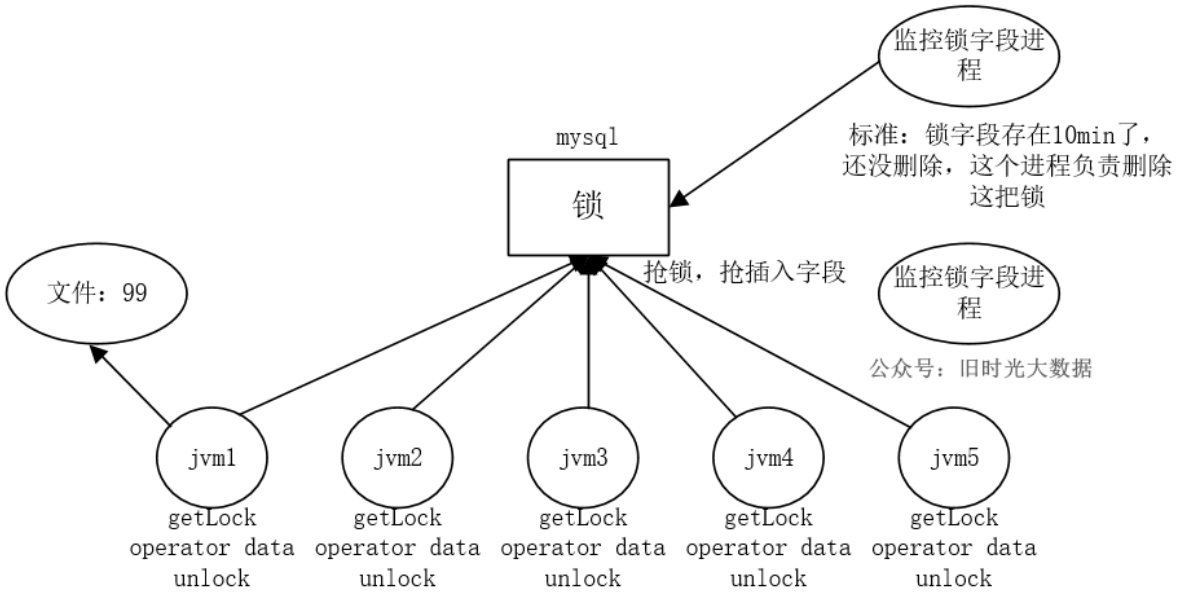
1）监视锁字段进程对于锁的监视时间周期过短，仍旧会造成多售（jvm1还没处理完其持有的锁就被主动销毁，造成多个服务同时持有锁进行操作）。

2）监视锁字段进程对于锁的监视时间周期过长，会造成整个服务卡顿过长，吞吐低下。

3）监视锁字段进程间的同步问题。

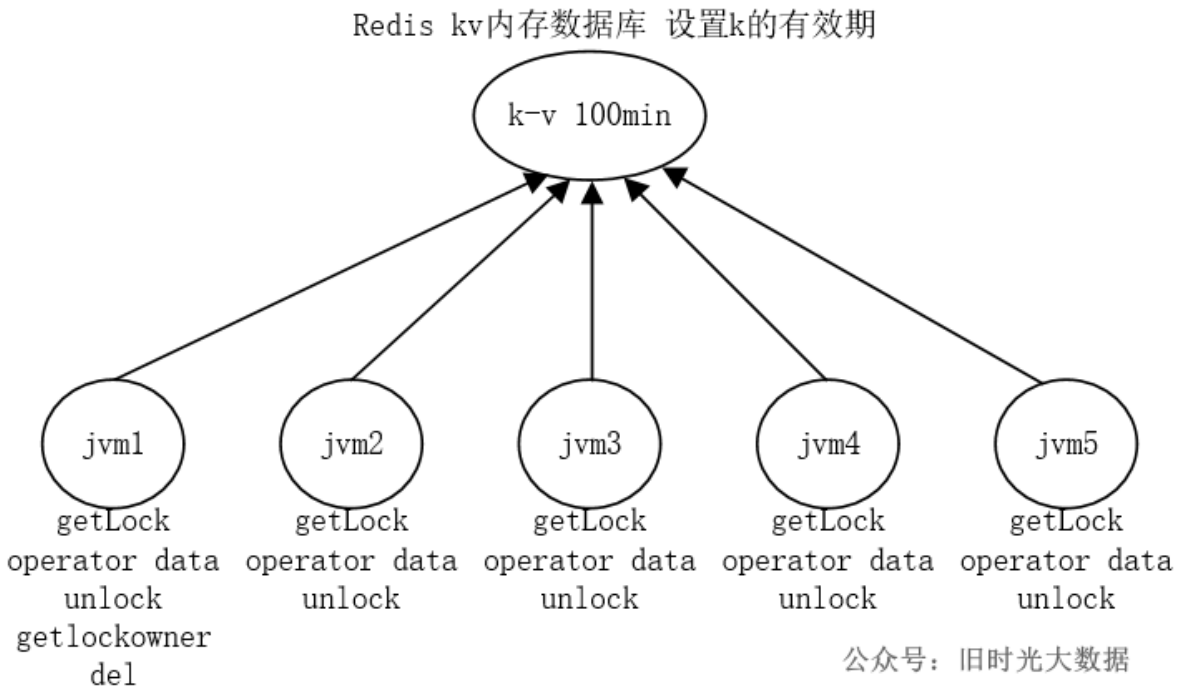
4）当一个jvm持有锁的时候，其余服务会一直访问数据库查看锁，会造成其余jvm的资源浪费。

公众号：旧时光大数据



基于Redis实现分布式锁

相比较于基于数据库实现分布式锁的方案来说，基于缓存来实现在性能方面会表现的更好一点，Redis就是其中一种。由于Redis可以设置字段的有效期，因此可以实现自动释放超期的锁，不需要多个监视锁字段进程进行锁守护，可以依旧存在上述mysql实现中除了3以外1、2、4中的问题。



基于Zookeeper实现分布式锁

相比于前面两种实现方式，基于zookeeper实现分布式锁的方案优于上述两种方案。由于zookeeper有以下特点：

1）维护了一个有层次的数据节点，类似文件系统。

2）有以下数据节点：临时节点、持久节点、临时有序节点**（分布式锁实现基于的数据节点）**、持久有序节点。

3）zookeeper可以和client客户端通过心跳的机制保持长连接，如果客户端链接zookeeper创建了一个临时节点，那么这个客户端与zookeeper断开连接后会自动删除。

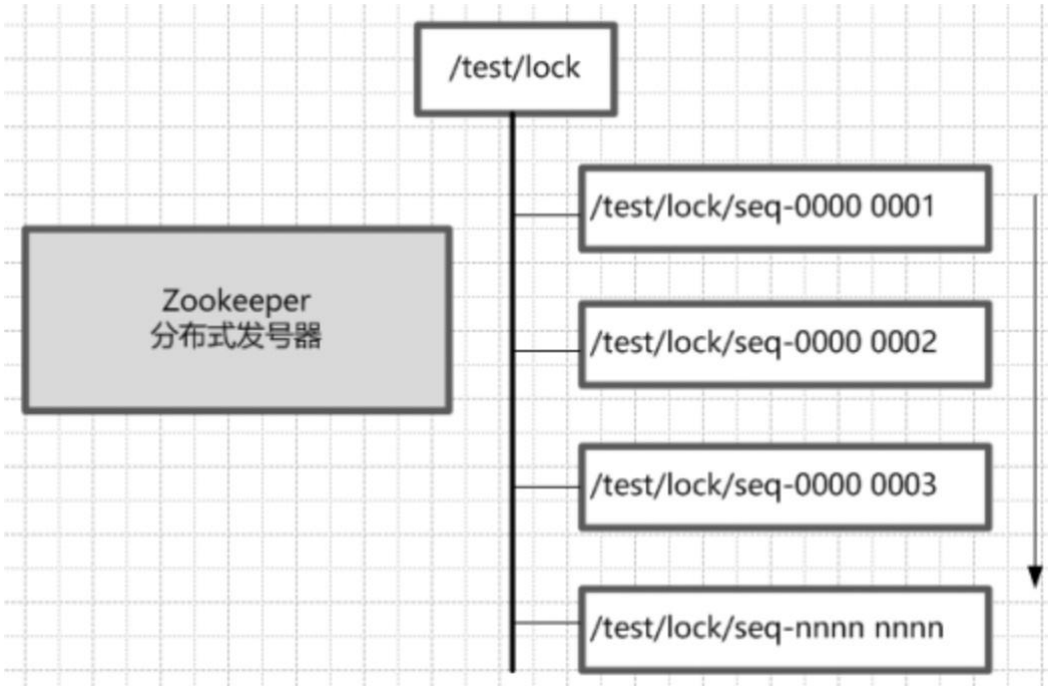
4）zookeeper的节点上可以注册上用户事件(自定义)，如果节点数据删除等事件都可以触发自定义事件。

5）zookeeper保持了统一视图，各服务对于状态信息获取满足一致性。

Zookeeper的每一个节点，都是一个天然的顺序发号器。

在每一个节点下面创建子节点时，只要选择的创建类型是有序（EPHEMERAL_SEQUENTIAL 临时有序或者PERSISTENT_SEQUENTIAL 永久有序）类型，那么，新的子节点后面，会加上一个次序编号。这个次序编号，是上一个生成的次序编号加一。

比如，创建一个用于发号的节点“/test/lock”，然后以他为父亲节点，可以在这个父节点下面创建相同前缀的子节点，假定相同的前缀为“/test/lock/seq-”，在创建子节点时，同时指明是有序类型。如果是第一个创建的子节点，那么生成的子节点为/test/lock/seq-0000000000，下一个节点则为/test/lock/seq-0000000001，依次类推，等等。



3、如何使用Zookeeper实现分布式锁

大致思想为：每个客户端对某个方法加锁时，在 Zookeeper 上与该方法对应的指定节点的目录下，生成一个唯一的临时有序节点。 判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。 当释放锁的时候，只需将这个临时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。

3.1 排它锁

排他锁，又称写锁或独占锁。如果事务T1对数据对象O1加上了排他锁，那么在整个加锁期间，只允许事务T1对O1进行读取或更新操作，其他任务事务都不能对这个数据对象进行任何操作，直到T1释放了排他锁。

排他锁核心是保证当前有且仅有一个事务获得锁，并且锁释放之后，所有正在等待获取锁的事务都能够被通知到。

Zookeeper 的强一致性特性，能够很好地保证在分布式高并发情况下节点的创建一定能够保证全局唯一性，即Zookeeper将会保证客户端无法重复创建一个已经存在的数据节点。可以利用Zookeeper这个特性，实现排他锁。

1）定义锁：通过Zookeeper上的数据节点来表示一个锁

2）获取锁：客户端通过调用 create 方法创建表示锁的临时节点，可以认为创建成功的客户端获得了锁，同时可以让没有获得锁的节点在该节点上注册Watcher监听，以便实时监听到lock节点的变更情况

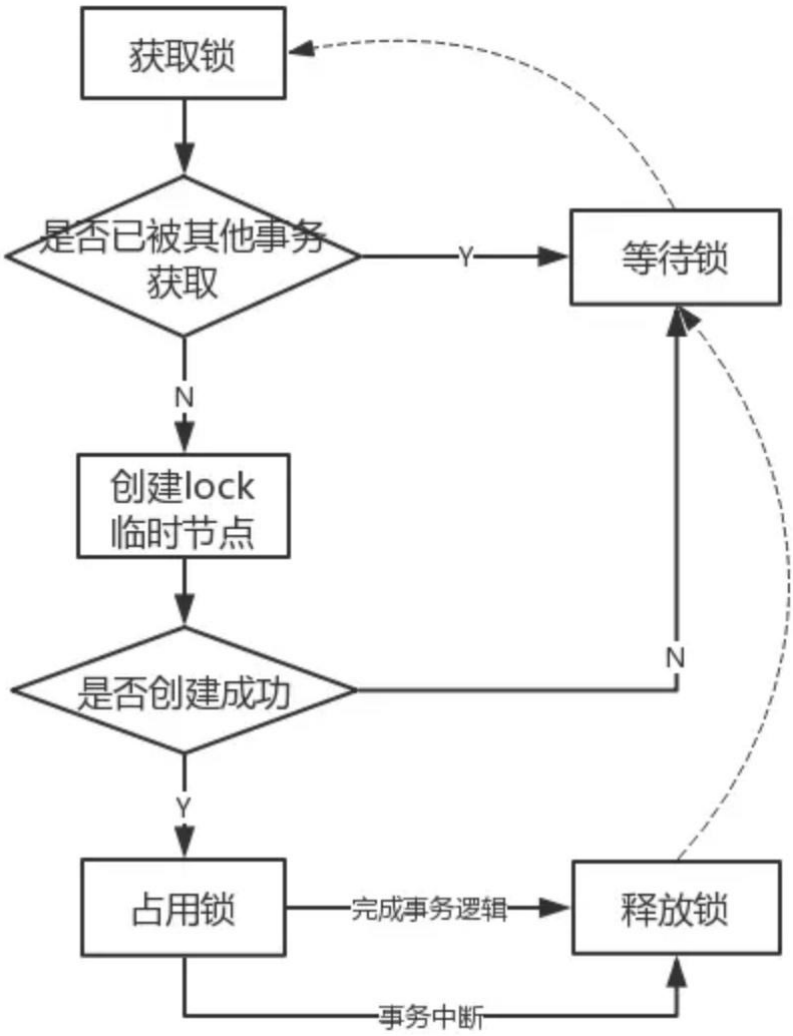
3）释放锁：以下两种情况都可以让锁释放

当前获得锁的客户端发生宕机或异常，那么Zookeeper上这个临时节点就会被删除

正常执行完业务逻辑，客户端主动删除自己创建的临时节点

公众号：旧时光大数据

基于Zookeeper实现排他锁流程：



3.2 共享锁

共享锁，又称读锁。如果事务T1对数据对象O1加上了共享锁，那么当前事务只能对O1进行读取操作，其他事务也只能对这个数据对象加共享锁，直到该数据对象上的所有共享锁都被释放。

共享锁与排他锁的区别在于，加了排他锁之后，数据对象只对当前事务可见，而加了共享锁之后，数据对象对所有事务都可见。

1）定义锁：通过Zookeeper上的数据节点来表示一个锁，是一个类似于/lockpath/[hostname]-请求类型-序号的临时顺序节点

2）获取锁：客户端通过调用 create 方法创建表示锁的临时顺序节点，如果是读请求，则创建 /lockpath/[hostname]-R-序号节点，如果是写请求则创建 /lockpath/[hostname]-W-序号节点

3）判断读写顺序：大概分为4个步骤

① 创建完节点后，获取 /lockpath 节点下的所有子节点，并对该节点注册子节点变更的Watcher监听

② 确定自己的节点序号在所有子节点中的顺序

③ 对于读请求：

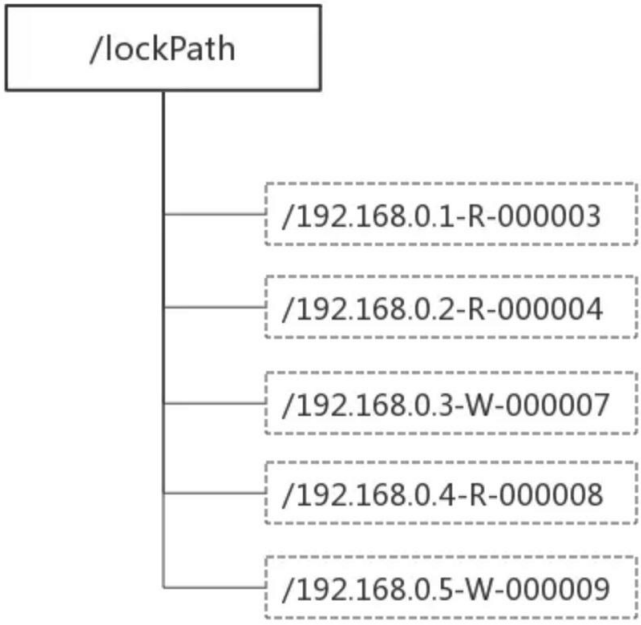
- 如果没有比自己序号更小的子节点，或者比自己序号小的子节点都是读请求，那么表明自己已经成功获取到了共享锁，同时开始执行读取逻辑
- 如果有比自己序号小的子节点有写请求，那么等待

对于写请求，如果自己不是序号最小的节点，那么等待

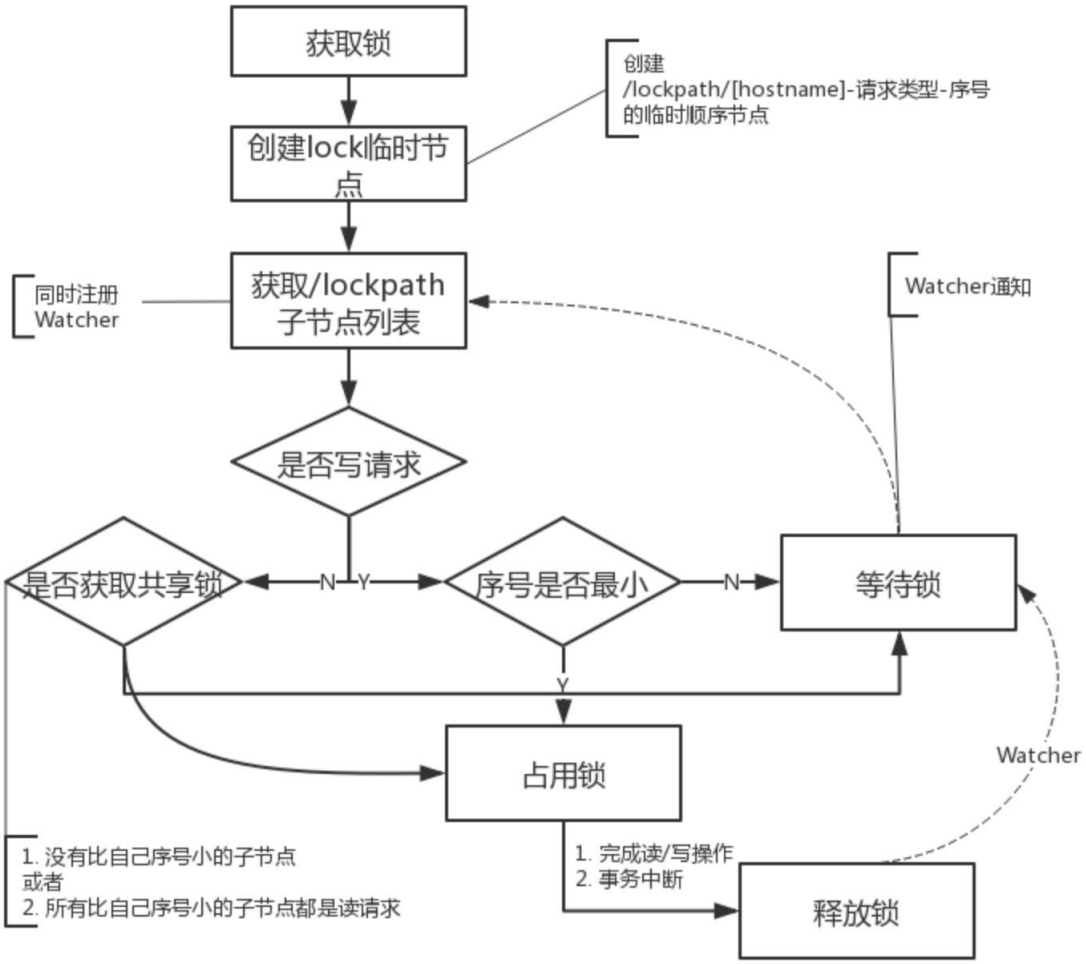
④ 接收到Watcher通知后，重复步骤①

公众号：旧时光大数据

4）释放锁：与排他锁逻辑一致



基于Zookeeper实现共享锁流程：



3.3 羊群效应

在实现共享锁的 "判断读写顺序" 的第1个步骤是：创建完节点后，获取 /lockpath 节点下的所有子节点，并对该节点注册子节点变更的Watcher监听。这样的话，任何一次客户端移除共享锁之后，Zookeeper将会发送子节点变更的Watcher通知给所有机器，系统中将有大量的 "Watcher通知" 和 "子节点列表获取" 这个操作重复执行，然后所有节点再判断自己是否是序号最小的节点(写请求)或者判断比自己序号小的子节点是否都是读请求(读请求)，从而继续等待下一次通知。

公众号：旧时光大数据

然而，这些重复操作很多都是 "无用的"，实际上每个锁竞争者只需要关注序号比自己小的那个节点是否存在即可。

当集群规模比较大时，这些 "无用的" 操作不仅会对Zookeeper造成巨大的性能影响和网络冲击，更为严重的是，如果同一时间有多个客户端释放了共享锁，Zookeeper服务器就会在短时间内向其余客户端发送大量的事件通知，这就是所谓的 "羊群效应"。

改进后的分布式锁实现：

1）客户端调用 create 方法创建一个类似于 /lockpath/[hostname]-请求类型-序号 的临时顺序节点。

2）客户端调用 getChildren 方法获取所有已经创建的子节点列表(这里不注册任何Watcher)。

3）如果无法获取任何共享锁，那么调用 exist 来对比自己小的那个节点注册Watcher

- 读请求：向比自己序号小的最后一个写请求节点注册Watcher监听
- 写请求：向比自己序号小的最后一个节点注册Watcher监听

等待Watcher监听，继续进入步骤2）

Zookeeper羊群效应改进前后Watcher监听图：

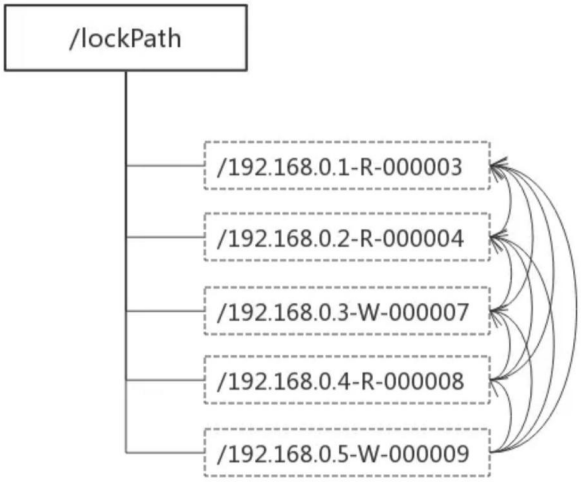


图1. 改进前

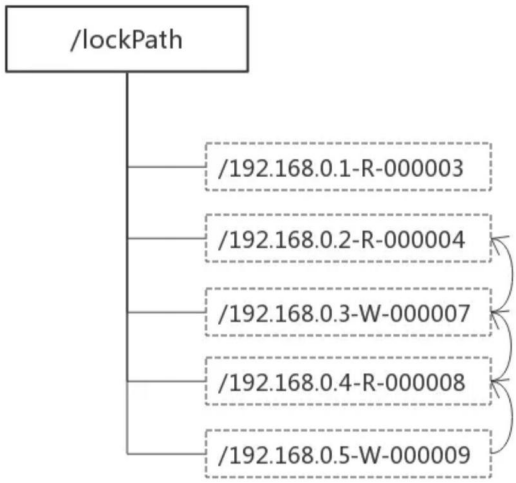


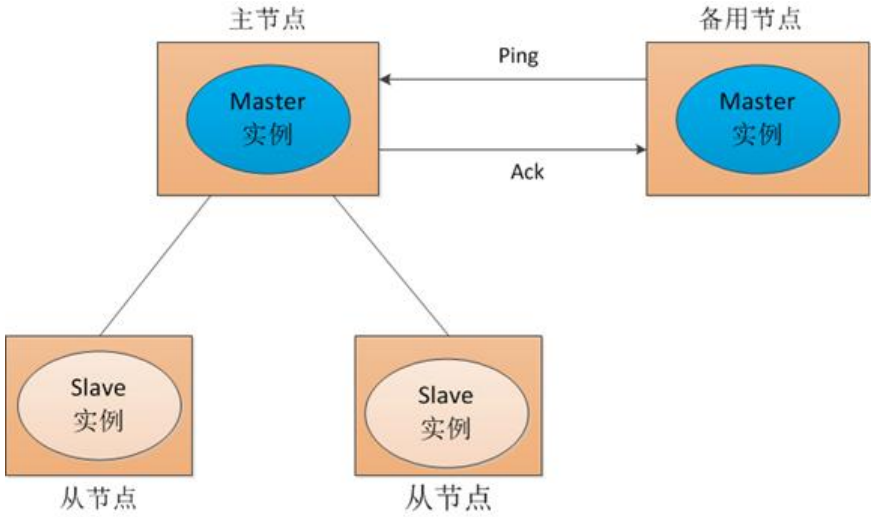
图2. 改进后

4、master选举使用场景及结构

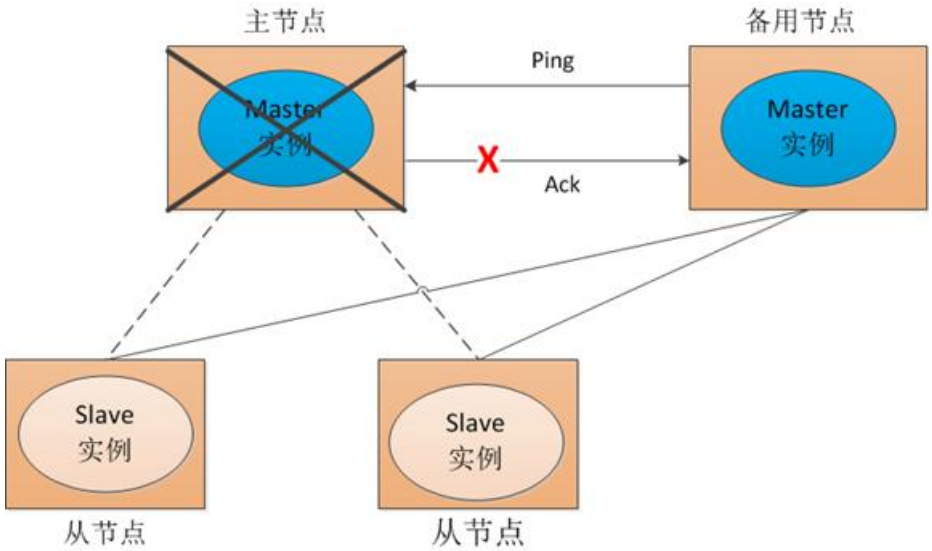
现在很多时候我们的服务需要7*24小时工作，假如一台机器挂了，我们希望能有其它机器顶替它继续工作。此类问题现在多采用master-salve模式，也就是常说的主从模式，正常情况下主机提供服务，备机负责监听主机状态，当主机异常时，可以自动切换到备机继续提供服务（这里有点儿类似于数据库主库跟备库，备机正常情况下只监听，不工作），这个切换过程中选出下一个主机的过程就是master选举。

对于以上提到的场景，传统的解决方式是采用一个备用节点，这个备用节点定期给当前主节点发送ping包，主节点收到ping包后会向备用节点发送应答ack，当备用节点收到应答，就认为主节点还活着，让它继续提供服务，否则就认为主节点挂掉了，自己将开始行使主节点职责。如图1所示：

公众号：旧时光大数据



但这种方式会存在一个隐患，就是网络故障问题。看一下图2：

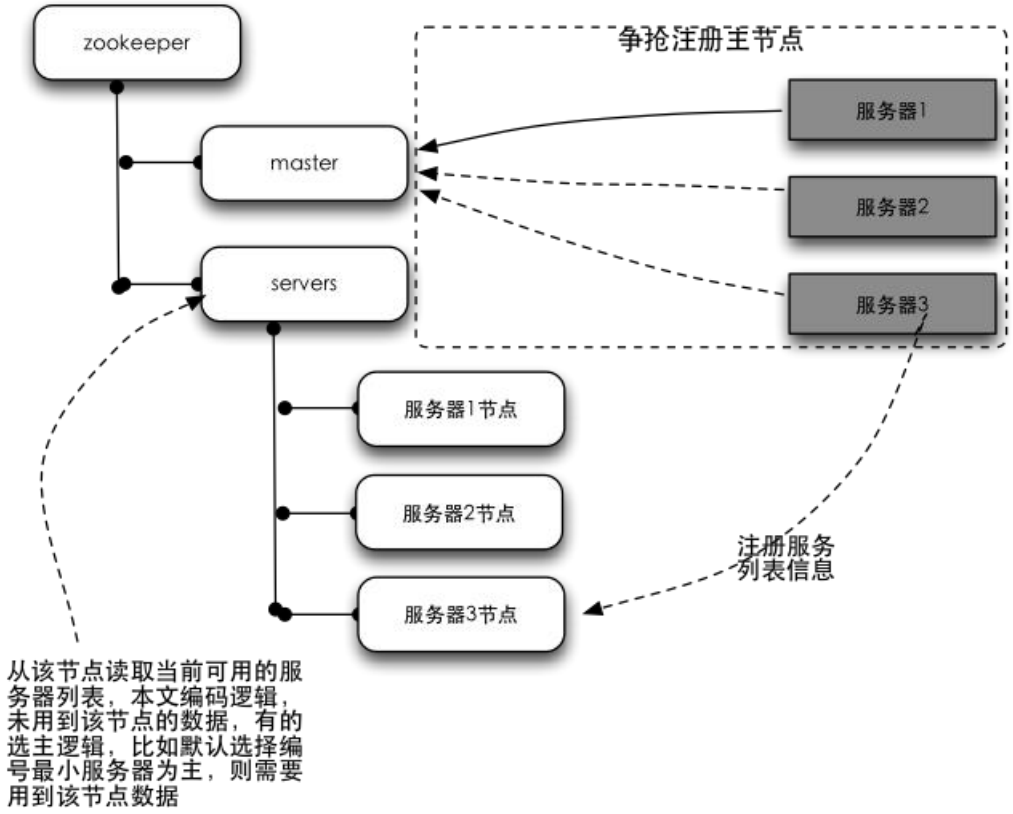


也就是说，我们的主节点并没有挂掉，只是在备用节点ping主节点，请求应答的时候发生网络故障，这样我们的备用节点同样收不到应答，就会认为主节点挂掉，然后备机会启动自己的master实例。这样就会导致系统中有两个主节点，也就是双master。出现双master以后，我们的从节点会将它做的事情一部分汇报给主节点，一部分汇报给备用节点，这样服务就乱套了。为了防止这种情况出现，我们可以考虑采用zookeeper，虽然它不能阻止网络故障的出现，但它能保证同一时刻系统中只存在一个主节点。我们来看zookeeper是怎么实现的：在此处，抢主程序是包含在服务程序中，需要程序员来手动写抢主逻辑的，比如当当开源框架elastic-job中，就有关于选主的部分，参见：elastic-job-core/main/java/com/dangdang/ddframe/job/internal/election文件夹下的选主代码。

Zookeeper自己在集群环境下的抢主算法有三种，可以通过配置文件来设定，默认采用FastLeaderElection，不作赘述；此处主要讨论集群环境中，应用程序利用master的特点，自己选主的过程。程序自己选主，每个人都有自己的一套算法，有采用“最小编号”的，有采用类似“多数投票”的，各有优劣，本文的算法仅作演示理解使用：

结构图如下：

公众号：旧时光大数据



左侧树状结构为zookeeper集群，右侧为程序服务器。所有的服务器在启动的时候，都会订阅zookeeper中master节点的删除事件，以便在主服务器挂掉的时候进行抢主操作；所有服务器同时会在servers节点下注册一个临时节点（保存自己的基本信息），以便于应用程序读取当前可用的服务器列表。

选主原理介绍：Zookeeper的节点有两种类型，持久节点跟临时节点。临时节点有个特性，就是如果注册这个节点的机器失去连接（通常是宕机），那么这个节点会被zookeeper删除。选主过程就是利用这个特性，在服务器启动的时候，去zookeeper特定的一个目录下注册一个临时节点（这个节点作为master，谁注册了这个节点谁就是master），注册的时候，如果发现该节点已经存在，则说明已经有别的服务器注册了（也就是有别的服务器已经抢主成功），那么当前服务器只能放弃抢主，作为从机存在。同时，抢主失败的当前服务器需要订阅该临时节点的删除事件，以便该节点删除时（也就是注册该节点的服务器宕机了或者网络断了之类的）进行再次抢主操作。从机具体需要去哪里注册服务器列表的临时节点，节点保存什么信息，根据具体的业务不同自行约定。选主的过程，其实就是简单的争抢在zookeeper注册临时节点的操作，谁注册了约定的临时节点，谁就是master。

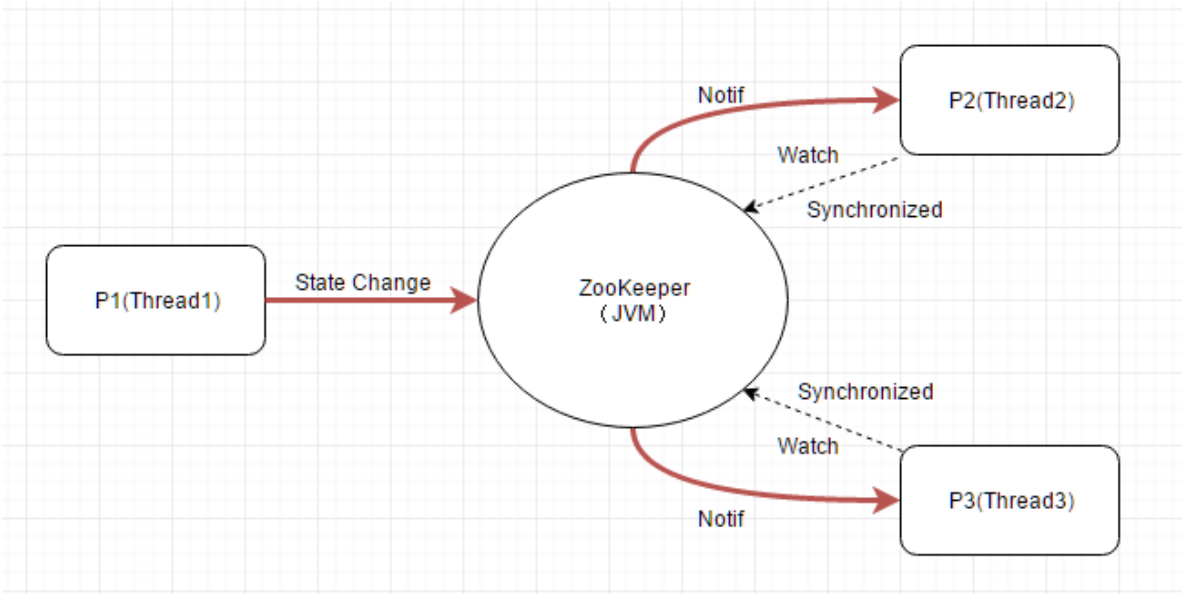
5、 Watch机制

5.1 为什么添加Watch

ZooKeeper是用来协调（同步）分布式进程的服务，提供了一个简单高性能的协调内核，用户可以在此之上构建更多复杂的分布式协调功能。

多个分布式进程通过ZooKeeper提供的API来操作共享的ZooKeeper内存数据对象ZNode来达成某种一致的行为或结果，这种模式本质上是基于状态共享的并发模型，与Java的多线程并发模型一致，他们的线程或进程都是”共享式内存通信“。Java没有直接提供某种响应式通知接口来监控某个对象状态的变化，只能要么浪费CPU时间毫无响应式的轮询重试，或基于Java提供的某种主动通知（Notif）机制（内置队列）来响应状态变化，但这种机制是需要循环阻塞调用。而ZooKeeper实现这些分布式进程的状态（ZNode的Data、Children）共享时，基于性能的考虑采用了类似的异步非阻塞的主动通知模式即Watch机制，使得分布式进程之间的“共享状态通信”更加实时高效，其实这也是ZooKeeper的主要任务决定的—协调。Consul虽然也实现了Watch机制，但它是阻塞的长轮询。

公众号：旧时光大数据



5.2 Watch机制简介

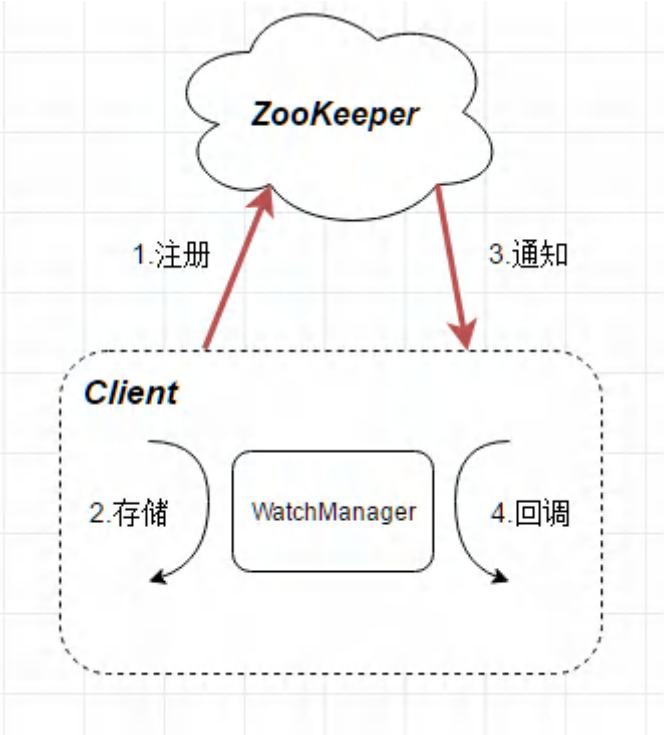
ZooKeeper 提供了分布式数据发布/订阅功能，一个典型的发布/订阅模型系统定义了一种一对多的订阅关系，能让多个订阅者同时监听某一个主题对象，当这个主题对象自身状态变化时，会通知所有订阅者，使他们能够做出相应的处理。

在 ZooKeeper 中，引入了 Watch 机制来实现这种分布式的通知功能。

ZooKeeper 允许客户端向服务端注册一个 Watch 监听，当服务端的一些事件触发了这个 Watch ，那么就会向指定客户端发送一个事件通知，来实现分布式的通知功能。

5.3 ZooKeeper的Watch架构

Watch的整体流程如下图所示，客户端先向ZooKeeper服务端成功注册想要监听的节点状态，同时客户端本地会存储该监听器相关的信息在WatchManager中，当ZooKeeper服务端监听的数据状态发生变化时，ZooKeeper就会主动通知发送相应事件信息给相关会话客户端，客户端就会在本地响应式的回调相关Watcher的Handler。



5.4 Watch机制的特点

一次性触发

公众号：旧时光大数据

- 当 Watch 的对象发生改变时，将会触发此对象上 Watch 所对应的事件，这种监听是一次性的，后续再次发生同样的事件，也不会再次触发。

事件封装

- ZooKeeper 使用 WatchedEvent 对象来封装服务端事件并传递。该对象包含了每个事件的 3 个基本属性，即通知状态(keeperState)、事件类型(EventType)和节点路径(path)。

异步发送

- Watch 的通知事件是从服务端异步发送到客户端的

先注册再触发

- ZooKeeper 中的 Watch 机制，必须由客户端先去服务端注册监听，这样才会触发事件的监听，并通知给客户端。

5.5 Watch 机制的通知状态和事件类型

同一个事件类型在不同的连接状态中代表的含义有所不同，下表列举了常见的连接状态和事件类型

连接状态	状态含义	事件类型	事件含义
Disconnected	连接失败	NodeCreated	节点被创建
SyncConnected	连接成功	NodeDataChanged	节点数据变更
AuthFailed	认证失败	NodeChildrenChanged	子节点数据变更
Expired	会话过期	NodeDeleted	节点被删除

从表可知， ZooKeeper 常见的连接状态和事件类型分别有 4 种，具体含义如下。

状态

- 当客户端断开连接，这时客户端和服务器的连接就是 Disconnected 状态，说明连接失败;
- 当客户端和服务器的某一个节点建立连接，并完成一次 version 、 zxid 的同步，这时客户端和服务器的连接状态就是 SyncConnected ，说明连接成功;
- 当 ZooKeeper 客户端连接认证失败，这时客户端和服务器的连接状态就是 AuthFailed ，说明认证失败;
- 当客户端发送 Request 请求，通知服务器其上一个发送心跳的时间，服务器收到这个请求后，通知客户端下一个发送心跳的时间是哪个时间点。当客户端时间戳达到最后一个发送心跳的时间，而没有收到服务器发来的新发送心跳的时间，即认为自己下线，这时客户端和服务器的连接状态就是 Expired 状态，说明会话过期。

事件

- 当节点被创建时，NodeCreated 事件被触发;
- 当节点的数据发生变更时，NodeDataChanged 事件被触发;
- 当节点的直接子节点被创建、被删除、子节点数据发生变更时，NodeChildrenChanged 事件被触发;
- 当节点被删除时，NodeDeleted 事件被触发。

介绍下Zookeeper消息的发布订阅功能

问过的一些公司：字节

参考答案：

公众号：旧时光大数据

1、发布订阅基本概念

发布订阅模式可以看成一对多的关系：多个订阅者对象同时监听一个主题对象，这个主题对象在自身状态发生变化时，会通知所有的订阅者对象，使他们能够自动的更新自己的状态。

发布订阅模式，可以让发布方和订阅方，独立封装，独立改变，当一个对象的改变，需要同时改变其他的对象，而且它不知道有多少个对象需要改变时，可以使用发布订阅模式

发布订阅模式在分布式系统的典型应用有：**配置管理和服务发现。**

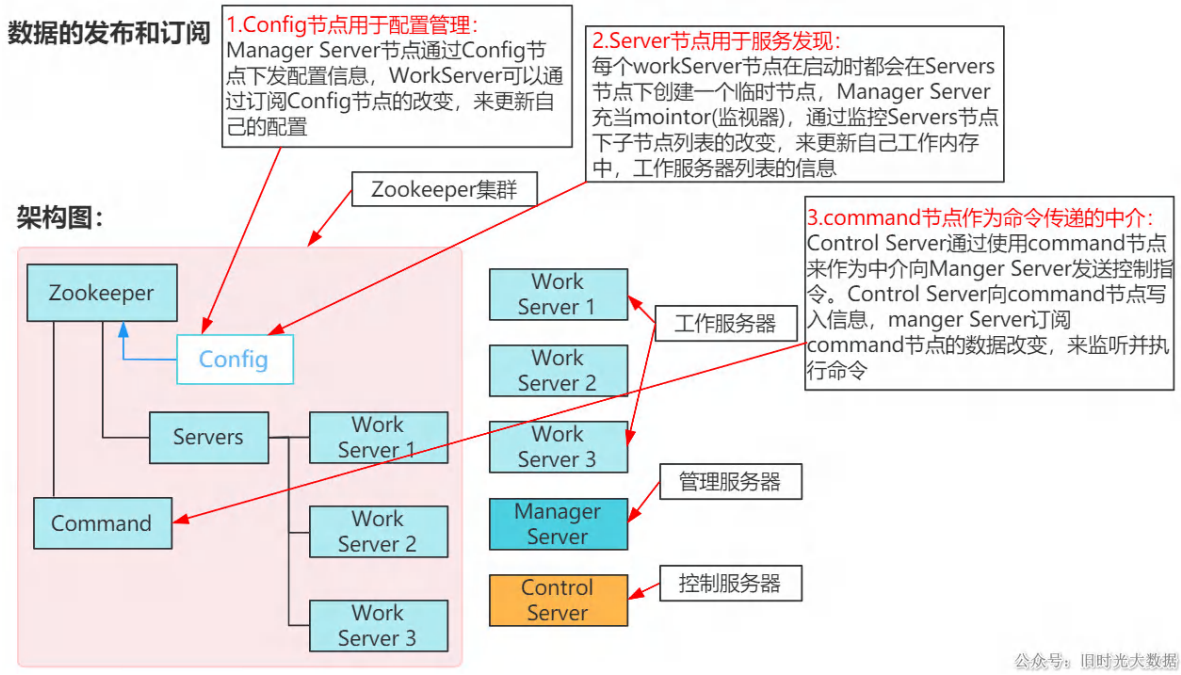
配置管理：是指如果集群中机器拥有某些相同的配置，并且这些配置信息需要动态的改变，我们可以使用发布订阅模式，对配置文件做统一的管理，让这些机器各自订阅配置文件的改变，当配置文件发生改变的时候这些机器就会得到通知，把自己的配置文件更新为最新的配置

服务发现：是指对集群中的服务上下线做统一的管理，每个工作服务器都可以作为数据的发布方，向集群注册自己的基本信息，而让模型机器作为订阅方，订阅工　作服务器的基本信息，当工作服务器的基本信息发生改变时如上下线，服务器的角色和服务范围变更，监控服务器就会得到通知，并响应这些变化。

2、发布订阅的架构图

Zookeeper的消息订阅及发布最典型的应用实例是作为系统的配置中心，发布者将数据发布到ZK节点上，供订阅者动态获取数据，实现配置信息的集中式管理和动态更新。例如全局的配置信息，服务式服务框架的服务地址列表等就非常适合使用。

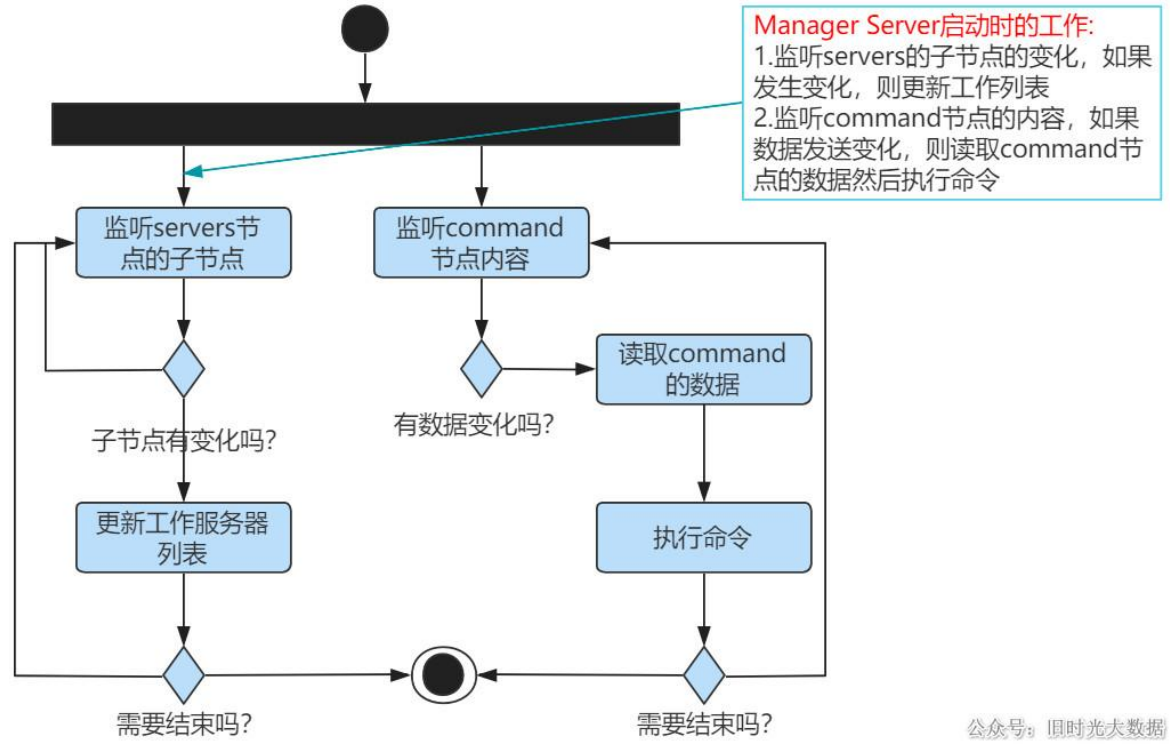
Zookeeper 主要有两种监听方式，**监听子节点状态**，**监听节点数据。**下图中，work server节点可存储应用服务器元数据（如:服务器ip和端口等信息），查询server节点可获取服务器列表，创建zookeeper客户端监听server节点的字节点状态，在应用服务器暂停使用（删除对应work server子节点）或增加应用服务器时（增加对应work server 子节点）时，zookeeper客户端可获及时的通知并进行处理。config节点可存储分布式集群的全局配置信息，在全局配置信息需要修改时，可将配置信息发布到config节点下，所有对config节点数据进行监听的zookeeper客户端可及时收到通知并对服务器的配置信息进行修改。



3、Manager Server的工作流程

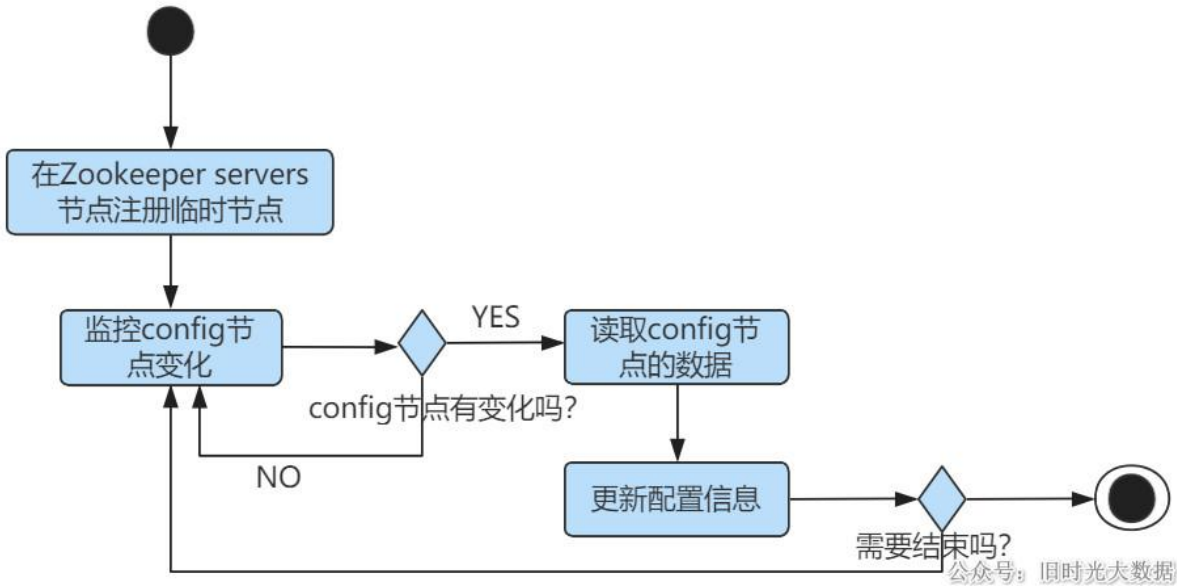
公众号：旧时光大数据

Manager Server程序主体流程：



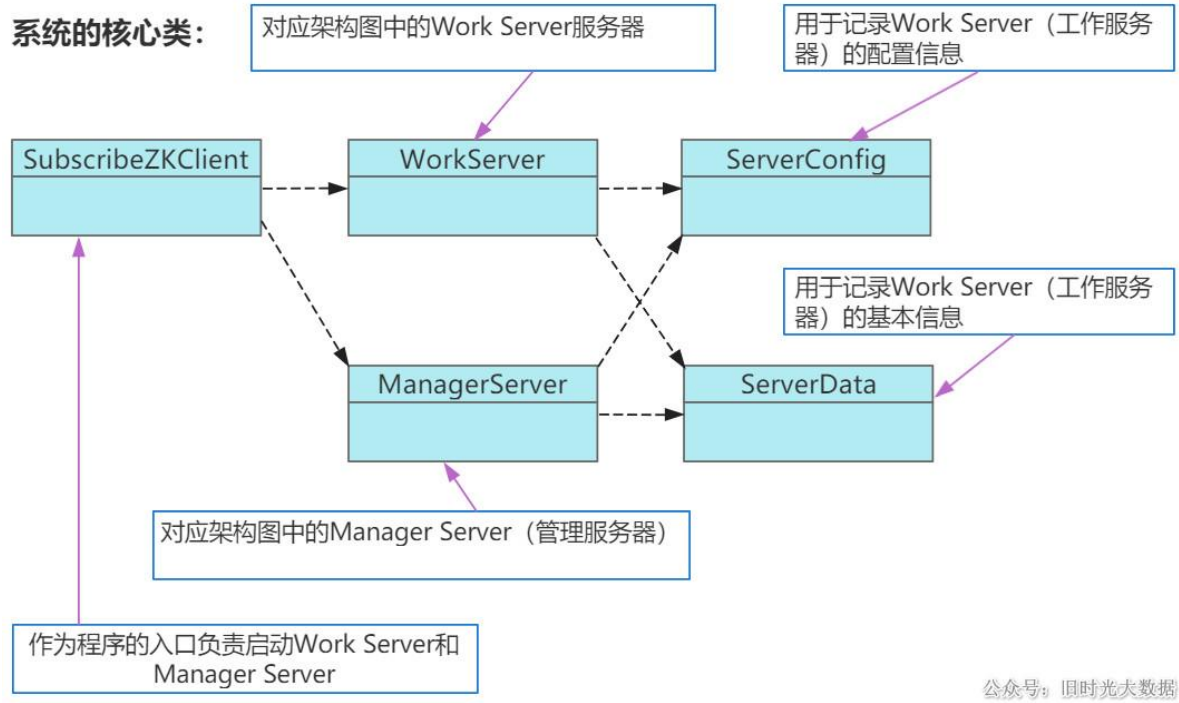
4、Work Server的工作流程

Work Server程序主体流程：



5、发布订阅程序的结构图

公众号：旧时光大数据



Zookeeper的分布式锁实现方式?

问过的一些公司：陌陌，小米，阿里x2，vivo，阿里,，快手，恒生电子，Shopee(2021.07)

参考答案：

分布式锁是控制分布式系统之间同步访问共享资源的一种方式。

zookeeper有以下特点：

1）维护了一个有层次的数据节点，类似文件系统。

2）有以下数据节点：临时节点、持久节点、临时有序节点**（分布式锁实现基于的数据节点）**、持久有序节点。

3）zookeeper可以和client客户端通过心跳的机制保持长连接，如果客户端链接zookeeper创建了一个临时节点，那么这个客户端与zookeeper断开连接后会自动删除。

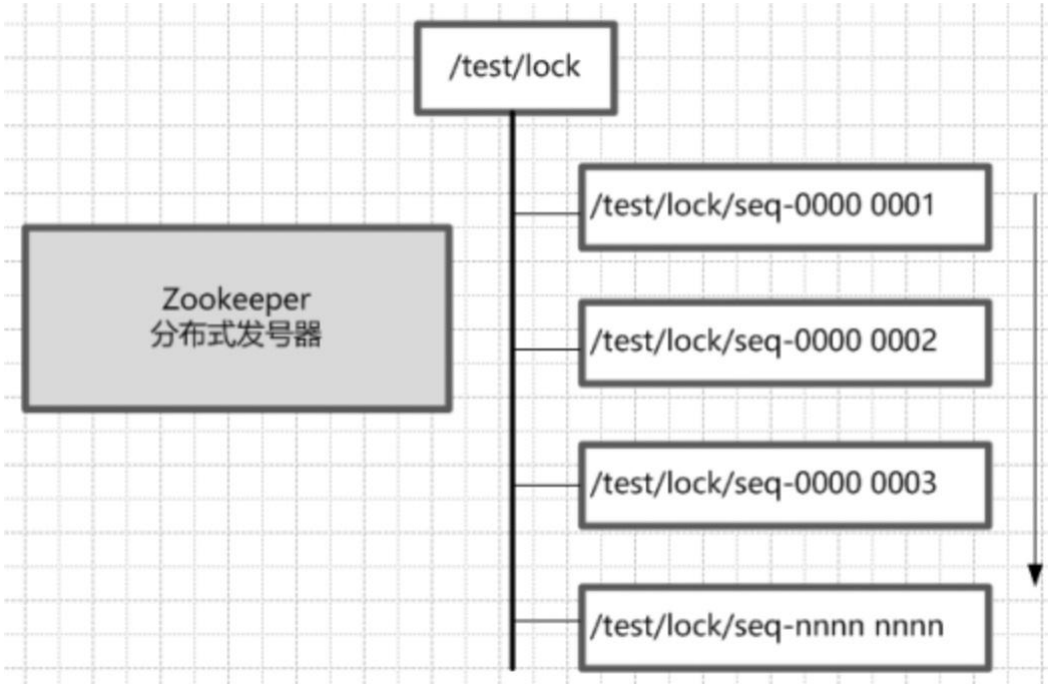
4）zookeeper的节点上可以注册上用户事件(自定义)，如果节点数据删除等事件都可以触发自定义事件。

5）zookeeper保持了统一视图，各服务对于状态信息获取满足一致性。

Zookeeper的每一个节点，都是一个天然的顺序发号器。

在每一个节点下面创建子节点时，只要选择的创建类型是有序（EPHEMERAL_SEQUENTIAL 临时有序或者PERSISTENT_SEQUENTIAL 永久有序）类型，那么，新的子节点后面，会加上一个次序编号。这个次序编号，是上一个生成的次序编号加一。

比如，创建一个用于发号的节点“/test/lock”，然后以他为父亲节点，可以在这个父节点下面创建相同前缀的子节点，假定相同的前缀为“/test/lock/seq-”，在创建子节点时，同时指明是有序类型。如果是第一个创建的子节点，那么生成的子节点为/test/lock/seq-0000000000，下一个节点则为/test/lock/seq-0000000001，依次类推，等等。



如何使用Zookeeper实现分布式锁

大致思想为：每个客户端对某个方法加锁时，在 Zookeeper 上与该方法对应的指定节点的目录下，生成一个唯一的临时有序节点。 判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。 当释放锁的时候，只需将这个临时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。

1、排它锁

排他锁，又称写锁或独占锁。如果事务T1对数据对象O1加上了排他锁，那么在整个加锁期间，只允许事务T1对O1进行读取或更新操作，其他任务事务都不能对这个数据对象进行任何操作，直到T1释放了排他锁。

排他锁核心是保证当前有且仅有一个事务获得锁，并且锁释放之后，所有正在等待获取锁的事务都能够被通知到。

Zookeeper 的强一致性特性，能够很好地保证在分布式高并发情况下节点的创建一定能够保证全局唯一性，即Zookeeper将会保证客户端无法重复创建一个已经存在的数据节点。可以利用Zookeeper这个特性，实现排他锁。

1）定义锁：通过Zookeeper上的数据节点来表示一个锁

2）获取锁：客户端通过调用 create 方法创建表示锁的临时节点，可以认为创建成功的客户端获得了锁，同时可以让没有获得锁的节点在该节点上注册Watcher监听，以便实时监听到lock节点的变更情况

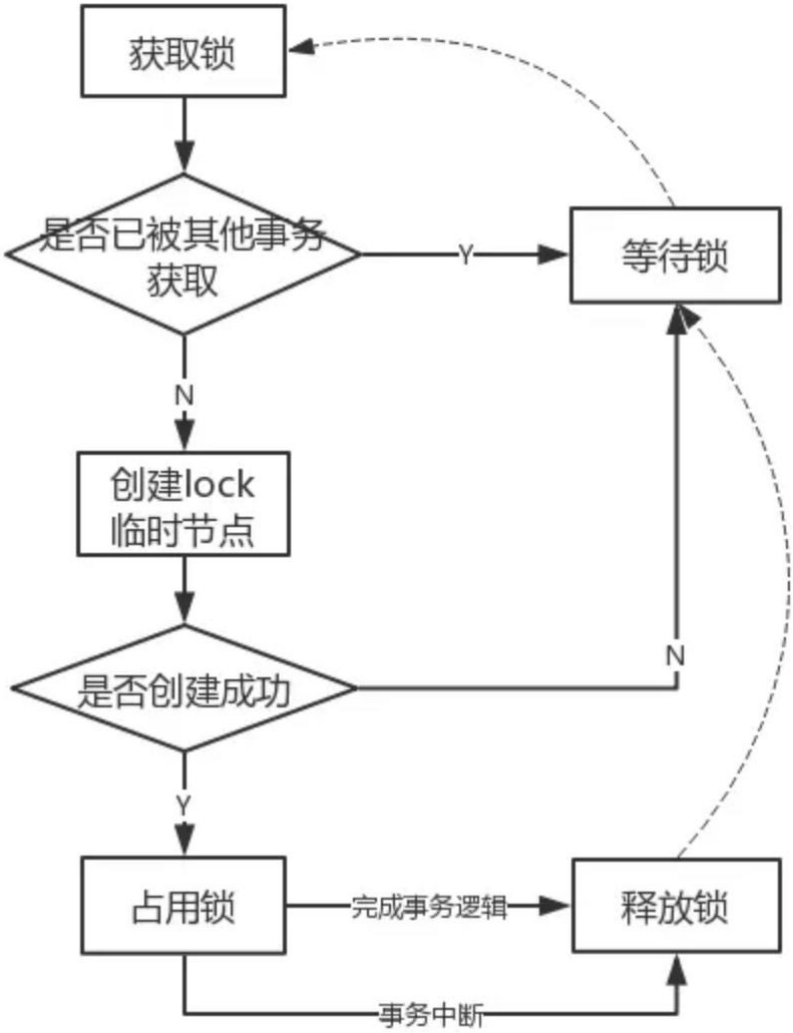
3）释放锁：以下两种情况都可以让锁释放

当前获得锁的客户端发生宕机或异常，那么Zookeeper上这个临时节点就会被删除

正常执行完业务逻辑，客户端主动删除自己创建的临时节点

基于Zookeeper实现排他锁流程：

公众号：旧时光大数据



2、共享锁

共享锁，又称读锁。如果事务T1对数据对象O1加上了共享锁，那么当前事务只能对O1进行读取操作，其他事务也只能对这个数据对象加共享锁，直到该数据对象上的所有共享锁都被释放。

共享锁与排他锁的区别在于，加了排他锁之后，数据对象只对当前事务可见，而加了共享锁之后，数据对象对所有事务都可见。

1）定义锁：通过Zookeeper上的数据节点来表示一个锁，是一个类似于/lockpath/[hostname]-请求类型-序号的临时顺序节点

2）获取锁：客户端通过调用 create 方法创建表示锁的临时顺序节点，如果是读请求，则创建 /lockpath/[hostname]-R-序号节点，如果是写请求则创建 /lockpath/[hostname]-W-序号节点

3）判断读写顺序：大概分为4个步骤

① 创建完节点后，获取 /lockpath 节点下的所有子节点，并对该节点注册子节点变更的Watcher监听

② 确定自己的节点序号在所有子节点中的顺序

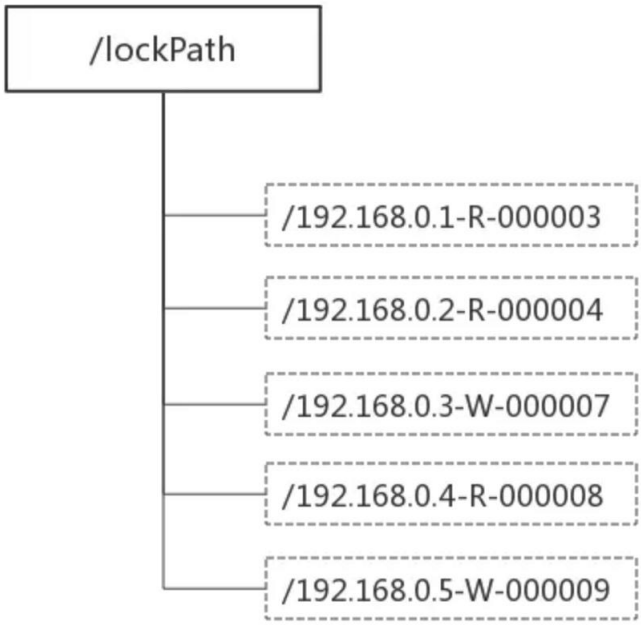
③ 对于读请求：

- 如果没有比自己序号更小的子节点，或者比自己序号小的子节点都是读请求，那么表明自己已经成功获取到了共享锁，同时开始执行读取逻辑
- 如果有比自己序号小的子节点有写请求，那么等待

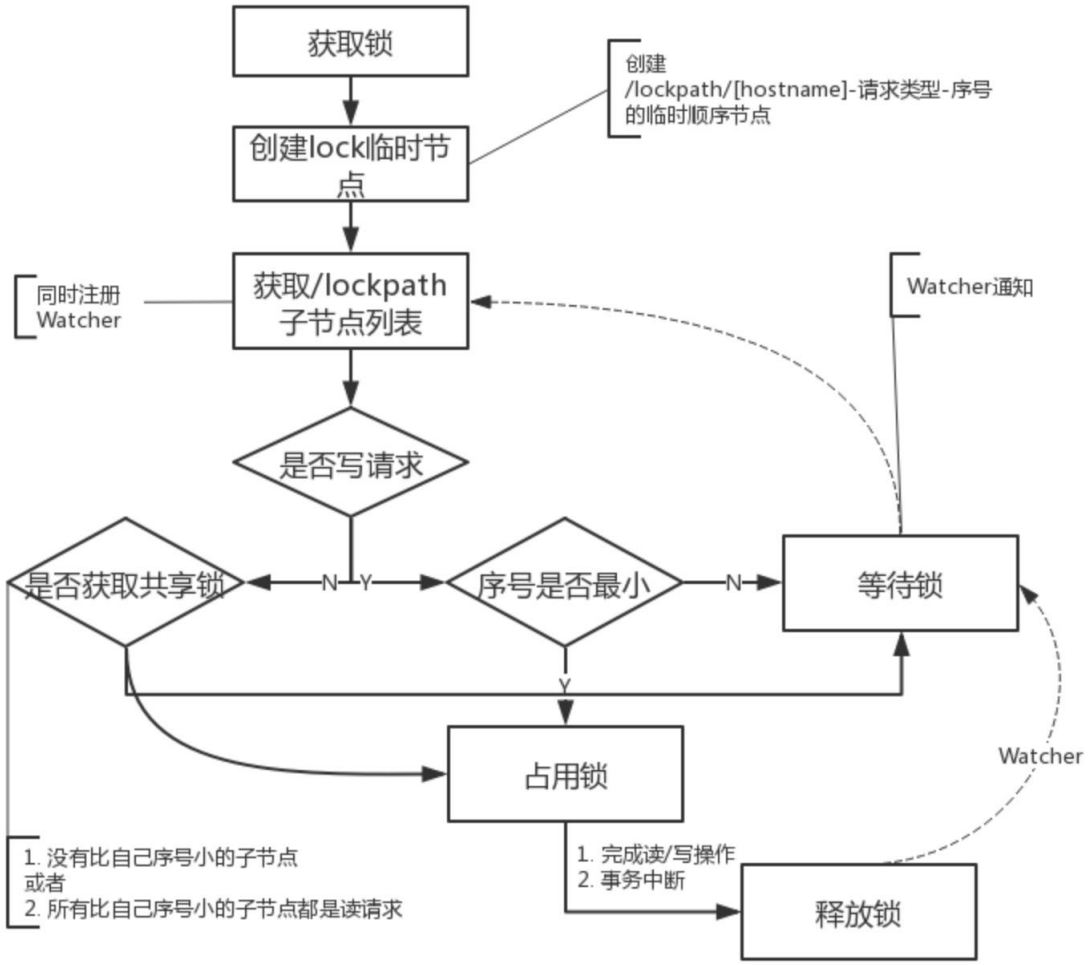
对于写请求，如果自己不是序号最小的节点，那么等待

④ 接收到Watcher通知后，重复步骤①

4）释放锁：与排他锁逻辑一致



基于Zookeeper实现共享锁流程：



Zookeeper怎么保证一致性的

问过的一些公司：阿里，快手

参考答案：

依赖了ZAB协议

ZAB协议是伪分布式协调服务Zookeeper专门设计的一种崩溃恢复的原子广播协议

公众号：旧时光大数据

它有两种基本的模式：

- 崩溃恢复
- 消息广播

这两个模式是相辅相成的，消息广播模式就是**zookeeper**不出现任何问题，并且正常工作的模式，崩溃恢复看字面意思就是当Zookeeper出现故障时用于恢复的。

当出现故障时这两种模式怎么用？(不故障时就一种都没必要说)

当整个zookeeper集群刚刚启动或者Leader服务器宕机、重启或者网络故障导致不存在过半的服务器与Leader服务器保持正常通信时(Leader选举，必须要存在过半的服务器)，所有进程（服务器）进入**崩溃恢复模式**，首先选举产生新的Leader服务器，然后集群中Follower服务器开始与新的Leader服务器进行数据同步，当集群中超过半数机器与该Leader服务器完成数据同步之后，退出恢复模式进入**消息广播模式**，Leader服务器开始接收客户端的事务请求生成事物提案来进行事务请求处理。

Zookeeper的zab协议（原子广播协议）？

可回答：说下Zookeeper的一致性算法

问过的一些公司：海康(2021.08)

Zab协议是Zookeeper保证数据一致性的核心算法，Zab借鉴了Paxos算法，但又不像Paxos那样，是一种**通用**的分布式一致性算法，基于该协议，zk实现了一种主备模型（即Leader和Follower模型），保证了事务的最终一致性。

ZAB协议介绍

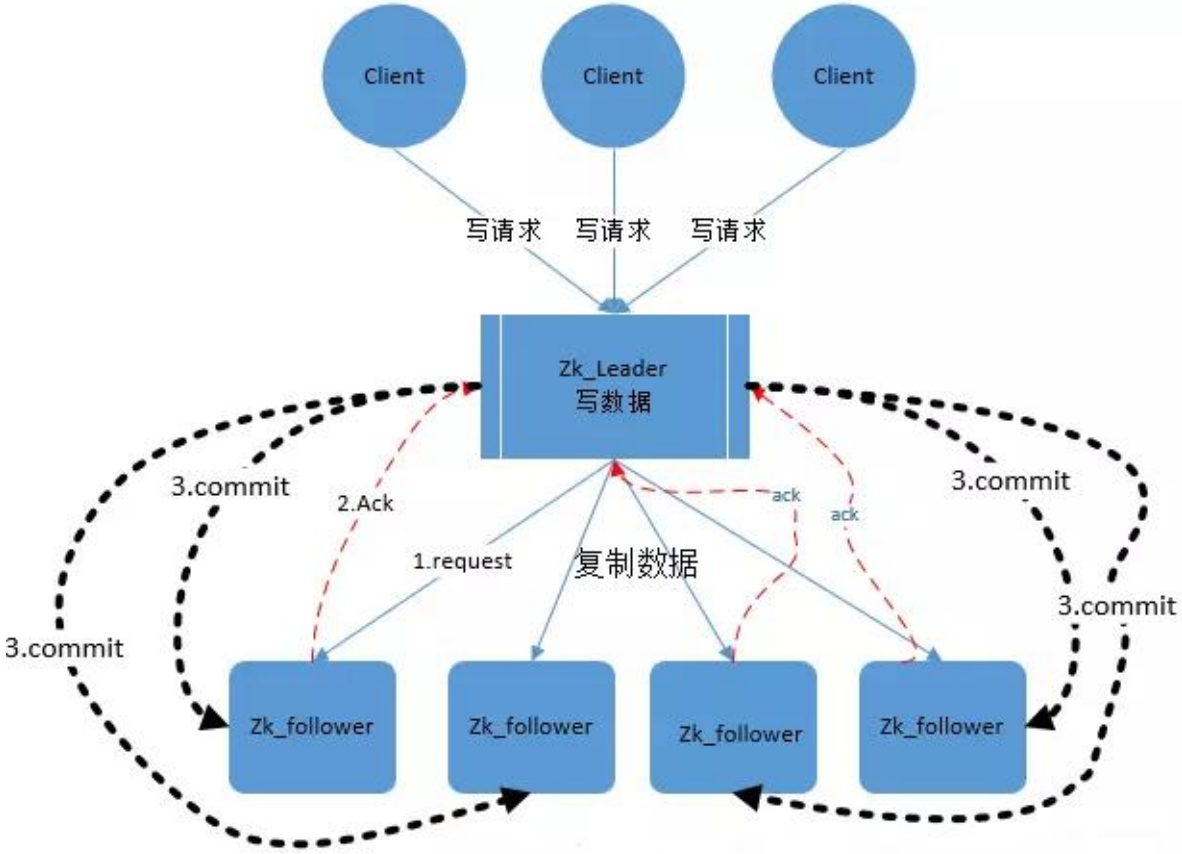
ZAB 协议全称：Zookeeper Atomic Broadcast（Zookeeper 原子广播协议）。

Zookeeper 是一个为分布式应用提供高效且可靠的分布式协调服务。在解决分布式一致性方面，Zookeeper 并没有使用Paxos ， 而是采用了ZAB协议。

ZAB 协议定义：ZAB 协议是为分布式协调服务 Zookeeper 专门设计的一种支持**崩溃恢复**和**原子广播**协议。

基于该协议，Zookeeper 实现了**一种主备模式的系统架构**来保持集群中各个副本之间数据一致性

公众号：旧时光大数据



首先客户端所有的写请求都由一个Leader接收，而其余的都是Follower（从者）。

Leader 负责将一个客户端事务请求，转换成一个事务Proposal，并将该Proposal分发给集群中所有的Follower（备份），也就是向所有Follower节点发送数据广播请求（或数据复制）记住这里发送给的不光是数据本身，还有其他的，相当于包装。

分发之后Leader需要等待所有Follower的反馈（Ack请求），在Zab协议中，只要超过半数的Follower进行了正确的反馈后（也就是收到半数以上的Follower的Ack请求），那么 Leader 就会再次向所有的Follower发送Commit消息，要求其将上一个事务proposal进行提交，就是相当于通过，如果是用dubbo进行微服务就是zookeeper这个协调服务通过了，开始往服务端发送数据。

ZAB协议的两种模式：崩溃恢复和消息广播

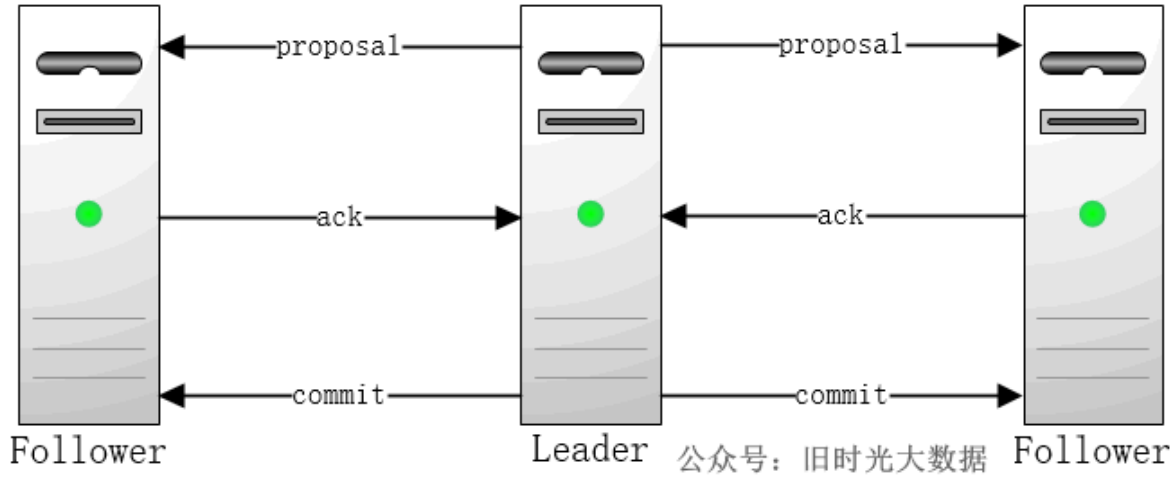
当整个集群正在启动时，或者当Leader节点出现网络中断、崩溃等情况时，ZAB协议就会进入恢复模式并选举产生新的Leader，当Leader服务器选举出来后，并且集群中有过半的机器和该Leader节点完成数据同步后（同步指的是数据同步，用来保证集群中过半的机器能够和Leader服务器的数据状态保持一致），ZAB协议就会退出恢复模式。

当集群中已经有过半的Follower节点完成了和Leader状态同步以后，那么整个集群就进入了消息广播模式。这个时候，在Leader节点正常工作时，启动一台新的服务器加入到集群，那这个服务器会直接进入数据恢复模式，和Leader节点进行数据同步。同步完成后即可正常对外提供非事务请求的处理。

消息广播（原子广播）

消息广播实际上是一个简化版的2PC提交过程。

公众号：旧时光大数据



过程如下：

1）leader接收到消息请求后，将消息赋予一个全局唯一的64位自增id，叫：zxid，通过zxid的大小比较就可以实现因果有序这个特征。

2）leader为每个follower准备了一个FIFO队列（通过TCP协议来实现，以实现全局有序这一个特点）将带有zxid的消息作为一个提案（proposal）分发给所有的 follower。

3）当follower接收到proposal，先把proposal写到磁盘，写入成功以后再向leader回复一个ack。

4）当leader接收到合法数量（超过半数节点）的ack后，leader就会向这些follower发送commit命令，同时会在本地执行该消息。

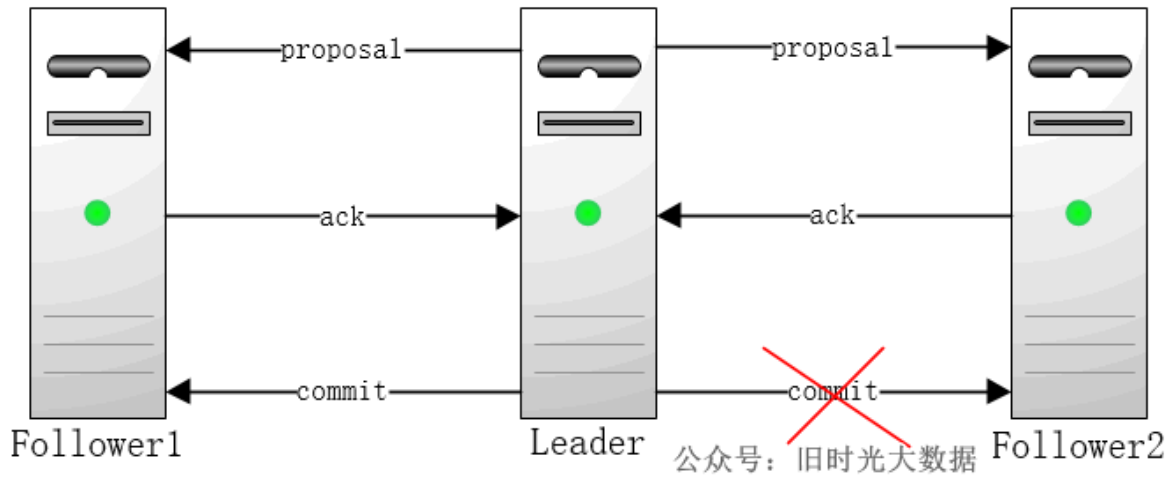
5）当follower收到消息的commit命令以后，会提交该消息。

崩溃恢复（数据恢复）

ZAB协议的这个基于原子广播协议的消息广播过程，在正常情况下是没有任何问题的，但是一旦Leader节点崩溃，或者由于网络问题导致Leader服务器失去了过半的Follower节点的联系（leader失去与过半follower节点联系，可能是leader节点和 follower节点之间产生了网络分区，那么此时的leader不再是合法的leader了），那么就会进入到崩溃恢复模式。在ZAB协议中，为了保证程序的正确运行，整个恢复过程结束后需要选举出一个新的Leader。
为了使leader挂了后系统能正常工作，需要解决以下两个问题：

1）已经被处理的消息不能丢失

当leader收到合法数量follower的ack后，就向各个follower广播commit命令，同时也会在本地执行commit并向连接的客户端返回「成功」。但是如果各个follower在收到commit命令前leader就挂了，导致剩下的服务器并没有执行到这条消息。



leader对事务消息发起commit操作，该消息在follower1上执行了，但是follower2还没有收到commit，leader就已经挂了，而实际上客户端已经收到该事务消息处理成功的回执了。所以在zab协议下需要保证所有机器都要执行这个事务消息，必须满足已经被处理的消息不能丢失。

2）被丢弃的消息不能再次出现

当leader接收到消息请求生成proposal后就挂了，其他follower并没有收到此proposal，因此经过恢复模式重新选了leader后，这条消息是被跳过的。 此时，之前挂了的leader重新启动并注册成了follower，他保留了被跳过消息的proposal状态，与整个系统的状态是不一致的，需要将其删除。（leader都换代了，所以以前leader的proposal失效了）。

针对崩溃恢复的两种情况分析

ZAB协议需要满足上面两种情况，就必须要设计一个leader选举算法，能够确保已经被leader提交的事务Proposal能够提交、同时丢弃已经被跳过的事务Proposal。

针对这个要求：
如果leader选举算法能够保证新选举出来的Leader服务器拥有集群中所有机器最高编号（ZXID 最大）的事务Proposal，那么就可以保证这个新选举出来的leader一定具有已经提交的提案。因为所有提案被commit之前必须有超过半数的follower ack，即必须有超过半数节点的服务器的事务日志上有该提案的proposal，因此只要有合法数量的节点正常工作，就必然有一个节点保存了所有被commit消息的proposal状态。

另外一个，zxid是64位，高32位是epoch编号，每经过一次Leader选举产生一个新的leader，新的leader会将epoch号+1，低32位是消息计数器，每接收到一条消息这个值+1，新leader选举后这个值重置为0。这样设计的好处在于老的leader挂了以后重启，它不会被选举为leader，因此此时它的zxid肯定小于当前新的leader。当老的leader作为follower接入新的leader后，新的leader会让它将所有的拥有旧的epoch号的未被commit的proposal清除。

关于ZXID

zxid，也就是事务id，为了保证事务的顺序一致性，zookeeper采用了递增的事务id号（zxid）来标识事务。所有的提议（proposal）都在被提出的时候加上了zxid，实际中zxid是一个64位的数字，它高32位是epoch（ZAB协议通过epoch编号来区分Leader周期变化的策略）用来标识leader关系是否改变，每次一个leader被选出来，它都会有一个新的epoch=（原来的epoch+1），标识当前属于那个leader的统治时期。低32位用于递增计数。

epoch：可以理解为当前集群所处的年代或者周期，每个leader就像皇帝，都有自己的年号，所以每次改朝换代，leader变更之后，都会在前一个年代的基础上加1。这样就算旧的leader崩溃恢复之后，也没有人听他的了，因为follower只听从当前年代的leader的命令。

ZAB协议的原理总结

发现：必须有一个Leader，而Leader拥有一个Follower的列表，里面是可以通讯的Follower，为后面的同步，广播做准备

同步：Leader 要负责将本身的数据与 Follower 完成同步，做到多副本存储。这样也是提现了CAP中的高可用和分区容错。Follower将队列中未处理完的请求消费完成后，写入本地事务日志中

广播：Leader 可以接受客户端新的事务Proposal请求，将新的Proposal请求广播给所有的 Follower。

ZAB是以什么算法为基础的？ZAB流程？

问过的一些公司：恒生电子

参考答案：

ZAB是在Paxos算法基础上进行了扩展改造而来的。

公众号：旧时光大数据

算法阶段

ZAB协议定义了**选举（election）、发现（discovery）、同步（sync）、广播(Broadcast)**四个阶段；ZAB选举（election）时当Follower存在ZXID（事务ID）时判断所有Follower节点的事务日志，只有lastZXID的节点才有资格成为Leader，这种情况下选举出来的Leader总有最新的事务日志，基于这个原因所以ZooKeeper实现的时候把**发现（discovery）与同步（sync）合并为恢复（recovery）**阶段。

1）**Election**：在Looking状态中选举出Leader节点，Leader的lastZXID总是最新的；

2）**Discovery**：Follower节点向准Leader推送FOllOWERINFO，该信息中包含了上一周期的epoch，接受准Leader的NEWLEADER指令，检查newEpoch有效性，准Leader要确保Follower的epoch与ZXID小于或等于自身的；

3）**sync**：将Follower与Leader的数据进行同步，由Leader发起同步指令，最总保持集群数据的一致性；

4）**Broadcast**：Leader广播Proposal与Commit，Follower接受Proposal与Commit；

5）**Recovery**：在Election阶段选举出Leader后本阶段主要工作就是进行数据的同步，使Leader具有highestZXID，集群保持数据的一致性；

1、选举（Election）

election阶段必须确保选出的Leader具有highestZXID，否则在Recovery阶段没法保证数据的一致性，Recovery阶段Leader要求Follower向自己同步数据没有Follower要求Leader保持数据同步，所有选举出来的Leader要具有最新的ZXID；
在选举的过程中会对每个Follower节点的ZXID进行对比只有highestZXID的Follower才可能当选Leader；
选举流程：

```
1        1．每个Follower都向其他节点发送选自身为Leader的Vote投票请求，等待回复；
2          2．Follower接受到的Vote如果比自身的大（ZXID更新）时则投票，并更新自身的Vote，否则拒绝投票；
3                    3．每个Follower中维护着一个投票记录表，当某个节点收到过半的投票时，结束投票并把该Follower选为Leader，投票结束；
```

ZAB协议中使用ZXID作为事务编号，ZXID为64位数字，低32位为一个递增的计数器，每一个客户端的一个事务请求时Leader产生新的事务后该计数器都会加1，高32位为Leader周期epoch编号，当新选举出一个Leader节点时Leader会取出本地日志中最大事务Proposal的ZXID解析出对应的epoch把该值加1作为新的epoch，将低32位从0开始生成新的ZXID；ZAB使用epoch来区分不同的Leader周期；

2、恢复（Recovery）

在election阶段选举出来的Leader已经具有最新的ZXID，所有本阶段的主要工作是根据Leader的事务日志对Follower节点数据进行更新；
Leader：Leader生成新的ZXID与epoch，接收Follower发送过来的FOllOWERINFO（含有当前节点的LastZXID）然后往Follower发送NEWLEADER；Leader根据Follower发送过来的LastZXID根据数据更新策略向Follower发送更新指令；
同步策略：

1. **SNAP**：如果Follower数据太老，Leader将发送快照SNAP指令给Follower同步数据；
2. **DIFF**：Leader发送从Fololwer.lastZXID到Leader.lastZXID议案的DIFF指令给Follower同步数据；
3. **TRUNC**：当Follower.lastZXID比Leader.lastZXID大时，Leader发送从Leader.lastZXID到Follower.lastZXID的TRUNC指令让Follower丢弃该段数据；

Follower：往Leader发送FOLLOERINFO指令，Leader拒绝就转到Election阶段；接收Leader的NEWLEADER指令，如果该指令中epoch比当前Follower的epoch小那么Follower转到Election阶段；Follower还有主要工作是接收SNAP/DIFF/TRUNC指令同步数据与ZXID，同步成功后回复ACKNETLEADER，然后进入下一阶段；Follower将所有事务都同步完成后Leader会把该节点添加到可用Follower列表中；

公众号：旧时光大数据

SNAP与DIFF用于保证集群中Follower节点已经Committed的数据的一致性，TRUNC用于抛弃已经被处理但是没有Committed的数据；

3、**广播(Broadcast)**

客户端提交事务请求时Leader节点为每一个请求生成一个事务Proposal，将其发送给集群中所有的Follower节点，收到过半Follower的反馈后开始对事务进行提交，ZAB协议使用了原子广播协议；在ZAB协议中只需要得到过半的Follower节点反馈Ack就可以对事务进行提交，这也导致了Leader几点崩溃后可能会出现数据不一致的情况，ZAB使用了崩溃恢复来处理数字不一致问题；消息广播使用了TCP协议进行通讯所有保证了接受和发送事务的顺序性。广播消息时Leader节点为每个事务Proposal分配一个全局递增的ZXID（事务ID），每个事务Proposal都按照ZXID顺序来处理。

Leader节点为每一个Follower节点分配一个队列按事务ZXID顺序放入到队列中，且根据队列的规则FIFO来进行事务的发送。Follower节点收到事务Proposal后会将该事务以事务日志方式写入到本地磁盘中，成功后反馈Ack消息给Leader节点，Leader在接收到过半Follower节点的Ack反馈后就会进行事务的提交，以此同时向所有的Follower节点广播Commit消息，Follower节点收到Commit后开始对事务进行提交；

Zookeeper的通知机制

问过的一些公司：东方头条

参考答案：

客户端注册监听它关心的目录节点，当目录节点发生变化（数据改变、被删除、子目录节点增加删除）时，zookeeper会通知客户端。

1、watch是什么

ZooKeeper 支持watch(观察)的概念。客户端可以在每个znode结点上设置一个观察。如果被观察服务端的znode结点有变更，那么watch就会被触发，这个watch所属的客户端将接收到一个通知包被告知结点已经发生变化，把相应的事件通知给设置过Watcher的Client端。

Zookeeper里的所有读取操作：getData(),getChildren()和exists()都有设置watch的选项

总结来说就是：异步回调的触发机制

2、watch事件理解

1）一次触发

- 当数据有了变化时zkserver向客户端发送一个watch,它是一次性的动作，即触发一次就不再有效，类似一次性纸杯。
- 只监控一次
- 如果想继续Watch的话，需要客户端重新设置Watcher。因此如果你得到一个watch事件且想在将来的变化得到通知，必须新设置另一个watch。

2）发往客户端

- Watches是异步发往客户端的，Zookeeper提供一个顺序保证：在看到watch事件之前绝不会看到变化，这样不同客户端看到的是一致性的顺序。
- 在（导致观察事件被触发的）修改操作的成功返回码到达客户端之前，事件可能在去往客户端的路上，但是可能不会到达客户端。观察事件是异步地发送给观察者（客户端）的。ZooKeeper会保证次序：在收到观察事件之前，客户端不会看到已经为之设置观察的节点的改动。网络延迟或者其他因素可能会让不同的客户端在不同的时间收到观察事件和更新操作的返回码。这里的要点是：不同客户端看到的事情都有一致的次序。

公众号：旧时光大数据

3）为数据设置watch

- 节点有不同的改动方式。可以认为ZooKeeper维护两个观察列表：**数据观察和子节点观察。** getData()和exists()设置数据观察。getChildren()设置子节点观察。此外，还可以认为不同的返回数据有不同的观察。getData()和exists()返回节点的数据，而getChildren()返回子节点列表。所以，setData()将为znode触发数据观察。成功的create()将为新创建的节点触发数据观察，为其父节点触发子节点观察。成功的delete()将会为被删除的节点触发数据观察以及子节点观察（因为节点不能再有子节点了），为其父节点触发子节点观察。

- 观察维护在客户端连接到的ZooKeeper服 务器中。这让观察的设置、维护和分发是轻量级的。客户端连接到新的服务器时，所有会话事件将被触发。同服务器断开连接期间不会收到观察。客户端重新连接 时，如果需要，先前已经注册的观察将被重新注册和触发。通常这都是透明的。有一种情况下观察事件将丢失：对还没有创建的节点设置存在观察，而在断开连接期 间创建节点，然后删除。

4）时序性和一致性

- Watches是在client连接到Zookeeper服务端的本地维护，这可让watches成为轻量的，可维护的和派发的。当一个client连接到新server，watch将会触发任何session事件，断开连接后不能接收到。当客户端重连，先前注册的watches将会被重新注册并触发。

- 关于watches，Zookeeper维护这些保证：
  （1）Watches和其他事件、watches和异步恢复都是有序的。Zookeeper客户端保证每件事都是有序派发
  （2）客户端在看到新数据之前先看到watch事件
  （3）对应更新顺序的watches事件顺序由Zookeeper服务所见

Zookeeper脑裂问题

问过的一些公司：阿里

参考答案：

1、什么是脑裂

脑裂（split-brain）就是“大脑分裂”，也就是本来一个“大脑”被拆分了两个或多个“大脑”，我们都知道，如果一个人有多个大脑，并且相互独立的话，那么会导致人体“手舞足蹈”，“不听使唤”。

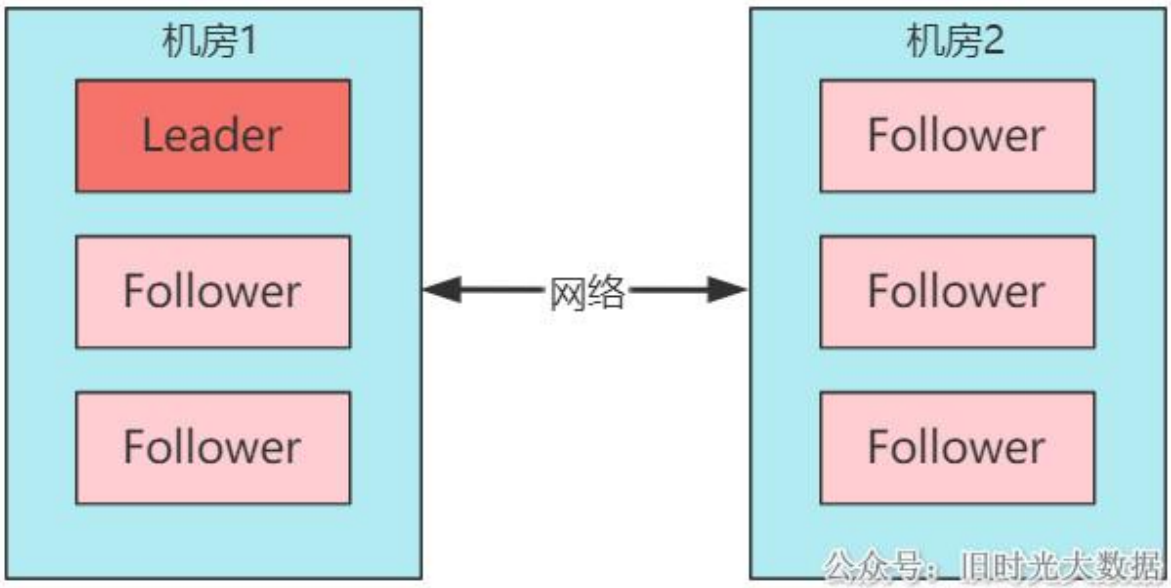
脑裂通常会出现在集群环境中，比如ElasticSearch、Zookeeper集群，而这些集群环境有一个统一的特点，就是它们有一个大脑，比如ElasticSearch集群中有Master节点，Zookeeper集群中有Leader节点。

当集群中出现故障（比如网络故障），分成了“两个集群”，这两个集群无法互相通信，此时就会让集群觉得Leader挂了，这时候就会出现两个Leader节点，这就是脑裂。

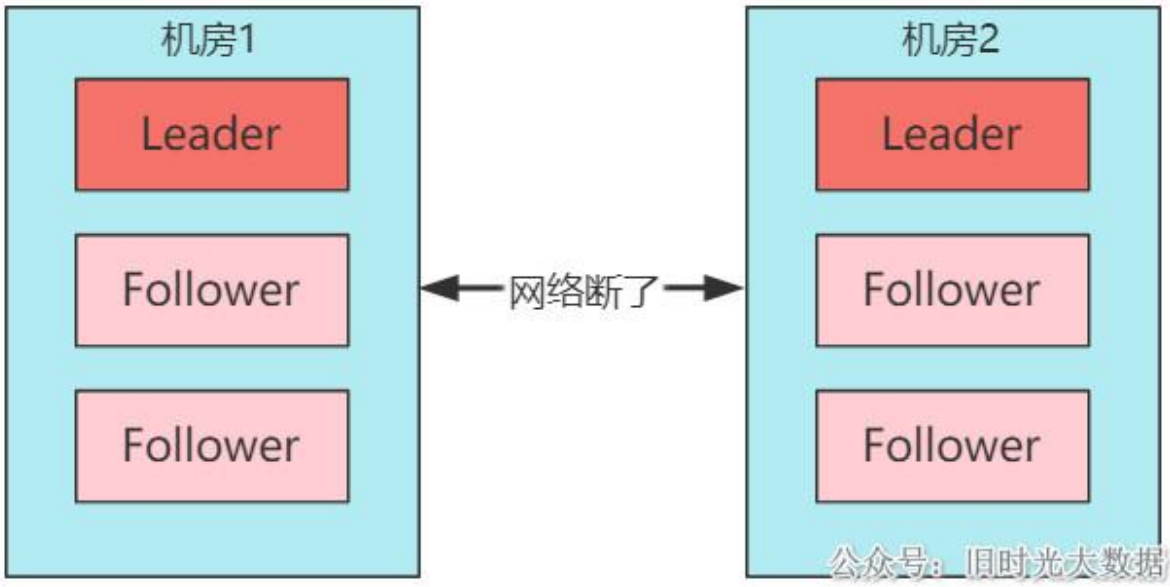
2、Zookeeper集群中脑裂场景

对于一个集群，想要提高这个集群的可用性，通常会采用多机房部署，比如现在有一个由6台zkServer所组成的一个集群，部署在了两个机房：

公众号：旧时光大数据



正常情况下，此集群只会有一个Leader，那么如果机房之间的网络断了之后，两个机房内的zkServer还是可以相互通信的，如果**不考虑过半机制**，那么就会出现每个机房内部都将选出一个Leader。



这就相当于原本一个集群，被分成了两个集群，出现了两个“大脑”，这就是脑裂。

对于这种情况，我们也可以看出来，原本应该是统一的一个集群对外提供服务的，现在变成了两个集群同时对外提供服务，如果过了一会，断了的网络突然联通了，那么此时就会出现问题了，两个集群刚刚都对外提供服务了，数据该怎么合并，数据冲突怎么解决等等问题。

刚刚在说明脑裂场景时，有一个前提条件就是没有考虑过半机制，所以实际上Zookeeper集群中是不会出现脑裂问题的，而不会出现的原因就跟过半机制有关。

3、过半机制

在领导者选举的过程中，如果某台zkServer获得了超过半数的选票，则此zkServer就可以成为Leader了。

过半机制的源码实现其实非常简单：

```
1   public class QuorumMaj implements QuorumVerifier {
2       private static final Logger LOG = LoggerFactory.getLogger(QuorumMaj.class);
3
4       int half;
5
6       // n表示集群中zkServer的个数（准确的说是参与者的个数，参与者不包括观察者节点）
7       public QuorumMaj(int n){
```

公众号：旧时光大数据

```
            this.half = n/2;
        }

        // 验证是否符合过半机制
        public boolean containsQuorum(Set<Long> set){
            // half是在构造方法里赋值的
            // set.size()表示某台zkServer获得的票数
            return (set.size() > half);
        }

    }
```

核心代码就是下面两行：

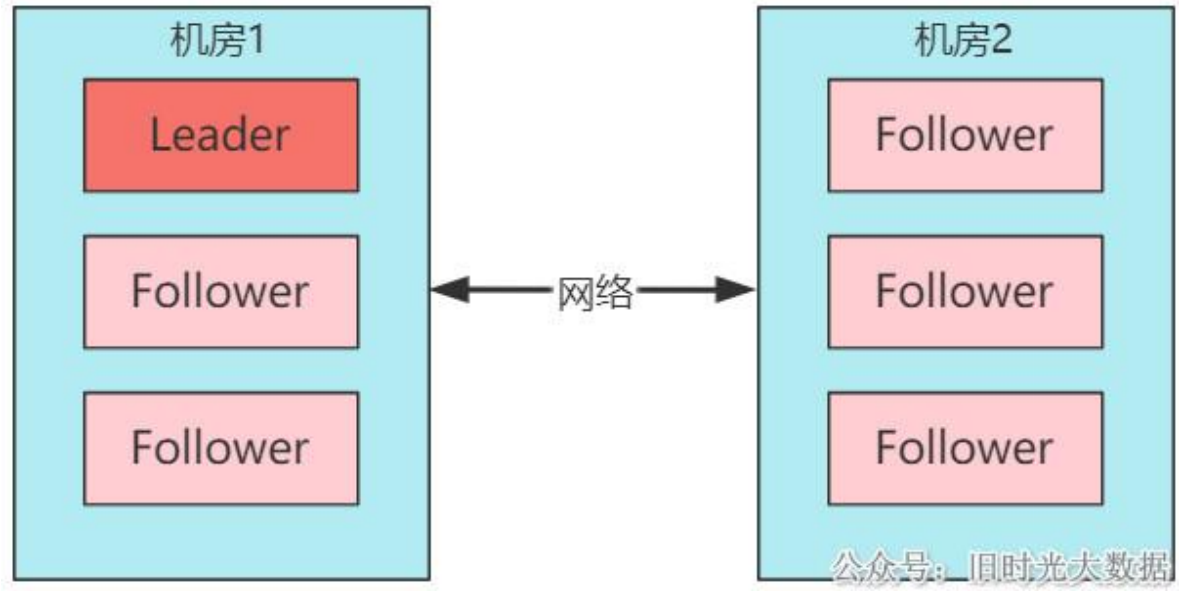
```
this.half = n/2;
return (set.size() > half);
```

举个简单的例子： 如果现在集群中有5台zkServer，那么half=5/2=2，那么也就是说，领导者选举的过程中至少要有三台zkServer投了同一个zkServer，才会符合过半机制，才能选出来一个Leader。

那么有一个问题我们想一下，**选举的过程中为什么一定要有一个过半机制验证?** 因为这样不需要等待所有zkServer都投了同一个zkServer就可以选举出来一个Leader了，这样比较快，所以叫快速领导者选举算法呗。

那么再来想一个问题，**过半机制中为什么是大于，而不是大于等于呢?**

这就是更脑裂问题有关系了，比如回到上文出现脑裂问题的场景：

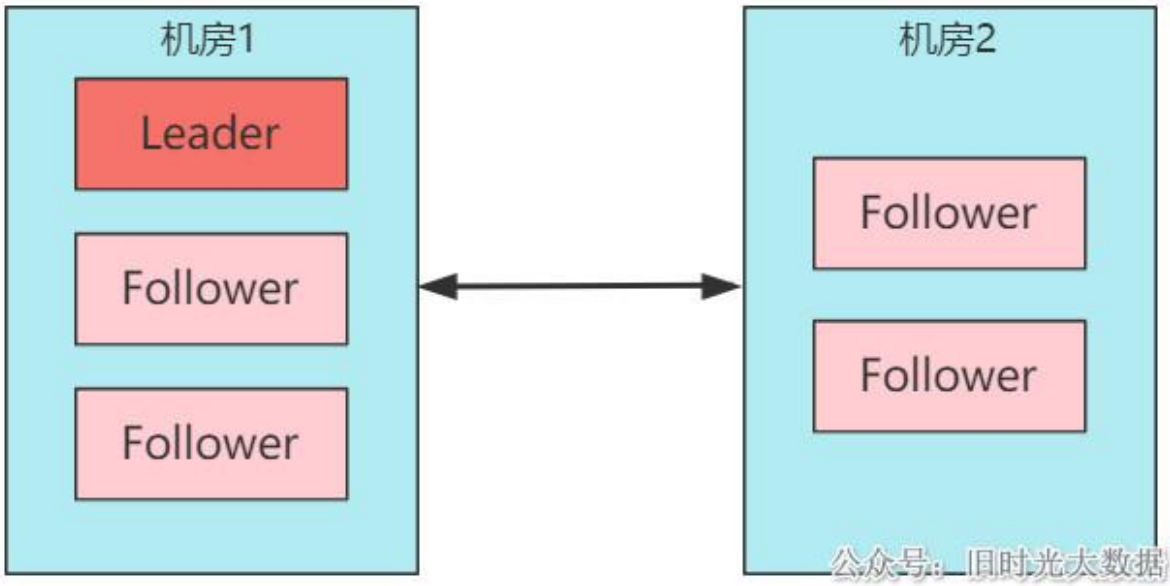


当机房中间的网络断掉之后，机房1内的三台服务器会进行领导者选举，但是此时过半机制的条件是set.size() > 3，也就是说至少要4台zkServer才能选出来一个Leader，所以对于机房1来说它不能选出一个Leader，同样机房2也不能选出一个Leader，这种情况下整个集群当机房间的网络断掉后，整个集群将没有Leader。

而如果过半机制的条件是set.size() >= 3，那么机房1和机房2都会选出一个Leader，这样就出现了脑裂。所以我们就知道了，为什么过半机制中是**大于**，而不是**大于等于**。就是为了防止脑裂。

如果假设我们现在只有5台机器，也部署在两个机房：

公众号：旧时光大数据



此时过半机制的条件是set.size() > 2，也就是至少要3台服务器才能选出一个Leader，此时机房件的网络断开了，对于机房1来说是没有影响的，Leader依然还是Leader，对于机房2来说是选不出来Leader的，此时整个集群中只有一个Leader。

所以，我们可以总结得出，**有了过半机制，对于一个Zookeeper集群，要么没有Leader，要没只有1个Leader，这样就避免了脑裂问题。**

Zookeeper的Paxos算法

问过的一些公司：字节，ebay

参考答案：

1、背景介绍

Paxos算法是分布式技术大师Lamport提出的，主要目的是通过这个算法，让参与分布式处理的每个参与者逐步达成一致意见。用好理解的方式来说，就是在一个选举过程中，让不同的选民最终做出一致的决定。

Lamport为了讲述这个算法，假想了一个叫做Paxos的希腊城邦进行选举的情景，这个算法也是因此而得名。在他的假想中，这个城邦要采用民主提议和投票的方式选出一个最终的决议，但由于城邦的居民没有 人愿意把全部时间和精力放在这种事情上，所以他们只能不定时的来参加提议，不定时来了解提议、投票 进展，不定时的表达自己的投票意见。Paxos算法的目标就是让他们按照少数服从多数的方式，最终达成 一致意见。

2、Paxos算法的具体情况

1）在整个提议和投票过程中，主要的角色就是“提议者”（向“接受者”提出提议）和“接受者”（收 到“提议者”的提议后，向“提议者”表达自己的意见）。

2）整个算法的大致过程为：

第一阶段：因为存在多个“提议者”，如果都提意见，那么“接受者”接受谁的不接受谁的？太混乱了。 所以，要先明确哪个“提议者”是意见领袖有权提出提议，未来，“接受者”们就主要处理这个“提议 者”的提议了（这样，也可以在提出提议时就尽量让意见统一，谋求尽早形成多数派）。

第二阶段：由上阶段选出的意见领袖提出提议，“接受者”反馈意见。如果多数“接受者”接受了一个提议，那么提议就通过了。

3、必须要了解的其他相关背景

公众号：旧时光大数据

1）怎么明确意见领袖呢？通过编号。每个“提议者”在第一阶段先报个号，谁的号大，谁就是意见领袖。 如果不好理解，可以想象为贿选。每个提议者先拿着钞票贿赂一圈“接受者”，谁给的钱多，第二阶段“接受者”就听谁的。（注：这里和下文提到的“意见领袖”，并不是一个新的角色，而是代表在那一 轮贿赂成功的“提议者”。所以，请把意见领袖理解为贿赂中胜出的“提议者”即可）

2）有个跟选举常识不一样的地方，就是每个“提议者”不会执着于让自己的提议通过，而是每个“提议者”会执着于让提议尽快达成一致意见。所以，为了这个目标，如果“提议者”在贿选的时候，发现“接 受者”已经接受过前面意见领袖的提议了，即便“提议者”贿选成功，也会默默的把自己的提议改为前面 意见领袖的提议。所以一旦贿赂成功，胜出的“提议者”再提出提议，提议内容也是前面意见领袖的提议（这样，在谋求尽早形成多数派的路上，又前进了一步）。

3）钱的多少很重要，如果钱少了，无论在第一还是第二阶段“接受者”都不会尿你，直接拒绝。

4）上面2）中讲到，如果“提议者”在贿选时，发现前面已经有意见领袖的提议，那就将自己的提议默默改成前面意见领袖的提议。这里有一种情况，如果你是“提议者”，在贿赂的时候，“接受者1”跟你 说“他见过的意见领袖的提议是方案1”，而“接受者2”跟你说“他见过的意见领袖提议是方案2”，你 该怎么办？这时的原则也很简单，还是：钱的多少很重要！你判断一下是“接受者1”见过的意见领袖有 钱，还是“接受者2”见过的意见领袖有钱？如何判断呢？因为“接受者”在被“提议者”贿赂的时候，自 己会记下贿赂的金额。所以当你贿赂“接受者”时，一旦你给的贿赂多而胜出，“接受者”会告诉你两件 事情：a.前任意见领袖的提议内容（如果有的话），b.前任意见领袖当时贿赂了多少钱。这样，再面对刚才 的情景时，你只需要判断一下“接受者1”和“接受者2”告诉你的信息中，哪个意见领袖当时给的钱多， 那你就默默的把自己的提议，改成那个意见领袖的提议。

5）最后这一部分最有意思，但描述起来有点绕，如果不能一下子就理解可以先看后面的例子。在整个选举 过程中，每个人谁先来谁后到，“接受者”什么时间能够接到“提议者”的信息，是完全不可控的。所以 很可能一个意见领袖已经产生了，但是由于这个意见领袖的第二阶段刚刚开始，绝大部分“接受者”还没 有收到这个意见领袖的提议。结果，这时突然冲进来了一个新的土豪“提议者”，那么这个土豪“提议者”也是有机会让自己的提议胜出的！这时就形成了一种博弈：

- 上一个意见领袖要赶在土豪“提议 者”贿赂到“接受者”前，赶到“接受者”面前让他接受自己的提议，否则会因为自己的之前贿赂的钱比 土豪少而被拒绝。
- 土豪“提议者”要赶在上一个意见领袖将提议传达给“接受者”前，贿赂到“接受 者”，否则土豪“提议者”即便贿赂成功，也要默默的将自己的提议改为前任意见领袖的提议。这整个博 弈的过程，最终就看这两个“提议者”谁的进展快了。但最终一定会有一个意见领袖，先得到多数“接受 者”的认可，那他的提议就胜出了。

4、小结

总结一下，其实Paxos算法就下面这么几个原则：

1）Paxos算法包括两个阶段：第一个阶段主要是贿选，还没有提出提议；第二个阶段主要根据第一阶段的 结果，明确接受谁的提议，并明确提议的内容是什么（这个提议可能是贿选胜出“提议者”自己的提议， 也可能是前任意见领袖的提议，具体是哪个提议，见下面第3点原则）。

2）编号（贿赂金额）很重要，无论在哪个阶段，编号（贿赂金额）小的，都会被鄙视（被拒绝）。

3）在第一阶段中，一旦“接受者”已经接受了之前意见领袖的提议，那后面再来找这个“接受者”的“提议者”，即便在贿赂中胜出，也要被洗脑，默默将自己的提议改为前任意见领袖的提议，然后他会在第二阶段提出该提议（也就是之前意见领袖的提议，以力争让大家的意见趋同）。如果“接受者”之前没有接受过任何提议，那贿选胜出的“提议者”就可以提出自己的提议了。

Zookeeper的协议有哪些?

问过的一些公司：网易

参考答案：

公众号：旧时光大数据

ZAB协议：Zab是一个高性能的广播协议，主要用于主备系统，它是专门为ZooKeeper设计的。

Zookeeper如何保证数据的一致性?

可回答：Zookeeper的一致性协议

问过的一些公司：字节x2，网易

参考答案：

依赖于ZAB协议来保证数据一致性，ZAB协议是伪分布式协调服务Zookeeper专门设计的一种崩溃恢复的原子广播协议。

两种基本的模式：

- 崩溃恢复
- 消息广播

这两个模式是相辅相成的。

消息广播模式就是zookeeper不出现任何问题，并且正常工作的模式。

崩溃恢复看字面意思就是当zookeeper 出现故障时用于恢复的。

保证数据的一致性有两种情况：第一是故障恢复阶段的数据同步，即新leader选举之后的数据同步；第二是leader处理完事务请求与follower保持数据同步。

故障恢复阶段的数据同步：leader为每个follower准备一个队列，将没有被同步的事务以proposal请求形式放入队列，然后对每个事务请求追加一个commit请求，表示该事务被提交，完成数据的同步。

leader处理事务请求与follower保持数据同步：leader收到事务请求之后，将事务转化为proposal，由leader为每个follower创建一个队列，将事务请求按顺序放入请求队列当中，之后按事务请求向follower发出消息广播该提案，follower收到后会以事务的形式写入本地日志，并且向leader发送ack请求，leader收到一半以上的follower响应，会向其他follower发出commit消息，同时leader提交这个提案。

Zookeeper的数据存储在什么地方?

问过的一些公司：字节x2

参考答案：

1、内存数据

Zookeeper的数据模型是树结构，在内存数据库中，存储了整棵树的内容，包括所有的节点路径、节点数据、ACL信息，Zookeeper会定时将这个数据存储到磁盘上。

DataTree

DataTree是内存数据存储的核心，是一个树结构，代表了内存中一份完整的数据。DataTree不包含任何与网络、客户端连接及请求处理相关的业务逻辑，是一个独立的组件。

DataNode

DataNode是数据存储的最小单元，其内部除了保存了结点的数据内容、ACL列表、节点状态之外，还记录了父节点的引用和子节点列表两个属性，其也提供了对子节点列表进行操作的接口。

ZKDatabase

Zookeeper的内存数据库，管理Zookeeper的所有会话、DataTree存储和事务日志。ZKDatabase会定时向磁盘dump快照数据，同时在Zookeeper启动时，会通过磁盘的事务日志和快照文件恢复成一个完整的内存数据库。

2、事务日志

在配置Zookeeper集群时需要配置**dataDir目录，其用来存储事务日志文件。**也可以为事务日志单独分配一个文件存储目录：dataLogDir。若配置dataLogDir为/home/admin/zkData/zk_log，那么Zookeeper在运行过程中会在该目录下建立一个名字为version-2的子目录，该目录确定了当前Zookeeper使用的事务日志格式版本号，当下次某个Zookeeper版本对事务日志格式进行变更时，此目录也会变更，即在version-2子目录下会生成一系列文件大小一致(64MB)的文件。

3、snapshot-数据快照

数据快照是Zookeeper数据存储中非常核心的运行机制，**数据快照用来记录Zookeeper服务器上某一时刻的全量内存数据内容，并将其写入指定的磁盘文件中。**

与事务文件类似，Zookeeper快照文件也可以指定特定磁盘目录，通过dataDir属性来配置。若指定dataDir为/home/admin/zkData/zk_data，则在运行过程中会在该目录下创建version-2的目录，该目录确定了当前Zookeeper使用的快照数据格式版本号。在Zookeeper运行时，会生成一系列文件。

Zookeeper从三台扩容到七台怎么做?

问过的一些公司：蘑菇街

参考答案：

1）先检查三台设备的状态，两台为follower，一台为leader；

2）修改节点数量，由三台增加到七台，修改相应的zoo.cfg文件；

3）完成相关配置后，启动对应zk实例，确认启动ok，节点四到七依次验证实例状态；

4）在leader节点上查看目前状态同步的follower数，确认新增节点已经成功加入集群；

5）确认新增节点加入集群后，滚动更新原有集群的配置，并重启，这里需要注意，在重启follwer节点的时候并无任何影响，不过在重启leader节点的时候，这个时候会触发一次新的leader选举，zxid最新的默认优先当选新的leader，当zxid相同，myid最大的优先当选新的leader。

Hive面试题

说下为什么要使用Hive？Hive的优缺点？Hive的作用是什么?

问过的一些公司：头条，字节x2，阿里

参考答案：

1、为什么要使用Hive?

Hive是Hadoop生态系统中比不可少的一个工具，它提供了一种SQL(结构化查询语言)方言，可以查询存储在Hadoop分布式文件系统（HDFS）中的数据或其他和Hadoop集成的文件系统，如MapR-FS、Amazon的S3和像HBase（Hadoop数据仓库）和Cassandra这样的数据库中的数据。

公众号：旧时光大数据

大多数数据仓库应用程序都是使用关系数据库进行实现的，并使用SQL作为查询语言。**Hive降低了将这些应用程序转移到Hadoop系统上的难度。**凡是会使用SQL语言的开发人员都可以很轻松的学习并使用Hive。如果没有Hive，那么这些用户就必须学习新的语言和工具，然后才能应用到生产环境中。另外，**相比其他工具，Hive更便于开发人员将基于SQL的应用程序转移到Hadoop中。**如果没有Hive，那么开发者将面临一个艰巨的挑战，如何将他们的SQL应用程序移植到Hadoop上。

2、Hive优缺点

优点

1）操作接口采用类SQL语法，提供快速开发的能力（简单、容易上手）。

2）避免了去写MapReduce，减少开发人员的学习成本。

3）Hive的执行延迟比较高，因此Hive常用于数据分析，对实时性要求不高的场合。

4）Hive优势在于处理大数据，对于处理小数据没有优势，因为Hive的执行延迟比较高。

5）Hive支持用户自定义函数，用户可以根据自己的需求来实现自己的函数。

缺点

1）Hive的HQL表达能力有限

- 迭代式算法无法表达
- 数据挖掘方面不擅长，由于MapReduce数据处理流程的限制，效率更高的算法却无法实现。

2）Hive的效率比较低

- Hive自动生成的MapReduce作业，通常情况下不够智能化
- Hive调优比较困难，粒度较粗

Hive不是一个完整的数据库。Hadoop以及HDFS的设计本身约束和局限性地限制了Hive所能胜任的工作。其中最大的限制就是Hive不支持记录级别的更新、插入或者删除操作。但是用户可以通过查询生成新表或者将查询结果导入到文件中。同时，因为Hadoop是面向批处理的系统，而MapReduce任务（job）的启动过程需要消耗较长的时间，所以Hive查询延时比较严重。传统数据库中在秒级别可以完成的查询，在Hive中，即使数据集相对较小，往往也需要执行更长的时间。

3、Hive的作用

Hive是由Facebook开源**用于解决海量结构化日志的数据统计工具。**

Hive是**基于Hadoop的一个数据仓库工具**，可以将结构化的数据文件映射为一张表，并提供类SQL查询功能。

Hive的**本质是将HQL转化成MapReduce程序**

- Hive处理的数据存储在HDFS
- Hive分析数据底层的实现是MapReduce
- 执行程序运行在Yarn上

说下Hive是什么？跟数据仓库区别？

可回答：1）说下对Hive的了解；2）介绍下Hive

问过的一些公司：美团，360，字节，大华(2021.07)，海康(2021.09)

参考答案：

Hive是由Facebook开源**用于解决海量结构化日志的数据统计工具。**

公众号：旧时光大数据

Hive是**基于Hadoop的一个数据仓库工具**，可以将结构化的数据文件映射为一张表，并提供类SQL查询功能。

Hive的**本质是将HQL转化成MapReduce程序**

- Hive处理的数据存储在HDFS
- Hive分析数据底层的实现是MapReduce
- 执行程序运行在Yarn上

数据仓库是**为企业所有级别的决策制定过程，提供所有类型数据支持的战略集合。**它是**单个数据存储，出于分析性报告和决策支持目的而创建。**为需要业务智能的企业，提供指导业务流程改进、监视时间、成本、质量以及控制。

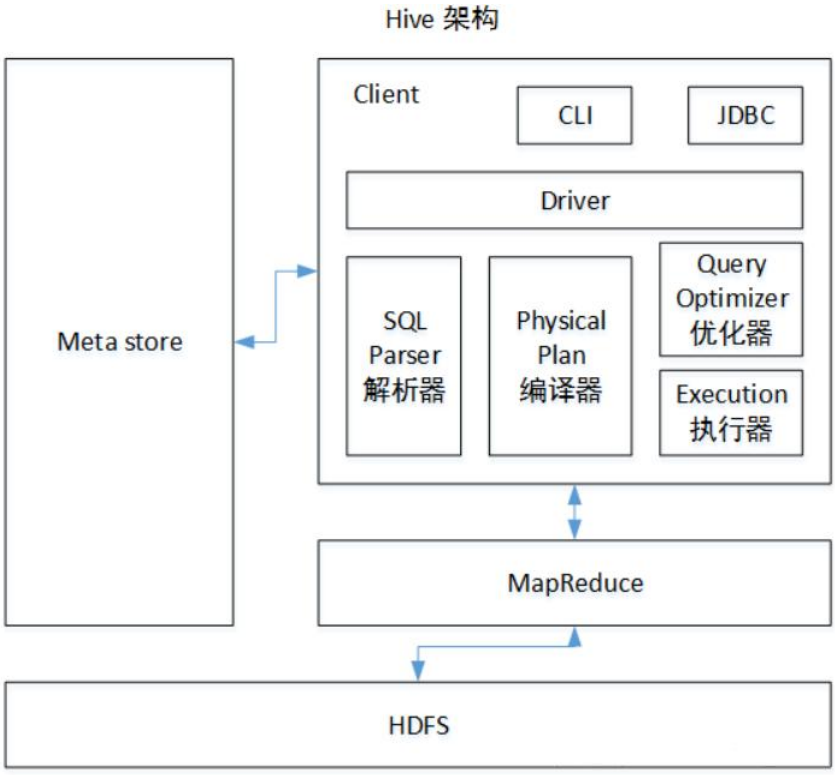
数据仓库存在的意义在于对企业的所有数据进行汇总，为企业各个部门提供统一的， 规范的数据出口。

Hive架构

可回答：Hive的基本架构，角色，与HDFS的关系?

问过的一些公司：腾讯微众，好未来，恒生(2021.09)

参考答案：



用户接口：Client

CLI（command-line interface）、JDBC/ODBC(jdbc访问hive)、WEBUI（浏览器访问hive）

元数据：Metastore

元数据包括：表名、表所属的数据库（默认是default）、表的拥有者、列/分区字段、表的类型（是否是外部表）、表的数据所在目录等；

默认存储在自带的derby数据库中，推荐使用MySQL存储Metastore

Hadoop

公众号：旧时光大数据

使用HDFS进行存储，使用MapReduce进行计算。

驱动器：Driver

1）解析器（SQL Parser）：将SQL字符串转换成抽象语法树AST，这一步一般都用第三方工具库完成，比如antlr；

对AST进行语法分析，比如表是否存在、字段是否存在、SQL语义是否有误。

2）编译器（Physical Plan）：将AST编译生成逻辑执行计划。

3）优化器（Query Optimizer）：对逻辑执行计划进行优化。

4）执行器（Execution）：把逻辑执行计划转换成可以运行的物理计划。对于Hive来说，就是MR/Spark。

Hive内部表和外部表的区别?

问过的一些公司：字节，阿里社招，快手，美团x2，蘑菇街x2，祖龙娱乐，作业帮x3，360，小米，竞技世界，猿辅导，好未来，多益，多益(2021.09)，富途，冠群驰聘，大华(2021.07)，字节(2021.08)

参考答案：

内部表（managed table）：未被external修饰

外部表（external table）：被external修饰

区别：

1）内部表数据由Hive自身管理，外部表数据由HDFS管理；

2）内部表数据存储的位置是hive.metastore.warehouse.dir（默认：/user/hive/warehouse），外部表数据的存储位置由自己制定（如果没有LOCATION，Hive将在HDFS上的/user/hive/warehouse文件夹下以外部表的表名创建一个文件夹，并将属于这个表的数据存放在这里）；

3）删除内部表会直接删除元数据（metadata）及存储数据；删除外部表仅仅会删除元数据，HDFS上的文件并不会被删除；

4）对内部表的修改会将修改直接同步给元数据，而对外部表的表结构和分区进行修改，则需要修复（MSCK REPAIR TABLE table_name;）。

为什么内部表的删除，就会将数据全部删除，而外部表只删除表结构? 为什么用外部表更好?

问过的一些公司：大华

参考答案：

外部表和内部表创建表以及删除时的区别

- **创建表时：**创建内部表时，会将数据移动到数据仓库指向的路径；若创建外部表，仅记录数据所在的路径，不对数据的位置做任何改变。
- **删除表时：**在删除表的时候，内部表的元数据和数据会被一起删除， 而外部表只删除元数据，不删除数据。这样外部表相对来说更加安全些，数据组织也更加灵活，方便共享源数据。

外部表的优点

公众号：旧时光大数据

1）外部表不会加载数据到Hive的默认仓库（挂载数据），减少了数据的传输，同时还能和其他外部表共享数据。

2）使用外部表，Hive不会修改源数据，不用担心数据损坏或丢失。

3）Hive在删除外部表时，删除的只是表结构，而不会删除数据。

Hive建表语句？创建表时使用什么分隔符？

问过的一些公司：作业帮，美团，京东

参考答案：

1、建表语句

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] table_name
[(col_name data_type [COMMENT col_comment], ...)]
[COMMENT table_comment]
[PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
[CLUSTERED BY (col_name, col_name, ...)
[SORTED BY (col_name [ASC|DESC], ...)] INTO num_buckets BUCKETS]
[ROW FORMAT row_format]
[STORED AS file_format]
[LOCATION hdfs_path]
[TBLPROPERTIES (property_name=property_value, ...)]
[AS select_statement]
```

字段解释说明

（1）CREATE TABLE 创建一个指定名字的表。如果相同名字的表已经存在，则抛出异常； 用户可以用 IF NOT EXISTS 选项来忽略这个异常。

（2）EXTERNAL 关键字可以让用户创建一个外部表，在建表的同时可以指定一个指向实 际数据的路径（LOCATION），在删除表的时候，内部表的元数据和数据会被一起删除，而外 部表只删除元数据，不删除数据。

（3）COMMENT：为表和列添加注释。

（4）PARTITIONED BY：创建分区表

（5）CLUSTERED BY：创建分桶表

（6）SORTED BY：不常用，对桶中的一个或多个列另外排序

（7）ROW FORMAT

```
DELIMITED [FIELDS TERMINATED BY char] [COLLECTION ITEMS TERMINATED BY char] [MAP KEYS TERMINATED BY char] [LINES TERMINATED BY char]
| SERDE serde_name [WITH SERDEPROPERTIES (property_name=property_value, property_name=property_value, ...)]
```

用户在建表的时候可以自定义SerDe或者使用自带的SerDe。如果没有指定ROW FORMAT或者ROW FORMAT DELIMITED，将会使用自带的SerDe。在建表的时候，用户还需要为表指定列，用户在指定表的列的同时也会指定自定义的SerDe，Hive通过SerDe确定表的具体的列的数据。

SerDe是Serialize/Deserilize的简称，hive使用Serde进行行对象的序列与反序列化。

（8）STORED AS：指定存储文件类型 常用的存储文件类型：SEQUENCEFILE（二进制序列文件）、TEXTFILE（文本）、RCFILE（列式存储格式文件）。

公众号：旧时光大数据

如果文件数据是纯文本，可以使用 STORED AS TEXTFILE。如果数据需要压缩，使用 STORED AS SEQUENCEFILE。

（9）LOCATION：指定表在HDFS上的存储位置。

（10）AS：后跟查询语句，根据查询结果创建表。

（11）LIKE：允许用户复制现有的表结构，但是不复制数据。

2、分隔符

我们在建表的时候就指定了导入数据时的分隔符，建表的时候会有三种场景需要考虑：

- 正常建表(default)；
- 指定特定的特殊符号作为分隔符；
- 使用多字符作为分隔符。

1）正常建表，采用默认的分隔符

hive 默认的字段分隔符为ascii码的控制符\001,建表的时候用fields terminated by '\001',如果要测试的话，造数据在vi 打开文件里面，用ctrl+v然后再ctrl+a可以输入这个控制符\001。按顺序，\002的输入方式为ctrl+v，ctrl+b。以此类推。

2）指定特定的特殊符号作为分隔符

```
CREATE TABLE test(id int, name string ,tel string) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'LINES TERMINATED BY '\n'STORED AS TEXTFILE;
```

上面使用了'\t'作为了字段分隔符，'\n'作为换行分隔符。如果有特殊需求，自己动手改一下这两个符号就行了。

3、使用多字符作为分隔符

假设我们使用【##】来作为字段分隔符，【\n】作为换行分隔符，则这里有两个方法：

使用**MultiDelimitSerDe**的方法来实现：

```
CREATE TABLE test(id int, name string ,tel string) ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.MultiDelimitSerDe' WITH SERDEPROPERTIES ("field.delim"="##") LINES TERMINATED BY '\n'STORED AS TEXTFILE;
```

使用**RegexSerDe**的方法实现：

```
CREAET TABLE test(id int, name string ,tel string) ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe' WITH SERDEPROPERTIES ("input.regex" = "^(.*)\\#\\#(.*)$") LINES TERMINATED BY '\n'STORED AS TEXTFILE;
```

Hive删除语句外部表删除的是什么？

问过的一些公司：作业帮

参考答案：

外部表只删除元数据，不删除数据

Hive数据倾斜以及解决方案

公众号：旧时光大数据

可回答：1）Hive使用过程中遇到了什么问题？2）两张大表的join怎么执行的；综合去回答，不要太死板

问过的一些公司：字节，字节(2021.07)-(2021.08)x2，美团x7，阿里云x2，快手x3，小米，有赞x2，腾讯，顺丰x2，携程x2，东软集团，多益，多益(2021.09)，抖音，一点咨询，作业帮，东方头条，网易，好未来，端点数据(2021.07)，58同城(2021.08)，海康(2021.08)，阿里蚂蚁(2021.08)，兴业数金(2021.09)，有赞(2021.09)，京东(2021.09)，唯品会(2021.10)，联想(2021.10)

参考答案：

1、什么是数据倾斜

数据倾斜主要表现在，map/reduce程序执行时，reduce节点大部分执行完毕，但是有一个或者几个reduce节点运行很慢，导致整个程序的处理时间很长，这是因为某一个key的条数比其他key多很多(有时是百倍或者千倍之多)，这条Key所在的reduce节点所处理的数据量比其他节点就大很多，从而导致某几个节点迟迟运行不完。

2、数据倾斜的原因

一些操作：

关键词	情形	后果
Join	其中一个表较小，但是key集中	分发到某一个或几个Reduce上的数据远高于平均值
大表与大表，但是分桶的判断字段0值或空值过多	这些空值都由一个reduce处理，灰常慢	
group by	group by 维度过小，某值的数量过多	处理某值的reduce灰常耗时
Count Distinct	某特殊值过多	处理此特殊值的reduce耗时

原因：

- key分布不均匀
- 业务数据本身的特性
- 建表时考虑不周
- 某些SQL语句本身就有数据倾斜

现象：

- 任务进度长时间维持在99%（或100%），查看任务监控页面，发现只有少量（1个或几个）reduce子任务未完成。因为其处理的数据量和其他reduce差异过大。
- 单一reduce的记录数与平均记录数差异过大，通常可能达到3倍甚至更多。 最长时长远大于平均时长。

3、数据倾斜的解决方案

1）参数调节

```
hive.map.aggr = true
```

Map 端部分聚合，相当于Combiner

```
hive.groupby.skewindata=true
```

公众号：旧时光大数据

有数据倾斜的时候进行负载均衡，当选项设定为true，生成的查询计划会有两个MR Job。第一个MR Job中，Map 的输出结果集合会随机分布到Reduce中，每个Reduce做部分聚合操作，并输出结果，这样处理的结果是相同的Group By Key有可能被分发到不同的Reduce中，从而达到负载均衡的目的；第二个MR Job再根据预处理的数据结果按照Group By Key分布到Reduce中（这个过程可以保证相同的Group By Key被分布到同一个Reduce中），最后完成最终的聚合操作。

2）SQL语句调节

如何join：

关于驱动表的选取，选用join key分布最均匀的表作为驱动表，做好列裁剪和filter操作，以达到两表做join的时候，数据量相对变小的效果。

大小表Join：

使用map join让小的维度表（1000条以下的记录条数）先进内存。在map端完成reduce。

大表Join大表：

把空值的key变成一个字符串加上随机数，把倾斜的数据分到不同的reduce上，由于null值关联不上，处理后并不影响最终结果。

count distinct大量相同特殊值：

count distinct时，将值为空的情况单独处理，如果是计算count distinct，可以不用处理，直接过滤，在最后结果中加1。如果还有其他计算，需要进行group by，可以先将值为空的记录单独处理，再和其他计算结果进行union。

group by维度过小：

采用sum() group by的方式来替换count(distinct)完成计算。

特殊情况特殊处理：

在业务逻辑优化效果的不大情况下，有些时候是可以将倾斜的数据单独拿出来处理。最后union回去。

4、典型的业务场景

1）空值产生的数据倾斜

场景：如日志中，常会有信息丢失的问题，比如日志中的 user_id，如果取其中的 user_id 和 用户表中的 user_id 关联，会碰到数据倾斜的问题。

解决方法一：user_id为空的不参与关联

```
select * from log a
  join users b
  on a.user_id is not null
  and a.user_id = b.user_id
union all
select * from log a
  where a.user_id is null;
```

解决方法二：赋与空值分新的key值

```
select *
  from log a
  left outer join users b
  on case when a.user_id is null then concat('hive',rand() ) else a.user_id end =
b.user_id;
```

结论：

公众号：旧时光大数据

方法2比方法1效率更好，不但io少了，而且作业数也少了。解决方法一中log读取两次，jobs是2。解决方法二job数是1。这个优化适合无效id (比如 -99 , '', null 等) 产生的倾斜问题。把空值的key变成一个字符串加上随机数，就能把倾斜的数据分到不同的reduce上，解决数据倾斜问题。

2）不同数据类型关联产生数据倾斜

场景：用户表中user_id字段为int，log表中user_id字段既有string类型也有int类型。当按照user_id进行两个表的Join操作时，默认的Hash操作会按int型的id来进行分配，这样会导致所有string类型id的记录都分配到一个Reducer中。

解决方法：把数字类型转换成字符串类型

```
select * from users a
  left outer join logs b
  on a.usr_id = cast(b.user_id as string);
```

3）小表不小不大，怎么用 map join 解决倾斜问题

使用map join解决小表（记录数少）关联大表的数据倾斜问题，这个方法使用的频率非常高，但如果小表很大，大到map join会出现bug或异常，这时就需要特别的处理。

例子：

```
select * from log a
  left outer join users b
  on a.user_id = b.user_id;
```

users表有600w+的记录，把users分发到所有的map上也是个不小的开销，而且map join不支持这么大的小表。如果用普通的join，又会碰到数据倾斜的问题。

解决方法：

```
select /*+mapjoin(x)*/* from log a
  left outer join (
    select  /*+mapjoin(c)*/d.*
      from ( select distinct user_id from log ) c
      join users d
      on c.user_id = d.user_id
    ) x
  on a.user_id = b.user_id;
```

假如，log里user_id有上百万个，这就又回到原来map join问题。所幸，每日的会员uv不会太多，有交易的会员不会太多，有点击的会员不会太多，有佣金的会员不会太多等等。所以这个方法能解决很多场景下的数据倾斜问题。

Hive如果不用参数调优，在map和reduce端应该做什么

问过的一些公司：阿里

参考答案：

1、map阶段优化

Map阶段的优化，主要是确定合适的map数。那么首先要了解map数的计算公式

```
num_reduce_tasks = min[${hive.exec.reducers.max},
(${input.size}/${hive.exec.reducers.bytes.per.reducer})]
```

- mapred.min.split.size: 指的是数据的最小分割单元大小；min的默认值是1B
- mapred.max.split.size: 指的是数据的最大分割单元大小；max的默认值是256MB
- dfs.block.size: 指的是HDFS设置的数据块大小。个已经指定好的值，而且这个参数默认情况下hive是识别不到的。

通过调整max可以起到调整map数的作用，减小max可以增加map数，增大max可以减少map数。需要提醒的是，**直接调整mapred.map.tasks这个参数是没有效果的。**

2、reduce阶段优化

这里说的reduce阶段，是指前面流程图中的reduce phase（实际的reduce计算）而非图中整个reduce task。Reduce阶段优化的主要工作也是选择合适的reduce task数量, 与map优化不同的是，reduce优化时，可以直接设置mapred.reduce.tasks参数从而直接指定reduce的个数。

```
num_reduce_tasks = min[${hive.exec.reducers.max},
(${input.size}/${hive.exec.reducers.bytes.per.reducer})]
```

`hive.exec.reducers.max`：此参数从Hive 0.2.0开始引入。在Hive 0.14.0版本之前默认值是999；而从Hive 0.14.0开始，默认值变成了1009，这个参数的含义是最多启动的Reduce个数

`hive.exec.reducers.bytes.per.reducer`：此参数从Hive 0.2.0开始引入。在Hive 0.14.0版本之前默认值是1G(1,000,000,000)；而从Hive 0.14.0开始，默认值变成了256M(256,000,000)，可以参见HIVE-7158和HIVE-7917。这个参数的含义是每个Reduce处理的字节数。比如输入文件的大小是1GB，那么会启动4个Reduce来处理数据。

也就是说，根据输入的数据量大小来决定Reduce的个数，默认Hive.exec.Reducers.bytes.per.Reducer为1G，而且Reduce个数不能超过一个上限参数值，这个参数的默认取值为999。所以我们可以调整Hive.exec.Reducers.bytes.per.Reducer来设置Reduce个数。

注意：

- Reduce的个数对整个作业的运行性能有很大影响。如果Reduce设置的过大，**那么将会产生很多小文件，对NameNode会产生一定的影响**，而且整个作业的运行时间未必会减少；如果Reduce设置的过小，那么单个Reduce处理的数据将会加大，**很可能会引起OOM异常。**
- 如果设置了`mapred.reduce.tasks/mapreduce.job.reduces`参数，那么**Hive会直接使用它的值作为Reduce的个数**；
- 如果mapred.reduce.tasks/mapreduce.job.reduces的值没有设置（也就是-1），那么Hive会根据输入文件的大小估算出Reduce的个数。根据输入文件估算Reduce的个数可能未必很准确，因为Reduce的输入是Map的输出，而Map的输出可能会比输入要小，所以最准确的数根据Map的输出估算Reduce的个数。

Hive的用户自定义函数实现步骤与流程

问过的一些公司：阿里，恒生(2021.09)

参考答案：

1、如何构建UDF?

用户创建的UDF使用过程如下：

第一步：继承UDF或者UDAF或者UDTF，实现特定的方法；

第二步：将写好的类打包为jar，如hivefirst.jar；

第三步：进入到Hive外壳环境中，利用add jar /home/hadoop/hivefirst.jar注册该jar文件；

公众号：旧时光大数据

第四步：为该类起一个别名，create temporary function mylength as 'com.whut.StringLength'，这里注意UDF只是为这个Hive会话临时定义的；

第五步：在select中使用mylength()。

2、函数自定义实现步骤

1）继承Hive提供的类

```
org.apache.hadoop.hive.ql.udf.generic.GenericUDF
org.apache.hadoop.hive.ql.udf.generic.GenericUDTF;
```

2）实现类中的抽象方法

3）在 hive 的命令行窗口创建函数添加 jar

```
add jar linux_jar_path
# 创建 function
create [temporary] function [dbname.]function_name AS class_name;
```

4）在 hive 的命令行窗口删除函数

```
drop [temporary] function [if exists] [dbname.]function_name;

```

3、自定义UDF案例

1）需求

自定义一个UDF实现计算给定字符串的长度，例如：

```
hive(default)> select my_len("abcd");
4

```

2）导入依赖

```
<dependencies>
    <dependency>
        <groupId>org.apache.hive</groupId>
        <artifactId>hive-exec</artifactId>
        <version>3.1.2</version>
    </dependency>
</dependencies>

```

3）创建一个类，继承于Hive自带的UDF

```
/**
* 自定义 UDF 函数，需要继承 GenericUDF 类
* 需求: 计算指定字符串的长度
*/
public class MyStringLength extends GenericUDF {
    /**
    *
    *@param arguments 输入参数类型的鉴别器对象
    * @return 返回值类型的鉴别器对象
    *@throws UDFArgumentException
    */
    @Override
```

```
13      public ObjectInspector initialize(ObjectInspector[] arguments) throws
    UDFArgumentException {
14          // 判断输入参数的个数
15          if(arguments.length !=1) {
16              throw new UDFArgumentLengthException("Input Args Length Error!!!");
17          }
18  // 判断输入参数的类型
19
20
    if(!arguments[0].getCategory().equals(ObjectInspector.Category.PRIMITIVE)
21            ) {
22              throw new UDFArgumentTypeException(0,"Input Args Type Error!!!");
23          }
24  //函数本身返回值为 int，需要返回 int 类型的鉴别器对象
25          return PrimitiveObjectInspectorFactory.javaIntObjectInspector;
26      }
27
28      /**
29      * 函数的逻辑处理
30      *@param arguments 输入的参数
31      *@return 返回值
32      *@throws HiveException
33      */
34      @Override
35      public Object evaluate(DeferredObject[] arguments) throws HiveException {
36          if(arguments[0].get() == null) {
37              return 0;
38          }
39          return arguments[0].get().toString().length();
40      }
41
42      @Override
43      public String getDisplayString(String[] children) {
44          return "";
45      }
46  }
47
```

4）打成jar包上传到服务器/opt/module/data/myudf.jar

5）将jar包添加到hive的classpath

```
1   hive (default)> add jar /opt/module/data/myudf.jar;
2
```

6）创建临时函数与开发好的java class关联

7）即可在hql中使用自定义的函数

```
1   hive (default)> select ename,my_len(ename) ename_len from emp;
2
```

Hive的三种自定义函数是什么？实现步骤与流程？它们之间的区别？作用是什么？

可回答：1）怎么实现Hive的UDF（UDF函数的开发流程）；2）Hive中有哪些UDF

公众号：旧时光大数据

问过的一些公司：阿里云x3，快手x3，祖龙娱乐，知乎，大华，跟谁学，滴滴，途牛，网易，富途，搜狐

参考答案：

Hive自带了一些函数，比如：max/min等，但是数量有限，自己可以通过自定义UDF来 方便的扩展。

当Hive提供的内置函数无法满足你的业务处理需要时，此时就可以考虑使用用户自定义函数（UDF：user-defined function）。

根据用户自定义函数类别分为以下三种：

- **UDF**（User-Defined-Function）：一进一出
- **UDAF**（User-Defined Aggregation Function）：聚集函数，多进一出
  - 类似于：count/max/min
- **UDTF**（User-Defined Table-Generating Functions）：一进多出
  - 如lateral view explode()

实现步骤和流程参考上一题

Hive的cluster by 、sort by、distribute by 、order by 区别?

可回答：Hive的排序函数

问过的一些公司：字节x3，百度社招，美团x3，头条，FreeWheel，蘑菇街，快手，百度校招，端点数据(2021.07)，大华(2021.07)，兴业数金(2021.09)，唯品会(2021.10)，好未来(2021.08)

参考答案：

共有四种排序：Order By，Sort By，Distribute By，Cluster By

1、Order By：全局排序

- 对输入的数据做排序，故此只有一个reducer(多个reducer无法保证全局有序)；
- 只有一个reducer，会导致当输入规模较大时，需要较长的计算时间；

1）使用 ORDER BY 子句排序

ASC（ascend）: 升序（默认）

DESC（descend）: 降序

2）ORDER BY 子句在SELECT语句的结尾

3）案例

查询员工信息按工资升序排列

```
select * from emp order by sal;
```

2、Sort By：非全局排序

- 在数据进入reducer前完成排序；
- 当mapred.reduce.tasks > 1时，只能保证每个reducer的输出有序，不保证全局有序；

3、Distribute By：分区排序

- 按照指定的字段对数据进行划分输出到不同的reduce中，通常是为了进行后续的聚集操作；
- 常和sort by一起使用，并且distribute by必须在sort by前面；

公众号：旧时光大数据

4、Cluster By

相当于distribute by+sort by，只能默认升序，不能使用倒序。

Hive分区和分桶的区别

可回答：Hive分区和分桶的逻辑

问过的一些公司：字节，小米，阿里云社招，京东x2，猿辅导，竞技世界，美团，抖音

参考答案：

1、定义上

分区

- Hive的分区使用HDFS的子目录功能实现。每一个子目录包含了分区对应的列名和每一列的值。
- Hive的分区方式：由于Hive实际是存储在HDFS上的抽象，Hive的一个分区名对应一个目录名，子分区名就是子目录名，并不是一个实际字段。
- 所以可以这样理解，当我们在插入数据的时候指定分区，其实就是新建一个目录或者子目录，或者在原有的目录上添加数据文件。

注意：partitned by子句中定义的列是表中正式的列（分区列），但是数据文件内并不包含这些列。

```
# 创建分区表
create table student(
id int,
name string,
age int,
address string
)
partitioned by (dt string,type string)                  # 制定分区
row format delimited fields terminated by '\t'          # 指定字段分隔符为tab
collection items terminated by ','                      # 指定数组中字段分隔符为逗号
map keys terminated by ':'                              # 指定字典中KV分隔符为冒号
lines terminated by '\n'                                # 指定行分隔符为回车换行
stored as textfile                                      # 指定存储类型为文件
;

# 将数据加载到表中（此时时静态分区）
load data local inpath '/root/student.txt' into  test.student partition(class='一班');

```

分桶：

- 分桶表是在表或者分区表的基础上，进一步对表进行组织，Hive使用 对分桶所用的值；
- 进行hash，并用hash结果除以桶的个数做取余运算的方式来分桶，保证了每个桶中都有数据，但每个桶中的数据条数不一定相等。

注意：

创建分区表时：

- 可以使用distribute by(sno) sort by(sno asc) 或是使用clustered by(字段)
- 当排序和分桶的字段相同的时候使用cluster by， 就等同于分桶+排序(sort)

```
# 创建分桶表
create table student(
```

公众号：旧时光大数据

```
id int,
name string,
age int,
address string
)
clustered by(id) sorted by(age) into 4 buckets
row format delimited fields terminated by '\t'
stored as textfile;

# 开启分桶
set hive.enforce.bucketing = true;
# 插入数据
insert overwrite table studentselect id ,name ,age ,address from employees;
# 也可以用另一种插入方式
load data local inpath '/root/student.txt' into test.student;

```

2、数据类型上

分桶随机分割数据库，分区是非随机分割数据库。因为分桶是按照列的哈希函数进行分割的，相对比较平均；而分区是按照列的值来进行分割的，容易造成数据倾斜。

分桶是对应不同的文件（细粒度），分区是对应不同的文件夹（粗粒度）。桶是更为细粒度的数据范围划分，分桶的比分区获得更高的查询处理效率，使取样更高效。

注意：普通表（外部表、内部表）、分区表这三个都是对应HDFS上的目录，桶表对应是目录里的文件。

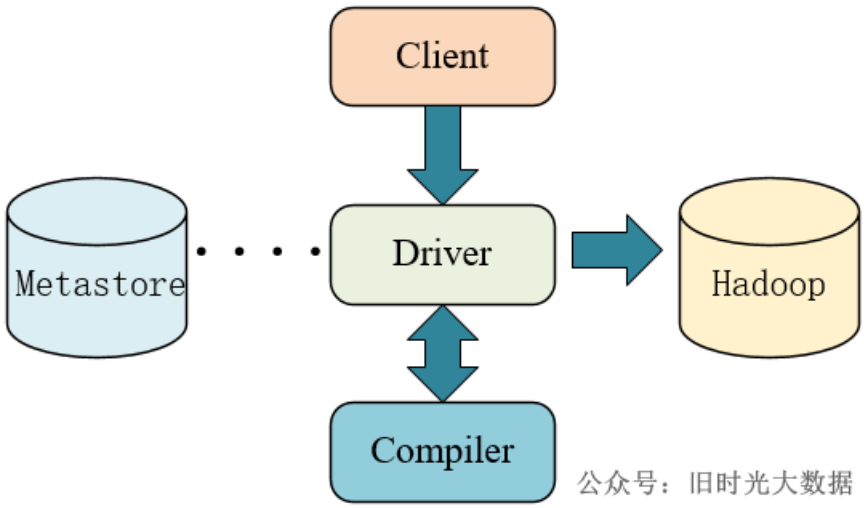
Hive的执行流程

可回答：Hive是怎么执行的

问过的一些公司：美团，多益

参考答案：

1、执行流程概述



查看hive语句的执行流程：explain select ….from t_table …;

- 查看hive语句的执行流程：explain select ….from t_table …;
- 操作符是hive的最小执行单元
- Hive通过execmapper和execreducer执行MapReduce程序，执行模式有本地模式和分布式模式
- 每个操作符代表一个 HDFS 操作或者 MapReduce 作业

公众号：旧时光大数据

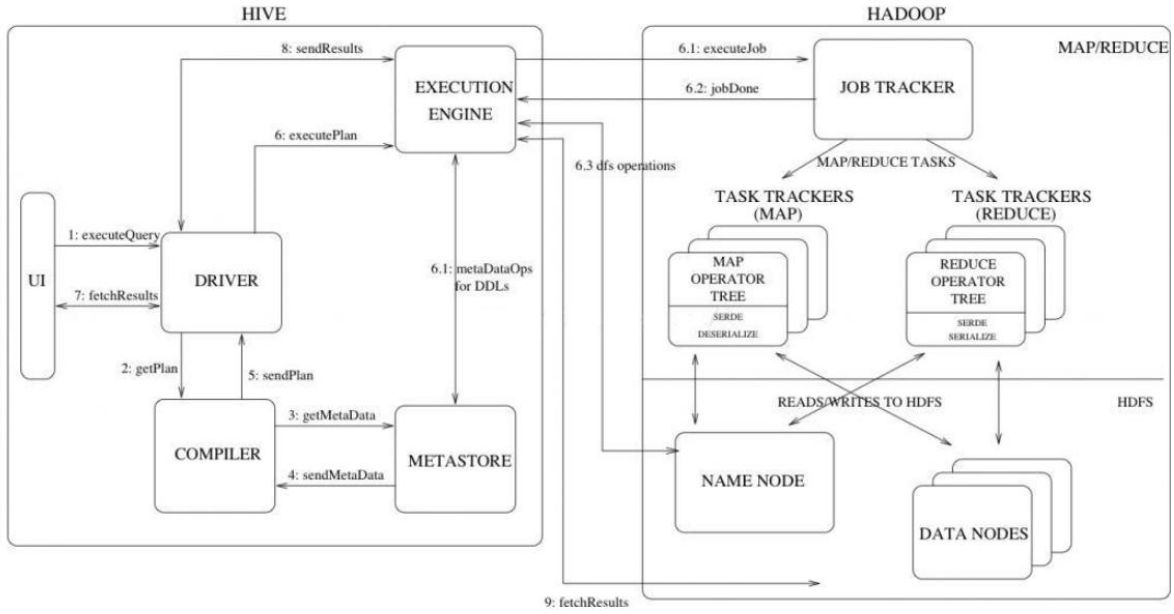
hive操作符：

Hive 操作符列表	
TablescanOpertor	扫描 hive 表数据
ReduceSinkOperator	创建将发送到 reduce 端的<key，value>
JoinOperator	Join 两份数据
Selectoperator	选择输出列
FileSinkoperator	建立结果数据，输出到文件
FilterOperator	过滤输入数据
Groupbyoperator	Groupby 语句
MapJoinoperator	/*+mapjoin(t)*/
LimitOperator	Limit 语句
UnionOperator	Union 语句

Hive编译器的工作职责：

- Parser：将Hql语句转换成抽像的语法书（Abstract Syntax Tree）
- Semantic Analyzer：将抽象语法树转换成查询块
- Logic Plan Generator：将查询树，转换成逻辑查询计划
- Logic Optimizer：重写逻辑查询计划，优化逻辑执行计划
- Physical Plan Gernerator：将逻辑执行计划转化为物理计划
- Physical Optimizer：选择最佳的join策略，优化物理执行计划

2、Hive工作原理



流程步骤：

1）用户提交查询等任务给Driver。

2）编译器获得该用户的任务Plan。

3）编译器Compiler根据用户任务去MetaStore中获取需要的Hive的元数据信息。

4）编译器Compiler得到元数据信息，对任务进行编译，先将HiveQL转换为抽象语法树，然后将抽象语法树转换成查询块，将查询块转化为逻辑的查询计划，重写逻辑查询计划，将逻辑计划转化为物理的计划（MapReduce），最后选择最佳的策略。

5）将最终的计划提交给Driver。

公众号：旧时光大数据

6）Driver将计划Plan转交给ExecutionEngine去执行，获取元数据信息，提交给JobTracker或者SourceManager执行该任务，任务会直接读取HDFS中文件进行相应的操作。

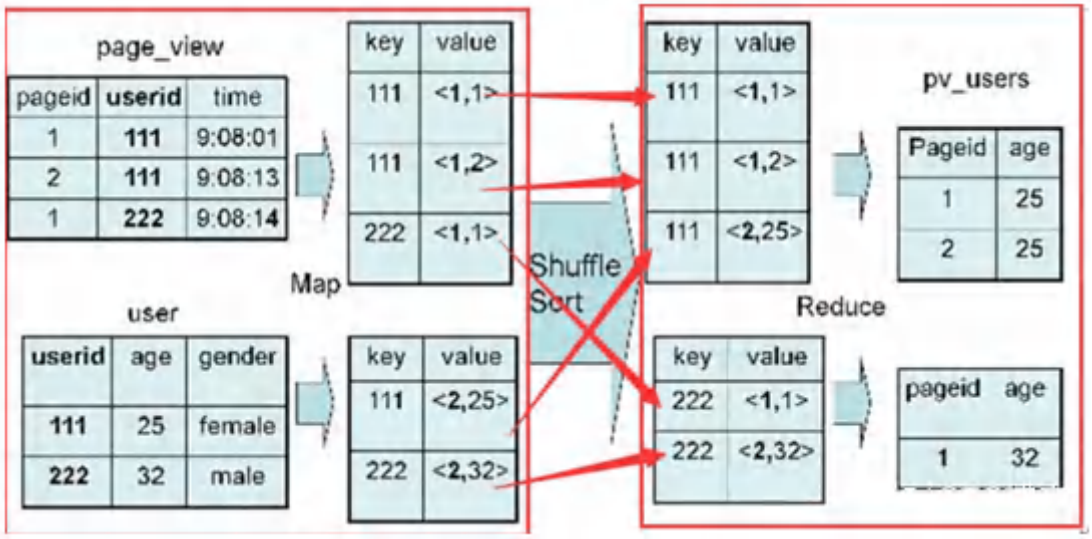
7）获取执行的结果。

8）取得并返回执行结果。

3、hive的具体执行过程分析

1）Join（reduce join）

例：SELECT pv.pageid, u.age FROM page_view pv JOIN user u ON pv.userid = u.userid;



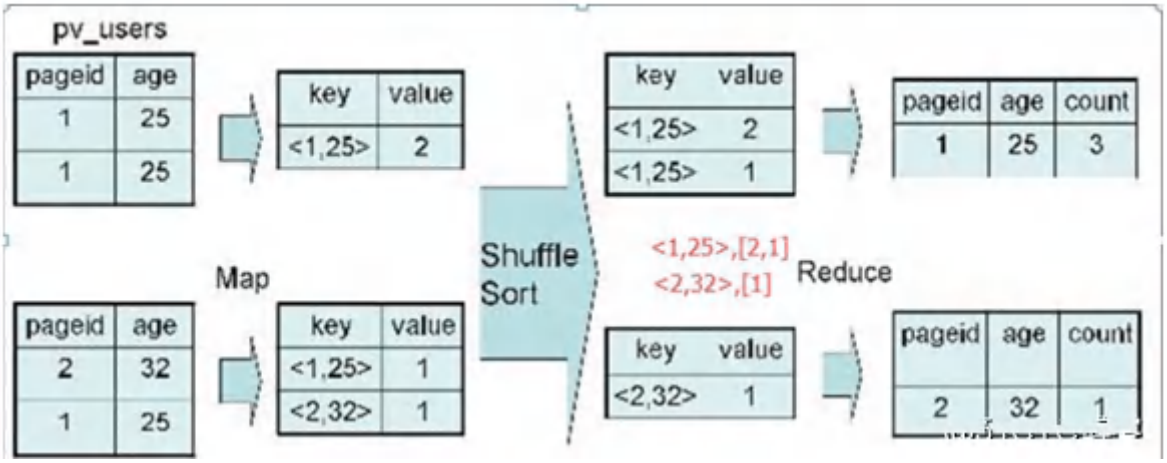
map 端：以 JOIN ON 条件中的列作为 Key，以page_view表中的需要字段，表标识作为value，最终通过key进行排序，也就是join字段进行排序。

shuffle端：根据 Key 的值进行 Hash，并将 Key/Value 对按照 Hash 值推 至不同对 Reduce 中

reduce 端：根据key进行分组，根据不同的表的标识，拿出不同的数据，进行拼接。

2）group by

例：SELECT pageid, age, count(1) FROM pv_users GROUP BY pageid, age;



map 端：

key：以pageid, age作为key,并且在map输出端有combiner。

value ： 1次

reduce 端：对value进行求和

3）distinct

公众号：旧时光大数据

例：select distinct age from log;

map端：

key：age

value：null

reduce端：

一组只要一个输出context.write(key,null)。

4）distinct+count

例：select count(distinct userid) from weibo_temp;

即使设置了reduce个数为3个，最终也只会执行一个，因为，count()是全局，只能开启一个reducetask。

map端：

key：userid

value： null

reduce端：

一组只要一个，定义一个全局变量用于计数，在cleanup（Context context）中输出context.write(key,count)，当然distinct+count是一个容易产生数据倾斜的做法，应该尽量避免，如果无法避免，那么就使用这种方法：select count(1) from (select distinct userid from weibo_temp); 这样可以并行多个reduce task任务，从而解决单节点的压力过大。

Hive SQL转化为MR的过程?

可回答：1）Hive SQL的解析过程；2）Hive SQL的底层实现

问过的一些公司：阿里，头条，小米，滴滴，美团x2，网易，快手58，好未来

参考答案：

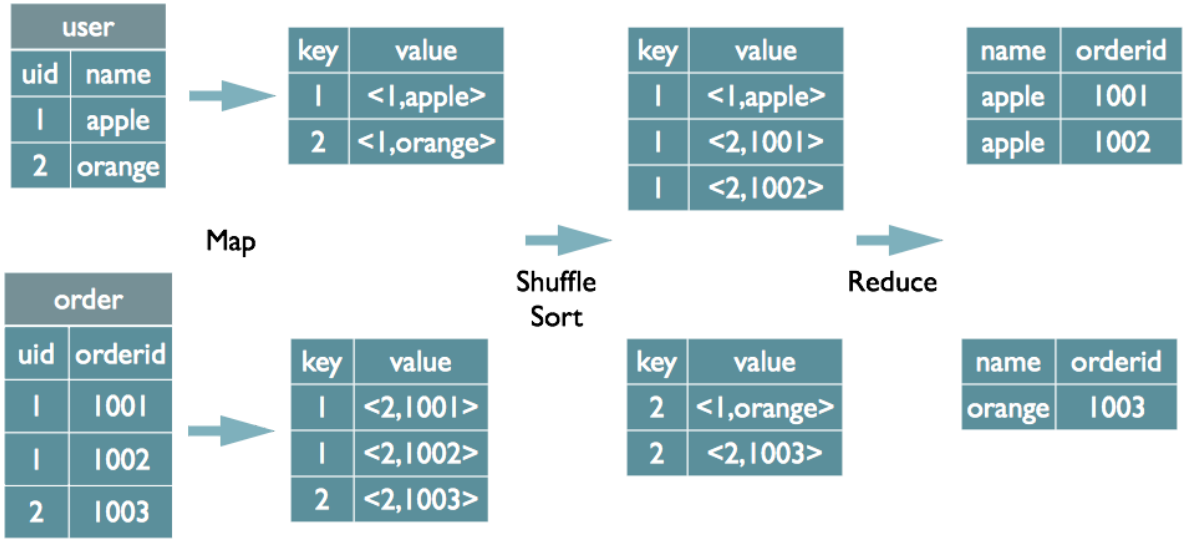
我们先来看下**MapReduce框架实现SQL基本操作的原理**：

1、Join的实现原理

```
select u.name, o.orderid from order o join user u on o.uid = u.uid;

```

在map的输出value中为不同表的数据打上tag标记，在reduce阶段根据tag判断数据来源。MapReduce的过程如下（这里只是说明最基本的Join的实现，还有其他的实现方式）

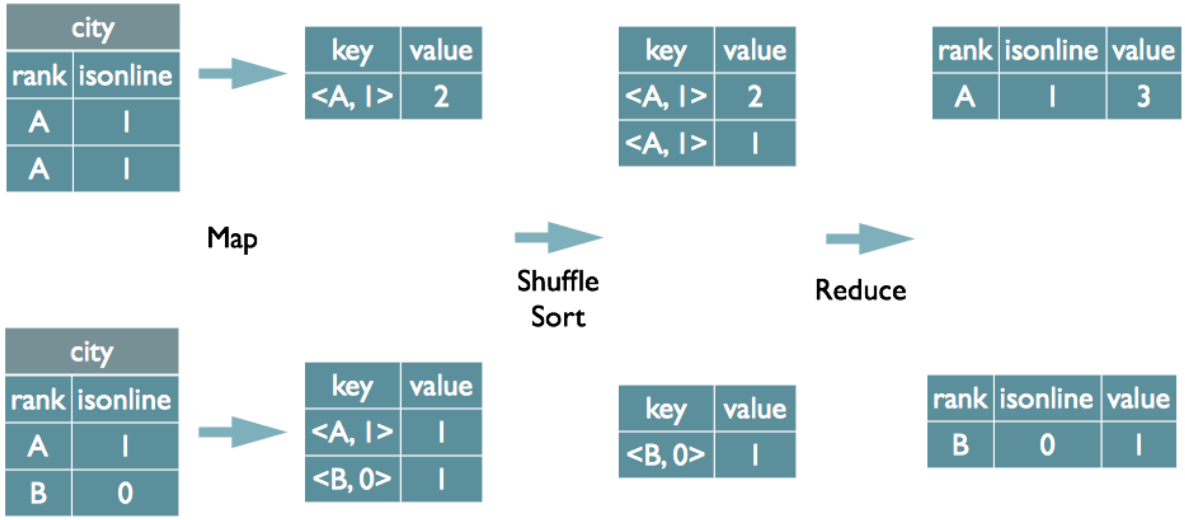
公众号：旧时光大数据



2、Group By的实现原理

```
1   select rank, isonline, count(*) from city group by rank, isonline;
2
```

将GroupBy的字段组合为map的输出key值，利用MapReduce的排序，在reduce阶段保存LastKey区分不同的key。MapReduce的过程如下（当然这里只是说明Reduce端的非Hash聚合过程）

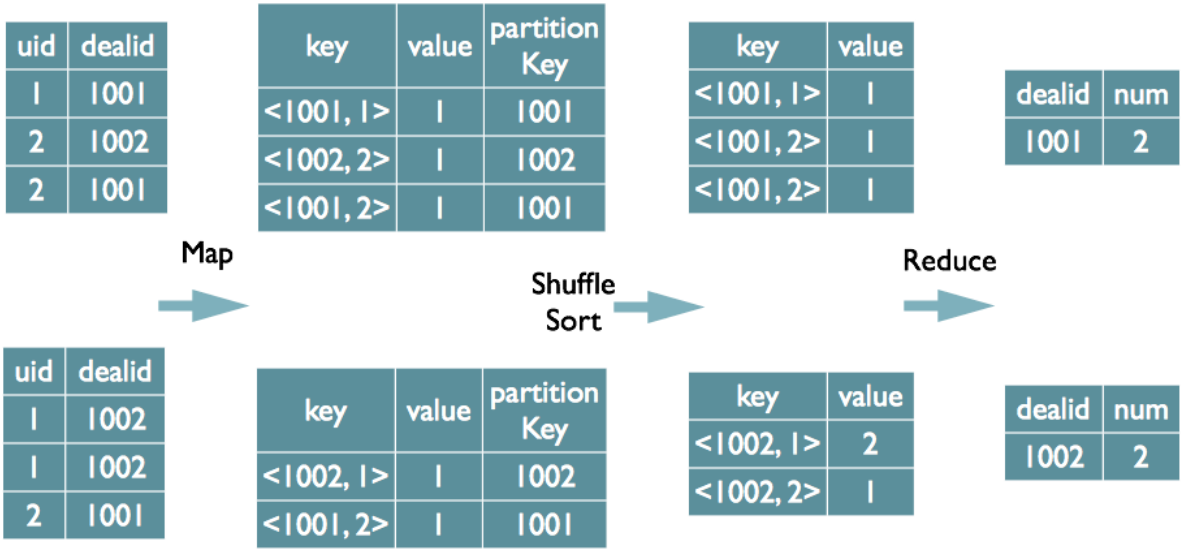


3、Distinct的实现原理

```
1   select dealid, count(distinct uid) num from order group by dealid;
2
```

当只有一个distinct字段时，如果不考虑Map阶段的Hash GroupBy，只需要将GroupBy字段和Distinct字段组合为map输出key，利用mapreduce的排序，同时将GroupBy字段作 为reduce的key，在reduce阶段保存LastKey即可完成去重.

公众号：旧时光大数据

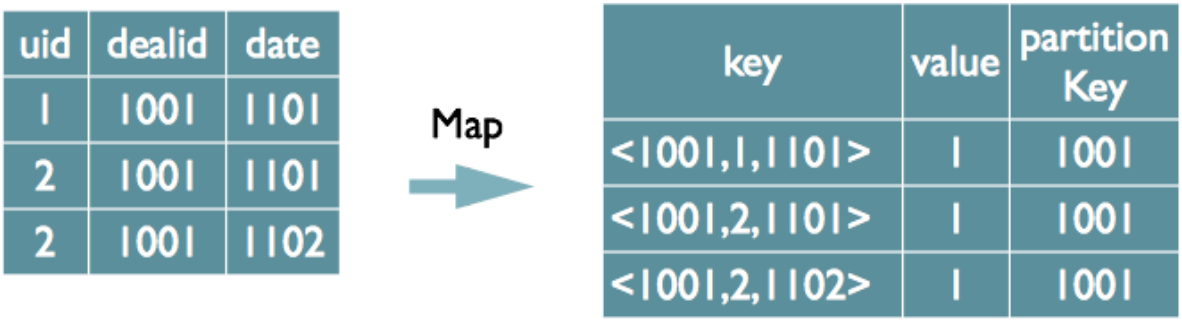


如果有多个distinct字段呢，如下面的SQL

```
select dealid, count(distinct uid), count(distinct date) from order group by dealid;
```

实现方式有两种：

1）如果仍然按照上面一个distinct字段的方法，即下图这种实现方式，无法跟据uid和date分别排序，也就无法通过LastKey去重，仍然需要在reduce阶段在内存中通过Hash去重



2）第二种实现方式，可以对所有的distinct字段编号，每行数据生成n行数据，那么相同字段就会分别排序，这时只需要在reduce阶段记录LastKey即可去重。

这种实现方式很好的利用了MapReduce的排序，节省了reduce阶段去重的内存消耗，但是缺点是增加了shuffle的数据量。

需要注意的是，在生成reduce value时，除第一个distinct字段所在行需要保留value值，其余distinct数据行value字段均可为空。

uid	dealid	date
1	1001	1101
2	1001	1101
2	1001	1102

key	value	partition Key
<1001,0,1>	1	1001
<1001,1,1101>	1	1001
<1001,0,2>	1	1001
<1001,1,1101>	1	1001
<1001,0,2>	1	1001
<1001,1,1102>	1	1001

SQL转化为MapReduce的过程

通过前面的部分了解了MapReduce实现SQL基本操作之后，我们来看看**Hive是如何将SQL转化为MapReduce任务**的，整个编译过程分为六个阶段：

- Antlr定义SQL的语法规则，完成SQL词法，语法解析，将SQL转化为抽象语法树AST Tree
- 遍历AST Tree，抽象出查询的基本组成单元QueryBlock
- 遍历QueryBlock，翻译为执行操作树OperatorTree
- 逻辑层优化器进行OperatorTree变换，合并不必要的ReduceSinkOperator，减少shuffle数据量
- 遍历OperatorTree，翻译为MapReduce任务
- 物理层优化器进行MapReduce任务的变换，生成最终的执行计划

详细部分篇幅过大，可在网上查看

Hive SQL优化处理

问过的一些公司：携程，快手(2021.09)

参考答案：

优化的根本思想

- 尽早尽量过滤数据，减少每个阶段的数据量
- 减少job数
- 解决数据倾斜问题

1、Map Join

如果不指定 MapJoin 或者不符合 MapJoin 的条件，那么 Hive 解析器会将 Join 操作转换 成 Common Join，即：在 Reduce 阶段完成 join。容易发生数据倾斜。可以用MapJoin把小 表全部加载到内存在 map 端进行 join，避免 reducer 处理。

2、行列过滤

列处理：在 SELECT 中，只拿需要的列，如果有，尽量使用分区过滤，少用 SELECT *。

行处理：在分区剪裁中，当使用外关联时，如果将副表的过滤条件写在 Where 后面，那 么就会先全表关联，之后再过滤。

3、多采用分桶技术

4、结合实际环境合理设置 Map 数

公众号：旧时光大数据

通常情况下，**作业会通过 input的目录产生一个或者多个map任务。** 主要的决定因素有：input的文件总个数，input的文件大小，集群设置的文件块大小；

map数不是越多越好；如果一个任务有很多小文件（远远小于块大小 128m），则每个小文件 也会被当做一个块，用一个 map 任务来完成，而一个 map 任务启动和初始化的时间远远大 于逻辑处理的时间，就会造成很大的资源浪费。而且，同时可执行的 map 数是受限的。解决这个问题需要减少map数。

并不是每个map处理接近128m的文件块就是完美的；比如有一个 127m 的文件，正常会用一个 map 去完成，但这个文件只 有一个或者两个小字段，却有几千万的记录，如果 map 处理的逻辑比较复杂，用一个 map 任务去做，肯定也比较耗时。解决这个问题需要增加map数。

5、 合并大量小文件

在Map执行前合并小文件，可以减少Map数：CombineHiveInputFormat 具有对小文件进行合并的功能（系统默认的格式）。HiveInputFormat 没有对小文件合并功能。

6、设置合理的Reduce数

Reduce 个数也并不是越多越好

- 过多的启动和初始化 Reduce 也会消耗时间和资源；
- 有多少个 Reduce，就会有多少个输出文件，如果生成了很多个小文件，那么如果这些小文件作为下一个任务的输入，则也会出现小文件过多的问题；
- 在设置Reduce个数的时候也需要考虑这两个原则：处理大数据量利用合适的 Reduce 数；使单个 Reduce 任务处理数据量大小要合适。

7、输出合并小文件常用参数

```
SET hive.merge.mapfiles = true; -- 默认 true，在 map-only 任务结束时合并小文件
SET hive.merge.mapredfiles = true; -- 默认 false，在 map-reduce 任务结束时合并小文件
SET hive.merge.size.per.task = 268435456; -- 默认 256M
SET hive.merge.smallfiles.avgsize = 16777216; -- 当输出文件的平均大小小于 16m 该值时，启动一个独立的 map-reduce 任务进行文件 merge

```

8、开启 map 端 combiner（不影响最终业务逻辑）

```
# 开启命令
set hive.map.aggr=true;

```

9、中间结果压缩

设置 map 端输出、中间结果压缩。（不完全是解决数据倾斜的问题，但是减少了 IO 读写和网络传输，能提高很多效率）

Hive的存储引擎和计算引擎

可回答：1）Hive的存储和计算；2）Hive的底层引擎模式?

问过的一些公司：字节，快手，顺丰，恒生(2021.09)

参考答案：

1、计算引擎

目前**Hive支持MapReduce、Tez和Spark三种计算引擎。**

在低版本（Hive 1.1之前）中，Hive支持MapReduce、Tez两种计算引擎。

公众号：旧时光大数据

在高版本（Hive 1.1之后）中，Hive支持MapReduce、Tez和Spark三种就算引擎。

配置命令如下：

```
# 配置mapreduce计算引擎
set hive.execution.engine=mr;
# 配置spark计算引擎
set hive.execution.engine=spark;
# 配置tez 计算引擎
set hive.execution.engine=tez;

```

注意版本问题

MapReduce计算引擎：

Map在读取数据时，先将数据拆分成若干数据，并读取到Map方法中被处理。数据在输出的时候，被分成若干分区并写入内存缓存（buffer）中，内存缓存被数据填充到一定程度会溢出到磁盘并排序，当Map执行完后会将一个机器上输出的临时文件进行归并存入到HDFS中。

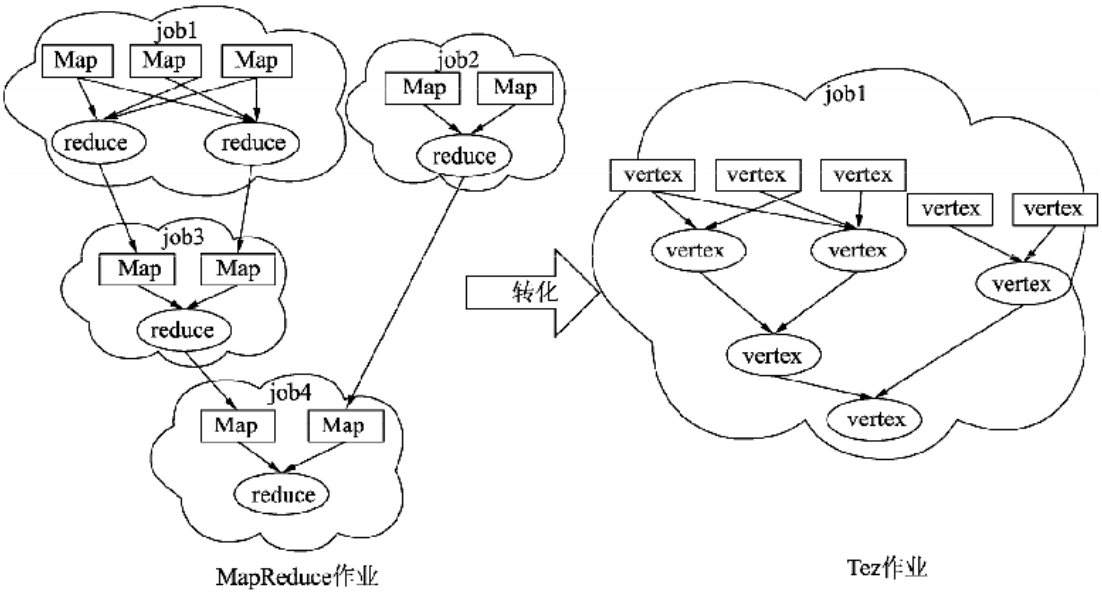
当Reduce启动时，会启动一个线程去读取Map输出的数据，并写入到启动Reduce机器的内存中，在数据溢出到磁盘时会对数据进行再次排序。当读取数据完成后会将临时文件进行合并，作为Reduce函数的数据源。

Tez计算引擎：

Apache Tez是进行大规模数据处理且支持DAG作业的计算框架，它直接源于MapReduce框架，除了能够支持MapReduce特性，还支持新的作业形式，并允许不同类型的作业能够在一个集群中运行。

Tez将原有的Map和Reduce两个操作简化为一个概念——Vertex，并将原有的计算处理节点拆分成多个组成部分：Vertex Input、Vertex Output、Sorting、Shuffling和Merging。计算节点之间的数据通信被统称为Edge，这些分解后的元操作可以任意灵活组合，产生新的操作，这些操作经过一些控制程序组装后，可形成一个大的DAG作业。

通过允许Apache Hive运行复杂的DAG任务，Tez可以用来处理数据，之前需要多个MR jobs，现在一个Tez任务中。



Spark计算引擎：

Apache Spark是专为大规模数据处理而设计的快速、通用支持DAG（有向无环图）作业的计算引擎，类似于Hadoop MapReduce的通用并行框架，可用来构建大型的、低延迟的数据分析应用程序。

Tez和MapReduce作业的比较

Tez绕过了MapReduce很多不必要的中间的数据存储和读取的过程，直接在一个作业中表达了MapReduce需要多个作业共同协作才能完成的事情。

Tez和MapReduce一样都运行使用YARN作为资源调度和管理。但与MapReduce on YARN不同，Tez on YARN并不是将作业提交到ResourceManager，而是提交到AMPoolServer的服务上，AMPoolServer存放着若干已经预先启动ApplicationMaster的服务。

当用户提交一个作业上来后，AMPoolServer从中选择一个ApplicationMaster用于管理用户提交上来的作业，这样既可以节省ResourceManager创建ApplicationMaster的时间，而又能够重用每个ApplicationMaster的资源，节省了资源释放和创建时间。

Tez相比于MapReduce有几点重大改进：

当查询需要有多个reduce逻辑时，Hive的MapReduce引擎会将计划分解，每个Redcue提交一个MR作业。这个链中的所有MR作业都需要逐个调度，每个作业都必须从HDFS中重新读取上一个作业的输出并重新洗牌。而在Tez中，几个reduce接收器可以直接连接，数据可以流水线传输，而不需要临时HDFS文件，这种模式称为MRR（Map-reduce-reduce）。

Tez还允许一次发送整个查询计划，实现应用程序动态规划，从而使框架能够更智能地分配资源，并通过各个阶段流水线传输数据。对于更复杂的查询来说，这是一个巨大的改进，因为它消除了IO/sync障碍和各个阶段之间的调度开销。

在MapReduce计算引擎中，无论数据大小，在洗牌阶段都以相同的方式执行，将数据序列化到磁盘，再由下游的程序去拉取，并反序列化。Tez可以允许小数据集完全在内存中处理，而MapReduce中没有这样的优化。仓库查询经常需要在处理完大量的数据后对小型数据集进行排序或聚合，Tez的优化也能极大地提升效率。

2、存储引擎

Hive的文件存储格式（存储引擎）有四种：`TEXTFILE`、`SEQUENCEFILE`、`ORC`、`PARQUET`，前面两种是行式存储，后面两种是列式存储。如果为textfile的文件格式，直接load，不需要走MapReduce；如果是其他的类型就需要走MapReduce了，因为其他的类型都涉及到了文件的压缩，这需要借助MapReduce的压缩方式来实现。

`TEXTFILE`：按行存储，不支持块压缩，默认格式，数据不做压缩，磁盘开销大，加载数据的速度最高

`RCFILE`：

- 数据按行分块，每块按列存储，结合了行存储和列存储的优点
- RCFile 保证同一行的数据位于同一节点，因此元组重构的开销很低
- RCFile 能够利用列维度的数据压缩，并且能跳过不必要的列读取

`ORCFile`：

- 存储方式：数据按行分块，每块按照列存储
- 压缩快，快速列存取
- 效率比rcfile高，**是rcfile的改良版本**，使用了索引
- 使用ORC文件格式可以提高hive读、写和处理数据的能力

`PARQUET`：按列存储，相对于ORC，Parquet压缩比较低，查询效率较低

`SequenceFile`：

- Hadoop API提供的一种二进制文件，以<key,value>的形式序列化到文件中
- 存储方式：行存储

总结

压缩比：ORC > Parquet > textFile（textfile没有进行压缩）

公众号：旧时光大数据

查询速度：三者几乎一致

Hive的文件存储格式都有哪些

可回答：Hive四种文件类型和压缩情况

问过的一些公司：Shopee(2021.07)x2，贝壳找房(2021.11)

参考答案：

Hive的文件存储格式有四种：`TEXTFILE`、`SEQUENCEFILE`、`ORC`、`PARQUET`，前面两种是行式存储，后面两种是列式存储。如果为textfile的文件格式，直接load，不需要走MapReduce；如果是其他的类型就需要走MapReduce了，因为其他的类型都涉及到了文件的压缩，这需要借助MapReduce的压缩方式来实现。

`TEXTFILE`：按行存储，不支持块压缩，默认格式，数据不做压缩，磁盘开销大，加载数据的速度最高

`RCFILE`：

- 数据按行分块，每块按列存储，结合了行存储和列存储的优点
- RCFile 保证同一行的数据位于同一节点，因此元组重构的开销很低
- RCFile 能够利用列维度的数据压缩，并且能跳过不必要的列读取

`ORCFile`：

- 存储方式：数据按行分块，每块按照列存储
- 压缩快，快速列存取
- 效率比rcfile高，**是rcfile的改良版本**，使用了索引
- 使用ORC文件格式可以提高hive读、写和处理数据的能力

`PARQUET`：按列存储，相对于ORC，Parquet压缩比较低，查询效率较低

`SequenceFile`：

- Hadoop API提供的一种二进制文件，以<key,value>的形式序列化到文件中
- 存储方式：行存储

总结

压缩比：ORC > Parquet > textFile（textfile没有进行压缩）

查询速度：三者几乎一致

Hive中如何调整Mapper和Reducer的数目

问过的一些公司：小米，美团，快手

参考答案：

调整Mapper数量

之前MapReduce部分也有提到这个类似的问题

- map数量=split数量
- split数量=文件大小/split size
- splitszie=Math.max(minSize, Math.min(maxSize, blockSize))

默认情况下，split size=blockSize，也就是128M

公众号：旧时光大数据

控制Mapper数量

```
set mapred.max.split.size=256000000;    -- 决定每个map处理的最大的文档大小，单位为B
set mapred.min.split.size.per.node=1;    -- 节点中可以处理的最小的文档大小
set mapred.min.split.size.per.rack=1;    -- 机架中可以处理的最小的文档大小

```

其设置原则就是

```
要增加map的个数，调整maxSize<blockSize;
要减小map的个数，调整minSize>blockSize。

```

调整Reducer数量

修改下面两个参数就行

```
方法1
set mapred.reduce.tasks=10;  -- 设置reduce的数量
方法2
set hive.exec.reducers.bytes.per.reducer=1073741824 -- 每个reduce处理的数据量,默认1GB

```

介绍下知道的Hive窗口函数，举一些例子

问过的一些公司：小米，池鸢，快手，网易

参考答案：

Hive中的窗口函数和sql中的窗口函数相类似，都是用来做一些数据分析类的工作，一般用于OLAP分析（在线分析处理）。

在sql中有一类函数叫做聚合函数，例如sum()、avg()、max()等等，这类函数可以将多行数据按照规则聚集为一行，一般来讲聚集后的行数是要少于聚集前的行数的。但是有时我们想要既显示聚集前的数据，又要显示聚集后的数据,这时我们便引入了窗口函数。

注意：在SQL处理中，窗口函数都是最后一步执行，而且仅位于Order by字句之前。

准备一张order表，字段分别为name，orderdate，cost。数据内容如下：

```
jack,2015-01-01,10
tony,2015-01-02,15
jack,2015-02-03,23
tony,2015-01-04,29
jack,2015-01-05,46
jack,2015-04-06,42
tony,2015-01-07,50
jack,2015-01-08,55
mart,2015-04-08,62
mart,2015-04-09,68
neil,2015-05-10,12
mart,2015-04-11,75
neil,2015-06-12,80
mart,2015-04-13,94

```

在hive中建立一张表t_window，将数据插入进去。

公众号：旧时光大数据

聚合函数+over

假如说我们想要查询在2015年4月份购买过的顾客及总人数，我们便可以使用窗口函数去去实现

```
select name,count(*) over ()
from t_window
where substring(orderdate,1,7) = '2015-04'

```

结果如下：

```
name    count_window_0
mart    5
mart    5
mart    5
mart    5
jack    5

```

在2015年4月一共有5次购买记录，mart购买了4次，jack购买了1次。事实上，大多数情况下，我们是只看去重后的结果的。针对于这种情况，我们有两种实现方式：

第一种：distinct

```
select distinct name,count(*) over ()
from t_window
where substring(orderdate,1,7) = '2015-04'

```

结果如下：

```
name    count_window_0
mart    2
jack    2

```

第二种：group by

```
select name,count(*) over ()
from t_window
where substring(orderdate,1,7) = '2015-04'
group by name

```

结果如下：

```
name    count_window_0
mart    2
jack    2

```

partition by子句

Partition By子句也可以称为查询分区子句，非常类似于Group By，都是将数据按照边界值分组，而Over之前的函数在每一个分组之内进行，如果超出了分组，则函数会重新计算。

我们想要去看顾客的购买明细及月购买总额：

公众号：旧时光大数据

```
select name,orderdate,cost,sum(cost) over(partition by month(orderdate))
from t_window

```

结果如下：

```
name    orderdate   cost    sum_window_0
jack    2015-01-01  10          205
jack    2015-01-08  55          205
tony    2015-01-07  50          205
jack    2015-01-05  46          205
tony    2015-01-04  29          205
tony    2015-01-02  15          205
jack    2015-02-03  23          23
mart    2015-04-13  94          341
jack    2015-04-06  42          341
mart    2015-04-11  75          341
mart    2015-04-09  68          341
mart    2015-04-08  62          341
neil    2015-05-10  12          12
neil    2015-06-12  80          80

```

数据已经按照月进行汇总了

order by子句

假如我们想要将cost按照月进行累加.这时我们引入order by子句

order by子句会让输入的数据强制排序（文章前面提到过，窗口函数是SQL语句最后执行的函数，因此可以把SQL结果集想象成输入数据）。Order By子句对于诸如Row_Number()，Lead()，LAG()等函数是必须的，因为如果数据无序，这些函数的结果就没有任何意义。因此如果有了Order By子句，则Count()，Min()等计算出来的结果就没有任何意义。

在上面的sql中加入order by

```
select name,orderdate,cost,sum(cost) over(partition by month(orderdate) order by
orderdate )
from t_window

```

结果如下：（order by默认情况下聚合从起始行到当前行的数据）

```
name    orderdate   cost    sum_window_0
jack    2015-01-01  10          10
tony    2015-01-02  15          25 //10+15
tony    2015-01-04  29          54 //10+15+29
jack    2015-01-05  46          100 //10+15+29+46
tony    2015-01-07  50          150
jack    2015-01-08  55          205
jack    2015-02-03  23          23
jack    2015-04-06  42          42
mart    2015-04-08  62          104
mart    2015-04-09  68          172
mart    2015-04-11  75          247
mart    2015-04-13  94          341
neil    2015-05-10  12          12
neil    2015-06-12  80          80

```

公众号：旧时光大数据

window子句

首先要理解两个概念：

- 如果只使用partition by子句，未指定order by的话，我们的聚合是分组内的聚合。
- 使用了order by子句，未使用window子句的情况下，默认从起点到当前行。

当同一个select查询中存在多个窗口函数时，他们相互之间是没有影响的，每个窗口函数应用自己的规则。

window子句：

- PRECEDING：往前
- FOLLOWING：往后
- CURRENT ROW：当前行
- UNBOUNDED：起点，UNBOUNDED PRECEDING 表示从前面的起点，UNBOUNDED FOLLOWING：表示到后面的终点

按照name进行分区，按照购物时间进行排序，做cost的累加。

```
select name,orderdate,cost,
sum(cost) over() as sample1,--所有行相加
sum(cost) over(partition by name) as sample2,--按name分组，组内数据相加
sum(cost) over(partition by name order by orderdate) as sample3,--按name分组，组内数据累加
sum(cost) over(partition by name order by orderdate rows between UNBOUNDED PRECEDING and current row )  as sample4 ,--和sample3一样,由起点到当前行的聚合
sum(cost) over(partition by name order by orderdate rows between 1 PRECEDING and current row) as sample5, --当前行和前面一行做聚合
sum(cost) over(partition by name order by orderdate rows between 1 PRECEDING AND 1 FOLLOWING  ) as sample6,--当前行和前边一行及后面一行
sum(cost) over(partition by name order by orderdate rows between current row and UNBOUNDED FOLLOWING ) as sample7 --当前行及后面所有行
from t_window;

```

结果如下：

```
name    orderdate   cost    sample1 sample2 sample3 sample4 sample5 sample6 sample7
jack    2015-01-01  10      661     176     10      10      10      56      176
jack    2015-01-05  46      661     176     56      56      56      111     166
jack    2015-01-08  55      661     176     111     111     101     124     120
jack    2015-02-03  23      661     176     134     134     78      120     65
jack    2015-04-06  42      661     176     176     176     65      65      42
mart    2015-04-08  62      661     299     62      62      62      130     299
mart    2015-04-09  68      661     299     130     130     130     205     237
mart    2015-04-11  75      661     299     205     205     143     237     169
mart    2015-04-13  94      661     299     299     299     169     169     94
neil    2015-05-10  12      661     92      12      12      12      92      92
neil    2015-06-12  80      661     92      92      92      92      92      80
tony    2015-01-02  15      661     94      15      15      15      44      94
tony    2015-01-04  29      661     94      44      44      44      94      79
tony    2015-01-07  50      661     94      94      94      79      79      50

```

公众号：旧时光大数据

窗口函数中的序列函数

Hive中常用的序列函数有下面几个：

1） **Row_Number，Rank，Dense_Rank**这三个窗口函数的使用场景非常多

row_number()：从1开始，按照顺序，生成分组内记录的序列,row_number()的值不会存在重复,当排序的值相同时,按照表中记录的顺序进行排列;通常用于获取分组内排序第一的记录;获取一个session中的第一条refer等。

rank()：生成数据项在分组中的排名，排名相等会在名次中留下空位。

dense_rank()：生成数据项在分组中的排名，排名相等会在名次中不会留下空位。

注意： rank和dense_rank的区别在于排名相等时会不会留下空位

举例如下：

```
SELECT
cookieid,
createtime,
pv,
RANK() OVER(PARTITION BY cookieid ORDER BY pv desc) AS rn1,
DENSE_RANK() OVER(PARTITION BY cookieid ORDER BY pv desc) AS rn2,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY pv DESC) AS rn3
FROM lxw1234
WHERE cookieid = 'cookie1';

cookieid day           pv       rn1     rn2     rn3

cookie1 2015-04-12      7       1       1       1
cookie1 2015-04-11      5       2       2       2
cookie1 2015-04-15      4       3       3       3
cookie1 2015-04-16      4       3       3       4
cookie1 2015-04-13      3       5       4       5
cookie1 2015-04-14      2       6       5       6
cookie1 2015-04-10      1       7       6       7

```

```
rn1：15号和16号并列第3，13号排第5
rn2：15号和16号并列第3，13号排第4
rn3：如果相等，则按记录值排序，生成唯一的次序，如果所有记录值都相等，或许会随机排吧。

```

2） **LAG和LEAD函数**

这两个函数为常用的窗口函数,可以返回上下数据行的数据。

以这里的订单表为例，假如我们想要查看顾客上次的购买时间可以这样去查询。

```
select name,orderdate,cost,
lag(orderdate,1,'1900-01-01') over(partition by name order by orderdate ) as time1,
lag(orderdate,2) over (partition by name order by orderdate) as time2
from t_window;

```

查询后的数据为：

```
name    orderdate    cost    time1    time2
```

公众号：旧时光大数据

```
jack    2015-01-01  10  1900-01-01  NULL
jack    2015-01-05  46  2015-01-01  NULL
jack    2015-01-08  55  2015-01-05  2015-01-01
jack    2015-02-03  23  2015-01-08  2015-01-05
jack    2015-04-06  42  2015-02-03  2015-01-08
mart    2015-04-08  62  1900-01-01  NULL
mart    2015-04-09  68  2015-04-08  NULL
mart    2015-04-11  75  2015-04-09  2015-04-08
mart    2015-04-13  94  2015-04-11  2015-04-09
neil    2015-05-10  12  1900-01-01  NULL
neil    2015-06-12  80  2015-05-10  NULL
tony    2015-01-02  15  1900-01-01  NULL
tony    2015-01-04  29  2015-01-02  NULL
tony    2015-01-07  50  2015-01-04  2015-01-02
```

time1取的为按照name进行分组，分组内升序排列，取上一行数据的值，见下图。

time2取的为按照name进行分组，分组内升序排列，取上面2行的数据的值，注意当lag函数未设置行数值时，默认为1行。设定取不到时的默认值时，取null值。

lead函数与lag函数方向相反，取向下的数据。

```
name    orderdate   cost    time1   time2
jack    2015-01-01  10  1900-01-01  NULL
jack    2015-01-05  46  2015-01-01  NULL
jack    2015-01-08  55  2015-01-05  2015-01-01
jack    2015-02-03  23  2015-01-08  2015-01-05
jack    2015-04-06  42  2015-02-03  2015-01-08
mart    2015-04-08  62  1900-01-01  NULL
mart    2015-04-09  68  2015-04-08  NULL
mart    2015-04-11  75  2015-04-09  2015-04-08
mart    2015-04-13  94  2015-04-11  2015-04-09
neil    2015-05-10  12  1900-01-01  NULL
neil    2015-06-12  80  2015-05-10  NULL
tony    2015-01-02  15  1900-01-01  NULL
tony    2015-01-04  29  2015-01-02  NULL
tony    2015-01-07  50  2015-01-04  2015-01-02
```

time1取的为按照name进行分组，分组内升序排列，取上一行数据的值。

time2取的为按照name进行分组，分组内升序排列，取上面2行的数据的值，注意当lag函数为设置行数值时，默认为1行。未设定取不到时的默认值时，去null值。

3） **first_value和last_value**

first_value取分组内排序后，截止到当前行，第一个值

last_value取分组内排序后，截止到当前行，最后一个值

```
select name,orderdate,cost,
first_value(orderdate) over(partition by name order by orderdate) as time1,
last_value(orderdate) over(partition by name order by orderdate) as time2
from t_window
```

查询结果如下：

```
name    orderdate   cost    time1   time2
jack    2015-01-01  10  2015-01-01  2015-01-01
jack    2015-01-05  46  2015-01-01  2015-01-05
jack    2015-01-08  55  2015-01-01  2015-01-08
jack    2015-02-03  23  2015-01-01  2015-02-03
```

```
jack    2015-04-06  42  2015-01-01  2015-04-06
mart    2015-04-08  62  2015-04-08  2015-04-08
mart    2015-04-09  68  2015-04-08  2015-04-09
mart    2015-04-11  75  2015-04-08  2015-04-11
mart    2015-04-13  94  2015-04-08  2015-04-13
neil    2015-05-10  12  2015-05-10  2015-05-10
neil    2015-06-12  80  2015-05-10  2015-06-12
tony    2015-01-02  15  2015-01-02  2015-01-02
tony    2015-01-04  29  2015-01-02  2015-01-04
tony    2015-01-07  50  2015-01-02  2015-01-07
```

扩展部分：

row_number的用途非常广泛，排序最好用它，它会为查询出来的每一行记录生成一个序号，依次排序且不会重复，注意使用row_number函数时必须要用over子句选择对某一列进行排序才能生成序号。

rank函数用于返回结果集的分区内每行的排名，行的排名是相关行之前的排名数加一。简单来说rank函数就是对查询出来的记录进行排名，与row_number函数不同的是，rank函数考虑到了over子句中排序字段值相同的情况，如果使用rank函数来生成序号，over子句中排序字段值相同的序号是一样的，后面字段值不相同的序号将跳过相同的排名号排下一个，也就是相关行之前的排名数加一，可以理解为根据当前的记录数生成序号，后面的记录依此类推。

dense_rank函数的功能与rank函数类似，dense_rank函数在生成序号时是连续的，而rank函数生成的序号有可能不连续。dense_rank函数出现相同排名时，将不跳过相同排名号，rank值紧接上一次的rank值。在各个分组内，rank()是跳跃排序，有两个第一名时接下来就是第四名，dense_rank()是连续排序，有两个第一名时仍然跟着第二名。

关于Parttion by：

Parttion by关键字是Oracle中分析性函数的一部分，用于给结果集进行分区。它和聚合函数Group by不同的地方在于它只是将原始数据进行名次排列，能够返回一个分组中的多条记录（记录数不变），而Group by是对原始数据进行聚合统计，一般只有一条反映统计值的结果（每组返回一条）。

在使用排名函数的时候需要注意以下三点：

- 排名函数必须有 OVER 子句。
- 排名函数必须有包含 ORDER BY 的 OVER 子句。
- 分组内从1开始排序。

Hive的count的用法

问过的一些公司：小米

参考答案：

`count(*)`：所有行进行统计，包括NULL行

`count(1)`：所有行进行统计，包括NULL行

`count(column)`：对column中非Null进行统计

注意：count(*)执行时间比count(1)和count(column)都长，count(1)和count(column)执行时间差不多。

Hive的union和union all的区别

公众号：旧时光大数据

问过的一些公司：小米

参考答案：

`Union`：对两个结果集进行并集操作，不包括重复行，同时进行默认规则的排序；

`Union All`：对两个结果集进行并集操作，包括重复行，不进行排序。

Hive的join操作原理，left join、right join、inner join、outer join的异同?

可回答：Hive的join的几种操作

问过的一些公司：小米，腾讯，大华，米哈游，有赞

参考答案：

hive执行引擎会将HQL“翻译”成为map-reduce任务，如果多张表使用同一列做join则将被翻译成一个reduce，否则将被翻译成多个map-reduce任务。

比如，hive执行引擎会将HQL“翻译”成为map-reduce任务，如果多张表使用同一列做join则将被翻译成一个reduce，否则将被翻译成多个map-reduce任务。

```
SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key1)
-- 将被翻译成1个map-reduce任务

```

```
SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key2)
-- 将被翻译成2个map-reduce任务

```

这个很好理解，一般来说（map side join除外），map过程负责分发数据，具体的join操作在reduce完成，因此，如果多表基于不同的列做join，则无法在一轮map-reduce任务中将所有相关数据shuffle到统一个reducer，对于多表join，hive会将前面的表缓存在reducer的内存中，然后后面的表会流式的进入reducer和reducer内存中其它的表做join。

为了防止数据量过大导致oom，将数据量最大的表放到最后，或者通过“STREAMTABLE”显示指定reducer流式读入的表。

1、Join的操作原理

Hive中的Join可分为`Map Join`（Map阶段完成join）和`Common Join`（Reduce阶段完成join）。

`Common Join`

```
select u.name, o.orderid from order o join user u on o.uid = u.uid;

```

Map阶段

读取源表的数据，Map输出时候以Join on条件中的列为key，如果Join有多个关联键，则以这些关联键的组合作为key;

Map输出的value为join之后所关心的(select或者where中需要用到的)列；同时在value中还会包含表的Tag信息，用于标明此value对应哪个表；

按照key进行排序。

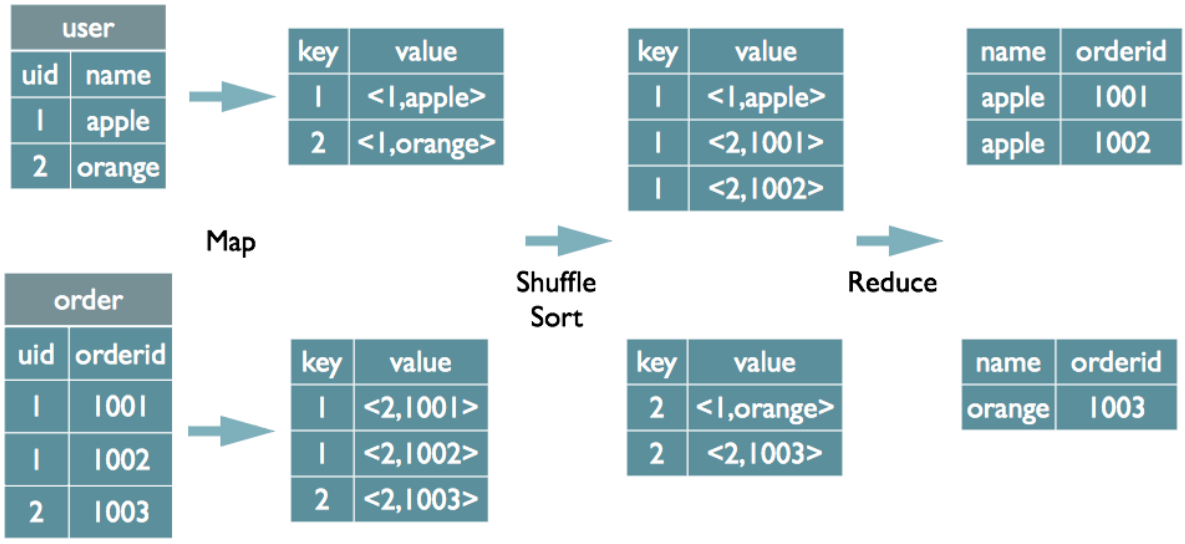
公众号：旧时光大数据

Shuffle阶段

根据key的值进行hash,并将key/value按照hash值推送至不同的reduce中，这样确保两个表中相同的key位于同一个reduce中

Reduce阶段

根据key的值完成join操作，期间通过Tag来识别不同表中的数据。

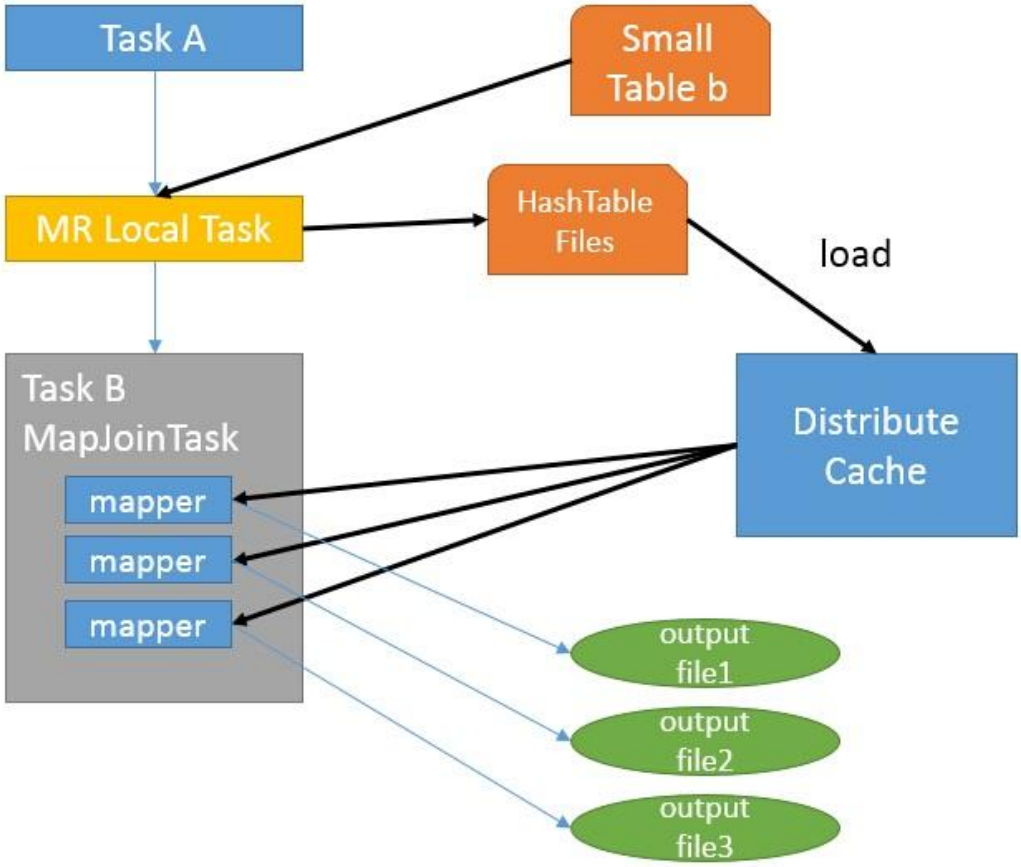


`Map Join`

MapJoin通常用于一个很小的表和一个大表进行join的场景，具体小表有多小，由参数hive.mapjoin.smalltable.filesize来决定，该参数表示小表的总大小，默认值为25000000字节，即25M。

Hive0.7之前，需要使用hint提示 /+ mapjoin(table) /才会执行MapJoin,否则执行Common Join，但在0.7版本之后，默认自动会转换Map Join，由参数hive.auto.convert.join来控制，默认为true。

假设a表为一张大表，b为小表，并且hive.auto.convert.join=true,那么Hive在执行时候会自动转化为MapJoin。



如图中的流程，首先是Task A，它是一个Local Task（在客户端本地执行的Task），负责扫描小表b的数据，将其转换成一个HashTable的数据结构，并写入本地的文件中，之后将该文件加载到DistributeCache中，该HashTable的数据结构可以抽象为：

key	Value
1	26
2	34

```
Execution log at: /tmp/liuxiaowen/liuxiaowen_20150625091414_6e7713f9-bc22-4030-b976-6754aaa61658.log
2015-06-25 09:14:13     Starting to launch local task to process map join;      maximum memory = 954728448
2015-06-25 09:14:14     Dump the side-table into file: file:/tmp/liuxiaowen/hive_2015-06-25_09-14-10_024_1998235
able-Stage-2/MapJoin-mapfile01--.hashtable
2015-06-25 09:14:14     Uploaded 1 File to: file:/tmp/liuxiaowen/hive_2015-06-25_09-14-10_024_199823552092746999
2/MapJoin-mapfile01--.hashtable (435 bytes)
2015-06-25 09:14:14     End of local task; Time Taken: 0.785 sec.
Execution completed successfully
MapredLocal task succeeded
```

图中红框圈出了执行Local Task的信息。

接下来是Task B，该任务是一个没有Reduce的MR，启动MapTasks扫描大表a,在Map阶段，根据a的每一条记录去和DistributeCache中b表对应的HashTable关联，并直接输出结果。

由于MapJoin没有Reduce，所以由Map直接输出结果文件，有多少个Map Task，就有多少个结果文件。

总的来说，因为小表的存在，可以在Map阶段直接完成Join的操作，为了优化小表的查找速度，将其转化为HashTable的结构，并加载进分布式缓存中。

2、inner join、left join、right join、outer join

`inner join`

公众号：旧时光大数据

等值连接，只返回两个表中联结字段相等的行

`left join`

左联接，返回包括左表中的所有记录和右表中联结字段相等的记录

`right join`

右联接，返回包括右表中的所有记录和左表中联结字段相等的记录

inner join等价于join，可以理解为join是inner join的缩写；

left join等价于left outer join；right join等价于right outer join。

Hive如何优化join操作

问过的一些公司：作业帮，池鹭，米哈游

参考答案：

1、在map端产生join

mapJoin的主要意思就是，当链接的两个表是一个比较小的表和一个特别大的表的时候，我们把比较小的table直接放到内存中去，然后再对比较大的表格进行map操作。join就发生在map操作的时候，每当扫描一个大的table中的数据，就要去去查看小表的数据，哪条与之相符，继而进行连接。这里的join并不会涉及reduce操作。map端join的优势就是在于没有shuffle，真好。在实际的应用中设置如下：

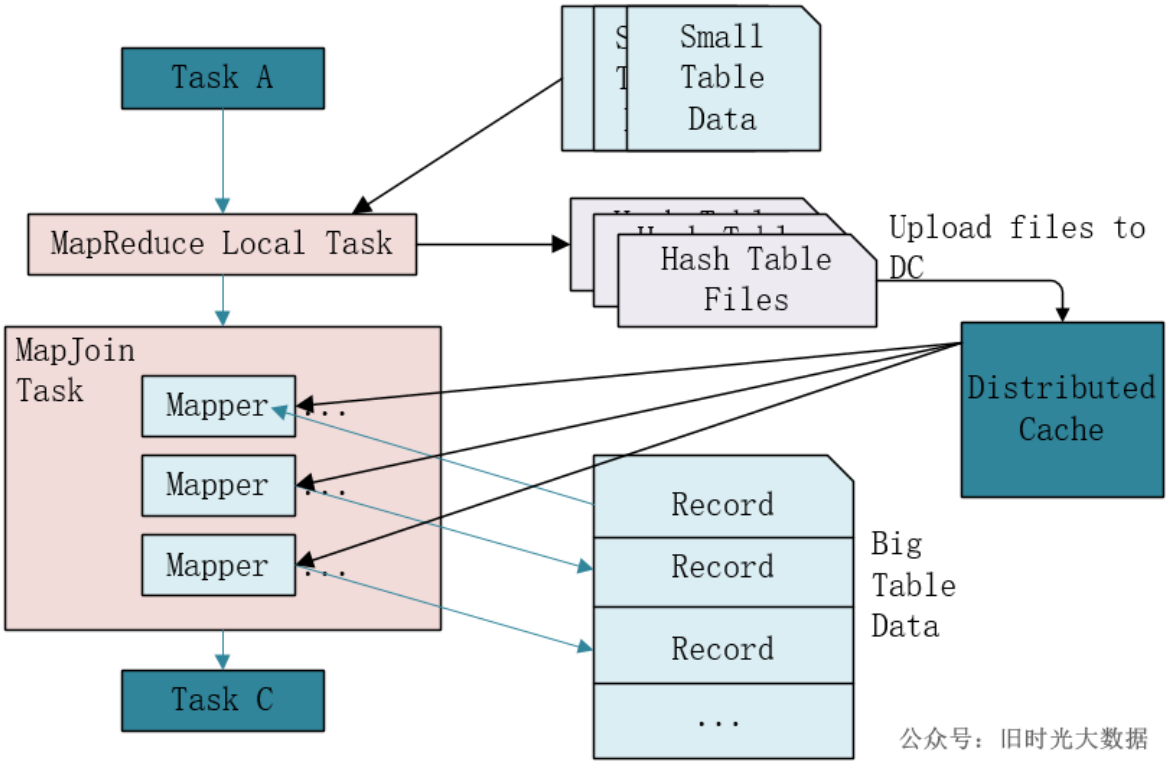
```
1   set hive.auto.convert.join=true;
2
```

这样设置，hive就会自动的识别比较小的表，继而用mapJoin来实现两个表的联合。看看下面的两个表格的连接。这里的dept相对来讲是比较小的。我们看看会发生什么，如图所示：

```
hive.auto.convert.join=true
hive (emp_dept)> select count(*) from emp join dept on(emp.deptno==dept.deptno);
Total jobs = 1
Execution log at: /tmp/root/root_20160130072222_cdc75d2c-9d29-45aa-9263-a43f95dcdff1.log
2016-01-30 07:23:00     Starting to launch local task to process map join;      maximum memory = 477102
080
2016-01-30 07:23:02     Dump the side-table into file: file:/tmp/root/hive_2016-01-30_07-22-54_043_4136
720104773279416-1/-local-10004/HashTable-Stage-2/MapJoin-mapfile01--.hashtable
2016-01-30 07:23:02     Uploaded 1 File to: file:/tmp/root/hive_2016-01-30_07-22-54_043_413672010477327
9416-1/-local-10004/HashTable-Stage-2/MapJoin-mapfile01--.hashtable (332 bytes)
2016-01-30 07:23:02     End of local task; Time Taken: 1.776 sec.
Execution completed successfully
```

注意看，这里的第一句话就是运行本地的map join任务，继而转存文件到XXX.hashtable下面，在给这个文件里面上传一个文件进行map join，之后才运行了MR代码去运行计数任务。说白了，在本质上mapjoin根本就没有运行MR进程，仅仅是在内存就进行了两个表的联合。具体运行如下图：

公众号：旧时光大数据



2、common join

common join也叫做shuffle join，reduce join操作。这种情况下生再两个table的大小相当，但是又不是很大的情况下使用的。具体流程就是在map端进行数据的切分，一个block对应一个map操作，然后进行shuffle操作，把对应的block shuffle到reduce端去，再逐个进行联合，这里优势会涉及到数据的倾斜，大幅度的影响性能有可能会运行speculation，这块儿在后续的数据倾斜会讲到。因为平常我们用到的数据量小，所以这里就不具体演示了。

3、SMB Join

smb是sort merge bucket操作，首先进行排序，继而合并，然后放到所对应的bucket中去，bucket是hive中和分区表类似的技术，就是按照key进行hash，相同的hash值都放到相同的buck中去。在进行两个表联合的时候。我们首先进行分桶，在join会大幅度的对性能进行优化。也就是说，在进行联合的时候，是table1中的一小部分和table1中的一小部分进行联合，table联合都是等值连接，相同的key都放到了同一个bucket中去了，那么在联合的时候就会大幅度的减小无关项的扫描。

Hive的map join

问过的一些公司：58，米哈游

参考答案：

1、什么是MapJoin?

MapJoin顾名思义，就是在Map阶段进行表之间的连接。而不需要进入到Reduce阶段才进行连接。这样就节省了在Shuffle阶段时要进行的大量数据传输。从而起到了优化作业的作用。

2、MapJoin的原理

通常情况下，要连接的各个表里面的数据会分布在不同的Map中进行处理。即同一个Key对应的Value可能存在不同的Map中。这样就必须等到Reduce中去连接。

要使MapJoin能够顺利进行，那就必须满足这样的条件：除了一份表的数据分布在不同的Map中外，其他连接的表的数据必须在每个Map中有完整的拷贝。

公众号：旧时光大数据

3、MapJoin适用的场景

通过上面分析可以有发现，并不是所有的场景都适合用MapJoin。它通常会用在如下的一些情景：在二个要连接的表中，有一个很大，有一个很小，这个小表可以存放在内存中而不影响性能。

这样我们就把小表文件复制到每一个Map任务的本地，再让Map把文件读到内存中待用。

4、MapJoin的实现方法：

在Map-Reduce的驱动程序中使用静态方法 `DistributedCache.addCacheFile()` 增加要拷贝的小表文件。JobTracker在作业启动之前会获取这个URI列表，并将相应的文件拷贝到各个TaskTracker的本地磁盘上。

在Map类的setup方法中使用DistributedCache.getLocalCacheFiles()方法获取文件目录，并使用标准的文件读写API读取相应的文件。

5、Hive内置提供的优化机制之一就包括MapJoin

在Hive v0.7之前，需要使用hint提示 /*+ mapjoin(table) */才会执行MapJoin。Hive v0.7之后的版本已经不需要给出MapJoin的指示就进行优化。它是通过如下配置参数来控制的：

```
hive> set hive.auto.convert.join=true;

```

Hive还提供另外一个参数--表文件的大小作为开启和关闭MapJoin的阈值。

```
hive.mapjoin.smalltable.filesize=25000000   -- 即25M

```

Hive语句的运行机制，例如包含where、having、group by、order by，整个的执行过程?

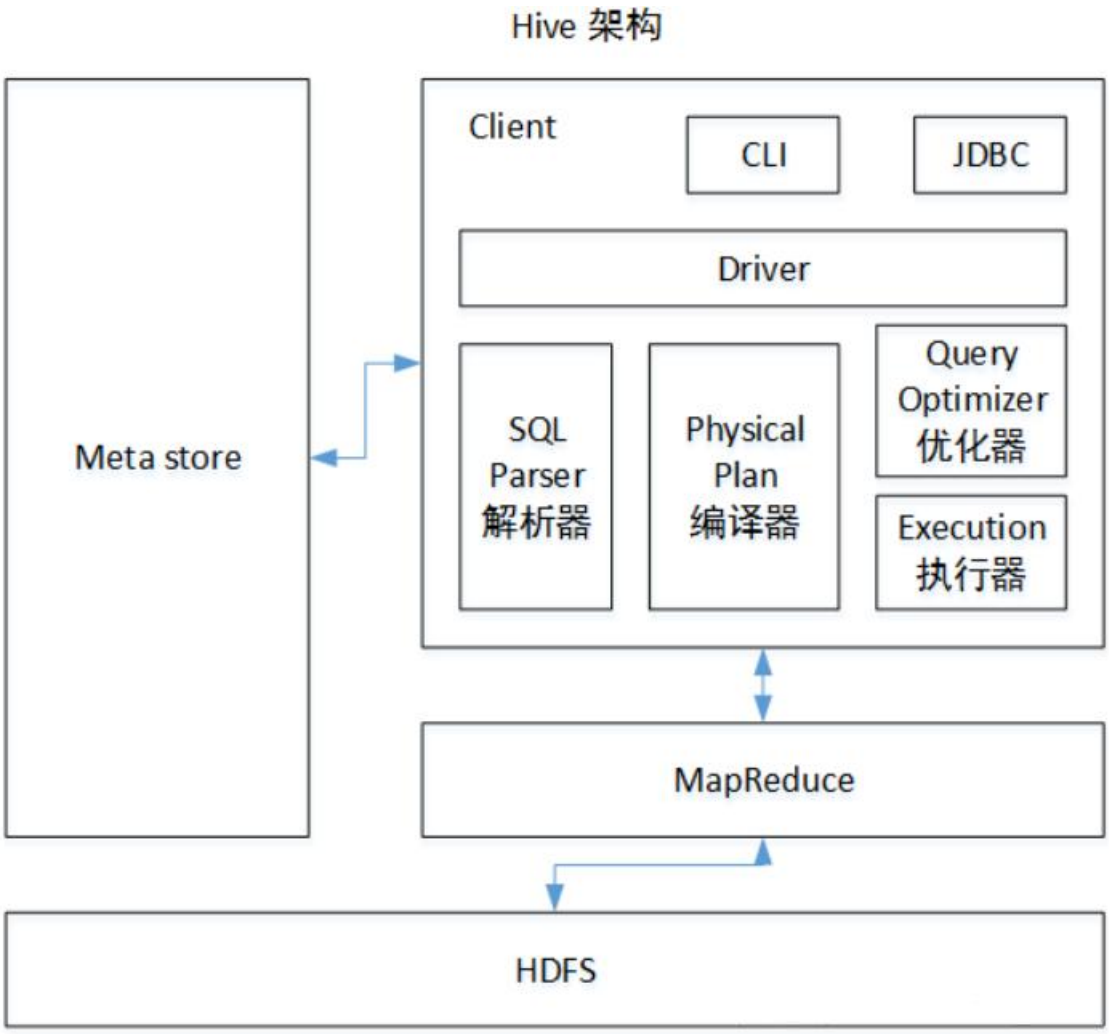
问过的一些公司：小米

参考答案：

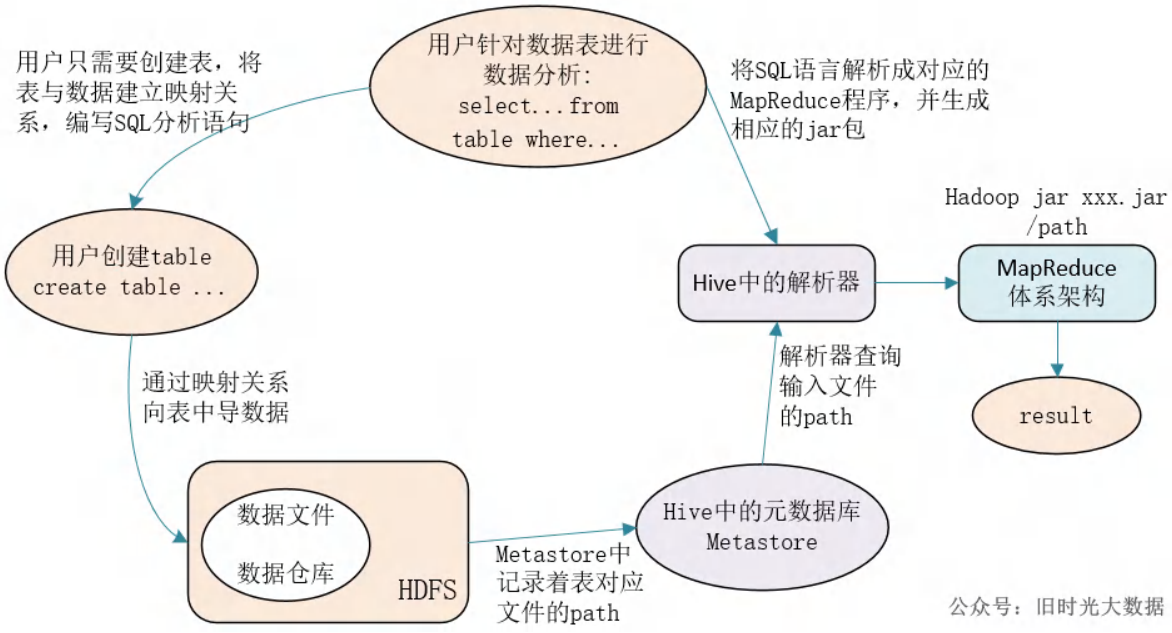
1、Hive语句运行机制

1）架构图

公众号：旧时光大数据



2）运行机制图



流程大致步骤为：

1）用户提交查询等任务给Driver。

2）编译器获得该用户的任务Plan。

3）编译器Compiler根据用户任务去MetaStore中获取需要的Hive的元数据信息。

公众号：旧时光大数据

4）编译器Compiler得到元数据信息，对任务进行编译，先将HiveQL转换为抽象语法树，然后将抽象语法树转换成查询块，将查询块转化为逻辑的查询计划，重写逻辑查询计划，将逻辑计划转化为物理的计划（MapReduce），最后选择最佳的策略。

5）将最终的计划提交给Driver。

6）Driver将计划Plan转交给ExecutionEngine去执行，获取元数据信息，提交给JobTracker或者SourceManager执行该任务，任务会直接读取HDFS中文件进行相应的操作。

7）获取执行的结果。

8）取得并返回执行结果。

创建表时：

解析用户提交的Hive语句->对其进行解析->分解为表、字段、分区等Hive对象。根据解析到的信息构建对应的表、字段、分区等对象，从SEQUENCE_TABLE中获取构建对象的最新的ID，与构建对象信息（名称、类型等等）一同通过DAO方法写入元数据库的表中，成功后将SEQUENCE_TABLE中对应的最新ID+5.实际上常见的RDBMS都是通过这种方法进行组织的，其系统表中和Hive元数据一样显示了这些ID信息。通过这些元数据可以很容易的读取到数据。

2、Hive语句执行顺序

1）Hive语句书写顺序

```
(1)select
(2)from
(3)join on
(4) where
(5)group by
(6)having
(7)distribute by/cluster by
(8) sort by
(9) order by
(10) limit
(11) union(去重不排序)/union all（不去重不排序）

```

2）Hive语句执行顺序

```
(1)from
(2)on
(3)join
(4)where
(5)group by
(6)having
(7)select
(8)distinct
(9)distribute by /cluster by
(10)sort by
(11) order by
(12) limit
(13) union /union all

```

Hive使用的时候会将数据同步到HDFS，小文件问题怎么解决的?

公众号：旧时光大数据

问过的一些公司：360，陌陌(2021.10)

参考答案：

首先，我们要弄明白两个问题：

1）**哪里会产生小文件**

- 源数据本身有很多小文件
- 动态分区会产生大量小文件
- reduce个数越多, 小文件越多
- 按分区插入数据的时候会产生大量的小文件, 文件个数 = maptask个数 * 分区数

2）**小文件太多造成的影响**

- 从Hive的角度看，小文件会开很多map，一个map开一个JVM去执行，所以这些任务的初始化，启动，执行会浪费大量的资源，严重影响性能。
- HDFS存储太多小文件, 会导致namenode元数据特别大, 占用太多内存, 制约了集群的扩展

小文件解决方案：

方法一：通过调整参数进行合并

1）在Map输入的时候, 把小文件合并

```
-- 每个Map最大输入大小，决定合并后的文件数
set mapred.max.split.size=256000000;
-- 一个节点上split的至少的大小 ，决定了多个data node上的文件是否需要合并
set mapred.min.split.size.per.node=100000000;
-- 一个交换机下split的至少的大小，决定了多个交换机上的文件是否需要合并
set mapred.min.split.size.per.rack=100000000;
-- 执行Map前进行小文件合并
set hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFormat;

```

2）在Reduce输出的时候, 把小文件合并

```
-- 在map-only job后合并文件，默认true
set hive.merge.mapfiles = true;
-- 在map-reduce job后合并文件，默认false
set hive.merge.mapredfiles = true;
-- 合并后每个文件的大小，默认256000000
set hive.merge.size.per.task = 256000000;
-- 平均文件大小，是决定是否执行合并操作的阈值，默认16000000
set hive.merge.smallfiles.avgsize = 100000000;

```

方法二：针对按分区插入数据的时候产生大量的小文件的问题，可以使用DISTRIBUTE BY rand() 将数据随机分配给Reduce，这样可以使得每个Reduce处理的数据大体一致。

```
-- 设置每个reducer处理的大小为5个G
set hive.exec.reducers.bytes.per.reducer=5120000000;

-- 使用distribute by rand()将数据随机分配给reduce, 避免出现有的文件特别大, 有的文件特别小
insert overwrite table test partition(dt)
select * from iteblog_tmp
DISTRIBUTE BY rand();

```

方法三：使用Sequencefile作为表存储格式，不要用textfile，在一定程度上可以减少小文件

公众号：旧时光大数据

方法四：使用hadoop的archive归档

```
-- 用来控制归档是否可用
set hive.archive.enabled=true;
-- 通知Hive在创建归档时是否可以设置父目录
set hive.archive.har.parentdir.settable=true;
-- 控制需要归档文件的大小
set har.partfile.size=1099511627776;

-- 使用以下命令进行归档
ALTER TABLE srcpart ARCHIVE PARTITION(ds='2008-04-08', hr='12');

-- 对已归档的分区恢复为原文件
ALTER TABLE srcpart UNARCHIVE PARTITION(ds='2008-04-08', hr='12');

-- 注意，归档的分区不能够INSERT OVERWRITE，必须先unarchive

```

Hadoop自带的三种小文件处理方案

- Hadoop Archive
  - Hadoop Archive或者HAR，是一个高效地将小文件放入HDFS块中的文件存档工具，它能够将多个小文件打包成一个HAR文件，这样在减少namenode内存使用的同时，仍然允许对文件进行透明的访问。
- Sequence file
  - sequence file由一系列的二进制key/value组成，如果为key小文件名，value为文件内容，则可以将大批小文件合并成一个大文件。
- CombineFileInputFormat
  - 它是一种新的inputformat，用于将多个文件合并成一个单独的split，另外，它会考虑数据的存储位置。

Hive Shuffle的具体过程

问过的一些公司：竞技世界

参考答案：

Shuffle其实就是“洗牌”，整个过程也就是清洗重发的过程，map方法后，reduce方法之前。

首先是inputsplit，每个切片对应一个map（可以通过yarn去观察），map端首先对读入数据做按不同数据类型做分区，之后根据不同分区做排序（这里采用的快速排序），map过程中会有内存缓冲区（环形缓冲区），它的作用是在数据写入过程中，将数据存入内存从而达到减少IO开启的资源消耗，提高分区，排序的资源量（hive优化中提到map join小表在前存入内存也源于此处），当缓冲区的写入达到默认的0.8（80mb,可更改设置），将开启溢写将内容写入临时文件同时剩下的写入会继续写入到剩余0.2。

这整个过程中会产生大量临时文件，通过merge最后合并成一个文件，分区且有序（归并排序），到这map端基本结束。（写入内存前可以通过开启 combine，一般公司都会在map开启来达到减小数据量提高效率，实现的效果 : map的输出是 （key,value ），combine 后输出自然减少了）。

reduce端是通过http协议抓取数据（fetch），map跑完reduce开始抓数，这里涉及到数据倾斜的问题（需要注意），reduce对数据同样通过partition，sort (归并排序），整理好的数据最后进入reduce操作。

Hive有哪些保存元数据的方式，都有什么特点?

公众号：旧时光大数据

问过的一些公司：冠群驰骋

参考答案：

内嵌模式：将元数据保存在本地内嵌的derby数据库中，内嵌的derby数据库每次只能访问一个数据文件，也就意味着它不支持多会话连接。

本地模式：将元数据保存在本地独立的数据库中（一般是mysql），这可以支持多会话连接。

远程模式：把元数据保存在远程独立的mysql数据库中，避免每个客户端都去安装mysql数据库。

三种配置方式区别

- 内嵌模式使用的是内嵌的Derby数据库来存储元数据，也不需要额外起Metastore服务。这个是默认的，配置简单，但是一次只能一个客户端连接，适用于用来实验，不适用于生产环境。不常用。
- 本地元存储和远程元存储都采用外部数据库来存储元数据，目前支持的数据库有：MySQL、Postgres、Oracle、MS SQL Server。MySQL较常用。
- 本地元存储和远程元存储的区别是：本地元存储不需要单独起metastore服务，用的是跟hive在同一个进程里的metastore服务。远程元存储需要单独起metastore服务，然后每个客户端都在配置文件里配置连接到该metastore服务。远程元存储的metastore服务和hive运行在不同的进程。不常用。

Hive SQL实现查询用户连续登陆，讲讲思路

问过的一些公司：美团

参考答案：

这里连续活跃登陆的用户指至少**连续2天都活跃登录**的用户

解决类似场景的问题

创建数据

```
CREATE TABLE test5active(
dt string,
user_id string,
age int)
ROW format delimited fields terminated BY ',';

INSERT INTO TABLE test5active VALUES
('2019-02-11','user_1',23),('2019-02-11','user_2',19),
('2019-02-11','user_3',39),('2019-02-11','user_1',23),
('2019-02-11','user_3',39),('2019-02-11','user_1',23),
('2019-02-12','user_2',19),('2019-02-13','user_1',23),
('2019-02-15','user_2',19),('2019-02-16','user_2',19);

```

思路一：

1、因为每天用户登录次数可能不止一次，所以需要先将用户每天的登录日期去重。

2、再用row_number() over(partition by _ order by _)函数将用户id分组，按照登陆时间进行排序。

3、计算登录日期减去第二步骤得到的结果值，用户连续登陆情况下，每次相减的结果都相同。

4、按照id和日期分组并求和，筛选大于等于2的即为连续活跃登陆的用户。

第一步：用户登录日期去重

```
select DISTINCT dt,user_id from test5active;

```

```
2019-02-11      user_1
2019-02-11      user_2
2019-02-11      user_3
2019-02-12      user_2
2019-02-13      user_1
2019-02-15      user_2
2019-02-16      user_2
```

第二步：用row_number() over()函数计数

```
select
t1.user_id,t1.dt,
row_number() over(partition by t1.user_id order by t1.dt) day_rank
from
(
    select DISTINCT dt,user_id from test5active
)t1;

```

```
user_1  2019-02-11      1
user_1  2019-02-13      2
user_2  2019-02-11      1
user_2  2019-02-12      2
user_2  2019-02-15      3
user_2  2019-02-16      4
user_3  2019-02-11      1
```

第三步：日期减去计数值得到结果

```
select
t2.user_id,t2.dt,date_sub(t2.dt,t2.day_rank) as dis
from
(
    select
    t1.user_id,t1.dt,
    row_number() over(partition by t1.user_id order by t1.dt) day_rank
from
(
    select DISTINCT dt,user_id from test5active
)t1)t2;

```

```
user_1  2019-02-11      2019-02-10
user_1  2019-02-13      2019-02-11
user_2  2019-02-11      2019-02-10
user_2  2019-02-12      2019-02-10
user_2  2019-02-15      2019-02-12
user_2  2019-02-16      2019-02-12
user_3  2019-02-11      2019-02-10
```

第四步：根据id和结果分组并计算总和，大于等于2的即为连续登陆的用户，得到 用户id，开始日期，结束日期 ， 连续登录天数

公众号：旧时光大数据

```
select
t3.user_id,min(t3.dt),max(t3.dt),count(1)
from
(
    select
    t2.user_id,t2.dt,date_sub(t2.dt,t2.day_rank) as dis
        from
        (
            select
            t1.user_id,t1.dt,
            row_number() over(partition by t1.user_id order by t1.dt) day_rank
                from
                (
                    select DISTINCT dt,user_id from test5active
                )t1
        )t2
)t3 group by t3.user_id,t3.dis having count(1)>1;

```

结果：

```
用户id        开始日期        结束日期        连续登录天数
user_2      2019-02-11     2019-02-12          2
user_2      2019-02-15     2019-02-16          2

```

第五步：连续登陆的用户

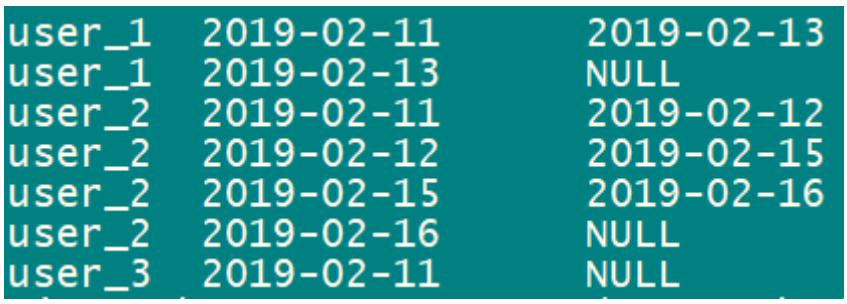
```
select distinct t4.user_id
from
(
    select
    t3.user_id,min(t3.dt),max(t3.dt),count(1)
    from
    (
        select
        t2.user_id,t2.dt,date_sub(t2.dt,t2.day_rank) as dis
        from
        (
            select
            t1.user_id,t1.dt,
            row_number() over(partition by t1.user_id order by t1.dt) day_rank
            from
            (
                select DISTINCT dt,user_id from test5active
            )t1
        )t2
    )t3 group by t3.user_id,t3.dis having count(1)>1
)t4;

```

最后结果：

```
user_2
```

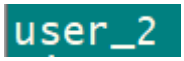
思路二：**使用lag（向后）或者lead（向前）**

```
select
user_id,t1.dt,
lead(t1.dt) over(partition by user_id order by t1.dt) as last_date_id
from
(
    select DISTINCT dt,user_id from test5active
)t1;

```




```
select
distinct t2.user_id
from
(
    select
    user_id,t1.dt,
    lead(t1.dt) over(partition by user_id order by t1.dt) as last_date_id
    from
    (
        select DISTINCT dt,user_id from test5active
    )t1
)t2 where datediff(last_date_id,t2.dt)=1;

```

最后结果：



Hive的开窗函数有哪些

问过的一些公司：北京元安物联社招，京东(2021.09)

参考答案：

分析函数用于计算基于组的某种聚合值，它和聚合函数的不同之处是：**对于每个组返回多行，而聚合函数对于每个组只返回一行。**

开窗函数指定了分析函数工作的**数据窗口**大小，这个数据窗口大小可能会随着行的变化而变化!

基础结构如下：

```
分析函数（如:sum(),max(),row_number()...） + 窗口子句（over函数）

```

1、SUM函数

求和，窗口函数和聚合函数的不同，sum()函数可以根据每一行的窗口返回各自行对应的值，有多少行记录就有多少个sum值，而group by只能计算每一组的sum，每组只有一个值。sum()计算的是分区内排序后一个个叠加的值，**和order by有关。**

公众号：旧时光大数据

2、NTILE函数

NTILE(n)，用于将分组数据按照顺序切分成n片，返回当前切片值

注意：

- 如果切片不均匀，**默认增加第一个切片的分布**
- NTILE不支持ROWS BETWEEN

3、ROW_NUMBER 函数

ROW_NUMBER()：从1开始，按照顺序，生成分组内记录的序列

ROW_NUMBER()：的应用场景非常多，比如获取分组内排序第一的记录、获取一个session中的第一条refer等：

4、RANK 和 DENSE_RANK 函数

RANK()：生成数据项在分组中的排名，**排名相等会在名次中留下空位**
DENSE_RANK()：生成数据项在分组中的排名，**排名相等会在名次中不会留下空位**

5、CUME_DIST 函数

cume_dist：返回小于等于当前值的行数/分组内总行数

6、PERCENT_RANK 函数

percent_rank：分组内当前行的RANK值-1/分组内总行数-1

注意：一般不会用到该函数，可能在一些特殊算法的实现中可以用到吧

7、LAG 和 LEAD 函数

LAG(col,n,DEFAULT)：用于统计窗口内往上第n行值

第一个参数为列名，第二个参数为往上第n行（可选，默认为1），第三个参数为默认值（当往上第n行为NULL时候，取默认值，如不指定，则为NULL）

LEAD 函数则与 LAG 相反：

LEAD(col,n,DEFAULT)：用于统计窗口内往下第n行值

第一个参数为列名，第二个参数为往下第n行（可选，默认为1），第三个参数为默认值（当往下第n行为NULL时候，取默认值，如不指定，则为NULL）

8、FIRST_VALUE 和 LAST_VALUE 函数

FIRST_VALUE：取分组内排序后，截止到当前行，第一个值

LAST_VALUE 函数则相反：

LAST_VALUE：取分组内排序后，截止到当前行，最后一个值

这两个函数还是经常用到的（往往和排序配合使用），比较实用！

Hive存储数据吗

问过的一些公司：vivo(2021.06)

参考答案：

Hive本身不存储数据。

公众号：旧时光大数据

Hive的数据分为**表数据**和**元数据**，**表数据**是Hive中表格(table)具有的数据；而**元数据**是用来存储表的名字，表的列和分区及其属性，表的属性(是否为外部表等)，表的数据所在目录等。

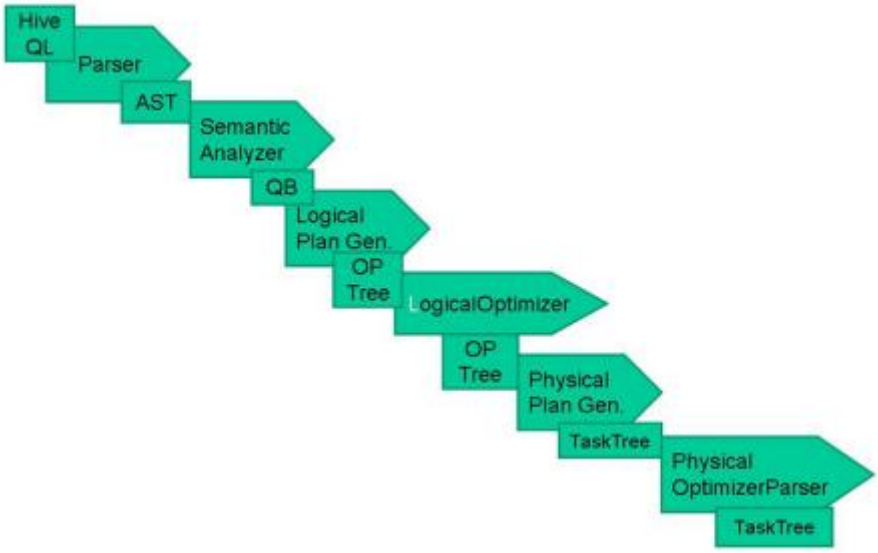
Hive建表后，**表的元数据存储在关系型数据库中（如：mysql）**，**表的数据（内容）存储在hdfs中**，这些数据是以文本的形式存储在hdfs中（关系型数据库是以二进制形式存储的），既然是存储在hdfs上，那么这些数据本身也是有元数据的（在NameNode中），而数据在DataNode中。这里注意两个元数据的不同。

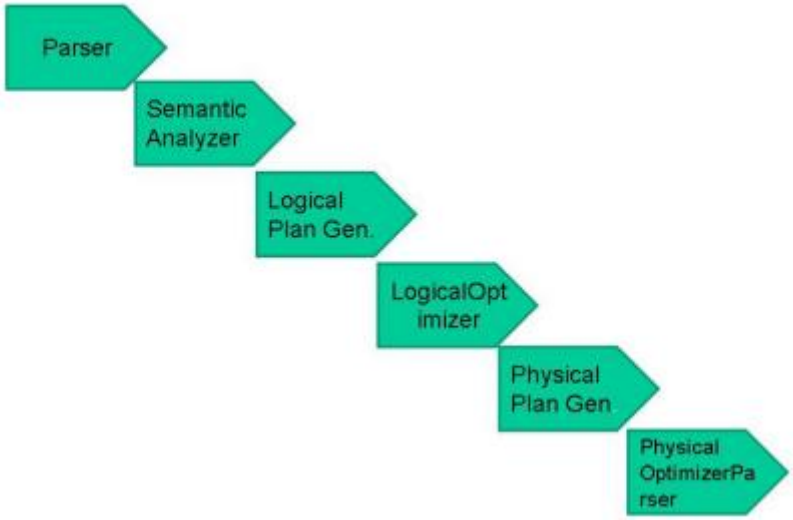
Hive的SQL转换为MapReduce的过程?

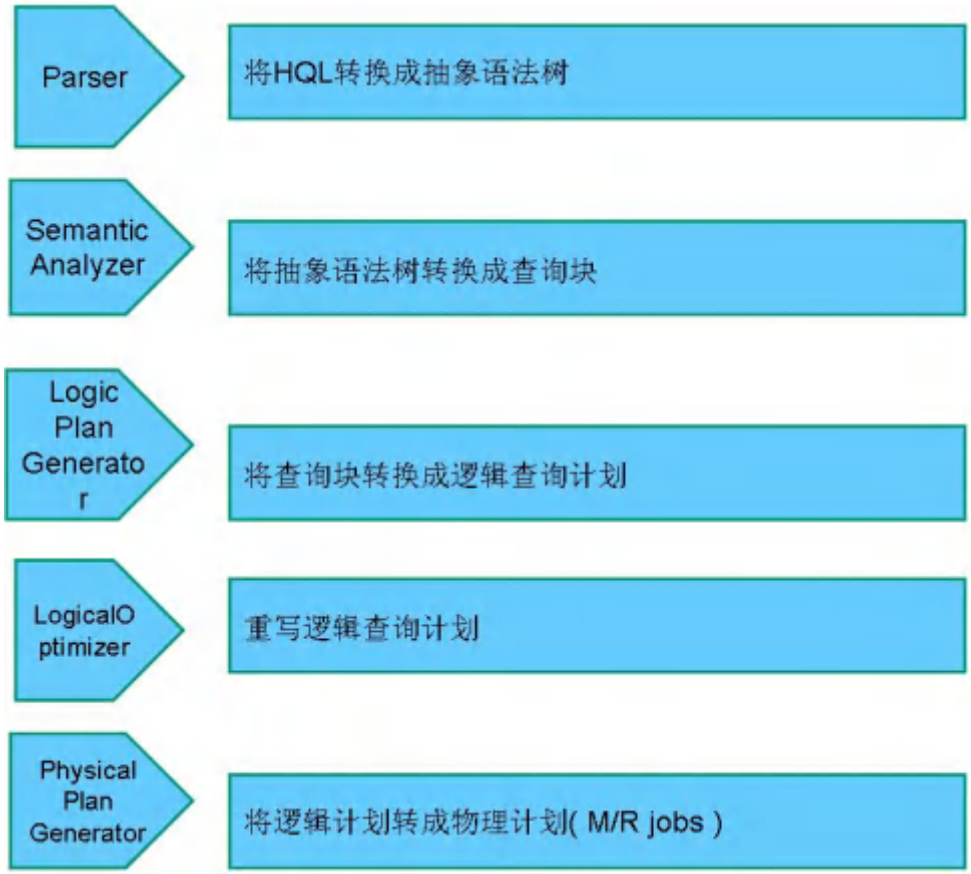
问过的一些公司：远景智能(2021.08)，美团买菜(2021.09)，京东(2021.09)

参考答案：

HiveSQL ->AST(抽象语法树) -> QB(查询块) ->OperatorTree（操作树）->优化后的操作树->mapreduce任务树->优化后的mapreduce任务树







过程描述如下：

SQL Parser：Antlr定义SQL的语法规则，完成SQL词法，语法解析，将SQL转化为抽象 语法树AST Tree

Semantic Analyzer：遍历AST Tree，抽象出查询的基本组成单元QueryBlock

Logical plan：遍历QueryBlock，翻译为执行操作树OperatorTree

Logical plan optimizer：逻辑层优化器进行OperatorTree变换，合并不必要的ReduceSinkOperator，减少shuffle数据量

Physical plan：遍历OperatorTree，翻译为MapReduce任务

Logical plan optimizer：物理层优化器进行MapReduce任务的变换，生成最终的执行计划

Hive的函数：UDF、UDAF、UDTF的区别?

问过的一些公司：好未来(2021.08)，字节(2021.08)

参考答案：

UDF：单行进入，单行输出

UDAF：多行进入，单行输出

UDTF：单行输入，多行输出

UDF是怎么在Hive里执行的

问过的一些公司：携程(2021.09)

参考答案：

公众号：旧时光大数据

打包成jar上传到集群，注册自定义函数，通过类加载器载入系统，在sql解析的过程中去调用函数

Hive优化

问过的一些公司：货拉拉(2021.07)，字节(2021.08)-(2021.09)，京东(2021.09)x3，蔚来(2021.09)

参考答案：

Fetch抓取

Fetch抓取是指，Hive中对某些情况的查询可以不必使用MapReduce计算。例如：SELECT * FROM employees;在这种情况下，Hive可以简单地读取employee对应的存储目录下的文件，然后输出查询结果到控制台

在hive-default.xml.template文件中hive.fetch.task.conversion默认是more，老版本hive默认是minimal，该属性修改为more以后，在全局查找、字段查找、limit查找等都不走mapreduce

本地模式

大多数的Hadoop Job是需要Hadoop提供的完整的可扩展性来处理大数据集的。不过，有时Hive的输入数据量是非常小的。在这种情况下，为查询触发执行任务时消耗可能会比实际job的执行时间要多的多。对于大多数这种情况，Hive可以通过本地模式在单台机器上处理所有的任务。对于小数据集，执行时间可以明显被缩短

用户可以通过设置hive.exec.mode.local.auto的值为true，来让Hive在适当的时候自动启动这个优化

表的优化

1）**小表、大表join**

将key相对分散，并且数据量小的表放在join的左边，这样可以有效减少内存溢出错误发生的几率；再进一步，可以使用Group让小的维度表（1000条以下的记录条数）先进内存。在map端完成reduce

实际测试发现：新版的hive已经对小表JOIN大表和大表JOIN小表进行了优化。小表放在左边和右边已经没有明显区别

2）**大表Join小表**

空KEY过滤

有时join超时是因为某些key对应的数据太多，而相同key对应的数据都会发送到相同的reducer上，从而导致内存不够。此时我们应该仔细分析这些异常的key，很多情况下，这些key对应的数据是异常数据，我们需要在SQL语句中进行过滤。例如key对应的字段为空

3）**Group By**

默认情况下，Map阶段同一Key数据分发给一个reduce，当一个key数据过大时就倾斜了

并不是所有的聚合操作都需要在Reduce端完成，很多聚合操作都可以先在Map端进行部分聚合，最后在Reduce端得出最终结果

1）开启Map端聚合参数设置

（1）是否在Map端进行聚合，默认为True

```
hive.map.aggr = true
```

（2）在Map端进行聚合操作的条目数目

```
hive.groupby.mapaggr.checkinterval = 100000
```

（3）有数据倾斜的时候进行负载均衡（默认是false）

```
hive.groupby.skewindata = true
```

当选项设定为 true，生成的查询计划会有两个MR Job。第一个MR Job中，Map的输出结果会随机分布到Reduce中，每个Reduce做部分聚合操作，并输出结果，这样处理的结果是相同的Group By Key有可能被分发到不同的Reduce中，从而达到负载均衡的目的；第二个MR Job再根据预处理的数据结果按照Group By Key分布到Reduce中（这个过程可以保证相同的Group By Key被分布到同一个Reduce中），最后完成最终的聚合操作

4）**Count(Distinct) 去重统计**

数据量小的时候无所谓，数据量大的情况下，由于COUNT DISTINCT操作需要用一个Reduce Task来完成，这一个Reduce需要处理的数据量太大，就会导致整个Job很难完成，一般COUNT DISTINCT使用先GROUP BY再COUNT的方式替换

5）**笛卡尔积**

尽量避免笛卡尔积，join的时候不加on条件，或者无效的on条件，Hive只能使用1个reducer来完成笛卡尔积

6）**行列过滤**

列处理：在SELECT中，只拿需要的列，如果有，尽量使用分区过滤，少用SELECT *。

行处理：在分区剪裁中，当使用外关联时，如果将副表的过滤条件写在Where后面，那么就会先全表关联，之后再过滤

数据倾斜部分就参考前面的题目就行

row_number，rank，dense_rank的区别

问过的一些公司：美团(2021.09)

参考答案：

row_number()：从1开始，按照顺序，生成分组内记录的序列,row_number()的值不会存在重复,当排序的值相同时,按照表中记录的顺序进行排列;通常用于获取分组内排序第一的记录;获取一个session中的第一条refer等。

rank()：生成数据项在分组中的排名，排名相等会在名次中留下空位。

dense_rank()：生成数据项在分组中的排名，排名相等会在名次中不会留下空位。

注意： rank和dense_rank的区别在于排名相等时会不会留下空位

Hive count(distinct)有几个reduce，海量数据会有什么问题

问过的一些公司：字节(2021.07)

参考答案：

count(distinct)只有1个reduce。

公众号：旧时光大数据

为什么只有一个reducer呢，因为使用了distinct和count(full aggreates)，这两个函数产生的mr作业只会产生一个reducer，而且哪怕显式指定set mapred.reduce.tasks=100000也是没用的。

当使用count(distinct)处理海量数据（比如达到一亿以上）时，会使得运行速度变得很慢，熟悉mr原理的就明白这时sql跑的慢的原因，因为出现了很严重的数据倾斜。

案例分析：

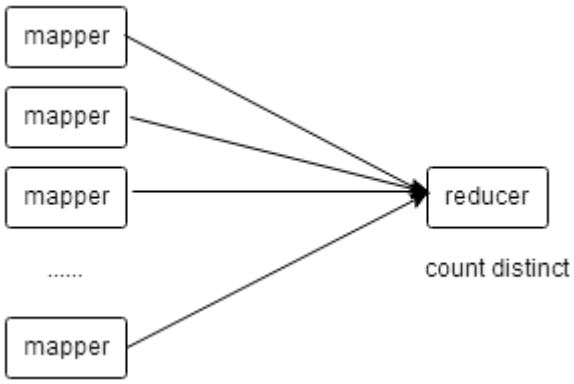
做去重统计时，一般都这么写：

```
select
  count(distinct (bill_no)) as visit_users
from
  i_usoc_user_info_d
where
  p_day = '20210508'
  and bill_no is not null
  and bill_no != ''

```

其实看起来，这没有任务毛病，但我们需要注意的是，此时写的是hql，它的底层引擎是MapReduce，是分布式计算的，所以就会出现数据倾斜这种分布式计算的典型问题，比如上面的使用数仓中一张沉淀了所有用户信息的融合模型来统计所有的手机号码的个数，这种写法肯定是能跑出结果的，但运行时长可能就会有点长。

我们去查下，就会发现记录数至少上亿，去hdfs中查看文件会发现这个分区很大，并且此时，我们通过查看执行计划和日志可以发现只有一个stage。也就是说最后只有一个reduce。

熟悉mr原理的已经明白了这条sql跑的慢的原因，因为出现了很严重的数据倾斜，几百个mapper，1个reducer，所有的数据在mapper处理过后全部只流向了一个reducer，逻辑计划大概如下：



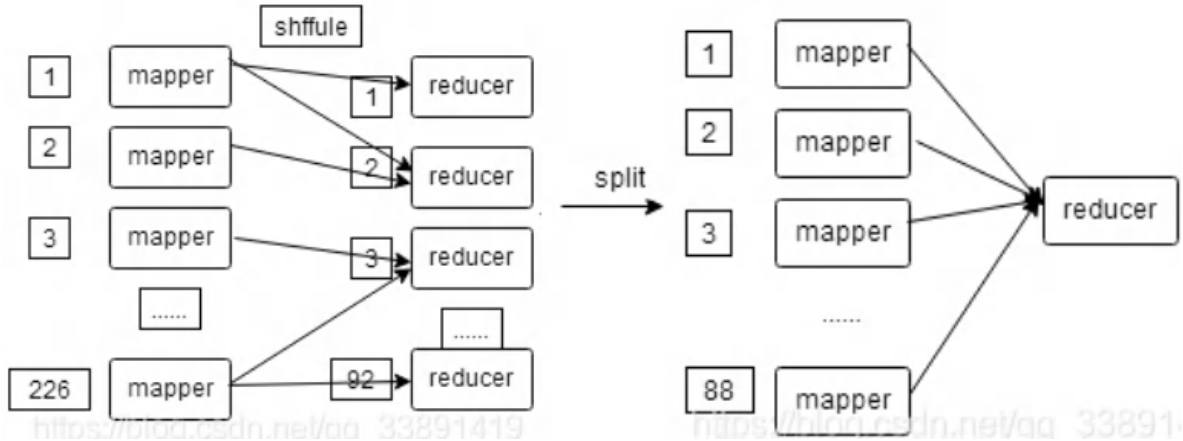
为什么只有一个reducer呢，因为使用了distinct和count(full aggreates)，这两个函数产生的mr作业只会产生一个reducer，而且哪怕显式指定set mapred.reduce.tasks=100000也是没用的。

所以对于这种去重统计，如果在数据量够大，一般是一亿记录数以上(视公司的集群规模，计算能力而定)，建议选择使用count加group by去进行统计：

```
select
  count(a.bill_no)
from
  (
    select
      bill_no
    from
      dwfu_hive_db.i_usoc_user_info_d
    where
      p_day = '20200408'
      and bill_no is not null
```

```
        and bill_no != ''
      group by
        bill_no
    ) a

```

这时候再测试，会发现速度会快很多，查看执行计划和日志，会发现启动了多个stage，也就是多个mr作业，这是因为引入了group by将数据分组到了多个reducer上进行处理。逻辑执行图大致如下：



总结：在数据量很大的情况下，使用count+group by替换count(distinct)能使作业执行效率和速度得到很大的提升，一般来说数据量越大提升效果越明显。

注意：开发前最好核查数据量，别什么几万条几十万条几十M数据去重统计就count加groupby就咔咔往上写，最后发现速度根本没有直接count(distinct)快，作业还没起起来人家count(distinct)就完事结果出来了，所以优化还得建立在一个数据量的问题上，这也是跟其他sql的区别。

HQL：行转列、列转行

可回答：Hive中怎么实现列转行，行转列?

问过的一些公司：Shopee(2021.07)，美团(2021.08)x2

参考答案：

1、行转列：UDF聚合函数

1）相关函数

concat：返回输入字符串连接后的结果，支持任意个输入字符串;

concat_ws(separator, str1, str2,...)：它是一个特殊形式的 concat()。第一个 **参数** 剩余参数间的分隔符。分隔符可以是与剩余参数一样的字符串。如果分隔符是 NULL，返回值也将为 NULL。这个函数会跳过分隔符参数后的任何 NULL 和空字符串。分隔符将被加到被连接的字符串之间;

collect_set(col)：函数只接受基本数据类型，它的主要作用是将某字段的值进行去重汇总，产生array类型字段。

2）数据准备

公众号：旧时光大数据

name	constellation	blood_type
孙悟空	白羊座	A
大海	射手座	A
宋宋	白羊座	B
猪八戒	白羊座	A
凤姐	射手座	A

3）创建hive表并导入数据

```
create table person_info(
    name string,
    constellation string,
    blood_type string)
row format delimited fields terminated by "t";
load data local inpath "/opt/module/datas/constellation.txt" into table
person_info;

```

4）需求

把星座和血型一样的人归类到一起。结果如下：

射手座,A	大海|凤姐
白羊座,A	孙悟空|猪八戒
白羊座,B	宋宋

```
select name,concat(constellation, ",",blood_type) as base from person_info;

```

```
 select t1.base concat_ws('|',collect_set(t1.name)) name
 from(
     select
     name,
     concat(constellation, ",",blood_type) as base
     from
     person_info;
     ) t1
 group by t1.base;

```

2、列转行：UDTF爆炸函数

1）相关函数

explode(col)：将hive一列中复杂的array或者map结构拆分成多行。

lateral view

用法：lateral view udtf(expression) tableAlias AS columnAlias

解释：用于和split, explode等UDTF一起使用，它能够将一列数据拆成多行数据，在此基础上可以对拆分后的数据进行聚合。

公众号：旧时光大数据

2）数据准备

movie	category
《疑犯追踪》	悬疑，动作，科幻，剧情
《Lie to me》	悬疑，警匪，动作，心里，剧情
《战狼2》	战争，动作，灾难

3）创建hive表并导入数据

```
create table movie_info(
movie string,
category array<string>)
row format delimited fields terminated by "t"
collection items terminated by ",";

```

```
load data local inpath "/opt/module/datas/movie.txt" into table movie_info;

```

4）需求

将电影分类中的数组数据展开

```
select movie,category_name from movie_info lateral view explode(category)
table_category as category_name;

```

一条HQL从代码到执行的过程

问过的一些公司：Shopee(2021.08)，兴业数金(2021.09)

参考答案：

Hive在执行一条HQL的时候，会经过以下步骤：

1）语法解析：Antlr定义SQL的语法规则，完成SQL词法，语法解析，将SQL转化为抽象 语法树AST Tree；

2）语义解析：遍历AST Tree，抽象出查询的基本组成单元QueryBlock；

3）生成逻辑执行计划：遍历QueryBlock，翻译为执行操作树OperatorTree；

4）优化逻辑执行计划：逻辑层优化器进行OperatorTree变换，合并不必要的ReduceSinkOperator，减少shuffle数据量；

5）生成物理执行计划：遍历OperatorTree，翻译为MapReduce任务；

6）优化物理执行计划：物理层优化器进行MapReduce任务的变换，生成最终的执行计划；

另一种说法：

1）输入一条HQL查询语句（select * from tab）

2）解析器对这条HQL语句进行语法分析。

3）编译器对这条HQL语句生成HQL的执行计划。

公众号：旧时光大数据

4）优化器生成最佳的HQL的执行计划。

5）执行这条最佳HQL语句。

了解Hive SQL吗？讲讲分析函数？

问过的一些公司：

参考答案：

1、Hive SQL

Hive是一个数据仓库基础的应用工具，在Hadoop中用来处理结构化数据，它架构在Hadoop之上，通过SQL来对数据进行操作。Hive将用户的Hive SQL语句通过解释器转换为MapReduce作业提交到Hadoop集群上，Hadoop 监控作业执行过程，然后返回执行结果给用户。

Hive SQL跟SQL是有区别的：

- HQL不支持行级别的增、改、删，所有数据在加载时就已经确定，不可更改。
- 不支持事务。
- 支持分区存储。

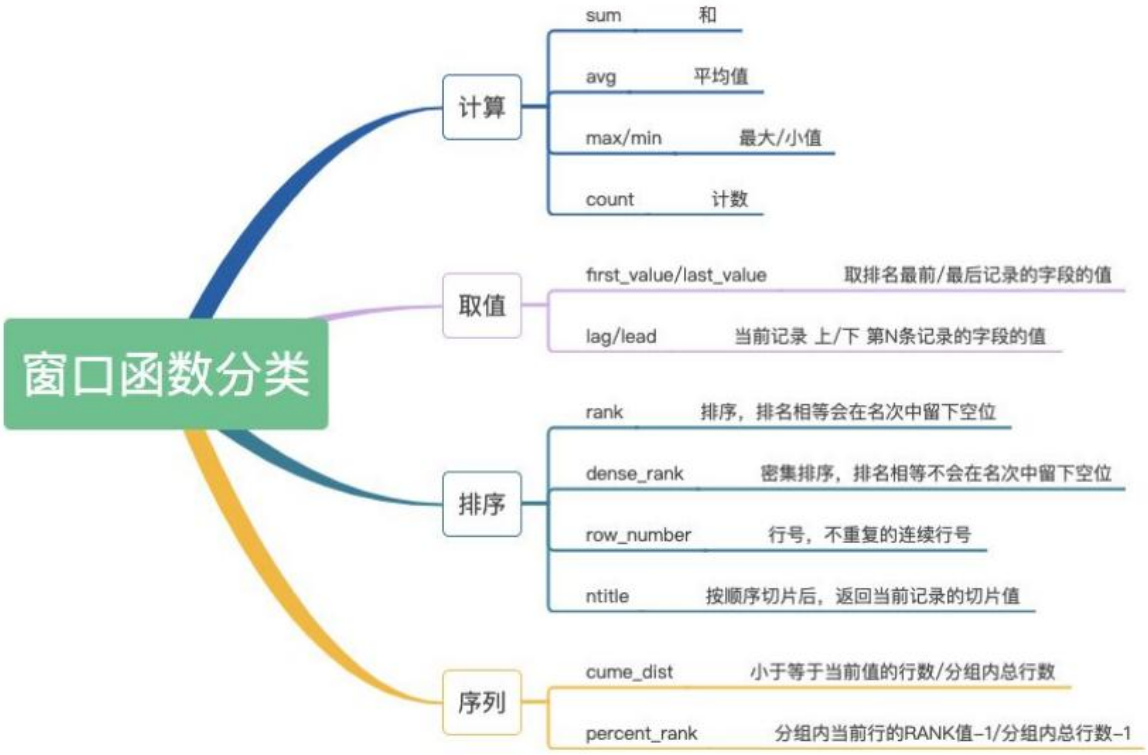
Hive下的SQL特点：

- 不支持等值连接，一般使用left join、right join 或者inner join替代。
- 不能智能识别concat(';',key)，只会将'；'当做SQL结束符号。
- 不支持INSERT INTO 表 Values（），UPDATE, DELETE等操作
- Hive SQL中String类型的字段若是空(empty)字符串, 即长度为0, 那么对它进行IS NULL的判断结果是False，使用left join可以进行筛选行。
- 不支持 '< dt <'这种格式的范围查找，可以用dt in("","")或者between替代。

2、分析函数

窗口函数又叫OLAP函数/分析函数，窗口函数兼具分组和排序功能。

按照窗口函数的功能分为：**计算、取值、排序、序列**四种，前三种的使用场景比较常见，容易理解，最后一种(序列)的使用场景比较少。



窗口函数使用场景

后续补

分析函数中加Order By和不加Order By的区别?

问过的一些公司：虎牙(2021.08)

参考答案：

over()开窗函数前分排序函数和聚合函数两种

当为**排序函数**，如row_number(),rank()等时，over中的order by只起到窗口内排序作用。

当为**聚合函数**，如max，min，count等时，over中的order by不仅起到窗口内排序，还起到窗口内从当前行到之前所有行的聚合（多了一个范围）。

1）以**聚合函数**sum举例，下面的结果中，左图为不加order by的结果，右图为加上order by的结果

发现不加order by score的话，就是针对一整个分区进行sum求和。加上order by score后，就是根据当前行到之前所有行聚合。

```
1   select *,sum(score) over (partition by clazz) as num from windowtest;
2   select *,sum(score) over (partition by clazz order by score) as num from
    windowtest;
3
```

查询结果：

111	69	class1	930
216	74	class1	930
215	82	class1	930
214	94	class1	930
213	94	class1	930
212	83	class1	930
211	93	class1	930
115	93	class1	930
114	94	class1	930
113	74	class1	930
112	80	class1	930
223	74	class2	724
222	78	class2	724
221	99	class2	724
225	85	class2	724
124	70	class2	724
123	78	class2	724
122	86	class2	724
121	74	class2	724
224	80	class2	724

111	69	class1	69
113	74	class1	217
216	74	class1	217
112	80	class1	297
215	82	class1	379
212	83	class1	462
211	93	class1	648
115	93	class1	648
213	94	class1	930
114	94	class1	930
214	94	class1	930
124	70	class2	70
121	74	class2	218
223	74	class2	218
222	78	class2	374
123	78	class2	374
224	80	class2	454
225	85	class2	539
122	86	class2	625
221	99	class2	724

也就是说上面的select *,sum(score) over (partition by clazz order by score) as num from windowtest;其实**等价于**下面这句

```
1   select *,sum(score) over (partition by clazz order by score range between
    unbounded preceding and current row) as num from windowtest;
2
```

测试后，发现结果一致，说明聚合函数加上order by，就等同于加上从当前行到之前所有行的窗口帧。

2）以**排序函数**为例，下面的排序函数结果中，左图为加上order by的结果，右图为不加order by的结果

```
1   select *,row_number() over (partition by clazz order by score) from windowtest;
2   select *,row_number() over (partition by clazz) from windowtest;
3
```

查询结果：

111	69	class1	1
113	74	class1	2
216	74	class1	3
112	80	class1	4
215	82	class1	5
212	83	class1	6
211	93	class1	7
115	93	class1	8
213	94	class1	9
114	94	class1	10
214	94	class1	11
124	70	class2	1
121	74	class2	2
223	74	class2	3
222	78	class2	4
123	78	class2	5
224	80	class2	6
225	85	class2	7
122	86	class2	8
221	99	class2	9

111	69	class1	1
216	74	class1	2
215	82	class1	3
214	94	class1	4
213	94	class1	5
212	83	class1	6
211	93	class1	7
115	93	class1	8
114	94	class1	9
113	74	class1	10
112	80	class1	11
223	74	class2	1
222	78	class2	2
221	99	class2	3
225	85	class2	4
124	70	class2	5
123	78	class2	6
122	86	class2	7
121	74	class2	8
224	80	class2	9

公众号：旧时光大数据

Hive优化方法

问过的一些公司：字节(2021.08)

参考答案：

1、列裁剪和分区裁剪

最基本的操作。所谓列裁剪就是在查询时只读取需要的列，分区裁剪就是只读取需要的分区。

案例：

```
select uid,event_type,record_data
from calendar_record_log
where pt_date >= 20190201 and pt_date <= 20190224
and status = 0;

```

当列很多或者数据量很大时，如果select *或者不指定分区，全列扫描和全表扫描效率都很低。
Hive中与列裁剪优化相关的配置项是`hive.optimize.cp`，与分区裁剪优化相关的则是`hive.optimize.pruner`，默认都是true。在HiveQL解析阶段对应的则是ColumnPruner逻辑优化器。

2、谓词下推

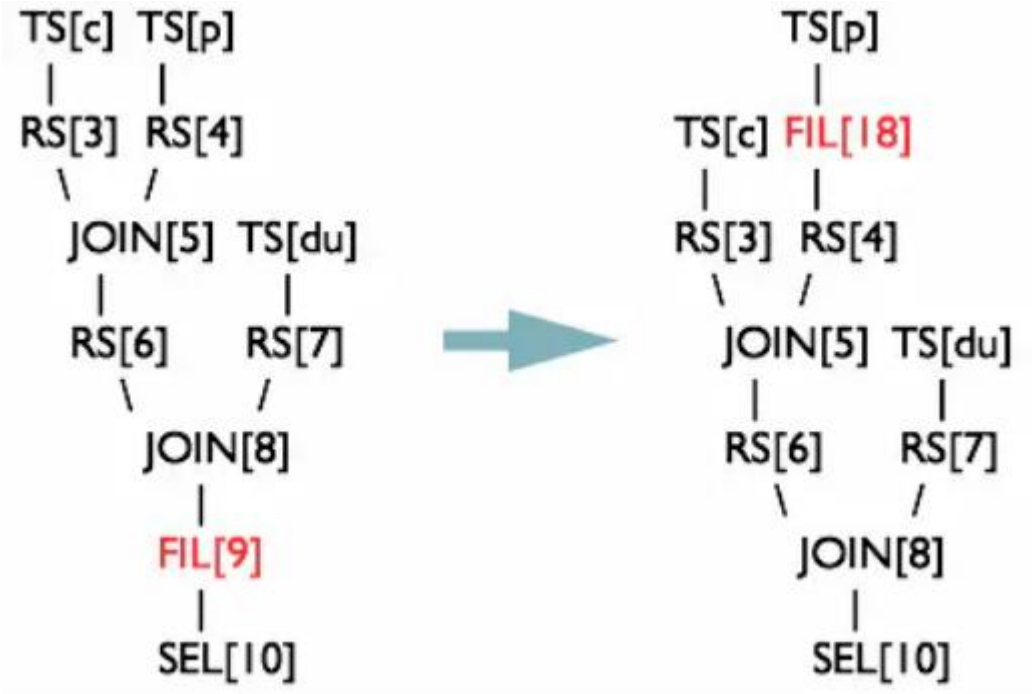
在关系型数据库如MySQL中，也有谓词下推（Predicate Pushdown，PPD）的概念。它就是将SQL语句中的where谓词逻辑都尽可能提前执行，减少下游处理的数据量。

例如以下HQL语句：

```
select a.uid,a.event_type,b.topic_id,b.title
from calendar_record_log a
left outer join (
  select uid,topic_id,title from forum_topic
  where pt_date = 20190224 and length(content) >= 100
) b on a.uid = b.uid
where a.pt_date = 20190224 and status = 0;

```

对forum_topic做过滤的where语句写在子查询内部，而不是外部。Hive中有谓词下推优化的配置项`hive.optimize.ppd`，默认值true，与它对应的逻辑优化器是PredicatePushDown。该优化器就是将OperatorTree中的FilterOperator向上提，见下图。

公众号：旧时光大数据



3、sort by代替order by

HQL中的order by与其他SQL方言中的功能一样，就是将结果按某字段全局排序，这会导致所有map端数据都进入一个reducer中，在数据量大时可能会长时间计算不完。

如果使用sort by，那么还是会视情况启动多个reducer进行排序，并且保证每个reducer内局部有序。为了控制map端数据分配到reducer的key，往往还要配合distribute by一同使用。如果不加distribute by的话，map端数据就会随机分配到reducer。

举个例子，假如要以UID为key，以上传时间倒序、记录类型倒序输出记录数据：

```
select uid,upload_time,event_type,record_data
from calendar_record_log
where pt_date >= 20190201 and pt_date <= 20190224
distribute by uid
sort by upload_time desc,event_type desc;

```

4、group by代替distinct

当要统计某一列的去重数时，如果数据量很大，count(distinct)就会非常慢，原因与order by类似，count(distinct)逻辑只会有很少的reducer来处理。这时可以用group by来改写：

```
select count(1) from (
  select uid from calendar_record_log
  where pt_date >= 20190101
  group by uid
) t;

```

但是这样写会启动两个MR job（单纯distinct只会启动一个），所以要确保数据量大到启动job的overhead远小于计算耗时，才考虑这种方法。当数据集很小或者key的倾斜比较明显时，group by还可能会比distinct慢。

那么如何用group by方式同时统计多个列？下面是解决方法：

公众号：旧时光大数据

```
select t.a,sum(t.b),count(t.c),count(t.d) from (
  select a,b,null c,null d from some_table
  union all
  select a,0 b,c,null d from some_table group by a,c
  union all
  select a,0 b,null c,d from some_table group by a,d
) t;

```

5、group by配置调整

1）map端预聚合

group by时，如果先起一个combiner在map端做部分预聚合，可以有效减少shuffle数据量。预聚合的配置项是 `hive.map.aggr` ，默认值true，对应的优化器为GroupByOptimizer，简单方便。
通过 `hive.groupby.mapaggr.checkinterval` 参数也可以设置map端预聚合的行数阈值，超过该值就会分拆job，默认值100000。

2）倾斜均衡配置项

group by时如果某些key对应的数据量过大，就会发生数据倾斜。Hive自带了一个均衡数据倾斜的配置项 `hive.groupby.skewindata` ，默认值false。

其实现方法是在group by时启动两个MR job。第一个job会将map端数据随机输入reducer，每个reducer做部分聚合，相同的key就会分布在不同的reducer中。第二个job再将前面预处理过的数据按key聚合并输出结果，这样就起到了均衡的效果。

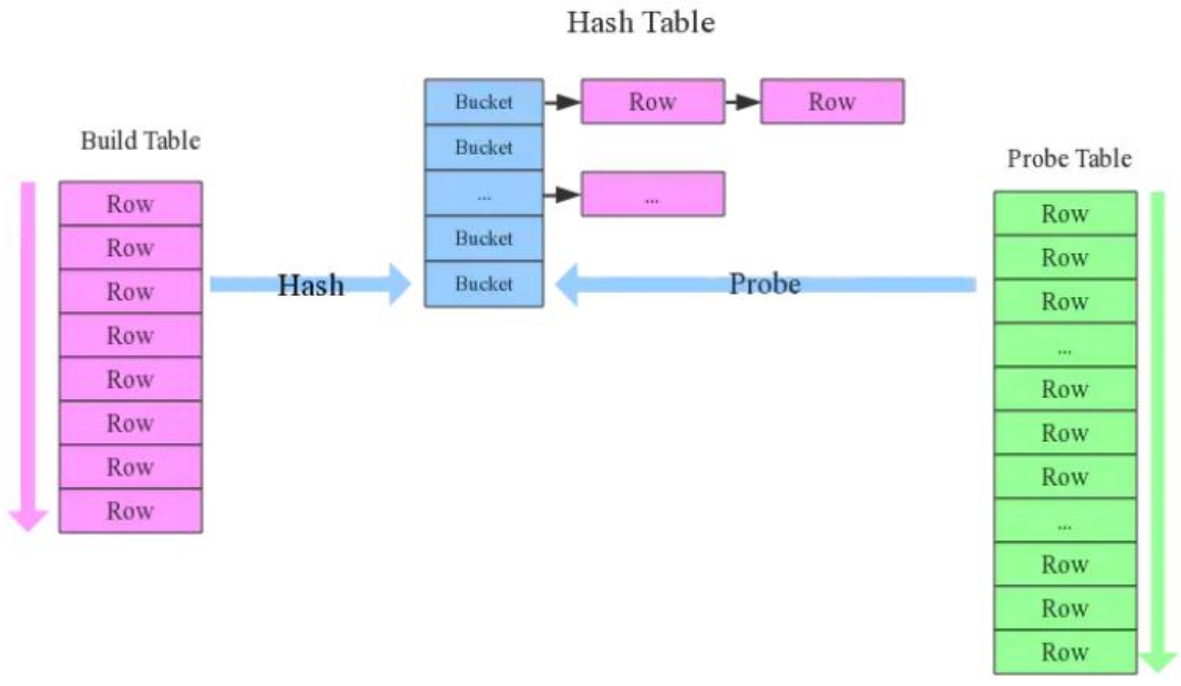
但是，配置项毕竟是死的，单纯靠它有时不能根本上解决问题，因此还是建议自行了解数据倾斜的细节，并优化查询语句。

6、join基础优化

join优化是一个复杂的话题，下面先说5点最基本的注意事项。

1）build table（小表）前置

在最常见的hash join方法中，一般总有一张相对小的表和一张相对大的表，小表叫build table，大表叫probe table。如下图所示。



公众号：旧时光大数据

Hive在解析带join的SQL语句时，会默认将最后一个表作为probe table，将前面的表作为build table并试图将它们读进内存。如果表顺序写反，probe table在前面，引发OOM的风险就高了。

在维度建模数据仓库中，事实表就是probe table，维度表就是build table。假设现在要将日历记录事实表和记录项编码维度表来join：

```
select a.event_type,a.event_code,a.event_desc,b.upload_time
from calendar_event_code a
inner join (
  select event_type,upload_time from calendar_record_log
  where pt_date = 20190225
) b on a.event_type = b.event_type;

```

2）多表join时key相同

这种情况会将多个join合并为一个MR job来处理，例如：

```
select a.event_type,a.event_code,a.event_desc,b.upload_time
from calendar_event_code a
inner join (
  select event_type,upload_time from calendar_record_log
  where pt_date = 20190225
) b on a.event_type = b.event_type
inner join (
  select event_type,upload_time from calendar_record_log_2
  where pt_date = 20190225
) c on a.event_type = c.event_type;

```

如果上面两个join的条件不相同，比如改成`a.event_code = c.event_code`，就会拆成两个MR job计算。

3）利用map join特性

map join特别适合大小表join的情况。Hive会将build table和probe table在map端直接完成join过程，消灭了reduce，效率很高。

```
select /*+mapjoin(a)*/ a.event_type,b.upload_time
from calendar_event_code a
inner join (
  select event_type,upload_time from calendar_record_log
  where pt_date = 20190225
) b on a.event_type < b.event_type;

```

上面的语句中加了一条map join hint，以显式启用map join特性。早在Hive 0.8版本之后，就不需要写这条hint了。map join还支持不等值连接，应用更加灵活。

map join的配置项是`hive.auto.convert.join`，默认值true，对应逻辑优化器是MapJoinProcessor。

还有一些参数用来控制map join的行为，比如`hive.mapjoin.smalltable.filesize`，当build table大小小于该值就会启用map join，默认值25000000（25MB）。还有`hive.mapjoin.cache.numrows`，表示缓存build table的多少行数据到内存，默认值25000。

4）分桶表map join

map join对分桶表还有特别的优化。由于分桶表是基于一列进行hash存储的，因此非常适合抽样（按桶或按块抽样）。

它对应的配置项是`hive.optimize.bucketmapjoin`，优化器是BucketMapJoinOptimizer。

5）倾斜均衡配置项

这个配置与上面group by的倾斜均衡配置项异曲同工，通过`hive.optimize.skewjoin`来配置，默认false。

如果开启了，在join过程中Hive会将计数超过阈值`hive.skewjoin.key`（默认100000）的倾斜key对应的行临时写进文件中，然后再启动另一个job做map join生成结果。通过`hive.skewjoin.mapjoin.map.tasks`参数还可以控制第二个job的mapper数量，默认10000。

再重复一遍，通过自带的配置项经常不能解决数据倾斜问题。join是数据倾斜的重灾区，后面还要介绍在SQL层面处理倾斜的各种方法。

7、优化SQL处理join数据倾斜

1）空值或无意义值

这种情况很常见，比如当事实表是日志类数据时，往往会有一些项没有记录到，我们视情况会将它置为null，或者空字符串、-1等。如果缺失的项很多，在做join时这些空值就会非常集中，拖累进度。

因此，若不需要空值数据，就提前写where语句过滤掉。需要保留的话，将空值key用随机方式打散，例如将用户ID为null的记录随机改为负值：

```
1   select a.uid,a.event_type,b.nickname,b.age
2   from (
3     select
4     (case when uid is null then cast(rand()*-10240 as int) else uid end) as uid,
5     event_type from calendar_record_log
6     where pt_date >= 20190201
7   ) a left outer join (
8     select uid,nickname,age from user_info where status = 4
9   ) b on a.uid = b.uid;
10
```

2）单独处理倾斜key

这其实是上面处理空值方法的拓展，不过倾斜的key变成了有意义的。一般来讲倾斜的key都很少，我们可以将它们抽样出来，对应的行单独存入临时表中，然后打上一个较小的随机数前缀（比如0~9），最后再进行聚合。SQL语句与上面的相仿。

3）不同数据类型

这种情况不太常见，主要出现在相同业务含义的列发生过逻辑上的变化时。

举个例子，假如我们有一旧一新两张日历记录表，旧表的记录类型字段是(event_type int)，新表的是(event_type string)。为了兼容旧版记录，新表的event_type也会以字符串形式存储旧版的值，比如'17'。当这两张表join时，经常要耗费很长时间。其原因就是如果不转换类型，计算key的hash值时默认是以int型做的，这就导致所有“真正的”string型key都分配到一个reducer上。所以要注意类型转换：

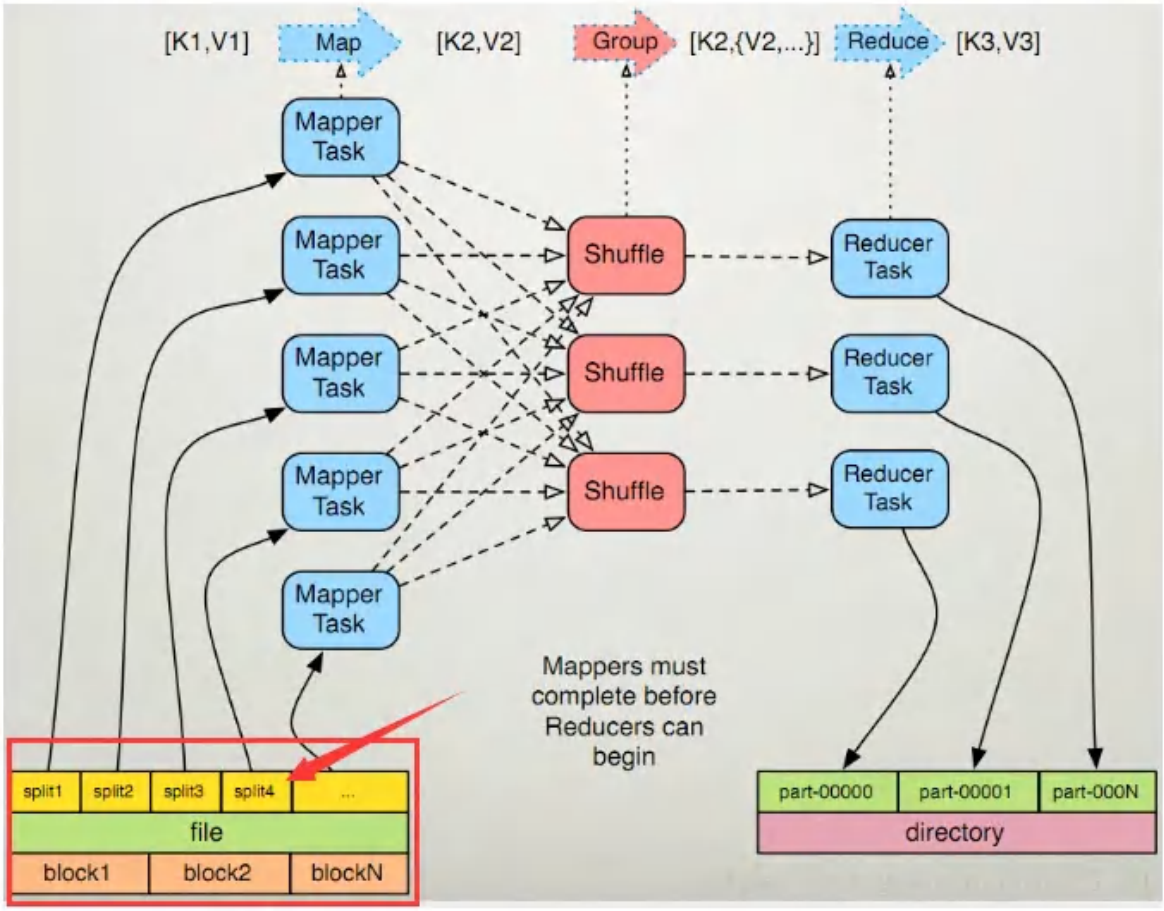
```
1   select a.uid,a.event_type,b.record_data
2   from calendar_record_log a
3   left outer join (
4     select uid,event_type from calendar_record_log_2
5     where pt_date = 20190228
6   ) b on a.uid = b.uid and b.event_type = cast(a.event_type as string)
7   where a.pt_date = 20190228;
8
```

4）build table过大

有时，build table会大到无法直接使用map join的地步，比如全量用户维度表，而使用普通join又有数据分布不均的问题。这时就要充分利用probe table的限制条件，削减build table的数据量，再使用map join解决。代价就是需要进行两次join。举个例子：

```
select /*+mapjoin(b)*/ a.uid,a.event_type,b.status,b.extra_info
from calendar_record_log a
left outer join (
  select /*+mapjoin(s)*/ t.uid,t.status,t.extra_info
  from (select distinct uid from calendar_record_log where pt_date = 20190228) s
  inner join user_info t on s.uid = t.uid
) b on a.uid = b.uid
where a.pt_date = 20190228;

```

8、MapReduce优化



1）调整mapper数

mapper数量与输入文件的split数息息相关，在Hadoop源码`org.apache.hadoop.mapreduce.lib.input.FileInputFormat`类中可以看到split划分的具体逻辑。这里不贴代码，直接叙述mapper数是如何确定的。

- 可以直接通过参数`mapred.map.tasks`（默认值2）来设定mapper数的期望值，但它不一定会生效，下面会提到。
- 设输入文件的总大小为`total_input_size`。HDFS中，一个块的大小由参数`dfs.block.size`指定，默认值64MB或128MB。在默认情况下，mapper数就是：`default_mapper_num = total_input_size / dfs.block.size`。
- 参数`mapred.min.split.size`（默认值1B）和`mapred.max.split.size`（默认值64MB）分别用来指定split的最小和最大大小。split大小和split数计算规则是：`split_size = MAX(mapred.min.split.size, MIN(mapred.max.split.size,`

公众号：旧时光大数据

`dfs.block.size))`；
`split_num = total_input_size / split_size`。
- 得出mapper数：
`mapper_num = MIN(split_num, MAX(default_num, mapred.map.tasks))`。

可见，如果想减少mapper数，就适当调高`mapred.min.split.size`，split数就减少了。如果想增大mapper数，除了降低`mapred.min.split.size`之外，也可以调高`mapred.map.tasks`。

一般来讲，如果输入文件是少量大文件，就减少mapper数；如果输入文件是大量非小文件，就增大mapper数；至于大量小文件的情况，参考下面“合并小文件”的方法处理。

2）调整reducer数

reducer数量的确定方法比mapper简单得多。使用参数`mapred.reduce.tasks`可以直接设定reducer数量，不像mapper一样是期望值。但如果不设这个参数的话，Hive就会自行推测，逻辑如下：

- 参数`hive.exec.reducers.bytes.per.reducer`用来设定每个reducer能够处理的最大数据量，默认值1G（1.2版本之前）或256M（1.2版本之后）。
- 参数`hive.exec.reducers.max`用来设定每个job的最大reducer数量，默认值999（1.2版本之前）或1009（1.2版本之后）。
- 得出reducer数：
`reducer_num = MIN(total_input_size / reducers.bytes.per.reducer, reducers.max)`。

reducer数量与输出文件的数量相关。如果reducer数太多，会产生大量小文件，对HDFS造成压力。如果reducer数太少，每个reducer要处理很多数据，容易拖慢运行时间或者造成OOM。

3）合并小文件

输入阶段合并

- 需要更改Hive的输入文件格式，即参数`hive.input.format`，默认值是`org.apache.hadoop.hive.ql.io.HiveInputFormat`，我们改成`org.apache.hadoop.hive.ql.io.CombineHiveInputFormat`。
这样比起上面调整mapper数时，又会多出两个参数，分别是`mapred.min.split.size.per.node`和`mapred.min.split.size.per.rack`，含义是单节点和单机架上的最小split大小。如果发现有split大小小于这两个值（默认都是100MB），则会进行合并。具体逻辑可以参看Hive源码中的对应类。

输出阶段合并

- 直接将`hive.merge.mapfiles`和`hive.merge.mapredfiles`都设为true即可，前者表示将map-only任务的输出合并，后者表示将map-reduce任务的输出合并。
- `hive.merge.size.per.task`可以指定每个task输出后合并文件大小的期望值，`hive.merge.size.smallfiles.avgsize`可以指定所有输出文件大小的均值阈值，默认值都是1GB。如果平均大小不足的话，就会另外启动一个任务来进行合并。

4）启用压缩

压缩job的中间结果数据和输出数据，可以用少量CPU时间节省很多空间。压缩方式一般选择Snappy，效率最高。

要启用中间压缩，需要设定`hive.exec.compress.intermediate`为true，同时指定压缩方式`hive.intermediate.compression.codec`为`org.apache.hadoop.io.compress.SnappyCodec`。另外，参数`hive.intermediate.compression.type`可以选择对块（BLOCK）还是记录（RECORD）压缩，BLOCK的压缩率比较高。

输出压缩的配置基本相同，打开`hive.exec.compress.output`即可。

公众号：旧时光大数据

5）JVM重用

在MR job中，默认是每执行一个task就启动一个JVM。如果task非常小而碎，那么JVM启动和关闭的耗时就会很长。可以通过调节参数 `mapred.job.reuse.jvm.num.tasks` 来重用。例如将这个参数设成5，那么就代表同一个MR job中顺序执行的5个task可以重复使用一个JVM，减少启动和关闭的开销。但它对不同MR job中的task无效。

9、并行执行与本地模式

并行执行

- Hive中互相没有依赖关系的job间是可以并行执行的，最典型的就是多个子查询union all。在集群资源相对充足的情况下，可以开启并行执行，即将参数 `hive.exec.parallel` 设为true。另外 `hive.exec.parallel.thread.number` 可以设定并行执行的线程数，默认为8，一般都够用。

本地模式

- Hive也可以不将任务提交到集群进行运算，而是直接在一台节点上处理。因为消除了提交到集群的overhead，所以比较适合数据量很小，且逻辑不复杂的任务。
- 设置 `hive.exec.mode.local.auto` 为true可以开启本地模式。但任务的输入数据总量必须小于 `hive.exec.mode.local.auto.inputbytes.max` （默认值128MB），且mapper数必须小于 `hive.exec.mode.local.auto.tasks.max` （默认值4），reducer数必须为0或1，才会真正用本地模式执行。

10、严格模式

所谓严格模式，就是强制不允许用户执行3种有风险的HiveQL语句，一旦执行会直接失败。这3种语句是：

- 查询分区表时不限定分区列的语句；
- 两表join产生了笛卡尔积的语句；
- 用order by来排序但没有指定limit的语句。

要开启严格模式，需要将参数 `hive.mapred.mode` 设为strict。

11、采用合适的存储格式

在HiveQL的create table语句中，可以使用 `stored as ...` 指定表的存储格式。Hive表支持的存储格式有TextFile、SequenceFile、RCFile、Avro、ORC、Parquet等。

存储格式一般需要根据业务进行选择，在我们的实操中，绝大多数表都采用TextFile与Parquet两种存储格式之一。

TextFile是最简单的存储格式，它是纯文本记录，也是Hive的默认格式。虽然它的磁盘开销比较大，查询效率也低，但它更多地是作为跳板来使用。RCFile、ORC、Parquet等格式的表都不能由文件直接导入数据，必须由TextFile来做中转。

Parquet和ORC都是Apache旗下的开源列式存储格式。列式存储比起传统的行式存储更适合批量OLAP查询，并且也支持更好的压缩和编码。我们选择Parquet的原因主要是它支持Impala查询引擎，并且我们对update、delete和事务性操作需求很低。

Hive里metastore是干嘛的

问过的一些公司：兴业数金(2021.09)

参考答案：

1、Metadata概念

元数据包含用Hive创建的database、table等的元信息。元数据存储在关系型数据库中。如Derby、MySQL等。

2、Metastore作用

客户端连接metastore服务，metastore再去连接MySQL数据库来存取元数据。有了metastore服务，就可以有多个客户端同时连接，而且这些客户端不需要知道MySQL数据库的用户名和密码，只需要连接metastore 服务即可。

3、Hive的元数据存储(Metastore三种配置方式)

由于元数据不断地修改、更新，所以Hive元数据不适合存储在HDFS中，一般存在RDBMS中。

1）内嵌模式（Embedded）

hive服务和metastore服务运行在同一个进程中，derby服务也运行在该进程中.内嵌模式使用的是内嵌的Derby数据库来存储元数据，也不需要额外起Metastore服务。

这个是默认的，配置简单，但是一次只能一个客户端连接，适用于用来实验，不适用于生产环境。

2）本地模式（Local）：

本地安装mysql 替代derby存储元数据不再使用内嵌的Derby作为元数据的存储介质，而是使用其他数据库比如MySQL来存储元数据。hive服务和metastore服务运行在同一个进程中，mysql是单独的进程，可以同一台机器，也可以在远程机器上。

这种方式是一个多用户的模式，运行多个用户client连接到一个数据库中。这种方式一般作为公司内部同时使用Hive。每一个用户必须要有对MySQL的访问权利，即每一个客户端使用者需要知道MySQL的用户名和密码才行。

3）远程模式（Remote）：远程安装mysql 替代derby存储元数据

Hive服务和metastore在不同的进程内，可能是不同的机器，该模式需要将hive.metastore.local设置为false，将hive.metastore.uris设置为metastore服务器URL，远程元存储需要单独起metastore服务，然后每个客户端都在配置文件里配置连接到该metastore服务。将metadata作为一个单独的服务进行启动。各种客户端通过beeline来连接，连接之前无需知道数据库的密码。

仅连接远程的mysql并不能称之为“远程模式”，是否远程指的是metastore和hive服务是否在同一进程内.

HiveServer2是什么?

问过的一些公司：兴业数金(2021.09)

参考答案：

HiveServer2（HS2）是一个服务端接口，使远程客户端可以执行对Hive的查询并返回结果。目前基于Thrift RPC的实现是HiveServer的改进版本，并支持多客户端并发和身份验证.

HiveServer的核心是基于Thrift，Thrift负责hive的查询服务，Thtift是构建跨平台的rpc框架，主要由四层组成：server，Transport，Protocol和处理器

server

- HiveServer在TCP模式下使用ThreadPoolServer，在HTTP下使用jetty server
- server主要为每个tpc连接分配一个工作线程

Transport

- 如果客户端与服务器之间需要代码（安全原因），则需要http模式，通过hive配置属性hive.server2.transport.mode指定Thrift服务的传输模式

公众号：旧时光大数据

Protocol

- 协议主要负责序列化和反序列化

处理器

- 处理请求的应用程序框架，实现了编译和执行hive查询的逻辑，负责准备各种执行引擎的物理执行计划

Hive表字段换类型怎么办

问过的一些公司：贝壳找房(2021.11)

参考答案：

1、使用alter table change column命令直接修改

2、如果表中已经有数据的话，通过上面的语句进行修改后会导致数据无法展示。

可以通过下面的方式操作

1）先将要改字段结构的表名 修改为一个临时表

```
1   alter table 表1 rename to 表2;
2
```

2）在创建一个和原来（表1）一模一样的表3

3）将修改表名之后的数据插入到新建的表

```
1   insert into 表3 select * from 表2;
2
```

parquet文件优势

问过的一些公司：

参考答案：

1、更高的压缩比

列存使得更容易对每个列使用高效的压缩和编码，降低磁盘空间。（网上的case是不压缩、gzip、snappy分别能达到11/27/19的压缩比）

2、更小的IO操作

使用映射下推和谓词下推，只读取需要的列，跳过不满足条件的列，能够减少不必要的数据扫描，带来性能的提升并在表字段比较多的时候更加明显。

Flume面试题

介绍下Flume

可回答：Flume主要是用来做什么的?

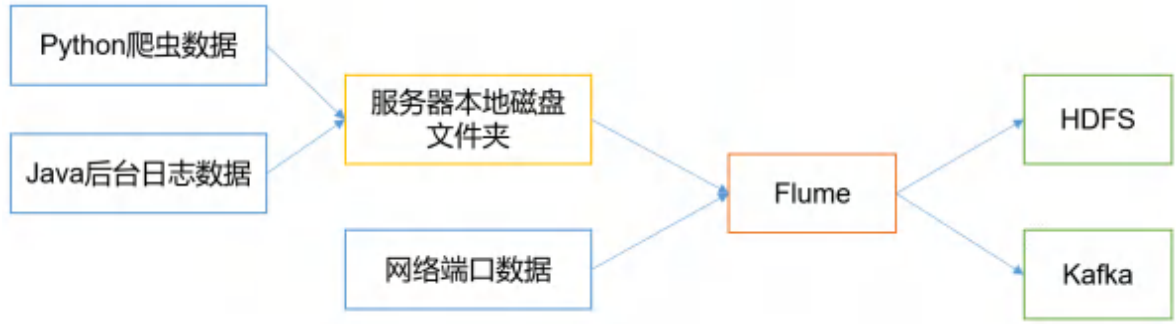
问过的一些公司：阿里，作业帮，快手

参考答案：

1、什么是Flume

Flume是Cloudera提供的**一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统。**2009年被捐赠了apache软件基金会，为hadoop相关组件之一。尤其近几年随着flume的不断被完善以及升级版本的逐一推出，特别是flume-ng，同时flume内部的各种组件不断丰富，用户在开发的过程中使用的便利性得到很大的改善，现已成为apache top项目之一。

Flume可以收集例如日志，事件等数据资源，并将这些数量庞大的数据从各项数据资源中集中起来存储的工具/服务。flume具有高可用，分布式，配置工具，其设计的原理是基于数据流（流式架构，灵活简单），如日志数据从各种网站服务器上汇集起来存储到HDFS，HBase等集中存储器中。其结构如下图所示：



Flume最主要的作用是实时读取服务器本地磁盘的数据，将数据写入HDFS或Kafka等。

2、Flume文件目录

文件夹	说明
bin	存放了启动脚本
lib	启动所需的所有组件jar包
conf	提供了几个测试配置文件
docs	文档

3、Flume的Agent组件

Flume内部有一个或者多个Agent，然而对于每一个Agent来说，它就是一共独立的守护进程(JVM)，它从客户端哪儿接收收集，或者从其他的 Agent哪儿接收，然后迅速的将获取的数据传给下一个目的节点sink，或者agent。

每个Agent包含了Source、Channel和Sink，具体介绍见下一题Flume架构。

4、Flume优缺点

优点

1）Flume可以将应用产生的数据存储到任何集中存储器中，比如HDFS,HBase。

2）当收集数据的速度超过将写入数据的时候，也就是当收集信息遇到峰值时，这时候收集的信息非常大，甚至超过了系统的写入数据能力，这时候，Flume会在数据生产者和数据收容器间做出调整，保证其能够在两者之间提供一共平稳的数据。

3）提供上下文路由特征。

4）Flume的管道是基于事务，保证了数据在传送和接收时的一致性。

公众号：旧时光大数据

5）Flume是可靠的，容错性高的，可升级的，易管理的,并且可定制的。

6）实时性，Flume有一个好处可以实时的将分析数据并将数据保存在数据库或者其他系统中。

缺点

Flume的配置很繁琐，source，channel，sink的关系在配置文件里面交织在一起，不便于管理。

5、应用场景

1）电子商务网站

比如我们在做一个电子商务网站，然后我们想从消费用户中访问点特定的节点区域来分析消费者的行为或者购买意图。这样我们就可以更加快速的将他想要的推送到界面上，实现这一点，我们需要将获取到的她访问的页面以及点击的产品数据等日志数据信息收集并移交给Hadoop平台上去分析，而Flume正是帮我们做到这一点。

2）内容推送

现在流行的内容推送，比如广告定点投放以及新闻私人定制也是基于此。

3）ETL工具

可以利用插件把关系型数据实时增量的导入到Hdfs外部数据源。

6、Flume插件

1）Interceptors拦截器

用于source和channel之间，用来更改或者检查Flume的events数据

2）管道选择器channels Selectors

在多管道是被用来选择使用那一条管道来传递数据(events)。管道选择器又分为如下两种:

- 默认管道选择器: 每一个管道传递的都是相同的events
- 多路复用通道选择器: 依据每一个event的头部header的地址选择管道

3）sink线程

用于激活被选择的sinks群中特定的sink，用于负载均衡

7、其它类似Flume框架

Facebook的Scribe，还有Apache另一个明星项目chukwa，还有淘宝Time Tunnel。

Flume架构

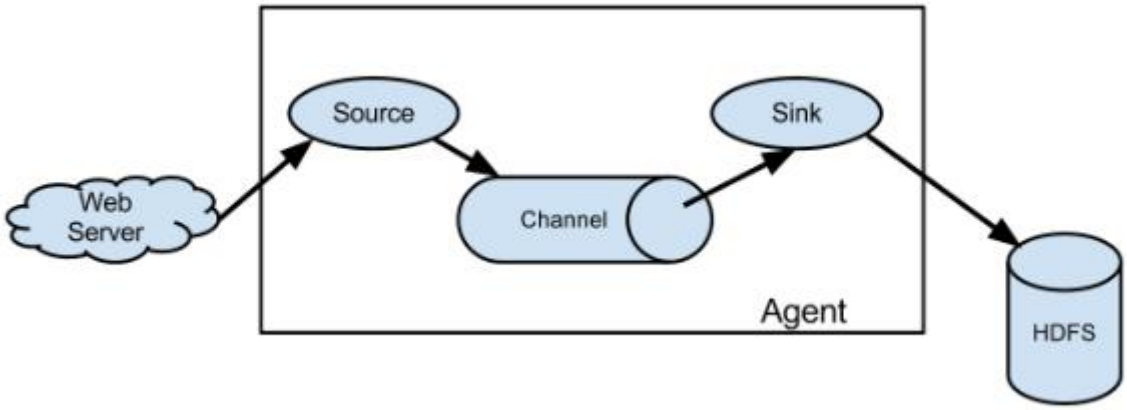
可回答：1）Flume的source、channel、sink分别有什么类型的？2）Flume收集工具有哪些部分组成。

问过的一些公司：腾讯，字节x2，bigo，阿里社招，快手x2，流利说，转转，猿辅导，宇信科技，创略科技，多益x3，富途，兴业数金(2021.09)

参考答案：

Flume组成架构如下图所示

公众号：旧时光大数据



Agent

Agent是一个JVM进程，它以事件的形式将数据从源头送至目的。

Agent主要有3个部分组成，Source、Channel、Sink。

Source

Source负责数据的产生或搜集，一般是对接一些RPC的程序或者是其他的flume节点的sink，从数据发生器接收数据，并将接收的数据以Flume的event格式传递给一个或者多个通道Channel，Flume提供多种数据接收的方式，包括**avro**、thrift、**exec**、jms、**spooling directory**、**netcat**、**taildir**、sequence generator、syslog、http、legacy、自定义。

类型	描述
Avro	监听Avro端口并接收Avro Client的流数据
Thrift	监听Thrift端口并接收Thrift Client的流数据
Exec	基于Unix的command在标准输出上生产数据
JMS	从JMS（Java消息服务）采集数据
Spooling Directory	监听指定目录
Twitter 1%	通过API持续下载Twitter数据（实验阶段）
Kafka	采集Kafka topic中的message
NetCat	监听端口（要求所提供的数据是换行符分隔的文本）
Sequence Generator	序列产生器，连续不断产生event，用于测试
Syslog	采集syslog日志消息，支持单端口TCP、多端口TCP和UDP日志采集
HTTP	接收HTTP POST和GET数据
Stress	用于source压力测试
Legacy	向下兼容，接收低版本Flume的数据
Custom	自定义source的接口
Scribe	从facebook Scribe采集数据

Channel

公众号：旧时光大数据

Channel 是一种短暂的存储容器，负责数据的存储持久化，可以持久化到jdbc、file、memory，将从source处接收到的event格式的数据缓存起来，直到它们被sinks消费掉。可以把channel看成是一个队列，队列的优点是先进先出，放好后尾部一个个event出来，Flume比较看重数据的传输，因此几乎没有数据的解析预处理。仅仅是数据的产生，封装成event然后传输。数据只有存储在下一个存储位置（可能是最终的存储位置，如HDFS，也可能是下一个Flume节点的channel），数据才会从当前的channel中删除。这个过程是通过事务来控制的，这样就保证了数据的可靠性。

不过flume的持久化也是有容量限制的，比如内存如果超过一定的量，不够分配，也一样会爆掉。

类型	描述
Memory	Event数据存储在内存中
JDBC	Event数据存储在持久化存储中，当前Flume Channel内置支持Derby
Kafka	Event存储在kafka集群
File Channel	Event数据存储在磁盘文件中
Spillable Memory Channel	Event数据存储在内存中和磁盘上，当内存队列满了，会持久化到磁盘文件（当前试验性的，不建议生产环境使用）
Pseudo Transaction Channel	测试用途
Custom Channel	自定义Channel实现

Channel是位于Source和Sink之间的缓冲区。因此，Channel允许Source和Sink运作在不同的速率上。Channel是线程安全的，可以同时处理几个Source的写入操作和几个Sink的读取操作。

注意

- Memory Channel是内存中的队列。Memory Channel在不需要关心数据丢失的情景下适用。如果需要关心数据丢失，那么Memory Channel就不应该使用，因为程序死亡、机器宕机或者重启都会导致数据丢失。
- File Channel将所有事件写到磁盘。因此在程序关闭或机器宕机的情况下不会丢失数据。

`Sink`

Sink不断地轮询Channel中的事件且批量地移除它们，并将这些事件批量写入到存储或索引系统、或者被发送到另一个Flume Agent。

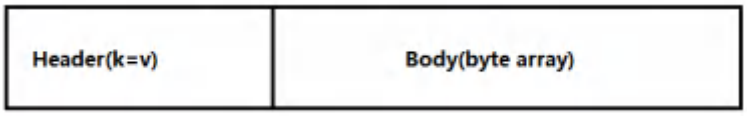
Sink组件目的地包括**hdfs、Kafka、logger、avro**、thrift、ipc、**file、HBase**、solr、自定义。

公众号：旧时光大数据

类型	描述
HDFS	数据写入HDFS
HIVE	数据导入到HIVE中
Logger	数据写入日志文件
Avro	数据被转换成Avro Event，然后发送到配置的RPC端口上
Thrift	数据被转换成Thrift Event，然后发送到配置的RPC端口上
IRC	数据在IRC上进行回放
File Roll	存储数据到本地文件系统
Null	丢弃到所有数据
Hive	数据写入Hive
HBase	数据写入HBase数据库
Morphline Solr	数据发送到Solr搜索服务器（集群）
ElasticSearch	数据发送到Elastic Search搜索服务器（集群）
Kite Dataset	写数据到Kite Dataset，试验性质的
Kafka	数据写到Kafka Topic
Custom	自定义Sink实现

Event

传输单元，Flume数据传输的基本单元，以Event的形式将数据从源头送至目的地。Event由Header和Body两部分组成，Header用来存放该event的一些属性，为K-V结构，Body用来存放该条数据，形式为字节数组。



Flume有哪些Source

问过的一些公司：兴业数金(2021.09)

参考答案：

Source负责数据的产生或搜集，一般是对接一些RPC的程序或者是其他的flume节点的sink，从数据发生器接收数据，并将接收的数据以Flume的event格式传递给一个或者多个通道Channel，Flume提供多种数据接收的方式，包括**avro**、thrift、**exec**、jms、**spooling directory**、**netcat**、**taildir**、sequence generator、syslog、http、legacy、自定义。

类型	描述
Avro	监听Avro端口并接收Avro Client的流数据
Thrift	监听Thrift端口并接收Thrift Client的流数据
Exec	基于Unix的command在标准输出上生产数据
JMS	从JMS（Java消息服务）采集数据
Spooling Directory	监听指定目录
Twitter 1%	通过API持续下载Twitter数据（实验阶段）
Kafka	采集Kafka topic中的message
NetCat	监听端口（要求所提供的数据是换行符分隔的文本）
Sequence Generator	序列产生器，连续不断产生event，用于测试
Syslog	采集syslog日志消息，支持单端口TCP、多端口TCP和UDP日志采集
HTTP	接收HTTP POST和GET数据
Stress	用于source压力测试
Legacy	向下兼容，接收低版本Flume的数据
Custom	自定义source的接口
Scribe	从facebook Scribe采集数据

说下Flume事务机制

问过的一些公司：流利说

参考答案：

Flume的事务机制（类似数据库的事务机制）：Flume使用两个独立的事务分别负责从Soucrce到Channel，以及从Channel到Sink的事件传递。比如spooling directory source 为文件的每一行创建一个事件，一旦事务中所有的事件全部传递到Channel且提交成功，那么Soucrce就将该文件标记为完成。同理，事务以类似的方式处理从Channel到Sink的传递过程，如果因为某种原因使得事件无法记录，那么事务将会回滚。且所有的事件都会保持到Channel中，等待重新传递。

介绍下Flume采集数据的原理？底层实现？

问过的一些公司：百度，蘑菇街，大华

参考答案：

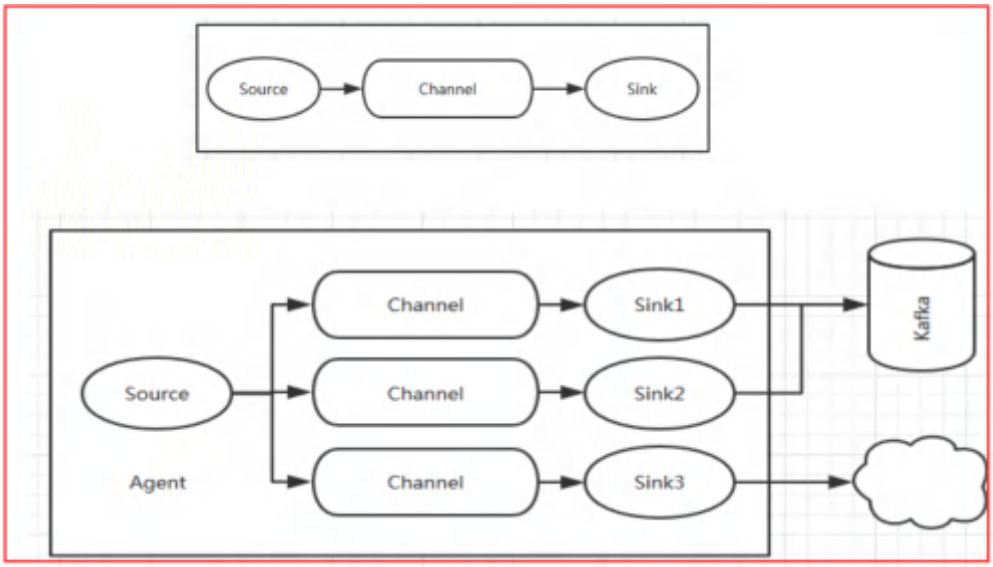
1）Flume的核心是把数据从数据源收集过来，再送到目的地。为了保证输送一定成功，在送到目的地之前，会先缓存数据，待数据真正到达目的地后，删除自己缓存的数据。

2）Flume传输的数据的基本单位是Event，如果是文本文件，通常是一行记录，这也是事务的基本单位。Event从Source流向Channel，再到Sink，Event本身为一个byte数组，并可携带headers信息。Event代表着一个数据流的最小完整单元，从外部数据源来，向外部的目的地去。

值得注意的是，Flume提供了大量内置的Source、Channel和Sink类型。不同类型的Source,Channel和Sink可以自由组合。组合方式基于用户设置的配置文件，非常灵活。

比如：Channel可以把事件暂存在内存里，也可以持久化到本地硬盘上。Sink可以把日志写入HDFS, HBase，甚至是另外一个Source等等。Flume支持用户建立多级流，

也就是说，多个agent可以协同工作，并且支持Fan-in、Fan-out、Contextual Routing、Backup Routes，这也正是Flume强大之处。如下图所示：



Flume如何保证数据的可靠性

可回答：1）Flume的数据传输如何保证有消息（不漏传或者重复传）；2）Flume使用的是什么channel，如何去保证可靠性

问过的一些公司：大华，多益，陌陌(2021.07)

参考答案：

1、事务

Flume使用两个独立的事务分别负责从soucrce到channel，以及从channel到sink的事件传递。一旦事务中所有的事件全部传递到channel且提交成功，那么source就将该文件标记为完成。同理，事务以类似的方式处理从channel到sink的传递过程，如果因为某种原因使得事件无法记录，那么事务将会回滚。且所有的事件都会保持到channel中，等待重新传递。

Flume的事务机制，总的来说，保证了source产生的每个事件都会传送到sink中。这样就造成每个source产生的事件至少到达sink一次，换句话说就是同一事件有可能重复到达。

2、Channel配置

Flume中提供的Channel实现主要有三个：

- Memory Channel：event保存在Java Heap中。如果允许数据小量丢失，推荐使用。（宕机可能丢失数据）
- File Channel：event保存在本地文件中，可靠性高，但吞吐量低于Memory Channel
- JDBC Channel：event保存在关系数据中，一般不推荐使用

Flume传输数据时如何保证数据一致性（可靠性）

公众号：旧时光大数据

问过的一些公司：富途

参考答案：

1、Flume的事务机制

Flume使用两个独立的事务分别负责从soucrce到channel，以及从channel到sink的事件传递。

比如：spooling directory source 为文件的每一行创建一个事件，一旦事务中所有的事件全部传递到channel且提交成功，那么source就将该文件标记为完成。同理，事务以类似的方式处理从channel到sink的传递过程，如果因为某种原因使得事件无法记录，那么事务将会回滚。且所有的事件都会保持到channel中，等待重新传递。

```
# example.conf: A single-node Flume configuration

# Name the components on this agent
a1.sources = r1
a1.channels = c1
a1.sinks = k1

# Describe/configure the source
a1.sources.r1.type = spooldir
a1.sources.r1.spoolDir = /home/ghb/HadoopCluster/Call
a1.sources.r1.fileHeader = true
a1.sources.r1.fileSuffix = .delete
a1.sources.r1.batchSize = 100

# Use file channel
a1.channels.c1.type = file
a1.channels.c1.checkpointDir=/home/ghb/HadoopCluster/flume-1.6.0/checkpoint
a1.channels.c1.dataDirs=/home/ghb/HadoopCluster/flume-1.6.0/dataDir

# Describe the sink
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.topic = CallLog
a1.sinks.k1.kafka.bootstrap.servers =127.0.0.1:6667
a1.sinks.k1.kafka.flumeBatchSize = 20
a1.sinks.k1.kafka.producer.acks=1
```

2、Flume的At-Least-Once提交方式

Flume的事务机制，总的来说，保证了source产生的每个事件都会传送到sink中。但是值得一说的是，实际上Flume作为高容量并行采集系统**采用的是At-least-once**（传统的企业系统采用的是exactly-once机制）提交方式，这样就造成**每个source产生的事件至少到达sink一次，换句话说就是同一事件有可能重复到达**。这样虽然看上去是一个缺陷，但是相比为了保证Flume能够可靠地将事件从source,channel传递到sink,这也是一个可以接受的权衡。如spooldir的使用，Flume会对已经处理完的数据进行标记。

3、Flume的批处理机制

为了提高效率，Flume尽可能的以事务为单位来处理事件，而不是逐一基于事件进行处理。比如上篇博客提到的spooling directory source以100行文本作为一个批次来读取（BatchSize属性来配置，类似数据库的批处理模式）。批处理的设置尤其有利于提高file channle的效率，这样整个事务只需要写入一次本地磁盘，或者调用一次fsync，速度回快很多。

4、Channel配置说明

- MemoryChannel可以实现高速的吞吐， 但是无法保证数据完整性。
- MemoryRecoverChannel在官方文档的建议上已经建义使用FileChannel来替换。FileChannel保证数据的完整性与一致性。在具体配置不现的FileChannel时，建议FileChannel设置的目录和程序日志文

公众号：旧时光大数据

件保存的目录设成不同的磁盘，以便提高效率。

Flume拦截器

问过的一些公司：转转，阿里

参考答案：

1、Flume拦截器介绍

拦截器是简单的插件式组件，设置在source和channel之间。source接收到的事件event，在写入channel之前，拦截器都可以进行转换或者删除这些事件。每个拦截器只处理同一个source接收到的事件。可以自定义拦截器。

2、Flume内置的拦截器

1）时间拦截器

flume中一个最经常使用的拦截器，该拦截器的作用是将时间戳插入到flume的事件报头中。如果不使用任何拦截器，flume接受到的只有message。

时间戳拦截器的配置：

参数	默认值	描述
type	timestamp	类型名称timestamp，也可以使用类名的全路径 org.apache.flume.interceptor.TimestampInterceptor$Builder
preserveExisting	false	如果设置为true，若事件中报头已经存在，不会替换时间戳报头的值

source连接到时间拦截器的配置：

```
a1.sources.r1.interceptors=i1
a1.sources.r1.interceptors.i1.type=timestamp
a1.sources.r1.interceptors.i1.preserveExisting=false
```

2）主机拦截器

主机拦截器插入服务器的ip地址或者主机名，agent将这些内容插入到事件的报头中。事件报头中的key使用hostHeader配置，默认是host。

主机拦截器的配置：

参数	默认值	描述
type	host	类型名称host，也可以使用类名的全路径 org.apache.flume.interceptor.HostInterceptor$Builder
hostHeader	host	事件头的key
useIP	true	如果设置为false，host键插入主机名
preserveExisting	false	如果设置为true，若事件中报头已经存在，不会替换时间戳报头的值

公众号：旧时光大数据

source连接到主机拦截器的配置：

```
a1.sources.r1.interceptors=i2
a1.sources.r1.interceptors.i2.type=host
a1.sources.r1.interceptors.i2.useIP=false
a1.sources.r1.interceptors.i2.preserveExisting=false
```

3）静态拦截器

静态拦截器的作用是将k/v插入到事件的报头中。

配置如下：

参数	默认值	描述
type	static	类型名称static，也可以使用类全路径名称org.apache.flume.interceptor.StaticInterceptor$Builder
key	key	事件头的key
value	value	key对应的value值
preserveExisting	true	如果设置为true，若事件中报头已经存在该key，不会替换value的值

source连接到静态拦截器的配置：

```
a1.sources.r1.interceptors= i3
a1.sources.r1.interceptors.static.type=static
a1.sources.r1.interceptors.static.key=logs
a1.sources.r1.interceptors.static.value=logFlume
a1.sources.r1.interceptors.static.preserveExisting=false
a1.sources.r1.interceptors= i3
a1.sources.r1.interceptors.static.type=static
a1.sources.r1.interceptors.static.key=logs
a1.sources.r1.interceptors.static.value=logFlume
a1.sources.r1.interceptors.static.preserveExisting=false
```

4）正则拦截器

在日志采集的时候，可能有一些数据是我们不需要的，这样添加过滤拦截器，可以过滤掉不需要的日志，也可以根据需要收集满足正则条件的日志。

配置如下：

参数	默认值	描述
type	REGEX_FILTER	类型名称REGEX_FILTER，也可以使用类全路径名称org.apache.flume.interceptor.RegexFilteringInterceptor$Builder
regex	.*	匹配除“\n”之外的任何个字符
excludeEvents	false	默认收集匹配到的事件。如果为true，则会删除匹配到的event，收集未匹配到的

source连接到正则过滤拦截器的配置：

公众号：旧时光大数据

```
a1.sources.r1.interceptors=i4
a1.sources.r1.interceptors.i4.type=REGEX_FILTER
a1.sources.r1.interceptors.i4.regex=(rm)|(kill)
a1.sources.r1.interceptors.i4.excludeEvents=false
```

如何监控消费型Flume的消费情况

问过的一些公司：招银网络

参考答案：

使用第三方框架Ganglia实时监控Flume

Kafka和Flume是如何对接的?

问过的一些公司：招银网络

参考答案：

我们都知道flume可以跨节点进行数据的传输，那么flume与spark streaming对接不好吗？主要是flume对接到kafka的topic，可以给多个consumer group去生成多条业务线。虽然flume中的channel selector中的副本策略也可以给多个sink传输数据，但是每个channel selector都是很消耗资源的。其次，kafka也可以起到一个消峰的作用。

flume可以通过配置kafka sink与kafka进行对接。

为什么要使用Flume进行数据采集

问过的一些公司：端点数据(2021.07)

参考答案：

Flume功能强大，自己比较熟悉

Kafka面试题

介绍下Kafka，Kafka的作用？Kafka的组件？适用场景?

可回答：1）Kafka基本原理；2）下Kafka你知道的东西；3）为什么用Kafka

问过的一些公司：阿里x4，字节x2，字节(2021.07)，快手x3，头条，腾讯x2，小米x3，蘑菇街x3，猿辅导，网易，京东x2，京东(2021.07)，小鹏汽车，祖龙娱乐，跟谁学，星环科技，大华(2021.07)，58同城(2021.08)，海康(2021.08)

参考答案：

Kafka是一种分布式、高吞吐量的分布式发布订阅消息系统，它可以处理消费者规模的网站中的所有动作流数据，主要应用于大数据实时处理领域。简单地说，Kafka就相比是一个邮箱，生产者是发送邮件的人，消费者是接收邮件的人，Kafka就是用来存东西的，只不过它提供了一些处理邮件的机制。

1、作用

公众号：旧时光大数据

1）发布和订阅消息流

2）以容错的方式记录消息流，kafka以文件的方式来存储消息流

3）可以在消息发布的时候进行处理

2、优势

高吞吐量、低延迟： kafka每秒可以处理几十万条消息，它的延迟最低只有几毫秒；

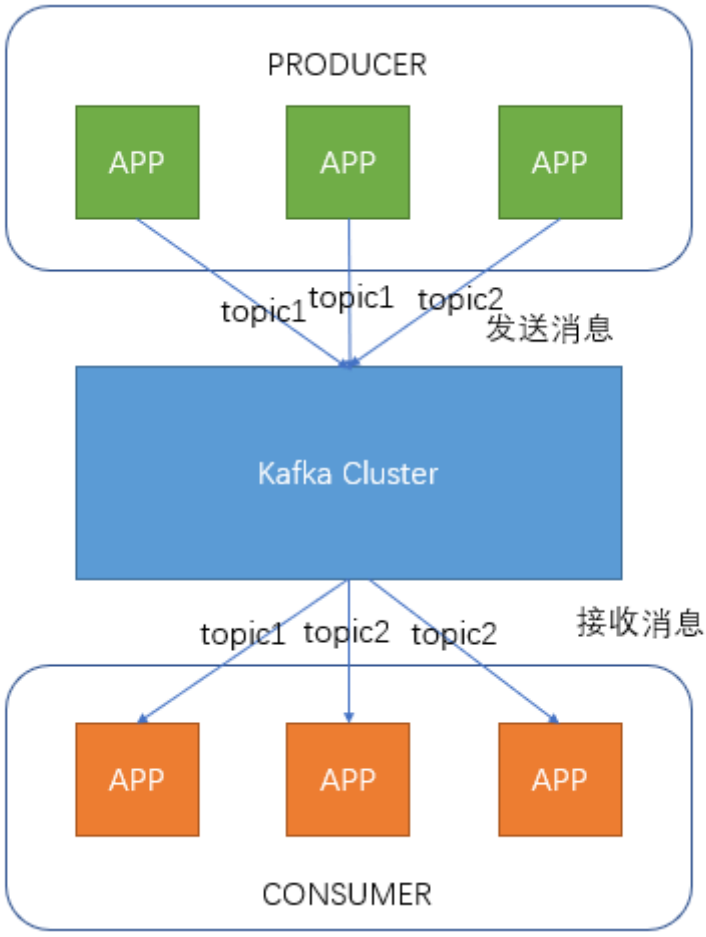
可扩展性： kafka集群支持热扩展；

持久性、可靠性： 消息被持久化到本地磁盘，并且支持数据备份防止数据丢失；

容错性： 允许集群中节点故障（若副本数量为n,则允许n-1个节点故障）；

高并发： 支持数千个客户端同时读写。

3、组件



`Topic` ：可以理解为一个队列，生产者和消费者面向的都是一个 topic；

`Producer` ：消息生产者，就是向 kafka broker 发消息的客户端；

`Consumer` ：消息消费者，向 kafka broker 取消息的客户端；

`Broker` ：一台 kafka 服务器就是一个 broker。一个集群由多个 broker 组成。一个 broker可以容纳多个 topic。

4、使用场景

1）在系统或应用程序之间构建可靠的用于传输实时数据的管道，消息队列功能

2）构建实时的流数据处理程序来变换或处理数据流，数据处理功能

通俗一点来说

公众号：旧时光大数据

日志收集：一个公司可以用Kafka可以收集各种服务的log，通过kafka以统一接口服务的方式开放给各种consumer；

消息系统：解耦生产者和消费者、缓存消息等；

用户活动跟踪：kafka经常被用来记录web用户或者app用户的各种活动，如浏览网页、搜索、点击等活动，这些活动信息被各个服务器发布到kafka的topic中，然后消费者通过订阅这些topic来做实时的监控分析，亦可保存到数据库；

运营指标：kafka也经常用来记录运营监控数据。包括收集各种分布式应用的数据，生产各种操作的集中反馈，比如报警和报告；

流式处理：比如Spark streaming和Flink。

Kafka作为消息队列，它可解决什么样的问题?

问过的一些公司：Shopee(2021.07)

参考答案：

kafka是消息中间件的一种，它可以让合适的数据以合适的形式出现在合适的地方。Kafka的做法是提供消息队列，让生产者单往队列的末尾添加数据，让多个消费者从队列里面依次读取数据然后自行处理。

Kafka主要用途是数据集成，或者说是流数据集成，以Pub/Sub形式的消息总线形式提供。但是，Kafka不仅仅是一套传统的消息总线，本质上Kafka是分布式的流数据平台，因为以下特性而著名：

- 提供Pub/Sub方式的海量消息处理。
- 以高容错的方式存储海量数据流。
- 保证数据流的顺序。

缓冲和削峰：上游数据时有突发流量，下游可能扛不住，或者下游没有足够多的机器来保证冗余，kafka在中间可以起到一个缓冲的作用，把消息暂存在kafka中，下游服务就可以按照自己的节奏进行慢慢处理。

解耦和扩展性：项目开始的时候，并不能确定具体需求。消息队列可以作为一个接口层，解耦重要的业务流程。只需要遵守约定，针对数据编程即可获取扩展能力。

冗余：可以采用一对多的方式，一个生产者发布消息，可以被多个订阅topic的服务消费到，供多个毫无关联的业务使用。

健壮性：消息队列可以堆积请求，所以消费端业务即使短时间死掉，也不会影响主要业务的正常进行。

异步通信：很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

说下Kafka架构

可回答：1）Kafka中producer，broker，cousumer的关系；2）Kafka中topic和partition和broker的关系；3）Kafka的相关结构；4）说一下Kafka的topic，partition，和副本

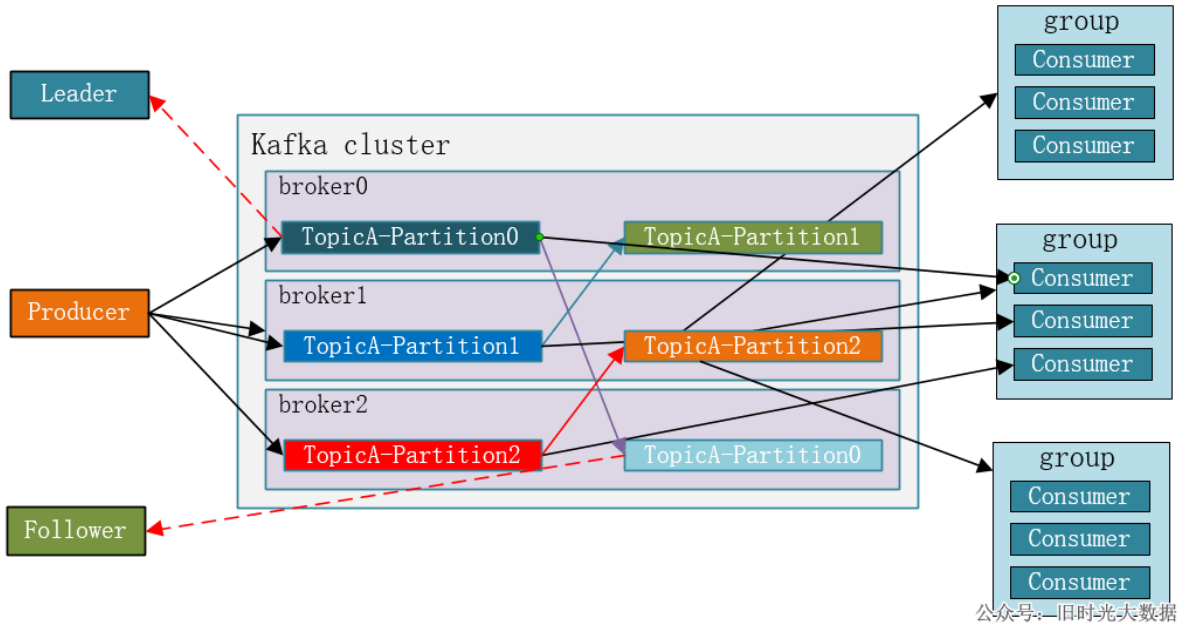
问过的一些公司：字节x2，阿里，网易x2，转转，昆仑万维，触宝(2021.07)，soul(20210.09)，京东(20210.09)

参考答案：

Kafka基础架构

公众号：旧时光大数据

- 为方便扩展，并提高吞吐量，一个topic分为多个partition
- 配合分区的设计，提出消费者组的概念，组内每个消费者并行消费
- 为提高可用性，为每个partition增加若干副本，类似NameNode HA



1）Producer：消息生产者，就是向kafka broker发消息的客户端；

2）Consumer：消息消费者，向kafka broker取消息的客户端；

3）Consumer Group（CG）：消费者组，由多个consumer组成。消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个消费者消费；消费者组之间互不影响。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。

4）Broker：一台kafka服务器就是一个broker。一个集群由多个broker组成。一个broker可以容纳多个topic。

5）Topic：可以理解为一个队列，生产者和消费者面向的都是一个topic；

6）Partition：为了实现扩展性，一个非常大的topic可以分布到多个broker（即服务器）上，一个topic可以分为多个partition，每个partition是一个有序的队列；

7）Replica：副本，为保证集群中的某个节点发生故障时，该节点上的partition数据不丢失，且kafka仍然能够继续工作，kafka提供了副本机制，一个topic的每个分区都有若干个副本，一个leader和若干个follower。

8）leader：每个分区多个副本的“主”，生产者发送数据的对象，以及消费者消费数据的对象都是leader。

9）follower：每个分区多个副本中的“从”，实时从leader中同步数据，保持和leader数据的同步。leader发生故障时，某个follower会成为新的leader。

说下Kafka的特点，优缺点

问过的一些公司：字节x2，快手，猿辅导，深信服

参考答案：

特点

- 高吞吐量、低延迟：kafka每秒可以处理几十万条消息，它的延迟最低只有几毫秒；
- 可扩展性：kafka集群支持热扩展；

公众号：旧时光大数据

- 持久性、可靠性：消息被持久化到本地磁盘，并且支持数据备份防止数据丢失；
- 容错性：允许集群中节点故障（若副本数量为n,则允许n-1个节点故障）；
- 高并发：支持数千个客户端同时读写。

优点

- 支持多个生产者和消费者
- 支持broker的横向拓展
- 副本集机制，实现数据冗余，保证数据不丢失
- 通过topic将数据进行分类
- 通过分批发送压缩数据的方式，减少数据传输开销，提高吞高量
- 支持多种模式的消息
- 基于磁盘实现数据的持久化
- 高性能的处理)信息，在大数据的情况下，可以保证亚秒级的消息延迟
- 一个消费者可以支持多种topic的消息
- 对CPU和内存的消耗比较小
- 对网络开销也比较小
- 支持跨数据中心的数据复制
- 支持镜像集群

缺点

- 由于是批量发送，所以数据达不到真正的实时
- 对于mqtt协议不支持
- 不支持物联网传感数据直接接入
- 只能支持统一分区内消息有序，无法实现全局消息有序
- 监控不完善，需要安装插件
- 需要配合zookeeper进行元数据管理
- 会丢失数据，并且不支持事务
- 可能会重复消费数据，消息会乱序，可用保证一个固定的partition内部的消息是有序的，但是一个topic有多个partition的话，就不能保证有序了，需要zookeeper的支持，topic一般需要人工创建，部署和维护一般都比mq高

四大主流MQ（kafka、ActiveMQ、RabbitMQ、RocketMQ）各自的优缺点

公众号：旧时光大数据

–	RabbitMQ	ActiveMQ	RocketMQ	Kafka
所属社区/公司	Mozilla Public License	Apache	Ali	Apache
成熟度	成熟	成熟	比较成熟	成熟
授权方式	开源	开源	开源	开源
开发语言	Erlang	Java	Java	Scala&Java
客户端支持语言	官方支持Erlang, Java, Ruby等, 社区产出多种语言API，几乎支持所有常用语言	Java、C、C++、Python、PHP、Perl、.net 等	Java C++（不成熟）	官方支持Java, 开源社区有多语言版本, 如PHP, Python, Go, C/C++, Ruby, NodeJS等编程语言, 详见Kafka 客户端列表
协议支持	多协议支持:AMQP，XMPP, SMTP, STOMP	OpenWire、STOMP、REST、XMPP、AMQP	自己定义的一套(社区提供JMS--不成熟)	自有协议，社区封装了HTTP协议支持
消息批量操作	不支持	支持	支持	支持
消息推拉模式	多协议, Pull/Push均有支持	多协议, Pull/Push均有支持	多协议, Pull/Push均有支持	Pull
HA	master/slave模式, master提供服务, slave仅作备份	基于ZooKeeper + LevelDB 的 Master-Slave 实现方式	支持多Master 模式、多 Master 多 Slave 模式，异步复制模式、多 Master 多 Slave 模式，同步双写	支持replica机制, leader宕掉后, 备份自动顶替， 并重新选举leader(基于Zookeeper)
数据可靠性	可以保证数据不丢, 有slave用作备份	master/slave	支持异步实时刷盘，同步刷盘，同步复制，异步复制	数据可靠， 并且有replica机制, 有容错容灾能力
单机吞吐量	其次(万级)	最差(万级)	最高(十万级)	次之(十万级)
消息延迟	微秒级	\	比kafka快	毫秒级
持久化能力	内存、文件，支持数据堆积,但数据堆积反过来影响生产速率	内存、文件、数据库	磁盘文件	磁盘文件, 只要磁盘容量够， 可以做到无限消息堆积
是否有序	若想有序，只能使用一个Client	可以支持有序	有序	多Client保证有序
事务	不支持	支持	支持	不支持, 但可以通过Low Level API保证仅消费一次
集群	支持	支持	支持	支持
负载均衡	支持	支持	支持	支持
管理界面	较好	一般	命令行界面	官方只提供了命令行版, Yahoo开源自己的Kafka Web管理界面Kafka-Manager
部署方式	独立	独立	独立	独立

Kafka相比于其它消息组件有什么好处?

可回答：Kafka跟其它的消息队列的对比情况

问过的一些公司：阿里，蘑菇街，美团，虎牙(2021.09)

参考答案：

分布式可扩展。Kafka集群可以透明的扩展，增加新的服务器进集群。

高性能。Kafka 的性能大大超过传统的ActiveMQ、RabbitMQ等MQ实现，尤其是Kafka还支持batch操作。

容错。Kafka每个Partition的数据都会复制到几台服务器上。当某个Broker故障失效时，ZooKeeper服务将通知生产者和消费者，生产者和消费者转而使用其它Broker。

Kafka生产者与消费者

问过的一些公司：大华(2021.07)，荣耀(2021.09)

参考答案：

公众号：旧时光大数据

生产者：Producer

即消息的发布者，生产者将数据发布到他们选择的主题。

生产者负责选择将哪个记录分配给主题中的哪个分区。即：生产者生产的一条消息，会被写入到某一个Partition。

消费者：Consumer

可以从Broker中读取消息。

Kafka的消费者，一个消费者可以消费多个Topic的消息；一个消费者可以消费同一个Topic中的多个Partition中的消息；一个Partiton允许多个Consumer同时消费。

Kafka分区容错性

问过的一些公司：大华(2021.08)

参考答案：

Kafka集群中，单个节点或多个节点出问题（主要是网络问题）后，正常服务的服务器依然能正常提供服务，并且满足设计好的一致性和可用性。

重点在于：部分节点因网络问题，业务依然能够继续运行。

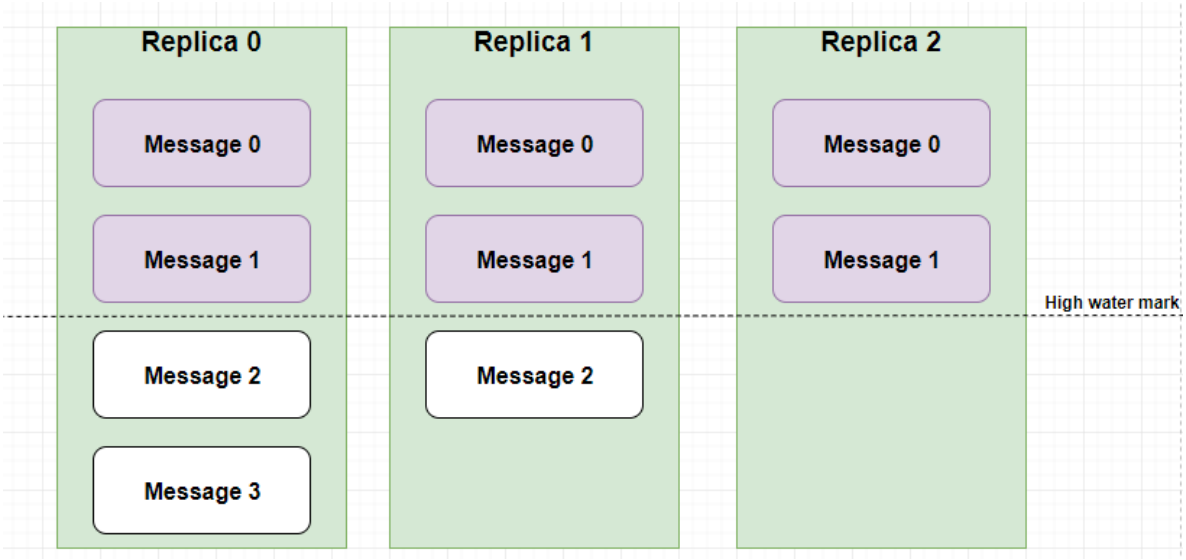
如果是整个网络都出现故障，那就没辙了。

Kafka的消费端的数据一致性

问过的一些公司：京东(2021.09)

参考答案：

这里介绍的**数据一致性主要是说不论是老的Leader还是新选举的Leader，Consumer都能读到一样的数据。**那么Kafka是如何实现的呢?



假设分区的副本为3，其中副本0是 Leader，副本1和副本2是 follower，并且在 ISR 列表里面。虽然副本0已经写入了 Message3，但是 Consumer 只能读取到 Message1。因为所有的 ISR 都同步了 Message1，只有 High Water Mark 以上的消息才支持 Consumer 读取，而 High Water Mark 取决于 ISR 列表里面偏移量最小的分区，对应于上图的副本2，这个很类似于木桶原理。

公众号：旧时光大数据

这样做的原因是还没有被足够多副本复制的消息被认为是“不安全”的，如果 Leader 发生崩溃，另一个副本成为新 Leader，那么这些消息很可能丢失了。如果我们允许消费者读取这些消息，可能就会破坏一致性。试想，一个消费者从当前 Leader（副本0） 读取并处理了 Message4，这个时候 Leader 挂掉了，选举了副本1为新的 Leader，这时候另一个消费者再去从新的 Leader 读取消息，发现这个消息其实并不存在，这就导致了数据不一致性问题。

当然，引入了 High Water Mark 机制，会导致 Broker 间的消息复制因为某些原因变慢，那么消息到达消费者的时间也会随之变长（因为我们会先等待消息复制完毕）。延迟时间可以通过参数 `replica.lag.time.max.ms` 参数配置，它指定了副本在复制消息时可被允许的最大延迟时间。

Kafka的leader挂掉之后处理方法

问过的一些公司：京东(2021.09)

参考答案：

当leader挂掉之后，会从ISR的follower中选举新的leader。

说下Kafka的ISR机制

可回答：从ISR踢出去之后呢

问过的一些公司：头条，腾讯，祖龙娱乐，虎牙(2021.09)x2

参考答案：

ISR（In-Sync Replicas）：副本同步队列

ISR是Leader维护了一个动态副本同步队列，是和leader保持同步的follower集合，ISR中包括Leader和Follower。当ISR中的follower完成数据的同步之后，leader就会给producer发送ack。如果follower长时间未向leader同步数据，则该follower将被踢出ISR，加入到OSR（Out-Sync Relipcas），该时间阈值由replica.lag.time.max.ms参数设定。Leader发生故障之后，就会从ISR中选举新的leader。

如果OSR集合中follower副本“追上”了Leader，再加入到ISR中，当Leader发生故障是，只有在ISR集合中的副本才有资格被选举为leader，而在OSR集合中的副本则没有机会。

Kafka的选举机制

可回答：Kafka的leader选举

问过的一些公司：阿里，顺丰x2，端点数据(2021.07)

参考答案：

在整个系统中，涉及到多出选举，下面从以下三个方面来说明下。

1、控制器（Broker）选举

所谓控制器就是一个Borker，在一个kafka集群中，有多个broker节点，但是它们之间需要选举出一个leader，其他的broker充当follower角色。集群中第一个启动的broker会通过在zookeeper中创建临时节点 `/controller` 来让自己成为控制器，其他broker启动时也会在zookeeper中创建临时节点，但是发现节点已经存在，所以它们会收到一个异常，意识到控制器已经存在，那么就会在zookeeper中创建watch对象，便于它们收到控制器变更的通知。

公众号：旧时光大数据

那么如果控制器由于网络原因与zookeeper断开连接或者异常退出，那么其他broker通过watch收到控制器变更的通知，就会去尝试创建临时节点 `/controller` ，如果有一个broker创建成功，那么其他broker就会收到创建异常通知，也就意味着集群中已经有了控制器，其他broker只需创建watch对象即可。

如果集群中有一个broker发生异常退出了，那么控制器就会检查这个broker是否有分区的副本leader，如果有那么这个分区就需要一个新的leader，此时控制器就会去遍历其他副本，决定哪一个成为新的leader，同时更新分区的ISR集合。

如果有一个broker加入集群中，那么控制器就会通过Broker ID去判断新加入的broker中是否含有现有分区的副本，如果有，就会从分区副本中去同步数据。

集群中每选举一次控制器，就会通过zookeeper创建一个 `controller epoch` ，每一个选举都会创建一个更大，包含最新信息的 `epoch` ，如果有broker收到比这个 `epoch` 旧的数据，就会忽略它们，kafka也通过这个 `epoch` 来防止集群产生“脑裂”。

2、分区副本选举机制

在kafka的集群中，会存在着多个主题topic，在每一个topic中，又被划分为多个partition，为了防止数据不丢失，每一个partition又有多个副本，在整个集群中，总共有三种副本角色：

- leader副本：也就是leader主副本，每个分区都有一个leader副本，为了保证数据一致性，所有的生产者与消费者的请求都会经过该副本来处理。
- follower副本：除了首领副本外的其他所有副本都是follower副本，follower副本不处理来自客户端的任何请求，只负责从leader副本同步数据，保证与首领保持一致。如果leader副本发生崩溃，就会从这其中选举出一个leader。
- 优先副本：创建分区时指定的优先leader。如果不指定，则为分区的第一个副本。

follower需要从leader中同步数据，但是由于网络或者其他原因，导致数据阻塞，出现不一致的情况，为了避免这种情况，follower会向leader发送请求信息，这些请求信息中包含了follower需要数据的偏移量offset，而且这些offset是有序的。

如果有follower向leader发送了请求1，接着发送请求2，请求3，那么再发送请求4，这时就意味着follower已经同步了前三条数据，否则不会发送请求4。leader通过跟踪 每一个follower的offset来判断它们的复制进度。

默认的，如果follower与leader之间超过10s内没有发送请求，或者说没有收到请求数据，此时该follower就会被认为“不同步副本”。而持续请求的副本就是“同步副本”，当leader发生故障时，只有“同步副本”才可以被选举为leader。其中的请求超时时间可以通过参数 `replica.lag.time.max.ms` 参数来配置。

我们希望每个分区的leader可以分布到不同的broker中，尽可能的达到负载均衡，所以会有一个优先leader，如果我们设置参数 `auto.leader.rebalance.enable` 为true，那么它会检查优先leader是否是真正的leader，如果不是，则会触发选举，让优先leader成为leader。

3、消费者选主

在kafka的消费端，会有一个消费者协调器以及消费组，组协调器 `GroupCoordinator` 需要为消费组内的消费者选举出一个消费组的leader，那么如何选举的呢？

如果消费组内还没有leader，那么第一个加入消费组的消费者即为消费组的leader，如果某一个时刻leader消费者由于某些原因退出了消费组，那么就会重新选举leader，如何选举？

```
1    private val members = new mutable.HashMap[String, MemberMetadata]
2    leaderId = members.keys.headOption
```

上面代码是kafka源码中的部分代码，member是一个hashmap的数据结构，key为消费者的 `member_id` ，value是元数据信息，那么它会将leaderId选举为Hashmap中的第一个键值对，它和随机基本没啥区别。

公众号：旧时光大数据

Kafka的ISR、OSR和ACK介绍，ACK分别有几种值?

问过的一些公司：快手，ebay

参考答案：

ISR（In-Sync Replicas）：副本同步队列

ISR是Leader维护了一个动态副本同步队列，是和leader保持同步的follower集合。

OSR（Out-Sync Relipcas）：

当ISR中的follower长时间（replica.lag.time.max.ms参数设定）未向leader同步数据，则该follower将被踢出ISR，加入到OSR。

ACK：producer的消息发送确认机制

ack=0

producer不等待broker的ack，这一操作提供了一个最低的延迟，broker一接收到还没有写入磁盘就已经返回，当broker故障时有可能丢失数据

ack=1

producer等待broker的ack，partition的leader落盘成功后返回ack，如果在follower同步成功之前leader故障，那么将会丢失数据；

ack=-1

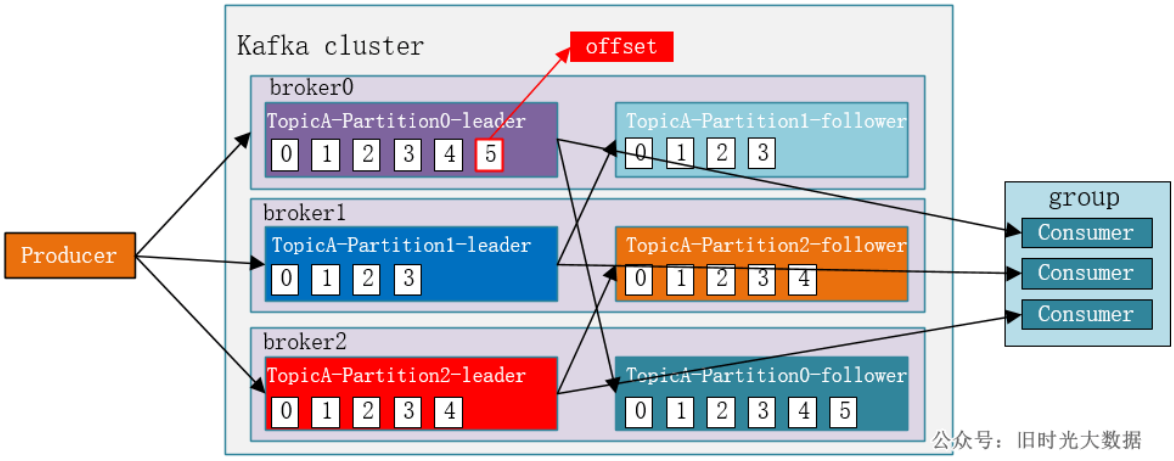
producer等待broker的ack，partition的leader和follower全部落盘成功后才返回ack。但是如果在follower同步完成后，broker发送ack之前，leader发生故障，那么会造成数据重复。

Kafka的工作原理?

可回答：Kafka的工作流程

问过的一些公司：阿里，字节社招，字节(2022.03)，b站，七牛云，创略科技，网易

先来看个简单版本的



Kafka中消息是以topic进行分类的，生产者生产消息，消费者消费消息，都是面向topic的。

topic是逻辑上的概念，而partition是物理上的概念，每个partition对应于一个log文件，该log文件中存储的就是producer生产的数据。Producer生产的数据会被不断追加到该log文件末端，且每条数据都有自己的offset。消费者组中的每个消费者，都会实时记录自己消费到了哪个offset，以便出错恢复时，从上次的位置继续消费。

公众号：旧时光大数据

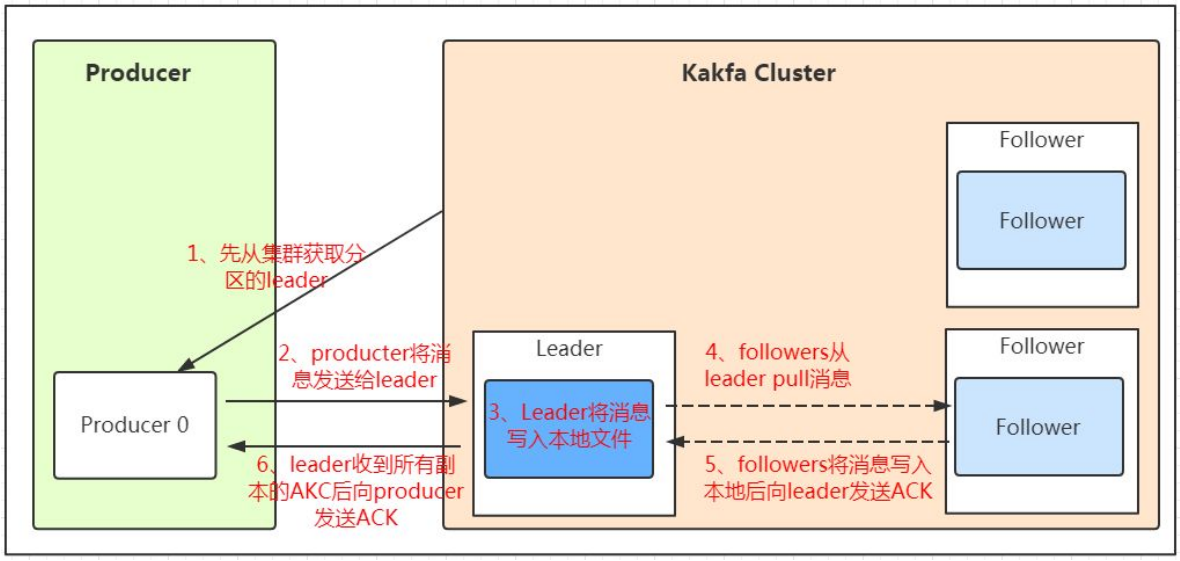
再来看个详细版本的

关于Kafka的架构就不多说了，前面有相关介绍。

1、发送数据

Producer是生产者，是数据的入口。注意看图中的红色箭头，Producer 在写入数据的时候永远在找 Leader，不会直接将数据写入 Follower！

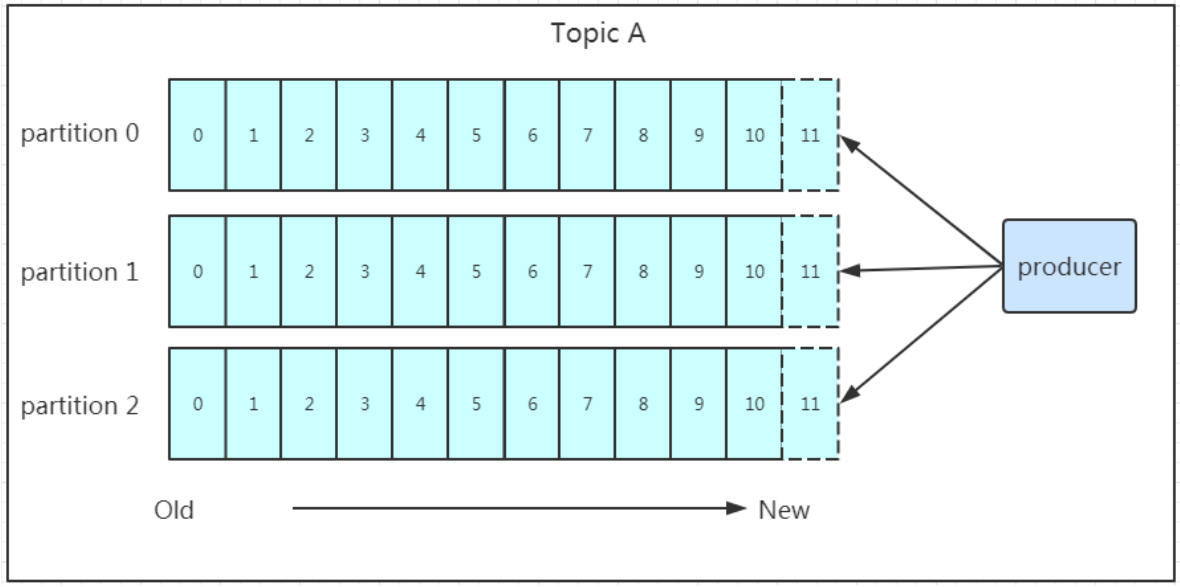
关于Leader的寻找以及写入流程可以参考下图：



消息写入Leader后，Follower是主动的去找Leader进行同步的！

Producer 采用 Push 模式将数据发布到 Broker，每条消息追加到分区中，顺序写入磁盘，所以保证同一分区内的数据是有序的！

写入示意图如下：



上面说到数据会写入到不同的分区，那 Kafka 为什么要做分区呢？相信大家应该也能猜到，分区的主要目的是：

- 方便扩展。因为一个 Topic 可以有多个 Partition，所以我们可以通过扩展机器去轻松的应对日益增长的数据量。
- 提高并发。以 Partition 为读写单位，可以多个消费者同时消费数据，提高了消息的处理效率。

熟悉负载均衡的应该知道，当我们向某个服务器发送请求的时候，服务端可能会对请求做一个负载，将流量分发到不同的服务器。

公众号：旧时光大数据

那在 Kafka 中，如果某个 Topic 有多个 Partition，Producer 又怎么知道该将数据发往哪个 Partition 呢？

Kafka 中有几个原则：

- Partition 在写入的时候可以指定需要写入的 Partition，如果有指定，则写入对应的 Partition。
- 如果没有指定 Partition，但是设置了数据的 Key，则会根据 Key 的值 Hash 出一个 Partition。
- 如果既没指定 Partition，又没有设置 Key，则会轮询选出一个 Partition。

保证消息不丢失是一个消息队列中间件的基本保证，那 Producer 在向 Kafka 写入消息的时候，怎么保证消息不丢失呢？

其实上面的写入流程图中有描述出来，那就是通过 ACK 应答机制！在生产者向队列写入数据的时候可以设置参数来确定是否确认 Kafka 接收到数据，这个参数可设置的值为 0、1、all：

- 0 代表 Producer 往集群发送数据不需要等到集群的返回，不确保消息发送成功。安全性最低但是效率最高。
- 1 代表 Producer 往集群发送数据只要 Leader 应答就可以发送下一条，只确保 Leader 发送成功。
- all 代表 Producer 往集群发送数据需要所有的 Follower 都完成从 Leader 的同步才会发送下一条，确保 Leader 发送成功和所有的副本都完成备份。安全性最高，但是效率最低。

最后要注意的是，如果往不存在的 Topic 写数据，能不能写入成功呢？Kafka 会自动创建 Topic，分区和副本的数量根据默认配置都是 1。

2、保存数据

Producer 将数据写入 Kafka 后，集群就需要对数据进行保存了！Kafka 将数据保存在磁盘，可能在我们的一般的认知里，写入磁盘是比较耗时的操作，不适合这种高并发的组件。Kafka 初始会单独开辟一块磁盘空间，顺序写入数据（效率比随机写入高）。

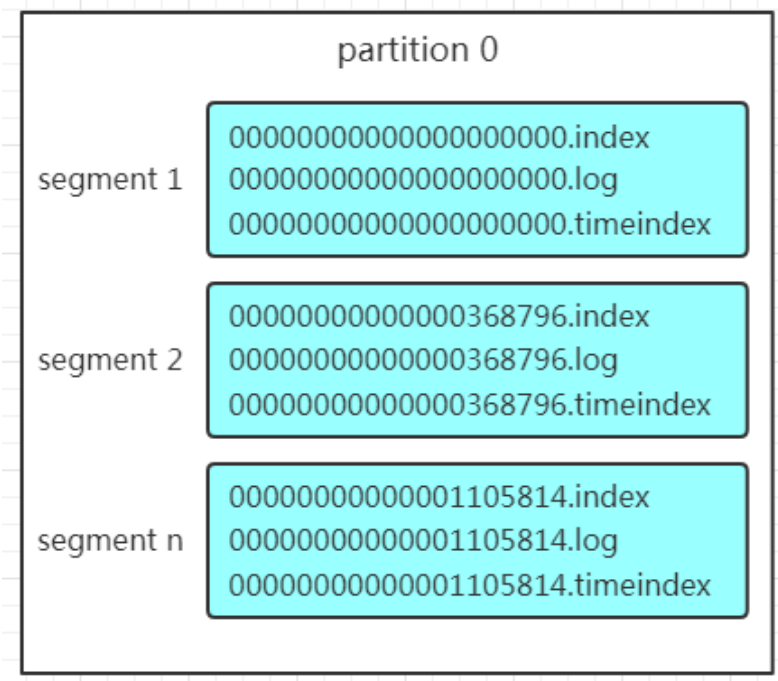
1）Partition结构

前面说过了每个 Topic 都可以分为一个或多个 Partition，如果你觉得 Topic 比较抽象，那 Partition 就是比较具体的东西了！

Partition 在服务器上的表现形式就是一个一个的文件夹，每个 Partition 的文件夹下面会有多组 Segment 文件。

每组 Segment 文件又包含 .index 文件、.log 文件、.timeindex 文件（早期版本中没有）三个文件。

Log 文件就是实际存储 Message 的地方，而 Index 和 Timeindex 文件为索引文件，用于检索消息。



公众号：旧时光大数据

如上图，这个 Partition 有三组 Segment 文件，每个 Log 文件的大小是一样的，但是存储的 Message 数量是不一定相等的（每条的 Message 大小不一致）。

文件的命名是以该 Segment 最小 Offset 来命名的，如 000.index 存储 Offset 为 0~368795 的消息，Kafka 就是利用分段+索引的方式来解决查找效率的问题。

2）Message结构

上面说到 Log 文件就实际是存储 Message 的地方，我们在 Producer 往 Kafka 写入的也是一条一条的 Message。

那存储在 Log 中的 Message 是什么样子的呢？消息主要包含消息体、消息大小、Offset、压缩类型……等等！

我们重点需要知道的是下面三个：

- Offset：Offset 是一个占 8byte 的有序 id 号，它可以唯一确定每条消息在 Parition 内的位置！
- 消息大小：消息大小占用 4byte，用于描述消息的大小。
- 消息体：消息体存放的是实际的消息数据（被压缩过），占用的空间根据具体的消息而不一样。

3）存储策略

无论消息是否被消费，Kafka 都会保存所有的消息。那对于旧数据有什么删除策略呢？

- 基于时间，默认配置是 168 小时（7 天）。
- 基于大小，默认配置是 1073741824。

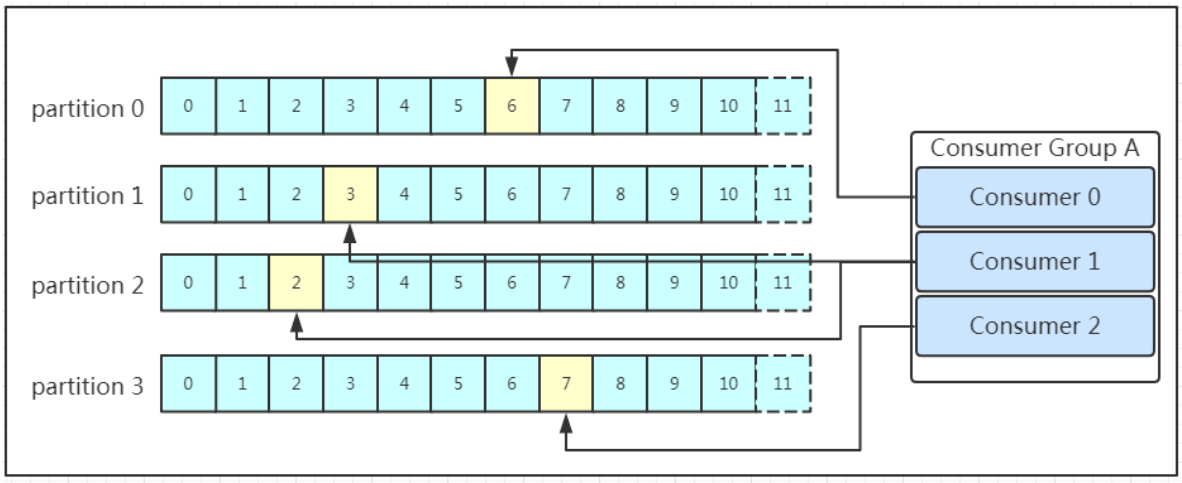
需要注意的是，Kafka 读取特定消息的时间复杂度是 O(1)，所以这里删除过期的文件并不会提高 Kafka 的性能！

3、消费数据

消息存储在 Log 文件后，消费者就可以进行消费了。Kafka 采用的是点对点的模式，消费者主动的去 Kafka 集群拉取消息，与 Producer 相同的是，消费者在拉取消息的时候也是找 Leader 去拉取。多个消费者可以组成一个消费者组（Consumer Group），每个消费者组都有一个组 id！

同一个消费组者的消费者可以消费同一 Topic 下不同分区的数据，但是不会组内多个消费者消费同一分区的数据！

详情如下图：



图示是消费者组内的消费者小于 Partition 数量的情况，所以会出现某个消费者消费多个 Partition 数据的情况，消费的速度也就不及只处理一个 Partition 的消费者的处理速度！

如果是消费者组的消费者多于 Partition 的数量，那会不会出现多个消费者消费同一个 Partition 的数据呢？

公众号：旧时光大数据

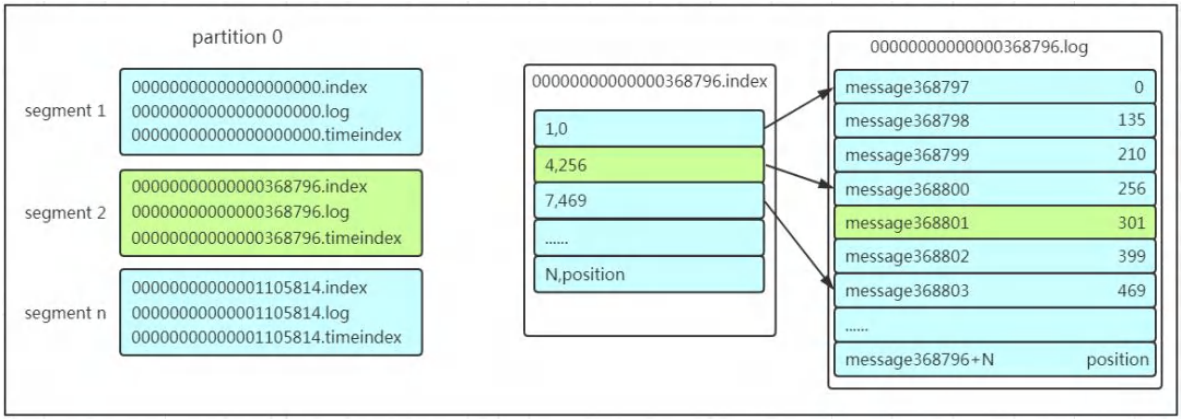
上面已经提到过不会出现这种情况！多出来的消费者不消费任何 Partition 的数据。

所以在实际的应用中，建议消费者组的 Consumer 的数量与 Partition 的数量一致！

在保存数据的小节里面，我们聊到了 Partition 划分为多组 Segment，每个 Segment 又包含.log、.index、.timeindex 文件，存放的每条 Message 包含 Offset、消息大小、消息体……

前面多次提到 Segment 和 Offset，查找消息的时候是怎么利用 Segment+Offset 配合查找的呢?

假如现在需要查找一个 Offset 为 368801 的 Message 是什么样的过程呢？我们先看看下面的图：



① 先找到 Offset 的 368801message 所在的 Segment 文件（利用二分法查找），这里找到的就是在第二个 Segment 文件。

② 打开找到的 Segment 中的 .index 文件（也就是 368796.index 文件，该文件起始偏移量为 368796+1。

我们要查找的 Offset 为 368801 的 Message 在该 Index 内的偏移量为 368796+5=368801，所以这里要查找的相对 Offset 为 5）。

由于该文件采用的是稀疏索引的方式存储着相对 Offset 及对应 Message 物理偏移量的关系，所以直接找相对 Offset 为 5 的索引找不到。

这里同样利用二分法查找相对 Offset 小于或者等于指定的相对 Offset 的索引条目中最大的那个相对 Offset，所以找到的是相对 Offset 为 4 的这个索引。

③根据找到的相对 Offset 为 4 的索引确定 Message 存储的物理偏移位置为 256。

打开数据文件，从位置为 256 的那个地方开始顺序扫描直到找到 Offset 为 368801 的那条 Message。

这套机制是建立在 Offset 为有序的基础上，利用 Segment+有序 Offset+稀疏索引+二分查找+顺序查找等多种手段来高效的查找数据！

至此，消费者就能拿到需要处理的数据进行处理了。那**每个消费者又是怎么记录自己消费的位置**呢?

在早期的版本中，消费者将消费到的 Offset 维护在 Zookeeper 中，Consumer 每间隔一段时间上报一次，这里容易导致重复消费，且性能不好！

在新的版本中消费者消费到的 Offset 已经直接维护在 Kafka 集群的 __consumer_offsets 这个 Topic 中！

Kafka怎么保证数据不丢失，不重复?

可回答：Kafka如何保证生产者不丢失数据，消费者不丢失数据?

问过的一些公司：字节跳动 x 2，快手x2，严选阿里，祖龙娱乐x2，招银网络，ebay，安恒信息

参考答案：

存在数据丢失的几种情况

公众号：旧时光大数据

- 使用同步模式的时候，有3种状态保证消息被安全生产，在配置为1（只保证写入leader成功）的话，如果刚好leader partition挂了，数据就会丢失。
- 还有一种情况可能会丢失消息，就是使用异步模式的时候，当缓冲区满了，如果配置为0（还没有收到确认的情况下，缓冲池一满，就清空缓冲池里的消息），数据就会被立即丢弃掉。

避免方法的一些概述

1、在数据生产时避免数据丢失的方法

只要能避免上述两种情况，那么就可以保证消息不会被丢失。

1）在同步模式的时候，确认机制设置为-1，也就是让消息写入leader和所有的副本。

2）在异步模式下，如果消息发出去了，但还没有收到确认的时候，缓冲池满了，在配置文件中设置成不限制阻塞超时的时间，也就说让生产端一直阻塞，这样也能保证数据不会丢失。

在数据消费时，避免数据丢失的方法：如果使用了storm，要开启storm的ackfail机制；如果没有使用storm，确认数据被完成处理之后，再更新offset值。低级API中需要手动控制offset值。

消息队列的问题都要从源头找问题，就是生产者是否有问题。

讨论一种情况，如果数据发送成功，但是接受response的时候丢失了，机器重启之后就会重发。

重发很好解决，消费端增加去重表就能解决，但是如果生产者丢失了数据，问题就很麻烦了。

2、数据重复消费的情况，处理情况如下

1）去重：将消息的唯一标识保存到外部介质中，每次消费处理时判断是否处理过；

2）不管：大数据场景中，报表系统或者日志信息丢失几条都无所谓，不会影响最终的统计分析结。

Kafka到底会不会丢数据(data loss)? 通常不会，但有些情况下的确有可能会发生。下面的参数配置及Best practice列表可以较好地保证数据的持久性(当然是trade-off，牺牲了吞吐量)。

如果想要高吞吐量就要能容忍偶尔的失败（重发漏发无顺序保证）。

```
block.on.buffer.full = true
acks = all
retries = MAX_VALUE
max.in.flight.requests.per.connection = 1
使用KafkaProducer.send(record, callback)
callback逻辑中显式关闭producer：close(0)
unclean.leader.election.enable=false
replication.factor = 3
min.insync.replicas = 2
replication.factor > min.insync.replicas
enable.auto.commit=false
```

消息处理完成之后再提交位移

给出列表之后，我们从两个方面来探讨一下数据为什么会丢失：

`Producer端`

新版本的Kafka替换了Scala版本的old producer，使用了由Java重写的producer。新版本的producer采用异步发送机制。KafkaProducer.send(ProducerRecord)方法仅仅是把这条消息放入一个缓存中(即RecordAccumulator，本质上使用了队列来缓存记录)，同时后台的IO线程会不断扫描该缓存区，将满足条件的消息封装到某个batch中然后发送出去。显然，**这个过程中就有一个数据丢失的窗口**：若IO线程发送之前client端挂掉了，累积在accumulator中的数据的确有可能会丢失。

Producer的另一个问题是**消息的乱序问题**。假设客户端代码依次执行下面的语句将两条消息发到相同的分区

```
producer.send(record1);
producer.send(record2);
```

如果此时由于某些原因(比如瞬时的网络抖动)导致record1没有成功发送，同时Kafka又配置了重试机制和max.in.flight.requests.per.connection大于1(默认值是5，本来就是大于1的)，那么重试record1成功后，record1在分区中就在record2之后，从而造成消息的乱序。很多某些要求强顺序保证的场景是不允许出现这种情况的。发送之后重发就会丢失顺序。

鉴于producer的这两个问题，我们应该如何规避呢？？对于消息丢失的问题，很容易想到的一个方案就是：既然异步发送有可能丢失数据， 我改成同步发送总可以吧？比如这样：

```
producer.send(record).get();
```

这样当然是可以的，但是性能会很差，不建议这样使用。以下的配置清单应该能够比较好地规避producer端数据丢失情况的发生：(特此说明一下，软件配置的很多决策都是trade-off，下面的配置也不例外：应用了这些配置，你可能会发现你的producer/consumer 吞吐量会下降，这是正常的，因为你换取了更高的数据安全性)。

block.on.buffer.full = true 尽管该参数在0.9.0.0已经被标记为“deprecated”，但鉴于它的含义非常直观，所以这里还是显式设置它为true，使得producer将一直等待缓冲区直至其变为可用。否则如果producer生产速度过快耗尽了缓冲区，producer将抛出异常。缓冲区满了就阻塞在那，不要抛异常，也不要丢失数据。

acks=all 很好理解，所有follower都响应了才认为消息提交成功，即"committed"。

retries = MAX 无限重试，直到你意识到出现了问题。

max.in.flight.requests.per.connection = 1 限制客户端在单个连接上能够发送的未响应请求的个数。设置此值是1表示kafka broker在响应请求之前client不能再向同一个broker发送请求。注意：设置此参数是为了避免消息乱序。

使用KafkaProducer.send(record, callback)而不是send(record)方法 自定义回调逻辑处理消息发送失败，比如记录在日志中，用定时脚本扫描重处理。

callback逻辑中最好显式关闭producer：close(0) 注意：设置此参数是为了避免消息乱序（仅仅因为一条消息发送没收到反馈就关闭生产者，感觉代价很大）。

unclean.leader.election.enable=false 关闭unclean leader选举，即不允许非ISR中的副本被选举为leader，以避免数据丢失。
replication.factor >= 3 参考了Hadoop及业界通用的三备份原则。

min.insync.replicas > 1 消息至少要被写入到这么多副本才算成功，也是提升数据持久性的一个参数。与acks配合使用保证replication.factor > min.insync.replicas 如果两者相等，当一个副本挂掉了分区也就没法正常工作了。通常设置replication.factor = min.insync.replicas + 1即可。

`Consumer端`

consumer端丢失消息的情形比较简单：如果在消息处理完成前就提交了offset，那么就有可能造成数据的丢失。由于Kafka consumer默认是自动提交位移的，所以在后台提交位移前一定要保证消息被正常处理了，因此不建议采用很重的处理逻辑，如果处理耗时很长，则建议把逻辑放到另一个线程中去做。为了避免数据丢失，现给出两点建议：

- enable.auto.commit=false 关闭自动提交位移
- 在消息被完整处理之后再手动提交位移

Kafka分区策略

公众号：旧时光大数据

问过的一些公司：快手，Shopee(2021.08)

参考答案：

对于消费者来说，一个consumer group中有多个consumer，一个 topic有多个partition，所以肯定会涉及到partition的分配问题，即确定每个partition由哪个consumer来消费，这就是分区分配策略（Partition Assignment Strategy）。

Kafka有三种分配策略，一是roundrobin，一是range。最新还有一个StickyAssignor策略。

1、分配分区的前提条件

首先kafka设定了默认的消费逻辑：一个分区只能被同一个消费组（ConsumerGroup）内的一个消费者消费。

在这个消费逻辑设定下，假设目前某消费组内只有一个消费者C0，订阅了一个topic，这个topic包含6个分区，也就是说这个消费者C0订阅了6个分区，这时候可能会发生下列三种情况：

- 如果这时候消费者组内**新增**了一个**消费者**C1，这个时候就需要把之前分配给C0的6个分区拿出来3个分配给C1；
- 如果这时候这个topic**多了一些分区**，就要按照某种策略，把多出来的分区分配给C0和C1；
- 如果这时候C1**消费者挂掉了或者退出**了，不在消费者组里了，那所有的分区需要再次分配给C0。

这三种情况其实就是kafka进行分区分配的前提条件：

- 同一个 Consumer Group 内新增消费者；
- 订阅的主题新增分区；
- 消费者离开当前所属的Consumer Group，包括shuts down 或 crashes。

只有满足了这三个条件的任意一个，才会进行分区分配 。**分区的所有权从一个消费者移到另一个消费者称为重新平衡（rebalance），如何rebalance就涉及到本节提到的分区分配策略。kafka提供了消费者客户端参数partition.assignment.strategy用来设置消费者与订阅主题之间的分区分配策略。默认情况下，此参数的值为：org.apache.kafka.clients.consumer.RangeAssignor，即采用range分配策略。除此之外，Kafka中还提供了roundrobin分配策略和sticky分区分配策略。消费者客户端参数partition.asssignment.strategy可以配置多个分配策略，把它们以逗号分隔就可以了。**

2、Range分配策略

Range分配策略是**面向每个主题**的，首先会对同一个主题里面的分区按照序号进行排序，并把消费者线程按照字母顺序进行排序。然后用分区数除以消费者线程数量来判断每个消费者线程消费几个分区。如果除不尽，那么前面几个消费者线程将会多消费一个分区。

我们假设有个名为T1的主题，包含了7个分区，它有两个消费者（C0和C1），其中C0的num.streams(消费者线程) = 1，C1的num.streams = 2。排序后的分区是：0，1，2，3，4，5，6；消费者线程排序后是：C0-0，C1-0，C1-1；一共有7个分区，3个消费者线程，进行计算7/3=2…1，商为2余数为1，则每个消费者线程消费2个分区，并且前面1个消费者线程多消费一个分区，结果会是这样的：

消费者线程	对应消费的分区序号
C0-0	0，1，2
C1-0	3，4
C1-1	5，6

这样看好像还没什么问题，但是一般在咱们实际生产环境下，会有多个主题，我们假设有3个主题（T1，T2，T3），都有7个分区，那么按照咱们上面这种Range分配策略分配后的消费结果如下：

公众号：旧时光大数据

消费者线程	对应消费的分区序号
C0-0	T1（0，1，2），T2（0，1，2），T3（0，1，2）
C1-0	T1（3，4），T2（3，4），T3（3，4）
C1-1	T1（5，6），T2（5，6），T3（5，6）

我们可以发现，在这种情况下，C0-0消费线程要多消费3个分区，这显然是不合理的，其实这就是Range分区分配策略的缺点。

3、RoundRobin分配策略

RoundRobin策略的原理是将消费组内所有消费者以及消费者所订阅的所有topic的partition按照字典序排序，然后通过轮询算法逐个将分区以此分配给每个消费者。

使用RoundRobin分配策略时会出现两种情况：

- 如果同一消费组内，所有的消费者订阅的消息都是相同的，那么 RoundRobin 策略的分区分配会是均匀的。
- 如果同一消费者组内，所订阅的消息是不相同的，那么在执行分区分配的时候，就不是完全的轮询分配，有可能会导致分区分配的不均匀。如果某个消费者没有订阅消费组内的某个 topic，那么在分配分区的时候，此消费者将不会分配到这个 topic 的任何分区。

分别举例说明：

第一种：比如我们有3个消费者（C0，C1，C2），都订阅了2个主题（T0 和 T1）并且每个主题都有 3 个分区(p0、p1、p2)，那么所订阅的所有分区可以标识为T0p0、T0p1、T0p2、T1p0、T1p1、T1p2。此时使用RoundRobin分配策略后，得到的分区分配结果如下：

消费者线程	对应消费的分区序号
C0	T0p0、T1p0
C1	T0p1、T1p1
C2	T0p2、T1p2

可以看到，这时候的分区分配策略是比较平均的。

第二种：比如我们依然有3个消费者（C0，C1，C2），他们合在一起订阅了 3 个主题：T0、T1 和 T2（C0订阅的是主题T0，消费者C1订阅的是主题T0和T1，消费者C2订阅的是主题T0、T1和T2），这 3 个主题分别有 1、2、3 个分区(即:T0有1个分区(p0)，T1有2个分区(p0、p1)，T2有3个分区(p0、p1、p2))，即整个消费者所订阅的所有分区可以标识为 T0p0、T1p0、T1p1、T2p0、T2p1、T2p2。此时如果使用RoundRobin分配策略，得到的分区分配结果如下：

消费者线程	对应消费的分区序号
C0	T0p0
C1	T1p0
C2	T1p1、T2p0、T2p1、T2p2

这时候显然分配是不均匀的，因此在使用RoundRobin分配策略时，为了保证得均匀的分区分配结果，需要满足两个条件：

- 同一个消费者组里的每个消费者订阅的主题必须相同；

公众号：旧时光大数据

- 同一个消费者组里面的所有消费者的num.streams必须相等。

如果无法满足，那最好不要使用RoundRobin分配策略。

4、Sticky分配策略

Sticky分配策略，这种分配策略是在kafka的0.11.X版本才开始引入的，是目前最复杂也是最优秀的分配策略。

Sticky分配策略的原理比较复杂，它的设计主要实现了两个目的：

- 分区的分配要尽可能的均匀；
- 分区的分配尽可能的与上次分配的保持相同。

如果这两个目的发生了冲突，优先实现第一个目的。

我们举例进行分析：比如我们有3个消费者（C0，C1，C2），都订阅了2个主题（T0 和 T1）并且每个主题都有 3 个分区(p0、p1、p2)，那么所订阅的所有分区可以标识为T0p0、T0p1、T0p2、T1p0、T1p1、T1p2。此时使用Sticky分配策略后，得到的分区分配结果如下：

消费者线程	对应消费的分区序号
C0	T0p0、T1p0
C1	T0p1、T1p1
C2	T0p2、T1p2

这里我们可以发现，情况怎么和前面RoundRobin分配策略一样，其实底层实现并不一样。这里假设C2故障退出了消费者组，然后需要对分区进行再平衡操作，如果使用的是RoundRobin分配策略，它会按照消费者C0和C1进行重新轮询分配，再平衡后的结果如下：

消费者线程	对应消费的分区序号
C0	T0p0、T0p2、T1p1
C1	T0p1、T1p0、T1p2

但是如果使用的是Sticky分配策略，再平衡后的结果会是这样：

消费者线程	对应消费的分区序号
C0	T0p0、T1p0、T0p2
C1	T0p1、T1p1、T1p2

Stiky分配策略保留了再平衡之前的消费分配结果，并将原来消费者C2的分配结果分配给了剩余的两个消费者C0和C1，最终C0和C1的分配还保持了均衡。这时候再体会一下sticky（翻译为：粘粘的）这个词汇的意思，是不是豁然开朗了。

为什么要这么处理呢?

这是因为发生分区重分配后，对于同一个分区而言有可能之前的消费者和新指派的消费者不是同一个，对于之前消费者进行到一半的处理还要在新指派的消费者中再次处理一遍，这时就会浪费系统资源。而使用Sticky策略就可以让分配策略具备一定的“粘性”，尽可能地让前后两次分配相同，进而可以减少系统资源的损耗以及其它异常情况的发生。

接下来，再来看一下上一节RoundRobin存在缺陷的地方，这种情况下sticky是怎么分配的?

公众号：旧时光大数据

比如我们依然有3个消费者（C0，C1，C2），他们合在一起订阅了 3 个主题：T0、T1 和 T2（C0订阅的是主题T0，消费者C1订阅的是主题T0和T1，消费者C2订阅的是主题T0、T1和T2），这 3 个主题分别有 1、2、3 个分区(即:T0有1个分区(p0)，T1有2个分区(p0、p1)，T2有3个分区(p0、p1、p2))，即整个消费者所订阅的所有分区可以标识为 T0p0、T1p0、T1p1、T2p0、T2p1、T2p2。此时如果使用sticky分配策略，得到的分区分配结果如下：

消费者线程	对应消费的分区序号
C0	T0p0
C1	T1p0、T1p1
C2	T2p0、T2p1、T2p2

由于C0消费者没有订阅T1和T2主题，因此如上这样的分配策略已经是这个问题的最优解了！

这时候，再补充一个例子，加入C0挂了，发生再平衡后的分配结果，RoundRobin和Sticky又有什么区别呢？

RoundRobin再平衡后的分配情况：

消费者线程	对应消费的分区序号
C1	T0p0、T1p1
C2	T1p0、T2p0、T2p1、T2p2

而如果使用Sticky策略，再平衡后分分配情况：

消费者线程	对应消费的分区序号
C1	T1p0、T1p1、T0p0
C2	T2p0、T2p1、T2p2

这里我们惊奇的发现sticky只是把之前C0消耗的T0p0分配给了C1，我们结合资源消耗来看，这相比RoundRobin能节省更多的资源。

综上所述，建议使用sticky分区分配策略。

Kafka如何尽可能保证数据可靠性?

可回答：Kafka如何保证消费者消费数据的可靠性

问过的一些公司：星环科技，多益，ebay，招银网络x2，荣耀(2021.09)

参考答案：

1、Topic分区副本

在 Kafka 0.8.0 之前，Kafka 是没有副本的概念的，那时候人们只会用 Kafka 存储一些不重要的数据，因为没有副本，数据很可能会丢失。但是随着业务的发展，支持副本的功能越来越强烈，所以为了保证数据的可靠性，Kafka 从 0.8.0 版本开始引入了分区副本（详情请参见 KAFKA-50）。也就是说每个分区可以人为的配置几个副本（比如创建主题的时候指定 replication-factor，也可以在 Broker 级别进行配置 default.replication.factor），一般会设置为3。

公众号：旧时光大数据

Kafka 可以保证单个分区里的事件是有序的，分区可以在线（可用），也可以离线（不可用）。在众多的分区副本里面有一个副本是 Leader，其余的副本是 follower，所有的读写操作都是经过 Leader 进行的，同时 follower 会定期地去 leader 上的复制数据。当 Leader 挂了的时候，其中一个 follower 会重新成为新的 Leader。通过分区副本，引入了数据冗余，同时也提供了 Kafka 的数据可靠性。

Kafka 的分区多副本架构是 Kafka 可靠性保证的核心，把消息写入多个副本可以使 Kafka 在发生崩溃时仍能保证消息的持久性。

2、ISR机制

Kafka有两种副本数据同步策略（Kafka选择第二种）

方案	优点	缺点
半数以上完成同步，就发送ack	延迟低	选举新的leader时，容忍n台节点的故障，需要2n+1个副本
全部完成同步，才发送ack	选举新的leader时，容忍n台节点的故障，需要n+1个副本	延迟高

Kafka选择了第二种方案，原因如下：

- 为了容忍n台节点的故障，第一种方案需要2n+1个副本，而第二种方案只需要n+1个副本，而Kafka的每个分区都有大量的数据，第一种方案会造成大量数据的冗余。
- 虽然第二种方案的网络延迟会比较高，但网络延迟对Kafka的影响较小。

为了防止Kafka在选择第二种数据同步策略时，因为某一个follower故障导致leader一直等下去，Leader维护了一个动态的in-sync replica set (ISR)。

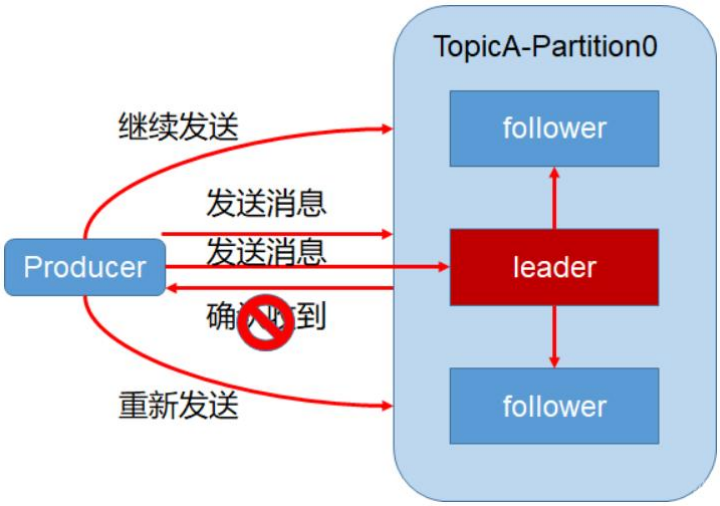
ISR：同步副本，和leader保持同步的follower集合。当ISR中的follower完成数据的同步之后，leader就会给生产者发送ack。

特殊情况：

- 如果follower长时间未向leader同步数据，则该follower将被踢出ISR，该时间阈值由replica.lag.time.max.ms参数设定（默认：10s）。
- 如果Leader发生故障，就会从ISR中选举新的leader。

3、ack应答机制

为保证producer发送的数据，能可靠的发送到指定的topic，topic的每个partition收到producer发送的数据后，都需要向producer发送ack（acknowledgement确认收到），如果producer收到ack，就会进行下一轮的发送，否则重新发送数据。



公众号：旧时光大数据

Kafka为用户提供了三种可靠性级别（acks参数）：

- **acks=0**

  producer不等待broker的ack，broker一接收到还没有写入磁盘就已经返回。

  当**broker故障时**，有可能**丢失数据**。

- **acks=1**

  producer等待broker的ack，partition的leader落盘成功后返回ack。

  如果在follower同步成功之前leader故障，那么将会**丢失数据**。

- **acks=-1(all)**

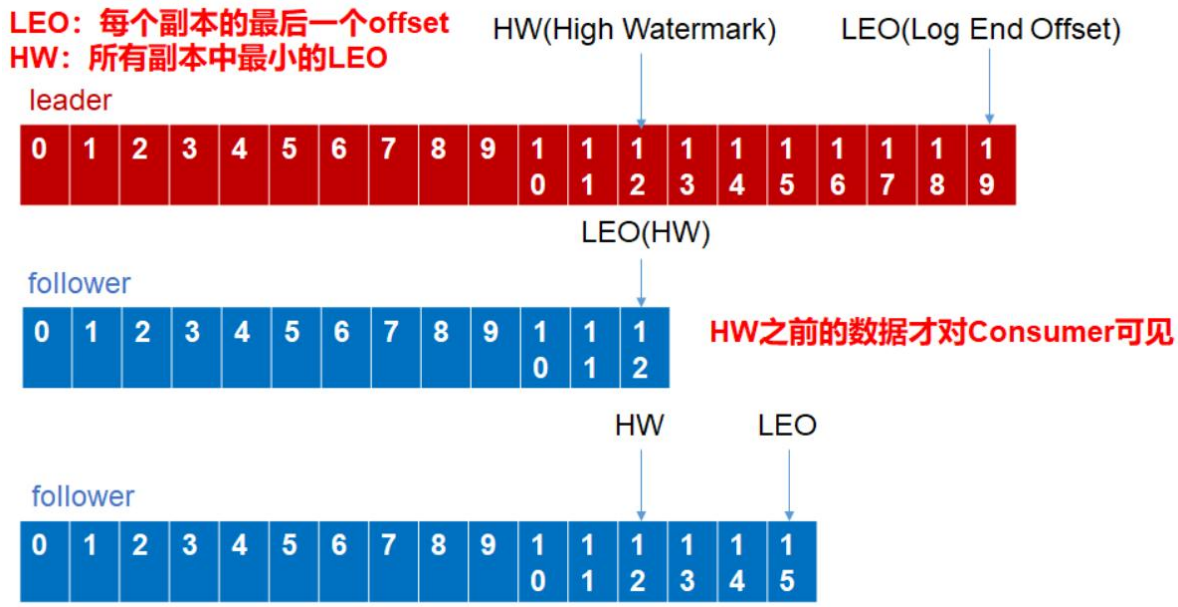
  producer等待broker的ack，partition的leader和follower（ISR中的所有follower）全部落盘成功后才返回ack。

  如果在**follower同步完成后**，broker**发送ack之前leader发生故障**，此时kafka从ISR中重新选举一个leader，生产者没有收到ack重新发送一份到新leader上，则造成**数据重复**。

  如果**ISR中只剩一个leader**时，此时**leader发生故障**，可能会造成**数据丢失**。

  如果**一个follower故障**，该节点被踢出ISR，只要ISR中所有节点都同步即可返回ack，**不影响**。

4、故障处理：HW、LEO



- LEO：每个副本的最后一个Offset
- HW：所有副本中最小的LEO

1）follower故障

follower发生故障后会被临时踢出ISR，待该follower恢复后，follower会读取本地磁盘记录的上次的HW，并**将log文件高于HW的部分截取掉**，从HW开始向leader进行同步。等**该follower的LEO大于等于该Partition的HW**，即follower追上leader之后，就可以重新加入ISR了。

2）leader故障

leader发生故障之后，会从ISR中选出一个新的leader，之后，为保证多个副本之间的数据一致性，其余的follower会**先将各自的log文件高于HW的部分截掉，然后从新的leader同步数据**。

注意：这只能保证副本之间的数据一致性，并不能保证数据不丢失或者不重复。（是否丢数据是acks保证）

公众号：旧时光大数据

Kafka数据丢失怎么处理?

可回答：1）Kafka的生产者写数据丢数据怎么办？2）Kafka传输过程中断电了，怎么保证可靠性?

问过的一些公司：小米，网易，严选，创略科技

参考答案：

Producer 数据不丢失：

- 同步模式：配置=1 （只有Leader收到，-1 所有副本成功，0 不等待）Leader Partition挂了，数据就会丢失

  解决方案：设置 -1 保证produce 写入所有副本算成功 producer.type = sync request.required.acks=-1

- 异步模式，当缓冲区满了，如果配置为0（没有收到确认，一满就丢弃），数据立刻丢弃

  解决方案：不限制阻塞超时时间。就是一满生产者就阻塞

```
producer.type = async
request.required.acks=1
queue.buffering.max.ms=5000
queue.buffering.max.messages=10000
queue.enqueue.timeout.ms = -1
batch.num.messages=200
```

Customer 不丢失数据

在获取kafka的消息后正准备入库（未入库），但是消费者挂了，那么如果让kafka自动去维护offset，它就会认为这条数据已经被消费了，那么会造成数据丢失。

解决方案：使用kafka低级API，自己手动维护偏移量，当数据入库之后进行偏移量的更新（适用于基本数据源）

流处理中的几种可靠性语义：

- at most once 每条数据最多被处理一次（0次或1次），会出现数据丢失的问题
- at least once 每条数据最少被处理一次（1次或更多），这个不会出现数据丢失，但是会出现数据重复
- exactly once 每种数据只会被处理一次，没有数据丢失，没有数据重复，这种语义是大家最想实现的，也是最难实现的

但是开启WAL后，依旧存在数据丢失问题，原因是任务中断时receiver也被强行终止了，将会造成数据丢失。在Streaming程序的最后添加代码，只有在确认所有receiver都关闭的情况下才终止程序。

Kafka如何保证全局有序?

可回答：1）Kafka消费者怎么保证有序性？2）Kafka生产者写入数据怎么保证有序？3）Kafka可以保证数据的局部有序，如何保证数据的全局有序？4）Kafka消息的有序性

问过的一些公司：快手x3，360x2，安恒信息，京东，京东(2021.07)，重庆富民银行(2021.09)

参考答案：

1、设置Key值，指定分区

kafka分区是存在K和V的，K就是分区，一般都是默认的，而默认的经常会发生一些我们并不像看到的结果，例如对同一数据进行多次操作不同分区会导致后进入先出，这就是因为跨分区导致的结果，因此我们要**设置key用来进行hash取模来确定分区**，并且，这个再kafka源码是存在的，就在DefaultPartitioner.java中。

```java
    /**
     * Compute the partition for the given record.
     *
     * @param topic The topic name
     * @param key The key to partition on (or null if no key)
     * @param keyBytes serialized key to partition on (or null if no key)
     * @param value The value to partition on or null
     * @param valueBytes serialized value to partition on or null
     * @param cluster The current cluster metadata
     */
    public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {
        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();
        if (keyBytes == null) {
            int nextValue = nextValue(topic);
            List<PartitionInfo> availablePartitions = cluster.availablePartitionsForTopic(topic);
            if (availablePartitions.size() > 0) {
                int part = Utils.toPositive(nextValue) % availablePartitions.size();
                return availablePartitions.get(part).partition();
            } else {
                // no partitions are available, give a non-available partition
                return Utils.toPositive(nextValue) % numPartitions;
            }
        } else {
            // hash the keyBytes to choose a partition
            return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;
        }
    }

    private int nextValue(String topic) {
        AtomicInteger counter = topicCounterMap.get(topic);
        if (null == counter) {
            counter = new AtomicInteger(ThreadLocalRandom.current().nextInt());
            AtomicInteger currentCounter = topicCounterMap.putIfAbsent(topic, counter);
            if (currentCounter != null) {
                counter = currentCounter;
            }
```

通过源码我们可以发现，再源码中默认的key为空，则系统会运算出一个partition，如果用这种方式，那么，就会导致分区内有序而分区无序，会导致数据无序，因此，**要指定一个key值，也就是指定分区，这样的话，同一个数据发送到同一个分区，而多个分区依旧可以并行，同时实现数据有序和多分区并行。**

如果数据是从MySQL传过来的话，一般来说都是会有自增长主键的，我们可以直接读取主键来作为运算的key值进行操作，这样就可以保证数据的操作唯一性了。

2、`max.in.flight.requests.per.connection`设置为1

`max.in.flight.requests.per.connection`设置为1,每个批次里面一次只写入一条消息到kafka里面，而这个参数，在kafka官网中是有说明的：

max.in.flight.requests.per.connection	The maximum number of unacknowledged requests the client will send on a single connection before blocking. Note that if this setting is set to be greater than 1 and there are failed sends, there is a risk of message re-ordering due to retries (i.e., if retries are enabled).	int	5	[1,...]	low

官网已经说的很明显了，如果设置打于1，那么每次重试可能存在从新排序（乱序）的情况，比如我设置为5，那么我传输的过程应该是1，2，3，4，5。

但是如果网络波动导致3丢失，那么kafka自动重试只会拉取3，也就是排序就成了1，2，4，5，3。

所以，记得将每个批次的消息数设置为1，防止出现重试乱序的情况。

3、设置重试次数大于100次

每一次kafka操作为了防止网络延时等问题，要设置重试次数大于100次

生产者消费者模式与发布订阅模式有何异同?

问过的一些公司：百度

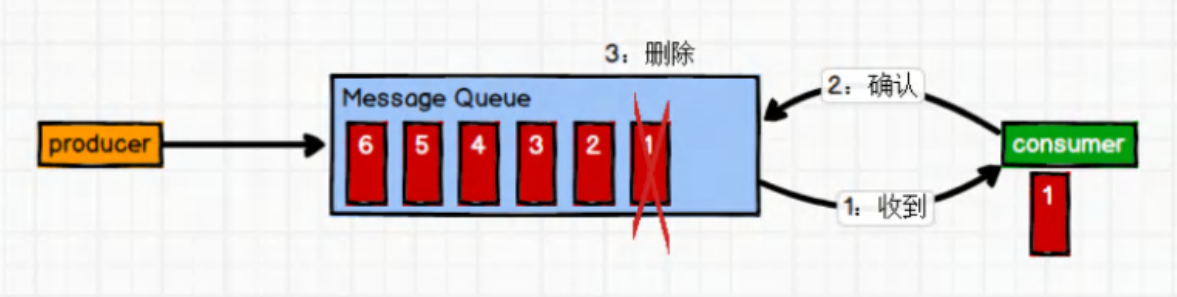
公众号：旧时光大数据

参考答案：

1、点对点模式（一对一，消费者主动拉取数据，消息收到后消息清除）

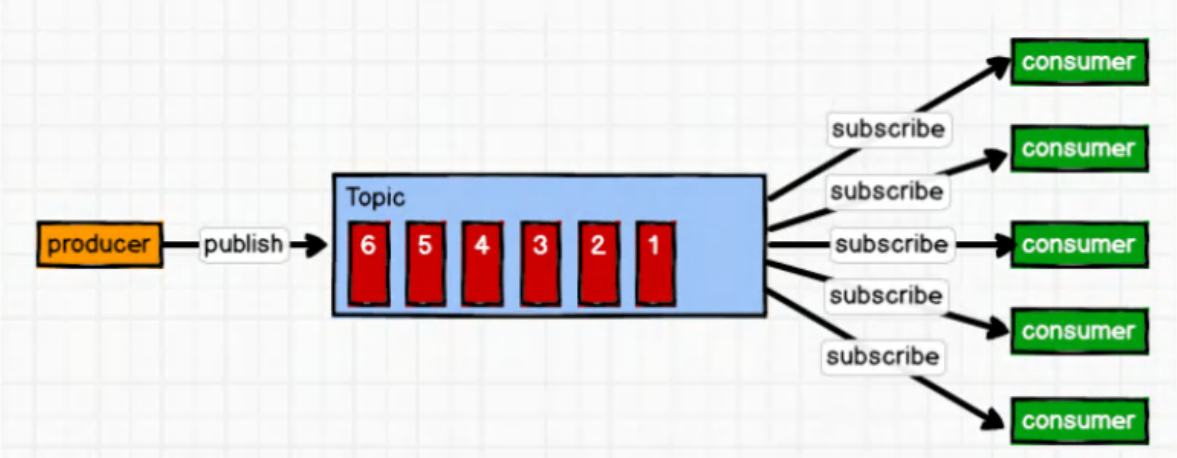
消息生产者生产消息发送到Queue中，然后消息消费者从Queue中取出并且消费消息。

消息被消费以后，Queue中不再有存储，所以消息消费者不可能消费到已经被消费的消息。Queue支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费。



2、发布/订阅模式（一对多，消费者消费数据之后不会清除消息）

消息生产者（发布）将消息发布到topic中，同时有多个消息消费者（订阅）消费该消息。和点对点模式不同，发布到topic的消息会被所有订阅者消费。



Kafka的消费者组是如何消费数据的

问过的一些公司：京东

参考答案：

Consumer Group （CG）

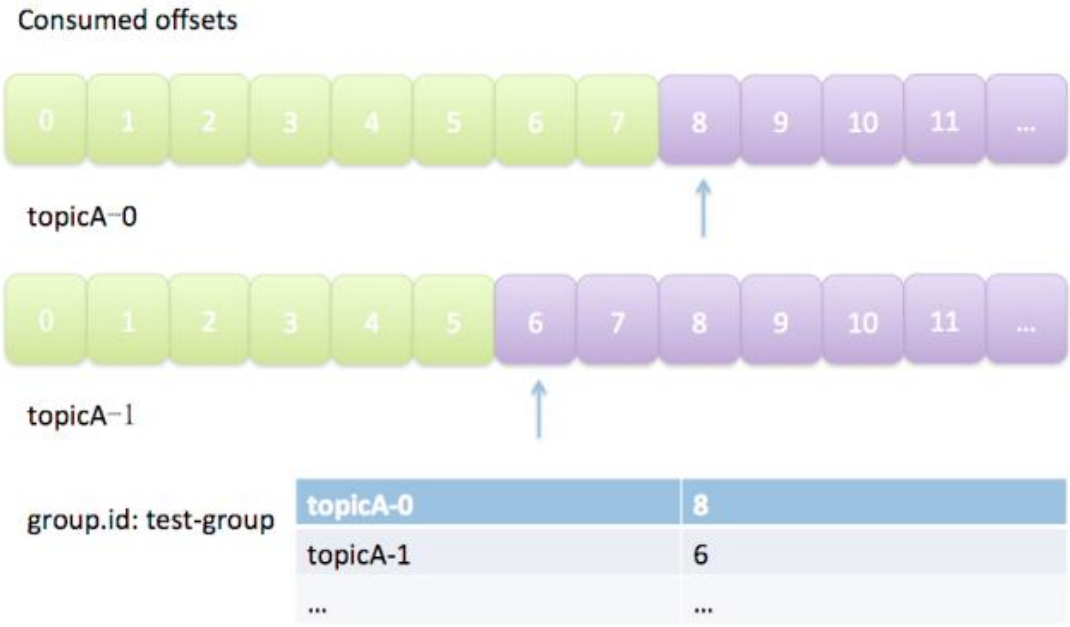
消费者组，由多个consumer组成。消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个消费者消费；消费者组之间互不影响。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。

Kafka的offset管理

问过的一些公司：招银网络

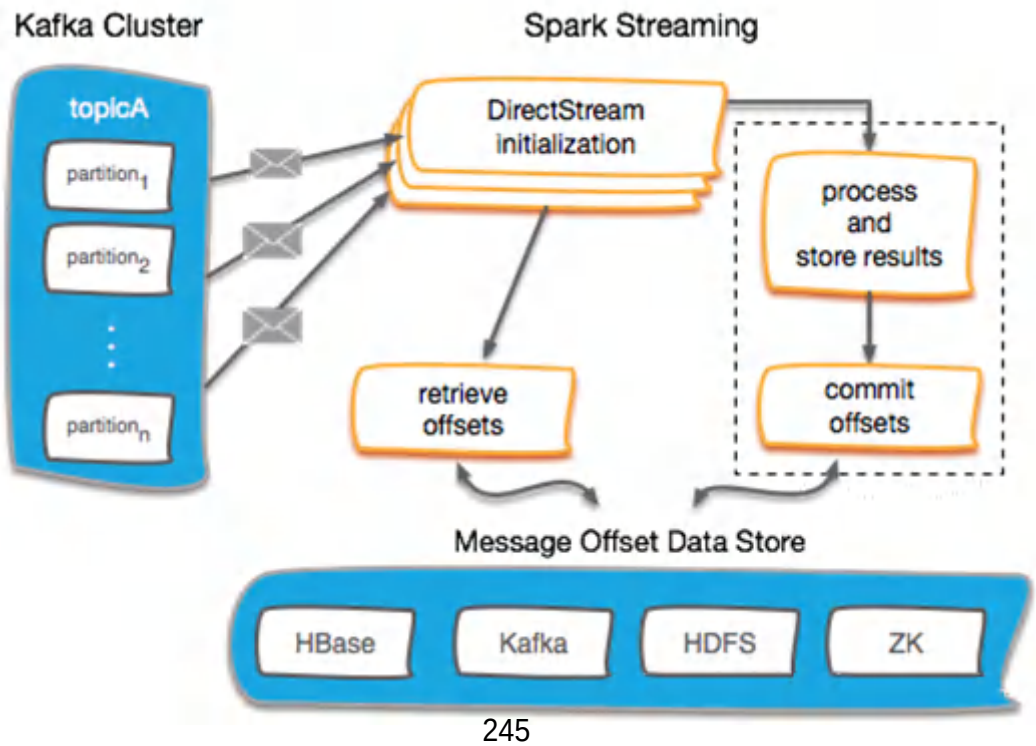
参考答案：

消费者在消费的过程中需要记录自己消费了多少数据，即消费 Offset。Kafka Offset 是Consumer Position，与 Broker 和 Producer 都无关。每个 Consumer Group、每个 Topic 的每个Partition 都有各自的 Offset，如下图所示。



通常有如下几种 Kafka Offset 的管理方式：

- Spark Checkpoint：在 Spark Streaming 执行Checkpoint 操作时，将 Kafka Offset 一并保存到 HDFS 中。这种方式的问题在于：当 Spark Streaming 应用升级或更新时，以及当Spark 本身更新时，Checkpoint 可能无法恢复。因而，不推荐采用这种方式。
- HBASE、Redis 等外部 NOSQL 数据库：这一方式可以支持大吞吐量的 Offset 更新，但它最大的问题在于：用户需要自行编写 HBASE 或 Redis 的读写程序，并且需要维护一个额外的组件。
- ZOOKEEPER：老版本的位移offset是提交到zookeeper中的，目录结构是：`/consumers/<group.id>/offsets/ <topic>/<partitionId>`，但是由于 ZOOKEEPER 的写入能力并不会随着 ZOOKEEPER 节点数量的增加而扩大，因而，当存在频繁的 Offset 更新时，ZOOKEEPER 集群本身可能成为瓶颈。因而，不推荐采用这种方式。
- Kafka自身的一个特殊Topic（__consumer_offsets）中：这种方式支持大吞吐量的Offset更新，又不需要手动编写 Offset 管理程序或者维护一套额外的集群，因而是迄今为止最为理想的一种实现方式。



公众号：旧时光大数据

另外几个与 Kafka Offset 管理相关的要点如下：

- Kafka 默认是定期帮你自动提交位移的（enable.auto.commit=true）。有时候，我们需要采用自己来管理位移提交，这时候需要设置 enable.auto.commit=false。
- 属性 auto.offset.reset 值含义解释如下：
  - earliest： 当各分区下有已提交的 Offset 时，从“提交的 Offset”开始消费；无提交的Offset 时，从头开始消费；
  - latest： 当各分区下有已提交的 Offse`t 时，从提交的 Offset 开始消费；无提交的 Offset时，消费新产生的该分区下的数据；
  - none： Topic 各分区都存在已提交的 Offset 时，从 Offset 后开始消费；只要有一个分区不存在已提交的 Offset，则抛出异常。

```
kafka-0.10.1.X版本之前：auto.offset.reset 的值为smallest,和,largest.(offest保存在zk中)；
kafka-0.10.1.X版本之后：auto.offset.reset 的值更改为:earliest,latest,和none (offest保存在kafka的一个特殊的topic名为:__consumer_offsets里面)；
```

Kafka为什么同一个消费者组的消费者不能消费相同的分区?

问过的一些公司：网易

参考答案：

Kafka通过消费者组机制同时实现了发布/订阅模型和点对点模型。多个组的消费者消费同一个分区属于多订阅者的模式，自然没有什么问题；而在单个组内某分区只交由一个消费者处理的做法则属于点对点模式。其实这就是设计上的一种取舍，如果Kafka真的允许组内多个消费者消费同一个分区，也不是什么灾难性的事情，只是没什么意义，而且还会重复消费消息。

通常情况下，我们还是希望一个组内所有消费者能够分担负载，让彼此做的事情没有交集，做一些重复性的劳动纯属浪费资源。就如同电话客服系统，每个客户来电只由一位客服人员响应。那么请问我就是想让多个人同时接可不可以？当然也可以了，技术上没什么困难，只是这么做没有任何意义罢了，既拉低了整体的处理能力，也造成了人力成本的浪费。

总之，这种设计不是出于技术上的考量而更多还是看效率等非技术方面。

如果有一条offset对应的数据，消费完成之后，手动提交失败，如何处理?

问过的一些公司：安恒信息

参考答案：

回滚，利用Kafka的事务解决。

正在消费一条数据，Kafka挂了，重启以后，消费的offset是哪一个

问过的一些公司：招银网络

参考答案：

公众号：旧时光大数据

可以在Java API中设置`auto.offset.reset`的值（ConsumerConfig.AUTO_OFFSET_RESET_CONFIG）来进行指定。

先来看几个测试实验，结论在最后会给出

auto.offset.reset值含义解释

`earliest`：当各分区下有已提交的offset时，从提交的offset开始消费；无提交的offset时，从头开始消费

`latest`：当各分区下有已提交的offset时，从提交的offset开始消费；无提交的offset时，消费新产生的该分区下的数据

`none`：topic各分区都存在已提交的offset时，从offset后开始消费；只要有一个分区不存在已提交的offset，则抛出异常

案例测试如下：

1、同分组测试

1）测试一

测试环境：Topic为lsztopic7，并生产30条信息。lsztopic7详情：

```
Topic:lsztopic7 PartitionCount:4        ReplicationFactor:3     Configs:
        Topic: lsztopic7        Partition: 0    Leader: 181     Replicas: 181,180,179   Isr: 181,180,179
        Topic: lsztopic7        Partition: 1    Leader: 179     Replicas: 179,181,180   Isr: 179,181,180
        Topic: lsztopic7        Partition: 2    Leader: 180     Replicas: 180,179,181   Isr: 180,179,181
        Topic: lsztopic7        Partition: 3    Leader: 181     Replicas: 181,179,180   Isr: 181,179,180
```

创建组为“testtopi7”的consumer，将enable.auto.commit设置为false，不提交offset。依次更改auto.offset.reset的值。此时查看offset情况为：

```
GROUP                          TOPIC                          PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             OWNER
testtopi7                      lsztopic7                      0          unknown         12              unknown         consumer-1_/10.10.10.1
testtopi7                      lsztopic7                      1          unknown         7               unknown         consumer-1_/10.10.10.1
testtopi7                      lsztopic7                      2          unknown         4               unknown         consumer-1_/10.10.10.1
testtopi7                      lsztopic7                      3          unknown         7               unknown         consumer-1_/10.10.10.1
```

测试结果：

earliest：客户端读取30条信息，且各分区的offset从0开始消费。

latest：客户端读取0条信息。

none：抛出NoOffsetForPartitionException异常。

```
Exception in thread "pool-1-thread-1" org.apache.kafka.clients.consumer.NoOffsetForPartitionException: Undefined offset with no reset policy for partition: lsztopic7-3
    at org.apache.kafka.clients.consumer.internals.Fetcher.resetOffset(Fetcher.java:288)
    at org.apache.kafka.clients.consumer.internals.Fetcher.updateFetchPositions(Fetcher.java:167)
    at org.apache.kafka.clients.consumer.KafkaConsumer.updateFetchPositions(KafkaConsumer.java:1302)
    at org.apache.kafka.clients.consumer.KafkaConsumer.pollOnce(KafkaConsumer.java:895)
    at org.apache.kafka.clients.consumer.KafkaConsumer.poll(KafkaConsumer.java:853)
    at kafka.consumer.GroupComsumer.run(GroupComsumer.java:49) <2 internal calls>
    at java.lang.Thread.run(Thread.java:748)
```

测试结论：

```
新建一个同组名的消费者时，auto.offset.reset值含义：
earliest 每个分区是从头开始消费的。
none 没有为消费者组找到先前的offset值时，抛出异常
```

2）测试二

测试环境：测试场景一下latest时未接受到数据，保证该消费者在启动状态，使用生产者继续生产10条数据，总数据为40条。

```
GROUP                          TOPIC                          PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             OWNER
testtopi7                      lsztopic7                      0          unknown         15              unknown         consumer-1_/10.10.10.1
testtopi7                      lsztopic7                      1          unknown         11              unknown         consumer-1_/10.10.10.1
testtopi7                      lsztopic7                      2          unknown         5               unknown         consumer-1_/10.10.10.1
testtopi7                      lsztopic7                      3          unknown         9               unknown         consumer-1_/10.10.10.1
```

测试结果：

公众号：旧时光大数据

latest：客户端取到了后生产的10条数据

测试结论：

```
1    当创建一个新分组的消费者时，auto.offset.reset值为latest时，表示消费新的数据（从consumer创建开始，后生产的数据），之前产生的数据不消费。
```

3）测试三

测试环境：在测试环境二，总数为40条，无消费情况下，消费一批数据。运行消费者消费程序后，取到5条数据。 即，总数为40条，已消费5条，剩余35条。

```
GROUP                TOPIC                PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             OWNER
testtopi7            lsztopic7            2          5               5               0               consumer-1_/10.10.10.1
testtopi7            lsztopic7            3          0               9               9               consumer-1_/10.10.10.1
testtopi7            lsztopic7            0          unknown         15              unknown         consumer-1_/10.10.10.1
testtopi7            lsztopic7            1          unknown         11              unknown         consumer-1_/10.10.10.1
```

测试结果：

earliest：消费35条数据，即将剩余的全部数据消费完。

latest：

```
1    消费9条数据，都是分区3的值。
2    offset:0 partition:3
3    offset:1 partition:3
4    offset:2 partition:3
5    offset:3 partition:3
6    offset:4 partition:3
7    offset:5 partition:3
8    offset:6 partition:3
9    offset:7 partition:3
10   offset:8 partition:3
```

none：抛出NoOffsetForPartitionException异常

```
Exception in thread "pool-1-thread-1" org.apache.kafka.clients.consumer.NoOffsetForPartitionException: Undefined offset with no reset policy for partition: lsztopic7-1
    at org.apache.kafka.clients.consumer.internals.Fetcher.resetOffset(Fetcher.java:288)
    at org.apache.kafka.clients.consumer.internals.Fetcher.updateFetchPositions(Fetcher.java:167)
    at org.apache.kafka.clients.consumer.KafkaConsumer.updateFetchPositions(KafkaConsumer.java:1302)
    at org.apache.kafka.clients.consumer.KafkaConsumer.pollOnce(KafkaConsumer.java:895)
    at org.apache.kafka.clients.consumer.KafkaConsumer.poll(KafkaConsumer.java:853)
    at kafka.consumer.GroupComsumer.run(GroupComsumer.java:49) <2 internal calls>
    at java.lang.Thread.run(Thread.java:748)
```

测试结论：

```
1    earliest：当分区下有已提交的offset时，从提交的offset开始消费；无提交的offset时，从头开始消费。
2    latest：当分区下有已提交的offset时，从提交的offset开始消费；无提交的offset时，消费新产生的该分区下的数据。
3    none：当该topic下所有分区中存在未提交的offset时，抛出异常。
```

4）测试四

测试环境：再测试三的基础上，将数据消费完，再生产10条数据，确保每个分区上都有已提交的offset。此时，总数为50，已消费40，剩余10条

```
GROUP                TOPIC                PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             OWNER
testtopi7            lsztopic7            0          15              21              6               consumer-1_/10.10.10.1
testtopi7            lsztopic7            1          11              11              0               consumer-1_/10.10.10.1
testtopi7            lsztopic7            2          5               6               1               consumer-1_/10.10.10.1
testtopi7            lsztopic7            3          9               12              3               consumer-1_/10.10.10.1
```

测试结果：

none：

公众号：旧时光大数据

```
消费10条信息，且各分区都是从offset开始消费
offset:9 partition:3
offset:10 partition:3
offset:11 partition:3
offset:15 partition:0
offset:16 partition:0
offset:17 partition:0
offset:18 partition:0
offset:19 partition:0
offset:20 partition:0
offset:5 partition:2
```

测试结论：

```
值为none时，topic各分区都存在已提交的offset时，从offset后开始消费；只要有一个分区不存在已提交的offset，则抛出异常。
```

2、不同分组下测试

1）测试五

测试环境：在测试四环境的基础上：总数为50，已消费40，剩余10条，创建不同组的消费者，组名为testother7。

```
GROUP                TOPIC                PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             OWNER
testother7           lsztopic7            0          unknown         21              unknown         consumer-1_/10.10.10.1
testother7           lsztopic7            1          unknown         11              unknown         consumer-1_/10.10.10.1
testother7           lsztopic7            2          unknown         6               unknown         consumer-1_/10.10.10.1
testother7           lsztopic7            3          unknown         12              unknown         consumer-1_/10.10.10.1
```

http://blog.csdn.net/lishuangzhe7047

测试结果：

earliest：消费50条数据，即将全部数据消费完。

latest：消费0条数据。

none：抛出异常

```
Exception in thread "pool-1-thread-1" org.apache.kafka.clients.consumer.NoOffsetForPartitionException: Undefined offset with no reset policy for partition: lsztopic7-3
    at org.apache.kafka.clients.consumer.internals.Fetcher.resetOffset(Fetcher.java:288)
    at org.apache.kafka.clients.consumer.internals.Fetcher.updateFetchPositions(Fetcher.java:167)
    at org.apache.kafka.clients.consumer.KafkaConsumer.updateFetchPositions(KafkaConsumer.java:1302)
    at org.apache.kafka.clients.consumer.KafkaConsumer.pollOnce(KafkaConsumer.java:895)
    at org.apache.kafka.clients.consumer.KafkaConsumer.poll(KafkaConsumer.java:853)
    at kafka.consumer.GroupComsumer.run(GroupComsumer.java:49) <2 internal calls>
    at java.lang.Thread.run(Thread.java:748)
```

http://blog.csdn.net/lishuangzhe7047

测试结论：

```
组与组间的消费者是没有关系的。
topic中已有分组消费数据，新建其他分组ID的消费者时，之前分组提交的offset对新建的分组消费不起作用。
```

3、总结

从上面的实验来看，基本上对auto.offset.reset不同的值进行了测试

根据下面的实验数据来进行分析

```
GROUP                TOPIC                PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             OWNER
testother7           lsztopic7            0          unknown         21              unknown         consumer-1_/10.10.10.1
testother7           lsztopic7            1          unknown         11              unknown         consumer-1_/10.10.10.1
testother7           lsztopic7            2          unknown         6               unknown         consumer-1_/10.10.10.1
testother7           lsztopic7            3          unknown         12              unknown         consumer-1_/10.10.10.1
```

http://blog.csdn.net/lishuangzhe7047

CURRENT-OFFSET：表示消费者消费了数据之后提交的offset，即消费者消费了的数据的偏移量。如果为unknown，则表示消费者未提交过offset。

LOG-END-OFFSET：表示的是该分区的HW。

LAG：表示延迟滞后，也就是生产者已经写到kafka集群了，然后有还没有被消费的数量，是logSize-currentOffset，logsize指的是消息总数。

得出以下的一些结论：

1）如果**CURRENT-OFFSET不是为unknown**（消费者以前消费过数据，提交过offset），重启消费者时earliest、latest、none都是会从CURRENT-OFFSET一直消费到LOG-END-OFFSET。也就是不会更新offset。

2）如果**CURRENT-OFFSET为unknown**，重启消费者时earliest、latest、none才会展现出他们各自的不同：

- **earliest**：会从该分区当前最开始的offset消息开始消费(即从头消费)，如果最开始的消息offset是0，那么消费者的offset就会被更新为0.
- **latest**：只消费当前消费者启动完成后生产者新生产的数据。旧数据不会再消费。offset被重置为分区的HW。
- **none**：启动消费者时，该消费者所消费的主题的分区没有被消费过，就会抛异常。(一般新建主题或者用新的消费者组是使用这个就会抛异常。。宕机重启的话，使用这个就没问题。。这个的作用是什么？我猜测应该是用于在重启消费者时检查该消费者所消费的主题以及所属的消费者组的名称是否写错了，导致该消费者没有消费原来主题分区)

这里再强调一遍：

在Java API中设置auto.offset.reset值（ConsumerConfig.AUTO_OFFSET_RESET_CONFIG）。

Kafka支持什么语义，怎么实现Exactly Once?

可回答：

问过的一些公司：网易，快手 x 2

参考答案：

Kafka支持的**消费语义**有以下几种：

- at most once：最多消费一次，消息可能会丢失-------log日志
- at least once：至少消费一次，但是会重复消费 例如手动异步提交offset
- exactly once：正好一次，不丢失，不重复

1）至少一次：at-least-once 表示的是关闭offset自动提交功能，消费端在消费数据的时候很可能在commitAync之前，已经保存在数据库，但是这个时候服务器宕机了，从而导致offset不能提交成功。这个时候再次启动消费者的时候，还是会再次写入数据库，也就是至少一次会重复消费，至少不会丢数据。

2）至多一次：at-most-once 表示有可能是0次或者1次，可以选择开启自动提交offset的功能，然后把自动提交offset的时间设置一下，有可能消费者在消费的时间段内就到了自动提交的时间，从而导致了offset已经提交了，但是数据库保存还没进行。下一次再消费的时候就会认为offset已经成功了，直接丢弃消息。就会造成丢数据。

3）仅一次：exactly-once 表示数据仅被消费一次，还是开启自动开启offset提交的功能，可以开启consumer.seek()方法，相当于自己处理分区和offset，可以在此基础上开启事务，保持原子性，只有数据库保存成功再提交offset，保证两者同时成功。

Exactly Once语义

将服务器的ACK级别设置为-1，可以保证Producer到Server之间不会丢失数据，即At Least Once语义。相对的，将服务器ACK级别设置为0，可以保证生产者每条消息只会被发送一次，即At Most Once语义。

At Least Once可以保证数据不丢失，但是不能保证数据不重复；相对的，At Least Once可以保证数据不重复，但是不能保证数据不丢失。但是，对于一些非常重要的信息，比如说交易数据，下游数据消费者要求数据既不重复也不丢失，即Exactly Once语义。在0.11版本以前的Kafka，对此是无能为力的，只能保证数据不丢失，再在下游消费者对数据做全局去重。对于多个下游应用的情况，每个都需要单独做全局去重，这就对性能造成了很大影响。

0.11版本的Kafka，引入了一项重大特性：**幂等性**。所谓的幂等性就是**指Producer不论向Server发送多少次重复数据，Server端都只会持久化一条。幂等性结合At Least Once语义，就构成了Kafka的Exactly Once语义**。即：

At Least Once + 幂等性 = Exactly Once

要启用幂等性，只需要将Producer的参数中enable.idompotence设置为true即可。Kafka的幂等性实现其实就是将原来下游需要做的去重放在了数据上游。开启幂等性的Producer在初始化的时候会被分配一个PID，发往同一Partition的消息会附带Sequence Number。而Broker端会对<PID, Partition, SeqNumber>做缓存，当具有相同主键的消息提交时，Broker只会持久化一条。

但是PID重启就会变化，同时不同的Partition也具有不同主键，所以幂等性无法保证跨分区跨会话的Exactly Once。

Kafka的消费者和消费者组有什么区别？为什么需要消费者组?

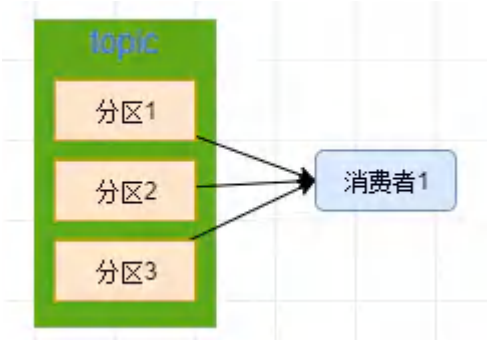
可回答：说下Kafka的消费者和消费者组，以及它们的作用是什么?

问过的一些公司：字节，电信云计算，京东，拼多多，bigo

参考答案：

1、什么是消费者

顾名思义，消费者就是从kafka集群消费数据的客户端，如下图，展示了一个消费者从一个topic中消费数据的模型



2、为什么需要消费者组

如果这个时候 kafka 上游生产的数据很快，超过了这个 `消费者1` 的消费速度，那么就会导致数据堆积，产生一些大家都知道的蛋疼事情了，那么我们只能加强 `消费者` 的消费能力，所以也就有了 `消费者组` 。

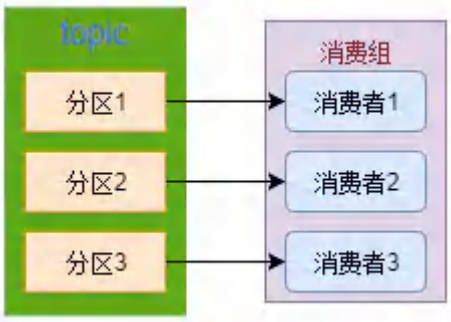
3、什么是消费者组

所谓 `消费者组` ，其实就是一组 `消费者` 的集合，当我们看到下面这张图是不是就特别舒服了，我们采用了一个 `消费组` 来消费这个 `topic` ，众人拾柴火焰高，其消费能力那是按倍数递增的，所以这里我们一般来说都是采用 `消费者组` 来消费数据，而不会是 `单消费者` 来消费数据的。

注意：

一个topic可以被多个消费者组消费，但是每个消费者组消费的数据是互不干扰的，也就是说，每个消费组消费的都是完整的数据 。

一个分区只能被同一个消费组内的一个消费者消费，而不能拆给多个消费者消费，也就是说如果你某个消费者组内的消费者数比该 Topic 的分区数还多，那么多余的消费者是不起作用的



扩展一下：

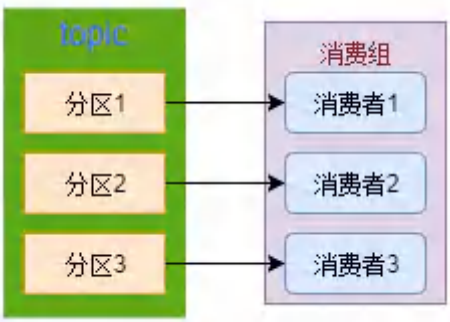
1）是不是一个消费组的消费者越多其消费能力就越强呢?

从下图我们就可以很好的可以回答这个问题了，我们可以看到消费者4是完全没有消费任何的数据的，所以如果你想要加强消费者组的能力，除了添加消费者，分区的数量也是需要跟着增加的，只有这样他们的并行度才能上的去，消费能力才会强。

2）为了提高消费组的消费能力，我是不是可以随便添加分区和消费者呢?

答案当然是否定的。。。我们看到下图，一般来说我们建议消费者数量和分区数量是一致的，当我们的消费能力不够时，就必须通过调整分区的数量来提高并行度，但是，我们应该尽量来避免这种情况发生。

比如：现在我们需要在下图的基础上增加一个分区4，那么这个分区4该由谁来消费呢？这个时候kafka会进行分区再均衡，来为这个分区分配消费者，分区再均衡期间该Topic是不可用的，并且作为一个被消费者，分区数的改动将影响到每一个消费者组 ，所以在创建 topic 的时候，我们就应该考虑好分区数，来尽量避免这种情况发生。



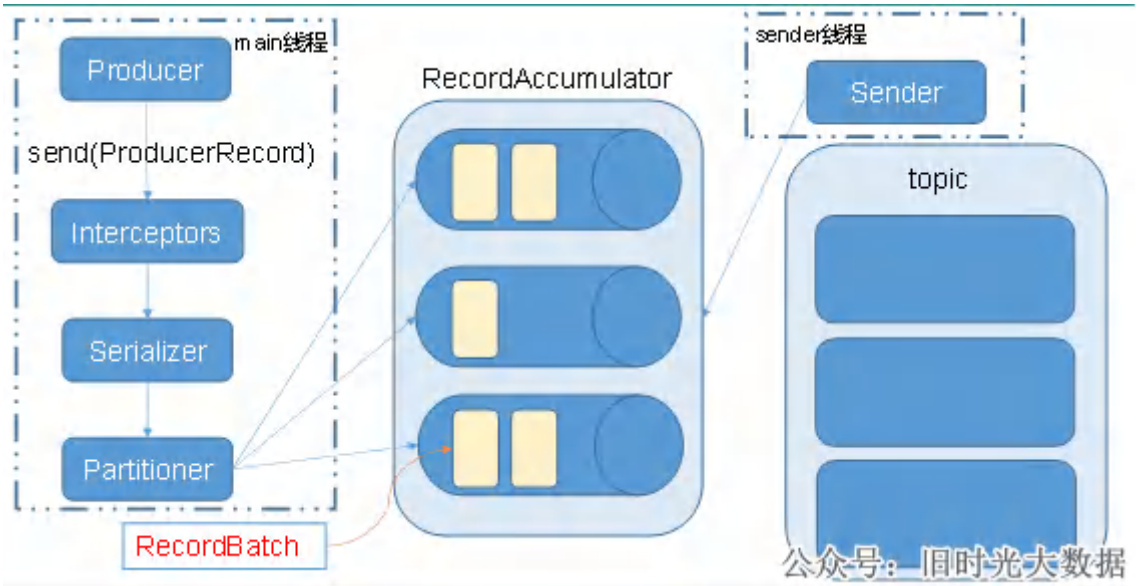
公众号：旧时光大数据

Kafka producer的写入数据过程?

问过的一些公司：流利说

参考答案：

Kafka的Producer发送消息采用的是异步发送的方式。在消息发送的过程中，涉及到了两个线程——main线程和Sender线程，以及一个线程共享变量——RecordAccumulator。main线程将消息发送给RecordAccumulator，Sender线程不断从RecordAccumulator中拉取消息发送到Kafka broker。



相关参数

batch.size：只有数据积累到batch.size之后，sender才会发送数据。

linger.ms：如果数据迟迟未达到batch.size，sender等待linger.time之后就会发送数据。

Kafka producer的ack设置

问过的一些公司：网易

参考答案：

Kafka producer有三种ack机制，初始化producer时在config中进行配置

ack=0

意味着producer不等待broker同步完成的确认，继续发送下一条(批)信息。提供了最低的延迟。但是最弱的持久性，当服务器发生故障时，就很可能发生数据丢失。例如leader已经死亡，producer不知情，还会继续发送消息broker接收不到数据就会数据丢失。

ack=1

意味着producer要等待leader成功收到数据并得到确认，才发送下一条message。此选项提供了较好的持久性较低的延迟性。Partition的Leader死亡，follwer尚未复制，数据就会丢失。

ack=-1

意味着producer得到follwer确认，才发送下一条数据，持久性最好，延时性最差。

三种机制性能递减，可靠性递增。

公众号：旧时光大数据

Kafka的ack机制，解决了什么问题?

问过的一些公司：端点数据(2021.07)

参考答案：

ack机制保证了Producer在发送的数据能可靠的到达指定的Topic，为Producer提供了消息确认机制。 生产者往Broker的Topic中发送消息时，**可以通过配置来决定有几个副本收到这条消息才算消息发送成功。**可以在定义Producer时通过**acks**参数指定，

这个参数支持以下三种值：

ack=0

意味着producer不等待broker同步完成的确认，继续发送下一条(批)信息。提供了最低的延迟。但是最弱的持久性，当服务器发生故障时，就很可能发生数据丢失。例如leader已经死亡，producer不知情，还会继续发送消息broker接收不到数据就会数据丢失。

ack=1

意味着producer要等待leader成功收到数据并得到确认，才发送下一条message。此选项提供了较好的持久性较低的延迟性。Partition的Leader死亡，follwer尚未复制，数据就会丢失。

ack=-1

意味着producer得到follwer确认，才发送下一条数据，持久性最好，延时性最差。

Kafka读取消息是推还是拉的模式？有什么好处?

可回答：Kafka为什么使用拉取消息的机制?

问过的一些公司：流利说，触宝

参考答案：

consumer采用pull（拉）模式从broker中读取数据。

push（推）模式很难适应消费速率不同的消费者，因为消息发送速率是由broker决定的。它的目标是尽可能以最快速度传递消息，但是这样很容易造成consumer来不及处理消息，典型的表现就是拒绝服务以及网络拥塞。而pull模式则可以根据consumer的消费能力以适当的速率消费消息。

pull模式不足之处是，如果kafka没有数据，消费者可能会陷入循环中，一直返回空数据。针对这一点，Kafka的消费者在消费数据时会传入一个时长参数timeout，如果当前没有数据可供消费，consumer会等待一段时间之后再返回，这段时长即为timeout。

Kafka如何实现高吞吐的原理?

可回答：1）Kafka高吞吐的原因；2）Kafka如何保证高吞吐量；3）Kafka为什么低延迟高吞吐？有哪些特点？4）Kafka为什么高可用、高吞吐？如何保证高可用？5）Kafka高性能的实现机制？6）Kafka高性能的原因？7）Kafka零拷贝的实现原理。8）Kafka的数据存储在磁盘但是为什么速度依旧很快？9）Kafka为什么那么快

问过的一些公司：字节，腾讯x3，蘑菇街x3，拼多多，阿里云，美团x3，小鹏汽车，京东，网易x2，快手，猿辅导，转转，字节社招，富途，陌陌(2021.07)x2，海康(2021.08)x2

参考答案：

公众号：旧时光大数据

Kafka 是分布式消息系统，需要处理海量的消息，Kafka 的设计是把所有的消息都写入速度低容量大的硬盘，以此来换取更强的存储能力，但实际上，使用硬盘并没有带来过多的性能损失。

kafka 主要使用了以下几个方式实现了超高的吞吐率

1）顺序读写

kafka 的消息是不断追加到文件中的，这个特性使 kafka 可以充分利用磁盘的顺序读写性能，顺序读写不需要硬盘磁头的寻道时间，只需很少的扇区旋转时间，所以速度远快于随机读写

Kafka 官方给出了测试数据(Raid-5，7200rpm)：

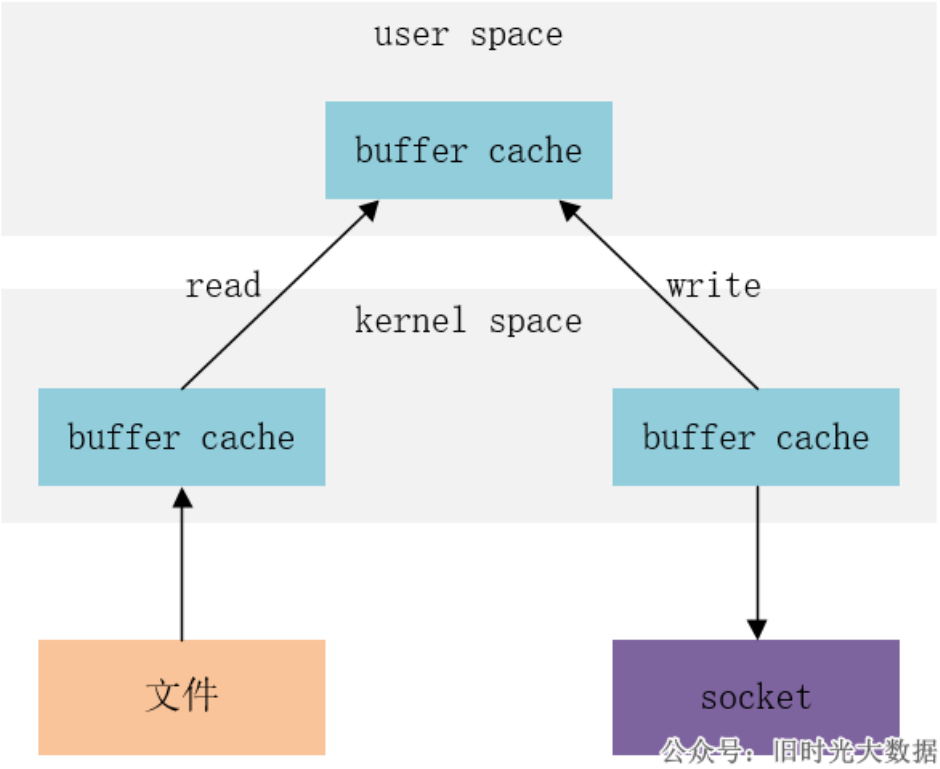
- 顺序 I/O：600MB/s
- 随机 I/O：100KB/s

2）零拷贝

先简单了解下文件系统的操作流程，例如一个程序要把文件内容发送到网络。

这个程序是工作在用户空间，文件和网络socket属于硬件资源，两者之间有一个内核空间。

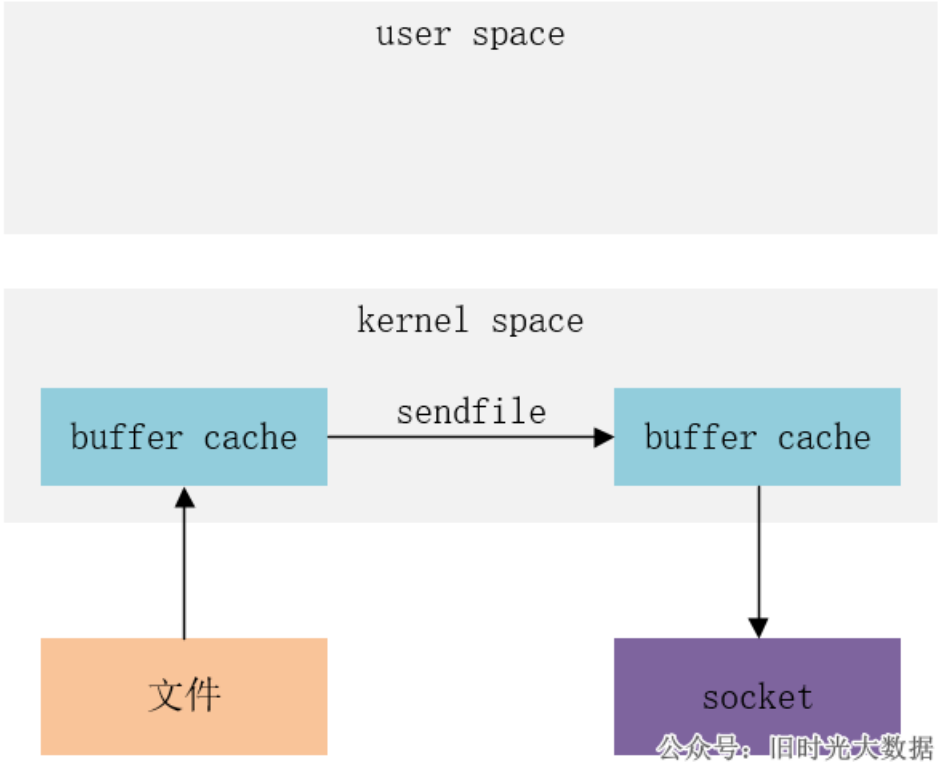
在操作系统内部，整个过程为：



在Linux kernel2.2之后出现了一种叫做"零拷贝(zero-copy)"系统调用机制，就是跳过“用户缓冲区”的拷贝，建立一个磁盘空间和内存的直接映射，数据不再复制到“用户态缓冲区”。

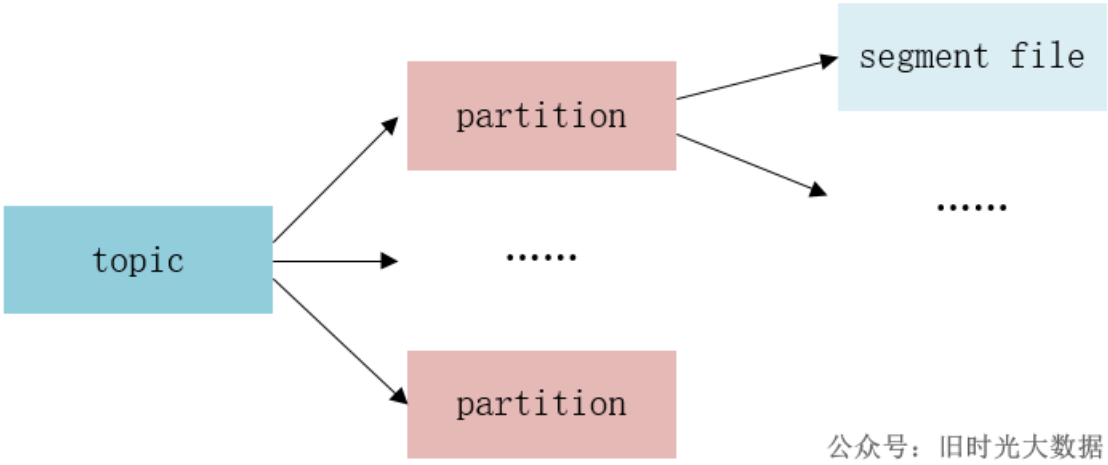
系统上下文切换减少为2次，可以提升一倍的性能

公众号：旧时光大数据



3）文件分段

kafka 的队列topic被分为了多个区partition，每个partition又分为多个段segment，所以一个队列中的消息实际上是保存在N多个片段文件中。



通过分段的方式，每次文件操作都是对一个小文件的操作，非常轻便，同时也增加了并行处理能力。

4）批量发送

Kafka 允许进行批量发送消息，先将消息缓存在内存中，然后一次请求批量发送出去，比如可以指定缓存的消息达到某个量的时候就发出去，或者缓存了固定的时间后就发送出去，如100条消息就发送，或者每5秒发送一次，这种策略将大大减少服务端的I/O次数。

5）数据压缩

Kafka 还支持对消息集合进行压缩，Producer可以通过GZIP或Snappy格式对消息集合进行压缩，压缩的好处就是减少传输的数据量，减轻对网络传输的压力，Producer压缩之后，在Consumer需进行解压，虽然增加了CPU的工作，但在对大数据处理上，瓶颈在网络上而不是CPU，所以这个成本很值得。

说下Kafka中的Partition?

公众号：旧时光大数据

可回答：1）Kafka为什么使用会有partition的数据结构，这样做的好处？（中间的一部分可回答），2）Kafka中的partition如何保证有序?

问过的一些公司：陌陌，阿里云，快手

参考答案：

为了后面更好的理解，前面先介绍下几个概念

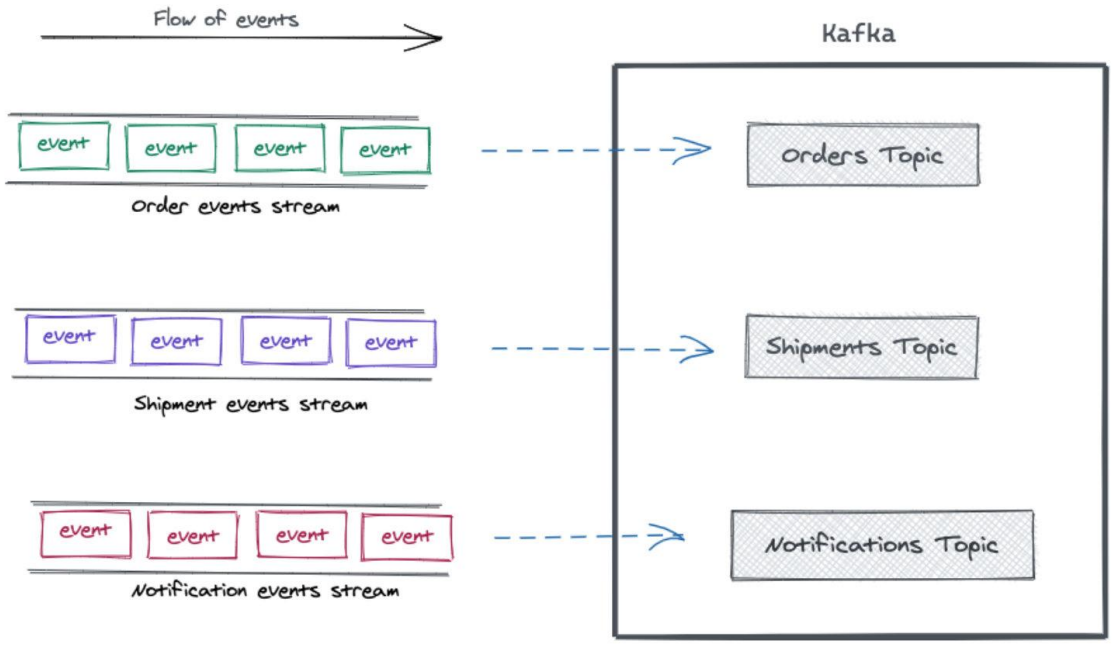
1、Events，Streams，Topics

Event（事件）代表过去发生的一个事实。简单理解就是一条消息、一条记录。

Event 是不可变的，但是很活跃，经常从一个地方流向另一个地方。

Stream 事件流表示运动中的相关事件。

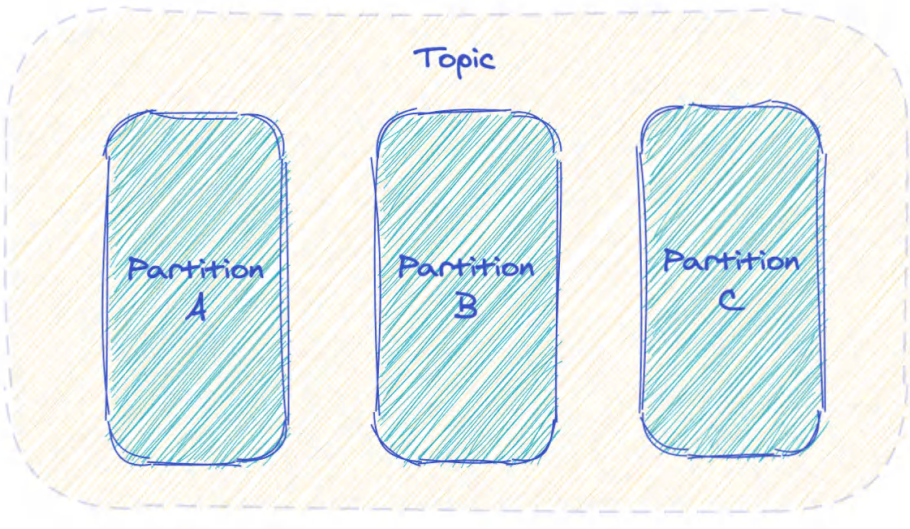
当一个事件流进入 Kafka 之后，它就成为了一个 **Topic** 主题。



所以，Topic 就是具体的事件流，也可以理解为一个 Topic 就是一个静止的 Stream。

Topic 把相关的 Event 组织在一起，并且保存。一个 Topic 就像数据库中的一张表。

2、Partition分区

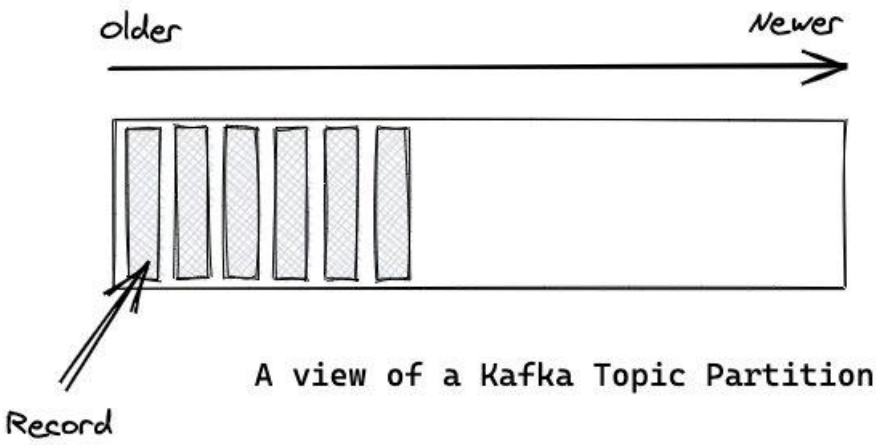


Kafka 中 Topic 被分成多个 Partition 分区。

公众号：旧时光大数据

Topic 是一个**逻辑概念**，Partition 是最小的**存储单元**，掌握着一个 Topic 的部分数据。

每个 Partition 都是一个单独的 log 文件，每条记录都以追加的形式写入。



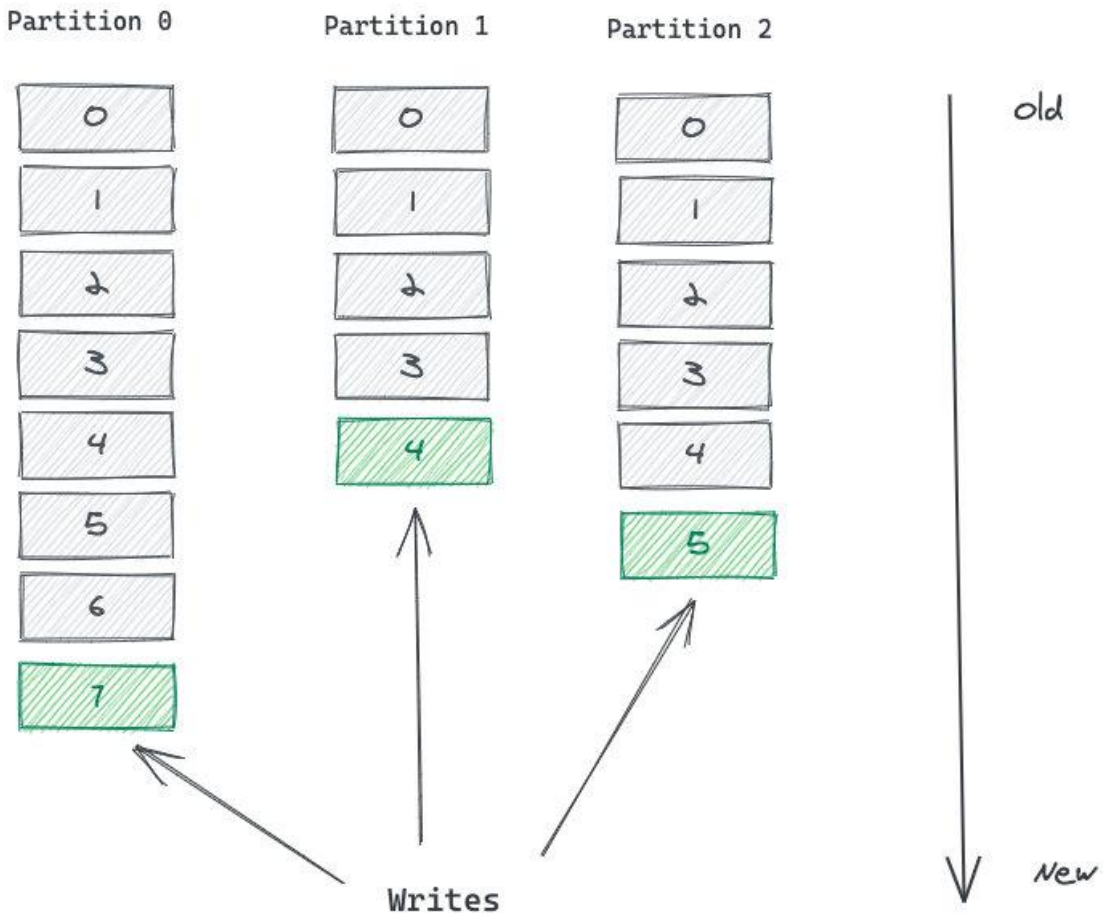
Record（记录） 和 Message（消息）是一个概念。

3、Offsets（偏移量）和消息的顺序（可回答：Kafka中的partition如何保证有序）

Partition 中的每条记录都会被分配一个唯一的序号，称为 **Offset**（偏移量）。

Offset 是一个递增的、不可变的数字，由 Kafka 自动维护。

当一条记录写入 Partition 的时候，它就被追加到 log 文件的末尾，并被分配一个序号，作为 Offset。



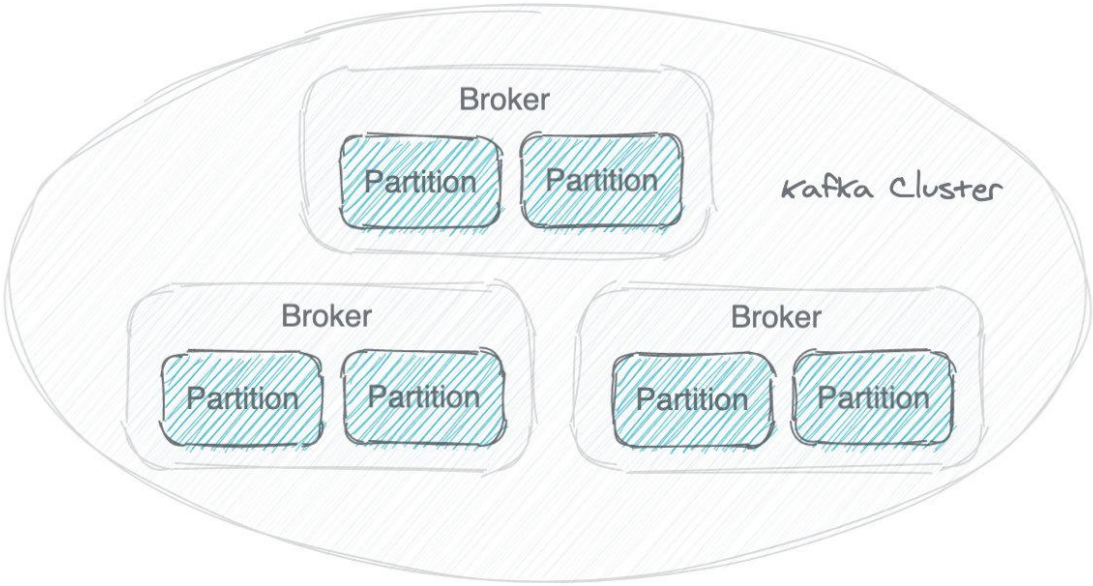
如上图，这个 Topic 有 3 个 Partition 分区，向 Topic 发送消息的时候，实际上是被写入某一个 Partition，并赋予 Offset。

消息的顺序性需要注意，一个 Topic 如果有多个 Partition 的话，那么从 Topic 这个层面来看，消息是无序的。

公众号：旧时光大数据

但单独看 Partition 的话，Partition 内部消息是有序的。所以，一个 Partition 内部消息有序，一个 Topic 跨 Partition 是无序的。如果强制要求 Topic 整体有序，就只能让 Topic 只有一个 Partition。

4、Partition 为 Kafka 提供了扩展能力（可回答partition的作用）



一个 Kafka 集群由多个 Broker（就是 Server） 构成，每个 Broker 中含有集群的部分数据。

Kafka 把 Topic 的多个 Partition 分布在多个 Broker 中。

这样会有多种好处：

- 如果把 Topic 的所有 Partition 都放在一个 Broker 上，那么这个 Topic 的可扩展性就大大降低了，会受限于这个 Broker 的 IO 能力。把 Partition 分散开之后，Topic 就可以水平扩展 。
- 一个 Topic 可以被多个 Consumer 并行消费。如果 Topic 的所有 Partition 都在一个 Broker，那么支持的 Consumer 数量就有限，而分散之后，可以支持更多的 Consumer。
- 一个 Consumer 可以有多个实例，Partition 分布在多个 Broker 的话，Consumer 的多个实例就可以连接不同的 Broker，大大提升了消息处理能力。可以让一个 Consumer 实例负责一个 Partition，这样消息处理既清晰又高效。

5、Partition 为 Kafka 提供了数据冗余（可回答partition的作用）

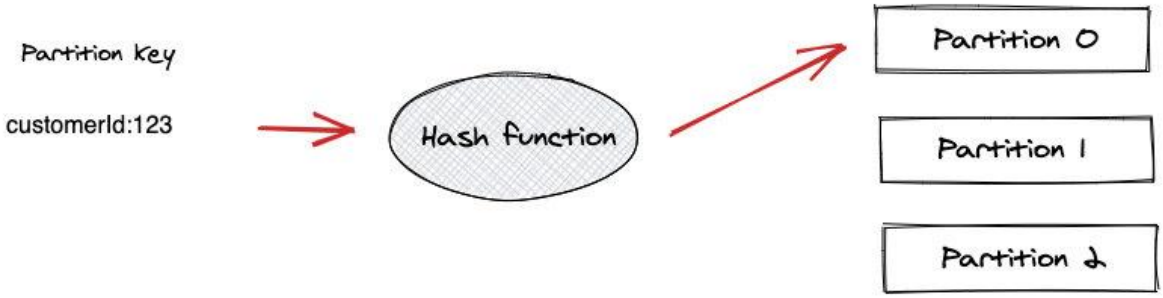
Kafka 为一个 Partition 生成多个副本，并且把它们分散在不同的 Broker。

如果一个 Broker 故障了，Consumer 可以在其他 Broker 上找到 Partition 的副本，继续获取消息。

6、写入Partition

一个Topic 有多个 Partition，那么，向一个 Topic 中发送消息的时候，具体是写入哪个 Partition 呢？有3种写入方式。

1）使用 Partition Key 写入特定 Partition



Producer 发送消息的时候，可以指定一个 Partition Key，这样就可以写入特定 Partition 了。

公众号：旧时光大数据

Partition Key 可以使用任意值，例如设备ID、User ID。Partition Key 会传递给一个 Hash 函数，由计算结果决定写入哪个 Partition。所以，有相同 Partition Key 的消息，会被放到相同的 Partition。

例如使用 User ID 作为 Partition Key，那么此 ID 的消息就都在同一个 Partition，这样可以保证此类消息的有序性。

这种方式需要注意 Partition 热点问题。

例如使用 User ID 作为 Partition Key，如果某一个 User 产生的消息特别多，是一个头部活跃用户，那么此用户的消息都进入同一个 Partition 就会产生热点问题，导致某个 Partition 极其繁忙。

2）由Kafka决定

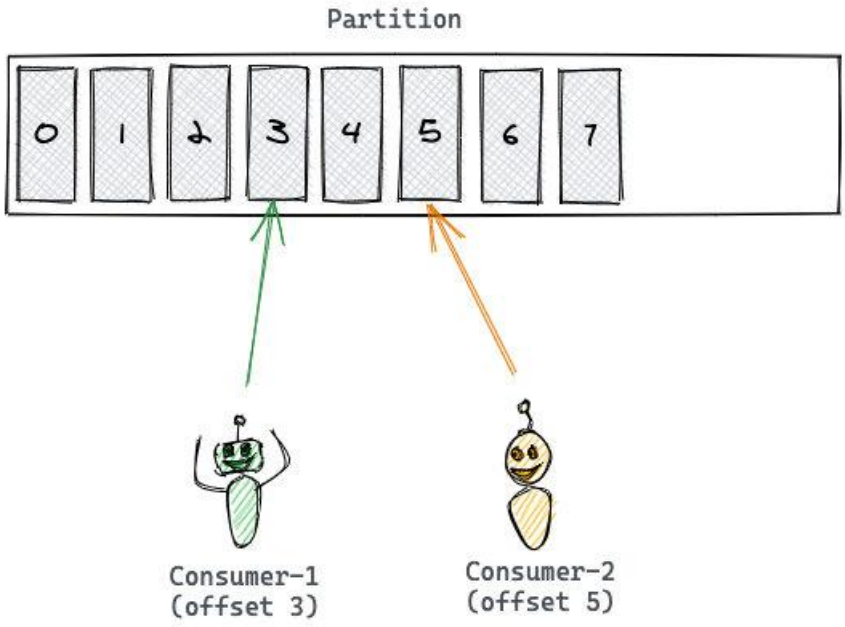
如果没有使用 Partition Key，Kafka 就会使用轮询的方式来决定写入哪个 Partition。这样，消息会均衡的写入各个 Partition。但这样无法确保消息的有序性。

3）自定义规则

Kafka 支持自定义规则，一个 Producer 可以使用自己的分区指定规则。

7、读取Partition

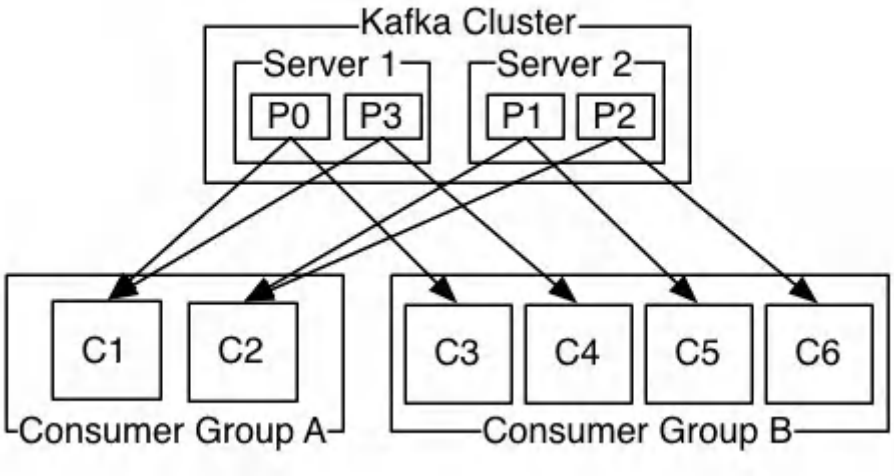
Kafka 不像普通消息队列具有发布/订阅功能，Kafka 不会向 Consumer 推送消息。Consumer 必须自己从 Topic 的 Partition 拉取消息。一个 Consumer 连接到一个 Broker 的 Partition，从中依次读取消息。



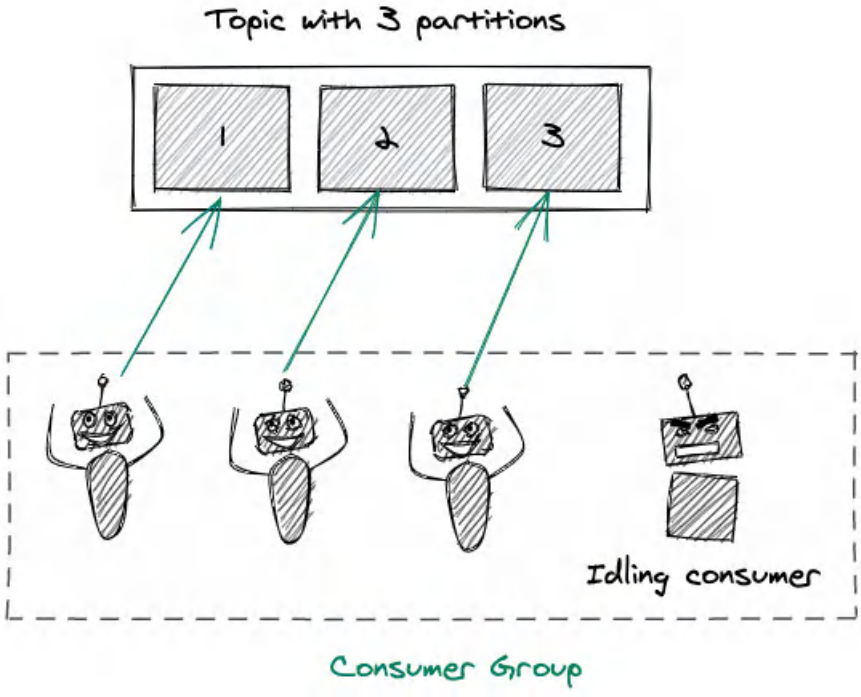
Each consumer has its own view about the partition.

消息的 Offset 就是 Consumer 的游标，根据 Offset 来记录消息的消费情况。读完一条消息之后，Consumer 会推进到 Partition 中的下一个 Offset，继续读取消息。Offset 的推进和记录都是 Consumer 的责任，Kafka 是不管的。

公众号：旧时光大数据



Kafka 中有一个 Consumer Group（消费组）的概念，多个 Consumer 组团去消费一个 Topic。同组的 Consumer 有相同的 Group ID。Consumer Group 机制会保障一条消息只被组内唯一一个 Consumer 消费，不会重复消费。消费组这种方式可以让多个 Partition 并行消费，大大提高了消息的消费能力，最大并行度为 Topic 的 Partition 数量。



例如一个 Topic 有 3 个 Partition，你有 4 个 Consumer 负责这个 Topic，也只会有 Consumer 工作，另一个作为后补队员，当某个 Consumer 故障了，它再补上去，是一种很好的容错机制。

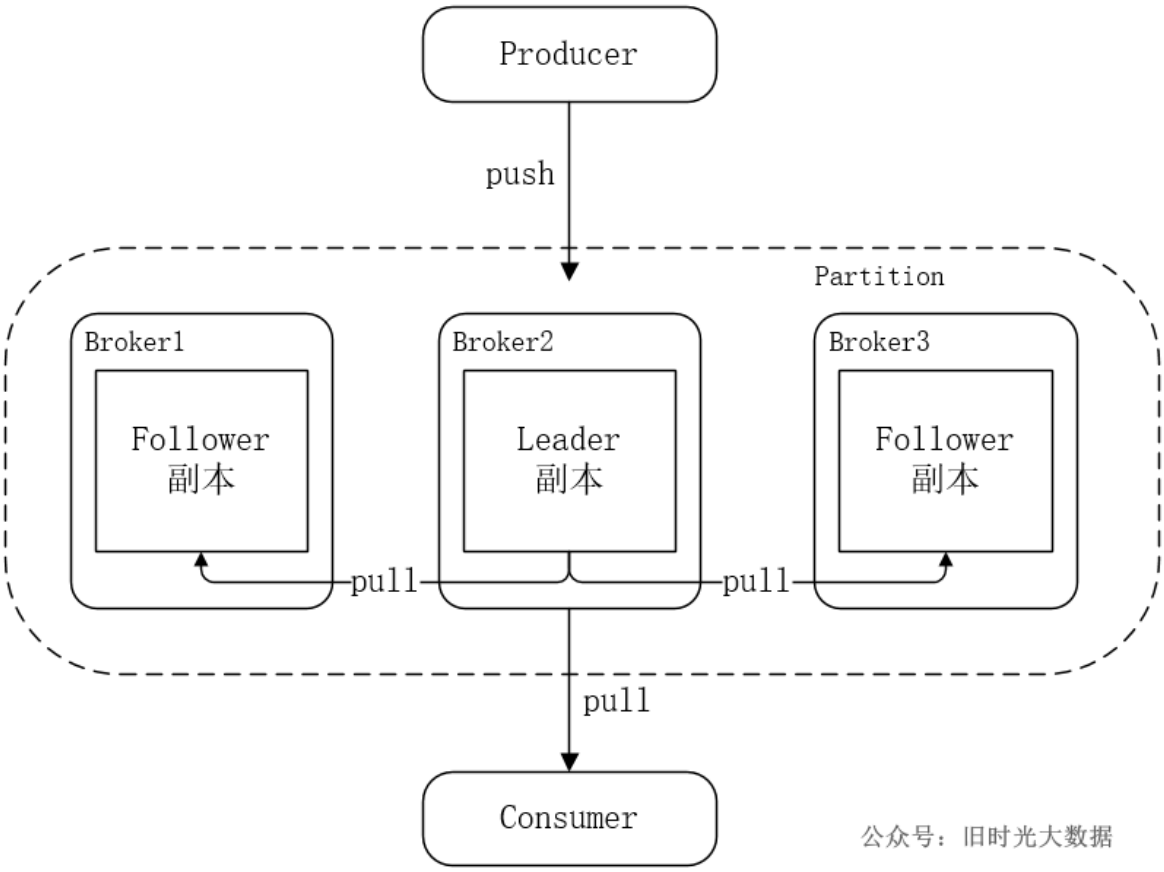
Kafka是如何进行数据备份的?

问过的一些公司：祖龙娱乐

参考答案：

Kafka的备份的单元是partition，也就是每个partition都都会有leader partiton和follow partiton。其中leader partition是用来进行和producer进行写交互，follow从leader副本进行拉数据进行同步，从而保证数据的冗余，防止数据丢失的目的。

公众号：旧时光大数据



Kafka里面存的数据格式是什么样的?

问过的一些公司：阿里云，祖龙娱乐

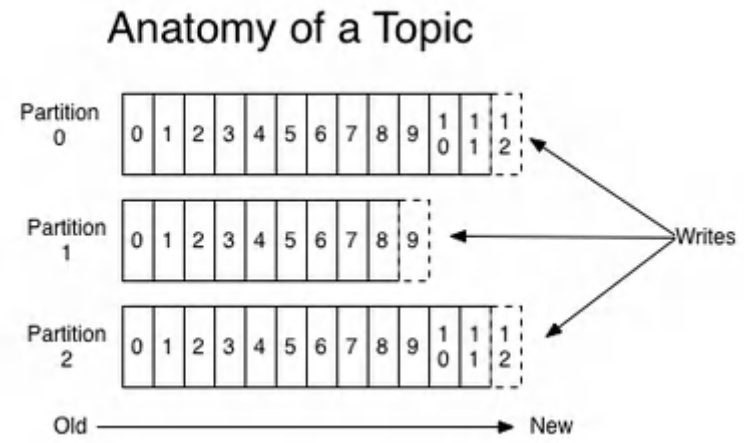
参考答案：

1、先介绍几个重要概念

- **Broker**：消息中间件处理结点，一个Kafka节点就是一个broker，多个broker可以组成一个Kafka集群；
- **Topic**：一类消息，例如page view日志、click日志等都可以以topic的形式存在，Kafka集群能够同时负责多个topic的分发；
- **Partition**：topic物理上的分组，一个topic可以分为多个partition，每个partition是一个有序的队；
- **Segment**：每个partition又由多个segment file组成；
- **offset**：每个partition都由一系列有序的、不可变的消息组成，这些消息被连续的追加到partition中。partition中的每个消息都有一个连续的序列号叫做offset，用于partition唯一标识一条消息；
- **message**：这个算是kafka文件中最小的存储单位，即是 a commit log。

kafka的message是以topic为基本单位，不同topic之间是相互独立的。每个topic又可分为几个不同的partition，每个partition存储一部的分message。topic与partition的关系如下：

公众号：旧时光大数据



其中，partition是以文件夹的形式存储在具体Broker本机上。

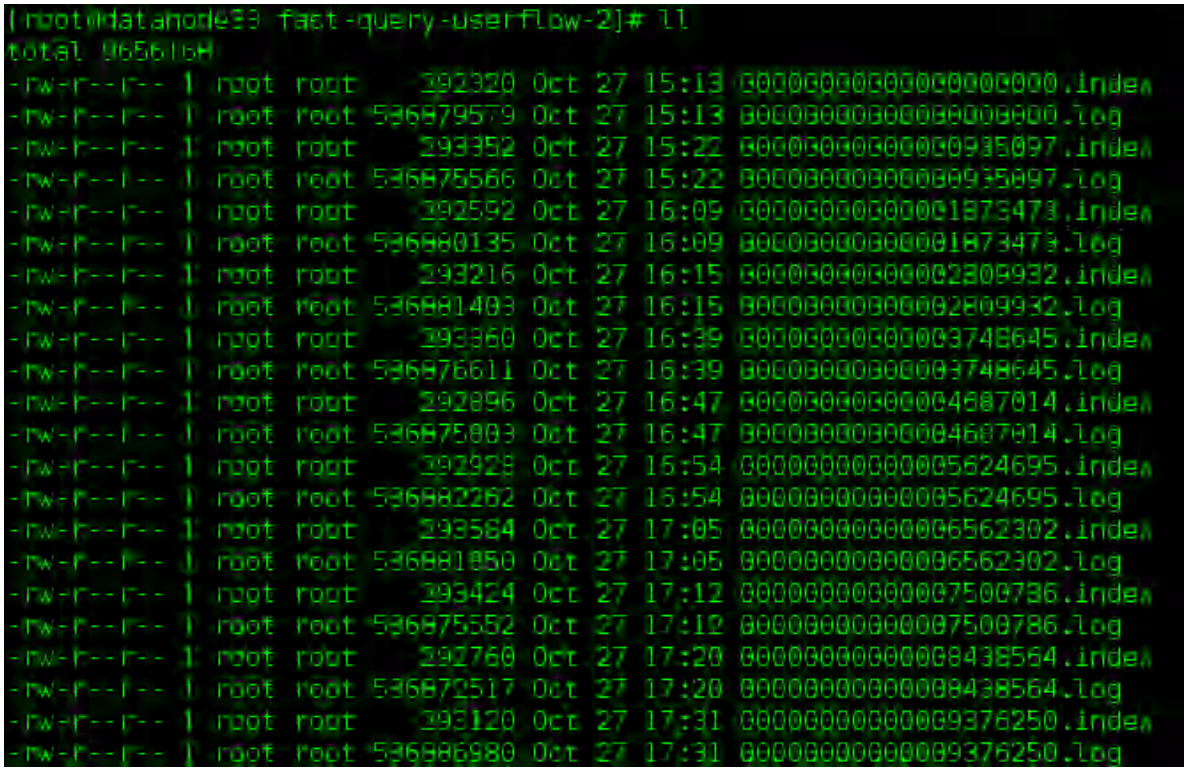
2、partition中的数据文件

1）segment中的文件

对于一个partition（在Broker中以文件夹的形式存在），里面又有很多大小相等的segment数据文件（这个文件的具体大小可以在`config/server.properties`中进行设置），这种特性可以方便old segment file的快速删除。

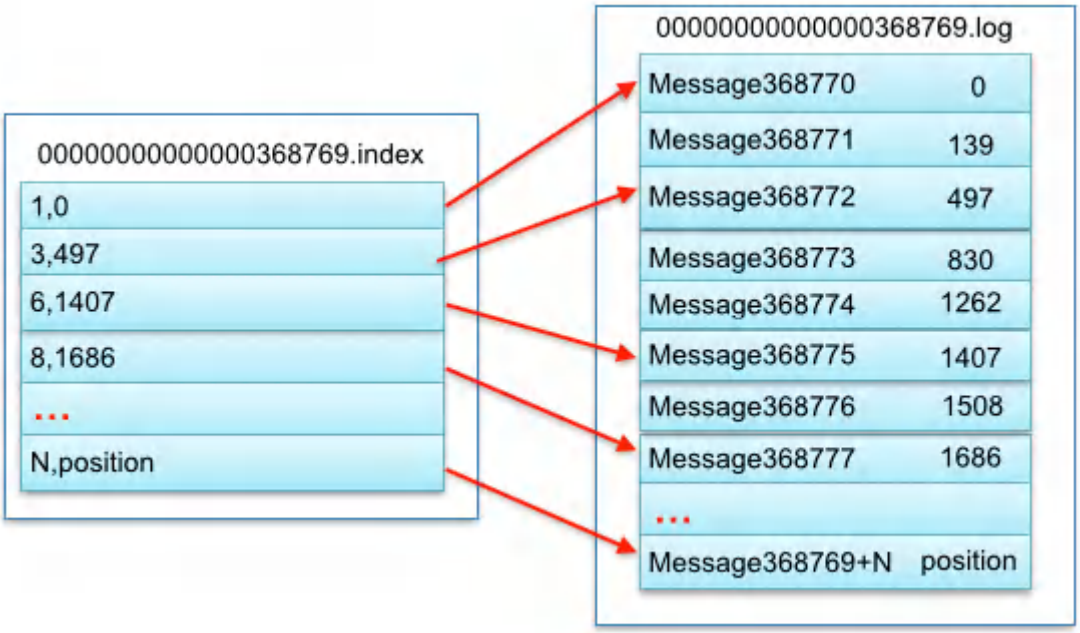
partition中的segment file的组成：

- segment file组成：由2部分组成，分别为index file和data file，这两个文件是一一对应的，后缀".index"和".log"分别表示索引文件和数据文件；
- segment file命名规则：partition的第一个segment从0开始，后续每个segment文件名为上一个segment文件最后一条消息的offset,ofsset的数值最大为64位（long类型），20位数字字符长度，没有数字用0填充。如下图所示：



关于segment file中index与data file对应关系图，这里我们选用网上的一个图片，如下所示：

公众号：旧时光大数据

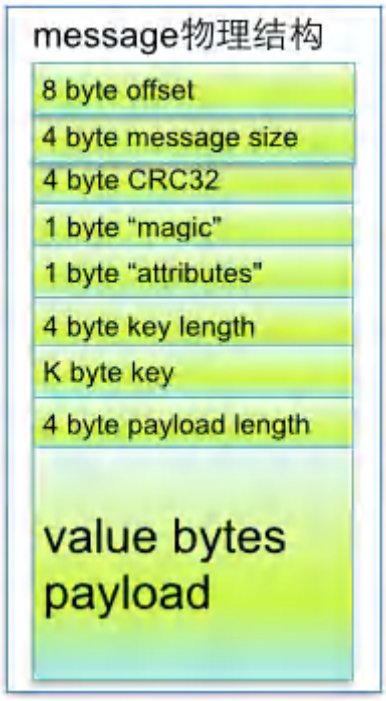


segment的索引文件中存储着大量的元数据，数据文件中存储着大量消息，索引文件中的元数据指向对应数据文件中的message的物理偏移地址。以索引文件中的 3，497 为例，在数据文件中表示第3个message（在全局partition表示第368772个message），以及该消息的物理偏移地址为497。

注：Partition中的每条message由offset来表示它在这个partition中的偏移量，这个offset并不是该Message在partition中实际存储位置，而是逻辑上的一个值（如上面的3），但它却唯一确定了partition中的一条Message（可以认为offset是partition中Message的id）。

2）message文件

message中的物理结构为：



参数说明：

公众号：旧时光大数据

关键字	解释说明
8 byte offset	在parition(分区)内的每条消息都有一个有序的id号，这个id号被称为偏移(offset),它可以唯一确定每条消息在parition(分区)内的位置。即offset表示partiion的第多少message
4 byte message size	message大小
4 byte CRC32	用crc32校验message
1 byte “magic”	表示本次发布Kafka服务程序协议版本号
1 byte “attributes”	表示为独立版本、或标识压缩类型、或编码类型
4 byte key length	表示key的长度,当key为-1时，K byte key字段不填
K byte key	可选
value bytes payload	表示实际消息数据

Kafka是如何清理过期文件的?

问过的一些公司：祖龙娱乐

参考答案：

Kafka将数据持久化到了硬盘上，我们可以配置一定的策略对数据进行清理，清理的策略有两个：**删除**和**压缩**。

数据清理方式一：删除

启用删除策略：

```
log.cleanup.policy=delete
```

直接删除，删除后的消息不可恢复。可配置以下两个策略：

清理超过指定时间清理：

```
log.retention.hours=16
```

超过指定大小后，删除旧的消息：

```
log.retention.bytes=1073741824
```

为了避免在删除时阻塞读操作，采用了copy-on-write形式的实现，删除操作进行时，读取操作的二分查找功能实际是在一个静态的快照副本上进行的，这类似于Java的CopyOnWriteArrayList。

数据清理方式二：压缩

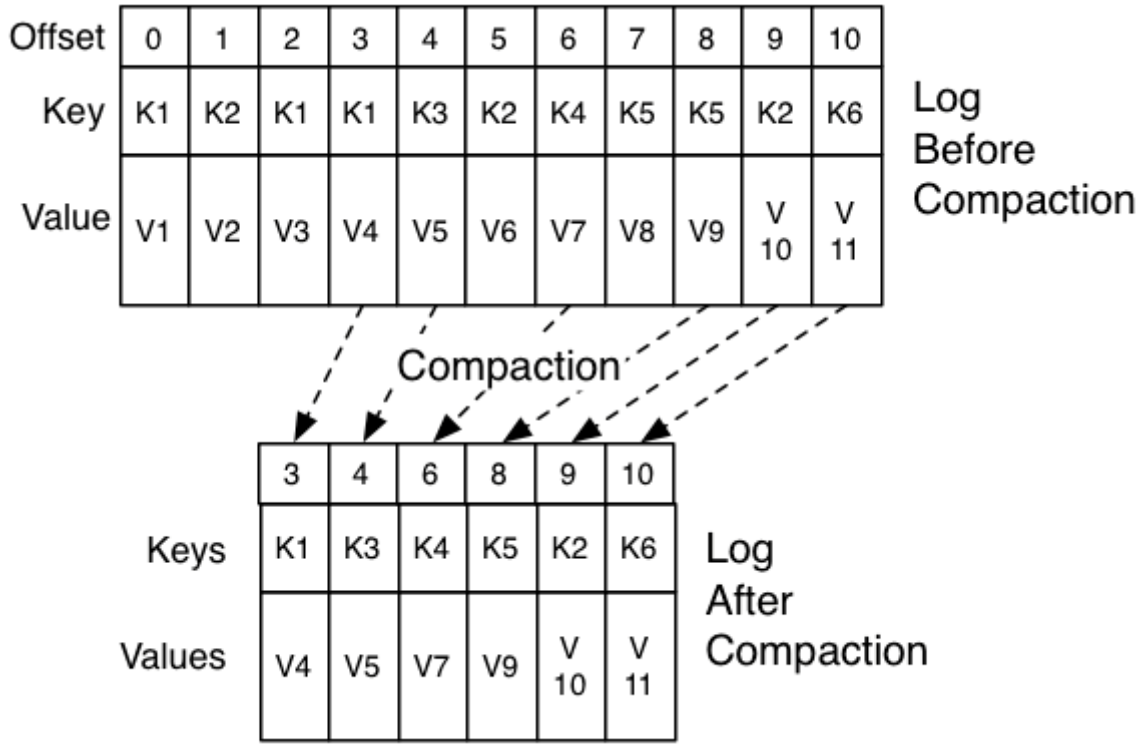
公众号：旧时光大数据

将数据压缩，只保留每个key最后一个版本的数据。

首先在broker的配置中设置`log.cleaner.enable=true`启用cleaner，这个默认是关闭的。

在topic的配置中设置`log.cleanup.policy=compact`启用压缩策略。

压缩策略的细节



如上图，在整个数据流中，每个Key都有可能出现多次，压缩时将根据Key将消息聚合，只保留最后一次出现时的数据。这样，无论什么时候消费消息，都能拿到每个Key的最新版本的数据。压缩后的offset可能是不连续的，比如上图中没有5和7，因为这些offset的消息被merge了，当从这些offset消费消息时，将会拿到比这个offset大的offset对应的消息，比如，当试图获取offset为5的消息时，实际上会拿到offset为6的消息，并从这个位置开始消费。

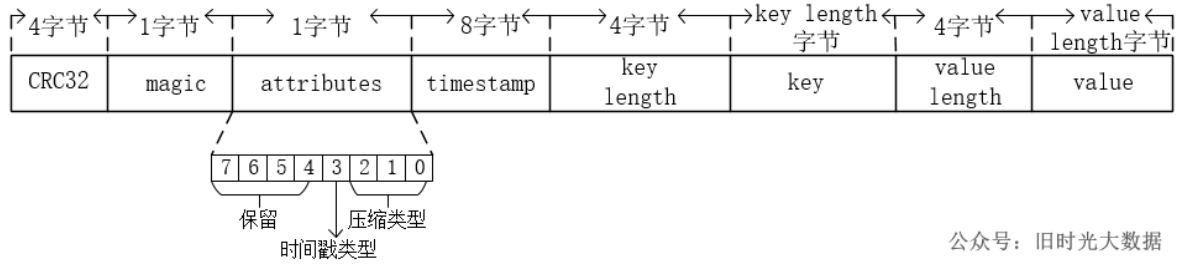
这种策略只适合特俗场景，比如消息的key是用户ID，消息体是用户的资料，通过这种压缩策略，整个消息集里就保存了所有用户最新的资料。

压缩策略支持删除，当某个Key的最新版本的消息没有内容时，这个Key将被删除，这也符合以上逻辑。

Kafka的一条message中包含了哪些信息?

问过的一些公司：祖龙娱乐

参考答案：



具体字段解释如下：

- CRC32：4个字节，消息的校验码。

公众号：旧时光大数据

- magic：1字节，魔数标识，与消息格式有关，取值为0或1。当magic为0时，消息的offset使用绝对offset且消息格式中没有timestamp部分；当magic为1时，消息的offset使用相对offset且消息格式中存在timestamp部分。所以，magic值不同，消息的长度是不同的。
- attributes： 1字节，消息的属性。其中第0~ 2位的组合表示消息使用的压缩类型，0表示无压缩，1表示gzip压缩，2表示snappy压缩，3表示lz4压缩。第3位表示时间戳类型，0表示创建时间，1表示追加时间。
- timestamp： 时间戳，其含义由attributes的第3位确定。
- key length：消息key的长度。
- key：消息的key。
- value length：消息的value长度。
- value：消息的内容

Kafka如何保证数据的Exactly Once?

问过的一些公司：360，美团，360社招，中信信用卡中心，Shopee(2021.08)

参考答案：

Kafka可以通过两种机制来确保消息消费的精确一次：

- 幂等性（Idempotence）
- 事务（Transaction）

1、幂等性：每个分区中精确一次且有序

将服务器的ACK级别设置为-1，可以保证Producer到Server之间不会丢失数据，即At Least Once语义。相对的，将服务器ACK级别设置为0，可以保证生产者每条消息只会被发送一次，即At Most Once语义。

At Least Once可以保证数据不丢失，但是不能保证数据不重复；相对的，At Least Once可以保证数据不重复，但是不能保证数据不丢失。但是，对于一些非常重要的信息，比如说交易数据，下游数据消费者要求数据既不重复也不丢失，即Exactly Once语义。在0.11版本以前的Kafka，对此是无能为力的，只能保证数据不丢失，再在下游消费者对数据做全局去重。对于多个下游应用的情况，每个都需要单独做全局去重，这就对性能造成了很大影响。

0.11版本的Kafka，引入了一项重大特性：**幂等性**。所谓的幂等性就是**指Producer不论向Server发送多少次重复数据，Server端都只会持久化一条。幂等性结合At Least Once语义，就构成了Kafka的Exactly Once语义**。即：

At Least Once + 幂等性 = Exactly Once

要启用幂等性，只需要将Producer的参数中enable.idompotence设置为true即可。Kafka的幂等性实现其实就是将原来下游需要做的去重放在了数据上游。开启幂等性的Producer在初始化的时候会被分配一个PID，发往同一Partition的消息会附带Sequence Number。而Broker端会对<PID, Partition, SeqNumber>做缓存，当具有相同主键的消息提交时，Broker只会持久化一条。

2、事务：跨分区原子写入

Kafka现在通过新的事务API支持跨分区原子写入。这将允许一个生产者发送一批到不同分区的消息，这些消息要么全部对任何一个消费者可见，要么对任何一个消费者都不可见。这个特性也允许你在一个事务中处理消费数据和提交消费偏移量，从而实现端到端的精确一次语义。下面是的代码片段演示了事务API的使用：

公众号：旧时光大数据

```
producer.initTransactions();
try {
  producer.beginTransaction();
  producer.send(record1);
  producer.send(record2);
  producer.commitTransaction();
} catch(ProducerFencedException e) {
  producer.close();
} catch(KafkaException e) {
  producer.abortTransaction();
}

```

上面的代码片段演示了你可以如何使用新生产者API来原子性地发送消息到topic的多个partition。值得注意的是，一个Kafka topic的分区中的消息，可以有些是在事务中，有些不在事务中。

因此在消费者方面，你有两种选择来读取事务性消息，通过隔离等级“isolation.level”消费者配置表示：

`read_commited`：除了读取不属于事务的消息之外，还可以读取事务提交后的消息。

`read_uncommited`：按照偏移位置读取所有消息，而不用等事务提交。这个选项类似Kafka消费者的当前语义。

为了使用事务，需要配置消费者使用正确的隔离等级，使用新版生产者，并且将生产者的“transactional.id”配置项设置为某个唯一ID。 需要此唯一ID来提供跨越应用程序重新启动的事务状态的连续性。

Kafka消费者怎么保证Exactly Once

问过的一些公司：快手 x 2

参考答案：

Consumer端丢失数据主要体现在：拉取了消息，并提交了消费位移，但是在消息处理结束之前突然发生了宕机等故障。消费者重生后，会从之前已提交的位移的下一个位置重新开始消费，之前未处理完成的消息不会再次处理，即相当于消费者丢失了消息。

解决Consumer端丢失消息的方法也很简单：将位移提交的时机改为消息处理完成后，确认消费完成了一批消息再提交相应的位移。这样做，即使处理消息的过程中发生了异常，由于没有提交位移，下次消费时还会从上次的位移处重新拉取消息，不会发生消息丢失的情况。

具体的实现方法为，Consumer在消费消息时，关闭自动提交位移，由应用程序手动提交位移。

Kafka监控实现？

问过的一些公司：腾讯

参考答案：

几种监控工具对比：

Kafka Manager：雅虎出品，可管理多个Kafka集群，是目前功能最全的管理工具。但是注意，当你的Topic太多，监控数据会占用你大量的带宽，造成你的机器负载增高。其监控功能偏弱，不满足需求。

Kafka Offset Monitor：程序一个jar包的形式运行，部署较为方便。只有监控功能，使用起来也较为安全。

公众号：旧时光大数据

Kafka Web Console：监控功能较为全面，可以预览消息，监控Offset、Lag等信息，不建议在生产环境中使用。

Burrow：是LinkedIn开源的一款专门监控consumer lag的框架。支持报警，只提供HTTP接口，没有webui。

Availability Monitor for Kafka：微软开源的Kafka可用性、延迟性的监控框架,提供JMX接口，用的很少。

Kafka中的数据能彻底删除吗?

问过的一些公司：腾讯

参考答案：

如果用kafka-topics.sh的delete命令删除topic，会有两种情况：

1）如果当前topic没有使用过即没有传输过信息：可以彻底删除。

2）如果当前topic有使用过即有过传输过信息：并没有真正删除topic只是把这个topic标记为删除（marked for deletion）。

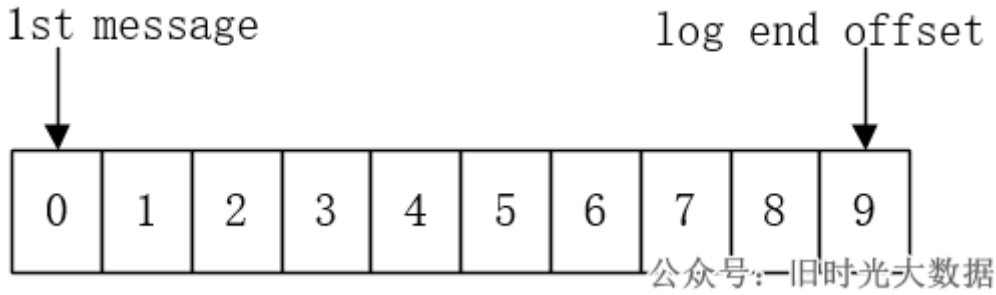
要彻底把情况2中的topic删除必须把kafka中与当前topic相关的数据目录和zookeeper与当前topic相关的路径一并删除。

Kafka复制机制?

问过的一些公司：腾讯

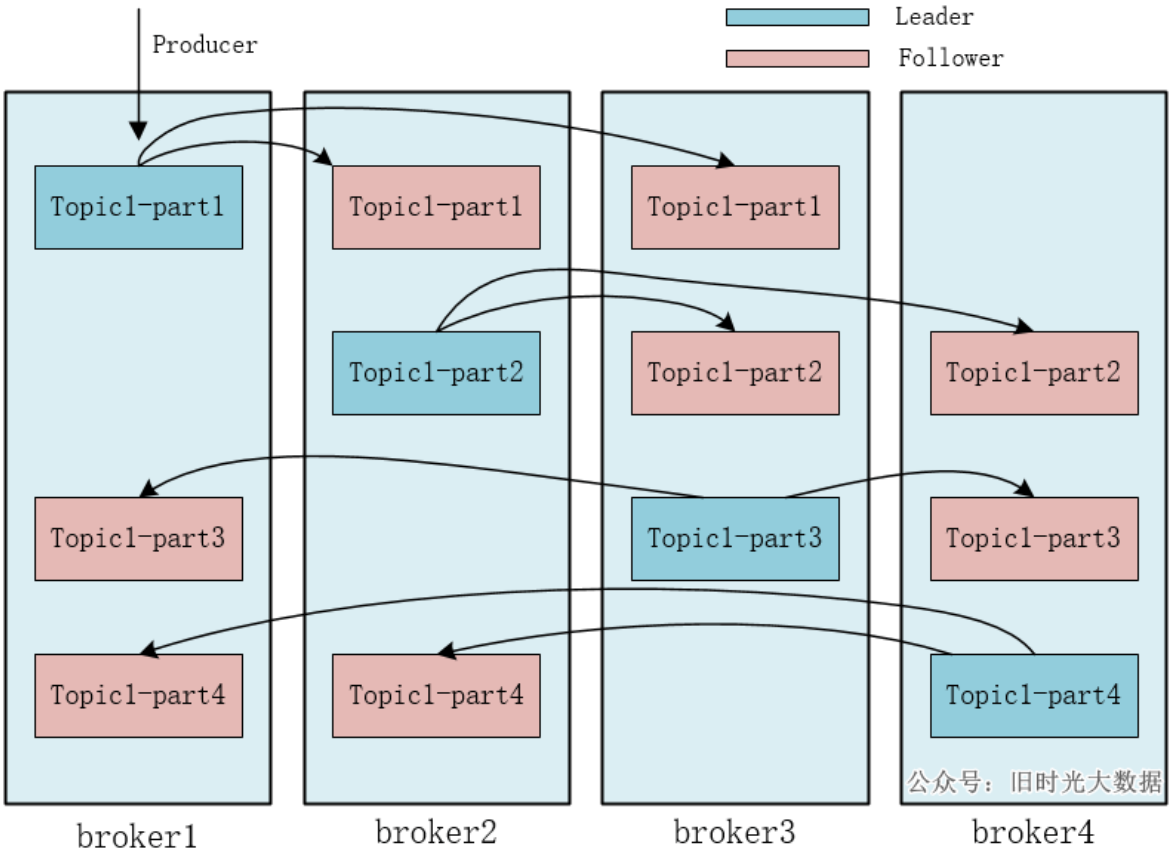
参考答案：

Kafka 主题中的每个分区都有一个预写日志（write-ahead log），我们写入 Kafka 的消息就存储在这里面。这里面的每条消息都有一个唯一的偏移量，用于标识它在当前分区日志中的位置。如下图所示：



Kafka 中的每个主题分区都被复制了 n 次，其中的 n 是主题的复制因子（replication factor）。这允许 Kafka 在集群服务器发生故障时自动切换到这些副本，以便在出现故障时消息仍然可用。**Kafka 的复制是以分区为粒度的，分区的预写日志被复制到 n 个服务器。** 在 n 个副本中，一个副本作为 leader，其他副本成为 followers。顾名思义，producer 只能往 leader 分区上写数据（读也只能从 leader 分区上进行），followers 只按顺序从 leader 上复制日志。

公众号：旧时光大数据



日志复制算法（log replication algorithm）必须提供的基本保证是，如果它告诉客户端消息已被提交，而当前 leader 出现故障，新选出的 leader 也必须具有该消息。在出现故障时，Kafka 会从挂掉 leader 的 ISR 里面选择一个 follower 作为这个分区新的 leader；换句话说，是因为这个 follower 是跟上 leader 写进度的。

每个分区的 leader 会维护一个 in-sync replica（同步副本列表，又称 ISR）。当 producer 往 broker 发送消息，消息先写入到对应 leader 分区上，然后复制到这个分区的所有副本中。只有将消息成功复制到所有同步副本（ISR）后，这条消息才算被提交。由于消息复制延迟受到最慢同步副本的限制，因此快速检测慢副本并将其从 ISR 中删除非常重要。

Kafka分区多副本机制?

可回答：1）Kafka副本机制；2）Kafka副本同步策略；3）Kafka副本间怎么同步的

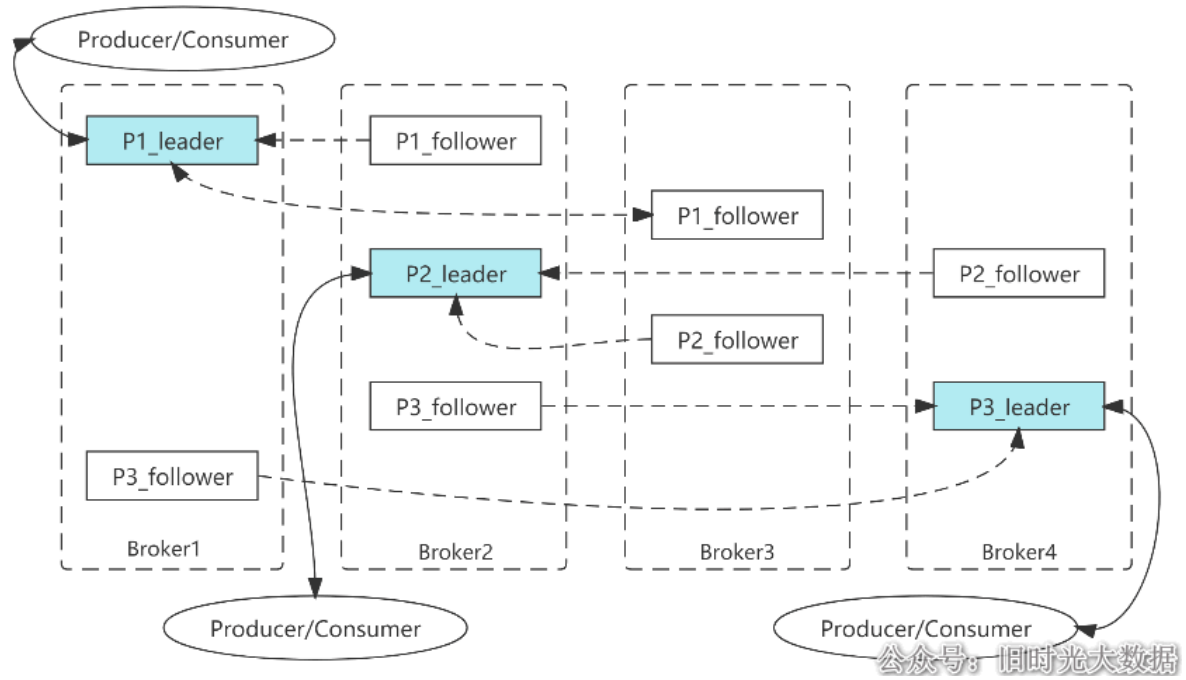
问过的一些公司：字节，京东，恒生电子，Shopee(2021.08)，soul(2021.09)

参考答案：

Kafka为分区引入了多副本（Replica）机制，通过增加副本数量可以提升容灾能力。同一分区的不同副本中保存的是相同消息（在同一时刻，副本之前并非完全一样），副本之间是“一主多从”的关系，其中leader副本负责处理读写请求，follower副本只负责与leader副本的消息同步。副本处于不同的broker中，当leader副本出现故障时，从follower副本中重新选举新的leader副本对外提供服务。Kafka通过多副本机制实现了故障的自动转义，当Kafka集群中某个broker失效时扔然能够保证服务可用。

如下图，Kfaka集群中有4个broker，某个主题中有3个分区，且副本因子（即副本个数）也为3，如此每个分区便有1个leader副本和2个follower副本。生产者和消费者只与leader副本进行交互，而follower副本只负责消息的同步，很多时候follower副本中的消息相对leader副本而言会有一定的滞后。

公众号：旧时光大数据



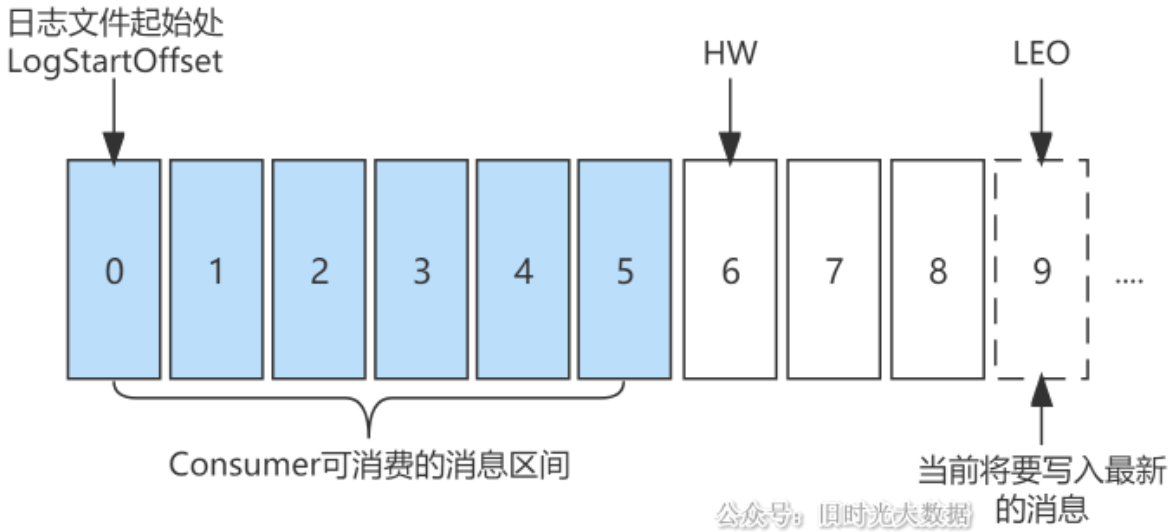
Kafka消费端也具备一定的容灾能力。Consumer使用拉（Pull）模式从服务端拉去消息，并且保存消费的具体位置，当消费者宕机后恢复上线时可以根据之前保存的消费位置重新拉取需要的消息进行消费，这样就不会造成消息丢失。

分区中的所有副本统称为 AR (Assigned Replicas) 。所有与 leader 副本保持一定程度同步的副本（包括 leader 副本在内〕组成 ISR On-Sync Replicas)，ISR集合是 AR 集合中的一个子集 。 消息会先发送到 leadr 副本，然后follower副本才能从 leader 副本中拉取消息进行同步，同步期间内 follower 副本相对于 leader 副本而言会有一定程度的滞后。前面所说的“一定程度的同步”是指可忍受的滞后范围，这个范围可以通过参数进行配置。与 leader 副本同步滞后过多的副本（不包括 leader 副本）组成 OSR (Out-of-Sync Replicas) ，由此可见，AR=ISR+OSR。在正常情况下， 所有的 follower 副本都应该与 leader 副本保持一定程度 的同步，即 AR=ISR，OSR 集合为空。

leader 副本负责维护和跟踪 ISR 集合中所有 follower 副 本 的滞后状态， 当 follower 副本落后太多或失效时， leader 副本会把它从 ISR 集合中剔除 。 如果 OSR 集合中有 follower 副本 “追上”了 leader 副本，那么 leader 副本会把它从 OSR 集合转移至 ISR 集合 。 默认情况下， 当 leader 副本发生故障时，只 有在 ISR 集合中的副本才有资格被选举为新的 leader， 而在 OSR 集合中的副本则没有任何机会（不过这个原则也可以通过修改相应的参数配置来改变）。

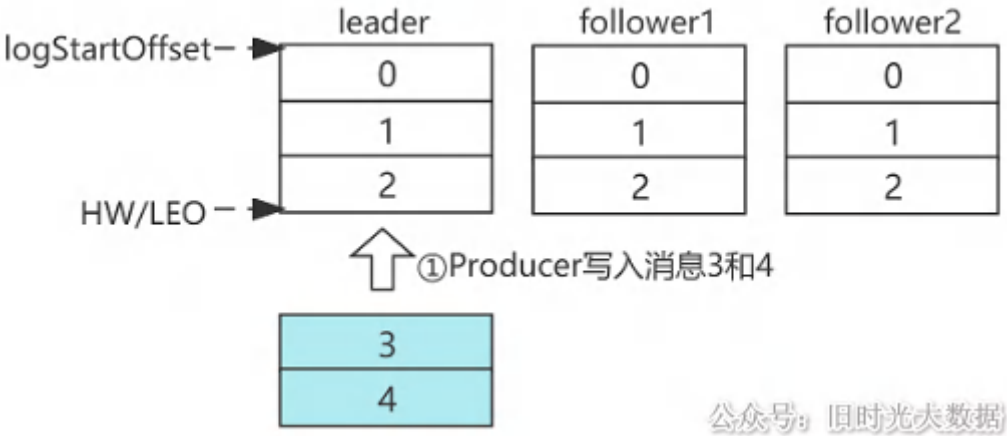
ISR 与 HW 和 LEO 也有紧密的关系 。 HW 是 High Watermark 的缩写，俗称高水位，它标识了一个特定的消息偏移量 (offset) ，消费者只能拉取到这个offset之前的消息 。如图所示，它代表一个日志文件，这个日志文件中有 9 条消息，第一条消息的 offset(LogStartOffset) 为 0，最后一条消息的 offset 为 8，offset 为 9 的消息用虚线框表示，代表下一条待写入的消息。日志文件的 HW 为 6，表示消费者只能拉取到 offset 在 0 至 5 之间的消息，而offset为 6 的消息对消费者而言是不可见的 。

公众号：旧时光大数据

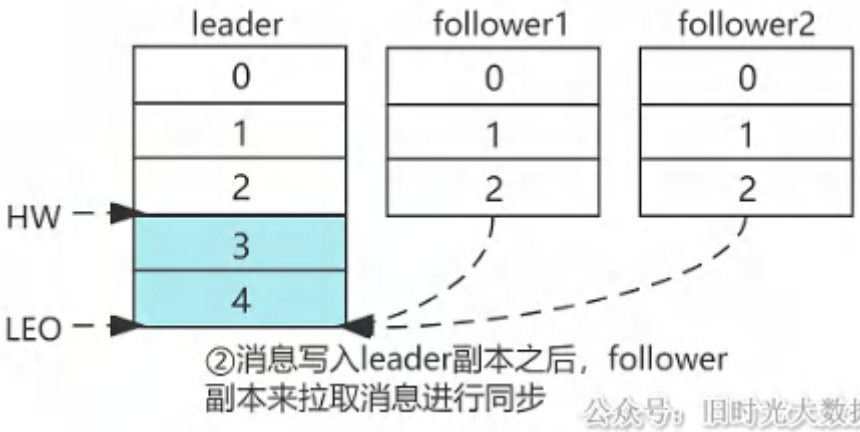


LEO是Log End Offset 的缩写，它标识当前日志文件中下一条待写入消息 的 offset，图中offset 为 9 的位置即为当前日志文件的LEO, LEO的大小相当于当前日志分区中最后一条消息的offset 值加 1。分区ISR 集合中的每个副本都会维护自身的LEO ， 而 ISR集合中最小的 LEO即为分区的LEO ， 对消费者而言只能消费HW之前的消息。

为了让读者更好地理解 ISR 集合， 以及 HW 和 LEO 之间的关系， 下面通过一个简单的示例来进行相关的说明 。 如下图所示，假设某个分区的 ISR 集合中有 3 个副本，即一个 leader副本和 2 个 follower 副本，此时分区的 LEO 和 HW 都为 3 。

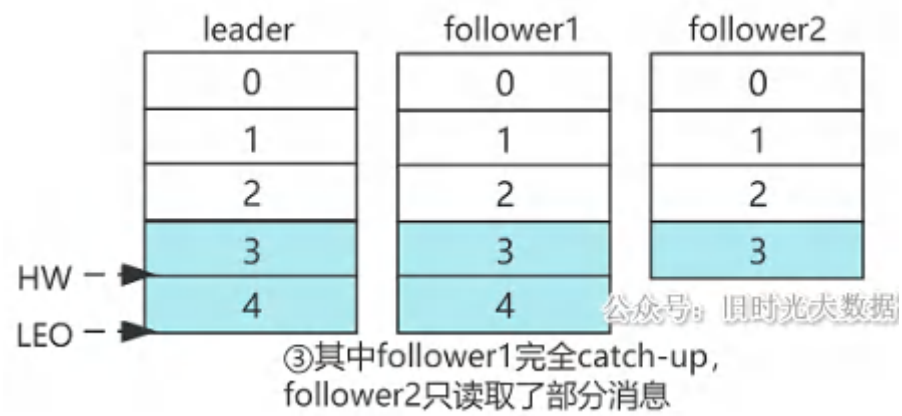


消息3和消息4从生产者发出之后会被先存入leader副本，如下图所示

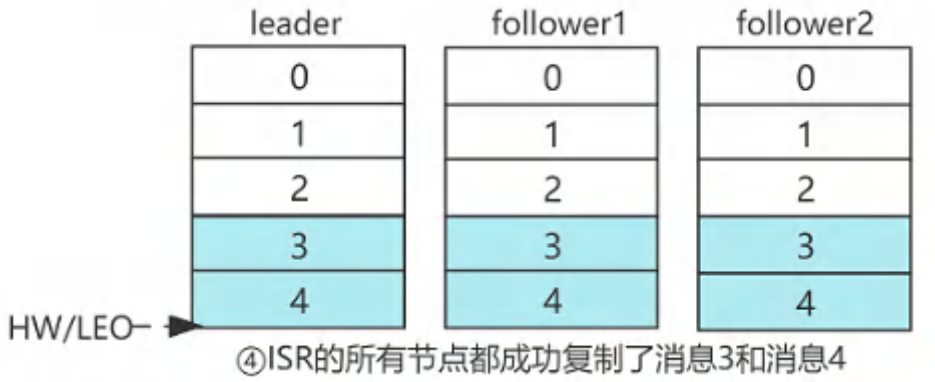


在消息写入 leader 副本之后， follower 副本会发送拉取请求来拉取消息 3 和消息 4 以进行消息同步。在同步过程中，不同的 follow副本的同步效率也不尽相同。如下图所示， 在某一时刻follower1完全跟上了 leader 副本而 follower2 只同步了消息 3 ， 如此 leader 副本的 LEO 为 5,follower1 的 LEO 为 5 , follower2 的 LEO 为 4 ， 那么当前分区的 HW 取最小值 4 ， 此时消费者可以消 费到 offset 为 0 至 3 之间的消息。

公众号：旧时光大数据



写入消息情形如下图所示，所有的副本都成功写入了消息 3 和消息 4，整个分区的HW 和 LEO 都变为 5，因此消费者可以消费到 offset 为 4 的消息了 。



由此可见， Kafka 的复制机制既不是完全的同步复制，也不是单纯的异步复制。事实上，同步复制要求所有能工作的 follower 副本都复制完，这条消息才会被确认为已成功提交，这种复制方式极大地影响了性能。而在异步复制方式下， follower 副本异步地从 leader 副本中 复制数据，数据只要被 leader 副本写入就被认为已经成功提交。在这种情况下，如果 follower 副本都还没有复制完而落后于 leader 副本，突然 leader 副本着机，则会造成数据丢失。 Kafka 使用的这种 ISR 的方式则有效地权衡了数据可靠性和性能之间的关系。

Kafka分区分配算法

可回答：Kafka的partition分区策略

问过的一些公司：阿里云，小米

参考答案：

1、生产者分区分配策略

生产者在将消息发送到某个Topic ，需要经过拦截器、序列化器和分区器（`Partitioner`）的一系列作用之后才能发送到对应的Broker，在发往Broker之前是需要确定它所发往的分区。

- 如果消息 `ProducerRecord` 指定了partition字段，那么就不需要分区器。
- 如果消息 `ProducerRecord` 没有指定partition字段，那么就需要依赖分区器，根据key这个字段来计算partition的值。**分区器的作用就是为消息分配分区。**

公众号：旧时光大数据

```java
public class ProducerRecord<K, V> {
    // 该消息需要发往的主题
    private final String topic;
    // 该消息需要发往的主题中的某个分区，如果该字段有值，则分区器不起作用，直接发往指定的分区
    // 如果该值为null，则利用分区器进行分区的选择
    private final Integer partition;
    private final Headers headers;
    // 如果partition字段为null，则使用分区器进行分区选择时会用到该key字段，该值可为空
    private final K key;
    private final V value;
    private final Long timestamp;

```

Kafka 中提供的默认分区器是 `DefaultPartitioner`，它实现了Partitioner接口（用户可以实现这个接口来自定义分区器），其中的partition方法就是用来实现**具体的分区分配逻辑**：

- **如果在发消息的时候指定了分区，则消息投递到指定的分区。**
- **如果没有指定分区，但是消息的key不为空，则使用称之为 `murmur` 的Hash算法（非加密型Hash函数，具备高运算性能及低碰撞率）来计算分区分配。**
- **如果既没有指定分区，且消息的key也是空，则用轮询的方式选择一个分区。**

```java
public class DefaultPartitioner implements Partitioner {

    private final ConcurrentMap<String, AtomicInteger> topicCounterMap = new
ConcurrentHashMap<>();

    public int partition(String topic, Object key, byte[] keyBytes, Object
value, byte[] valueBytes, Cluster cluster) {
        // 首先通过cluster从元数据中获取topic所有的分区信息
        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        // 拿到该topic的分区数
        int numPartitions = partitions.size();
        // 如果消息记录中没有指定key
        if (keyBytes == null) {
            // 则获取一个自增的值
            int nextValue = nextValue(topic);
            // 通过cluster拿到所有可用的分区（可用的分区这里指的是该分区存在首领副本）
            List<PartitionInfo> availablePartitions =
cluster.availablePartitionsForTopic(topic);
            // 如果该topic存在可用的分区
            if (availablePartitions.size() > 0) {
                // 那么将nextValue转成正数之后对可用分区数进行取余
                int part = Utils.toPositive(nextValue) %
availablePartitions.size();
                // 然后从可用分区中返回一个分区
                return availablePartitions.get(part).partition();
            } else { // 如果不存在可用的分区
                // 那么就从所有不可用的分区中通过取余的方式返回一个不可用的分区
                return Utils.toPositive(nextValue) % numPartitions;
            }
        } else { // 如果消息记录中指定了key
            // 则使用该key进行hash操作，然后对所有的分区数进行取余操作，这里的hash算法采用
的是murmur2算法，然后再转成正数
            //toPositive方法很简单，直接将给定的参数与0X7FFFFFFF进行逻辑与操作。
            return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;
        }
    }
```

公众号：旧时光大数据

```
32
33      // nextValue方法可以理解为是在消息记录中没有指定key的情况下，需要生成一个数用来代替key的
    hash值
34      // 方法就是最开始先生成一个随机数，之后在这个随机数的基础上每次请求时均进行+1的操作
35      private int nextValue(String topic) {
36          // 每个topic都对应着一个计数
37          AtomicInteger counter = topicCounterMap.get(topic);
38          if (null == counter) { // 如果是第一次，该topic还没有对应的计数
39              // 那么先生成一个随机数
40              counter = new AtomicInteger(ThreadLocalRandom.current().nextInt());
41              // 然后将该随机数与topic对应起来存入map中
42              AtomicInteger currentCounter = topicCounterMap.putIfAbsent(topic,
    counter);
43              if (currentCounter != null) {
44                  // 之后把这个随机数返回
45                  counter = currentCounter;
46              }
47          }
48          // 一旦存入了随机数之后，后续的请求均在该随机数的基础上+1之后进行返回
49          return counter.getAndIncrement();
50      }
51
```

2、消费者分区分配策略

消费者以组的名义订阅主题，主题有多个分区，消费者组中有多个消费者实例，同一时刻，**一条消息只能被组中的一个消费者实例消费。**

- **如果分区数大于或者等于组中的消费者实例数，一个消费者会负责多个分区。**
- **如果分区数小于组中的消费者实例数，有些消费者将处于空闲状态并且无法接收消息。**

如果多个消费者负责同一个分区，那么就意味着两个消费者同时读取分区的消息，由于消费者自己可以控制读取消息的Offset，就有可能C1才读到2，而C1读到1，C1还没处理完，C2已经读到3了，这就相当于多线程读取同一个消息，会造成消息处理的重复，且不能保证消息的顺序。

1）Range策略

`range` （默认分配策略）对应的实现类是`org.apache.kafka.clients.consumer.RangeAssignor`。

- 首先，将分区按数字顺序排行序，消费者按名称的字典序排序。
- 然后，用分区总数除以消费者总数。如果能够除尽，平均分配；若除不尽，则位于排序前面的消费者将多负责一个分区。

假设，有1个主题、10个分区、3个消费者线程， 10 / 3 = 3，而且除不尽，那么消费者C1将会多消费一个分区，分配结果是：

- C1将消费T1主题的0、1、2、3分区。
- C2将消费T1主题的4、5、6分区。
- C3将消费T1主题的7、8、9分区。

假设，有11个分区，分配结果是：

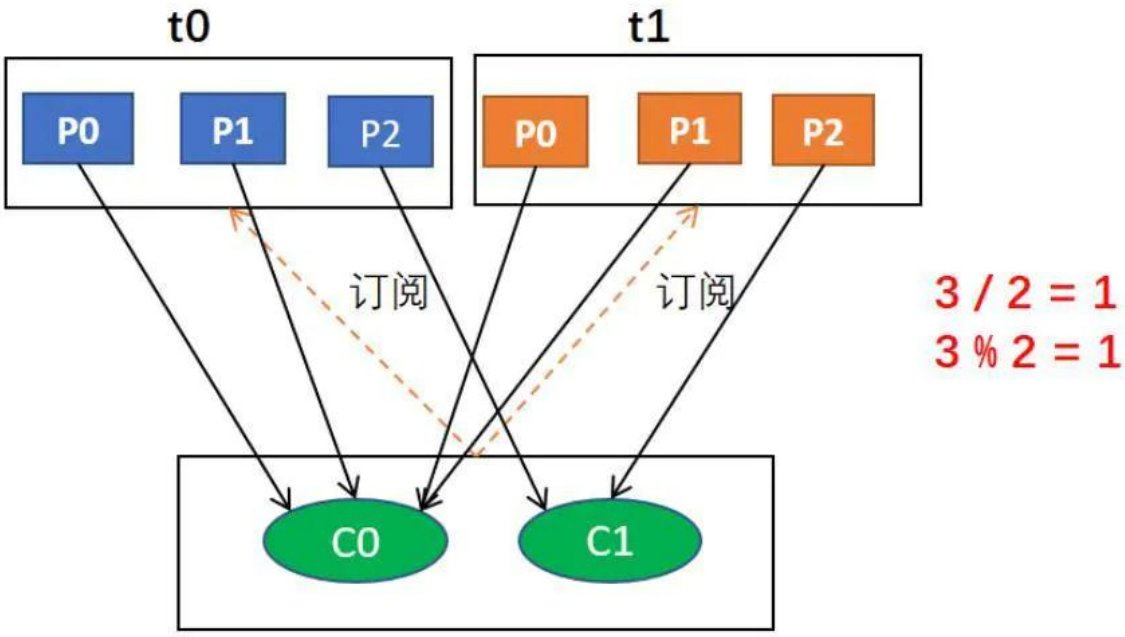
- C1将消费T1主题的0、1、2、3分区。
- C2将消费T1主题的4、5、 6、7分区。
- C2将消费T1主题的8、9、10分区。

假如，有2个主题（T0和T1），分别有3个分区，分配结果是：

- C1将消费T1主题的 0、1 分区，以及T1主题的 0、1 分区。

公众号：旧时光大数据

- C2将消费T1主题的2、3分区，以及T2主题的2、3分区。




```
public Map<String, List<TopicPartition>> assign(Map<String, Integer>
partitionsPerTopic,Map<String, Subscription> subscriptions) {
    // 主题与消费者的映射

    Map<String, List<String>> consumersPerTopic =
consumersPerTopic(subscriptions);
    Map<String, List<TopicPartition>> assignment = new HashMap<>();
    for (String memberId : subscriptions.keySet())
        assignment.put(memberId, new ArrayList<TopicPartition>());

    for (Map.Entry<String, List<String>> topicEntry :
consumersPerTopic.entrySet()) {
        String topic = topicEntry.getKey();    //  主题
        List<String> consumersForTopic = topicEntry.getValue();    //  消费者列表

        // partitionsPerTopic表示主题和分区数的映射
        // 获取主题下有多少个分区
        Integer numPartitionsForTopic = partitionsPerTopic.get(topic);
        if (numPartitionsForTopic == null)
            continue;

        // 消费者按字典序排序
        Collections.sort(consumersForTopic);

        // 分区数量除以消费者数量
        int numPartitionsPerConsumer = numPartitionsForTopic /
consumersForTopic.size();
        // 取模，余数就是额外的分区
        int consumersWithExtraPartition = numPartitionsForTopic %
consumersForTopic.size();

        List<TopicPartition> partitions =
AbstractPartitionAssignor.partitions(topic, numPartitionsForTopic);
        for (int i = 0, n = consumersForTopic.size(); i < n; i++) {
```

公众号：旧时光大数据

```
28              int start = numPartitionsPerConsumer * i + Math.min(i,
    consumersWithExtraPartition);
29              int length = numPartitionsPerConsumer + (i + 1 >
    consumersWithExtraPartition ? 0 : 1);
30              // 分配分区
31
     assignment.get(consumersForTopic.get(i)).addAll(partitions.subList(start, start
    + length));
32          }
33      }
34      return assignment;
35  }
36
```

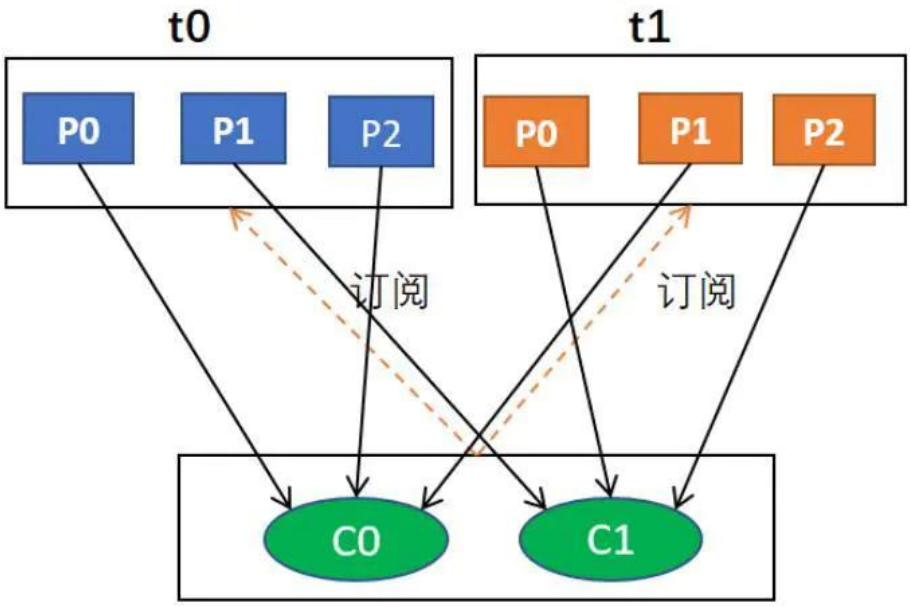
2）RoundRobin策略

RoundRobin基于轮询算法，对应的实现类是 `org.apache.kafka.clients.consumer.RoundRobinAssignor`

- 首先，将所有主题的分区组成 `TopicAndPartition` 列表。
- 然后对TopicAndPartition列表按照hashCode进行排序某个 topic。

假设，有两个消费者C0和C1，两个主题T0和T1，每个主题有3个分区，分配结果是：

- C0将消费T0主题的0、2分区，以及T1主题的1分区。
- C1将消费T0主题的1分区，以及T1主题的0、2分区。

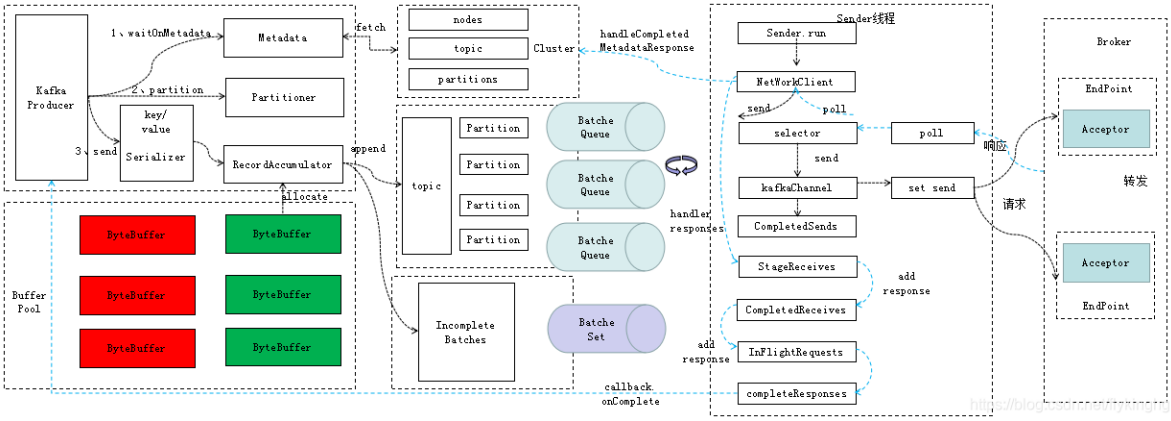


Kafka蓄水池机制

问过的一些公司：深信服

参考答案：

公众号：旧时光大数据



从上面的架构图可以看出，生产的流程主要就是一个producer线程和一个sender线程，它们之间通过BatchQueue来获取数据，它们的关系是一一对应的，所以kafka的生产过程都是异步过程，它的同步和异步指的是接收响应结果的模式是同步阻塞还是异步回调。同步和异步的生产者调用示例如下：

异步生产模式：

```
producer.send(new ProducerRecord<>(topic,messageNo,messageStr), new
DemoCallBack(startTime, messageNo, messageStr));

```

同步生产模式：

```
producer.send(new ProducerRecord<>(topic,messageNo,messageStr)).get();

```

同步接收是依据send之后返回Future，再调用Future的get方法进行阻塞等待。下面我们就从producer和sender两个类所对应的流程来进行分析，他们分别是消息收集过程和消息发送过程。从上面的架构图我们可以看到这个过程的数据最终是放在BatchQueue，像是将水流入了一个蓄水池的场景，这就是称其为”蓄水池”的含义了。

Kafka如何实现幂等性?

问过的一些公司：深信服

参考答案：

在之前的旧版本中，Kafka只能支持两种语义：At most once和At least once。At most once保证消息不会朝服，但是可能会丢失。在实践中，很有有业务会选择这种方式。At least once保证消息不会丢失，但是可能会重复，业务在处理消息需要进行去重。

Kafka在0.11.0.0版本支持增加了对幂等的支持。幂等是针对生产者角度的特性。**幂等可以保证上生产者发送的消息，不会丢失，而且不会重复。**

1、如何实现幂等

HTTP/1.1中对幂等性的定义是：一次和多次请求某一个资源对于资源本身应该具有同样的结果（网络超时等问题除外）。也就是说，其任意多次执行对资源本身所产生的影响均与一次执行的影响相同

实现幂等的关键点就是服务端可以区分请求是否重复，过滤掉重复的请求。要区分请求是否重复的有两点：

- 唯一标识：要想区分请求是否重复，请求中就得有唯一标识。例如支付请求中，订单号就是唯一标识。
- 记录下已处理过的请求标识：光有唯一标识还不够，还需要记录下那些请求是已经处理过的，这样当收到新的请求时，用新请求中的标识和处理记录进行比较，如果处理记录中有相同的标识，说明

公众号：旧时光大数据

是重复交易，拒绝掉。

2、Kafka幂等性实现原理

为了实现Producer的幂等性，Kafka引入了Producer ID（即PID）和Sequence Number。

- PID。每个新的Producer在初始化的时候会被分配一个唯一的PID，这个PID对用户是不可见的。
- Sequence Numbler。（对于每个PID，该Producer发送数据的每个<Topic, Partition>都对应一个从0开始单调递增的Sequence Number

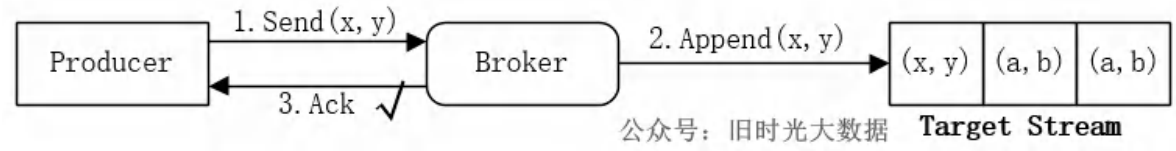
Kafka可能存在多个生产者，会同时产生消息，但对Kafka来说，只需要保证每个生产者内部的消息幂等就可以了，所有引入了PID来标识不同的生产者。

对于Kafka来说，要解决的是生产者发送消息的幂等问题。也即需要区分每条消息是否重复。Kafka通过为每条消息增加一个Sequence Numbler，通过Sequence Numbler来区分每条消息。每条消息对应一个分区，不同的分区产生的消息不可能重复。所有Sequence Numbler对应每个分区

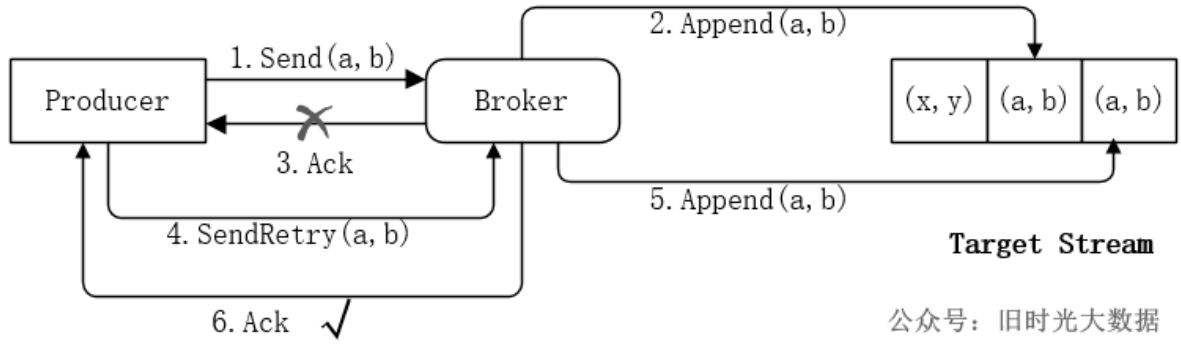
Broker端在缓存中保存了这seq number，对于接收的每条消息，如果其序号比Broker缓存中序号大于1则接受它，否则将其丢弃。这样就可以实现了消息重复提交了。但是，只能保证单个Producer对于同一个<Topic, Partition>的Exactly Once语义。不能保证同一个Producer一个topic不同的partion幂等。

3、实现幂等前后对比

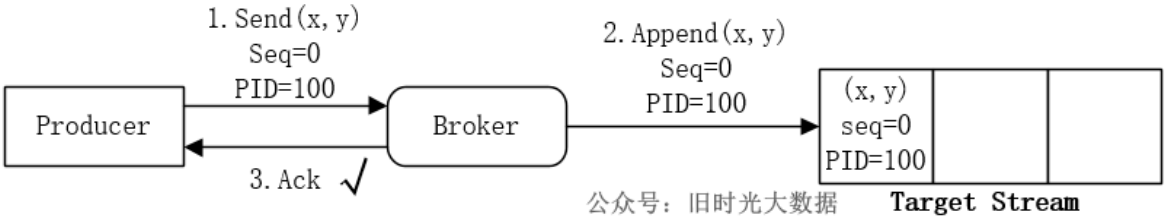
标准实现



发生重试时

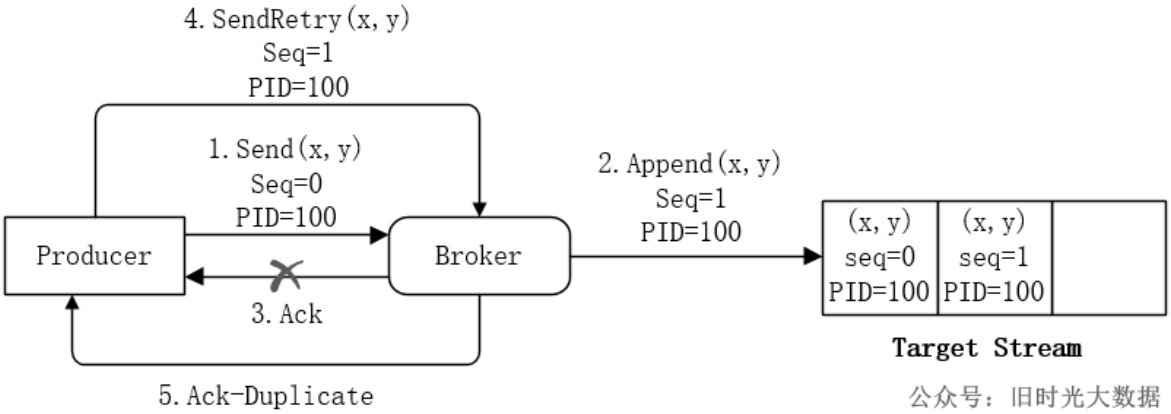


实现幂等之后



发生重试时

公众号：旧时光大数据



4、幂等性示例

生产者要使用幂等性很简单，只需要增加以下配置即可：

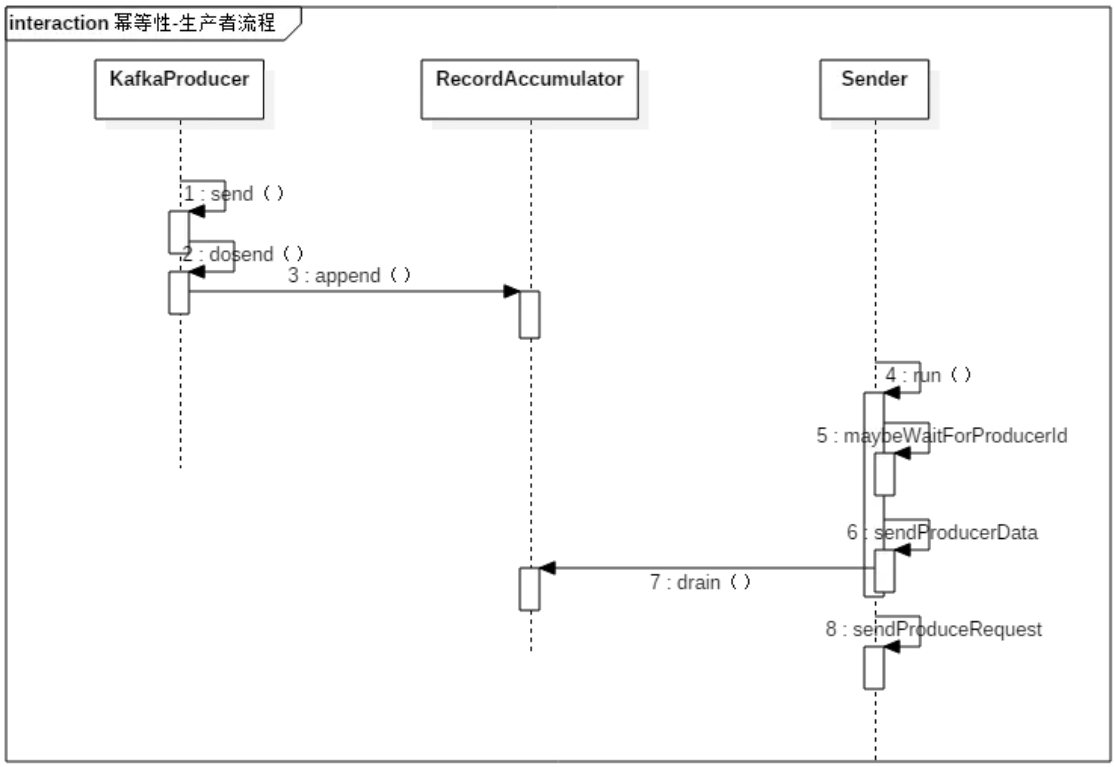
```
enable.idempotence=true

```

```
Properties props = new Properties();
props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "true");
props.put("acks", "all"); // 当 enable.idempotence 为 true，这里默认为 all
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

KafkaProducer producer = new KafkaProducer(props);

producer.send(new ProducerRecord(topic, "test");

```

Prodcuer 幂等性对外保留的接口非常简单，其底层的实现对上层应用做了很好的封装，应用层并不需要去关心具体的实现细节，对用户非常友好

5、幂等性实现生产者流程

此流程只展示了涉及生产者幂等性相关的重要操作



这里重点关注幂等性相关的内容，首先，KafkaProducer启动时，会初始化一个 TransactionManager 实例，它的作用有以下几个部分：

- 记录本地的事务状态（事务性时必须）
- 记录一些状态信息以保证幂等性，比如：每个 topic-partition 对应的下一个 sequence numbers 和 last acked batch（最近一个已经确认的 batch）的最大的 sequence number 等；
- 记录 ProducerIdAndEpoch 信息（PID 信息）。

幂等性时，Producer 的发送流程如下：

1）调用kafkaProducer的send方法将数据添加到 RecordAccumulator 中，添加时会判断是否需要新建一个 ProducerBatch，这时这个 ProducerBatch 还是没有 PID 和 sequence number 信息的；

2）Producer 后台发送线程 Sender，在 run() 方法中，会先根据 TransactionManager 的 shouldResetProducerStateAfterResolvingSequences() 方法判断当前的 PID 是否需要重置，重置的原因是因为：如果有topic-partition的batch已经超时还没处理完，此时可能会造成sequence number 不连续。因为sequence number 有部分已经分配出去了，而Kafka服务端没有收到这部分sequence number 的序号，Kafka服务端为了保证幂等性，只会接受同一个pid的sequence number 等于服务端缓存sequence number +1的消息，所有这时候需要重置Pid来保证幂等性。

```
1    synchronized boolean shouldResetProducerStateAfterResolvingSequences() {
2          /**
3           * 是否是事务即配置了Tid
4           * 如果是事务则不需重置Pid
5           */
6          if (isTransactional())
7              // We should not reset producer state if we are transactional. We
   will transition to a fatal error instead.
8              return false;
9          for (Iterator<TopicPartition> iter =
   partitionsWithUnresolvedSequences.iterator(); iter.hasNext(); ) {
10             TopicPartition topicPartition = iter.next();
11             if (!hasInflightBatches(topicPartition)) {//没有该分区的消息在发送中
```

公众号：旧时光大数据

```
                // The partition has been fully drained. At this point, the last ack'd sequence should be once less than
                // next sequence destined for the partition. If so, the partition is fully resolved. If not, we should
                // reset the sequence number if necessary.
                /**
                 * 判断SequenceNo是否连续
                 * 如果连续的，就不需要重置Pid
                 */
                if (isNextSequence(topicPartition, sequenceNumber(topicPartition))) {
                    // This would happen when a batch was expired, but subsequent batches succeeded.
                    iter.remove();
                } else {
                    // We would enter this branch if all in flight batches were ultimately expired in the producer.
                    log.info("No inflight batches remaining for {}, last ack'd sequence for partition is {}, next sequence is {}. " +
                            "Going to reset producer state.", topicPartition, lastAckedSequence(topicPartition), sequenceNumber(topicPartition));
                    return true;
                }
            }
        }
        return false;
    }

```

3）Sender线程调用maybeWaitForProducerId()方法判断是否要申请Pid，如果需要，会阻塞直到成功申请到Pid

```
ProducerIdAndEpoch producerIdAndEpoch = null;
boolean isTransactional = false;
if (transactionManager != null) {//有事务或者启用幂等
    //事务是否允许向此分区发送消息
    if (!transactionManager.isSendToPartitionAllowed(tp))
        break;
    producerIdAndEpoch = transactionManager.producerIdAndEpoch();
    if (!producerIdAndEpoch.isValid())
        // we cannot send the batch until we have refreshed the producer id
        break;
    //是否支持事务
    isTransactional = transactionManager.isTransactional();
    /**
     * 如果该分区的前面还有没发送完成的Batch，则需要跳过该分区的Batch，等待之前batch发送完成
     */
    if (!first.hasSequence() && transactionManager.hasUnresolvedSequence(first.topicPartition))

        break;

    /**
     * 该分区存在发送中的Batch，该Batch有Sequence，和first的不相等。则跳过、
     *
     * 也即first是个重试的Batch（因为它有Sequence），需要等待该分区发送中的Batch完成
     */
```

公众号：旧时光大数据

```
23      int firstInFlightSequence =
    transactionManager.firstInFlightSequence(first.topicPartition);

24      if (firstInFlightSequence != RecordBatch.NO_SEQUENCE && first.hasSequence()
25              && first.baseSequence() != firstInFlightSequence)
26          break;
27  }
28  ProducerBatch batch = deque.pollFirst();
29  /**
30   * 校验当前batch是否已经设置了Sequence
31   * 如果没有，则需要设置batch的Sequence，增加对应分区的Next Sequence，将batch加入到
    inflightBatchesBySequence中
32   */
33  if (producerIdAndEpoch != null && !batch.hasSequence()) {
34          //设置Batch的sequenceNumber 和isTransactional
35      batch.setProducerState(producerIdAndEpoch,
    transactionManager.sequenceNumber(batch.topicPartition), isTransactional);
36      //增加该分区的sequenceNumber，增加值为Batch中消息的个数
37      transactionManager.incrementSequenceNumber(batch.topicPartition,
    batch.recordCount);
38      log.debug("Assigned producerId {} and producerEpoch {} to batch with base
    sequence " +
39                      "{} being sent to partition {}",
    producerIdAndEpoch.producerId,
40              producerIdAndEpoch.epoch, batch.baseSequence(), tp);
41      //加入到发送队列中
42      transactionManager.addInFlightBatch(batch);
43  }
44  batch.close();//关闭此batch，不可追加消息
45  size += batch.records().sizeInBytes();//累计size
46  ready.add(batch);//加到集合中，最后一起返回出去
47  batch.drained(now);//更新drainedMs时间戳
48
```

5）最后调用sendProduceRequest方法将消息发送出去

Kafka的offset存在哪?

问过的一些公司：美团

参考答案：

kafka消费者在会保存其消费的进度，也就是offset，存储的位置根据选用的`kafka-api`不同而不同。

1、Zookeeper存储

首先来说说消费者如果是根据java-api来消费，也就是`kafka.javaapi.consumer.ConsumerConnector`，通过配置参数zookeeper.connect来消费。这种情况下，消费者的offset会更新到zookeeper的`consumers/{group}/offsets/{topic}/{partition}`目录下，例如：

```
1   [zk: 100.5.14.161:2181(CONNECTED) 20] get /consumers/console-consumer-
    21030/offsets/topic_test_1/0
2   37
3   cZxid = 0x50000053a
4   ctime = Thu Dec 20 17:26:58 CST 2018
5   mZxid = 0x50000053a
```

公众号：旧时光大数据

```
  mtime = Thu Dec 20 17:26:58 CST 2018
  pZxid = 0x50000053a
  cversion = 0
  dataVersion = 0
  aclVersion = 0
  ephemeralOwner = 0x0
  dataLength = 2
  numChildren = 0

```

2、kafka topic存储

如果是根据kafka默认的api来消费，即`org.apache.kafka.clients.consumer.KafkaConsumer`，我们会配置参数`bootstrap.servers`来消费。而其消费者的offset会更新到一个kafka自带的topic：`__consumer_offsets`下面，查看当前group的消费进度，则要依靠kafka自带的工具`kafka-consumer-offset-checker`，例如：

```
WARN WARNING: ConsumerOffsetChecker is deprecated and will be dropped in releases following 0.9.0. Use ConsumerGroupCommand instead.
(kafka.tools.ConsumerOffsetChecker$)
Group           Topic                Pid Offset      logSize         Lag             Owner
test_group_1    topic_test_1          0    489        491             2               none
test_group_1    topic_test_1          1    489        490             1               none
test_group_1    topic_test_1          2    490        491             1               none

```

顺带说一下offset更新方式

offset更新的方式，不区分是用的哪种api，大致分为两类：

- 自动提交，设置enable.auto.commit=true，更新的频率根据参数【auto.commit.interval.ms】来定。这种方式也被称为【at most once】，fetch到消息后就可以更新offset，无论是否消费成功。
- 手动提交，设置enable.auto.commit=false，这种方式称为【at least once】。fetch到消息后，等消费完成再调用方法【consumer.commitSync()】，手动更新offset；如果消费失败，则offset也不会更新，此条消息会被重复消费一次。

Kafka中如何保证数据一致性?

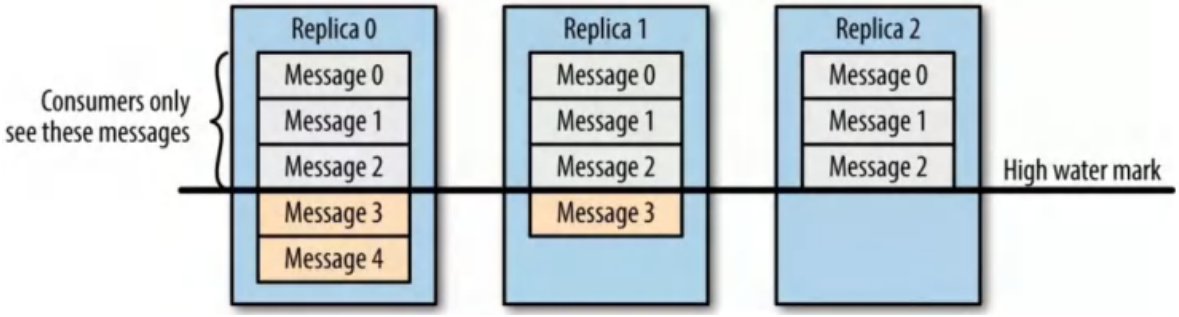
可回答：Kafka的一致性

问过的一些公司：字节，美团

参考答案：

不论是旧的Leader还是新选举产生的Leader，Consumer都能读到一样的数据，Kafka是通过引入HW（High Water Mark）机制来保证数据一致性。

公众号：旧时光大数据



假设分区的副本为3，其中副本0是Leader，副本1和副本2是follower，并且在ISR列表里面，虽然副本0已经写入了Message4，但是Consumer只能卖取到Message2。因为所有的ISR都同步了Message2，只有High Water Mark以上的消息才支持Consumer读取，而High Water Mark取决于ISR列表里面偏移量最小的分区，对应于上图的副本2，这个很类似于木桶原理。

这样做的原因是还没有被足够多副本复制的消息被认为是“不安全”的，如果 Leader 发生崩溃，另一个副本成为新Leader，那么这些消息很可能丢失了。如果我们允许消费者读取这些消息，可能就会破坏一致性。试想，一个消费者从当前Leader（副本0）读取并处理了Message4，这个时候Leader挂掉了，选举了副本1为新的Leader，这时候另一个消费者再去从新的Leader读取消息，发现这个消息其实并不存在，这就导致了数据不一致性的问题。

当然，引入了High Water Mark 机制，会导数Broker间的消息复制因为某些原因变慢，那么消息到达消费者的时间也会随之变长（因为我们会先等待消息复制完毕），延迟时间可以通过参数replica.lag.time.max.ms参数配置，它指定了副本在复制消息时可被允许的最大延迟时间。

Kafka新旧API区别

问过的一些公司：腾讯

参考答案：

1、高级API

优点：

- 高级API写起来简单
- 不需要去自行去管理offset，系统通过zookeeper自行管理
- 不需要管理分区，副本等情况，系统自动管理
- 消费者断线会自动根据上一次记录在 zookeeper中的offset去接着获取数据
- 可以使用group来区分对访问同一个topic的不同程序访问分离开来（不同的group记录不同的offset，这样不同程序读取同一个topic才不会因为offset互相影响）

缺点：

- 不能自行控制 offset（对于某些特殊需求来说）
- 不能细化控制如分区、副本、zk 等

2、低级API

优点：

- 能够开发者自己控制offset，想从哪里读取就从哪里读取
- 自行控制连接分区，对分区自定义进行负载均衡
- 对 zookeeper 的依赖性降低（如：offset 不一定非要靠 zk 存储，自行存储offset 即可，比如存在文件或者内存中）

缺点：

公众号：旧时光大数据

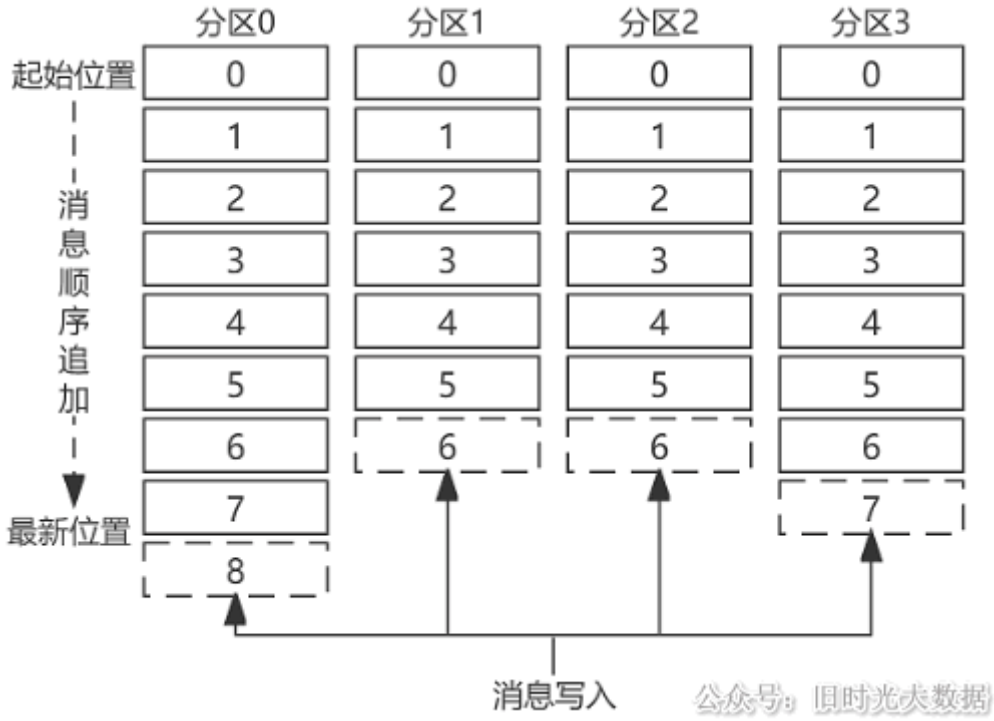
- 太过复杂，需要自行控制offset，连接哪个分区，找到分区leader等

Kafka消息在磁盘上的组织方式

问过的一些公司：字节

参考答案：

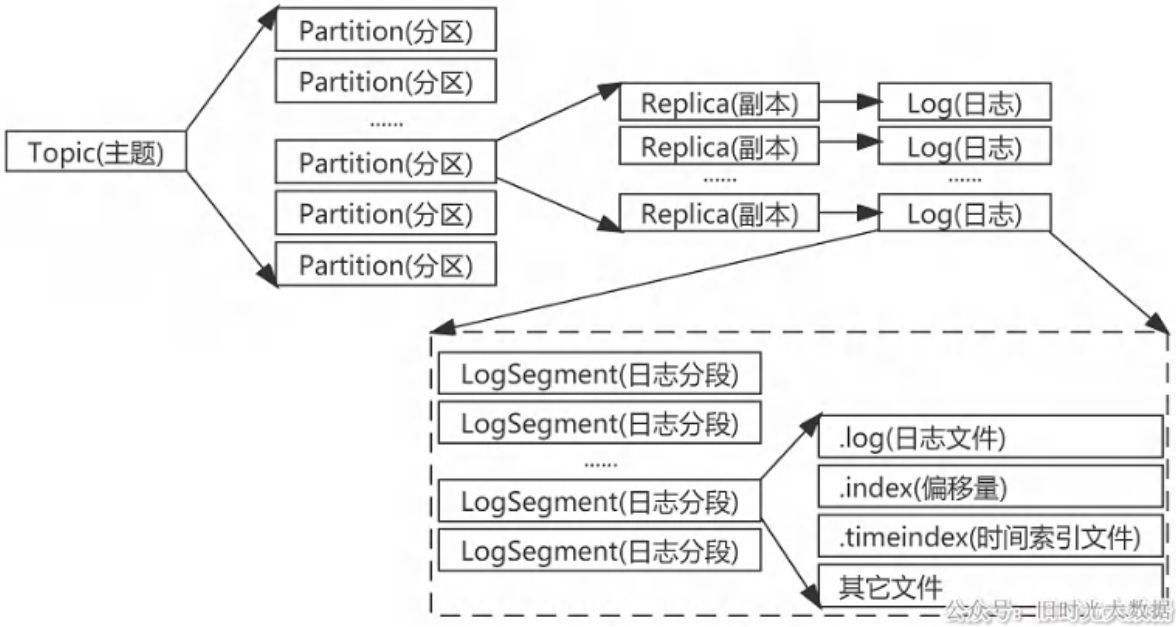
Kafka中的消息是以主题为基本单位进行归类的，各个主题在逻辑上相互独立。每个主题又可以分为一个或多个分区，分区的数量可以在主题创建的时候指定，也可以在之后修改。每条消息在发送的时候会根据分区规则被追加到指定的分区中，分区中的每条消息都会被分配一个唯一的序列号，也就是通常所说的偏移量（offset），具有4个分区的主题的逻辑结构见下图。



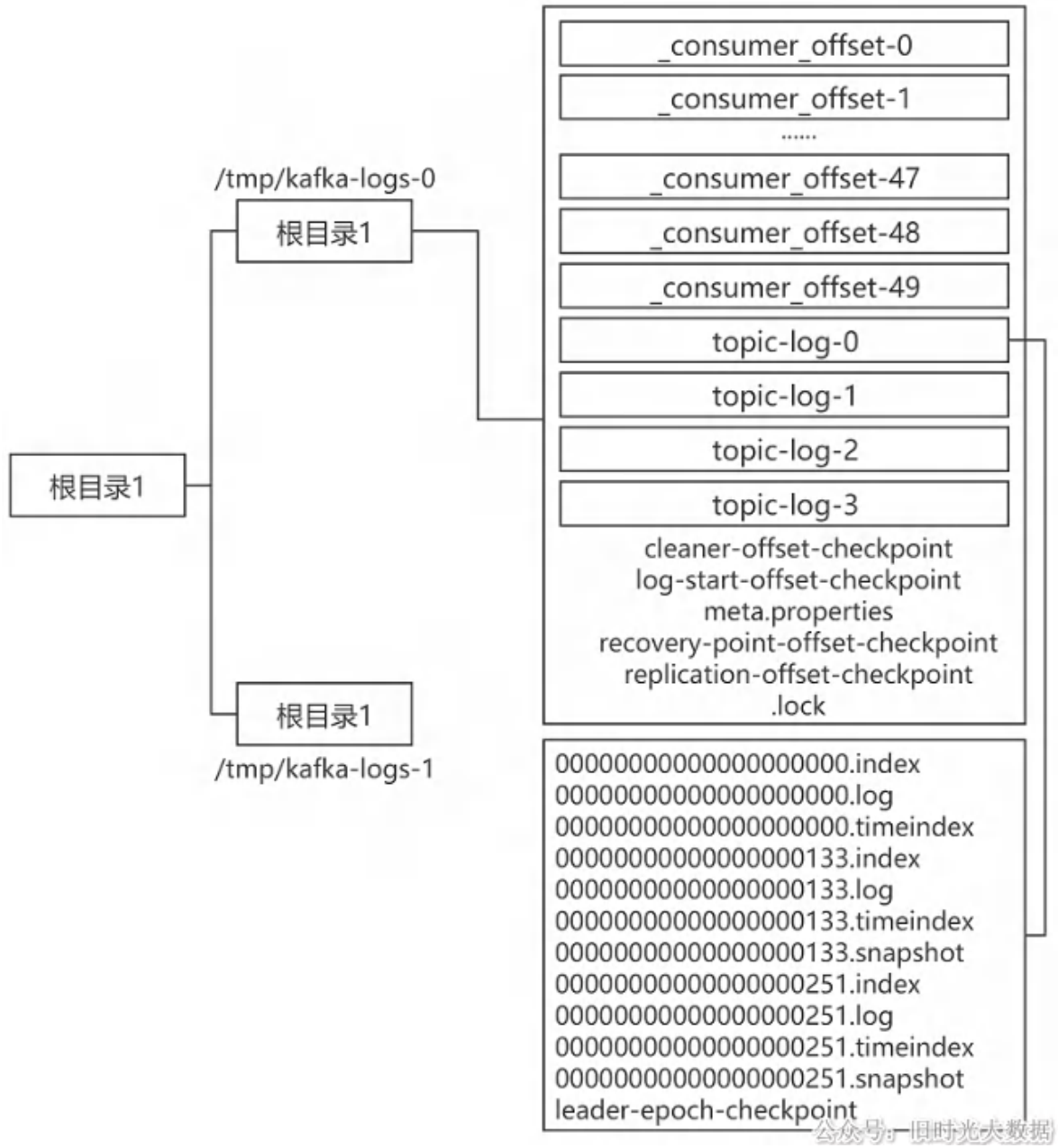
如果分区规则设置得合理，那么所有的消息可以均匀地分布到不同的分区中，这样就可以实现水平扩展。不考虑多副本的情况，一个分区对应一个日志（Log）。为了防止Log过大，Kafka又引入了日志分段（LogSegment）的概念，将Log切分为多个LogSegment，相当于一个巨型文件被平均分配为多个相对较小的文件，这样也便于消息的维护和清理。

事实上，Log和LogSegment也不是纯粹物理意义上的概念，Log在物理上只以文件夹的形式存储，而每个LogSegment对应于磁盘上的一个日志文件和两个索引文件，以及可能的其他文件（比如以“.txnindex”为后缀的事务索引文件）。下图描绘了主题、分区、副本、Log以及LogSegment之间的关系。

公众号：旧时光大数据



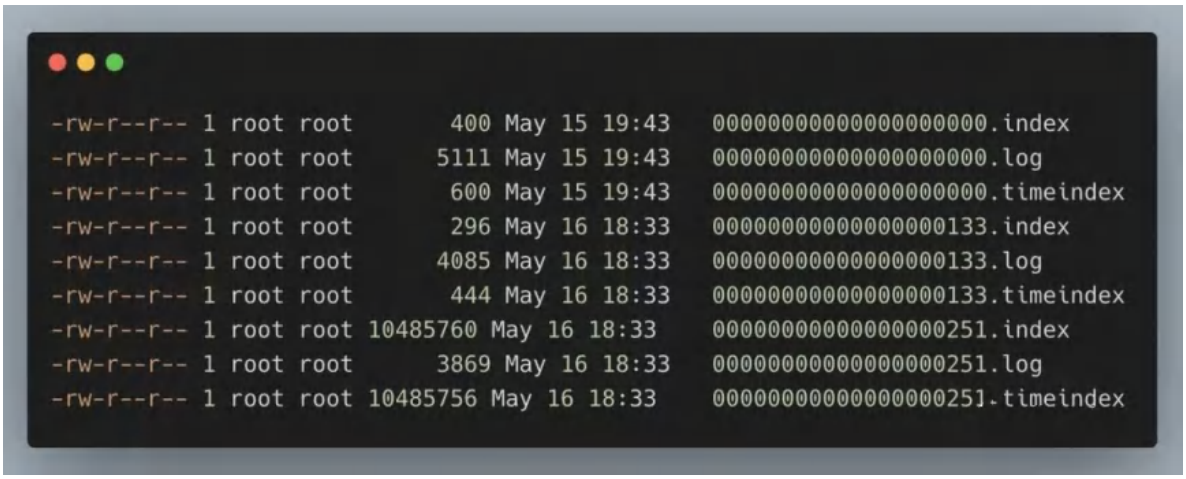
接触过Kafka的老司机一般都知晓Log对应了一个命名形式为-的文件夹。举个例子，假设有一个名为“topic-log”的主题，此主题中具有4个分区，那么在实际物理存储上表现为“topic-log-0”、“topic-log-1”、“topic-log-2”、“topic-log-3”这4个文件夹：



向Log中追加消息时是顺序写入的，只有最后一个LogSegment才能执行写入操作，在此之前所有的LogSegment都不能写入数据。为了方便描述，我们将最后一个LogSegment称为“activeSegment”，即表示当前活跃的日志分段。随着消息的不断写入，当activeSegment满足一定的条件时，就需要创建新的activeSegment，之后追加的消息将写入新的activeSegment。

为了便于消息的检索，每个LogSegment中的日志文件（以“.log”为文件后缀）都有对应的两个索引文件：偏移量索引文件（以“.index”为文件后缀）和时间戳索引文件（以“.timeindex”为文件后缀）。每个LogSegment都有一个基准偏移量baseOffset，用来表示当前LogSegment中第一条消息的offset。偏移量是一个64位的长整型数，日志文件和两个索引文件都是根据基准偏移量（baseOffset）命名的，名称固定为20位数字，没有达到的位数则用0填充。比如第一个LogSegment的基准偏移量为0，对应的日志文件为00000000000000000000.log。

举例说明，向主题topic-log中发送一定量的消息，某一时刻topic-log-0目录中的布局如下所示。



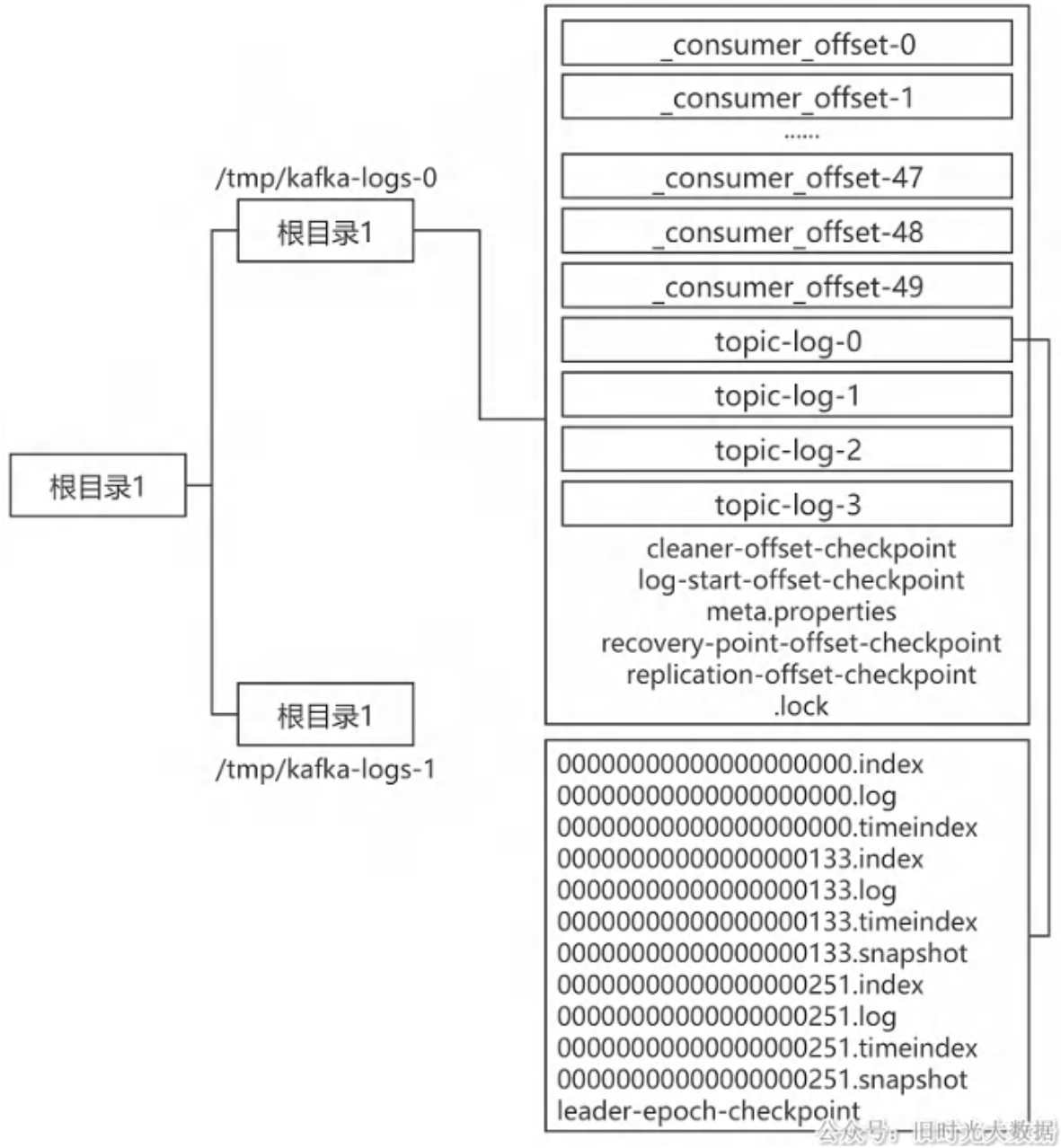
示例中第2个LogSegment对应的基准位移是133，也说明了该LogSegment中的第一条消息的偏移量为133，同时可以反映出第一个LogSegment中共有133条消息（偏移量从0至132的消息）。

注意每个LogSegment中不只包含“.log”、“.index”、“.timeindex”这3种文件，还可能包含“.deleted”、“.cleaned”、 “.swap”等临时文件，以及可能的“.snapshot”、“.txnindex”、“leader-epoch-checkpoint”等文件。

从更加宏观的视角上看，Kafka中的文件不只上面提及的这些文件，比如还有一些检查点文件，当一个Kafka服务第一次启动的时候，默认的根目录下就会创建以下5个文件：

消费者提交的位移是保存在Kafka内部的主题__consumer_offsets中的，初始情况下这个主题并不存在，当第一次有消费者消费消息时会自动创建这个主题。

公众号：旧时光大数据



在某一时刻，Kafka中的文件目录布局如上图所示。每一个根目录都会包含最基本的4个检查点文件（xxx-checkpoint）和meta.properties文件。在创建主题的时候，如果当前broker中不止配置了一个根目录，那么会挑选分区数最少的那个根目录来完成本次创建任务。

Kafka在哪些地方会有选举过程，使用什么工具支持选举?

问过的一些公司：字节

参考答案：

Kafka中的选举大致可以分为三大类：**控制器的选举（先到先得）**、**分区leader的选举（ISR）**以及**消费者相关的选举**

1、控制器的选举

在Kafka集群中会有一个或多个broker，其中有一个broker会被选举为控制器（Kafka Controller），它负责管理整个集群中所有分区和副本的状态等工作。比如当某个分区的leader副本出现故障时，由控制器负责为该分区选举新的leader副本。再比如当检测到某个分区的ISR集合发生变化时，由控制器负责通知所有broker更新其元数据信息。

公众号：旧时光大数据

Kafka Controller的选举是依赖Zookeeper来实现的，在Kafka集群中哪个broker能够成功创建/controller这个临时（EPHEMERAL）节点他就可以成为Kafka Controller。

2、分区Leader的选举

分区leader副本的选举由Kafka Controller负责具体实施。当创建分区（创建主题或增加分区都有创建分区的动作）或分区上线（比如分区中原先的leader副本下线，此时分区需要选举一个新的leader上线来对外提供服务）的时候都需要执行leader的选举动作。

基本思路是按照AR（Assigned Repllicas：分区中的所有副本）集合中副本的顺序查找第一个存活的副本，并且这个副本在ISR集合中。一个分区的AR集合在分配的时候就被指定，并且只要不发生重分配的情况，集合内部副本的顺序是保持不变的，而分区的ISR集合中副本的顺序可能会改变。注意这里是根据AR的顺序而不是ISR的顺序进行选举的。这个说起来比较抽象，有兴趣的读者可以手动关闭/开启某个集群中的broker来观察一下具体的变化。

还有一些情况也会发生分区leader的选举，比如当分区进行重分配（reassign）的时候也需要执行leader的选举动作。这个思路比较简单：从重分配的AR列表中找到第一个存活的副本，且这个副本在目前的ISR列表中。

再比如当发生优先副本（preferred replica partition leader election）的选举时，直接将优先副本设置为leader即可，AR集合中的第一个副本即为优先副本。

还有一种情况就是当某节点被优雅地关闭（也就是执行ControlledShutdown）时，位于这个节点上的leader副本都会下线，所以与此对应的分区需要执行leader的选举。这里的具体思路为：从AR列表中找到第一个存活的副本，且这个副本在目前的ISR列表中，与此同时还要确保这个副本不处于正在被关闭的节点上。

3、消费者相关的选举

组协调器GroupCoordinator需要为消费组内的消费者选举出一个消费组的leader，这个选举的算法也很简单，分两种情况分析。如果消费组内还没有leader，那么第一个加入消费组的消费者即为消费组的leader。如果某一时刻leader消费者由于某些原因退出了消费组，那么会重新选举一个新的leader，这个重新选举leader的过程又更“随意”了，相关代码如下：

```
1   private val members = new mutable.HashMap[String, MemberMetadata]
2   var leaderId = members.keys.head
3
```

解释一下这2行代码：在GroupCoordinator中消费者的信息是以HashMap的形式存储的，其中key为消费者的member_id，而value是消费者相关的元数据信息。leaderId表示leader消费者的member_id，它的取值为HashMap中的第一个键值对的key，这种选举的方式基本上和随机无异。总体上来说，消费组的leader选举过程是很随意的。

用过Kafka的对partition.assignment.strategy（取值为RangeAssignor、RoundRobinAssignor、StickyAssignor等）这个参数都并不陌生。每个消费者都可以设置自己的分区分配策略，对消费组而言需要从各个消费者呈报上来的各个分配策略中选举一个彼此都“信服”的策略来进行整体上的分区分配。这个分区分配的选举并非由leader消费者决定，而是根据消费组内的各个消费者投票来决定的。

Kafka搭建过程要配置什么参数?

问过的一些公司：Bigo

参考答案：

Kafka的重要配置是在server.propertis文件中，具体如下：

```
1   #broker的全局唯一编号，不能重复
```

公众号：旧时光大数据

```
broker.id=0
#删除topic功能使能
delete.topic.enable=true
#处理网络请求的线程数量
num.network.threads=3
#用来处理磁盘IO的现成数量
num.io.threads=8
#发送套接字的缓冲区大小
socket.send.buffer.bytes=102400
#接收套接字的缓冲区大小
socket.receive.buffer.bytes=102400
#请求套接字的缓冲区大小
socket.request.max.bytes=104857600
#kafka运行日志存放的路径
log.dirs=/opt/module/kafka/logs
#topic在当前broker上的分区个数
num.partitions=1
#用来恢复和清理data下数据的线程数量
num.recovery.threads.per.data.dir=1
#segment文件保留的最长时间，超时将被删除
log.retention.hours=168
#配置连接Zookeeper集群地址
zookeeper.connect=hadoop102:2181,hadoop103:2181,hadoop104:2181/kafka

```

基本上熟悉上面的一些参数就可以了，下面放一些更详细的，有需要可以看看。

1、Kafka配置参数

broker.id：broker的id，id是唯一的非负整数，集群的broker.id不能重复。

log.dirs：kafka存放数据的路径。可以是多个，多个使用逗号分隔即可。

port：server接受客户端连接的端口，默认6667

zookeeper.connect：zookeeper集群连接地址。

格式如：zookeeper.connect=server01:2181,server02:2181,server03:2181。

如果需要指定zookeeper集群的路径位置，可以：zookeeper.connect=server01:2181,server02:2181,server03:2181/kafka/cluster。这样设置后，在启动kafka集群前，需要在zookeeper集群创建这个路径/kafka/cluster。

message.max.bytes：server可以接受的消息最大尺寸。默认1000000。

重要的是，consumer和producer有关这个属性的设置必须同步，否则producer发布的消息对consumer来说太大。

num.network.threads：server用来处理网络请求的线程数，默认3。

num.io.threads：server用来处理请求的I/O线程数。这个线程数至少等于磁盘的个数。

background.threads：用于后台处理的线程数。例如文件的删除。默认4。

queued.max.requests：在网络线程停止读取新请求之前，可以排队等待I/O线程处理的最大请求个数。默认500。

host.name：broker的hostname

如果hostname已经设置的话，broker将只会绑定到这个地址上；如果没有设置，它将绑定到所有接口，并发布一份到ZK

公众号：旧时光大数据

advertised.host.name：如果设置，则就作为broker 的hostname发往producer、consumers以及其他brokers

advertised.port：此端口将给与producers、consumers、以及其他brokers，它会在建立连接时用到； 它仅在实际端口和server需要绑定的端口不一样时才需要设置。

socket.send.buffer.bytes：SO_SNDBUFF 缓存大小，server进行socket 连接所用，默认100*1024。

socket.receive.buffer.bytes：SO_RCVBUFF缓存大小，server进行socket连接时所用。默认100 * 1024。

socket.request.max.bytes：server允许的最大请求尺寸；这将避免server溢出，它应该小于Java heap size。

num.partitions：如果创建topic时没有给出划分partitions个数，这个数字将是topic下partitions数目的默认数值。默认1。

log.segment.bytes：topic partition的日志存放在某个目录下诸多文件中，这些文件将partition的日志切分成一段一段的；这个属性就是每个文件的最大尺寸；当尺寸达到这个数值时，就会创建新文件。此设置可以由每个topic基础设置时进行覆盖。默认1014 *1024* 1024

log.roll.hours：即使文件没有到达log.segment.bytes，只要文件创建时间到达此属性，就会创建新文件。这个设置也可以有topic层面的设置进行覆盖。默认24*7

log.cleanup.policy：log清除策略。默认delete。

log.retention.minutes和log.retention.hours：每个日志文件删除之前保存的时间。默认数据保存时间对所有topic都一样。

log.retention.minutes 和 log.retention.bytes 都是用来设置删除日志文件的，无论哪个属性已经溢出。

这个属性设置可以在topic基本设置时进行覆盖。

log.retention.bytes：每个topic下每个partition保存数据的总量。

注意，这是每个partitions的上限，因此这个数值乘以partitions的个数就是每个topic保存的数据总量。如果log.retention.hours和log.retention.bytes都设置了，则超过了任何一个限制都会造成删除一个段文件。

注意，这项设置可以由每个topic设置时进行覆盖。

log.retention.check.interval.ms：检查日志分段文件的间隔时间，以确定是否文件属性是否到达删除要求。默认5min。

log.cleaner.enable：当这个属性设置为false时，一旦日志的保存时间或者大小达到上限时，就会被删除；如果设置为true，则当保存属性达到上限时，就会进行log compaction。默认false。

log.cleaner.threads：进行日志压缩的线程数。默认1。

log.cleaner.io.max.bytes.per.second：进行log compaction时，log cleaner可以拥有的最大I/O数目。这项设置限制了cleaner，以避免干扰活动的请求服务。

log.cleaner.io.buffer.size：log cleaner清除过程中针对日志进行索引化以及精简化所用到的缓存大小。最好设置大点，以提供充足的内存。默认500 *1024* 1024。

log.cleaner.io.buffer.load.factor：进行log cleaning时所需要的I/O chunk尺寸。你不需要更改这项设置。默认512*1024。

log.cleaner.io.buffer.load.factor：log cleaning中所使用的hash表的负载因子；你不需要更改这个选项。默认0.9

log.cleaner.backoff.ms：进行日志是否清理检查的时间间隔，默认15000。

log.cleaner.min.cleanable.ratio：这项配置控制log compactor试图清理日志的频率（假定log compaction是打开的）。

默认避免清理压缩超过50%的日志。这个比率绑定了备份日志所消耗的最大空间（50%的日志备份时压缩率为50%）。更高的比率则意味着浪费消耗更少，也就可以更有效的清理更多的空间。这项设置在每个topic设置中可以覆盖。

log.cleaner.delete.retention.ms：保存时间；保存压缩日志的最长时间；也是客户端消费消息的最长时间，与log.retention.minutes的区别在于一个控制未压缩数据，一个控制压缩后的数据；会被topic创建时的指定时间覆盖。

log.index.size.max.bytes：每个log segment的最大尺寸。注意，如果log尺寸达到这个数值，即使尺寸没有超过log.segment.bytes限制，也需要产生新的log segment。默认10 *1024* 1024。

log.index.interval.bytes：当执行一次fetch后，需要一定的空间扫描最近的offset，设置的越大越好，一般使用默认值就可以。默认4096。

log.flush.interval.messages：log文件“sync”到磁盘之前累积的消息条数。

因为磁盘IO操作是一个慢操作，但又是一个“数据可靠性”的必要手段，所以检查是否需要固化到硬盘的时间间隔。需要在“数据可靠性”与“性能”之间做必要的权衡，如果此值过大，将会导致每次“发sync”的时间过长（IO阻塞），如果此值过小，将会导致“fsync”的时间较长（IO阻塞），导致”发sync“的次数较多，这也就意味着整体的client请求有一定的延迟，物理server故障，将会导致没有fsync的消息丢失。

log.flush.scheduler.interval.ms：检查是否需要fsync的时间间隔。默认Long.MaxValue

log.flush.interval.ms：仅仅通过interval来控制消息的磁盘写入时机，是不足的，这个数用来控制”fsync“的时间间隔，如果消息量始终没有达到固化到磁盘的消息数，但是离上次磁盘同步的时间间隔达到阈值，也将触发磁盘同步。

log.delete.delay.ms：文件在索引中清除后的保留时间，一般不需要修改。默认60000。

auto.create.topics.enable：是否允许自动创建topic。如果是true，则produce或者fetch 不存在的topic时，会自动创建这个topic。否则需要使用命令行创建topic。默认true。

controller.socket.timeout.ms：partition管理控制器进行备份时，socket的超时时间。默认30000。

controller.message.queue.size：controller-to-broker-channles的buffer尺寸，默认Int.MaxValue。

default.replication.factor：默认备份份数，仅指自动创建的topics。默认1。

replica.lag.time.max.ms：如果一个follower在这个时间内没有发送fetch请求，leader将从ISR重移除这个follower，并认为这个follower已经挂了，默认10000。

replica.lag.max.messages：如果一个replica没有备份的条数超过这个数值，则leader将移除这个follower，并认为这个follower已经挂了，默认4000。

replica.socket.timeout.ms：leader 备份数据时的socket网络请求的超时时间，默认30*1000

replica.socket.receive.buffer.bytes：备份时向leader发送网络请求时的socket receive buffer。默认64*1024。

replica.fetch.max.bytes：备份时每次fetch的最大值。默认1024*1024。

replica.fetch.max.bytes：leader发出备份请求时，数据到达leader的最长等待时间。默认500。

replica.fetch.min.bytes：备份时每次fetch之后回应的最小尺寸。默认1。

num.replica.fetchers：从leader备份数据的线程数。默认1。

replica.high.watermark.checkpoint.interval.ms：每个replica检查是否将最高水位进行固化的频率。默认5000.

fetch.purgatory.purge.interval.requests：fetch 请求清除时的清除间隔，默认1000

producer.purgatory.purge.interval.requests：producer请求清除时的清除间隔，默认1000

公众号：旧时光大数据

zookeeper.session.timeout.ms：zookeeper会话超时时间。默认6000

zookeeper.connection.timeout.ms：客户端等待和zookeeper建立连接的最大时间。默认6000

zookeeper.sync.time.ms：zk follower落后于zk leader的最长时间。默认2000

controlled.shutdown.enable：是否能够控制broker的关闭。如果能够，broker将可以移动所有leaders到其他的broker上，在关闭之前。这减少了不可用性在关机过程中。默认true。

controlled.shutdown.max.retries：在执行不彻底的关机之前，可以成功执行关机的命令数。默认3.

controlled.shutdown.retry.backoff.ms：在关机之间的backoff时间。默认5000

auto.leader.rebalance.enable：如果这是true，控制者将会自动平衡brokers对于partitions的leadership。默认true。

leader.imbalance.per.broker.percentage：每个broker所允许的leader最大不平衡比率，默认10。

leader.imbalance.check.interval.seconds：检查leader不平衡的频率，默认300

offset.metadata.max.bytes：允许客户端保存他们offsets的最大个数。默认4096

max.connections.per.ip：每个ip地址上每个broker可以被连接的最大数目。默认Int.MaxValue。

max.connections.per.ip.overrides：每个ip或者hostname默认的连接的最大覆盖。

connections.max.idle.ms：空连接的超时限制，默认600000

log.roll.jitter.{ms,hours}：从logRollTimeMillis抽离的jitter最大数目。默认0

num.recovery.threads.per.data.dir：每个数据目录用来日志恢复的线程数目。默认1。

unclean.leader.election.enable：指明了是否能够使不在ISR中replicas设置用来作为leader。默认true

delete.topic.enable：能够删除topic，默认false。

offsets.topic.num.partitions：默认50。由于部署后更改不受支持，因此建议使用更高的设置来进行生产（例如100-200）。

offsets.topic.retention.minutes：存在时间超过这个时间限制的offsets都将被标记为待删除。默认1440。

offsets.retention.check.interval.ms：offset管理器检查陈旧offsets的频率。默认600000。

offsets.topic.replication.factor：topic的offset的备份份数。建议设置更高的数字保证更高的可用性。默认3

offset.topic.segment.bytes：offsets topic的segment尺寸。默认104857600

offsets.load.buffer.size：这项设置与批量尺寸相关，当从offsets segment中读取时使用。默认5242880

offsets.commit.required.acks：在offset commit可以接受之前，需要设置确认的数目，一般不需要更改。默认-1。

2、Kafka生产者配置参数

boostrap.servers：用于建立与kafka集群连接的host/port组。

数据将会在所有servers上均衡加载，不管哪些server是指定用于bootstrapping。

这个列表格式：host1:port1,host2:port2,…

acks：此配置实际上代表了数据备份的可用性。

acks=0： 设置为0表示producer不需要等待任何确认收到的信息。副本将立即加到socket buffer并认为已经发送。没有任何保障可以保证此种情况下server已经成功接收数据，同时重试配置不会发生作用

公众号：旧时光大数据

acks=1： 这意味着至少要等待leader已经成功将数据写入本地log，但是并没有等待所有follower是否成功写入。这种情况下，如果follower没有成功备份数据，而此时leader又挂掉，则消息会丢失。

acks=all： 这意味着leader需要等待所有备份都成功写入日志，这种策略会保证只要有一个备份存活就不会丢失数据。这是最强的保证。

buffer.memory：producer可以用来缓存数据的内存大小。如果数据产生速度大于向broker发送的速度，producer会阻塞或者抛出异常，以“block.on.buffer.full”来表明。

compression.type：producer用于压缩数据的压缩类型。默认是无压缩。正确的选项值是none、gzip、snappy。压缩最好用于批量处理，批量处理消息越多，压缩性能越好。

retries：设置大于0的值将使客户端重新发送任何数据，一旦这些数据发送失败。注意，这些重试与客户端接收到发送错误时的重试没有什么不同。

允许重试将潜在的改变数据的顺序，如果这两个消息记录都是发送到同一个partition，则第一个消息失败第二个发送成功，则第二条消息会比第一条消息出现要早。

batch.size：producer将试图批处理消息记录，以减少请求次数。这将改善client与server之间的性能。这项配置控制默认的批量处理消息字节数。

client.id：当向server发出请求时，这个字符串会发送给server。目的是能够追踪请求源头，以此来允许ip/port许可列表之外的一些应用可以发送信息。这项应用可以设置任意字符串，因为没有任何功能性的目的，除了记录和跟踪。

linger.ms：producer组将会汇总任何在请求与发送之间到达的消息记录一个单独批量的请求。通常来说，这只有在记录产生速度大于发送速度的时候才能发生。

max.request.size：请求的最大字节数。这也是对最大记录尺寸的有效覆盖。注意：server具有自己对消息记录尺寸的覆盖，这些尺寸和这个设置不同。此项设置将会限制producer每次批量发送请求的数目，以防发出巨量的请求。

receive.buffer.bytes：TCP receive缓存大小，当阅读数据时使用。

send.buffer.bytes：TCP send缓存大小，当发送数据时使用。

timeout.ms：此配置选项控制server等待来自followers的确认的最大时间。如果确认的请求数目在此时间内没有实现，则会返回一个错误。这个超时限制是以server端度量的，没有包含请求的网络延迟。

block.on.buffer.full：当我们内存缓存用尽时，必须停止接收新消息记录或者抛出错误。

默认情况下，这个设置为真，然而某些阻塞可能不值得期待，因此立即抛出错误更好。设置为false则会这样：producer会抛出一个异常错误：BufferExhaustedException， 如果记录已经发送同时缓存已满。

metadata.fetch.timeout.ms：是指我们所获取的一些元素据的第一个时间数据。元素据包含：topic，host，partitions。此项配置是指当等待元素据fetch成功完成所需要的时间，否则会抛出异常给客户端。

metadata.max.age.ms：以微秒为单位的时间，是在我们强制更新metadata的时间间隔。即使我们没有看到任何partition leadership改变。

metric.reporters：类的列表，用于衡量指标。实现MetricReporter接口，将允许增加一些类，这些类在新的衡量指标产生时就会改变。JmxReporter总会包含用于注册JMX统计

metrics.num.samples：用于维护metrics的样本数。

metrics.sample.window.ms：metrics系统维护可配置的样本数量，在一个可修正的window size。这项配置配置了窗口大小，例如。我们可能在30s的期间维护两个样本。当一个窗口推出后，我们会擦除并重写最老的窗口。

recoonect.backoff.ms：连接失败时，当我们重新连接时的等待时间。这避免了客户端反复重连。

retry.backoff.ms：在试图重试失败的produce请求之前的等待时间。避免陷入发送-失败的死循环中。

公众号：旧时光大数据

3、Kafka消费者配置参数

group.id：用来唯一标识consumer进程所在组的字符串，如果设置同样的group id，表示这些processes都是属于同一个consumer group。

zookeeper.connect：指定zookeeper的连接的字符串，格式是hostname：port, hostname：port…

consumer.id：不需要设置，一般自动产生

socket.timeout.ms：网络请求的超时限制。真实的超时限制是max.fetch.wait+socket.timeout.ms。默认3000

socket.receive.buffer.bytes：socket用于接收网络请求的缓存大小。默认64*1024。

fetch.message.max.bytes：每次fetch请求中，针对每次fetch消息的最大字节数。默认1024*1024

这些字节将会督导用于每个partition的内存中，因此，此设置将会控制consumer所使用的memory大小。

这个fetch请求尺寸必须至少和server允许的最大消息尺寸相等，否则，producer可能发送的消息尺寸大于consumer所能消耗的尺寸。

num.consumer.fetchers：用于fetch数据的fetcher线程数。默认1

auto.commit.enable：如果为真，consumer所fetch的消息的offset将会自动的同步到zookeeper。这项提交的offset将在进程挂掉时，由新的consumer使用。默认true。

auto.commit.interval.ms：consumer向zookeeper提交offset的频率，单位是秒。默认60*1000。

queued.max.message.chunks：用于缓存消息的最大数目，每个chunk必须和fetch.message.max.bytes相同。默认2。

rebalance.max.retries：当新的consumer加入到consumer group时，consumers集合试图重新平衡分配到每个consumer的partitions数目。如果consumers集合改变了，当分配正在执行时，这个重新平衡会失败并重入。默认4

fetch.min.bytes：每次fetch请求时，server应该返回的最小字节数。如果没有足够的数据返回，请求会等待，直到足够的数据才会返回。

fetch.wait.max.ms：如果没有足够的数据能够满足fetch.min.bytes，则此项配置是指在应答fetch请求之前，server会阻塞的最大时间。默认100

rebalance.backoff.ms：在重试reblance之前backoff时间。默认2000

refresh.leader.backoff.ms：在试图确定某个partition的leader是否失去他的leader地位之前，需要等待的backoff时间。默认200

auto.offset.reset：zookeeper中没有初始化的offset时，如果offset是以下值的回应：

lastest：自动复位offset为lastest的offset

earliest：自动复位offset为earliest的offset

none：向consumer抛出异常

consumer.timeout.ms：如果没有消息可用，即使等待特定的时间之后也没有，则抛出超时异常

exclude.internal.topics：是否将内部topics的消息暴露给consumer。默认true。

paritition.assignment.strategy：选择向consumer 流分配partitions的策略，可选值：range，roundrobin。默认range。

client.id：是用户特定的字符串，用来在每次请求中帮助跟踪调用。它应该可以逻辑上确认产生这个请求的应用。

公众号：旧时光大数据

zookeeper.session.timeout.ms：zookeeper 会话的超时限制。默认6000

如果consumer在这段时间内没有向zookeeper发送心跳信息，则它会被认为挂掉了，并且reblance将会产生

zookeeper.connection.timeout.ms：客户端在建立通zookeeper连接中的最大等待时间。默认6000

zookeeper.sync.time.ms：ZK follower可以落后ZK leader的最大时间。默认1000

offsets.storage：用于存放offsets的地点： zookeeper或者kafka。默认zookeeper。

offset.channel.backoff.ms：重新连接offsets channel或者是重试失败的offset的fetch/commit请求的backoff时间。默认1000

offsets.channel.socket.timeout.ms：当读取offset的fetch/commit请求回应的socket 超时限制。此超时限制是被consumerMetadata请求用来请求offset管理。默认10000。

offsets.commit.max.retries：重试offset commit的次数。这个重试只应用于offset commits在shut-down之间。默认5。

dual.commit.enabled：如果使用“kafka”作为offsets.storage，你可以二次提交offset到zookeeper(还有一次是提交到kafka）。

在zookeeper-based的offset storage到kafka-based的offset storage迁移时，这是必须的。对任意给定的consumer group来说，比较安全的建议是当完成迁移之后就关闭这个选项

partition.assignment.strategy：在“range”和“roundrobin”策略之间选择一种作为分配partitions给consumer 数据流的策略。

循环的partition分配器分配所有可用的partitions以及所有可用consumer线程。它会将partition循环的分配到consumer线程上。如果所有consumer实例的订阅都是确定的，则partitions的划分是确定的分布。

循环分配策略只有在以下条件满足时才可以

1）每个topic在每个consumer实力上都有同样数量的数据流。

2）订阅的topic的集合对于consumer group中每个consumer实例来说都是确定的

Kafka的单播和多播

问过的一些公司：猿辅导

参考答案：

1、单播

一条消息只能被某一个消费者消费的模式称为单播。要实现消息单播，只要让这些消费者属于同一个消费者组即可。当生产者发送一条消息时，两个消费者中只有一个能收到消息。

2、多播

一条消息能够被多个消费者消费的模式称为多播。之所以不称之为广播，是因为一条消息只能被Kafka同一个分组下某一个消费者消费，而不是所有消费者都能消费，所以从严格意义上来讲并不能算是广播模式，当然如果希望实现广播模式只要保证每个消费者均属于不同的消费者组。针对Kafka同一条消息只能被同一个消费者组下的某一个消费者消费的特性，要实现多播只要保证这些消费者属于不同的消费者组即可。然后通过生产者发送几条消息，可以看到不同消费者组的消费者同时能消费到消息，然而同一个消费者组下的消费者却只能有一个消费者能消费到消息。

公众号：旧时光大数据

Kafka的高水位和Leader Epoch

问过的一些公司：ebay

参考答案：

高水位(High Watermark)，**通常被用在流式处理领域（比如Apache Flink、Apache Spark等），以表征元素或事件在基于时间层面上的进度。**一个比较经典的表述为：流式系统保证在水位t时刻，创建时间（event time） = t'且t' ≤ t的所有事件都已经到达或被观测到。**在Kafka中，水位的概念反而与时间无关，而是与位置信息相关。**严格来说，**它表示的就是位置信息，即位移（offset）。**通俗的说下HW作用：**Kafka使用HW值来决定副本备份的进度**

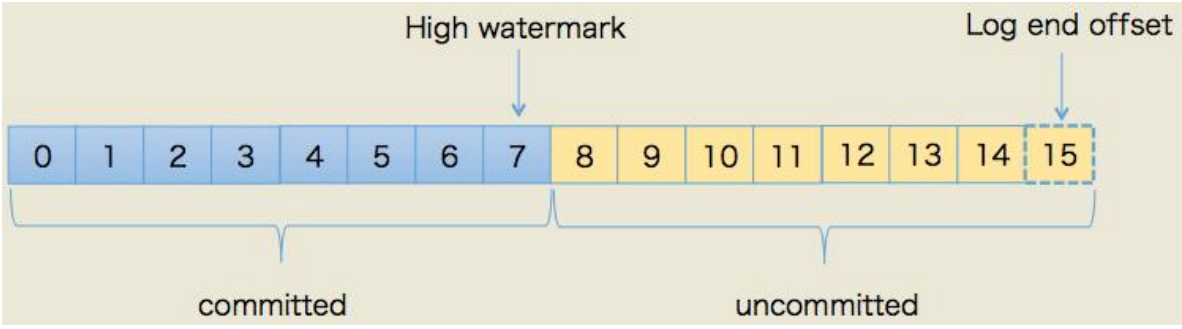
Kafka分区下有可能有很多个副本(replica)用于实现冗余，从而进一步实现高可用。副本根据角色的不同可分为3类：

- leader副本：响应clients端读写请求的副本
- follower副本：被动地备份leader副本中的数据，不能响应clients端读写请求。
- ISR副本：包含了leader副本和所有与leader副本保持同步的follower副本——如何判定是否与leader同步后面会提到

每个Kafka副本对象都有两个重要的属性：LEO和HW。**注意是所有的副本，而不只是leader副本。**

- LEO：即日志末端位移(log end offset)，记录了该副本底层日志(log)中下一条消息的位移值。注意是下一条消息！也就是说，如果LEO=10，那么表示该副本保存了10条消息，位移值范围是[0, 9]。另外，leader LEO和follower LEO的更新是有区别的。
- HW：即上面提到的水位值。对于同一个副本对象而言，其HW值不会大于LEO值。小于等于HW值的所有消息都被认为是“已备份”的（replicated）。同理，leader副本和follower副本的HW更新也是有区别的。

通过下图我们来了解下LEO和HW两者的关系：



上图中，HW值是7，表示位移是0 ~ 7的所有消息都已经处于“已备份状态”（committed），而LEO值是15，那么8~14的消息就是尚未完全备份（fully replicated）——为什么没有15？因为刚才说过了，LEO指向的是下一条消息到来时的位移，故上图使用虚线框表示。我们总说consumer无法消费未提交消息。这句话如果用以上名词来解读的话，应该表述为：consumer无法消费分区下leader副本中位移值大于**分区HW的任何消息。这里需要特别注意分区HW就是leader副本的HW值。**

1、follower副本何时更新LEO?

如前所述，follower副本只是被动地向leader副本请求数据，具体表现为follower副本不停地向leader副本所在的broker发送FETCH请求，一旦获取消息后写入自己的日志中进行备份。那么follower副本的LEO是何时更新的呢？Kafka有两套follower副本LEO：1. 一套LEO保存在follower副本所在broker的副本管理机中；2. 另一套LEO保存在leader副本所在broker的副本管理机中——换句话说，leader副本机器上保存了所有的follower副本的LEO。

为什么要保存两套？这是因为Kafka使用前者帮助follower副本更新其HW值；而利用后者帮助leader副本更新其HW使用。下面我们分别看下它们被更新的时机。

1）follower副本端的follower副本LEO何时更新?

公众号：旧时光大数据

follower副本端的LEO值就是其底层日志的LEO值，也就是说每当新写入一条消息，其LEO值就会被更新(类似于LEO += 1)。当follower发送FETCH请求后，leader将数据返回给follower，此时follower开始向底层log写数据，从而自动地更新LEO值。

2）leader副本端的follower副本LEO何时更新?

leader副本端的follower副本LEO的更新发生在leader在处理follower FETCH请求时。一旦leader接收到follower发送的FETCH请求，它首先会从自己的log中读取相应的数据，但是在给follower返回数据之前它先去更新follower的LEO(即上面所说的第二套LEO)。

2、follower副本何时更新HW?

follower更新HW发生在其更新LEO之后，一旦follower向log写完数据，它会尝试更新它自己的HW值。具体算法就是比较当前LEO值与FETCH响应中leader的HW值，取两者的小者作为新的HW值。这告诉我们一个事实：如果follower的LEO值超过了leader的HW值，那么follower HW值是不会越过leader HW值的。

3、leader副本何时更新LEO?

和follower更新LEO道理相同，leader写log时就会自动地更新它自己的LEO值。

4、leader副本何时更新HW值?

前面说过了，leader的HW值就是分区HW值，因此何时更新这个值是我们最关心的，因为它直接影响了分区数据对于consumer的可见性 。以下4种情况下leader会尝试去更新分区HW——切记是尝试，有可能因为不满足条件而不做任何更新：

- 副本成为leader副本时：当某个副本成为了分区的leader副本，Kafka会尝试去更新分区HW。这是显而易见的道理，毕竟分区leader发生了变更，这个副本的状态是一定要检查的！不过，本文讨论的是当系统稳定后且正常工作时备份机制可能出现的问题，故这个条件不在我们的讨论之列。
- broker出现崩溃导致副本被踢出ISR时：若有broker崩溃则必须查看下是否会波及此分区，因此检查下分区HW值是否需要更新是有必要的。本文不对这种情况做深入讨论
- producer向leader副本写入消息时：因为写入消息会更新leader的LEO，故有必要再查看下HW值是否也需要修改
- leader处理follower FETCH请求时：当leader处理follower的FETCH请求时首先会从底层的log读取数据，之后会尝试更新分区HW值

特别注意上面4个条件中的最后两个。它揭示了一个事实——当Kafka broker都正常工作时，分区HW值的更新时机有两个：leader处理PRODUCE请求时和leader处理FETCH请求时。另外，leader是如何更新它的HW值的呢？前面说过了，leader broker上保存了一套follower副本的LEO以及它自己的LEO。当尝试确定分区HW时，它会选出所有满足条件的副本，比较它们的LEO(当然也包括leader自己的LEO)，并选择最小的LEO值作为HW值。这里的满足条件主要是指副本要满足以下两个条件之一：

- 处于ISR中
- 副本LEO落后于leader LEO的时长不大于replica.lag.time.max.ms参数值(默认是10s)

乍看上去好像这两个条件说得是一回事，毕竟ISR的定义就是第二个条件描述的那样。但某些情况下Kafka的确可能出现副本已经“追上”了leader的进度，但却不在ISR中——比如某个从failure中恢复的副本。如果Kafka只判断第一个条件的话，确定分区HW值时就不会考虑这些未在ISR中的副本，但这些副本已经具备了“立刻进入ISR”的资格，因此就可能出现分区HW值越过ISR中副本LEO的情况——这肯定是不允许的，因为分区HW实际上就是ISR中所有副本LEO的最小值。

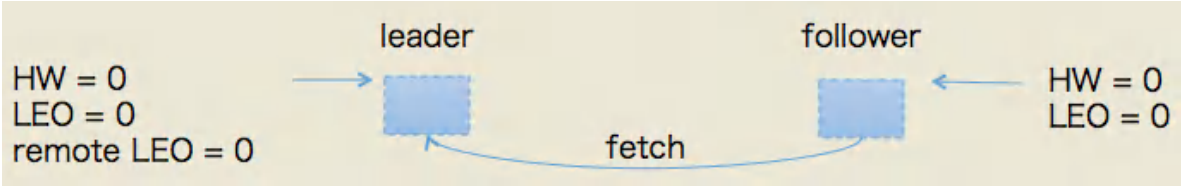
下面举个实际的例子。我们假设有一个topic，单分区，副本因子是2，即一个leader副本和一个follower副本。我们看下当producer发送一条消息时，broker端的副本到底会发生什么事情以及分区HW是如何被更新的。

下图是初始状态，稍微解释一下：初始时leader和follower的HW和LEO都是0(严格来说源代码会初始化LEO为-1，不过这不影响之后的讨论)。leader中的remote LEO指的就是leader端保存的follower LEO，也被初始化成0。此时，producer没有发送任何消息给leader，而follower已经开始不断地给leader发送FETCH请求了，但因为没有数据因此什么都不会发生。值得一提的是，follower发送过来的FETCH请求因

公众号：旧时光大数据

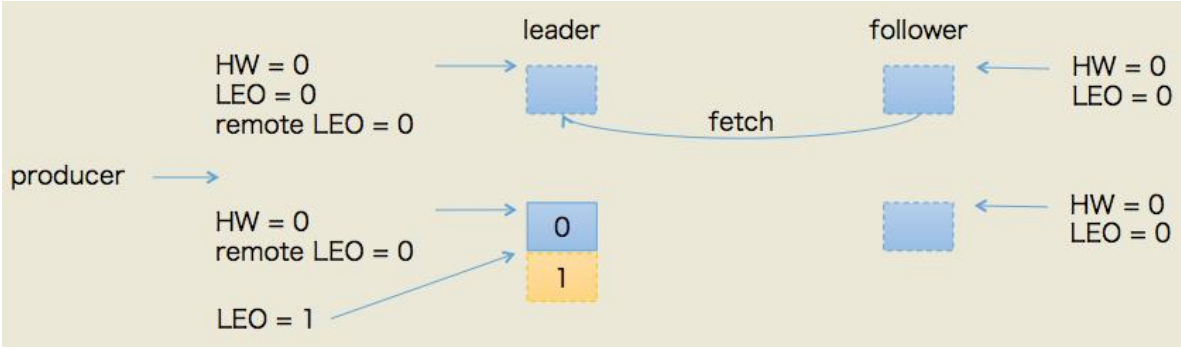
为无数据而暂时会被寄存到leader端的purgatory中，待500ms(replica.fetch.wait.max.ms参数)超时后会强制完成。倘若在寄存期间producer端发送过来数据，那么会Kafka会自动唤醒该FETCH请求，让leader继续处理之。

因为FETCH请求发送和PRODUCE请求处理的时机会影响后面的一些内容。因此后续我们也将分两种情况来讨论分区HW的更新。



第一种情况：follower发送FETCH请求在leader处理完PRODUCE请求之后

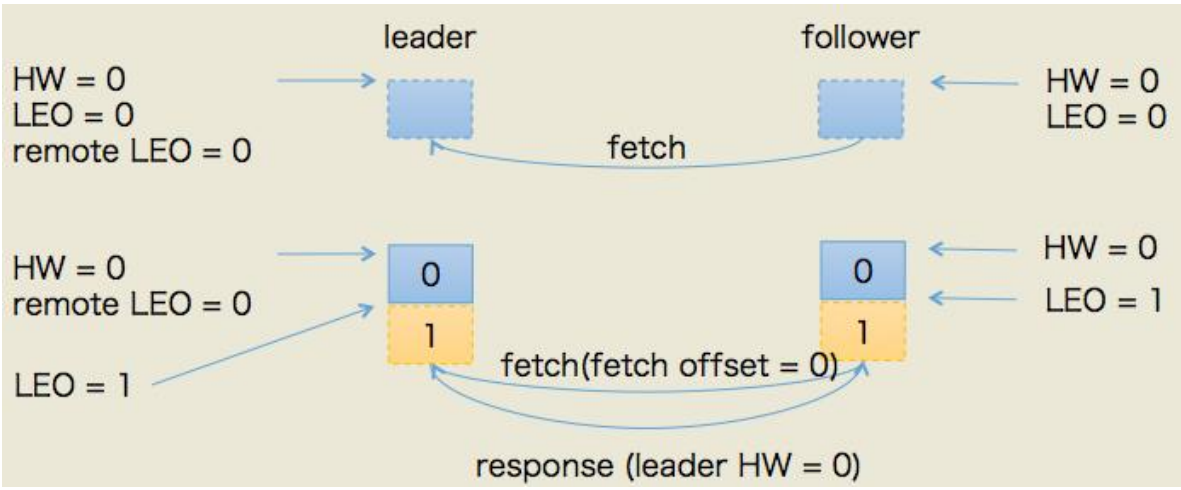
producer给该topic分区发送了一条消息。此时的状态如下图所示：



如图所示，leader接收到PRODUCE请求主要做两件事情：

- 把消息写入写底层log（同时也就自动地更新了leader的LEO）
- 尝试更新leader HW值（前面**leader副本何时更新HW值**一节中的第三个条件触发）。我们已经假设此时follower尚未发送FETCH请求，那么leader端保存的remote LEO依然是0，因此leader会比较它自己的LEO值和remote LEO值，发现最小值是0，与当前HW值相同，故不会更新分区HW值

所以，PRODUCE请求处理完成后leader端的HW值依然是0，而LEO是1，remote LEO是1。假设此时follower发送了FETCH请求(或者说follower早已发送了FETCH请求，只不过在broker的请求队列中排队)，那么状态变更如下图所示：



本例中当follower发送FETCH请求时，leader端的处理依次是：

1. 读取底层log数据
2. 更新remote LEO = 0（为什么是0？ 因为此时follower还没有写入这条消息。leader如何确认follower还未写入呢？这是通过follower发来的FETCH请求中的fetch offset来确定的）
3. 尝试更新分区HW——此时leader LEO = 1，remote LEO = 0，故分区HW值= min(leader LEO, follower remote LEO) = 0

公众号：旧时光大数据

4. 把数据和当前分区HW值（依然是0）发送给follower副本

而follower副本接收到FETCH response后依次执行下列操作：

1. 写入本地log（同时更新follower LEO）
2. 更新follower HW——比较本地LEO和当前leader HW取小者，故follower HW = 0

此时，第一轮FETCH RPC结束，我们会发现虽然leader和follower都已经在log中保存了这条消息，但分区HW值尚未被更新。实际上，它是在第二轮FETCH RPC中被更新的，如下图所示：

上图中，follower发来了第二轮FETCH请求，leader端接收到后仍然会依次执行下列操作：

1. 读取底层log数据
2. 更新remote LEO = 1（这次为什么是1了？ 因为这轮FETCH RPC携带的fetch offset是1，那么为什么这轮携带的就是1了呢，因为上一轮结束后follower LEO被更新为1了）
3. 尝试更新分区HW——此时leader LEO = 1，remote LEO = 1，故分区HW值= min(leader LEO, follower remote LEO) = 1。**注意分区HW值此时被更新了！！！**
4. 把数据（实际上没有数据）和当前分区HW值（已更新为1）发送给follower副本

同样地，follower副本接收到FETCH response后依次执行下列操作：

1. **写入本地log，当然没东西可写，故follower LEO也不会变化，依然是1**
2. **更新follower HW——比较本地LEO和当前leader LEO取小者。由于此时两者都是1，故更新follower HW = 1**

producer端发送消息后broker端完整的处理流程就讲完了。此时消息已经成功地被复制到leader和follower的log中且分区HW是1，表明consumer能够消费offset = 0的这条消息。下面我们来分析下PRODUCE和FETCH请求交互的第二种情况。

第二种情况：FETCH请求保存在purgatory中PRODUCE请求到来

这种情况实际上和第一种情况差不多。前面说过了，当leader无法立即满足FECTH返回要求的时候(比如没有数据)，那么该FETCH请求会被暂存到leader端的purgatory中，待时机成熟时会尝试再次处理它。不过Kafka不会无限期地将其缓存着，默认有个超时时间（500ms），一旦超时时间已过，则这个请求会被强制完成。不过我们要讨论的场景是在寄存期间，producer发送PRODUCE请求从而使之满足了条件从而被唤醒。此时，leader端处理流程如下：

1. leader写入本地log（同时自动更新leader LEO）
2. 尝试唤醒在purgatory中寄存的FETCH请求
3. 尝试更新分区HW

至于唤醒后的FETCH请求的处理与第一种情况完全一致，故这里不做详细展开了。

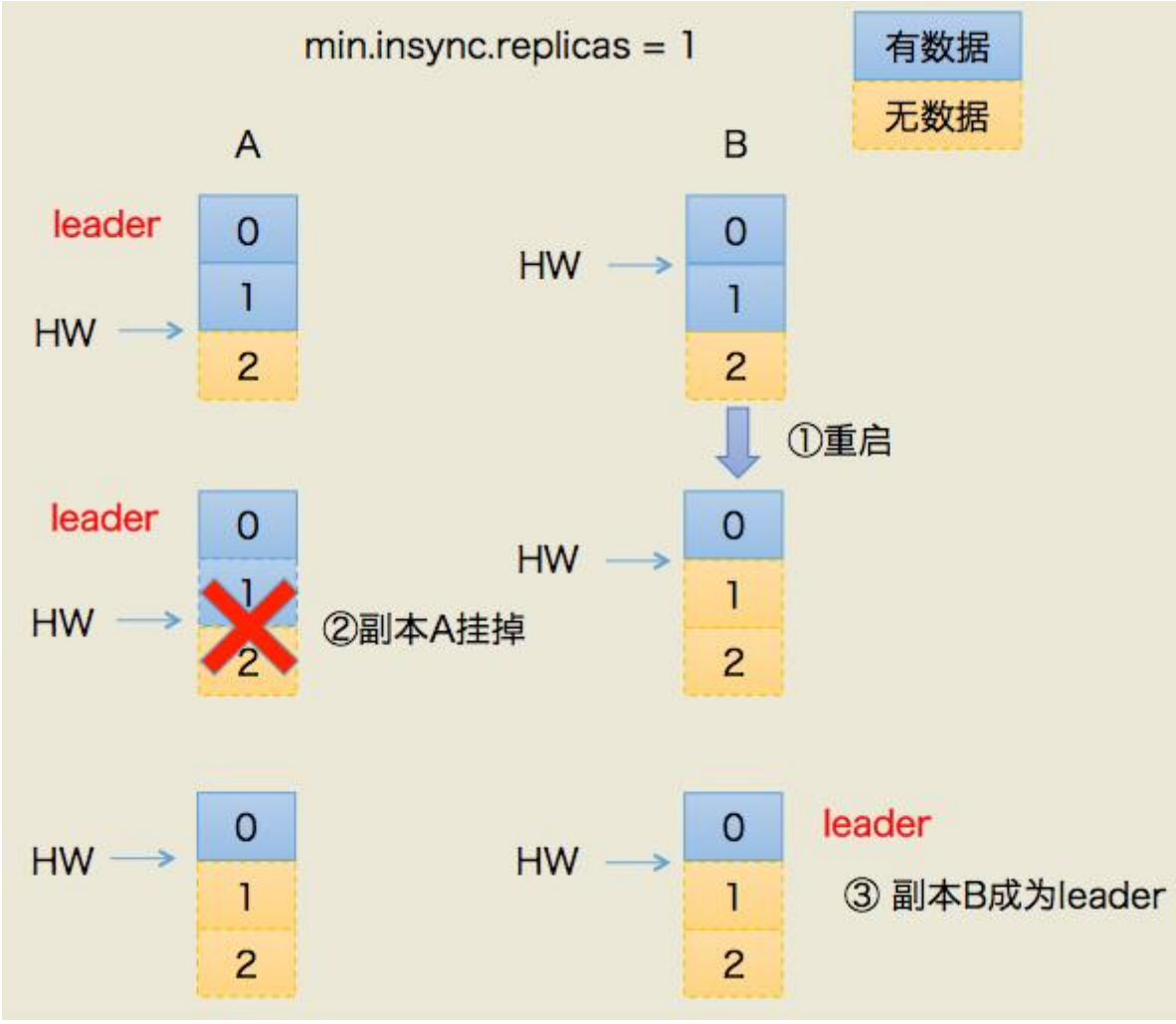
以上所有的东西其实就想说明一件事情：**Kafka使用HW值来决定副本备份的进度，而HW值的更新通常需要额外一轮FETCH RPC才能完成，故而这种设计是有问题的。**它们可能引起的问题包括：

- 备份数据丢失
- 备份数据不一致

我们来看下上述的两种情况：

1）数据丢失

如前所述，使用HW值来确定备份进度时其值的更新是在下一轮RPC中完成的。现在翻到上面使用两种不同颜色标记的步骤处思考下， 如果follower副本在蓝色标记的第一步与紫色标记的第二步之间发生崩溃，那么就有可能造成数据的丢失。我们举个例子来看下。



上图中有两个副本：A和B。开始状态是A是leader。我们假设producer端min.insync.replicas设置为1，那么当producer发送两条消息给A后，A写入到底层log，此时Kafka会通知producer说这两条消息写入成功。

但是在broker端，leader和follower底层的log虽都写入了2条消息且分区HW已经被更新到2，但follower HW尚未被更新（也就是上面紫色颜色标记的第二步尚未执行）。倘若此时副本B所在的broker宕机，那么重启回来后B会自动把LEO调整到之前的HW值，故副本B会做日志截断(log truncation)，将offset = 1的那条消息从log中删除，并调整LEO = 1，此时follower副本底层log中就只有一条消息，即offset = 0的消息。

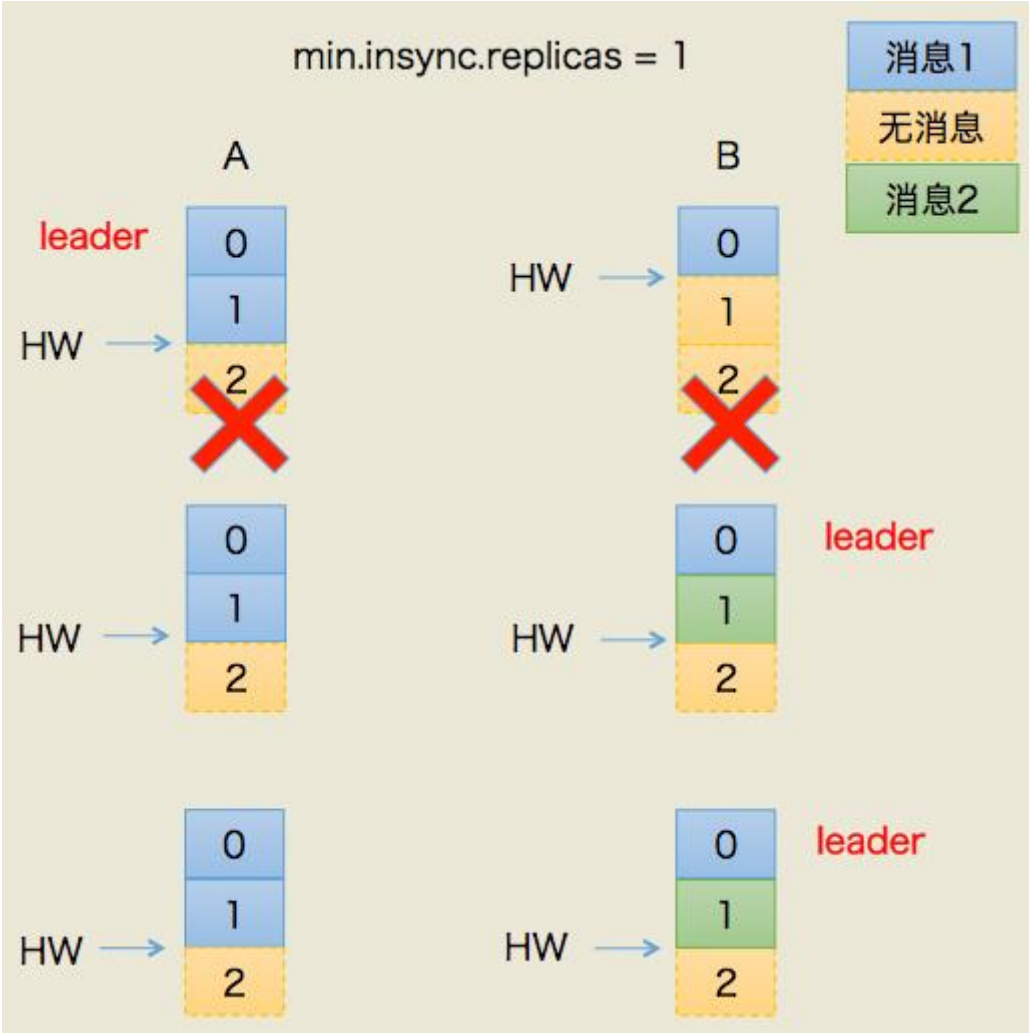
B重启之后需要给A发FETCH请求，但若A所在broker机器在此时宕机，那么Kafka会令B成为新的leader，而当A重启回来后也会执行日志截断，将HW调整回1。这样，位移=1的消息就从两个副本的log中被删除，即永远地丢失了。

这个场景丢失数据的前提是在min.insync.replicas=1时，一旦消息被写入leader端log即被认为是“已提交”，而延迟一轮FETCH RPC更新HW值的设计使得follower HW值是异步延迟更新的，倘若在这个过程中leader发生变更，那么成为新leader的follower的HW值就有可能是过期的，使得clients端认为是成功提交的消息被删除。

2）leader/follower数据离散

除了可能造成的数据丢失以外，这种设计还有一个潜在的问题，即造成leader端log和follower端log的数据不一致。比如leader端保存的记录序列是r1,r2,r3,r4,r5,....；而follower端保存的序列可能是r1,r3,r4,r5,r6...。这也是非法的场景，因为顾名思义，follower必须追随leader，完整地备份leader端的数据。

我们依然使用一张图来说明这种场景是如何发生的：



这种情况的初始状态与情况1有一些不同的：A依然是leader，A的log写入了2条消息，但B的log只写入了1条消息。分区HW更新到2，但B的HW还是1，同时producer端的min.insync.replicas = 1。

这次我们让A和B所在机器同时挂掉，然后假设B先重启回来，因此成为leader，分区HW = 1。假设此时producer发送了第3条消息(绿色框表示)给B，于是B的log中offset = 1的消息变成了绿色框表示的消息，同时分区HW更新到2（A还没有回来，就B一个副本，故可以直接更新HW而不用理会A）之后A重启回来，需要执行日志截断，但发现此时分区HW=2而A之前的HW值也是2，故不做任何调整。此后A和B将以这种状态继续正常工作。

显然，这种场景下，A和B底层log中保存在offset = 1的消息是不同的记录，从而引发不一致的情形出现。

关于上述问题的解决方案：

造成上述两个问题的根本原因在于HW值被用于衡量副本备份的成功与否以及在出现failture时作为日志截断的依据，但HW值的更新是异步延迟的，特别是需要额外的FETCH请求处理流程才能更新，故这中间发生的任何崩溃都可能导致HW值的过期。鉴于这些原因，Kafka 0.11引入了leader epoch来取代HW值。Leader端多开辟一段内存区域专门保存leader的epoch信息，这样即使出现上面的两个场景也能很好地规避这些问题。

所谓leader epoch实际上是一对值：(epoch，offset)。epoch表示leader的版本号，从0开始，当leader变更过1次时epoch就会+1，而offset则对应于该epoch版本的leader写入第一条消息的位移。因此假设有两对值：

(0, 0)和(1, 120)

则表示第一个leader从位移0开始写入消息；共写了120条[0, 119]；而第二个leader版本号是1，从位移120处开始写入消息。

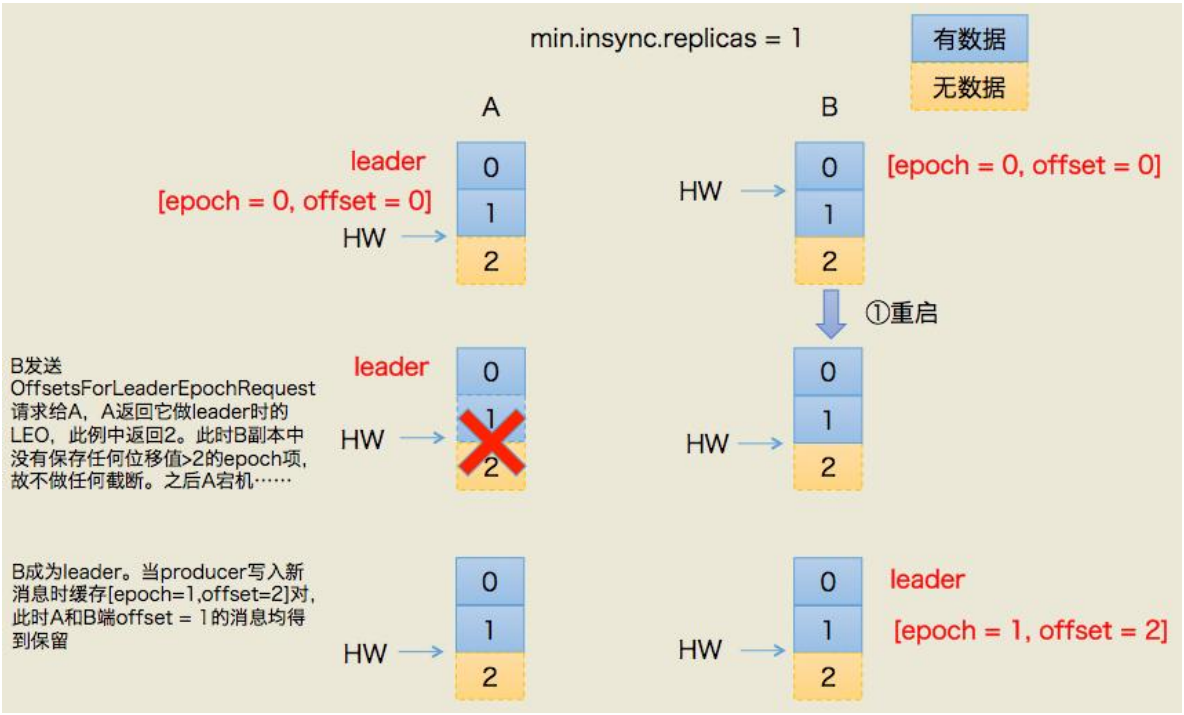
leader broker中会保存这样的一个缓存，并定期地写入到一个checkpoint文件中。

公众号：旧时光大数据

当leader写底层log时它会尝试更新整个缓存——如果这个leader首次写消息，则会在缓存中增加一个条目；否则就不做更新。而每次副本重新成为leader时会查询这部分缓存，获取出对应leader版本的位移，这就不会发生数据不一致和丢失的情况。

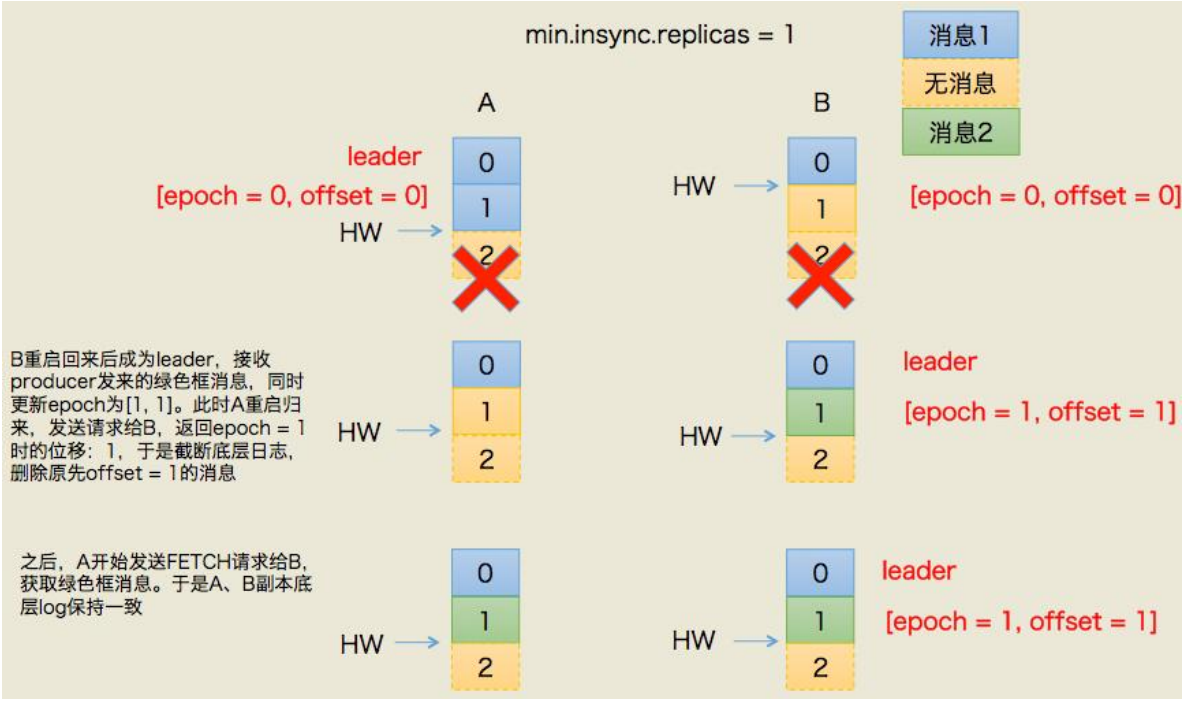
下面使用图的方式来说明下利用leader epoch如何规避上述两种情况

1）规避数据丢失



上图左半边已经给出了简要的流程描述，这里不详细展开具体的leader epoch实现细节（比如OffsetsForLeaderEpochRequest的实现），我们只需要知道每个副本都引入了新的状态来保存自己当leader时开始写入的第一条消息的offset以及leader版本。这样在恢复的时候完全使用这些信息而非水位来判断是否需要截断日志。

2）规避数据不一致



同样的道理，依靠leader epoch的信息可以有效地规避数据不一致的问题。

总结：0.11.0.0版本的Kafka通过引入leader epoch解决了原先依赖水位表示副本进度可能造成的数据丢失/数据不一致问题。有兴趣的读者可以阅读源代码进一步地了解其中的工作原理。

公众号：旧时光大数据

Kafka的分区器、拦截器、序列化器?

问过的一些公司：ebay

参考答案：

Kafka中，先执行拦截器对消息进行相应的定制化操作，然后执行序列化器将消息序列化，最后执行分区器选择对应分区

拦截器 -> 序列化器 -> 分区器

1、拦截器

Kafka有两种拦截器：生产者拦截器和消费者拦截器

生产者拦截器既可以用来在消息发送前做一些准备工作，比如按照某个规定过滤不符合要求的消息、修改消息内容等，也可以用来在发送回调逻辑前做一些定制化的需求，比如统计类工作。

生产者拦截器的实现，主要是自定义实现 org.apache.kafka.clients.producer. ProducerInterceptor 接口。ProducerInterceptor接口包含3个方法：

```
ProducerRecord<K, V> onSend(ProducerRecord<K, V> var1);
void onAcknowledgement(RecordMetadata var1, Exception var2);
void close();

```

KafkaProducer 在将消息序列化和计算分区之前会调用生产者拦截器的onSend()方法来对消息进行相应的定制化操作。一般来说最好不要修改消息 ProducerRecord 的 topic、key 和 pritition 等信息，如果修改需要保证对其有准确判断，否则会出现与预想不一致的偏差。比如修改 key 不仅会影响分区的计算还会影响 broker 端日志压缩（Log Compaction）功能。

KafkaProducer 会在消息被应答（Acknowledgement）之前或消息发送失败时调用生产者拦截器的 onAcknowledgement() 方法，优先于用户设定的 Callback 之前执行。**这个方法运行在 Producer的I/O线程中，所以这个方法的实现逻辑约简单越好，否则会影响消息的发送。**

close()方法主要用于在关闭拦截器时执行一些资源的清理工作。在这3个方法中抛出的异常都会被捕获并记录到日志中，但并不会再向上传递。

ProducerInterceptor接口与Protitioner 接口一样都有一个父接口Configurable。

Kafka中不仅可以指定一个拦截器还可以指定多个拦截器形成一个拦截器链。拦截器链会根据配置时配置的拦截器顺序来执行（配置的时候，各个拦截器之间使用逗号隔开）。

如果拦截器链中的某个拦截器的执行需要依赖上一个拦截器的输出，那么就有可能产生“副作用”。如果第一个拦截器因为异常执行失败，那么第二个也就不能继续执行。**在拦截链中，如果某个拦截器执行失败，那么下一个拦截器会接着从上一个执行成功的拦截器继续执行。**

2、序列化

生产者需要用序列化器（Serializer）将key和value序列化成字节数组才可以将消息传入Kafka。消费者需要用反序列化器（Deserializer）把从Kafka中收到的字节数组转化成相应的对象。在代码清单3-1中，key和value都使用了字符字符串，对应程序中的序列化器也使用了客户端自带的 StringSerializer，除了字符串类型的序列化器，还有 ByteArray、ByteBuffer、Bytes、Double、Integer、Long 这几种类型，它们都实现了 org.apache.kafka.common.serialization.Serializer 接口，此接口有3个方法：

公众号：旧时光大数据

```
// 用来配置当前类
public void configure(Map<String, ?> configs, boolean isKey)
// 用来执行序列化操作
public byte[] serializer(String topic, T data)
// 用来关闭当前的序列化器
public void close()

```

一般情况下 close() 是个空方法，如果实现了此方法，则必须确保此方法的幂等性（一次和多次请求某一个资源对于资源本身应该具有同样的结果（网络超时等问题除外）。也就是说，其任意多次执行对资源本身所产生的影响均与一次执行的影响相同。），因为这个方法可能会被 KafkaProducer调用多次（Serializer和KafkaProducer 所实现的接口都继承了 Closeable 接口）。

生产者使用的序列化和消费者使用的序列化是一一对应的，如果生产者使用了 StringSerializer，而消费者使用了另一种序列化器，那么是无法解析出相要数据的。

如果Kafka客户端提供的几种序列化器都无法满足应用需求，可以选择如 Avro、JSON等通用的序列化工具实现，或者使用自定义类型的序列化器来实现。

3、分区器

消息在通过 send() 发往 broker 的过程中，有可能需要经过拦截器（Interceptor）、序列化器（Serializer）和分区器（Partitioner）的一系列作用之后才能被真正地发往 broker。拦截器一般不是必须的，而序列化器是必须的。消息经过序列化之后就需要确定它发往的分区，如果消息 ProducerRecord 中指定了 partition 字段，那么就不需要分区器的作用，因为 partition 代表的就是要发往的分区号。

如果 ProducerRecode 中没有指定 partition 字段，那么就需要依赖分区器，根据 key 这个字段来计算 partition 的值。**分区器的作用就是为消息分配分区。**

Kafka 中提供的默认分区器是 org.apache.kafka.clients.producer.internals.DefaultPartitioner，它实现了 org.apache.kafka.clients.producer.Partitioner 接口，这个接口中定义了2个方法，具体如下所示。

```
public int partition(String topic, Object key, byte[] keyBytes, Object value,
byte[] valueBytes, Cluster cluster);
public void close();

```

partition() 方法用来计算分区号，返回值为ini类型。方法中的参数分别表示主题、键、序列化后的键、值、序列化后的值以及集群的元数据信息，通过这些信息可以实现分区器。close()方法在关闭分区器的时候回收一些资源。

Partitioner接口还有一个父接口 org.apache.kafka.common.Configurable，该接口只有一个方法

```
void configure(Map<String, ?> var1);

```

Configurable 接口中的 configure() 方法主要是用来获取配置信息及初始化数据。

在默认分区器 DefaultPartitioner 的实现中，close()是空方法，而 partition() 方法中定义了主要的分区分配逻辑。如果 key 不为 null，那么默认分区器会对 key 进行哈希（采用 MurmurHash2 算法，具备高运算性能及低碰撞率）根据最终得到的哈希值，与分区的数量取模运算得到分区编号来匹配分区，相同key得到的哈希值是一样的，所以当key一致，分区数量不变的情况下，会将消息写入同一个分区（注意：在不改变主题分区数量的情况下，key 与分区之间的映射可以保持不变。不过，一旦主题增加了分区，那么就难以保证key与分区的映射关系）。如果，key 是 null，那么消息会以轮询的方式写入分区。（注意：如果 key 不为null，那么计算得到的分区号会是所有分区中的一个。如果 key 为 null 并且有可用的分区的时候，那么计算得到的分区号仅为可用分区中的任意一个。）

公众号：旧时光大数据

除了使用 Kafka 提供的默认分区器进行分配，还可以使用自定义的分区器，只需要和 DefaultPartitioner 一样 实现 Partitioner 接口即可。默认的分区器在 key 为 null 不会选择不可用的分区，我们通过自定义分区器来实现。

Kafka连接Spark Streaming的几种方式

问过的一些公司：作业帮

参考答案：

Spark Streaming获取kafka数据的两种方式：**Receiver与Direct的方式，可以从代码中简单理解成Receiver方式是通过zookeeper来连接kafka队列，Direct方式是直接连接到kafka的节点上获取数据。**

1、基于Receiver的方式

这种方式使用Receiver来获取数据。Receiver是使用Kafka的高层次Consumer API来实现的。receiver从Kafka中获取的数据都是存储在Spark Executor的内存中的，然后Spark Streaming启动的job会去处理那些数据。

然而，在默认的配置下，这种方式可能会因为底层的失败而丢失数据。如果要启用高可靠机制，让数据零丢失，就必须启用Spark Streaming的预写日志机制（Write Ahead Log，WAL）。该机制会同步地将接收到的Kafka数据写入分布式文件系统（比如HDFS）上的预写日志中。所以，即使底层节点出现了失败，也可以使用预写日志中的数据进行恢复。

注意：

- Kafka中的topic的partition，与Spark中的RDD的partition是没有关系的。所以，在KafkaUtils.createStream()中，提高partition的数量，只会增加一个Receiver中，读取partition的线程的数量。不会增加Spark处理数据的并行度。
- 可以创建多个Kafka输入DStream，使用不同的consumer group和topic，来通过多个receiver并行接收数据。
- 如果基于容错的文件系统，比如HDFS，启用了预写日志机制，接收到的数据都会被复制一份到预写日志中。因此，在KafkaUtils.createStream()中，设置的持久化级别是StorageLevel.MEMORY_AND_DISK_SER。

2、基于Direct的方式

这种新的不基于Receiver的直接方式，是在Spark 1.3中引入的，从而能够确保更加健壮的机制。替代掉使用Receiver来接收数据后，这种方式会周期性地查询Kafka，来获得每个topic+partition的最新的offset，从而定义每个batch的offset的范围。当处理数据的job启动时，就会使用Kafka的简单consumer api来获取Kafka指定offset范围的数据。

这种方式有如下优点：

1）简化并行读取

如果要读取多个partition，不需要创建多个输入DStream然后对它们进行union操作。Spark会创建跟Kafka partition一样多的RDD partition，并且会并行从Kafka中读取数据。所以在Kafka partition和RDD partition之间，有一个一对一的映射关系。

2）高性能

如果要保证零数据丢失，在基于receiver的方式中，需要开启WAL机制。这种方式其实效率低下，因为数据实际上被复制了两份，Kafka自己本身就有高可靠的机制，会对数据复制一份，而这里又会复制一份到WAL中。而基于direct的方式，不依赖Receiver，不需要开启WAL机制，只要Kafka中作了数据的复制，那么就可以通过Kafka的副本进行恢复。

3）一次且仅一次的事务机制

基于receiver的方式，是使用Kafka的高阶API来在ZooKeeper中保存消费过的offset的。这是消费Kafka数据的传统方式。这种方式配合着WAL机制可以保证数据零丢失的高可靠性，但是却无法保证数据被处理一次且仅一次，可能会处理两次。因为Spark和ZooKeeper之间可能是不同步的。

4）降低资源

Direct不需要Receivers，其申请的Executors全部参与到计算任务中；而Receiver-based则需要专门的Receivers来读取Kafka数据且不参与计算。因此相同的资源申请，Direct 能够支持更大的业务。

5）降低内存

Receiver-based的Receiver与其他Exectuor是异步的，并持续不断接收数据，对于小业务量的场景还好，如果遇到大业务量时，需要提高Receiver的内存，但是参与计算的Executor并无需那么多的内存。而Direct 因为没有Receiver，而是在计算时读取数据，然后直接计算，所以对内存的要求很低。实际应用中我们可以把原先的10G降至现在的2-4G左右。

6）鲁棒性更好

Receiver-based方法需要Receivers来异步持续不断的读取数据，因此遇到网络、存储负载等因素，导致实时任务出现堆积，但Receivers却还在持续读取数据，此种情况很容易导致计算崩溃。Direct 则没有这种顾虑，其Driver在触发batch 计算任务时，才会读取数据并计算。队列出现堆积并不会引起程序的失败。

基于direct的方式，使用kafka的简单api，Spark Streaming自己就负责追踪消费的offset，并保存在checkpoint中。Spark自己一定是同步的，因此可以保证数据是消费一次且仅消费一次。

Kafka的生成者客户端有几个线程?

问过的一些公司：昆仑万维

参考答案：

2个，主线程和Sender线程。

主线程负责创建消息，然后通过分区器、序列化器、拦截器作用之后缓存到累加器RecordAccumulator中。

Sender线程负责将RecordAccumulator中消息发送到kafka中。

Kafka怎么防止脑裂

问过的一些公司：网易

参考答案：

1、先说下什么是kafka controller

控制器（controller）其实就是一个broker ，只不过它除了具有一般broker的功能之外，还负责分区首领的选举，相当于整个kafka集群的**master**，负责topic的创建、删除、以及partition的状态机转换，broker的上线、下线等。集群里第一个启动的broker通过在Zookeeper里创建一个临时节点/controller 让自己成为控制器。其它broker在启动时也会尝试创建这个节点，不过它们会收到一个“节点已存在”的异常，然后“意识”到控制器节点已存在，也就是说集群里已经有一个控制器了。其他broker在控制器节点上创建Zookeeper watch对象，这样它们就可以收到这个节点的变更通知。这种方式可以确保集群次只有一个控制器存在。

2、什么是脑裂

公众号：旧时光大数据

kafka中只有一个控制器controller 负责分区的leader选举，同步broker的新增或删除消息，但有时由于网络问题，可能同时有两个broker认为自己是controller，这时候其他的broker就会发生脑裂，不知道该听从谁的。

3、如何解决脑裂

通过controller epoch来解决。

每当新的controller产生时就会通过Zookeeper生成一个全新的、数值更大的controller epoch标识。其他broker在知道当前controller epoch后，如果收到由控制器发出的包含较旧epoch的消息，就会忽略它们。

Kafka高可用体现在哪里

问过的一些公司：

参考答案：

高可性(High Availability)，指系统无间断地执其功能。

Kafka从0.8版本开始提供高可机制，可**保障单个或多个Broker宕机后，其他Broker及所有Partition都能继续提供服务，避免存储的消息丢失。**

对分布式系来说，当集群规模上升到一定程度后，一台或者多台机宕机的可能性大增加；**Kafka采多机备份和消息应答确认方式解决了数据丢失问题，并通过一套失败恢复机制解决服务可问题。**

Zookeeper在Kafka的作用

可回答：Kafka在什么地方需要用到Zookeeper

问过的一些公司：字节x2，阿里，小米，有赞

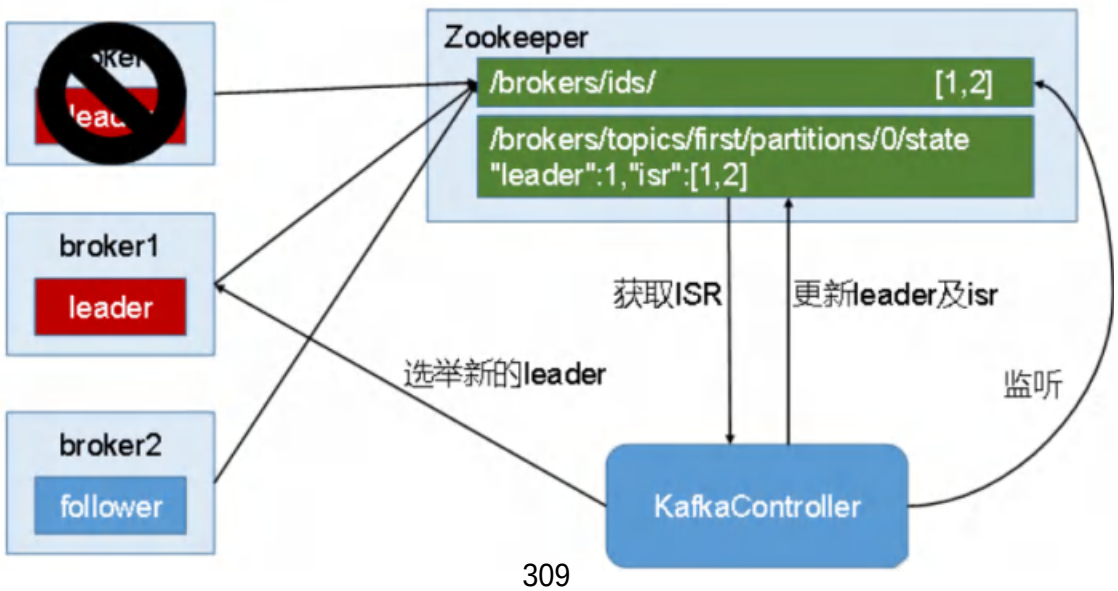
参考答案：

简洁版：

Kafka集群中有一个broker会被选举为Controller，负责管理集群broker的上下线，所有topic的分区副本分配和leader选举等工作。

Controller的管理工作都是依赖于Zookeeper的。也就是说ZK是辅助Controller的管理工作的。

以下为partition的leader选举过程：



公众号：旧时光大数据

详细版：

1、Broker注册

Broker是分布式部署并且相互之间相互独立，但是需要有一个注册系统能够将整个集群中的Broker管理起来，此时就使用到了Zookeeper。在Zookeeper上会有一个专门**用来进行Broker服务器列表记录**的节点：

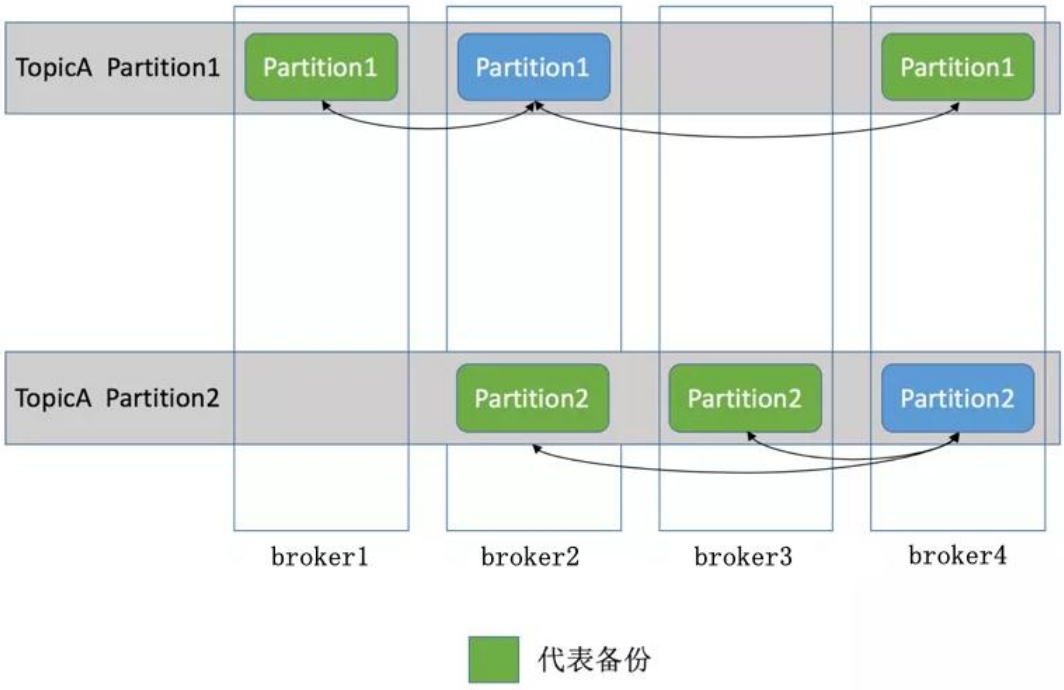
```
/brokers/ids
```

每个Broker在启动时，都会到Zookeeper上进行注册，即到/brokers/ids下创建属于自己的节点，如/brokers/ids/[0...N]。

Kafka使用了全局唯一的数字来指代每个Broker服务器，不同的Broker必须使用不同的Broker ID进行注册，创建完节点后，**每个Broker就会将自己的IP地址和端口信息记录**到该节点中去。其中，Broker创建的节点类型是临时节点，一旦Broker宕机，则对应的临时节点也会被自动删除。

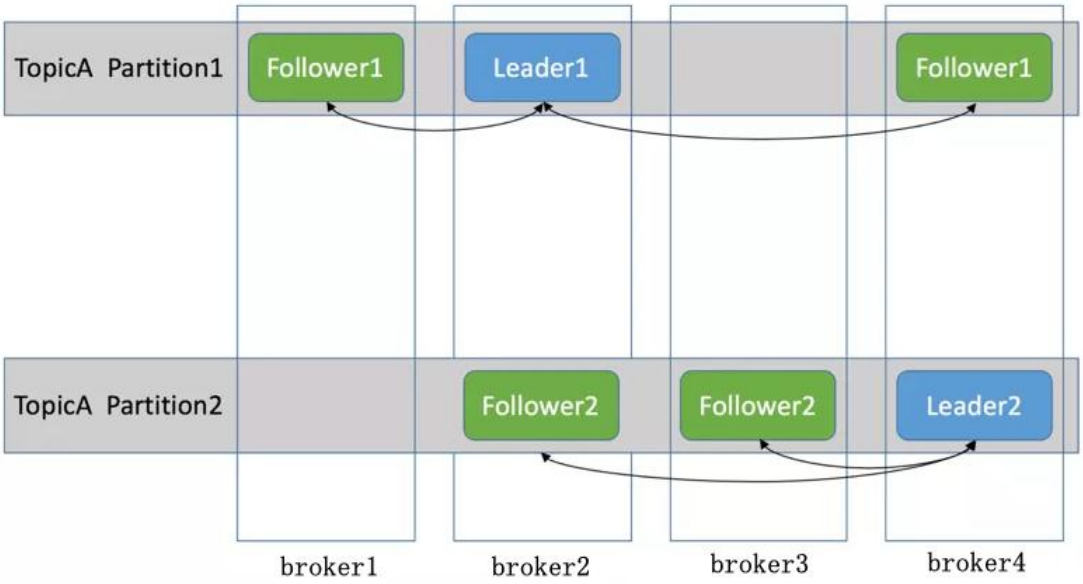
2、Topic注册

在 Kafka 中，所有 Topic 与 Broker 的对应关系都由 ZooKeeper 来维护，在 ZooKeeper 中，通过建立专属的节点来存储这些信息，其路径为： /brokers/topics/{topic_name}



为了保障数据的一致性，ZooKeeper 机制得以引入。基于 ZooKeeper，Kafka 为每一个 Partition 找一个节点作为 Leader，其余备份作为 Follower；接续上图的例子，就 TopicA 的 Partition1 而言，如果位于 Broker2（Kafka 节点）上的 Partition1 为 Leader，那么位于 Broker1 和 Broker4 上面的 Partition1 就充当 Follower，则有下图：

公众号：旧时光大数据



基于上图的架构，当 Producer Push 的消息写入 Partition（分区）时，作为 Leader 的 Broker（Kafka 节点）会将消息写入自己的分区，同时还会将此消息复制到各个 Follower，实现同步。如果某个 Follower 挂掉，Leader 会再找一个替代并同步消息；如果 Leader 挂了，Follower 们会选举出一个新的 Leader 替代，继续业务，这些都是由 ZooKeeper 完成的。

3、生产者负载均衡

由于同一个Topic消息会被分区并将其分布在多个Broker上，因此，**生产者需要将消息合理地发送到这些分布式的Broker上**，那么如何实现生产者的负载均衡，Kafka支持传统的四层负载均衡，也支持Zookeeper方式实现负载均衡。

1）四层负载均衡，根据生产者的IP地址和端口来为其确定一个相关联的Broker。通常，一个生产者只会对应单个Broker，然后该生产者产生的消息都发往该Broker。这种方式逻辑简单，每个生产者不需要同其他系统建立额外的TCP连接，只需要和Broker维护单个TCP连接即可。但是，其无法做到真正的负载均衡，因为实际系统中的每个生产者产生的消息量及每个Broker的消息存储量都是不一样的，如果有些生产者产生的消息远多于其他生产者的话，那么会导致不同的Broker接收到的消息总数差异巨大，同时，生产者也无法实时感知到Broker的新增和删除。

2)）使用Zookeeper进行负载均衡，由于每个Broker启动时，都会完成Broker注册过程，生产者会通过该节点的变化来动态地感知到Broker服务器列表的变更，这样就可以实现动态的负载均衡机制。

4、消费者负载均衡

与生产者类似，Kafka中的消费者同样需要进行负载均衡来实现多个消费者合理地从对应的Broker服务器上接收消息，每个消费者分组包含若干消费者，**每条消息都只会发送给分组中的一个消费者**，不同的消费者分组消费自己特定的Topic下面的消息，互不干扰。

5、分区与消费者的关系

消费组 (Consumer Group)：

consumer group下有多个Consumer（消费者）。对于每个消费者组 (Consumer Group)，Kafka都会为其分配一个全局唯一的Group ID，Group内部的所有消费者共享该ID。订阅的topic下的每个分区只能分配给某个group下的一个consumer(当然该分区还可以被分配给其他group)。

同时，Kafka为每个消费者分配一个Consumer ID，通常采用"Hostname:UUID"形式表示。

在Kafka中，规定了**每个消息分区 只能被同组的一个消费者进行消费**，因此，需要在Zookeeper上记录消息分区与Consumer之间的关系，每个消费者一旦确定了对一个消息分区的消费权力，需要将其Consumer ID写入到Zookeeper对应消息分区的临时节点上，例如：

公众号：旧时光大数据

```
/consumers/[group_id]/owners/[topic]/[broker_id-partition_id]
```

其中，[broker_id-partition_id]就是一个消息分区的标识，节点内容就是该消息分区上消费者的Consumer ID。

6、消费者注册

消费者服务器在初始化启动时加入消费者分组的步骤如下：

注册到消费者分组。每个消费者服务器启动时，都会到Zookeeper的指定节点下创建一个属于自己的消费者节点，例如/consumers/[group_id]/ids/[consumer_id]，完成节点创建后，消费者就会将自己订阅的Topic信息写入该临时节点。

7、记录Partition与Consumer的关系

Consumer Group 在 ZooKeeper 上的注册节点为 /consumers/[group_id]，而 Consumer Group 中的 Consumer 在 ZooKeeper 上的注册节点为 /consumers/[group_id] 下的子节点 owners，它们共享一个 Group ID。为了 Consumer 负载均衡，同一个 Group 订阅的 Topic 下的任一 Partition 都只能分配给一个 Consumer。Partition 与 Consumer 的对应关系也需要在 ZooKeeper 中记录，路径为：

```
/consumers/[group_id]/owners/[topic]/[broker_id-partition_id]
```

这个路径也是一个临时节点，进行 Rebalance 时会被删除，而后依据新的对应关系重建。此外，[broker_id-partition_id] 是一个消息分区的标识，其内容就是该消息分区消费者的 Consumer ID，通常采用 hostname:UUID 形式表示。

HBase面试题

介绍下HBase

问过的一些公司：远景，字节，百度(2021.09)

参考答案：

HBase是一种**分布式、可扩展、支持海量数据存储的非关系型分布式数据库**，它参考了谷歌的BigTable建模，主要用来存储非结构化和半结构化的松散数据，实现的编程语言为Java。它是Apache软件基金会的Hadoop项目的一部分，运行于HDFS文件系统之上，为 Hadoop 提供类似于BigTable 规模的服务。因此，它**可以容错地存储海量稀疏的数据。**

HBase的**目标是处理非常庞大的表**，可以通过水平扩展的方式，利用廉价计算机集群处理由超过10亿行数据和数百万列元素组成的数据表。

应用场景

1）交通方面

船舶GPS信息，全长江的船舶GPS信息，每天有1千万左右的数据存储。

2）金融方面

消费信息、贷款信息、信用卡还款信息等

3）电商方面

电商网站的交易信息、物流信息、游览信息等

公众号：旧时光大数据

4）电信方面

通话信息、语音详单等

HBase优缺点

问过的一些公司：美团，快手，东软集团

参考答案：

优点

1）海量存储

Hbase适合存储PB级别的海量数据，在PB级别的数据以及采用廉价PC存储的情况下，能在几十到百毫秒内返回数据。这与Hbase的极易扩展性息息相关。正式因为Hbase良好的扩展性，才为海量数据的存储提供了便利。

2）列式存储

这里的列式存储其实说的是列族（ColumnFamily）存储，Hbase是根据列族来存储数据的。列族下面可以有非常多的列，列族在创建表的时候就必须指定。

3）极易扩展

Hbase的扩展性主要体现在两个方面，一个是基于上层处理能力（RegionServer）的扩展，一个是基于存储的扩展（HDFS）。

通过横向添加RegionSever的机器，进行水平扩展，提升Hbase上层的处理能力，提升Hbsae服务更多Region的能力。

备注：RegionServer的作用是管理region、承接业务的访问，这个后面会详细的介绍通过横向添加Datanode的机器，进行存储层扩容，提升Hbase的数据存储能力和提升后端存储的读写能力。

4）高并发（多核）

由于目前大部分使用Hbase的架构，都是采用的廉价PC，因此单个IO的延迟其实并不小，一般在几十到上百ms之间。这里说的高并发，主要是在并发的情况下，Hbase的单个IO延迟下降并不多。能获得高并发、低延迟的服务。

5）稀疏

稀疏主要是针对Hbase列的灵活性，在列族中，你可以指定任意多的列，在列数据为空的情况下，是不会占用存储空间的。

缺点

1）原生不支持SQL

SQL查询也是HBase的一个弱项，虽然HBase是一个非关系型数据库但是它不支持SQL语句，不过这块可以通过引入Phoenix解决，Phoenix是专为HBase设计的SQL层。

2）原生不支持二级索引

单一RowKey固有的局限性决定了它不可能有效地支持多条件查询，只支持按照Row key来查询，因此正常情况下对非Rowkey列做查询比较慢。所以，我们一般会选择一个HBase二级索引解决方案，目前比较成熟的解决方案是Phoenix，此外还可以选择Elasticsearch/Solr等搜索引擎自己设计实现。

3）暂时不能支持Master server的故障切换, 当Master宕机后, 整个存储系统就会挂掉

4）数据分析能力弱

公众号：旧时光大数据

数据分析是HBase的弱项，比如聚合运算、多维度复杂查询、多表关联查询等。所以，我们一般在HBase之上架设Phoenix或Spark等组件，增强HBase数据分析处理的能力。

说下HBase原理

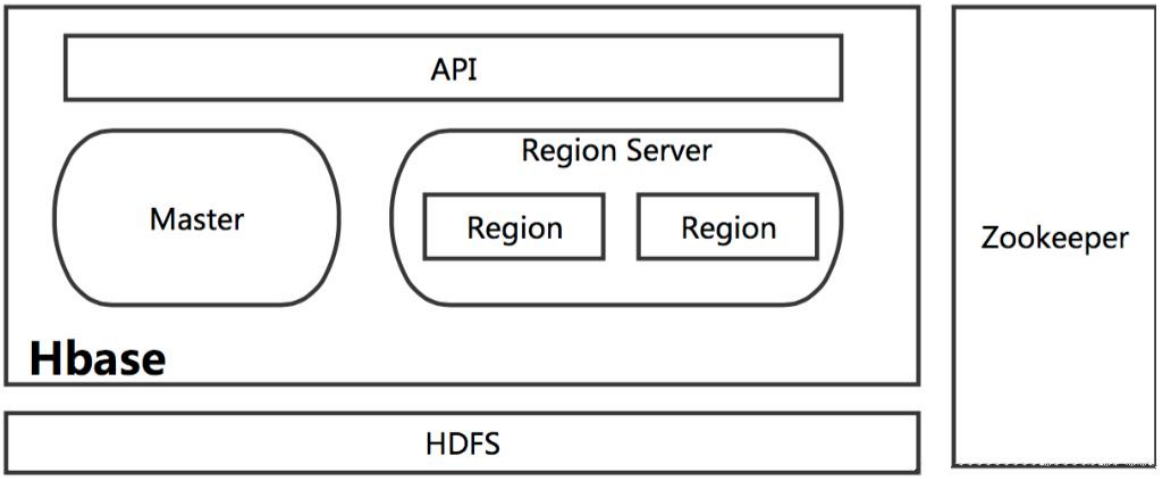
可回答：结合第一题的HBase介绍回答

问过的一些公司：百度，vivo，阿里云，触宝，哔哩哔哩，京东，携程

参考答案：

HBase是大数据NoSQL领域里非常重要的分布式KV数据库，是一个高可靠、高性能、高伸缩的分布式存储系统。

1、HBase模块



HBase相关模块的作用：

1）Master

Hbase Master用于协调多个Region Server，侦测各个RegionServer之间的状态，并平衡RegionServer之间的负载。HbaseMaster还有一个职责就是负责分配Region给RegionServer。Hbase允许多个Master节点共存，但是这需要Zookeeper的帮助。不过当多个Master节点共存时，只有一个Master是提供服务的，其他的Master节点处于待命的状态。当正在工作的Master节点宕机时，其他的Master则会接管Hbase的集群。

2）Region Server

对于一个RegionServer而言，其包括了多个Region。RegionServer的作用只是管理表格，以及实现读写操作。Client直接连接RegionServer，并通信获取Hbase中的数据。对于Region而言，则是真实存放Hbase数据的地方，也就说Region是Hbase可用性和分布式的基本单位。如果当一个表格很大，并由多个CF组成时，那么表的数据将存放在多个Region之间，并且在每个Region中会关联多个存储的单元（Store）。

3）Zookeeper

对于 Hbase 而言，Zookeeper的作用是至关重要的。首先Zookeeper是作为Hbase Master的HA解决方案。也就是说，是Zookeeper保证了至少有一个Hbase Master 处于运行状态。并且Zookeeper负责Region和Region Server的注册。其实Zookeeper发展到目前为止，已经成为了分布式大数据框架中容错性的标准框架。不光是Hbase，几乎所有的分布式大数据相关的开源框架，都依赖于Zookeeper实现HA。

2、HBase架构

参考下一题

3、HBase优缺点

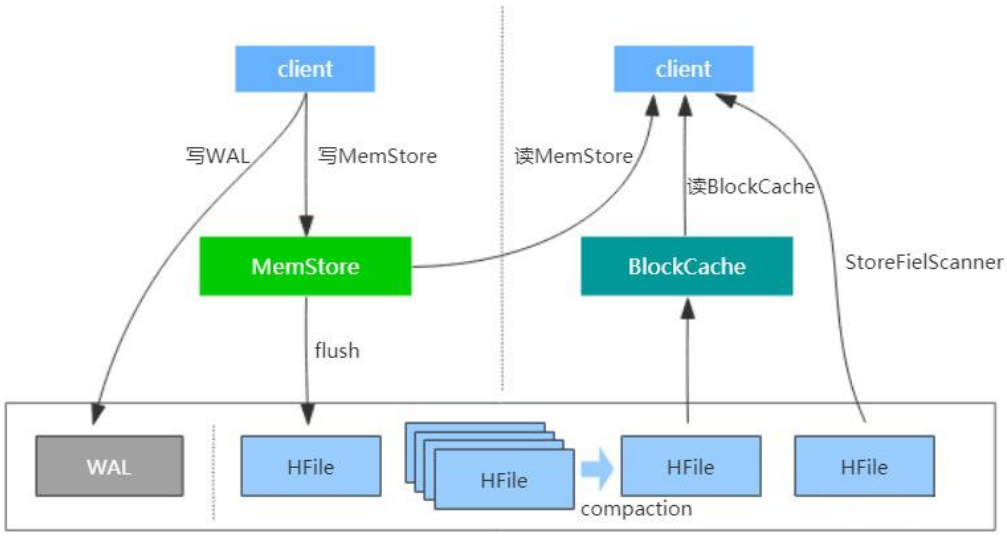
公众号：旧时光大数据

见上一题

4、HBase核心原理

1）存储引擎

HBase是Google的BigTable的开源实现，底层存储引擎是基于LSM-Tree数据结构设计的。写入数据时会先写WAL日志，再将数据写到写缓存MemStore中，等写缓存达到一定规模后或满足其他触发条件才会flush刷写到磁盘，这样就将磁盘随机写变成了顺序写，提高了写性能。每一次刷写磁盘都会生成新的HFile文件。可以参考如下的原理图：



随着时间推移，写入的HFile会越来越多，查询数据时就会因为要进行多次io导致性能降低，为了提升读性能，HBase会定期执行compaction操作以合并HFile。此外，HBase在读路径上也有诸多设计，其中一个重要的点是设计了BlockCache读缓存。这样以后，读取数据时会依次从BlockCache、MemStore以及HFile中seek数据，再加上一些其他设计比如布隆过滤器、索引等，保证了HBase的高性能。

2）数据模型

关于HBase的数据模型，和关系型数据类似，包括命名空间（namespace）、表、行、列、列族、列限定符、单元格（cell）、时间戳等，具体概念比较好理解就不多解释了。而HBase在实际存储数据的时候是以有序KV的形式组织的。

Rowkey	列族1 f			列族2 info	
	id	status	phone_num	city	area
row1	2340	1	13506789033	tianjin	
row2	3456	3		shanghai	浦东
row3	3459	2	15365569742	beijing	海淀

每一个cell单元格基本组成

这里重点从KV这个角度切入，Value是实际写入的数据，比较好理解。其中Key则是由Rowkey、Column Family : Column Qualifier、Timestamp、Type等几个维度组成，其中rowkey是HBase的行键；column family（列族）与qualifier（列限定符即列名）共同组成了HBase的列；timestamp表示的就是数据写入时的时间戳，主要用于标识HBase数据的版本号；type代表Put/Delete的操作类型，说明一点，HBase删除是给数据打上delete marker，在数据合并时才会真正物理删除。此外，HBase的表具有稀疏特性，一行中空值的列并不占用任何存储空间。

3）列族式存储

HBase并不是行式存储，也不是完全的列式存储，而是面向列族的列族式存储。前面也提到了，HBase的每一列数据在底层都是以 KV 形式存储的，而针对一行数据，同一列族的不同列的数据是顺序相邻存放的，这种模式实际上是行式存储；而如果一个列族下只有一个列的话，就是一种列式存储。因此我们可以说HBase是一种列族式存储。

4）关于索引

默认情况下HBase只对rowkey做了单列索引，所以HBase能通过rowkey进行高效的单点查询及小范围扫描。HBase索引还是比较单一的，通过非rowkey列查询性能比较低，除非对非Rowkey列做二级索引，否则不建议根据非rowkey列做查询。

HBase的二级索引一般是基于HBase协处理器实现，目前比较成熟的方案可以使用Phoenix，Phoenix不仅能够为HBase提供二级索引能力，还扮演着HBase的SQL层，增强了HBase即席查询的能力。

5、HBase应用场景

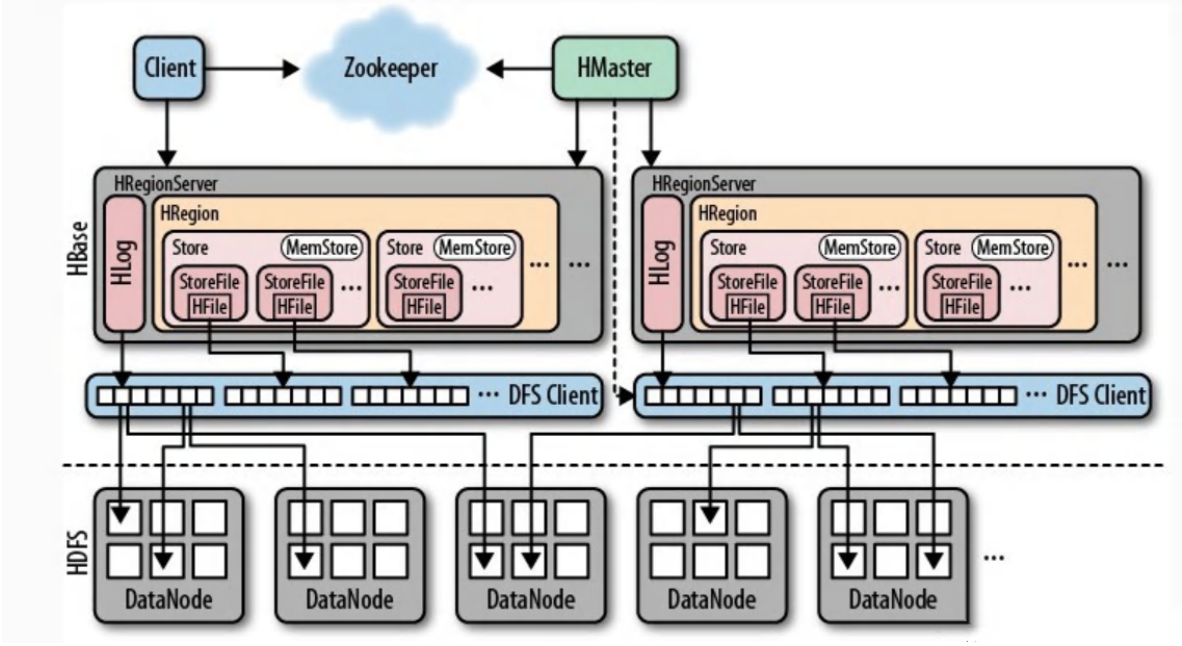
HBase经常应用在订单/消息存储、用户画像、搜索推荐、社交Feed流、安全风控、以及物联网时序数据等诸多场景。

介绍下HBase架构

可回答：HBase组件

问过的一些公司：阿里x2，字节，顺丰x2，网易云音乐，陌陌，美团，快手，有赞，科大讯飞，海康威视x2，多益，虎牙(2021.08)

参考答案：



从Hbase的架构图上可以看出，Hbase中的存储包括HMaster、HRegionSever、HRegion、HLog、Store、MemStore、StoreFile、HFile等。

Hbase中的每张表都通过键按照一定的范围被分割成多个子表（HRegion），默认一个HRegion超过256M就要被分割成两个，这个过程由HRegionServer管理,而HRegion的分配由HMaster管理。

1）HMaster的作用：

- 为HRegionServer分配HRegion
- 负责HRegionServer的负载均衡

公众号：旧时光大数据

- 发现失效的HRegionServer并重新分配
- HDFS上的垃圾文件回收
- 处理Schema更新请求

2）HRegionServer的作用：

- 维护HMaster分配给它的HRegion，处理对这些HRegion的IO请求
- 负责切分正在运行过程中变得过大的HRegion

通过架构可以得知，Client访问Hbase上的数据并不需要HMaster参与，寻址访问ZooKeeper和HRegionServer，数据读写访问HRegionServer，HMaster仅仅维护Table和Region的元数据信息，Table的元数据信息保存在ZooKeeper上，负载很低。HRegionServer存取一个子表时，会创建一个HRegion对象，然后对表的每个列簇创建一个Store对象，每个Store都会有一个MemStore和0或多个StoreFile与之对应，每个StoreFile都会对应一个HFile，HFile就是实际的存储文件。因此，一个HRegion有多少列簇就有多少个Store。

一个HRegionServer会有多个HRegion和一个HLog。

3）HRegion

Table在行的方向上分割为多个HRegion，HRegion是Hbase中分布式存储和负载均衡的最小单元，即不同的HRegion可以分别在不同的HRegionServer上，但同一个HRegion是不会拆分到多个HRegionServer上的。HRegion按大小分割，每个表一般只有一个HRegion，随着数据不断插入表，HRegion不断增大，当HRegion的某个列簇达到一个阀值（默认256M）时就会分成两个新的HRegion。

- <表名，StartRowKey, 创建时间>
- 由目录表(-ROOT-和.META.)记录该Region的EndRowKey

HRegion被分配给哪个HRegionServer是完全动态的，所以需要机制来定位HRegion具体在哪个HRegionServer，Hbase使用三层结构来定位HRegion：

- 通过zk里的文件/Hbase/rs得到-ROOT-表的位置。-ROOT-表只有一个region。
- 通过-ROOT-表查找.META.表的第一个表中相应的HRegion位置。其实-ROOT-表是.META.表的第一个region；.META.表中的每一个Region在-ROOT-表中都是一行记录。
- 通过.META.表找到所要的用户表HRegion的位置。用户表的每个HRegion在.META.表中都是一行记录。-ROOT-表永远不会被分隔为多个HRegion，保证了最多需要三次跳转，就能定位到任意的region。Client会将查询的位置信息保存缓存起来，缓存不会主动失效，因此如果Client上的缓存全部失效，则需要进行6次网络来回，才能定位到正确的HRegion，其中三次用来发现缓存失效，另外三次用来获取位置信息。

4）Store

每一个HRegion由一个或多个Store组成，至少是一个Store，Hbase会把一起访问的数据放在一个Store里面，即为每个ColumnFamily建一个Store，如果有几个ColumnFamily，也就有几个Store。一个Store由一个MemStore和0或者多个StoreFile组成。 Hbase以Store的大小来判断是否需要切分HRegion。

5）MemStore

MemStore 是放在内存里的，保存修改的数据即keyValues。当MemStore的大小达到一个阀值（默认64MB）时，MemStore会被Flush到文件，即生成一个快照。目前Hbase会有一个线程来负责MemStore的Flush操作。

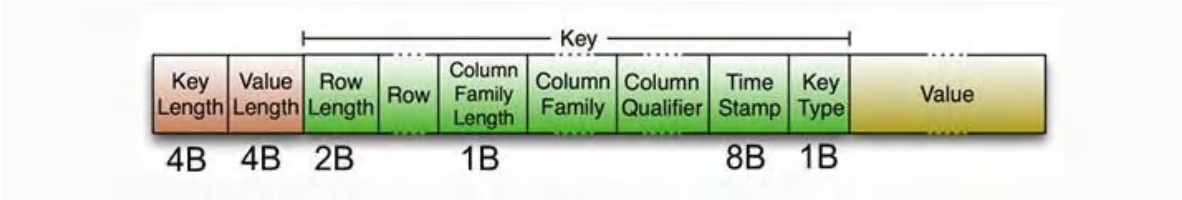
6）StoreFile

MemStore内存中的数据写到文件后就是StoreFile，StoreFile底层是以HFile的格式保存。

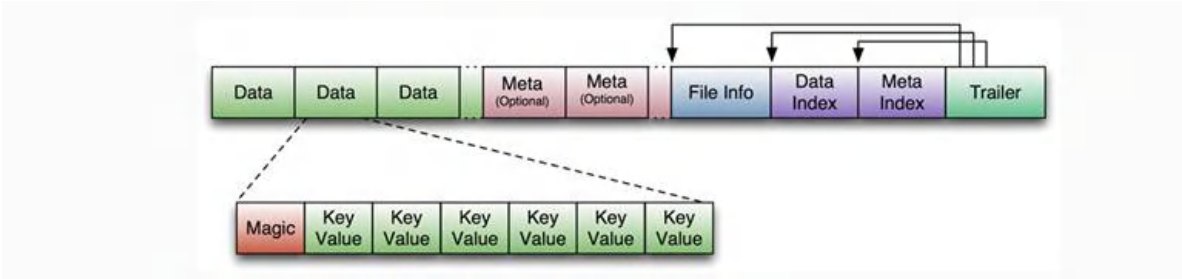
7）HFile

Hbase中KeyValue数据的存储格式，是Hadoop的二进制格式文件。 首先HFile文件是不定长的，长度固定的只有其中的两块：Trailer和FileInfo。

Trailer中有指针指向其他数据块的起始点，FileInfo记录了文件的一些meta信息。Data Block是Hbase IO的基本单元，为了提高效率，HRegionServer中有基于LRU的Block Cache机制。每个Data块的大小可以在创建一个Table的时候通过参数指定（默认块大小64KB），大号的Block有利于顺序Scan，小号的Block利于随机查询。每个Data块除了开头的Magic以外就是一个个KeyValue对拼接而成，Magic内容就是一些随机数字，目的是防止数据损坏，结构如下。



HFile结构图如下：



Data Block段用来保存表中的数据，这部分可以被压缩。 Meta Block段（可选的）用来保存用户自定义的kv段，可以被压缩。 FileInfo段用来保存HFile的元信息，不能被压缩，用户也可以在这一部分添加自己的元信息。 Data Block Index段（可选的）用来保存Meta Blcok的索引。 Trailer这一段是定长的。保存了每一段的偏移量，读取一个HFile时，会首先读取Trailer，Trailer保存了每个段的起始位置(段的Magic Number用来做安全check)，然后，DataBlock Index会被读取到内存中，这样，当检索某个key时，不需要扫描整个HFile，而只需从内存中找到key所在的block，通过一次磁盘io将整个 block读取到内存中，再找到需要的key。DataBlock Index采用LRU机制淘汰。 HFile的Data Block，Meta Block通常采用压缩方式存储，压缩之后可以大大减少网络IO和磁盘IO，随之而来的开销当然是需要花费cpu进行压缩和解压缩。（备注：DataBlock Index的缺陷：a) 占用过多内存　b) 启动加载时间缓慢）

8）HLog

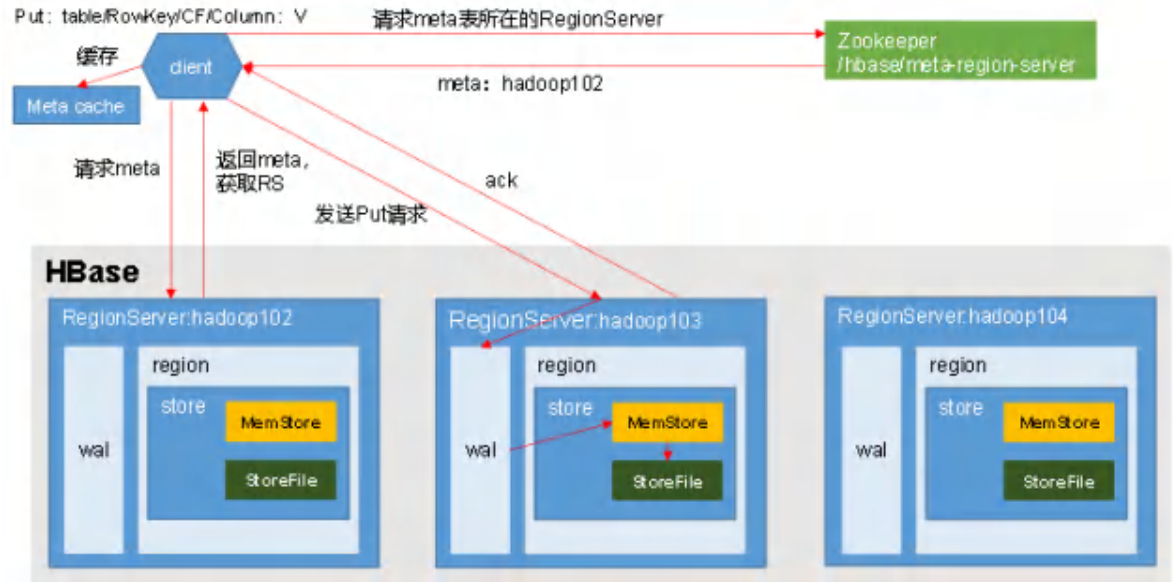
HLog(WAL log)：WAL意为write ahead log，用来做灾难恢复使用，HLog记录数据的所有变更，一旦region server 宕机，就可以从log中进行恢复。

HBase读写数据流程

问过的一些公司：阿里 x 2，多益 x 2，小米，电信云计算，陌陌，腾讯微众，vivo，海康威视，腾讯云 (2021.10)

参考答案：

1、写数据流程



1）Client先访问zookeeper，获取hbase:meta表位于哪个Region Server。

2）访问对应的Region Server，获取hbase:meta表，根据读请求的namespace:table/rowkey，查询出目标数据位于哪个Region Server中的哪个Region中。并将该table的region信息以及meta表的位置信息缓存在客户端的meta cache，方便下次访问。

3）与目标Region Server进行通讯；

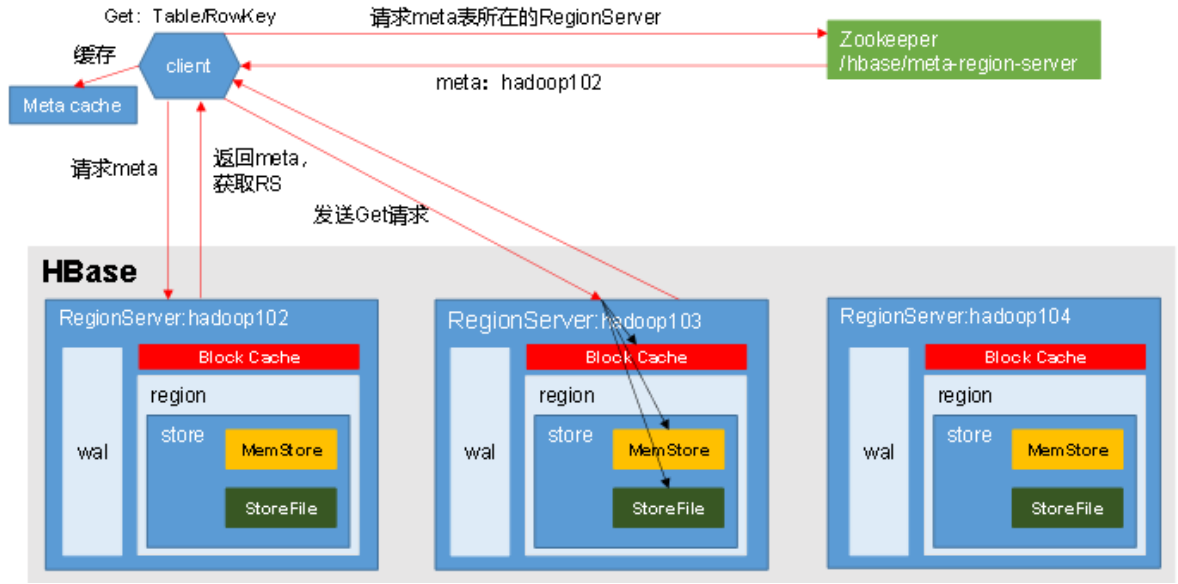
4）将数据顺序写入（追加）到WAL；

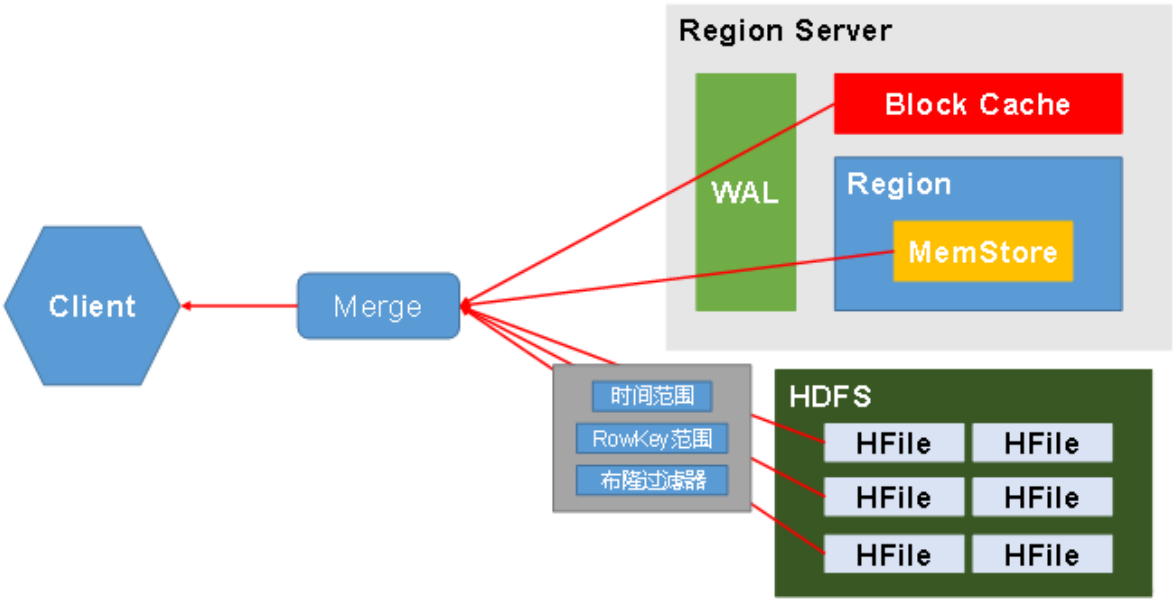
5）将数据写入对应的MemStore，数据会在MemStore进行排序；

6）向客户端发送ack；

7）等达到MemStore的刷写时机后，将数据刷写到HFile。

2、读数据流程





1）Client先访问zookeeper，获取hbase:meta表位于哪个Region Server。

2）访问对应的Region Server，获取hbase:meta表，根据读请求的namespace:table/rowkey，查询出目标数据位于哪个Region Server中的哪个Region中。并将该table的region信息以及meta表的位置信息缓存在客户端的meta cache，方便下次访问。

3）与目标Region Server进行通讯；

4）分别在Block Cache（读缓存），MemStore和Store File（HFile）中查询目标数据，并将查到的所有数据进行合并。此处所有数据是指同一条数据的不同版本（time stamp）或者不同的类型（Put/Delete）。

5）将查询到的数据块（Block，HFile数据存储单元，默认大小为64KB）缓存到Block Cache。

6）将合并后的最终结果返回给客户端。

HBase的读写缓存

问过的一些公司：字节

参考答案：

HBase上RegionServer的cache主要分为两个部分：**MemStore & BlockCache**。

MemStore是写缓存，BlockCache是读缓存。

当数据写入HBase时，会先写入memstore，RegionServer会给每个region提供一个memstore，memstore中的数据达到系统设置的阈值后，会触发flush将memstore中的数据刷写到磁盘。

客户的读请求会先到memstore中查数据，若查不到就到blockcache中查，再查不到就会从磁盘上读，并把读入的数据同时放入blockcahce。由于BlockCache采用的是LRU策略，因此BlockCache达到上限heapsize*hfile.block.cache .size * 0.85后，会启动淘汰机制，淘汰掉最老的一批数据。

BlockCache

为了高效获取数据，HBase设置了BlockCache机制，内存中缓存block，Block大体来分为两类，一类是JVM的heap内存，一类是heap off内存；**第一类的cache策略叫做LRUCache，第二类Cache策略有SlabCache以及BucketCache两类。**BlockCache是Region Server级别的，一个Region Server只有一个Block Cache，在Region Server启动的时候完成Block Cache的初始化工作。到目前为止，**HBase先后实现了3种Block Cache方案**，LRUBlockCache是最初的实现方案，也是默认的实现方案；HBase 0.92版本实现了第二种方案SlabCache；HBase 0.96之后官方提供了另一种可选方案BucketCache。

公众号：旧时光大数据

1、LRUBlockCache

LRUBlockCache是目前hbase默认的BlockCache机制，实现机制也比较简单，是使用一个ConcurrentHashMap管理BlockKey到Block的映射关系，缓存Block只需要将BlockKey和对应的Block放到该HashMap中，查询缓存就根据BlockKey从HashMap中获取即可。同时该方案采用严格的LRU淘汰算法，当BlockCache总量达到一定阈值之后就会启动淘汰机制，最近最少使用的Block会被置换出来。

LRUBlockCache将缓存分为三块：**single-access区、mutil-access区、in-memory区**，分别占到整个BlockCache大小的25%、50%、25%。Block Cache的实现机制核心思想是将Cache分级，这样的好处是避免Cache之间相互影响，尤其是对HBase来说像Meta表这样的Cache应该保证高优先级。

- single-access 优先级：当一个数据块第一次从HDFS读取时，它会具有这种优先级，并且在缓存空间需要被回收（置换）时，它属于优先被考虑范围内。它的优点在于：一般被扫描（scanned）读取的数据块，相较于之后会被用到的数据块，更应该被优先清除。
- mutil-access优先级：如果一个数据块，属于Single Access优先级，但是之后被再次访问，则它会升级为Multi Access优先级。在缓存里的内容需要被清除（置换）时，这部分内容属于次要被考虑的范围。
- in-memory-access优先级：表示数据可以常驻内存，一般用来存放访问频繁、数据量小的数据，比如元数据，用户也可以在建表的时候通过设置列族属性IN-MEMORY= true将此列族放入in-memory区。

加入Block Cache

- 这里假设不会对同一个已经被缓存的BlockCacheKey重复放入cache操作。
- 根据inMemory标志创建不同类别的CachedBlock对象：若inMemory为true则创建BlockPriority.MEMORY类型，否则创建BlockPriority.SINGLE；注意，这里只有这两种类型的Cache，因为BlockPriority.MULTI在Cache Block被重复访问时才进行创建。
- 将BlockCacheKey和创建的CachedBlock对象加入到全局的ConcurrentHashMap map中，同时做一些更新计数操作。
- 最后判断如果加入后的Block Size大于设定的临界值且当前没有淘汰线程运行，则调用runEviction()方法启动LRU淘汰过程。其中，EvictionThread线程即是LRU淘汰的具体实现线程。

淘汰Block Cache

EvictionThread线程主要用于与主线程的同步，从而完成Block Cache的LRU淘汰过程。EvictionThread线程启动后，调用wait被阻塞住，直到EvictionThread线程的evict方法被主线程调用时执行notify，开始执行LruBlockCache的evict方法进行真正的淘汰过程：

1. 首先获取锁，保证同一时刻只有一个淘汰线程运行；
2. 计算得到当前Block Cache总大小currentSize及需要被淘汰释放掉的大小bytesToFree，如果bytesToFree小于等于0则不进行后续操作；
3. 初始化创建三个BlockBucket队列，分别用于存放Single、Multi和InMemory类Block Cache，其中每个BlockBucket维护了一个CachedBlockQueue，按LRU淘汰算法维护该BlockBucket中的所有CachedBlock对象；
4. 遍历记录所有Block Cache的全局ConcurrentHashMap，加入到相应的BlockBucket队列中；
5. 将以上三个BlockBucket队列加入到一个优先级队列中，按照各个BlockBucket超出bucketSize的大小顺序排序（见BlockBucket的compareTo方法）；
6. 遍历优先级队列，对于每个BlockBucket，通过Math.min(overflow, (bytesToFree - bytesFreed) / remainingBuckets)计算出需要释放的空间大小，这样做可以保证尽可能平均地从三个BlockBucket中释放指定的空间；具体实现过程详见BlockBucket的free方法，从其CachedBlockQueue中取出即将被淘汰掉的CachedBlock对象；
7. 进一步调用了LruBlockCache的evictBlock方法，从全局ConcurrentHashMap中移除该CachedBlock对象，同时更新相关计数；
8. 释放锁，完成善后工作。

弊端：随着数据从single-access区晋升到multi-access区或者长时间停留在single-access区，对应的内存对象会从young区晋升到old区，晋升到old区的Block被淘汰后变为内存垃圾，最终由CMS回收。使用LRUBlockCache缓存机制会因为CMS GC策略导致内存碎片过多，从而可能引发Full GC，触发stop-the-world。

2、SlabCache

内部结构是划分为两块，80%和20%；缓存的数据如小于等于blocksize，则放在在前面的区域（80%区域）；如果block大于1x但是小于2x将会放置到后面区域（20%区域）；如果大于2x则不进行缓存。和LRUBlockCache相同，SlabCache也使用LRU算法对过期Block进行淘汰。和LRUBlockCache不同的是，SlabCache淘汰Block的时候只需要将对应的bufferbyte标记为空闲，后续cache对其上的内存直接进行覆盖即可。

线上集群环境中，不同表不同列族设置的BlockSize都可能不同，很显然，默认只能存储两种固定大小Block的SlabCache方案不能满足部分用户场景。因此HBase实际实现中将SlabCache和LRUBlockCache搭配使用，称为DoubleBlockCache。一次随机读中，一个Block块从HDFS中加载出来之后会在两个Cache中分别存储一份；缓存读时首先在LRUBlockCache中查找，如果Cache Miss再在SlabCache中查找，此时如果命中再将该Block放入LRUBlockCache中。

弊端：SlabCache设计中固定大小内存设置会导致实际内存使用率比较低，而且使用LRUBlockCache缓存Block依然会因为JVM GC产生大量内存碎片。因此在HBase 0.98版本之后，该方案已经被不建议使用。

3、BucketCache

BucketCache通过配置可以工作在三种模式下：heap，offheap和file。无论工作在那种模式下，BucketCache都会申请许多带有固定大小标签的Bucket，和SlabCache一样，一种Bucket存储一种指定BlockSize的数据块，但和SlabCache不同的是，BucketCache会在初始化的时候申请14个不同大小的Bucket，而且即使在某一种Bucket空间不足的情况下，系统也会从其他Bucket空间借用内存使用，不会出现内存使用率低的情况。heap模式表示这些Bucket是从JVM Heap中申请，offheap模式使用DirectByteBuffer技术实现堆外内存存储管理，而file模式使用类似SSD的高速缓存文件存储数据块。

弊端：HBase将BucketCache和LRUBlockCache搭配使用，称为CombinedBlockCache。和DoubleBlockCache不同，系统在LRUBlockCache中主要存储Index Block和Bloom Block，而将Data Block存储在BucketCache中。因此一次随机读需要首先在LRUBlockCache中查到对应的Index Block，然后再到BucketCache查找对应数据块。BucketCache通过更加合理的设计修正了SlabCache的弊端，极大降低了JVM GC对业务请求的实际影响，但也存在一些问题，比如使用堆外内存会存在拷贝内存的问题，一定程度上会影响读写性能。

在删除HBase中的一个数据的时候，它什么时候真正的进行删除呢？当你进行删除操作，它是立马就把数据删除掉了吗？

问过的一些公司：快手

参考答案：

1、Flush会进行删数据

Flush只会删除当前memStore中重复的数据（timestamp较小的删除）

StoreFile重复的并不会删

被标记为DeleteColumn的不会被删除

2、Major Compact也会进行删数据（当文件数>=3时，compact调用的也是major compact）

major compact 会将全部重复的数据进行删除，包括Storefile中的

公众号：旧时光大数据

major compact会将被标记为DeleteColumn的删除

3、在建表时如果指定存储2个版本，那么put进去相同rowkey的数据时，只会保留两个timestamp大的

为什么flush时被标记为删除的数据不会被删除?

如果flush时，被标记的数据删除了，StoreFile中有相同rowkey的数据。此时查看这个rowkey的数据时任然显示，这就不合常理了。

HBase中的二级索引

可回答：HBase如何利用phoiex实现二级索引的原理?

问过的一些公司：陌陌x2，蘑菇街

参考答案：

为什么需要HBse二级索引?

HBase里面只有rowkey作为一级索引， 如果要对库里的非rowkey字段进行数据检索和查询， 往往要通过MapReduce/Spark等分布式计算框架进行，硬件资源消耗和时间延迟都会比较高。

为了HBase的数据查询更高效、适应更多的场景， 诸如使用非rowkey字段检索也能做到秒级响应，或者支持各个字段进行模糊查询和多字段组合查询等， 因此需要在HBase上面构建二级索引， 以满足现实中更复杂多样的业务需求。

HBase二级索引方案

1、基于Coprocessor方案

1）官方特性

从0.94版本开始，HBase官方文档已经提出了hbase上面实现二级索引的一种路径：

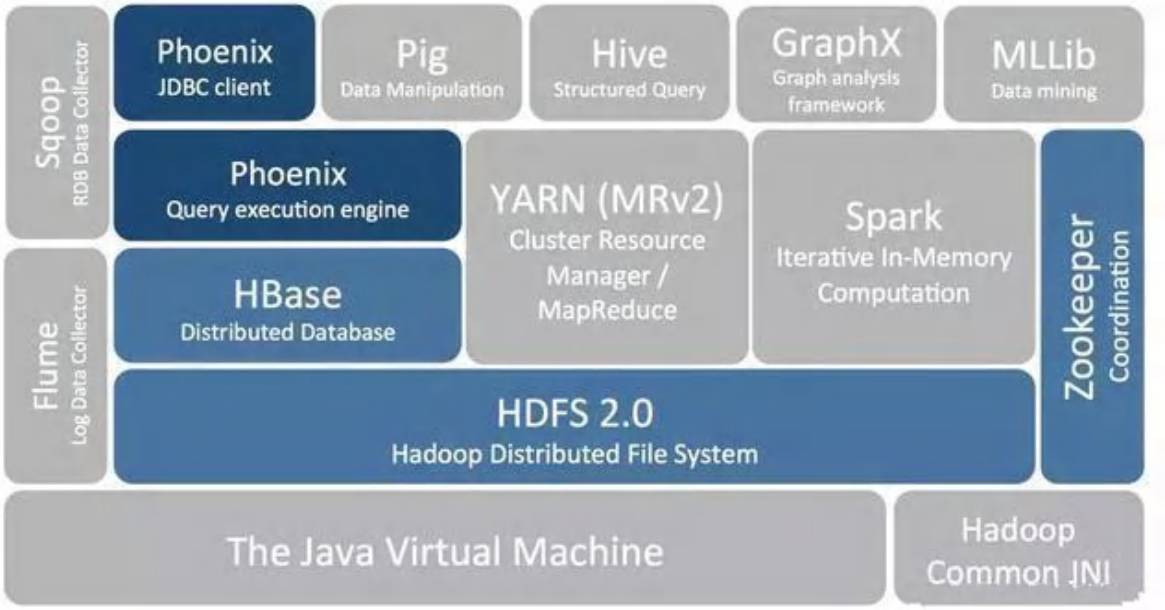
基于Coprocessor（0.92版本开始引入，达到支持类似传统RDBMS的触发器的行为）开发自定义数据处理逻辑，采用数据“双写”（dual-write）策略，在有数据写入同时同步到二级索引表。

2）开源方案

虽然官方一直也没提供内置的支持二级索引的工具， 不过业界也有些比较知名的基于Coprocessor的开源方案：

华为的hindex： 基于0.94版本，当年刚出来的时候比较火，但是版本较旧，看GitHub项目地址最近这几年就没更新过。Apache Phoenix： 功能围绕着SQL on hbase，支持和兼容多个hbase版本， 二级索引只是其中一块功能。 二级索引的创建和管理直接有SQL语法支持，使用起来很简便， 该项目目前社区活跃度和版本更新迭代情况都比较好。

ApachePhoenix在目前开源的方案中，是一个比较优的选择。主打SQL on HBase， 基于SQL能完成HBase的CRUD操作，支持JDBC协议。 Apache Phoenix在Hadoop生态里面位置：



3）Phoenix二级索引特点

Covered Indexes(覆盖索引)：把关注的数据字段也附在索引表上，只需要通过索引表就能返回所要查询的数据（列），所以索引的列必须包含所需查询的列(SELECT的列和WHRER的列)。

Functional indexes(函数索引)：索引不局限于列，支持任意的表达式来创建索引。

Global indexes(全局索引)：适用于读多写少场景。通过维护全局索引表，所有的更新和写操作都会引起索引的更新，写入性能受到影响。在读数据时，Phoenix SQL会基于索引字段，执行快速查询。

Local indexes(本地索引)：适用于写多读少场景。在数据写入时，索引数据和表数据都会存储在本地。在数据读取时，由于无法预先确定region的位置，所以在读取数据时需要检查每个region（以找到索引数据），会带来一定性能（网络）开销。

大多数基于Coprocessor实现二级索引的文章，都是遵循类似的思路：构建一份“索引”的映射关系，存储在另一张hbase表或者其他DB里面。

4）方案优缺点

优点：基于Coprocessor的方案，从开发设计的角度看，把很多对二级索引管理的细节都封装在的Coprocessor具体实现类里面，这些细节对外面读写的人是无感知的，简化了数据访问者的使用。

缺点：但是Coprocessor的方案入侵性比较强，增加了在Regionserver内部需要运行和维护二级索引关系表的代码逻辑等，对Regionserver的性能会有一定影响。

2、非Coprocessor方案

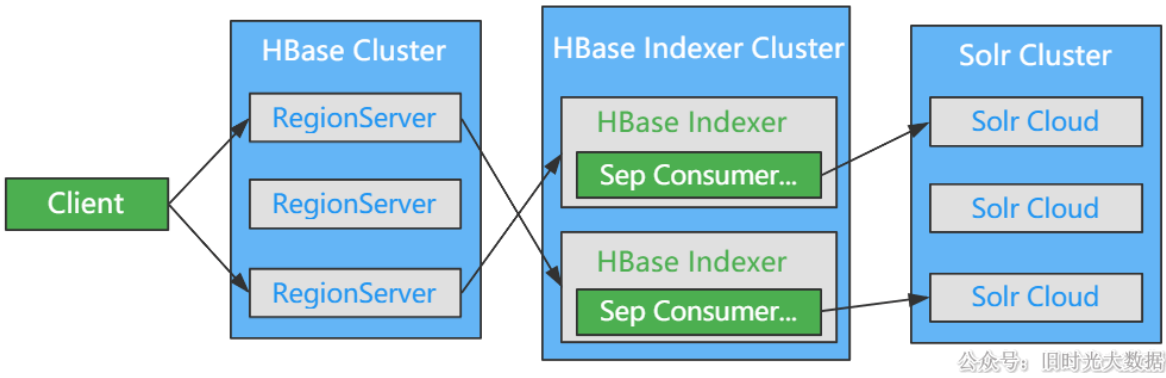
选择不基于Coprocessor开发，自行在外部构建和维护索引关系也是另外一种方式。

常见的是采用底层基于Apache Lucene的Elasticsearch(下面简称ES)或Apache Solr，来构建强大的索引能力、搜索能力，例如支持模糊查询、全文检索、组合查询、排序等。

1）Lily HBase Indexer

LilyHBase Indexer(也简称 HBase Indexer)是国外的NGDATA公司开源的基于solr的索引构建工具，特色是其基于HBase的备份机制，开发了一个叫SEP工具，通过监控HBase 的WAL日志（Put/Delete操作），来触发对solr集群索引的异步更新，基本对HBase无侵入性（但必须开启WAL），流程图如下所示：

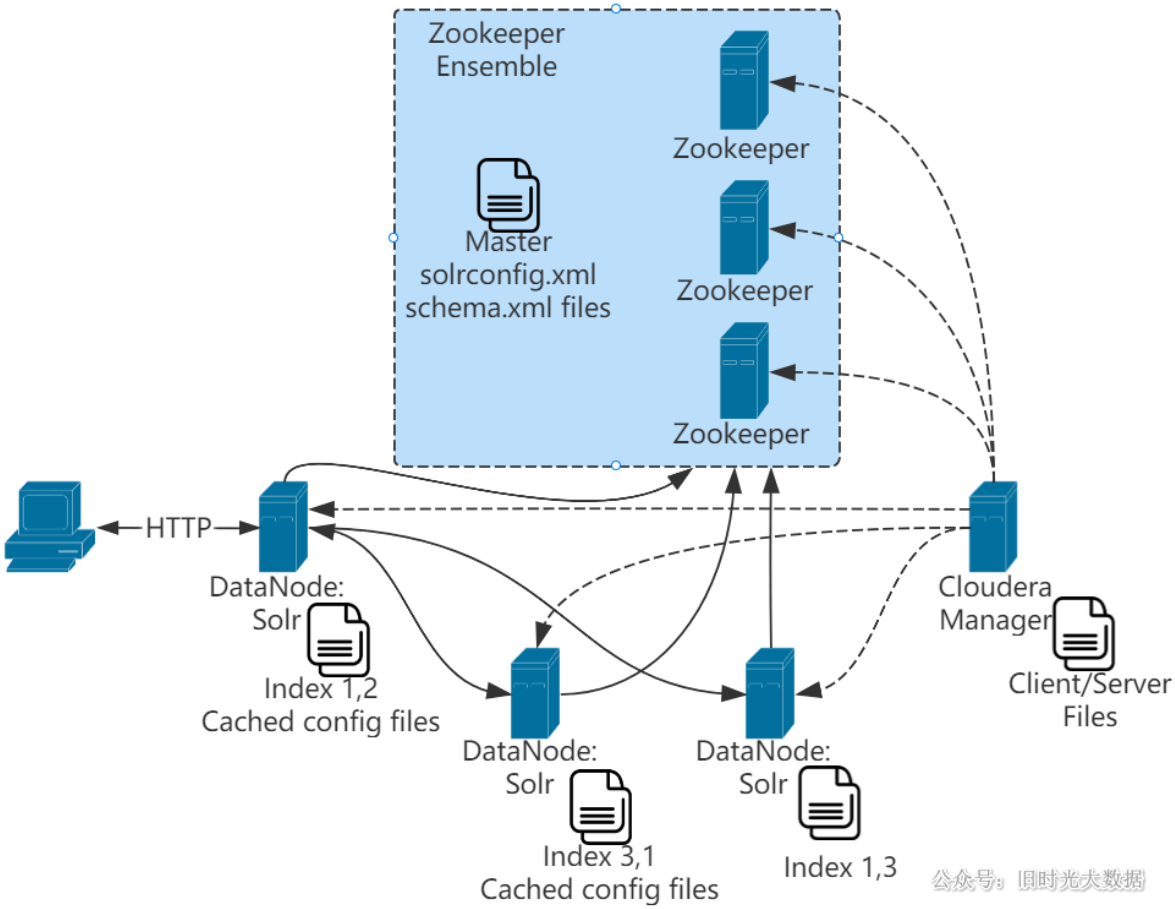
公众号：旧时光大数据



2）CDH Search

CDH Search是Hadoop发行商Cloudera公司开发的基于solr的HBase检索方案，部分集成了Lily HBase Indexer的功能。

下面是CDH search的核心组件交互图， 体现了在单次client端查询过程中，核心的zookeeper和solr等的交互流程：



3）CDH 支持构建索引方式

批量索引：

- 使用 Spark： CDH自带 spark 批量index工具
- 使用MapReduce： 集成Lily Indexer、自带MR index等工具

近实时索引（增量场景）： 使用 Flume 近实时（NRT）索引集成Lily NRT Indexer基于Solr REST API自定义索引场景

HBase的RegionServer宕机以后怎么恢复的?

公众号：旧时光大数据

问过的一些公司：海康

参考答案：

1、HBase常见故障

导致RegionServer故障的原因：

- FullGc引起长时间停顿
- HBase对Jvm堆内存管理不善，未合理使用堆外内存
- Jvm启动参数配置不合理
- 业务写入或吞吐量太大
- 写入读取字段太大
- HDFS异常
  - 读取写入数据都是直接操作hdfs的，若hdfs发生异常，会导致region server直接宕机
- 机器宕机
  - 物理节点直接宕机
  - 虚拟云主机不稳定，包括网络环境等

2、HBase常见故障

Master恢复：

- Master主要负责实现集群的负载均衡和读写调度，没有直接参与用户的请求，所以整体负载并不高
- 热备方式实现Master高可用，zookeeper上进行注册
- active master会接管整个系统的元数据管理任务，zk以及meta表中的元数据，相应用户的管理指令，创建、删除、修改，merge region等

RegionServer恢复：

- RegionServer宕机，HBase会检测到
- Master将宕机RegionServer上所有region重新分配到集群中其它正常的RegionServer上
- 根据HLog进行丢失数据恢复，恢复之后对外提供服务

大致流程：

1）master通过zk实现对RegionServer的宕机检测。RegionServer会周期性的向zk发送心跳，超过一定时间，zk会认为RegionServer离线，发送消息给master

2）切分为持久化的日志，所有region的数据都混合存储在同一个hlog文件里，为了使这些数据能够按照region进行组织回放，需要将hlog日志进行切分再合并，同一个region的数据合并在一起，方便后续按照region进行数据恢复。

3）master重新分配宕机regionserver上的所有region,regionserver宕机后，所有region处于不可用状态，所有路由到这些region上的请求都会返回异常。异常情况比较短暂，master会将这些region分配到其它regionserver上。

4）回放HLog日志补救数据

5）恢复完成，对外提供读写服务

HBase故障恢复流程总结如下：

- 故障检测
- 数据切分
- region上线
- 数据回放

HBase的一个region由哪些东西组成?

问过的一些公司：快手

参考答案：

Region是HBase集群分布数据的**最小单位。**

Region 是部分数据，所以是所有数据的一个自己，但Region包括完整的行，所以Region 是行为单位表的一个子集。

每个Region 有三个主要要素：

- 它所属于哪张表
- 它所包含的的第一行(第一个region 没有首行)
- 他所包含的最后一行(末一个region 没有末行)

当表初写数据时，此时表只有一个Region，当随着数据的增多，Region开始变大，等到它达到限定的阀值大小时，变化把Region分裂为两个大小基本相同的Region，而这个阀值就是StoreFile的设定大小（参数：hbase.hRegion.max.filesize），在第一次分裂Region之前，所有加载的数据都放在原始区域的那台服务器上，随着表的变大，Region的个数也会相应的增加，而Region是HBase集群分布数据的最小单位。

但Region也是由block组成，Region是属于单一的RegionServer，除非这个RegionServer 宕机，或者其它方式挂掉，再或者执行balance时，才可能会将这部分Region的信息转移到其它机器上。

这也就是为什么 Region比较少的时候，导致Region 分配不均，总是分派到少数的节点上，读写并发效果不显著，这就是HBase 读写效率比较低的原因。

HBase高可用怎么实现的?

问过的一些公司：腾讯微众，海康，快手，顺丰

参考答案：

没有高可用的HBase服务会出现哪些问题：

- 数据库管理人员失误，进行了不可逆的DDL操作
- 离线MR消耗过多的资源，造成线上服务受到影响
- 不可预计的另外一些情况：比如核心交换机故障，机房停电等等情况都会造成HBase服务中断

对于上述的那些问题，可以通过配置HBase的高可用来解决：

1、不可逆DDL问题

HBase的高可用不支持DDL操作，换句话说，在master上的DDL操作，不会影响到slave上的数据，所以即使在master上进行了DDL操作，slave上的数据依然没有变化。这个跟MySQL有很大不同，MySQL的DDL可以通过statement格式的Binlog进行复制。

2、离线MR影响线上业务问题

高可用的最大好处就是可以进行读写分离，离线MR可以直接跑在slave上，master继续对外提供写服务，这样也就不会影响到线上的业务，当然HBase的高可用复制是异步进行的，在slave上进行MR分析，数据可能会有稍微延迟。

3、意外情况

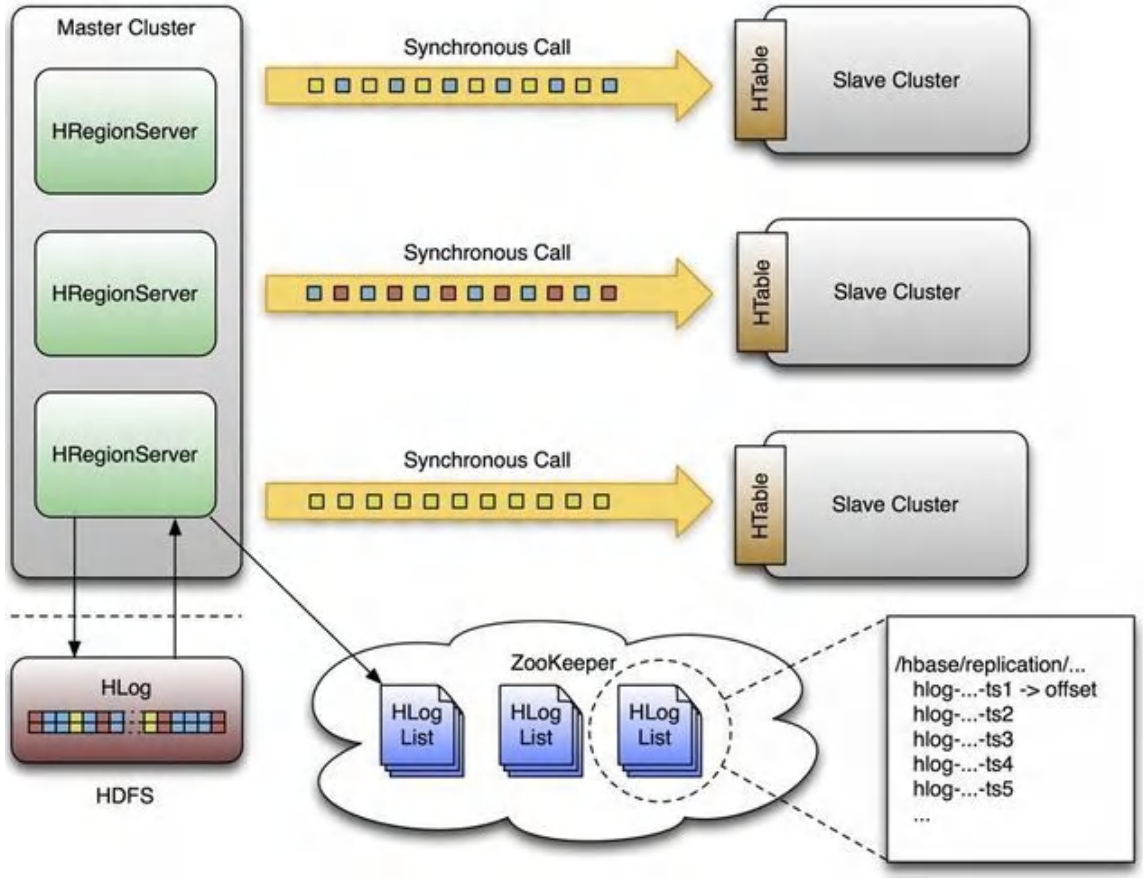
对于像核心交换机故障、断电等意外情况，slave跨机架或者跨机房部署都能解决该种情况。

公众号：旧时光大数据

基于以上原因，如果是核心服务，对于可用性要求非常高，可以搭建HBase的高可用来保障服务较高的可用性，在HBase的Master出现异常时，只需简单把流量切换到Slave上，即可完成故障转移，保证服务正常运行。

实现原理

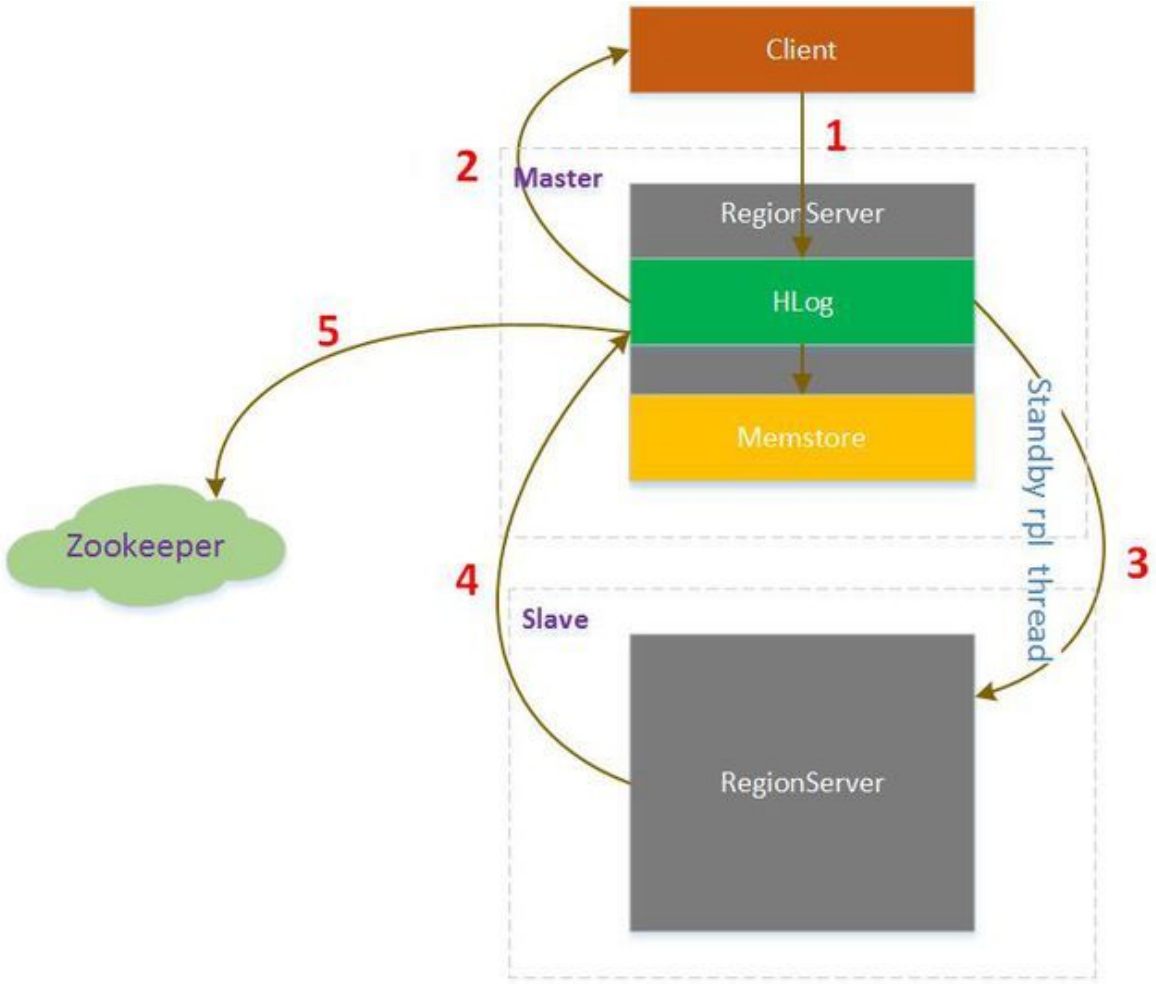
HBase高可用保证在出现异常时，快速进行故障转移。下面通过官方给出的一张图来看看HBase高可用的实现：



HBase Replication

需要声明的是，HBase的replication是以Column Family为单位的，每个Column Family都可以设置是否进行replication。

上图中，一个Master对应了3个Slave，Master上每个RegionServer都有一份HLog，在开启Replication的情况下，每个RegionServer都会开启一个线程用于读取该RegionServer上的HLog，并且发送到各个Slave，Zookeeper用于保存当前已经发送的HLog的位置。Master与Slave之间采用异步通信的方式，保障Master上的性能不会受到Slave的影响。用Zookeeper保存已经发送HLog的位置，主要考虑在Slave复制过程中如果出现问题后重新建立复制，可以找到上次复制的位置。



HBase Replication步骤：

1）HBase Client向Master写入数据

2）对应RegionServer写完HLog后返回Client请求

3）同时replication线程轮询HLog发现有新的数据，发送给Slave

4）Slave处理完数据后返回给Master

5）Master收到Slave的返回信息，在Zookeeper中标记已经发送到Slave的HLog位置

注意：

在进行replication时，Master与Slave的配置并不一定相同，比如Master上可以有3台RegionServer，Slave上并不一定是3台，Slave上的RegionServer数量可以不一样，数据如何分布这个HBase内部会处理。

为什么HBase适合写多读少业务?

问过的一些公司：快手

参考答案：

HBase更适合写多读少的场景。在写入这一块，每个table在合理预分区的情况，可以将负载均匀分配到不同的region server，而且可以LSM结构可以有效避免随机io，即使开启wal也是顺序写入，效率、吞吐量还是不错的。但这种结构也有一个问题，就是写放大，也就是常说的compation，这个问题要分析负载分布来调参，有的公司甚至还做了一些扩展以减小写放大的影响。

公众号：旧时光大数据

而读取默认hbase是没有二级索引的，唯一的索引就是rowkey，所以很多公司针对rowkey设计做文章，还有公司会扩展其他高效率查询的引擎作为二级索引，例如es，再或者通过phoneix。如果没有二级索引的支持，hbase本身对复杂一点的检索效率是很有问题的。

所以，光说hbase，写多读少更准确一些。

列式数据库的适用场景和优势？列式存储的特点？

问过的一些公司：美团

参考答案：

应用场景

列式存储适合数据分析类OLAP场景，比如进行客户流量预测，需要对数据进行反复的遍历，由于列式存储可以比行式存储有着更好的压缩比，所以存储量更小，读取速度更快。

架构特点

- 按列单独存储
- 一般使用LSM模型，数据先存在内存，进行排序，数据量大了在溢写到磁盘。

优势

- 列式存储由于每一列单独存放，所以数据即索引，可以使得在访问数据上只访问关心的字段部分，减少访问的吞吐量和系统的IO。
- 列式存储同样由于列式单独存放，所以可以支持并发处理。
- 列式存储一列的数据类型一致，数据特征相似，所以可以利用磁盘压缩算法进行高效压缩。

HBase的rowkey设计原则

可回答：1）HBase如何设计rowkey；2）你HBase的rowkey为什么这么设计？有什么优缺点？3）HBase的rowkey设计讲究

问过的一些公司：阿里 x 2，蘑菇街，陌陌 x 2，小米 x 4，腾讯 x 2，美团x3，美团(20210.08)-(2022.04)，冠群驰骋，顺丰，360，富途，vivo，携程，快手，重庆富民银行(2021.09)

参考答案：

HBase中，表会被划分为1...n个Region，被托管在RegionServer中。Region二个重要的属性：StartKey与EndKey表示这个Region维护的rowKey范围，当我们要读/写数据时，如果rowKey落在某个start-end key范围内，那么就会定位到目标region并且读/写到相关的数据。

那怎么快速精准的定位到我们想要操作的数据，就在于我们的rowkey的设计了。

设计原则如下：

1、rowkey长度原则

Rowkey是一个二进制码流，Rowkey的长度被很多开发者建议说设计在10~100个字节，不过建议是越短越好，不要超过16个字节。

原因如下：

1）数据的持久化文件HFile中是按照Key Value 存储的，如果Rowkey过长比如100个字节，1000万列数据光Rowkey就要占用100*1000 万=10亿个字节，将近1G数据，这会极大影响 HFile的存储效率；

2）MemStore将缓存部分数据到内存，如果 Rowkey字段过长内存的有效利用率会降低，系统将无法缓存更多的数据，这会降低检索效率。因此Rowkey的字节长度越短越好；

3）目前操作系统是都是64位系统，内存8字节对齐。控制在16个字节，8字节的整数倍利用操作系统的最佳特性。

2、rowkey散列原则

如果rowkey是按时间戳的方式递增，不要将时间放在二进制码的前面，建议将rowkey的高位作为散列字段，由程序循环生成，低位放时间字段，将会提高数据均衡分布在每个Regionserver实现负载均衡的几率。如果没有散列字段，首字段直接是时间信息将产生所有新数据都在一个RegionServer上堆积的热点现象，这样在做数据检索的时候负载将会集中在个别 RegionServer，降低查询效率。

3、rowkey唯一原则

必须在设计上保证其唯一性。rowkey是按照字典顺序排序存储的，因此，设计rowkey的时候，要充分利用这个排序的特点，将经常读取的数据存储到一块，将最近可能会被访问的数据放到一块。

HBase的rowkey为什么不能超过一定的长度？为什么要唯一？rowkey太长会影响Hfile的存储是吧？

问过的一些公司：快手

参考答案：

Hfile里面会有memstore，是占内存里的，太长的话放不了太多数据，减慢检索效率。

hbase是按字典序存储的，所以要利用这个排序的特点。

HBase的RowKey设置讲究有什么原因

问过的一些公司：字节

参考答案：

作用：

- 读写数据时，通过rowKey找到对应的Region
- MemStore中的数据按RowKey字典顺序排序
- HFile中的数据按RowKey字典顺序排序

设计原则：

- 唯一性，某一行数据的唯一标识符
- 长度，长度要占存储空间，越短越好
- 随机性，解决Hbase热点问题，避免只访问一个或几个节点

HBase的大合并、小合并是什么？

问过的一些公司：蘑菇街

参考答案：

删除一条记录，就会在该记录上打上标记，被打上标记的记录就成了墓碑记录，该记录使用get和scan查询不到，但还是在HFile中。只有进行大合并的时候才会删除HFile中的墓碑记录。

公众号：旧时光大数据

大合并：指定region的一个列族的所有HFile.合并完成后，这个列族的所有HFile文件合并成一个HFile文件，可以在shell中手动触发，但该动作相当耗资源。

小合并：将多个小的HFile文件内容读取出来合并生成一个大的HFile，把新文件设置成激活状态，然后删除小的HFile。

HBase和关系型数据库（传统数据库）的区别（优点）？

可回答：1）HBase为什么比MySQL快？2）HBase跟MySQL的区别

问过的一些公司：蚂蚁金服，阿里，美团，招银网络

参考答案：

1）数据类型：Hbase只有简单的数据类型，只保留字符串；传统数据库有丰富的数据类型。

2）数据操作：Hbase只有简单的插入、查询、删除、清空等操作，表和表之间是分离的，没有复杂的表和表之间的关系；传统数据库通常有各式各样的函数和连接操作。

3）存储模式：Hbase是基于列存储的，每个列族都由几个文件保存，不同列族的文件是分离的，这样的好处是数据即是索引，访问查询涉及的列大量降低系统的I/O，并且每一列由一个线索来处理，可以实现查询的并发处理；传统数据库是基于表格结构和行存储，其没有建立索引将耗费大量的I/O并且建立索引和物化试图需要耗费大量的时间和资源。

4）数据维护：Hbase的更新实际上是插入了新的数据；传统数据库只是替换和修改。

5）可伸缩性：Hbase可以轻松的增加或减少硬件的数目，并且对错误的兼容性比较高；传统数据库需要增加中间层才能实现这样的功能。

6）事务：Hbase只可以实现单行的事务性，意味着行与行之间、表与表之前不必满足事务性；传统数据库是可以实现跨行的事务性。

HBase数据结构

问过的一些公司：顺丰，科大讯飞，海康威视

参考答案：

1、Name Space

命名空间，类似于关系型数据库的database概念，每个命名空间下有多个表。HBase两个自带的命名空间，分别是hbase和default，hbase中存放的是HBase内置的表，default表是用户默认使用的命名空间。

一个表可以自由选择是否有命名空间，如果创建表的时候加上了命名空间后，这个表名字以`<Namespace>:<Table>`作为区分！

2、Table

类似于关系型数据库的表概念。不同的是，HBase定义表时只需要声明列族即可，数据属性，比如超时时间（TTL），压缩算法（COMPRESSION）等，都在列族的定义中定义，不需要声明具体的列。

这意味着，往HBase写入数据时，字段可以动态、按需指定。因此，和关系型数据库相比，HBase能够轻松应对字段变更的场景。

3、Row

公众号：旧时光大数据

HBase表中的每行数据都由一个RowKey和多个Column（列）组成。一个行包含了多个列，这些列通过列族来分类,行中的数据所属列族只能从该表所定义的列族中选取,不能定义这个表中不存在的列族，否则报错NoSuchColumnFamilyException。

4、RowKey

Rowkey由用户指定的一串不重复的字符串定义，是一行的唯一标识！数据是按照RowKey的字典顺序存储的，并且查询数据时只能根据RowKey进行检索，所以RowKey的设计十分重要。

如果使用了之前已经定义的RowKey，那么会将之前的数据更新掉！

5、Column Family

列族是多个列的集合。一个列族可以动态地灵活定义多个列。表的相关属性大部分都定义在列族上，同一个表里的不同列族可以有完全不同的属性配置，但是同一个列族内的所有列都会有相同的属性。

列族存在的意义是HBase会把相同列族的列尽量放在同一台机器上，所以说，如果想让某几个列被放到一起，你就给他们定义相同的列族。

官方建议一张表的列族定义的越少越好，列族太多会极大程度地降低数据库性能，且目前版本Hbase的架构，容易出BUG。

6、Column Qualifier

Hbase中的列是可以随意定义的，一个行中的列不限名字、不限数量，只限定列族。因此列必须依赖于列族存在！列的名称前必须带着其所属的列族！例如info：name，info：age。

因为HBase中的列全部都是灵活的，可以随便定义的，因此创建表的时候并不需要指定列！列只有在你插入第一条数据的时候才会生成。其他行有没有当前行相同的列是不确定，只有在扫描数据的时候才能得知！

7、TimeStamp

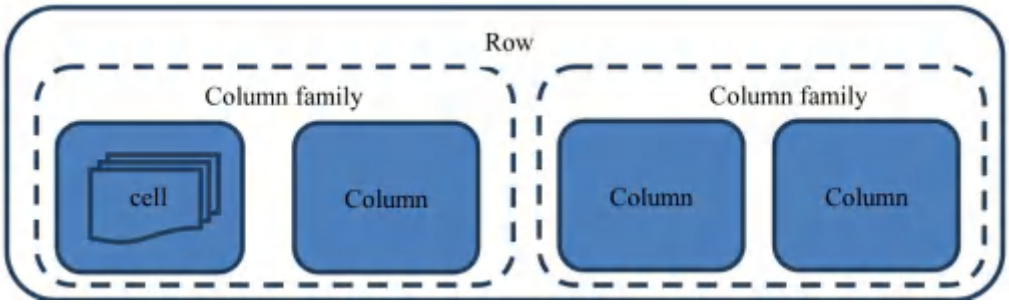
用于标识数据的不同版本（version）。时间戳默认由系统指定，也可以由用户显式指定。在读取单元格的数据时，版本号可以省略，如果不指定，Hbase默认会获取最后一个版本的数据返回！

8、Cell

一个列中可以存储多个版本的数据。而每个版本就称为一个单元格（Cell）。

Cell由{rowkey, column Family：column Qualifier, time Stamp}确定。

Cell中的数据是没有类型的，全部是字节码形式存贮。



9、Region

Region由一个表的若干行组成！在Region中行的排序按照行键（rowkey）字典排序。

Region不能跨RegionSever，且当数据量大的时候，HBase会拆分Region。

Region由RegionServer进程管理。HBase在进行负载均衡的时候，一个Region有可能会从当前RegionServer移动到其他RegionServer上。

Region是基于HDFS的，它的所有数据存取操作都是调用了HDFS的客户端接口来实现的。

公众号：旧时光大数据

10、RowKey

与 nosql 数据库们一样，RowKey是用来检索记录的主键。访问HBASE table中的行，只有三种方式：

- 通过单个RowKey访问
- 通过RowKey的range（正则）
- 全表扫描

RowKey行键（RowKey）可以是任意字符串（最大长度是64KB，实际应用中长度一般为10-100bytes），在HBase内部，RowKey保存为字节数组。存储时，数据按照RowKey的字典序（byte order）排序存储。设计RowKey时，要充分排序存储这个特性，将经常一起读取的行存储放到一起。(位置相关性)

HBase为什么随机查询很快?

可回答：HBase为什么查询速度快?

问过的一些公司：阿里，有赞，美团，字节(2021.10)

参考答案：

HBase能提供实时计算服务主要原因是由其架构和底层的数据结构决定的，即由LSM-Tree(Log-Structured Merge-Tree) + HTable(region分区) + Cache决定——客户端可以直接定位到要查数据所在的HRegion server服务器，然后直接在服务器的一个region上查找要匹配的数据，并且这些数据部分是经过cache缓存的。

前面说过HBase会将数据保存到内存中，在内存中的数据是有序的，如果内存空间满了，会刷写到HFile中，而在HFile中保存的内容也是有序的。当数据写入HFile后，内存中的数据会被丢弃。

HFile文件为磁盘顺序读取做了优化，按页存储。在内存中多个块存储并归并到磁盘，合并写入会产生新的结果块，最终多个块被合并为更大块。

多次刷写后会产生很多小文件，后台线程会合并小文件组成大文件，这样磁盘查找会限制在少数几个数据存储文件中。HBase的写入速度快是因为它其实并不是真的立即写入文件中，而是先写入内存，随后异步刷入HFile。所以在客户端看来，写入速度很快。另外，写入时候将随机写入转换成顺序写，数据写入速度也很稳定。

而读取速度快是因为它使用了LSM树型结构，而不是B或B+树。磁盘的顺序读取速度很快，但是相比而言，寻找磁道的速度就要慢很多。HBase的存储结构导致它需要磁盘寻道时间在可预测范围内，并且读取与所要查询的rowkey连续的任意数量的记录都不会引发额外的寻道开销。比如有5个存储文件，那么最多需要5次磁盘寻道就可以。而关系型数据库，即使有索引，也无法确定磁盘寻道次数。而且，HBase读取首先会在缓存（BlockCache）中查找，它采用了LRU（最近最少使用算法），如果缓存中没找到，会从内存中的MemStore中查找，只有这两个地方都找不到时，才会加载HFile中的内容，而上文也提到了读取HFile速度也会很快，因为节省了寻道开销。

案例：

1、快速查询

如果快速查询（从磁盘读数据），hbase是根据rowkey查询的，只要能快速的定位rowkey, 就能实现快速的查询，主要是以下因素：

- hbase是可划分成多个region，你可以简单的理解为关系型数据库的多个分区。
- 键是排好序了的
- 按列存储的

首先，能快速找到行所在的region(分区)，假设表有10亿条记录，占空间1TB，分列成了500个region, 1个region占2个G. 最多读取2G的记录，就能找到对应记录；

公众号：旧时光大数据

其次，是按列存储的，其实是列族，假设分为3个列族，每个列族就是666M， 如果要查询的东西在其中1个列族上，1个列族包含1个或者多个HStoreFile，假设一个HStoreFile是128M， 该列族包含5个HStoreFile在磁盘上. 剩下的在内存中。

再次，是排好序了的，你要的记录有可能在最前面，也有可能在最后面，假设在中间，我们只需遍历2.5个HStoreFile共300M。

最后，每个HStoreFile(HFile的封装)，是以键值对（key-value）方式存储，只要遍历一个个数据块中的key的位置，并判断符合条件可以了。 一般key是有限的长度，假设跟value是1:19（忽略HFile上其它块），最终只需要15M就可获取的对应的记录，按照磁盘的访问100M/S，只需0.15秒。 加上块缓存机制（LRU原则），会取得更高的效率。

2、实时查询

实时查询，可以认为是从内存中查询，一般响应时间在1秒内。HBase的机制是数据先写入到内存中，当数据量达到一定的量（如128M），再写入磁盘中， 在内存中，是不进行数据的更新或合并操作的，只增加数据，这使得用户的写操作只要进入内存中就可以立即返回，保证了HBase I/O的高性能。

实时查询，即反应根据当前时间的数据，可以认为这些数据始终是在内存的，保证了数据的实时响应。

HBase的LSM结构

问过的一些公司：网易云音乐

参考答案：

先看下三种基本的存储引擎，这样才能清楚**LSM树的由来**：

- 哈希存储引擎 是哈希表的持久化实现，支持增、删、改以及随机读取操作，**但不支持顺序扫描**，对应的存储系统为key-value存储系统。对于key-value的插入以及查询，哈希表的复杂度都是O(1)，明显比树的操作O(n)快,如果不需要有序的遍历数据，哈希表就是your Mr.Right
- B树存储引擎是B树的持久化实现，不仅支持单条记录的增、删、读、改操作，还支持顺序扫描（B+树的叶子节点之间的指针），对应的存储系统就是关系数据库（Mysql等）。
- LSM树（Log-Structured Merge Tree）存储引擎和B树存储引擎一样，同样支持增、删、读、改、顺序扫描操作。而且通过批量存储技术规避磁盘随机写入问题。**当然凡事有利有弊，LSM树和B+树相比，LSM树牺牲了部分读性能，用来大幅提高写性能。**

LSM树的设计思想非常朴素：**将对数据的修改增量保持在内存中，达到指定的大小限制后将这些修改操作批量写入磁盘**，不过读取的时候稍微麻烦，需要合并磁盘中历史数据和内存中最近修改操作，所以写入性能大大提升，读取时可能需要先看是否命中内存，否则需要访问较多的磁盘文件。极端的说，基于LSM树实现的HBase的写性能比Mysql高了一个数量级，读性能低了一个数量级。

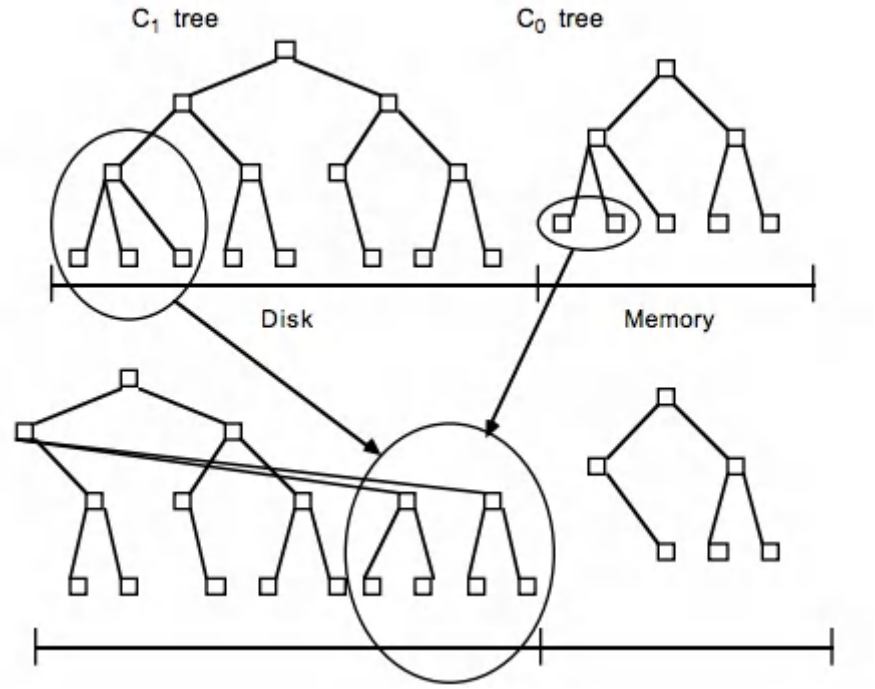
LSM树原理把一棵大树拆分成N棵小树，它首先写入内存中，随着小树越来越大，内存中的小树会flush到磁盘中，磁盘中的树定期可以做merge操作，合并成一棵大树，以优化读性能。

以上这些大概就是HBase存储的设计主要思想，这里分别对应说明下：

- 因为小树先写到内存中，为了防止内存数据丢失，写内存的同时需要暂时持久化到磁盘，对应了HBase的MemStore和HLog
- MemStore上的树达到一定大小之后，需要flush到HRegion磁盘中（一般是Hadoop DataNode），这样MemStore就变成了DataNode上的磁盘文件StoreFile，定期HRegionServer对DataNode的数据做merge操作，彻底删除无效空间，多棵小树在这个时机合并成大树，来增强读性能。

关于LSM Tree，对于最简单的二层LSM Tree而言，内存中的数据和磁盘你中的数据merge操作，如下图

公众号：旧时光大数据



LSM Tree，理论上，可以是内存中树的一部分和磁盘中第一层树做merge，对于磁盘中的树直接做update操作有可能会破坏物理block的连续性，但是实际应用中，一般lsm有多层，当磁盘中的小树合并成一个大树的时候，可以重新排好顺序，使得block连续，优化读性能。

hbase在实现中，是把整个内存在一定阈值后，flush到disk中，形成一个file，这个file的存储也就是一个小的B+树，因为hbase一般是部署在hdfs上，hdfs不支持对文件的update操作，所以hbase这么整体内存flush，而不是和磁盘中的小树merge update，这个设计也就能讲通了。内存flush到磁盘上的小树，定期也会合并成一个大树。整体上hbase就是用了lsm tree的思路。

HBase的Get和Scan的区别和联系?

问过的一些公司：小米

参考答案：

按指定RowKey 获取唯一一条记录， get方法（org.apache.hadoop.hbase.client.Get）

- Get的方法处理分两种：设置了ClosestRowBefore和没有设置的rowlock，主要是用来保证行的事务性，即每个get 是以一个row来标记的，一个row中可以有很多family和column。

按指定的条件获取一批记录，scan 方法(org.apache.Hadoop.hbase.client.Scan)实现条件查询功能使用的就是 scan 方式。

- scan可以通过setCaching与setBatch方法提高速度（以空间换时间）。
- scan可以通过setStartRow与setEndRow来限定范围([start， end]start 是闭区间， end 是开区间)。范围越小，性能越高。
- scan可以通过setFilter方法添加过滤器，这也是分页、多条件查询的基础。

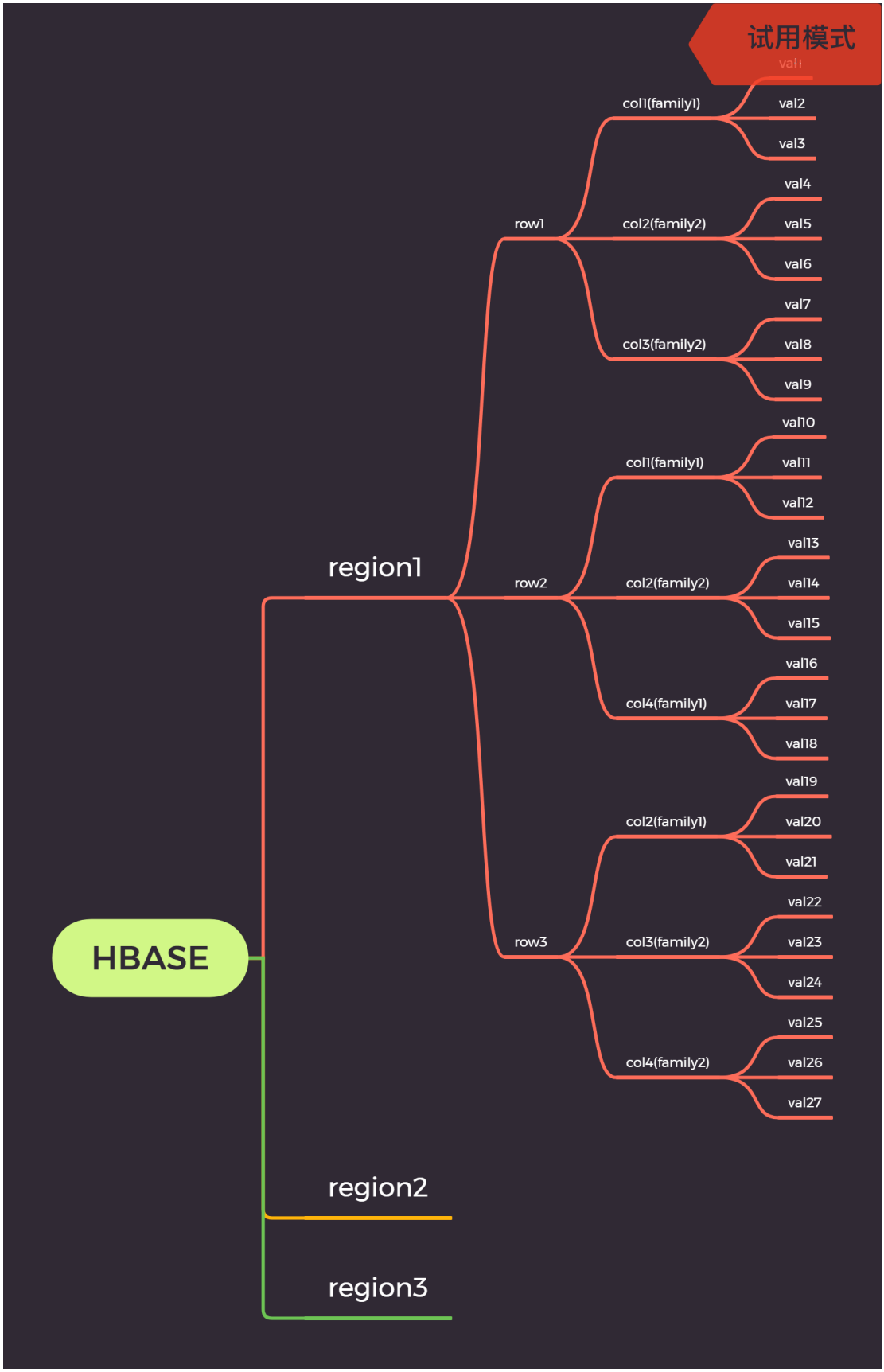
HBase数据的存储结构（底层存储结构）

可回答：HBase底层的存储结构

问过的一些公司：字节，有赞，远景x2，荣耀(2021.09)

参考答案：

公众号：旧时光大数据



hbase的数据存储结构如上所示，hbase的最小单位是列col，在列里存储着具体的值val，然后每一行row对应多个列col，并且每列col都有一个family标签，用于版本控制等区分，而根据行row的范围又可以分组为多个region，而region在hbase集群的对应的就是一个个的数据节点即region Server，当然hbase也会有一个master节点负责这些region节点的发现与管理。如果与集群中分片存储对应起来那么region就是一个个的分片了，通过这些分片在不同服务器上的冗余存储，可以保证数据的安全。

HBase数据compact流程?

公众号：旧时光大数据

问过的一些公司：阿里

参考答案：

HBase Compact是以牺牲磁盘io来换取读性能的基本稳定。

小合并：一个region下一个store中的部分HFile合并成一个比较大的StoreFile.

大合并：一个region下一个store中的所有HFile合并成一个更大的HFile.会清理ttl过期，版本超限定，标记删除的数据。

合并流程：

1）hbase基于下述某种触发条件触发Compaction。

- Memstore Flush（每次memstore在flush之后都会判断是否触发Compaction）
- 后台线程周期性检查
- 手动触发

2）hbase会单独有一个线程进行从该store中选择合适的HFile。

3）针对小合并，大合并，split等操作都会有对应的线程池进行处理

4）分别读出待合并Hfile文件的数据（K,V），进行归并排序，之后写到./tmp临时文件中。

5）将临时文件移动到对应的Store的数据目录

6）将Compaction的输入文件路径和输出路径封装成KV写入到HLog日志，并打上Compaction标记，最后强制执行sync。

7）将对应的Store数据目录下的Compaction输入文件全部删除。

优缺点

- 会尽量提高数据的本地化率，因为有些文件是在远程节点存储，通过Compaction会尽量本地化。降低数据读取响应延时，减少网络IO。
- 在Compaction过程中会对读请求造成较大的毛刺，为了使文件数趋于稳定，在操作过程中会有带宽压力和IO压力。
- 以牺牲短时间的性能资源来换取后续查询的稳定。
- 在Compaction过程中也会对写请求造成阻塞，比如Hfile较多时，达到默认配置可能会限制写请求的速度或者短时间的阻塞。

HBase的预分区

问过的一些公司：美团

参考答案：

预分区是指每一个region维护着startRow与endRowKey，如果加入的数据符合某个region维护的rowKey范围，则该数据交给这个region维护。那么依照这个原则，我们可以将数据所要投放的分区提前大致的规划好，以提高HBase性能。

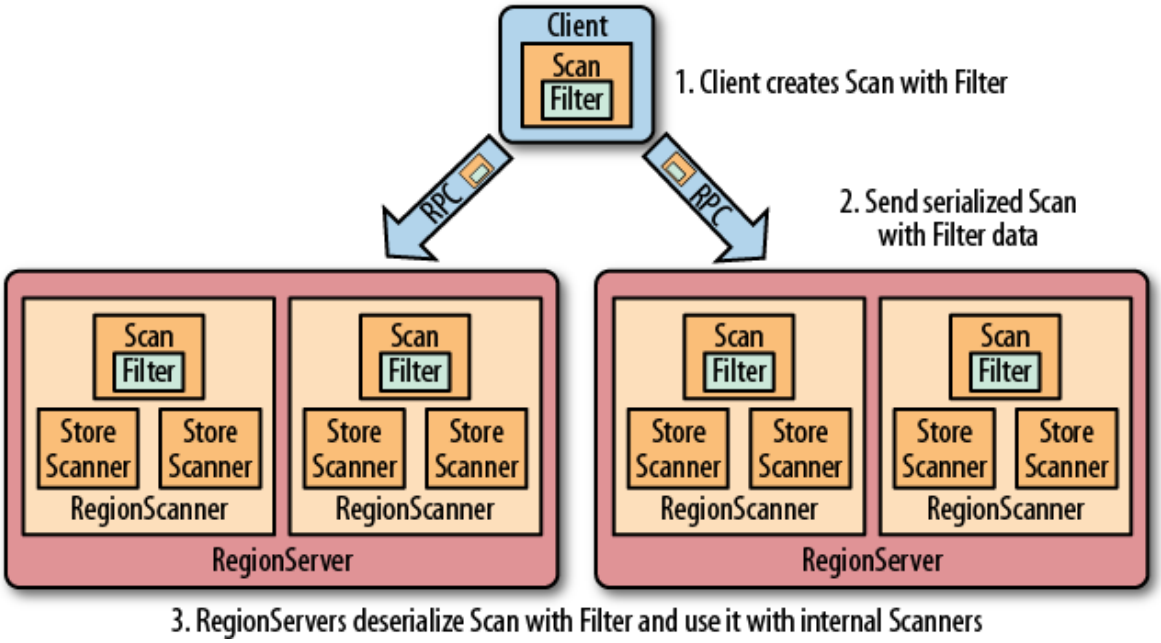
说几个HBase的过滤器及其作用?

问过的一些公司：东软集团

参考答案：

公众号：旧时光大数据

Hbase 提供了种类丰富的过滤器（filter）来提高数据处理的效率，用户可以通过内置或自定义的过滤器来对数据进行过滤，所有的过滤器都在服务端生效，即谓词下推（predicate push down）。这样可以保证过滤掉的数据不会被传送到客户端，从而减轻网络传输和客户端处理的压力。



1、过滤器基础

Filter 接口中定义了过滤器的基本方法，FilterBase 抽象类实现了 Filter 接口。所有内置的过滤器则直接或者间接继承自 FilterBase 抽象类。用户只需要将定义好的过滤器通过 `setFilter` 方法传递给 `Scan` 或 `put` 的实例即可。

```
setFilter(Filter filter)
```

```
// Scan 中定义的 setFilter
@Override
public Scan setFilter(Filter filter) {
    super.setFilter(filter);
    return this;
}
```

```
// Get 中定义的 setFilter
@Override
public Get setFilter(Filter filter) {
    super.setFilter(filter);
    return this;
}
```

HBase 内置过滤器可以分为三类：分别是**比较过滤器**，**专用过滤器**和**包装过滤器**。

2、比较过滤器

所有比较过滤器均继承自 `CompareFilter`。创建一个比较过滤器需要两个参数，分别是**比较运算符**和**比较器实例**。

```
public CompareFilter(final CompareOp compareOp,final ByteArrayComparable
comparator) {
    this.compareOp = compareOp;
    this.comparator = comparator;
}
```

2.1 比较运算符

公众号：旧时光大数据

- LESS (<)
- LESS_OR_EQUAL (<=)
- EQUAL (=)
- NOT_EQUAL (!=)
- GREATER_OR_EQUAL (>=)
- GREATER (>)
- NO_OP (排除所有符合条件的值)

比较运算符均定义在枚举类 `CompareOperator` 中

```
@InterfaceAudience.Public
public enum CompareOperator {
  LESS,
  LESS_OR_EQUAL,
  EQUAL,
  NOT_EQUAL,
  GREATER_OR_EQUAL,
  GREATER,
  NO_OP,
}
```

注意：在1.x版本的HBase中，比较运算符定义在 `CompareFilter.CompareOp` 枚举类中，但在2.0之后这个类就被标识为@deprecated ，并会在3.0移除。

2.2 比较器

所有比较器均继承自 `ByteArrayComparable` 抽象类，常用的有以下几种：

BinaryComparator : 使用 Bytes.compareTo(byte []，byte []) 按字典序比较指定的字节数组。

BinaryPrefixComparator : 按字典序与指定的字节数组进行比较，但只比较到这个字节数组的长度。

RegexStringComparator : 使用给定的正则表达式与指定的字节数组进行比较。仅支持 EQUAL 和 NOT_EQUAL 操作。

SubStringComparator : 测试给定的子字符串是否出现在指定的字节数组中，比较不区分大小写。仅支持 EQUAL 和 NOT_EQUAL 操作。

NullComparator： 判断给定的值是否为空。

BitComparator： 按位进行比较。

对于 `BinaryPrefixComparator` 和 `BinaryComparator` 的区别，这里举例说明一下：

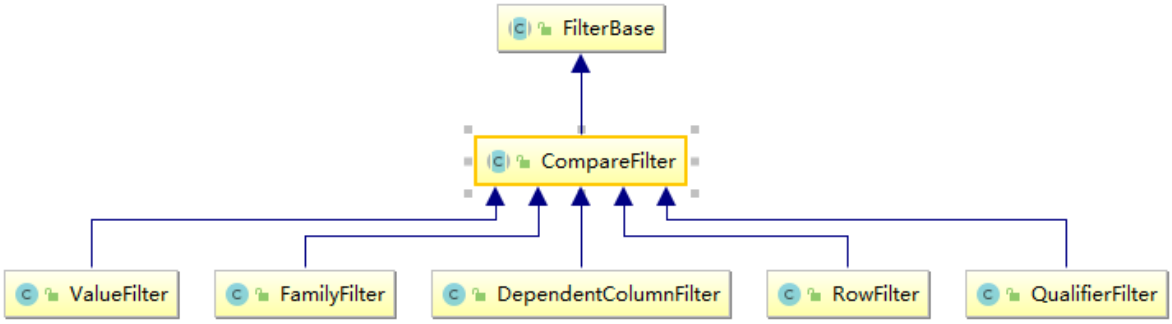
在进行 `EQUAL` 的比较时，如果比较器传入的是 `abcd` 的字节数组，但是待比较数据是 `abcdefgh`：

- 如果使用的是 BinaryPrefixComparator 比较器，则比较以 abcd 字节数组的长度为准，即 efgh 不会参与比较，这时候认为 abcd 与 abcdefgh 是满足 EQUAL 条件的；
- 如果使用的是 BinaryComparator 比较器，则认为其是不相等的。

2.3 比较过滤器种类

比较过滤器共有五个（Hbase 1.x 版本和 2.x 版本相同），见下图：

公众号：旧时光大数据



RowFilter：基于行键来过滤数据；

FamilyFilterr：基于列族来过滤数据；

QualifierFilterr：基于列限定符（列名）来过滤数据；

ValueFilterr：基于单元格 (cell) 的值来过滤数据；

DependentColumnFilter：指定一个参考列来过滤其他列的过滤器，过滤的原则是基于参考列的时间戳来进行筛选。

3、专用过滤器

专用过滤器通常直接继承自 `FilterBase`，适用于范围更小的筛选规则。

3.1 单列列值过滤器 (SingleColumnValueFilter)

基于某列（参考列）的值决定某行数据是否被过滤。其实例有以下方法：

setFilterIfMissing(boolean filterIfMissing)：默认值为 false，即如果该行数据不包含参考列，其依然被包含在最后的结果中；设置为 true 时，则不包含；

setLatestVersionOnly(boolean latestVersionOnly)：默认为 true，即只检索参考列的最新版本数据；设置为 false，则检索所有版本数据。

```
SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter(
                "student".getBytes(),
                "name".getBytes(),
                CompareOperator.EQUAL,
                new SubstringComparator("xiaolan"));
singleColumnValueFilter.setFilterIfMissing(true);
scan.setFilter(singleColumnValueFilter);
```

3.2 单列列值排除器 (SingleColumnValueExcludeFilter)

`SingleColumnValueExcludeFilter` 继承自上面的 `SingleColumnValueFilter`，过滤行为与其相反。

3.3 行键前缀过滤器 (PrefixFilter)

基于 RowKey 值决定某行数据是否被过滤。

```
PrefixFilter prefixFilter = new PrefixFilter(Bytes.toBytes("xxx"));
scan.setFilter(prefixFilter);
```

3.4 列名前缀过滤器 (ColumnPrefixFilter)

基于列限定符（列名）决定某行数据是否被过滤。

```
ColumnPrefixFilter columnPrefixFilter = new
ColumnPrefixFilter(Bytes.toBytes("xxx"));
scan.setFilter(columnPrefixFilter);
```

公众号：旧时光大数据

3.5 分页过滤器 (PageFilter)

可以使用这个过滤器实现对结果按行进行分页，创建 PageFilter 实例的时候需要传入每页的行数。

```
public PageFilter(final long pageSize) {
    Preconditions.checkArgument(pageSize >= 0, "must be positive %s", pageSize);
    this.pageSize = pageSize;
}
```

3.6 时间戳过滤器 (TimestampsFilter)

该过滤器允许针对返回给客户端的时间版本进行更细粒度的控制，使用的时候，可以提供一个返回的时间戳的列表，只有与时间戳匹配的单元才可以返回。当做多行扫描或者是单行检索时，如果需要一个时间区间，可以在Get或Scan对象上使用setTimeRange()方法来实现这一点。

```
List<Long> list = new ArrayList<>();
list.add(1554975573000L);
TimestampsFilter timestampsFilter = new TimestampsFilter(list);
scan.setFilter(timestampsFilter);
```

3.7 首次行键过滤器 (FirstKeyOnlyFilter)

`FirstKeyOnlyFilter` 只扫描每行的第一列，扫描完第一列后就结束对当前行的扫描，并跳转到下一行。相比于全表扫描，其性能更好，通常用于行数统计的场景，因为如果某一行存在，则行中必然至少有一列。

```
FirstKeyOnlyFilter firstKeyOnlyFilter = new FirstKeyOnlyFilter();
scan.set(firstKeyOnlyFilter);
```

4、包装过滤器

包装过滤器就是通过包装其他过滤器以实现某些拓展的功能。

4.1 SkipFilter过滤器

`SkipFilter` 包装一个过滤器，当被包装的过滤器遇到一个需要过滤的 KeyValue 实例时，则拓展过滤整行数据。下面是一个使用示例：

```
// 定义 ValueFilter 过滤器
Filter filter1 = new ValueFilter(CompareOperator.NOT_EQUAL,new
BinaryComparator(Bytes.toBytes("xxx")));
// 使用 SkipFilter 进行包装
Filter filter2 = new SkipFilter(filter1);
```

4.2 WhileMatchFilter过滤器

`WhileMatchFilter` 包装一个过滤器，当被包装的过滤器遇到一个需要过滤的 KeyValue 实例时，`WhileMatchFilter` 则结束本次扫描，返回已经扫描到的结果。下面是其使用示例：

```
Filter filter1 = new RowFilter(CompareOperator.NOT_EQUAL,
                               new BinaryComparator(Bytes.toBytes("rowKey4")));

Scan scan = new Scan();
scan.setFilter(filter1);
ResultScanner scanner1 = table.getScanner(scan);
for (Result result : scanner1) {
    for (Cell cell : result.listCells()) {
        System.out.println(cell);
    }
}
```

公众号：旧时光大数据

```
12    scanner1.close();
13
14    System.out.println("--------------------");
15
16    // 使用 WhileMatchFilter 进行包装
17    Filter filter2 = new WhileMatchFilter(filter1);
18
19    scan.setFilter(filter2);
20    ResultScanner scanner2 = table.getScanner(scan);
21    for (Result result : scanner1) {
22        for (Cell cell : result.listCells()) {
23            System.out.println(cell);
24        }
25    }
26    scanner2.close();
```

输出结果：

```
1   rowKey0/student:name/1555035006994/Put/vlen=8/seqid=0
2   rowKey1/student:name/1555035007019/Put/vlen=8/seqid=0
3   rowKey2/student:name/1555035007025/Put/vlen=8/seqid=0
4   rowKey3/student:name/1555035007037/Put/vlen=8/seqid=0
5   rowKey5/student:name/1555035007051/Put/vlen=8/seqid=0
6   rowKey6/student:name/1555035007057/Put/vlen=8/seqid=0
7   rowKey7/student:name/1555035007062/Put/vlen=8/seqid=0
8   rowKey8/student:name/1555035007068/Put/vlen=8/seqid=0
9   rowKey9/student:name/1555035007073/Put/vlen=8/seqid=0
10  --------------------
11  rowKey0/student:name/1555035006994/Put/vlen=8/seqid=0
12  rowKey1/student:name/1555035007019/Put/vlen=8/seqid=0
13  rowKey2/student:name/1555035007025/Put/vlen=8/seqid=0
14  rowKey3/student:name/1555035007037/Put/vlen=8/seqid=0
```

可以看到被包装后，只返回了 `rowKey4` 之前的数据。

5、FilterList

上述都是单个过滤器的作用，当需要多个过滤器共同作用于一次查询的时候，就需要使用 `FilterList`。`FilterList` 支持通过构造器或者 `addFilter` 方法传入多个过滤器。

```
1   // 构造器传入
2   public FilterList(final Operator operator, final List<Filter> filters)
3   public FilterList(final List<Filter> filters)
4   public FilterList(final Filter... filters)
5
6   // 方法传入
7   public void addFilter(List<Filter> filters)
8   public void addFilter(Filter filter)
```

多个过滤器组合的结果由 `operator` 参数定义，其可选参数定义在 `Operator` 枚举类中。只有 `MUST_PASS_ALL` 和 `MUST_PASS_ONE` 两个可选的值：

- **MUST_PASS_ALL**：相当于 AND，必须所有的过滤器都通过才认为通过；
- **MUST_PASS_ONE**：相当于 OR，只有要一个过滤器通过则认为通过。

公众号：旧时光大数据

```
@InterfaceAudience.Public
  public enum Operator {
    /** !AND */
    MUST_PASS_ALL,
    /** !OR */
    MUST_PASS_ONE
  }
```

HBase的热点问题

可回答：HBase避免数据倾斜的策略

问过的一些公司：阿里，多益，bigo，快手，美团(2021.08)

参考答案：

1、什么是热点

HBase中的行是按照rowkey的字典顺序排序的，这种设计优化了scan操作，可以将相关的行以及会被一起读取的行存取在临近位置，便于scan。然后糟糕的rowkey设计是热点的源头。

2、热点产生原因

- hbase的中的数据是按照字典序排序的，当大量连续的rowkey集中写在个别的region，各个region之间数据分布不均衡；
- 创建表时没有提前预分区，创建的表默认只有一个region，大量的数据写入当前region；
- 创建表已经提前预分区，但是设计的rowkey没有规律可循，设计的rowkey应该由regionNo+messageId组成。

3、热点解决方案

- HBase创建表时指定分区：HBase预分区
- 合理设计rowkey：参考rowkey设计原则
    - rowkey唯一原则
    - rowkey长度原则
    - rowkey散列原则
- 加盐
- 这里所说的加盐不是密码学中的加盐，而是**在rowkey的前面增加随机数**，具体就是给rowkey分配一个随机前缀以使得它和之前的rowkey的开头不同。给多少个前缀？这个数量应该和我们想要分散数据到不同的region的数量一致（类似hive里面的分桶）。加盐之后的rowkey就会根据随机生成的前缀分散到各个region上，以避免热点。
- 哈希
- 哈希会**使同一行永远用一个前缀加盐。**哈希**也可以使负载分散到整个集群**，但是读却是可以预测的。使用确定的哈希可以让客户端重构完整的rowkey，可以使用get操作准确获取某一个行数据。
- 反转
    - **反转固定长度或者数字格式的rowkey。**这样可以使得rowkey中经常改变的部分（最没有意义的部分）放在前面。这样可以有效的随机rowkey，但是牺牲了rowkey的有序性。

      反转rowkey的例子：以手机号为rowkey，可以将手机号反转后的字符串作为rowkey，从而避免诸如139、158之类的固定号码开头导致的热点问题。
- 时间戳反转

公众号：旧时光大数据

- 一个常见的数据处理问题是快速获取数据的最近版本，使用反转的时间戳作为rowkey的一部分对这个问题十分有用，可以用Long.Max_Value - timestamp追加到key的末尾，例如key，[key] 的最新值可以通过scan [key]获得[key]的第一条记录，因为HBase中rowkey是有序的，第一条记录是最后录入的数据。
- 尽量减少行和列的大小
- 在HBase中，value永远和它的key一起传输的。当具体的值在系统间传输时，它的rowkey，列名，时间戳也会一起传输。如果你的rowkey和列名很大，HBase storefiles中的索引（有助于随机访问）会占据HBase分配的大量内存，因为具体的值和它的key很大。可以增加block大小使得storefiles索引再更大的时间间隔增加，或者修改表的模式以减小rowkey和列名的大小。压缩也有助于更大的索引。
- 其他办法
    - 列族名的长度尽可能小，最好是只有一个字符。冗长的属性名虽然可读性好，但是更短的属性名存储在HBase中会更好。也可以在建表时预估数据规模，预留region数量，例如create 'myspace:mytable', SPLITS => [01,02,03,,...99]

4、HBase常见避免热点问题的方法

- 加盐：添加rowkey前缀，决定了在哪一个分区
- 哈希：哈希会使同一行永远用同一个前缀加盐。哈希也可以使负载分散到整个集群，但是读却是可以预测的。使用确定的哈希可以让客户端重构完成的rowkey，使用Get操作获取正常的某一行数据。
- 反转：反转固定长度或数字格式的rowkey，这样可以使得rowkey中经常改变的部分（最没有意义的部分）放在前面。这样就可以有效的随机rowkey，但是牺牲了rowkey 的有序性。

HBase的memstore冲刷条件

问过的一些公司：小米

参考答案：

主要有以下几种情况会触发 Memstore Flush：

- Region 中所有 MemStore 占用的内存超过相关阈值
- 整个 RegionServer 的 MemStore 占用内存总和大于相关阈值
- WAL数量大于相关阈值
- 定期自动刷写
- 数据更新超过一定阈值
- 手动触发刷写

HBase的MVCC

问过的一些公司：蘑菇街x2

参考答案：

1、什么是MVCC

MVCC(MultiVersionConsistencyControl， 多版本控制协议)，**是一种通过数据的多版本来解决读写一致性问题的解决方案。**在隔离性级别中，MVCC可以解决“可重复读”的隔离（即除了最后一级别的幻读无法解决，幻读只能事务串行化解决），基本是同一份数据并发条件下保证读写一致性的一个理想方案了。

一般情况下MVCC的一种实现思路是类似乐观锁(OCC，又叫乐观并发控制) 的实现机制。乐观锁适用于写冲突不大的并发场景，先执行写入，检查是否有冲突，若有冲突则回滚重来，否则提交写请求成功。MVCC获取最新的版本进行写操作，如果失败则回滚，成功则会将当前的版本作为可读点；读操作只能读大于或小于当前版本的数据。这里用版本概念可能会有点混淆，通常可能是timestamp或seqID。

对于单行数据，MVCC非常美好；但对于多行数据事务的更新操作就有问题了。MVCC是在最后检查才上锁，所以，如果Transaciton1执行理想的MVCC，修改Row1成功，而修改Row2失败，此时需要回滚Row1，但因为Row1没有被锁定，其数据可能又被Transaction2所修改，如果此时回滚Row1的内容，则会破坏Transaction2的修改结果，导致Transaction2违反ACID。因此，一般MVCC实际会配合二阶段锁(2PL)去实施。这样做虽然写事务被迫串行化了，但纯读取的事务不受锁影响且能保证最终的读写一致性。

2、HBase中的MVCC

HBase里虽然利用了版本这个概念做到了Cell层面的Version，HBase应用侧的version一般使用毫秒级时间戳作为版本，基于LSM的数据修改机制也是利用这个version来实现，包括官方一直未解决的同一毫秒内Delete和Put语义顺序问题。其实这个问题就体现了Version机制的好处，以及由于Version定为毫秒级时间戳不够唯一导致的Version机制崩溃的并发问题。

但HBase真正的MVCC实现则是在HRegion中的写操作的实现。HRegion采用的是一次封锁法。

封锁步骤：

1）对当前事务的所有行获取行锁（其中doMiniBatchMutation不会阻塞获取所有行锁，而是获取多少处理多少，然后在外部循环直至所有mutation操作完成；其他则会阻塞等待行锁）。

2）对region级别的updateLock 进行上读锁。updateLock是个读写锁，并发行级写操作的时候上读锁，region级别的写操作(flush,dropMemStore)的时候上写锁全部写操作阻塞。

MVCC的实现类是MultiVersionConsistencyControl，是个Region级别的MVCC控制。当有写操作来时，MVCC会做如下事情：

1）HRegion级别的seqID自增加一，并且当前 writeNo 设为 seqID + 1000000000。 这个大数的意义是防止别的写操作提交时把readNo提高了，导致当前writeNo成为一个可读状态的id，后面会将其设回正常的seqID （1.1.2 这里貌似有个坑）。

2）把当前的写操作的一个包含seqID的dummy对象 WriteEntry加进队列。

3）对于实际写操作本身，先写memstore，再写WAL，如果中间失败则回滚，否则则当做成功继续执行。

4）不管失败成功，当前这个seqID都是不可再用的了，然后MVCC内排队等待处理当前写请求提交。

5）写请求提交实际上就是把当前HRegion级别的readNo设为队列中已完成的写请求（包括别的线程的写请求）的seqID最大值，表示seqID以下的写请求都处理完了，可读。

HBase的大合并与小合并，大合并是如何做的？为什么要大合并

问过的一些公司：蘑菇街x2

参考答案：

HBase合并：删除一条记录，就会在该记录上打上标记，被打上标记的记录就成了墓碑记录，该记录使用get和scan查询不到，但还是在HFile中。**只有进行大合并的时候才会删除HFile中的墓碑记录。**

大合并：指定region的一个列族的所有HFile.合并完成后，这个列族的所有HFile文件合并成一个HFile文件，可以在shell中手动触发，但该动作相当耗资源。

小合并：将多个小的HFile文件内容读取出来合并生成一个大的HFile,把新文件设置成激活状态，然后删除小的HFile。

既然HBase底层数据是存储在HDFS上，为什么不直接使用HDFS，而还要用HBase

问过的一些公司：字节

参考答案：

HDFS面向批量访问模式，不是随机访问模式，Hadoop可以很好地解决大规模数据的离线批量处理问题，但是，受限于MapReduce编程框架的高延迟数据处理机制，使得Hadoop无法满足大规模数据实时处理应用的需求。

传统的通用关系型数据库无法应对在数据规模剧增时导致的系统扩展性和性能问题(分库分表也不能很好解决)。

传统关系数据库在数据结构变化时一般需要停机维护，空列浪费存储空间。

因此，业界出现了一类面向半结构化数据存储和处理的高可扩展、低写入/查询延迟的系统，例如键值数据库、文档数据库和列族数据库（如HBase等）。

HBase在HDFS之上提供了：

- 高并发实时随机写，通过LSM（内存+顺序写磁盘）的方式提供了HDFS所不拥有的实时随机写及修改功能；
- 高并发实时点读及扫描了解一下LSM算法，在文件系统之上有数据库，在业务层面，HBase完全可以独立于HDFS来理解。

因此，HBase可以满足大规模数据的实时处理需求。

HBase和Phoenix的区别

问过的一些公司：爱奇艺

参考答案：

在Phoenix中建的表，在HBase里是可以查到的，而在HBase里建的表，Phoenix中是查看不到的。

Phoenix支持Join操作，而HBase不支持Join操作，所以可以通过Phoenix来操作HBase。

HBase支持SQL操作吗

问过的一些公司：爱奇艺

参考答案：

不支持，可以通过Phoenix使用

HBase适合读多写少还是写多读少

问过的一些公司：爱奇艺

参考答案：

公众号：旧时光大数据

HBase更适合写多读少的场景。

在写入这一块，每个table在合理预分区的情况，可以将负载均匀分配到不同的region server，而且可以LSM结构可以有效避免随机io，即使开启wal也是顺序写入，效率、吞吐量还是不错的。但这种结构也有一个问题，就是写放大，也就是常说的compation，这个问题要分析负载分布来调参，有的公司甚至还做了一些扩展以减小写放大的影响。

而读取默认HBase是没有二级索引的，唯一的索引就是rowkey，所以很多公司针对rowkey设计做文章，还有公司会扩展其他高效率查询的引擎作为二级索引，例如es，再或者通过phoneix。如果没有二级索引的支持，HBase本身对复杂一点的检索效率是很有问题的。

如果单独是HBase，写多读少更准确一些。

HBase表设计

问过的一些公司：美团(2021.08)

参考答案：

1、预分区

默认情况下，在创建HBase表的时候会自动创建一个region分区，当导入数据的时候，所有的HBase客户端都向这一个region写数据，直到这个region足够大了才进行切分。一种可以加快批量写入速度的方法是通过**预先创建一些空的regions**，这样当数据写入HBase时，会按照region分区情况，在集群内做数据的负载均衡。

2、RowKey设计

参考前面的“HBase的rowkey设计原则”

3、列族数

不要在一张表里定义太多的column family。目前Hbase并不能很好的处理超过2~3个column family的表。因为某个column family在flush的时候，它邻近的column family也会因关联效应被触发flush，最终导致系统产生更多的I/O。

4、In Memory

创建表的时候，可以**通过HColumnDescriptor.setInMemory(true)将表放到RegionServer的缓存中，保证在读取表的时候被cache命中。**

5、Max Version

创建表的时候，可以通过HColumnDesriptor.setMaxVersions(int maxVersions)设置表中数据的最大版本数，如果只需要保存最新版本的数据，那么可以设置setMaxVersions(1)。

6、Time To Live

创建表的时候，可以通过HColumnDescriptor.setTimeToLive(int timeToLive)设置表中的数据的存储生命期，过期的数据自动被删除，例如如果只需要存储最近两天的数据，那么可以设置setTimeToLive(2 * 24 * 60 * 60)。

7、Compact & Split

在HBase中，数据在更新时首先写入WAL日志(HLog)和内存(MemStore)中，MemStore中的数据是排序的，当MemStore累计到一定阈值时，就会创建一个新的MemStore，并且将老的MemStore添加到flush队列，由单独的线程flush到磁盘上，成为一个StoreFile。于此同时， 系统会在zookeeper中记录一个redo point，表示这个时刻之前的变更已经持久化了(minor compact)。

公众号：旧时光大数据

StoreFile是只读的，一旦创建后就不可以再修改。因此Hbase的更新其实是不断追加的操作。当一个Store中的StoreFile达到一定的阈值后，就会进行一次合并(major compact)，将对同一个key的修改合并到一起，形成一个大的StoreFile，当StoreFile的大小达到一定阈值后，又会对 StoreFile进行分割(split)，等分为两个StoreFile。

由于对表的更新是不断追加的，处理读请求时，需要访问Store中全部的StoreFile和MemStore，将它们按照row key进行合并，由于StoreFile和MemStore都是经过排序的，并且StoreFile带有内存中索引，通常合并过程还是比较快的。

实际应用中，可以考虑必要时手动进行major compact，将同一个row key的修改进行合并形成一个大的StoreFile。同时，可以将StoreFile设置大些，减少split的发生。

Region分配

问过的一些公司：美团(2021.08)

参考答案：

1）一个region只能分配给一个region server

2）master节点记录有哪些可用的region server，region和region server映射信息，region的信息。

3）master发现没有分配的region时，就会去检查哪些region server可用，发给空闲region server一个load指令，把这个region 分配给这个region server，并且更新meta表信息。

4）master通过assignmentmanager进行分配工作。

5）如果发现负载不均衡，通过loadbalancerfactory进行重分配。

Region分配过程

1） **Root region 的分配**

HMaster 起动时，首先会通过 RegionManager 把 rootRegionLocation （里面包含 HServerAddress ）的值置为null ， 然后把 root region 从待处理 region 列表中移除（如果有的话），然后重新将其放入待处理 region 列表（regionsInTransition ） ，并将其状态设置为 UNASSIGNED(未分配).

当一个 Region server 启动完成时，它会调用 reportForDuty 函数向 HMaster 报告它的启动，报告当然是通过HMasterRegionInterface 的 regionServerStartup 方法。然后 HMaster 会把 Region server 的报告转交给ServerManager 的 regionServerStartup 方法处理。 ServerManager 会将这个新的 RegionServer 加入 region server 列表并且把它的 server load 设为空闲的。

RegionServer 会定期发送报告给 HMaster ， 请求 HMaster 进一步的指示。发送报告是通过 HMasterRegionInterface 的 regionServerReport 方法。 HMaster 接到报告后，移交报告给 ServerManager 的regionServerReport 方法处理。 ServerManager 会查询

regionServer 的状态，得到一个 HServerInfo 对像，然后检查 RegionServer 是否为正常的。如果为正常的话，ServerManager 会查询 regionserver 的负载（ HServerLoad ） ，更新一个 loadToServers 的 map 。然后进入ServerManager 的 processMsgs 函数处理。

ServerManager 会检查该 regionServer 的已经打开的 region 的数目，如果打开的 region 数目少于一个固定的值（对应配置文件中的 hbase.regions.nobalancing.count ） ，然后就会调用 RegionManager 的 assignRegions 方法。

RegionManager 会向 ServerManager 查询，现在已启动的 RegionServer 有几个，如果只有一个的话，会做特别处理。

然后 RegionManager 调用自己的 regionsAwaitingAssignment 方法去取得等待分配的 region 集合。它先会特别考虑 root region ， 如果它查到 root region 尚未被分配，它会马上返回只包含 root region 的集合。

如果没有任何 region 未分配的话而且并未处在安全模式， RegionManager 会让 loadBalancer 执行负载均衡的动作（就是可能把该 regionserver 负责的 region 分一点出去）。 反之如果有待分配的 region ，serverManager 会调用自己的 assignRegionsToMultipleServers方法。

在 assignRegionsToMultipleServers 中，参数 regionsToAssign 是所有待分配的 region 集合，因为存在多个regionServer, 所以 regionManager 会考虑到多个 regionServer 的负载。 regionManager 会先调用 regionsToGiveOtherServers 方法，求出其他 regionServer （比如相对负载较轻的）应该承载的 region 数目，那么当前 regionServer 可能承载的 region 个数就是待分配的 region 总数目减去其他 regionServer 应该承载的region 数目，这个数量我们暂称之为 N 。如果 N<=0 并且 Meta Region 如果已被分配的话，该 regionServer 会被略过，不会被要求承载 region 。因为比当前 regionServer 的负载轻的 regionServer 个数超过了待分配的region 个数，轮不到当前的 regionServer 。

regionManager 还会调用 computeNextHeaviestLoad 方法，算出 cluster 中有多少 regionServer 负载超过当前的 regionServer ， 这个数目可称之为 NS ， 同时该方法会抓出负载最重的 server 的负载。

接着 regionManager 会求出当前 regionServer 负责的 region 数目和负载最重的 regionServer 负责的 region 数目之间的差值。如果这个差值大于 N ， 那么 N 个 region 将全部会交由当前 regionServer 负责。反之这个差值小于N，如果 NS 大于零，当前 regionServer 要被分配的 region 数量为 (int)Math.ceil(1.0 * N/1.0 * NS), 如果 NS 等于零，那么当前 regionServer 要被分配 region 的数量为 (int)Math.ceil(1.0 * N/1.0 * regionServer 总数) 。

然后 root region 就会被分配到该 RegionServer 上。

注意：如果有多个 region serve 存在， HBase 不会把 root region 和 meta region 分配到一个RegionServer 上。

2） **Meta region 的分配**

一旦 root region 被分配完成， RootScanner 线程将被唤醒。然后它会 scan root region 。

在 scan 过程中，碰到每个 meta region 条目，它会调用 checkAssigned 函数检查，该 meta region 是否被分配，如果没有的话， regionManager 将会记录之，把该 region 加入待分配的 region 列表中。

一旦有 RegionServer 定期报告来了， meta region 会像 root region 一样的方式被分配。

3） **User region 的分配**

一旦 meta region 被分配完成， MetaScanner 将被唤醒，然后它会 scan meta region 。

在 scan 过程中，碰到每个 user region 条目，它会调用 checkAssigned 函数检查，该 user region 是否被分配，如果没有的话， regionManager 将会记录之，把该 region 加入待分配的 region 列表中。

一旦有 RegionServer 定期报告来了， user region 会被分配。

HBase的Region切分

问过的一些公司：腾讯(2021.10)

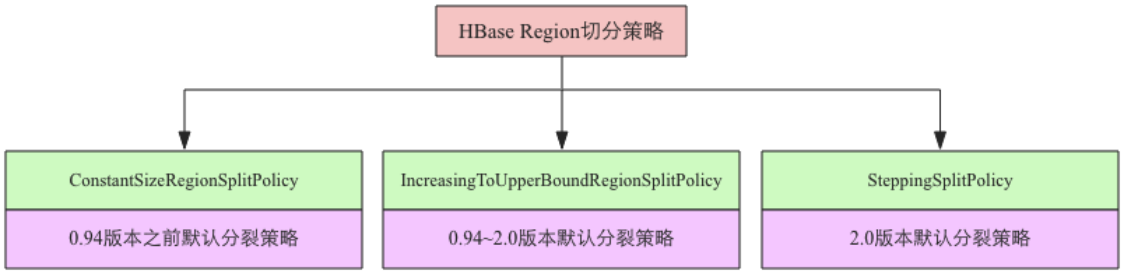
参考答案：

Region自动切分是HBase可以拥有良好扩张性的最重要因素之一，也必然是全部分布式系统追求无限扩展性的一副良药。

1、Region切分触发策略

HBase有多种切分触发策略，每种触发策略都有各自的适用场景，用户能够根据业务在表级别选择不一样的切分触发策略。常见的切分策略以下图：

公众号：旧时光大数据



ConstantSizeRegionSplitPolicy：0.94版本前默认切分策略。这是最容易理解但也最容易产生误解的切分策略，从字面意思来看，当region大小大于某个阈值（hbase.hregion.max.filesize）之后就会触发切分，实际上并不是这样，真正实现中这个阈值是对于某个store来说的，即一个region中最大store的大小大于设置阈值之后才会触发切分。另外一个大家比较关心的问题是这里所说的store大小是压缩后的文件总大小还是未压缩文件总大小，实际实现中store大小为压缩后的文件大小（采用压缩的场景）。ConstantSizeRegionSplitPolicy相对来来说最容易想到，但是在生产线上这种切分策略却有相当大的弊端：切分策略对于大表和小表没有明显的区分。阈值（hbase.hregion.max.filesize）设置较大对大表比较友好，但是小表就有可能不会触发分裂，极端情况下可能就1个，这对业务来说并不是什么好事。如果设置较小则对小表友好，但一个大表就会在整个集群产生大量的region，这对于集群的管理、资源使用、failover来说都不是一件好事。

IncreasingToUpperBoundRegionSplitPolicy：0.94版本~2.0版本默认切分策略。这种切分策略微微有些复杂，总体来看和ConstantSizeRegionSplitPolicy思路相同，一个region中最大store大小大于设置阈值就会触发切分。但是这个阈值并不像ConstantSizeRegionSplitPolicy是一个固定的值，而是会在一定条件下不断调整，调整规则和region所属表在当前regionserver上的region个数有关系：(#regions) * (#regions) * (#regions) * flush size * 2，当然阈值并不会无限增大，最大值为用户设置的MaxRegionFileSize。这种切分策略很好的弥补了ConstantSizeRegionSplitPolicy的短板，能够自适应大表和小表。而且在大集群条件下对于很多大表来说表现很优秀，但并不完美，这种策略下很多小表会在大集群中产生大量小region，分散在整个集群中。而且在发生region迁移时也可能会触发region分裂。

SteppingSplitPolicy：2.0版本默认切分策略。这种切分策略的切分阈值又发生了变化，相比IncreasingToUpperBoundRegionSplitPolicy简单了一些，依然和待分裂region所属表在当前regionserver上的region个数有关系，如果region个数等于1，切分阈值为flush size * 2，否则为MaxRegionFileSize。这种切分策略对于大集群中的大表、小表会比IncreasingToUpperBoundRegionSplitPolicy更加友好，小表不会再产生大量的小region，而是适可而止。

另外，还有一些其他分裂策略，比如使用DisableSplitPolicy：可以禁止region发生分裂；而KeyPrefixRegionSplitPolicy，DelimitedKeyPrefixRegionSplitPolicy对于切分策略依然依据默认切分策略，但对于切分点有自己的看法，比如KeyPrefixRegionSplitPolicy要求必须让相同的PrefixKey待在一个region中。

2、Region切分准备工作 - 寻找SplitPoint

region切分策略会触发region切分，切分开始之后的第一件事是寻找切分点 - splitpoint。所有默认切分策略，无论是ConstantSizeRegionSplitPolicy、IncreasingToUpperBoundRegionSplitPolicy抑或是SteppingSplitPolicy，对于切分点的定义都是一致的。当然，用户手动执行切分时是可以指定切分点进行切分的，这里并不讨论这种情况。

那切分点是如何定位的呢？整个region中最大store中的最大文件中最中心的一个block的首个rowkey。这是一句比较消耗脑力的语句，需要细细品味。另外，HBase还规定，如果定位到的rowkey是整个文件的首个rowkey或者最后一个rowkey的话，就认为没有切分点。

什么情况下会出现没有切分点的场景呢？最常见的就是一个文件只有一个block，执行split的时候就会发现无法切分。很多新同学在测试split的时候往往都是新建一张新表，然后往新表中插入几条数据并执行一下flush，再执行split，奇迹般地发现数据表并没有真正执行切分。原因可以通过debug查询。

```
2017-08-19 11:26:17,404 INFO  [PriorityRpcServer.handler=1,queue=1,port=60020] regionserver.RSRpcServices: Splitting spli
t,,1503112775843.6adfcc49ba04f307d6c6a604572d1bb2.
2017-08-19 11:26:17,409 DEBUG [PriorityRpcServer.handler=1,queue=1,port=60020] regionserver.StoreFile: cannot split becau
se midkey is the same as first or last row
2017-08-19 11:26:17,409 DEBUG [PriorityRpcServer.handler=1,queue=1,port=60020] regionserver.CompactSplitThread: Region sp
lit,,1503112775843.6adfcc49ba04f307d6c6a604572d1bb2. not splittable because midkey=null
```

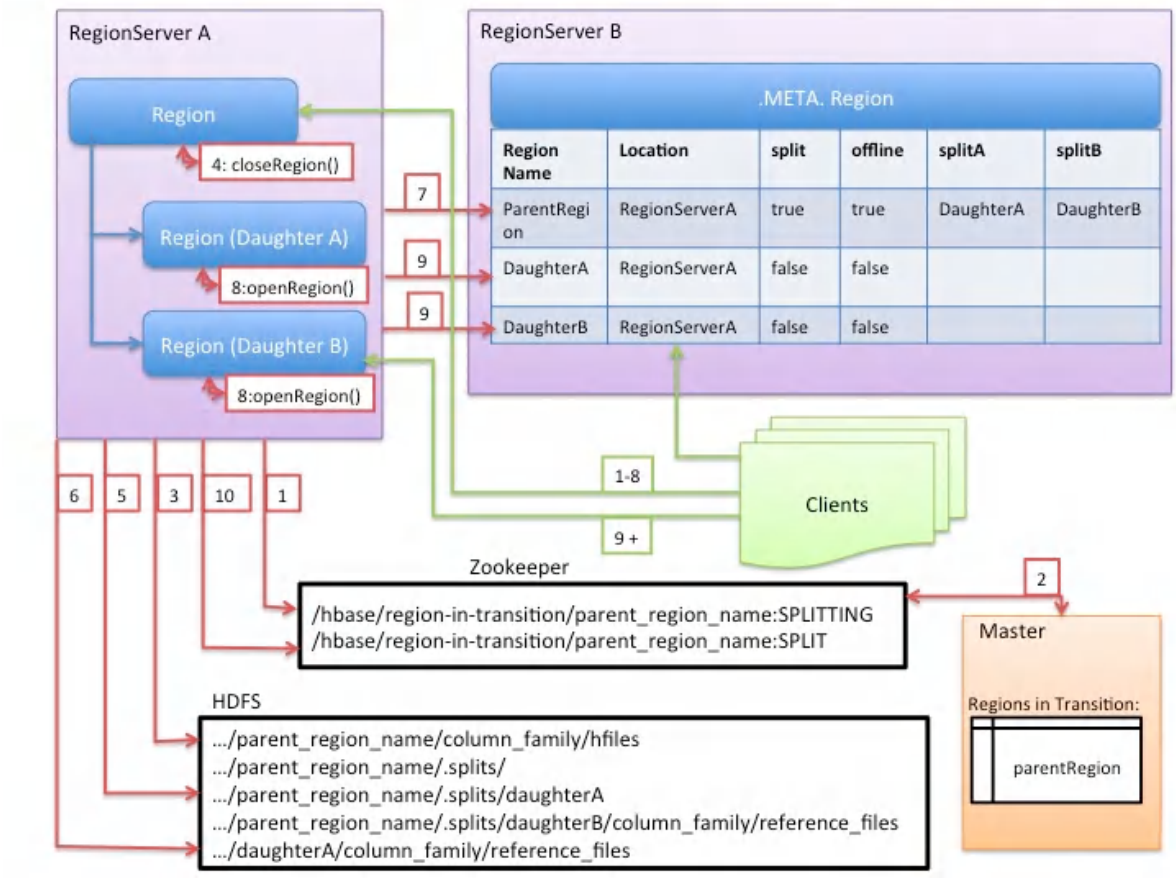
3、Region核心切分流程

HBase将整个切分过程包装成了一个事务，意图能够保证切分事务的原子性。整个分裂事务过程分为三个阶段：prepare – execute – (rollback)，操作模版如下：

```
if (!st.prepare()) return;
try {
  st.execute(this.server, this.server, user);
  success = true;
} catch (Exception e) {
  try{
    st.rollback(this.server, this.server);
  } catch (IOException re) {
    String msg = "Failed rollback of failed split of parent.getRegionNameAsString() -- aborting server";
    LOG.info(msg, re);
  }

}
```

prepare阶段：在内存中初始化两个子region，具体是生成两个HRegionInfo对象，包含tableName、regionName、startkey、endkey等。同时会生成一个transaction journal，这个对象用来记录切分的进展，具体见rollback阶段。

execute阶段：切分的核心操作。见下图



1）regionserver 更改ZK节点 /region-in-transition 中该region的状态为SPLITING。

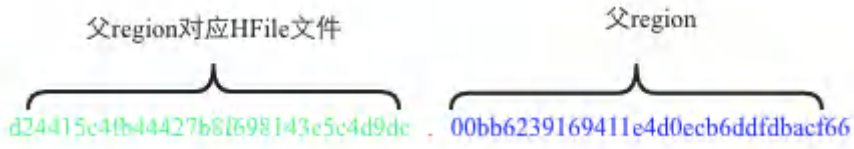
2）master通过watch节点/region-in-transition检测到region状态改变，并修改内存中region的状态，在master页面RIT模块就可以看到region执行split的状态信息。

3）在父存储目录下新建临时文件夹.split保存split后的daughter region信息。

4）关闭parent region：parent region关闭数据写入并触发flush操作，将写入region的数据全部持久化到磁盘。此后短时间内客户端落在父region上的请求都会抛出异常NotServingRegionException。

公众号：旧时光大数据

5）核心分裂步骤：在.split文件夹下新建两个子文件夹，称之为daughter A、daughter B，并在文件夹中生成reference文件，分别指向父region中对应文件。这个步骤是所有步骤中最核心的一个环节，生成reference文件日志如下所示：



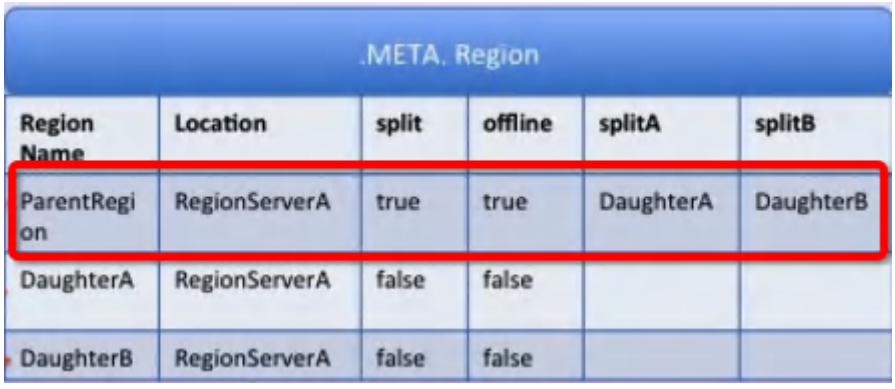
> DEBUG [StoreOpener-0155388346c3c919d3f05d7188e885e0-1] regionserver.StoreFileInfo: reference 'hdfs://hdfscluster/hbase-rsgroup/data/default/music/0155388346c3c919d3f05d7188e885e0/cf/d24415c4fb44427b8f698143e5c4d9dc.00bb6239169411e4d0ecb6ddfdbacf66' to region=00bb6239169411e4d0ecb6ddfdbacf66 hfile=d24415c4fb44427b8f698143e5c4d9dc

其中reference文件名为d24415c4fb44427b8f698143e5c4d9dc.00bb6239169411e4d0ecb6ddfdbacf66，格式看起来比较特殊，那这种文件名具体什么含义呢？那来看看该reference文件指向的父region文件，根据日志可以看到，切分的父region是00bb6239169411e4d0ecb6ddfdbacf66，对应的切分文件是d24415c4fb44427b8f698143e5c4d9dc，可见reference文件名是个信息量很大的命名方式。

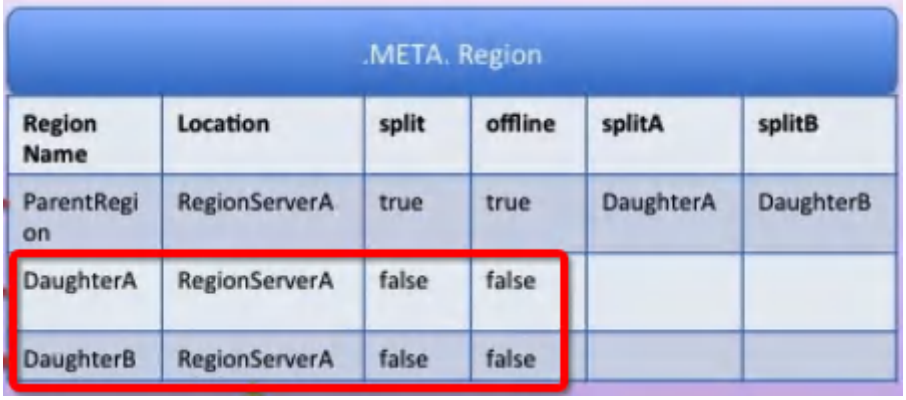
除此之外，还需要关注reference文件的文件内容，reference文件是一个引用文件（并非linux链接文件），文件内容很显然不是用户数据。文件内容其实非常简单，主要有两部分构成：其一是切分点splitkey，其二是一个boolean类型的变量（true或者false），true表示该reference文件引用的是父文件的上半部分（top），而false表示引用的是下半部分 （bottom）。

6）父region分裂为两个子region后，将daughter A、daughter B拷贝到HBase根目录下，形成两个新的region。

7）parent region通知修改 hbase.meta 表后下线，不再提供服务。下线后parent region在meta表中的信息并不会马上删除，而是标注split列、offline列为true，并记录两个子region。



8）开启daughter A、daughter B两个子region。通知修改 hbase.meta 表，正式对外提供服务。



公众号：旧时光大数据

rollback阶段：如果execute阶段出现异常，则执行rollback操作。为了实现回滚，整个切分过程被分为很多子阶段，回滚程序会根据当前进展到哪个子阶段清理对应的垃圾数据。代码中使用 JournalEntryType 来表征各个子阶段，具体见下图：

Journal Entey Type	拆分进度	回滚（清理垃圾数据）
STARTED	拆分逻辑被触发	NONE
PREPARED	执行拆分前的准备工作	NONE
SET_SPLITING_IN_ZK	Zookeeper中创建出拆分节点	删除Zookeeper中的拆分节点
CREATE_SPLIT_DIR	拆分目录 .split 被生成	清理父Region下的拆分目录
CLOSED_PARENT_REGION	父Region被关闭	重新对父Region执行初始化操作
OFFLINE_PARENT	父Region被下线，不再提供线上服务	重新将父Region上线
STARTED_REGION_A_CREATED	第一个子Region被成功创建	清理第一个子Region的存储目录
STARTED_REGION_A_CREATED	第二个子Region被成功创建	清理第二个子Region的存储目录
OPENED_REGION_A	第一个子Region被开启	关闭第一个子Region
OPENED_REGION_B	第二个子Region被开启	关闭第二个子Region

4、Region切分事务性保证

整个region切分是一个比较复杂的过程，涉及到父region中HFile文件的切分、两个子region的生成、系统meta元数据的更改等很多子步骤，因此必须保证整个切分过程的事务性，即要么切分完全成功，要么切分完全未开始，在任何情况下也不能出现切分只完成一半的情况。

为了实现事务性，hbase设计了使用状态机（见SplitTransaction类）的方式保存切分过程中的每个子步骤状态，这样一旦出现异常，系统可以根据当前所处的状态决定是否回滚，以及如何回滚。遗憾的是，目前实现中这些中间状态都只存储在内存中，因此一旦在切分过程中出现regionserver宕机的情况，有可能会出现切分处于中间状态的情况，也就是RIT状态。这种情况下需要使用hbck工具进行具体查看并分析解决方案。在2.0版本之后，HBase实现了新的分布式事务框架Procedure V2(HBASE-12439)，新框架将会使用HLog存储这种单机事务（DDL操作、Split操作、Move操作等）的中间状态，因此可以保证即使在事务执行过程中参与者发生了宕机，依然可以使用HLog作为协调者对事务进行回滚操作或者重试提交，大大减少甚至杜绝RIT现象。这也是是2.0在可用性方面最值得期待的一个亮点！！！

后面不看也没关系了

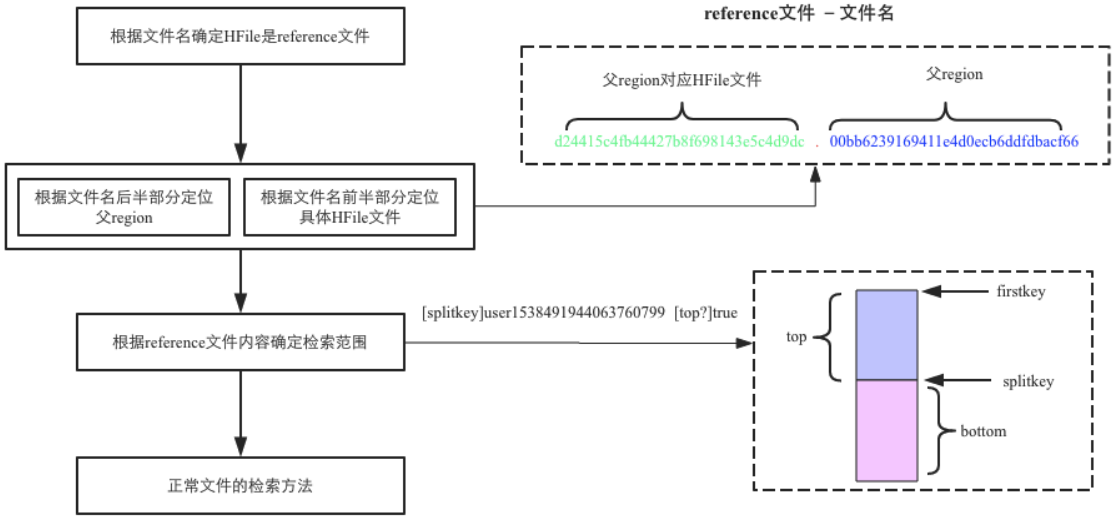
5、Region切分对其他模块的影响

通过region切分流程的了解，我们知道整个region切分过程并没有涉及数据的移动，所以切分成本本身并不是很高，可以很快完成。切分后子region的文件实际没有任何用户数据，文件中存储的仅是一些元数据信息-切分点rowkey等，那通过引用文件如何查找数据呢？子region的数据实际在什么时候完成真正迁移？数据迁移完成之后父region什么时候会被删掉？

1）通过reference文件如何查找数据？

这里就会看到reference文件名、文件内容的实际意义了。整个流程如下图所示：

公众号：旧时光大数据



（1）根据reference文件名（region名+真实文件名）定位到真实数据所在文件路径

（2）因为reference文件通常都只引用了数据文件的一半数据，以切分点为界，要么上半部分文件数据，要么下半部分数据。定位到真实数据文件还不能在整个文件中扫描待查KV，因为不能确定是哪部分数据，也不能确定切分点是哪个点，而这些信息就记录在reference文件中。

2）父region的数据什么时候会迁移到子region目录?

子region发生major_compaction时。compaction的执行实际上是将store中所有小文件一个KV一个KV从小到大读出来之后再顺序写入一个大文件，完成之后再将小文件删掉，因此compaction本身就需要读取并写入大量数据。子region执行major_compaction后会将父目录中属于该子region的所有数据读出来并写入子region目录数据文件中。可见将数据迁移放到compaction这个阶段来做，是一件顺便的事。

3）父region什么时候会被删除?

实际上HMaster会启动一个线程定期遍历检查所有处于splitting状态的父region，确定检查父region是否可以被清理。检测线程首先会在meta表中揪出所有split列为true的region，并加载出其分裂后生成的两个子region（meta表中splitA列和splitB列），只需要检查此两个子region是否还存在引用文件，如果都不存在引用文件就可以认为该父region对应的文件可以被删除。现在再来看看上文中父目录在meta表中的信息，就大概可以理解为什么会存储这些信息了。

4）split模块在生产线的一些坑?

有些时候会有同学反馈说集群中部分region处于长时间RIT，region状态为spliting。通常情况下都会建议使用hbck看下什么报错，然后再根据hbck提供的一些工具进行修复，hbck提供了部分命令对处于split状态的rit region进行修复，主要的命令如下：

```
-fixSplitParents  Try to force offline split parents to be online.
-removeParents    Try to offline and sideline lingering parents and keep daughter regions.
-fixReferenceFiles  Try to offline lingering reference store files
```

其中最常见的问题是：

```
ERROR: Found lingering reference file
hdfs://mycluster/hbase/news_user_actions/3b3ae24c65fc5094bc2acfebaa7a56de/meta/0f47cda55fa4
4cf9aa2599079894aed6.b7b3faab86527b88a92f2a248a54d3dc”
```

这个错误是说reference文件所引用的父region文件不存在了，如果查看日志的话有可能看到如下异常：

> java.io.IOException: java.io.IOException: java.io.FileNotFoundException: File does not exist:/hbase/news_user_actions/b7b3faab86527b88a92f2a248a54d3dc/meta/0f47cda55fa44cf9aa2599079894aed

至于父region文件为什么会莫名其妙不存在？有可能是因为官方bug导致，详见HBASE-13331。说HMaster在确认父目录是否可以被删除时，如果检查引用文件（检查是否存在、检查是否可以正常打开）抛出IOException异常，函数就会返回没有引用文件，导致父region被删掉。正常情况下应该保险起见返回存在引用文件，保留父region，并打印日志手工介入查看。也可以将修复patch打到线上版本或者升级版本。

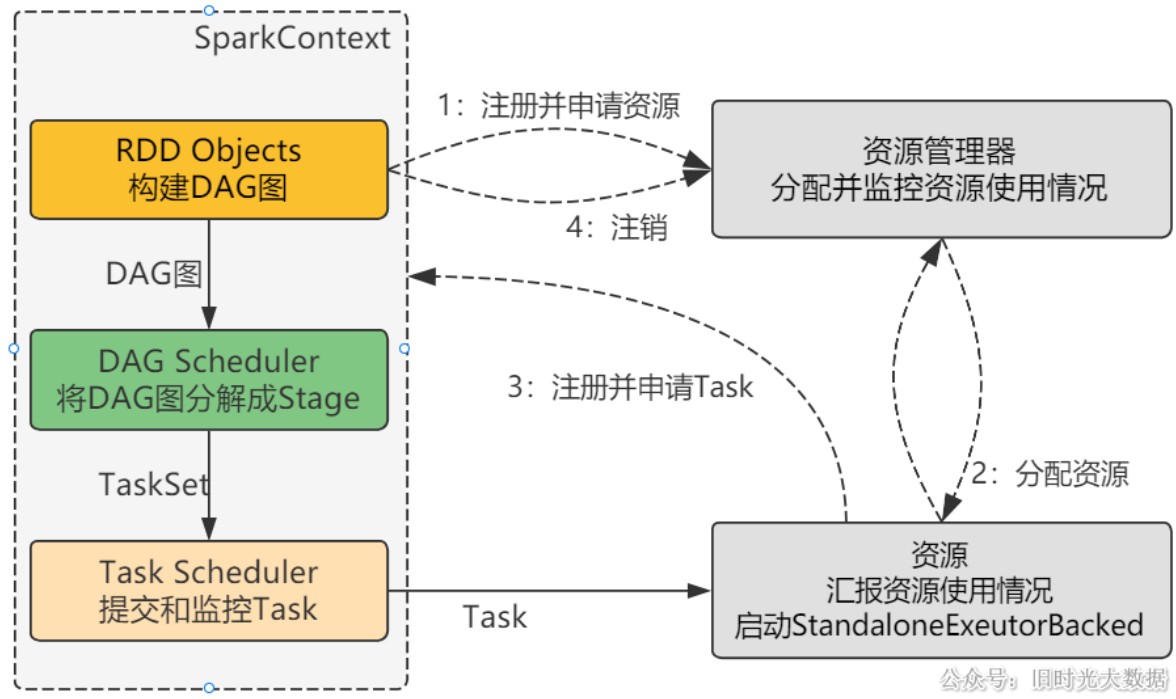
Spark面试题

Spark的任务执行流程

可回答：1）Spark的工作流程？2）Spark的调度流程；3）Spark的任务调度原理；4）Spark的任务提交和执行流程

问过的一些公司：今日头条x2，字节x4，字节(2021.07)，360，阿里x4，网易x2，阿里云，腾讯x2，小米x3，美团x4，美团(2021.08)，顺丰，快手，作业帮社招，一点咨询，ebay x 4，携程，妙盈科技x 2，流利说，爱奇艺x2，金山云，触宝，科大讯飞，海康威视，京东(2021.07)，陌陌(2021.07)

参考答案：



1）构建Application的运行环境，Driver创建一个SparkContext

```
val conf = new SparkConf()
conf.setAppName("test")
conf.setMaster("local")
val sc = new SparkContext(conf)
```

2）SparkContext向资源管理器（Standalone、Mesos、Yarn）申请Executor资源，资源管理器启动StandaloneExecutorbackend（Executor）

3）Executor向SparkContext申请Task

公众号：旧时光大数据

4）SparkContext将应用程序分发给Executor

5）SparkContext就建成DAG图，DAGScheduler将DAG图解析成Stage，每个Stage有多个task，形成taskset发送给task Scheduler，由task Scheduler将Task发送给Executor运行

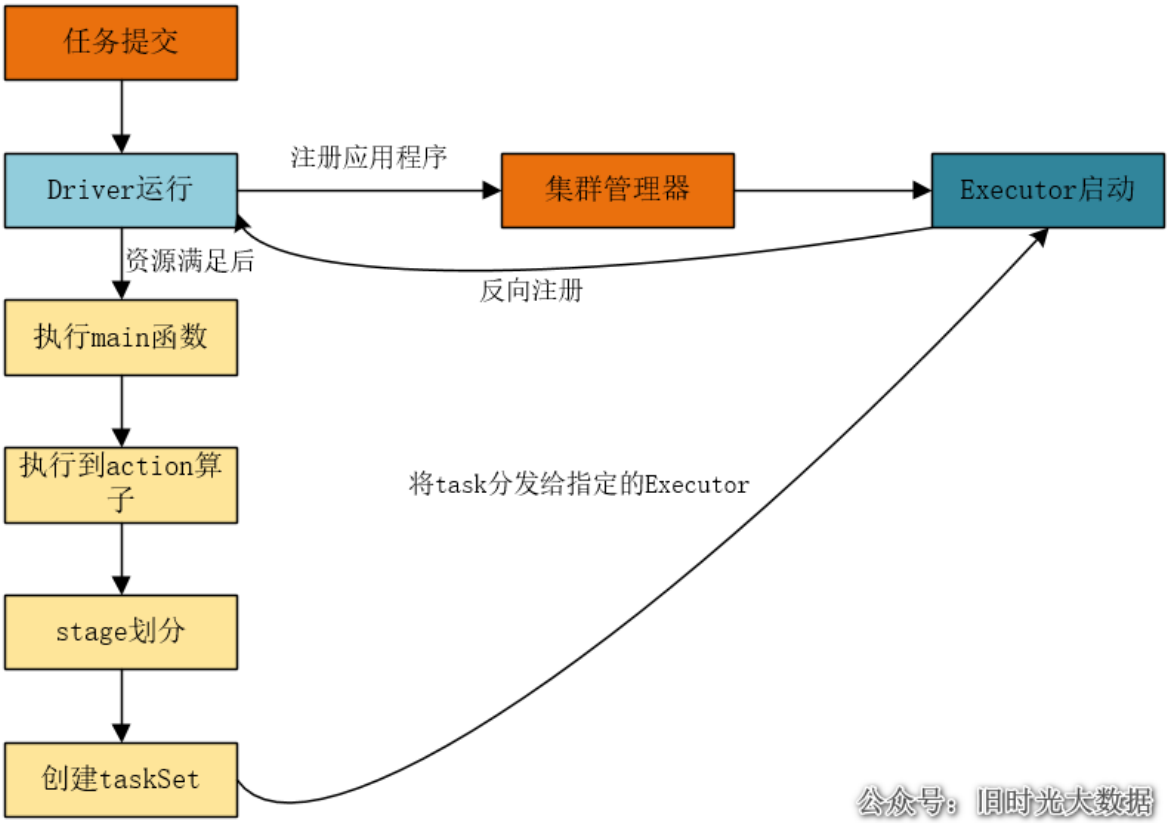
6）Task在Executor上运行，运行完释放所有资源

Spark的运行流程

可回答：1）Spark的执行机制；2）Spark提交一个任务的具体流程。

问过的一些公司：ebay，端点数据(2021.07)，字节(2021.08)x2，海康(2021.08)，百度(2021.08)，快手(2021.09)，唯品会(2021.10)

参考答案：



上图为Spark通用运行流程图，体现了基本的Spark应用程序在部署中的基本提交流程。

这个流程是按照如下的核心步骤进行工作的：

1）任务提交后，都会先启动Driver程序；

2）随后Driver向集群管理器注册应用程序；

3）之后集群管理器根据此任务的配置文件分配Executor并启动；

4）Driver开始执行main函数，Spark查询为懒执行，当执行到Action算子时开始反向推算，根据宽依赖进行Stage的划分，随后每一个Stage对应一个Taskset，Taskset中有多个Task，查找可用资源Executor进行调度；

5）根据本地化原则，Task会被分发到指定的Executor去执行，在任务执行的过程中，Executor也会不断与Driver进行通信，报告任务运行情况。

公众号：旧时光大数据

Spark的作业运行流程是怎么样的?

注意：详细版看这个，简洁版看上一题

问过的一些公司：

参考答案：

Spark运行模式：Standalone，YARN和Mesos。其中，Mesos和YARN模式类似。

目前用得比较多的是Standalone模式和YARN模式 。

1、 Standalone

Standalone模式是Spark实现的资源调度框架，其主要的节点有Client节点、Master节点和Worker节点。Driver既可以运行在Master节点上，也可以运行在本地Client端。当用spark-shell交互式工具提交spark的Job时，Driver在Master节点上运行。当使用spark-submit工具提交Job或者在Eclipse、IDEA等开发平台上使用“new SparkConf().setMaster(spark://master:7077)"方式运行Spark任务时，Driver是运行在本地Client端上的。

当用spark-shell交互式工具提交Spark的Job时，使用spark-shell启动脚本。该脚本启动一个交互式的Scala命令界面，可供用户来运行Spark相关命令。在Spark的安装目录下启动spark-shell，启动命令如下所示：

```
hadoop@master:/usr/local/spark$ ./bin/spark-shell
```

在spark-shell的启动过程中可以看到如下的信息，从中可以看到Spark的版本为2.1.0，spark内嵌的Scala版本为2.11.8，Java的版本为1.8.同时spark-shell在启动的过程中会初始化SparkContext为sc，以及初始化SQLContext为sqlContext。界面出现”scala>"这样的提示符，说明Spark交互式命令窗口启动成功，如图所示。用户可在该窗口下编写Spark相关代码。

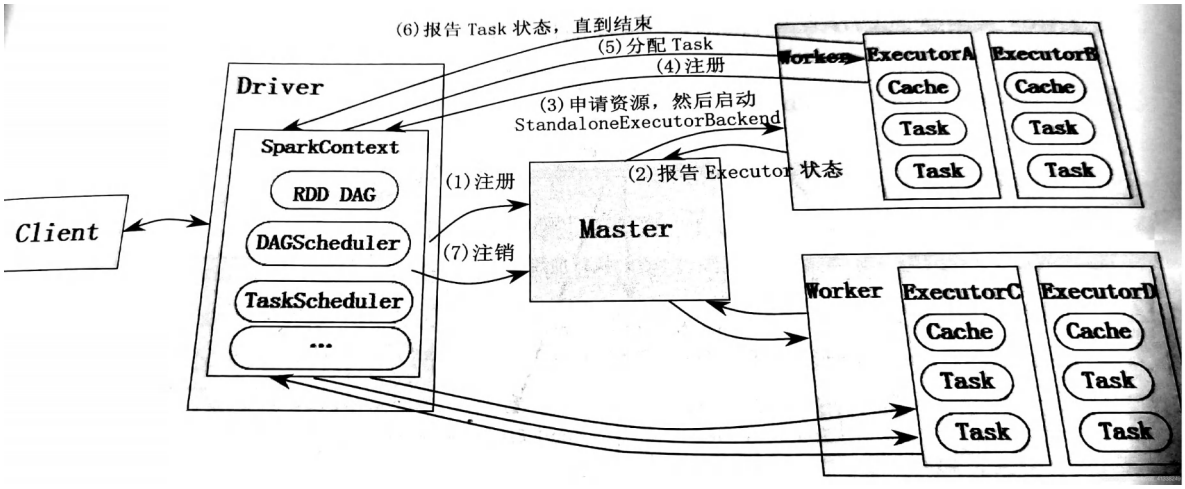
```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.1.0
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_181)
Type in expressions to have them evaluated.
Type :help for more information.
```

spark-shell启动的时候也可以手动指定每个节点的内存和Executor使用的cpu个数，启动命令如下所示：

```
hadoop@master:/usr/local/spark$ ./bin/spark-shell --executor-memory512m --total-executor-cores 3
```

当以Standalone模式向spark集群提交作业时，作业的运行流程如下图所示：

公众号：旧时光大数据



1）首先，SparkContext连接到Master，向Master注册并申请资源。

2）Worker定期发送心跳信息给Master并报告Executor状态。

3）Master根据SparkContext的资源申请要求和Worker心跳周期内报告的信息决定在哪个Worker上分配资源，然后在该Worker上获取资源，启动StandaloneExecutorBackend。

4）StandaloneExecutorBackend向SparkContext注册。

5）SparkContext将Application代码发送给StandaloneExecutorBackend，并且SparkContext解析Application代码，构建DAG图，并提交DAG Scheduler，分解成Stage(当碰到Action操作时)，就会催生Job，每个Job中含有一个或多个Stage），然后分配到相应的Worker，最后提交给StandaloneExecutorBackend执行。

6）StandaloneExecutorBackend会建立Executor线程池，开始执行Task，并向SparkContext报告，直至Task完成。

7）所有Task完成后，SparkContext向Master注销，释放资源。

2、YARN模式

YARN模式根据Driver在集群中的位置分为两种，一种是**YARN-Client模式（客户端模式）**，另一种是**YARN-Cluster模式（集群模式）**。

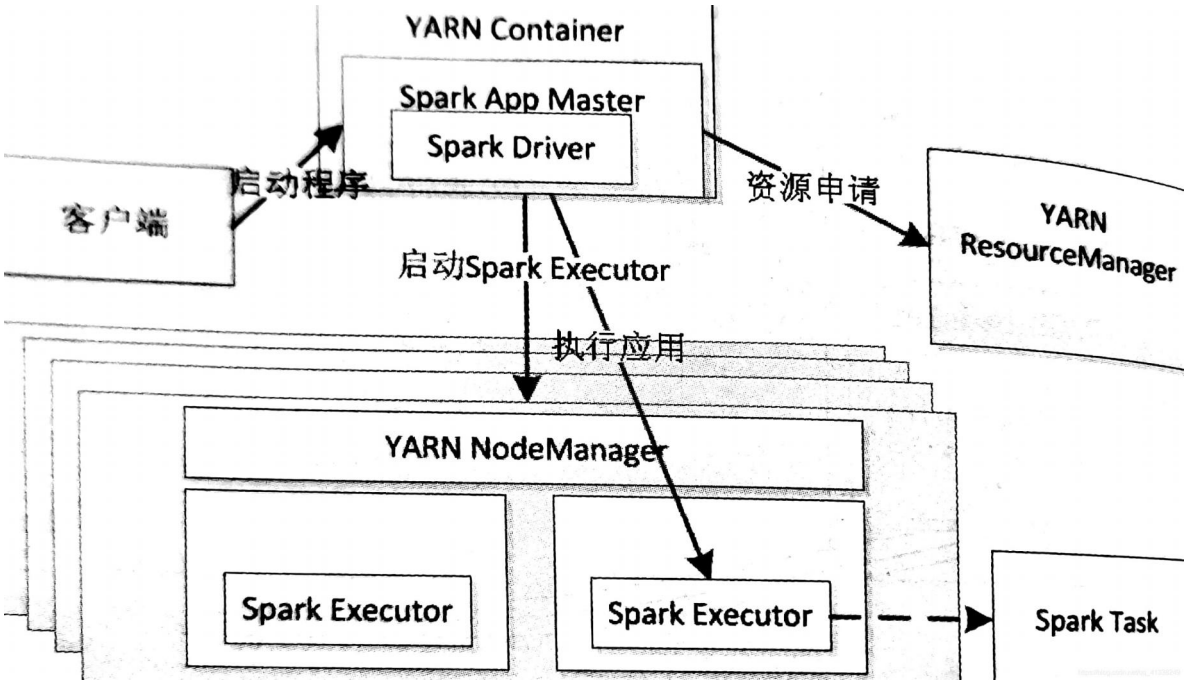
在YARN模式中，不需要启动Spark独立集群，所以这个时候去访问 http://master:8080 是访问不了的。启动YARN客户端模式的Spark Shell命令如下所示：

```
hadoop@master:/usr/local/spark$ ./bin/spark-shell --master yarn-client
```

在**YARN集群模式**下，Driver运行在Application Master上，Application Master进程同时负责驱动Application和从YARN中申请资源。该进程运行在YARN Container内，所以启动Application Master的Client可以立即关闭，而不必持续到Application的声明周期。

作业流程如下所示：

公众号：旧时光大数据



1）客户端生成作业信息提交给ResourceManager。

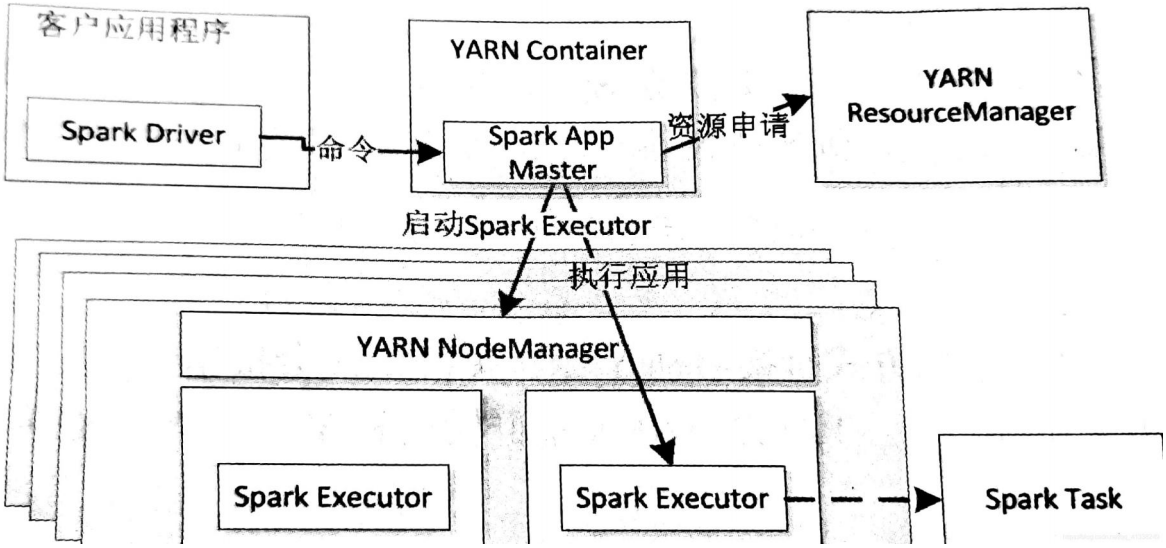
2）ResourceManager在某一个NodeManager(由YARN决定）启动Container，并将Application Master 分配给该NodeManager。

3）NodeManager接收到ResourceManager的分配，启动Application Master并初始化作业，此时NodeManager就称为Driver。

4）Application向ResourceManager申请资源，ResourceManager分配资源的同时通知其他NodeManager启动相应的Executor。

5）Executor向NodeManager上的Application Master注册汇报并完成相应的任务。

如下图是**YARN客户端模式**的作业运行流程。Application Master仅仅从YARN中申请资源给Executpr。之后Client会与Container通信进行作业调度。



YARN模式的作业运行调度描述如下

1）客户端生成作业信息提交给ResourceManager

2）ResourceManager在本地NodeManager启动Container，并将Application Master分配给该NodeManager。

3）NodeManager接收到ResourceManager的分配，启动Application Master并初始化作业，此时这个NodeManager就称为Driver。

4）Application向ResourceManager申请资源，ResouceManager分配资源同时通知其他NodeManager启动相应的Executor。

5）Executor向本地启动的Application Master注册汇报并完成相应的任务。

YARN-Cluster和YARN-Client区别

在YARN-Cluster模式下，SparkDriver运行在Application Master(AM)中，它负责向YARN申请资源，并监督作业的运行状况。当用户提交了作业之后，就可以关掉Client，作业会继续在YARN上运行，所以YARN-Cluster模式不适合运行交互类型的作业。然而在YARN-Client模式下，AM仅仅向YARN请求Executor，Client会与请求得到的Container通信来调度它们的工作，也就是是Client不能离开。

总结来说，集群模式的Spark Driver运行在AM中，而客户端模式的Spark Driver运行在客户端。所以，YARN-Cluster适用于生产，而YARN-Client适用于交互和调试，也就是希望快速地看到应用的输出信息。

Spark的特点

问过的一些公司：哈啰

参考答案：

1）快：与Hadoop的MapReduce相比，Spark基于内存的运算要快100倍以上，基于硬盘的运算也要快10倍以上。Spark实现了高效的DAG执行引擎，可以通过基于内存来高效处理数据流。计算的中间结果是存在于内存中的。

2）易用：Spark支持Java、Python和Scala的API，还支持超过80种高级算法，使用户可以快速构建不同的应用。而且Spark支持交互式的Python和Scala的Shell，可以非常方便地在这些Shell中使用Spark集群来验证解决问题的方法。

3）通用：Spark提供了统一的解决方案。Spark可以用于，交互式查询(Spark SQL)、实时流处理(Spark Streaming)、机器学习(Spark MLlib)和图计算(Graphx)。这些不同类型的处理都可以在同一个应用中无缝使用。减少了开发和维护的人力成本和部署平台的物力成本。

4）兼容性:Spark可以非常方便地与其他的开源产品进行融合。比如，Spark可以使用Hadoop的YARN和Apache Mesos作为它的资源管理和调度器，并且可以处理所有Hadoop支持的数据，包括HDFS、HBase等。这对于已经部署Hadoop集群的用户特别重要，因为不需要做任何数据迁移就可以使用Spark的强大处理能力。

Spark源码中的任务调度

问过的一些公司：字节，快手

参考答案：

DAG的生成

DAG(Directed Acyclic Graph)叫做有向无环图，原始的RDD通过一系列的转换就就形成了DAG，根据RDD之间的依赖关系的不同将DAG划分成不同的Stage，对于窄依赖，partition的转换处理在Stage中完成计算。对于宽依赖，由于有Shuffle的存在，只能在parent RDD处理完成后，才能开始接下来的计算，因此**宽依赖是划分Stage的依据。**

窄依赖：指的是每一个父RDD的Partition最多被子RDD的一个Partition使用

公众号：旧时光大数据

宽依赖：指的是多个子RDD的Partition会依赖同一个父RDD的Partition

DAGScheduler调度队列

```
class SparkContext(config: SparkConf) extends Logging with
ExecutorAllocationClient {
  。。。。。。

 private[spark] def createSparkEnv(
      conf: SparkConf,
      isLocal: Boolean,
      listenerBus: LiveListenerBus): SparkEnv = {
    //通过SparkEnv来创建createDriverEnv
    SparkEnv.createDriverEnv(conf, isLocal, listenerBus)
  }
  //在这里调用了createSparkEnv，返回一个SparkEnv对象，这个对象里面有很多重要属性，最重要的
ActorSystem
  private[spark] val env = createSparkEnv(conf, isLocal, listenerBus)
  SparkEnv.set(env)


  //创建taskScheduler
  // Create and start the scheduler
  private[spark] var (schedulerBackend, taskScheduler) =
    SparkContext.createTaskScheduler(this, master)

  //创建DAGScheduler
  dagScheduler = new DAGScheduler(this)

  //启动TaksScheduler
  taskScheduler.start()
    。。。。。
}
```

在构造方法中还创建了一个DAGScheduler对象，这个类的任务就是用来划分Stage任务的，构造方法中初始化了 `private[scheduler] val eventProcessLoop = new DAGSchedulerEventProcessLoop(this)`

DAGSchedulerEventProcessLoop是一个事件总线对象，用来负责任务的分发，在构造方法eventProcessLoop.start()被调用，该方法是父类EventLoop的start

```
 def start(): Unit = {
    if (stopped.get) {
      throw new IllegalStateException(name + " has already been stopped")
    }
    // Call onStart before starting the event thread to make sure it happens
before onReceive
    onStart()
    eventThread.start()
  }
```

调用了eventThread的start方法，开启了一个线程

```
  private val eventThread = new Thread(name) {
    setDaemon(true)

    override def run(): Unit = {
      try {
```

公众号：旧时光大数据

```
          while (!stopped.get) {
            val event = eventQueue.take()
            try {
              onReceive(event)
            } catch {
              case NonFatal(e) => {
                try {
                  onError(e)
                } catch {
                  case NonFatal(e) => logError("Unexpected error in " + name, e)
                }
              }
            }
          }
        } catch {
          case ie: InterruptedException => // exit even if eventQueue is not empty
          case NonFatal(e) => logError("Unexpected error in " + name, e)
        }
      }
    }
```

run方法中不断的从LinkedBlockingDeque阻塞队列中取消息，然后调用 `onReceive(event)` 方法，该方法是由子类DAGSchedulerEventProcessLoop实现的

```
  override def onReceive(event: DAGSchedulerEvent): Unit = event match {
    case JobSubmitted(jobId, rdd, func, partitions, allowLocal, callSite,
listener, properties) =>
      //调用dagScheduler来出来提交任务
      dagScheduler.handleJobSubmitted(jobId, rdd, func, partitions, allowLocal,
callSite,
        listener, properties)

    case StageCancelled(stageId) =>
      dagScheduler.handleStageCancellation(stageId)

    case JobCancelled(jobId) =>
      dagScheduler.handleJobCancellation(jobId)

    case JobGroupCancelled(groupId) =>
      dagScheduler.handleJobGroupCancelled(groupId)

    case AllJobsCancelled =>
      dagScheduler.doCancelAllJobs()

    case ExecutorAdded(execId, host) =>
      dagScheduler.handleExecutorAdded(execId, host)

    case ExecutorLost(execId) =>
      dagScheduler.handleExecutorLost(execId, fetchFailed = false)

    case BeginEvent(task, taskInfo) =>
      dagScheduler.handleBeginEvent(task, taskInfo)

    case GettingResultEvent(taskInfo) =>
      dagScheduler.handleGetTaskResult(taskInfo)

    case completion @ CompletionEvent(task, reason, _, _, taskInfo, taskMetrics)
=>
```

公众号：旧时光大数据

```
32          dagScheduler.handleTaskCompletion(completion)
33
34        case TaskSetFailed(taskSet, reason) =>
35          dagScheduler.handleTaskSetFailed(taskSet, reason)
36
37        case ResubmitFailedStages =>
38          dagScheduler.resubmitFailedStages()
39      }
40
```

onReceive中会匹配到传入的任务类型，执行相应的逻辑。到此DAGScheduler的调度队列会一直挂起，不断轮询队列中的任务。

DAG提交Task任务流程

当RDD经过一系列的转换Transformation方法后，最终要执行Action动作方法，这里比如WordCount程序中最后调用`collect()`方法时会将数据提交到Master上运行，任务真正的被执行，这里的方法执行过程如下

```
1     /**
2      * Return an array that contains all of the elements in this RDD.
3      */
4     def collect(): Array[T] = {
5       val results = sc.runJob(this, (iter: Iterator[T]) => iter.toArray)
6       Array.concat(results: _*)
7     }
```

`sc`是SparkContext对象，这里调用 一个`runJob`该方法调用多次重载的方法后,该方法最终会调用`dagScheduler.runJob`

```
1     def runJob[T, U: ClassTag](
2         rdd: RDD[T],
3         func: (TaskContext, Iterator[T]) => U,
4         partitions: Seq[Int],
5         allowLocal: Boolean,
6         resultHandler: (Int, U) => Unit) {
7       if (stopped) {
8         throw new IllegalStateException("SparkContext has been shutdown")
9       }
10      val callSite = getCallSite
11      val cleanedFunc = clean(func)
12      logInfo("Starting job: " + callSite.shortForm)
13      if (conf.getBoolean("spark.logLineage", false)) {
14        logInfo("RDD's recursive dependencies:\n" + rdd.toDebugString)
15      }
16      //dagScheduler出现了，可以切分stage
17      dagScheduler.runJob(rdd, cleanedFunc, partitions, callSite, allowLocal,
18        resultHandler, localProperties.get)
19      progressBar.foreach(_.finishAll())
20      rdd.doCheckpoint()
21    }
22
```

dagScheduler的`runJob`是我们比较关心的

```
1     def runJob[T, U: ClassTag](
2       。。。。。
```

公众号：旧时光大数据

```
    val waiter = submitJob(rdd, func, partitions, callSite, allowLocal,
resultHandler, properties)
    waiter.awaitResult() match {
      case JobSucceeded => {
        logInfo("Job %d finished: %s, took %f s".format
          (waiter.jobId, callSite.shortForm, (System.nanoTime - start) / 1e9))
      }
      case JobFailed(exception: Exception) =>
        logInfo("Job %d failed: %s, took %f s".format
          (waiter.jobId, callSite.shortForm, (System.nanoTime - start) / 1e9))
        throw exception
    }
  }
```

这里面的我们主要看的是`submitJob(rdd, func, partitions, callSite, allowLocal, resultHandler, properties)`提交任务

```
def submitJob[T, U](
      rdd: RDD[T],
      func: (TaskContext, Iterator[T]) => U,
      partitions: Seq[Int],
      callSite: CallSite,
      allowLocal: Boolean,
      resultHandler: (Int, U) => Unit,
      properties: Properties): JobWaiter[U] = {

     。。。。。。

    //把job加入到任务队列里面
    eventProcessLoop.post(JobSubmitted(
      jobId, rdd, func2, partitions.toArray, allowLocal, callSite, waiter,
properties))
    waiter
  }
```

这里比较关键的地方是`eventProcessLoop.post`往任务队列中加入一个JobSubmitted类型的任务，eventProcessLoop是在构造方法中就初始化好的事件总线对象，内部有一个线程不断的轮询队列里的任务

轮询到任务后调用`onReceive`方法匹配任务类型，在这里我们提交的任务是JobSubmitted类型

```
case JobSubmitted(jobId, rdd, func, partitions, allowLocal, callSite, listener,
properties) =>
     //调用dagScheduler来出来提交任务
     dagScheduler.handleJobSubmitted(jobId, rdd, func, partitions, allowLocal,
callSite,
      listener, properties)
```

调用了`handleJobSubmitted`方法，接下来查看该方法

```
private[scheduler] def handleJobSubmitted(jobId: Int,
      finalRDD: RDD[_],
      func: (TaskContext, Iterator[_]) => _,
      partitions: Array[Int],
      allowLocal: Boolean,
      callSite: CallSite,
      listener: JobListener,
      properties: Properties) {
```

公众号：旧时光大数据

```
    var finalStage: Stage = null
    try {
      // New stage creation may throw an exception if, for example, jobs are run on a
      // HadoopRDD whose underlying HDFS files have been deleted.
      //最终的stage
      finalStage = newStage(finalRDD, partitions.size, None, jobId, callSite)
    } catch {
      case e: Exception =>
        logWarning("Creating new stage failed due to exception - job: " + jobId, e)
        listener.jobFailed(e)
        return
    }
        。。。。
        submitStage(finalStage)
  }
```

上面的代码中，调用了 `newStage` 进行任务的划分，该方法是划分任务的核心方法，划分任务的根据最后一个依赖关系作为开始，通过递归，将每个宽依赖做为切分Stage的依据，切分Stage的过程是流程中的一环，但在这里不详细阐述，当任务切分完毕后，代码继续执行来到 `submitStage(finalStage)` 这里开始进行任务提交

这里以递归的方式进行任务的提交

```
//递归的方式提交stage
  private def submitStage(stage: Stage) {
    val jobId = activeJobForStage(stage)
    if (jobId.isDefined) {
      logDebug("submitStage(" + stage + ")")
      if (!waitingStages(stage) && !runningStages(stage) && !failedStages(stage)) {
        val missing = getMissingParentStages(stage).sortBy(_.id)
        logDebug("missing: " + missing)
        if (missing == Nil) {
          logInfo("Submitting " + stage + " (" + stage.rdd + "), which has no missing parents")
            //提交任务
          submitMissingTasks(stage, jobId.get)
        } else {
          for (parent <- missing) {
            submitStage(parent)
          }
          waitingStages += stage
        }
      }
    } else {
      abortStage(stage, "No active job for stage " + stage.id)
    }
  }
```

调用 `submitMissingTasks(stage, jobId.get)` 提交任务，将每一个Stage和jobId传入

```
  private def submitMissingTasks(stage: Stage, jobId: Int) {
   。。。。。

    if (tasks.size > 0) {
```

公众号：旧时光大数据

```
        logInfo("Submitting " + tasks.size + " missing tasks from " + stage + " (" + stage.rdd + ")")
        stage.pendingTasks ++= tasks
        logDebug("New pending tasks: " + stage.pendingTasks)
        //taskScheduler提交task
        taskScheduler.submitTasks(
          new TaskSet(tasks.toArray, stage.id, stage.newAttemptId(), stage.jobId, properties))
        stage.latestInfo.submissionTime = Some(clock.getTimeMillis())
      } else {
        // Because we posted SparkListenerStageSubmitted earlier, we should mark
        // the stage as completed here in case there are no tasks to run
        markStageAsFinished(stage, None)
        logDebug("Stage " + stage + " is actually done; %b %d %d".format(
          stage.isAvailable, stage.numAvailableOutputs, stage.numPartitions))
      }
    }
```

这里的代码我们需要关注的是`taskScheduler.submitTasks(new TaskSet(tasks.toArray, stage.id, stage.newAttemptId(), stage.jobId, properties))`创建了一个TaskSet对象，将所有任务的信息封装，包括task任务列表，stageId,任务id,分区数参数等

Task任务调度

```
override def submitTasks(taskSet: TaskSet) {
    val tasks = taskSet.tasks
    logInfo("Adding task set " + taskSet.id + " with " + tasks.length + " tasks")
    this.synchronized {
        //创建TaskSetManager保存了taskSet任务列表
      val manager = createTaskSetManager(taskSet, maxTaskFailures)
      activeTaskSets(taskSet.id) = manager
     //将任务加入调度池
      schedulableBuilder.addTaskSetManager(manager, manager.taskSet.properties)

      if (!isLocal && !hasReceivedTask) {
        starvationTimer.scheduleAtFixedRate(new TimerTask() {
          override def run() {
            if (!hasLaunchedTask) {
              logWarning("Initial job has not accepted any resources; " +
                "check your cluster UI to ensure that workers are registered " +
                "and have sufficient resources")
            } else {
              this.cancel()
            }
          }
        }, STARVATION_TIMEOUT, STARVATION_TIMEOUT)
      }
      hasReceivedTask = true
    }
    //接受任务
    backend.reviveOffers()
  }
```

该方法比较重要，主要将任务加入调度池，最后调用了`backend.reviveOffers()`这里的backend是CoarseGrainedSchedulerBackend一个Executor任务调度对象

公众号：旧时光大数据

```
override def reviveOffers() {
  //自己给自己发消息
  driverActor ! ReviveOffers
}
```

这里用了内部的DriverActor对象发送了一个内部消息给自己，接下来查看receiver方法接受的消息

```
case ReviveOffers =>
    makeOffers()
```

收到消息后调用了`makeOffers()`方法

```
    def makeOffers() {
      launchTasks(scheduler.resourceOffers(executorDataMap.map { case (id,
executorData) =>
        new WorkerOffer(id, executorData.executorHost, executorData.freeCores)
      }.toSeq))
    }

```

makeOffers方法中，将Executor的信息集合与调度池中的Tasks封装成WokerOffers列表传给了`launchTasks`

```
    def launchTasks(tasks: Seq[Seq[TaskDescription]]) {
      for (task <- tasks.flatten) {
        。。。。。。
        //把task序列化
        val serializedTask = ser.serialize(task)

          。。。。。
        val executorData = executorDataMap(task.executorId)
        executorData.freeCores -= scheduler.CPUS_PER_TASK
        //把序列化好的task发送给Executor
        executorData.executorActor ! LaunchTask(new
SerializableBuffer(serializedTask))
      }
    }
  }

```

aunchTasks方法将遍历Tasks集合,每个Task任务序列化，发送启动Task执行消息的给Executor

Executor的onReceive方法

```
  //DriverActor发送给Executor的启动Task的消息
    case LaunchTask(data) =>
      if (executor == null) {
        logError("Received LaunchTask command but executor was null")
        System.exit(1)
      } else {
        val ser = env.closureSerializer.newInstance()
        //把Task反序列化
        val taskDesc = ser.deserialize[TaskDescription](data.value)
        logInfo("Got assigned task " + taskDesc.taskId)
        //启动task
        executor.launchTask(this, taskId = taskDesc.taskId, attemptNumber =
taskDesc.attemptNumber,
          taskDesc.name, taskDesc.serializedTask)
```

公众号：旧时光大数据

```
        }

```

Executor收到DriverActor发送的启动Task的消息，这里才开始真正执行任务了，将收到的Task序列化信息反序列化,调用 `Executor` 的 `launchTask` 方法执行任务

```
    def launchTask(
        context: ExecutorBackend,
        taskId: Long,
        attemptNumber: Int,
        taskName: String,
        serializedTask: ByteBuffer) {
      //把task的描述信息放到了一份TaskRunner
      val tr = new TaskRunner(context, taskId = taskId, attemptNumber =
attemptNumber, taskName,
        serializedTask)
      runningTasks.put(taskId, tr)
      //然后把TaskRunner丢到线程池里面
      threadPool.execute(tr)
    }

```

launchTask内将Task提交到线程池去运行，TaskRunner是Runnable对象，里面的run方法执行了我们app生成的每一个RDD的链上的逻辑。

Spark作业调度

问过的一些公司：字节

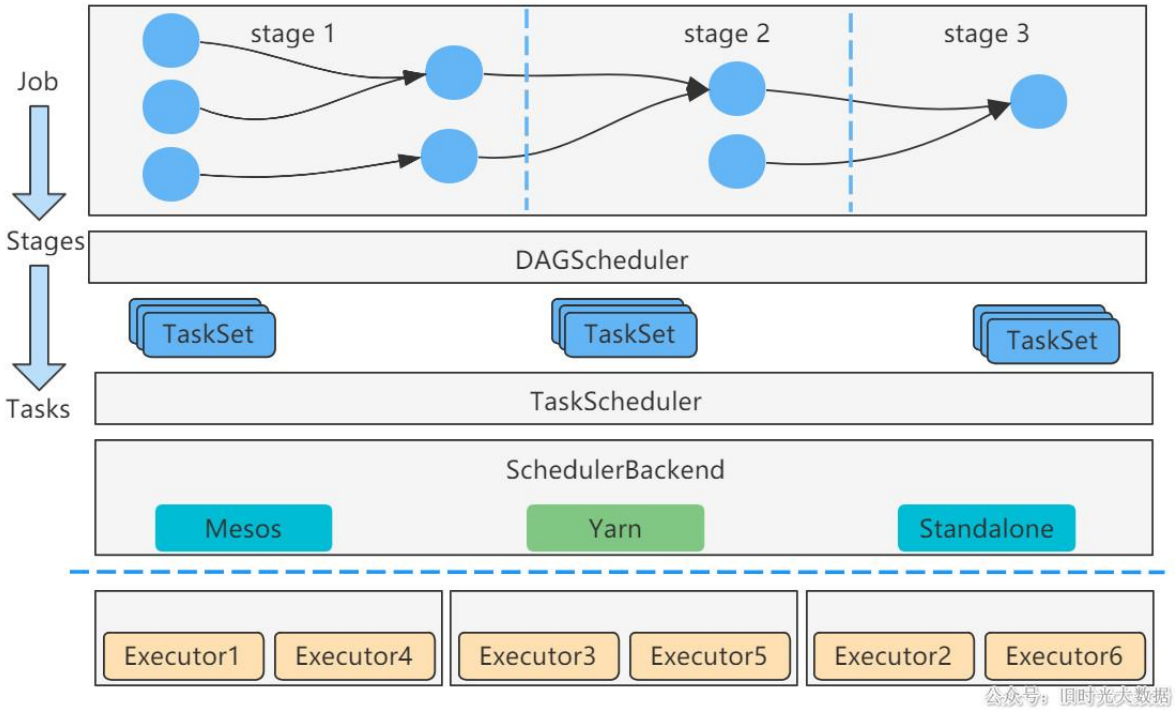
参考答案：

版本一：

Driver线程主要是初始化SparkContext对象，准备运行所需的上下文，然后一方面保持与ApplicationMaster的RPC连接，通过ApplicationMaster申请资源，另一方面根据用户业务逻辑开始调度任务，将任务下发到已有的空闲Executor上。当Driver起来后，Driver则会根据用户程序逻辑准备任务，并根据Executor资源情况逐步分发任务。

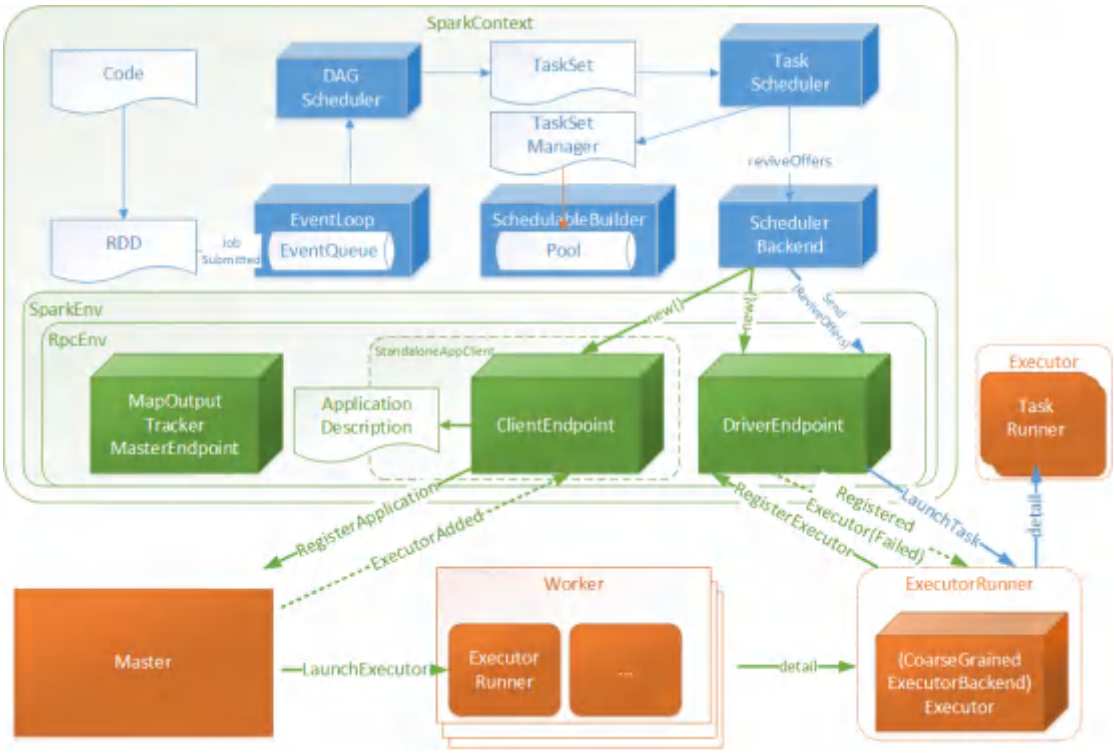
Spark的任务调度总体来说分两路进行，**一路是Stage级的调度**，**一路是Task级的调度**，总体调度流程如下图所示：

公众号：旧时光大数据



Spark RDD通过其Transactions操作，形成了RDD血缘（依赖）关系图，即DAG，最后通过Action的调用，触发Job并调度执行，执行过程中会创建两个调度器：DAGScheduler和TaskScheduler。

- DAGScheduler负责Stage级的调度，主要是将job切分成若干Stages，并将每个Stage打包成TaskSet交给TaskScheduler调度。
- TaskScheduler负责Task级的调度，将DAGScheduler给过来的TaskSet按照指定的调度策略分发到Executor上执行，调度过程中SchedulerBackend负责提供可用资源，其中SchedulerBackend有多种实现，分别对接不同的资源管理系统。

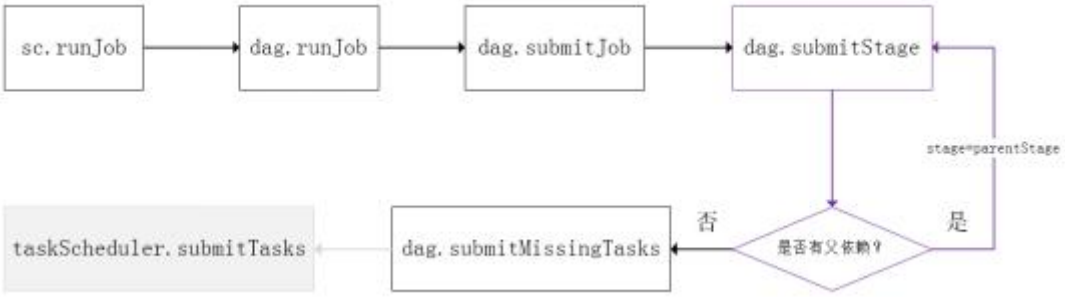


Driver初始化SparkContext过程中，会分别初始化DAGScheduler、TaskScheduler、SchedulerBackend以及HeartbeatReceiver，并启动SchedulerBackend以及HeartbeatReceiver。SchedulerBackend通过ApplicationMaster申请资源，并不断从TaskScheduler中拿到合适的Task分发到Executor执行。HeartbeatReceiver负责接收Executor的心跳信息，监控Executor的存活状况，并通知到TaskScheduler。

1、Spark Stage级调度

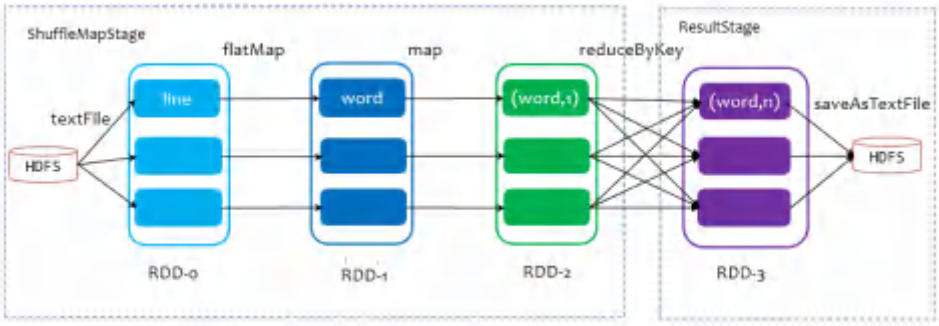
公众号：旧时光大数据

Spark的任务调度是从DAG切割开始，主要是由DAGScheduler来完成。当遇到一个Action操作后就会触发一个Job的计算，并交给DAGScheduler来提交，下图是涉及到Job提交的相关方法调用流程图。



Job由最终的RDD和Action方法封装而成

SparkContext将Job交给DAGScheduler提交，它会根据RDD的血缘关系构成的DAG进行切分，将一个Job划分为若干Stages，具体划分策略是，由最终的RDD不断通过依赖回溯判断父依赖是否是宽依赖，即以Shuffle为界，划分Stage，窄依赖的RDD之间被划分到同一个Stage中，可以进行pipeline式的计算。划分的Stages分两类，一类叫做ResultStage，为DAG最下游的Stage，由Action方法决定，另一类叫做ShuffleMapStage，为下游Stage准备数据，下面看一个简单的例子WordCount。



Job由saveAsTextFile触发，该Job由RDD-3和saveAsTextFile方法组成，根据RDD之间的依赖关系从RDD-3开始回溯搜索，直到没有依赖的RDD-0，在回溯搜索过程中，RDD-3依赖RDD-2，并且是宽依赖，所以在RDD-2和RDD-3之间划分Stage，RDD-3被划到最后一个Stage，即ResultStage中，RDD-2依赖RDD-1，RDD-1依赖RDD-0，这些依赖都是窄依赖，所以将RDD-0、RDD-1和RDD-2划分到同一个Stage，形成pipeline操作，。即ShuffleMapStage中，实际执行的时候，数据记录会一气呵成地执行RDD-0到RDD-2的转化。不难看出，其本质上是一个深度优先搜索（Depth First Search）算法。

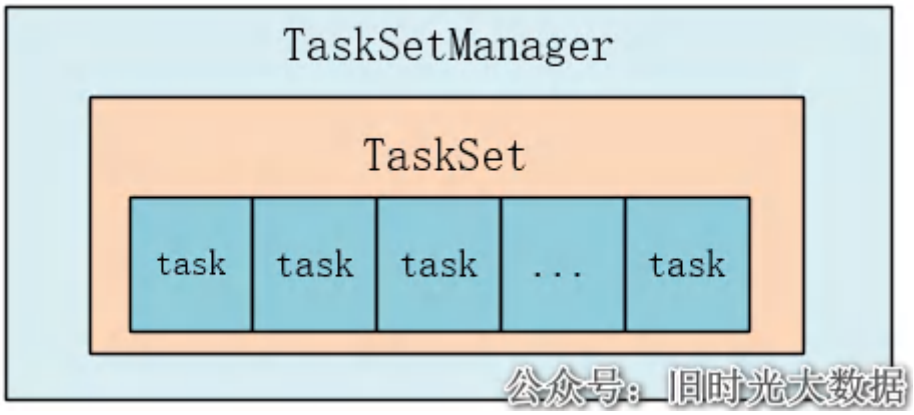
一个Stage是否被提交，需要判断它的父Stage是否执行，只有在父Stage执行完毕才能提交当前Stage，如果一个Stage没有父Stage，那么从该Stage开始提交。Stage提交时会将Task信息（分区信息以及方法等）序列化并被打包成TaskSet交给TaskScheduler，一个Partition对应一个Task，另一方面TaskScheduler会监控Stage的运行状态，只有Executor丢失或者Task由于Fetch失败才需要重新提交失败的Stage以调度运行失败的任务，其他类型的Task失败会在TaskScheduler的调度过程中重试。

相对来说DAGScheduler做的事情较为简单，仅仅是在Stage层面上划分DAG，提交Stage并监控相关状态信息。TaskScheduler则相对较为复杂，下面详细阐述其细节。

2、Spark Task级调度

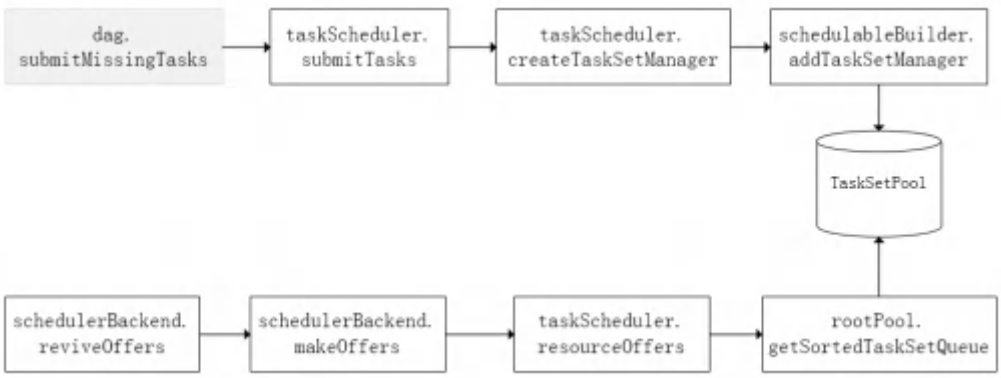
Spark Task的调度是由TaskScheduler来完成，由前文可知，DAGScheduler将Stage打包到交给TaskScheTaskSetduler，TaskScheduler会将TaskSet封装为TaskSetManager加入到调度队列中，TaskSetManager结构如下图所示。

公众号：旧时光大数据



TaskSetManager负责监控管理同一个Stage中的Tasks，TaskScheduler就是以TaskSetManager为单元来调度任务。

前面也提到，TaskScheduler初始化后会启动SchedulerBackend，它负责跟外界打交道，接收Executor的注册信息，并维护Executor的状态，所以说SchedulerBackend是管“粮食”的，同时它在启动后会定期地去“询问”TaskScheduler有没有任务要运行，也就是说，它会定期地“问”TaskScheduler“我有这么余粮，你要不要啊”，TaskScheduler在SchedulerBackend“问”它的时候，会从调度队列中按照指定的调度策略选择TaskSetManager去调度运行，大致方法调用流程如下图所示：



上图中，将TaskSetManager加入rootPool调度池中之后，调用SchedulerBackend的riviveOffers方法给driverEndpoint发送ReviveOffer消息；driverEndpoint收到ReviveOffer消息后调用makeOffers方法，过滤出活跃状态的Executor（这些Executor都是任务启动时反向注册到Driver的Executor），然后将Executor封装成WorkerOffer对象；准备好计算资源（WorkerOffer）后，taskScheduler基于这些资源调用resourceOffer在Executor上分配task。

版本二：

Spark调度机制可以理解为两个层面的调度。**Spark Application调度**(Spark应用程序在集群中运行的调度,包括Driver调度和Executor调度)和**单个Spark应用程序SparkContext的内部调度。**

SparkContext内部调度就是每个Spark Application都会有若干Jobs(Spark Actions)，然后这些job是以何种机制在Executor上执行的，也是需要一个调度管理的机制，该层面调度可以理解为SparkContext内部调度。若干 Jobs 由不同的线程提交，它们可以同时运行，此时可以使用 Spark Application 的 Fair 机制的调度资源。

1、Applications间的调度

在集群上运行时，每个Spark应用程序都获得一组独立的 executor jvm，这些jvm只运行该应用程序的 tasks 和缓存该应用程序的数据。当有多个应用或者多个程序在你的集群中运行时，根据集群管理器的不同，可以使用不同的选项来管理分配。

最简单的方式是提供静态资源分配，也就是给每个应用程序分配固定资源，资源数在该程序整个运行期间都不会变动。这种方式出现在Spark的 Standalone，YARN 和粗粒度 Mesos 模式。

资源的分配方式，在每种集群运行模式中有所不同：

公众号：旧时光大数据

1）standalone模式

默认情况下，提交到独立模式群集的应用程序将以FIFO(先进先出)方式运行，每个应用程序将尝试使用所有可用节点。可以通过 `spark.cores.max` 配置属性来限制应用程序使用的节点数，或者通过 `spark.deploy.defaultCores` 更改未设置此应用程序的默认值。最后，除了控制 cores 外，每个应用程序可以使用 `spark.executor.memory` 配置控制其内存使用。

2）Mesos

要在Mesos上使用静态分区，请将spark.mesos.coarse配置属性设置为true，并可选择像Standalone模式一样设置 `spark.cores.max` 来限制每个应用程序的资源。您还应该设置 `spark.executor.memory` 来控制执行程序内存。

3）YARN

YARN客户端的 `--num-executors Spark` 选项控制在集群上分配的Executor数量（ `spark.executor.instances` 配置属性），而 `--executor-memory` （spark.executor.memory配置属性）和 `--executor-cores` (spark.executor.cores配置属性)控制每个执行程序的资源。

在Mesos模式下还有一个可选项是动态共享 Core 。在这种模式下，
每个Spark应用程序仍然拥有固定数量和独立的内存(spark.executor.memory设置)，但是当Spark 应用程序在一个机器上没有运行的 task 的时候，其它的应用程序可以使用这些 cores 运行 tasks 。当你需要运行大量不活跃的 Spark 应用程序(例如来自不同用户的shell会话)时，此模式非常有用。但是此模式存在延迟可预测性较低的风险，因为当Spark App需要恢复使用这些cores的时候，需要等待一些时间才能使用这些core去执行任务。要使用此模式，只需要使用 mesos://URL并将spark.executor.coarse设置为false。

注意： 目前没有一种模式可以跨应用程序提供内存共享。如果想进行跨应用程序共享数据，建议运行单个服务器应用程序，通过查询相同的RDD来提供多个请求或使用第三方存储，例如 Tachyon(是一个高性能、高容错、基于内存的开源分布式存储系统)来实现内存共享。

2、动态资源分配

Spark提供了一种可根据工作负载动态调整应用程序占用资源的机制。这意味着如果不再使用应用程序，应用程序可能会将资源返回给群集，并在需要时再次请求它们。如果多个应用程序共享Spark群集中的资源，则此功能特别有用。

默认情况下此功能禁用，并且可在所有粗粒度集群管理器上使用，即 Standalone，YARN模式和 Mesos粗粒度模式。

1）配置

使用此功能有两个要求。

第一，Spark 应用程序必须设置 spark.dynamicAllocation.enabled 为 true。

第二，必须在同一集群中的每个工作节点上设置外部shuffle服务，并设置 spark.shuffle.service.enabled 为 true。外部shuffle服务的目的是允许删除执行程序而不删除由它们写入的 shuffle 文件。

不同的集群管理器，该服务的设置方式有所不同：

- **Standalone 模式**
  - 只需在 `spark.shuffle.service.enabled` 设置为 true 的情况下启动 Worker 即可。
- **粗粒度 Mesos 模式**
  - 在 `spark.shuffle.service.enabled` 设置为 true 的所有从属节点上，运行 $SPARK_HOME/sbin/start-mesos-shuffle-service.sh 脚本
- YARN模式
  - 按如下所示在每个 NodeManager 上启动 shuffle 服务:

公众号：旧时光大数据

- 在编译 Spark 的时候要添加 YARN 配置。如果已经添加该属性，并分发到集群中，跳过此步骤。
- 找到 `spark-<version>-yarn-shuffle.jar`，如果是你自己编译的 Spark 该 jar 应该在 `$SPARK_HOME/common/network-yarn/target/scala-<version>` 目录下，如果使用的是一个发行版，则应该在 yarn 下面。
- 将此 jar 添加到集群中所有的 NodeManager 的 Classpath 下。
- 在每个节点的 `yarn-site.xml` 上，给属性 `yarn.nodemanager.aux-services` 增加一个 `spark_shuffle` 值，然后将 `yarn.nodemanager.aux-services.spark_shuffle.class` 设置为 `org.apache.spark.network.yarn.YarnShuffleService`。
- 通过设置 NodeManager 节点的 `etc/hadoop/yarn-env.sh` 文件中的 `YARN_HEAPSIZE` (默认为 1000)来增加堆大小，以避免在shuffle期间出现垃圾收集问题。
- 重启集群中的所有 NodeManager。当shuffle服务在YARN上运行时，可以使用以下额外配置选项：

参数	默认	含义
spark.yarn.shuffle.stopOnFailure	false	是否在Spark Shuffle Service初始化失败时停止NodeManager。可以防止在NodeManagers上运行Containers却未运行Spark Shuffle Service而导致的应用程序故障。

2）资源配置策略

在较高的层面上，Spark 应该在不再使用执行程序时放弃执行程序，并在需要时获取执行程序。由于没有确定的方法可以预测即将被删除的执行程序是否会在不久的将来运行任务，或者即将添加的新执行程序实际上是否是空闲的，所以需要一组启发式来确定何时删除或请求 executors。

请求策略

当启用动态分配的 Spark 应用程序有等待调度的挂起任务时，它会请求额外的执行器。这意味着现有的 Executors 不足以同时满足所有已提交但尚未完成的任务。

Spark会轮询申请资源。如果挂起的任务持续 spark.dynamicAllocation.schedulerBacklogTimeout 数秒时会触发实际请求，如果待处理任务队列持续存在，则会在此后每隔 spark.dynamicAllocation.sustainedSchedulerBacklogTimeout 秒将再次触发。此外，每轮请求的 Executors 数量与上一轮相比会呈指数增长。例如，一个应用程序将在第一轮中添加1个 Executor，然后在后续轮次中添加 2,4,8 等个 Executor。

指数增长政策的动机是双重的。首先，应用程序应该在开始时谨慎地申请 Executor，以防只有几个额外的 Executor 就足够了。这与TCP慢启动的理由相呼应。其次，应用程序应该能够及时提高其资源使用率，以防实际需要许多执行程序。

删除策略

删除 Executors 的策略要简单得多。Spark 应用程序在空闲时间超过 spark.dynamicAllocation.executorIdleTimeout 秒时会删除 Executors。请注意，在大多数情况下，此条件与请求条件互斥，因为如果仍有待执行的任务时，则 Executor 不应处于空闲状态。

3）Executors 退出

在动态分配之前，Spark Executors在出现故障或退出相关应用程序时退出。在这两种情况下，与 Executors 相关联的所有状态不再需要，可以安全地丢弃。但是，通过动态分配，当显式删除 Executors 时，应用程序仍在运行。如果应用程序尝试访问由 Executors 存储或写入的状态存，则必须执行重新计算状态。因此，Spark需要一种机制，通过在删除执行程序之前保留其状态来优雅地停用 Executors。

公众号：旧时光大数据

这一要求对于 Shuffle 尤其重要。在 Shuffle 期间，Spark Executor 首先将 map 输出本地写入磁盘，然后当 Executor 尝试获取这些文件时充当这些文件的服务器。如果存在运行时间比其他 Executor 长得多的任 tasks 时，动态申请的 Executor 有可能在 Shuffle 未结束之前就被移除了，在这种情况下，必须重新计算由该 Executors 写入的 Shuffle 文件。

可以使用外部 shuffle 服务保存 shuffle 输出文件，从 Spark 1.2 开始引入。此服务指的是一个长期运行的进程，它独立于 Spark应用程序及其 Executors ，在集群的每个节点上运行。如果启用该服务，Spark执行程序将从服务获取shuffle文件，而不是从其它 Executor。这意味着由执行人员写入的任何 shuffle 状态可能会继续执行超出 Executor 的生命周期。

除了写shuffle文件外，执行程序还可以在磁盘或内存中缓存数据。但是，删除执行程序后，所有缓存的数据将不再可访问。为了避免这种情况，默认情况下，永远不会删除包含缓存数据的 Executors。您可以使用 spark.dynamicAllocation.cachedExecutorIdleTimeout 配置此行为。在将来的版本中，缓存的数据可以通过堆外存储来保存，这在本质上类似于通过外部shuffle服务保存shuffle文件。

3、Spark App 内部调度

在给定的Spark应用程序(SparkContext实例)中，如果从单独的线程提交多个并行作业，那么它们可以同时运行。在本节中，我们所说的 job 是指一个 Spark action (例如，save，collect)和任何需要运行的任务以评估该操作。Spark 的调度程序是完全线程安全的，并支持此用例来启用服务于多个请求的应用程序(例如针对多个用户的查询)。

默认情况下，Spark的调度程序以FIFO方式运行作业。每个作业被划分为“阶段”(例如 map和reduce阶段)，第一个 job 运行结束之后，第二个job才有会去执行。如果在队列头部的job不需要使用集群的全部资源，那么后面的job可以立即执行。但是如果队列头部的 job 很大的话，其余的 job 必须推迟执行。

从Spark 0.8开始，还可以配置作业之间的公平共享。在公平共享下，Spark以“循环”的方式在作业之间分配 tasks，以便所有作业获得大致相等的集群资源共享。这意味着长job运行期间提交的短工作可以立即开始接收资源，并且仍然可以获得良好的响应时间，而不必等待长 job完成。此模式最适合多用户设置。

要启用公平调度程序，只需在配置 SparkContext 时将 spark.scheduler.mode 属性设置为 FAIR :

```
1   val conf = new SparkConf().setMaster(...).setAppName(...)
2   conf.set("spark.scheduler.mode", "FAIR")
3   val sc = new SparkContext(conf)
4
```

1）公平调度池

公平调度程序还支持将作业分组到池中，并为每个池设置不同的调度选项（例如: 权重）。这可以用于为更重要的job创建“高优先级”池，或者将每个用户的job组合在一起，并为用户提供相同的份额，而不管他们拥有多少并发作业，而不是给予作业相等的份额。此方法以Hadoop Fair Scheduler为模型 。

在没有任何干预的情况下，新提交的作业将进入默认池，可以通过向提交线程中的SparkContext 添加 spark.scheduler.pool “local property” 来设置作业池。如下:

```
1   // Assuming sc is your SparkContext variable
2   sc.setLocalProperty("spark.scheduler.pool", "pool1")
3
```

设置这个本地属性后，在此线程中提交所有的工作(通过此线程调用的 RDD.save，count，collect等)将使用此 pool 名称。该设置是每个线程，以便让线程代表同一个用户运行多个作业变得容易。如果要清除与线程关联的池，只需调用:

```
1   sc.setLocalProperty("spark.scheduler.pool", null)
2
```

公众号：旧时光大数据

2）池的默认行为

默认情况下，每个 pool 获得的群集份额相等(默认池中的每个作业的份额也相等)，但是每个池中的作业依然是FIFO的顺序运行。例如，如果您为每个用户创建一个池，这意味着每个用户将获得该群集的相等份额，并且每个用户的查询将按顺序运行，而不是以后的查询从该用户的早期查询中获取资源。

3）配置池属性

可以通过配置文件修改特定池的属性。每个池支持三个属性：

- schedulingMode
  - 这可以是FIFO或FAIR，用于控制池中的作业是否以队列的形式顺序执行(默认)或公平地共享池的资源。
- weight
  - 这可以控制池相对于其他池的共享。默认情况下，所有池的权重均为1.例如，如果将特定池权重设置为2，则将比其他活跃池多获得2倍的资源。设置高权重(例如1000)可以实现池之间的优先级 - 实质上，只要有活动作业，weight-1000池将始终首先启动任务。
- minShare
  - 除了总体权重之外，每个池都可以获得管理员希望拥有的最小份额(如一些CPU内核)。公平调度程序总是尝试在根据权重重新分配额外的资源之前满足所有活动池的最小份额。因此，minShare 属性可以是另一种确保池可以始终快速达到一定数量的资源（例如10个核心）的方法，而不会为集群的其余部分赋予高优先级。默认情况下，每个池的 minShare 值为0。

可以通过创建一个类似于conf/fairscheduler.xml.template 的 XML 文件来设置池属性，并在类路径中放置一个 fairscheduler.xml 文件，或者在SparkConf中 `spark.scheduler.allocation.file` 属性来设置池属性 。

```
conf.set("spark.scheduler.allocation.file", "/path/to/file")

```

XML文件的格式只是每个池的一个元素，其中包含不同的元素用于各种设置。例如：

```
<?xml version="1.0"?>
<allocations>
  <pool name="production">
    <schedulingMode>FAIR</schedulingMode>
    <weight>1</weight>
    <minShare>2</minShare>
  </pool>
  <pool name="test">
    <schedulingMode>FIFO</schedulingMode>
    <weight>2</weight>
    <minShare>3</minShare>
  </pool>
</allocations>

```

conf / fairscheduler.xml.template 中提供了一个完整示例。注意，未在XML文件中配置的任何池将只获取所有设置的默认值(调度模式FIFO，权重1和minShare 0)。

5、使用JDBC连接进行调度

要为JDBC客户端会话设置Fair Scheduler池，用户可以设置spark.sql.thriftserver.scheduler.pool变量：

```
SET spark.sql.thriftserver.scheduler.pool=accounting;

```

公众号：旧时光大数据

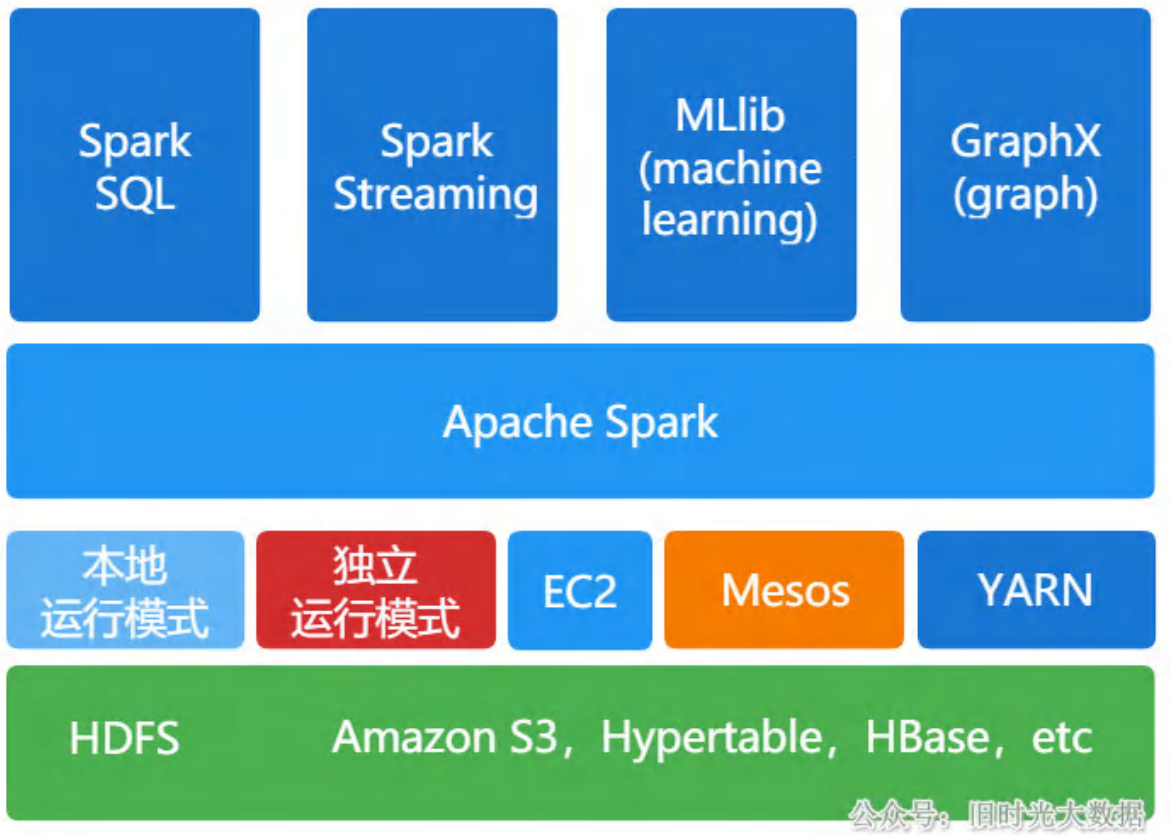
Spark的架构

可回答：Spark的运行架构?

问过的一些公司：阿里，今日头条，一点咨询，端点数据(2021.07)，美团(2021.08)

参考答案：

Spark架构示意图



Spark Core：包含Spark的基本功能；尤其是定义RDD的API、操作以及这两者上的动作。其他Spark的库都是构建在RDD和Spark Core之上的

Spark SQL：提供通过Apache Hive的SQL变体Hive查询语言（HiveQL）与Spark进行交互的API。每个数据库表被当做一个RDD，Spark SQL查询被转换为Spark操作。

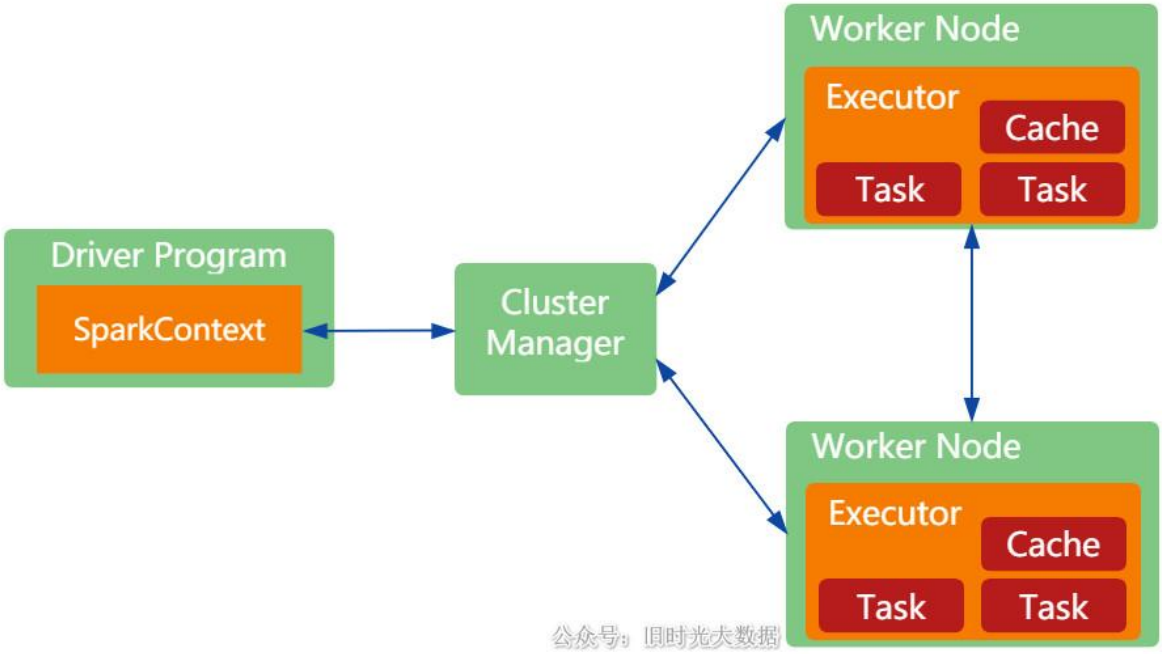
Spark Streaming：对实时数据流进行处理和控制。Spark Streaming允许程序能够像普通RDD一样处理实时数据

MLlib：一个常用机器学习算法库，算法被实现为对RDD的Spark操作。这个库包含可扩展的学习算法，比如分类、回归等需要对大量数据集进行迭代的操作。

GraphX：控制图、并行图操作和计算的一组算法和工具的集合。GraphX扩展了RDD API，包含控制图、创建子图、访问路径上所有顶点的操作

Spark架构的组成图

公众号：旧时光大数据



Cluster Manager：在standalone模式中即为Master主节点，控制整个集群，监控worker。在YARN模式中为资源管理器

Worker节点：从节点，负责控制计算节点，启动Executor或者Driver。

Driver： 运行Application 的main()函数

Executor：执行器，是为某个Application运行在worker node上的一个进程

Spark的使用场景

大数据场景主要有以下几种类型：

- 复杂的批处理（Batch Data Processing），偏重点在于处理。海量数据的能力，至于处理速度可忍受，通常的时间可能是在数十分钟到数小时；
- 基于历史数据的交互式查询（Interactive Query），通常的时间在数十秒到数十分钟之间；
- 基于实时数据流的大数据处理（Streaming Data Processing），通常在数百毫秒到数秒之间；

对于上述的情况，不使用Spark来处理的框架如下：

- 第一种情况可以用 Hadoop 的 MapReduce 来进行批量海量数据处理
- 第二种情况可以 Impala、Kylin 进行交互式查询
- 第三中情况可以用 Storm 分布式处理框架处理实时流式数据

使用Spark来进行处理：

- 第一种情况使用 Spark Core 解决
- 第二种情况使用 Spark SQL 解决
- 第三种情况使用 Spark Streaming 解决

综上所述，Spark使用场景如下：

- Spark 是基于内存的迭代计算框架，适用于需要多次操作特定数据集的应用场合。需要反复操作的次数越多，所需读取的数据量越大，受益越大，数据量小但是计算密集度较大 的场合，受益就相对较小；
- 由于 RDD 的特性，Spark 不适用那种异步细粒度更新状态的应用，例如 web 服务的存储或者是增量的 web 爬虫和索引。就是对于那种增量修改的应用模型不适合；

公众号：旧时光大数据

- 数据量不是特别大，但是要求实时统计分析需求；

满足以上条件的均可采用Spark技术进行处理，在实际应用中，目前大数据在互联网公司主要应用在广告、报表、推荐系统等业务上，在广告业务方面需要大数据做应用分析、效果分析、定向优化等，在推荐系统方面则需要大数据优化相关排名、个性化推荐以及热点点击分析等。

这些应用场景的普遍特点是计算量大、效率要求高，Spark恰恰可以满足这些要求，该项目一经推出便受到开源社区的广泛关注和好评，并在近两年内发展成为大数据处理领域炙手可热的开源项目。

Spark使用Scala语言进行实现，它是一种面向对象、函数式编程语言，能够像操作本地集合对象一样轻松地操作分布式数据集，具有运行速度快、易用性好、通用性强以及随处运行等特点，适合大多数批处理工作，并已成为大数据时代企业大数据处理优选技术，其中有代表性企业有腾讯、Yahoo、淘宝以及优酷土豆等。

Spark on standalone模型、YARN架构模型（画架构图）

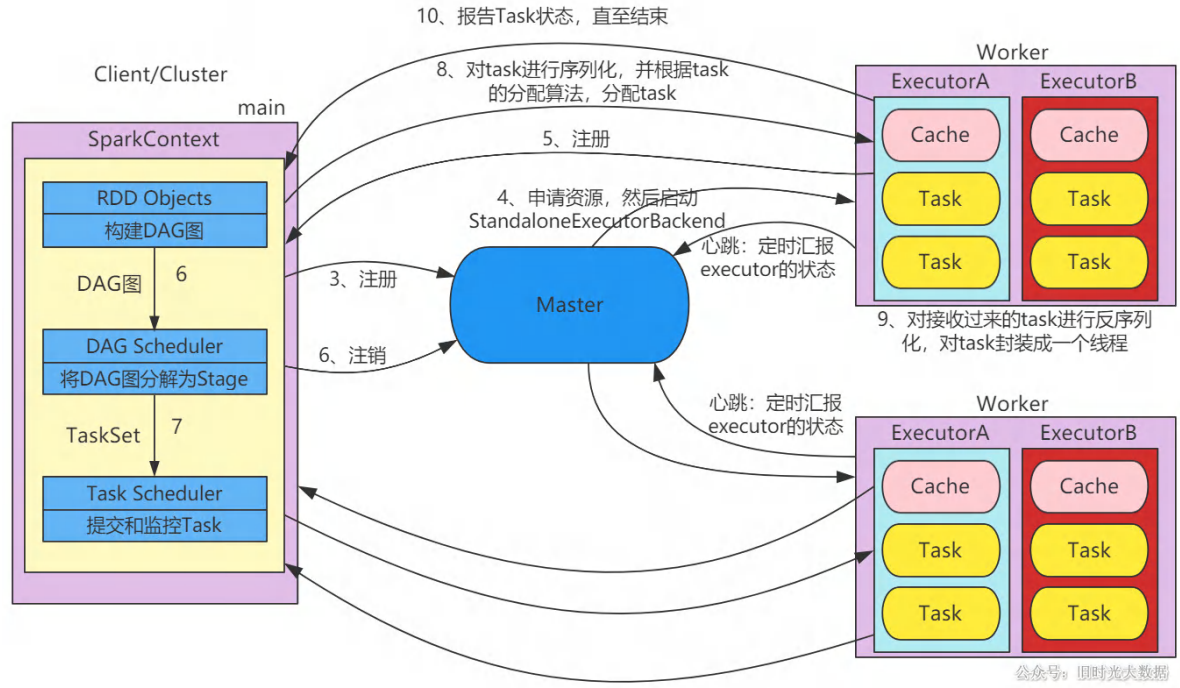
可回答：Spark client和cluster模式有什么区别

问过的一些公司：字节，字节(2021.10)

参考答案：

1、Spark on Standalone运行过程（架构）

Standalone模式是Spark实现的资源调度框架，其主要的节点有Client节点、Master节点和Worker节点。其中Driver既可以运行在Master节点上中，也可以运行在本地Client端。当用spark-shell交互式工具提交Spark的Job时，Driver在Master节点上运行；当使用spark-submit工具提交Job或者在Eclipse、IDEA等开发平台上使用 `new SparkConf().setMaster("spark://master:7077")` 方式运行Spark任务时，Driver是运行在本地Client端上的。



运行流程如下：

1）我们提交一个任务，任务就叫Application；

2）初始化程序的入口SparkContext；

- 初始化DAG Scheduler
- 初始化Task Scheduler

公众号：旧时光大数据

3）Task Scheduler向master去进行注册并申请资源（CPU Core和Memory）；

4）Master根据SparkContext的资源申请要求和Worker心跳周期内报告的信息决定在哪个Worker上分配资源，然后在该Worker上获取资源，然后启动StandaloneExecutorBackend；顺便初始化好了一个线程池；

5）StandaloneExecutorBackend向Driver(SparkContext)注册，这样Driver就知道哪些Executor为他进行服务了。到这个时候其实我们的初始化过程基本完成了，我们开始执行transformation的代码，但是代码并不会真正的运行，直到我们遇到一个action操作。生产一个job任务，进行stage的划分；

6）SparkContext将Applicaiton代码发送给StandaloneExecutorBackend；并且SparkContext解析Applicaiton代码，构建DAG图，并提交给DAG Scheduler分解成Stage（当碰到Action操作时，就会催生Job；每个Job中含有1个或多个Stage，Stage一般在获取外部数据和shuffle之前产生）；

7）将Stage（或者称为TaskSet）提交给Task Scheduler。Task Scheduler负责将Task分配到相应的Worker，最后提交给StandaloneExecutorBackend执行；

8）对task进行序列化，并根据task的分配算法，分配task；

9）executor对接收过来的task进行反序列化，把task封装成一个线程；

10）开始执行Task，并向SparkContext报告，直至Task完成；

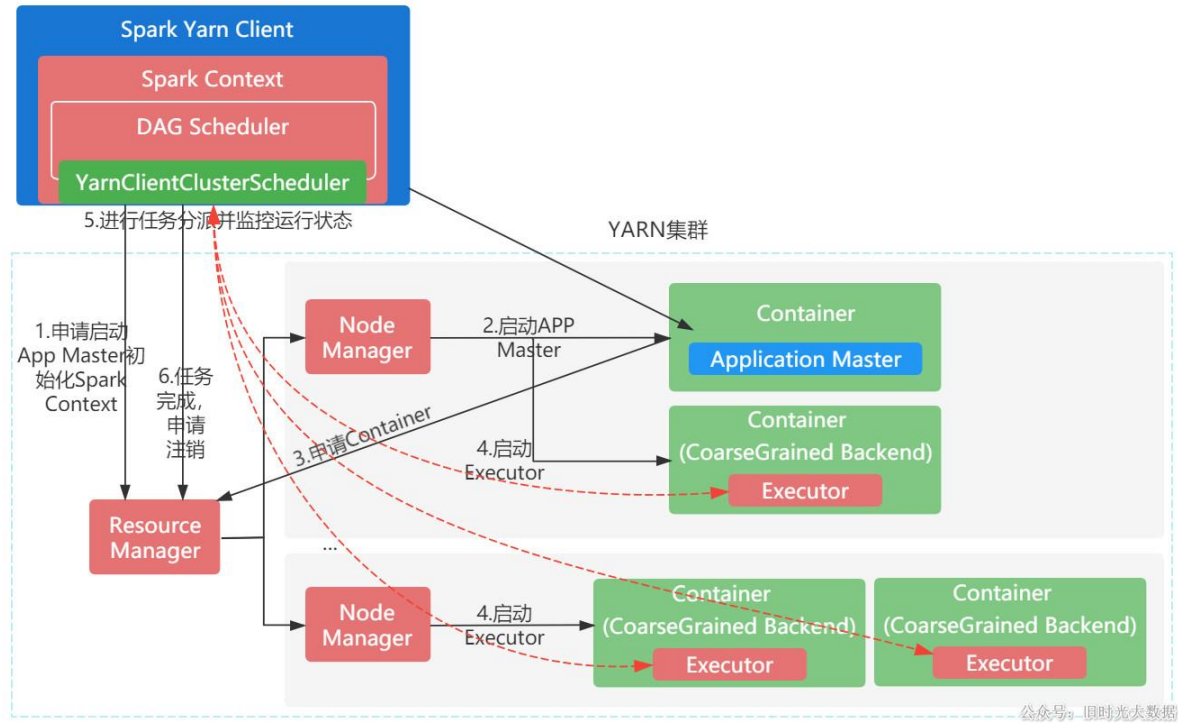
11）资源注销

2、Spark on YARN运行过程（架构）

Spark on YARN模式根据Driver在集群中的位置分为两种模式：一种是YARN-Client模式，另一种是YARN-Cluster（或称为YARN-Standalone模式）。

YARN-Client模式

Yarn-Client模式中，Driver在客户端本地运行，这种模式可以使得Spark Application和客户端进行交互，因为Driver在客户端，所以可以通过webUI访问Driver的状态，默认是 `http://hadoop1:4040` 访问，而YARN通过 `http:// hadoop1:8088` 访问。



YARN-client的工作流程分为以下几个步骤：

1）Spark Yarn Client向YARN的ResourceManager申请启动Application Master。同时在SparkContent初始化中将创建DAGScheduler和TASKScheduler等，由于我们选择的是Yarn-Client模式，程序会选择YarnClientClusterScheduler和YarnClientSchedulerBackend；

公众号：旧时光大数据

2）ResourceManager收到请求后，在集群中选择一个NodeManager，为该应用程序分配第一个Container，要求它在这个Container中启动应用程序的ApplicationMaster，与YARN-Cluster区别的是在该ApplicationMaster不运行SparkContext，只与SparkContext进行联系进行资源的分派；

3）Client中的SparkContext初始化完毕后，与ApplicationMaster建立通讯，向ResourceManager注册，根据任务信息向ResourceManager申请资源（Container）；

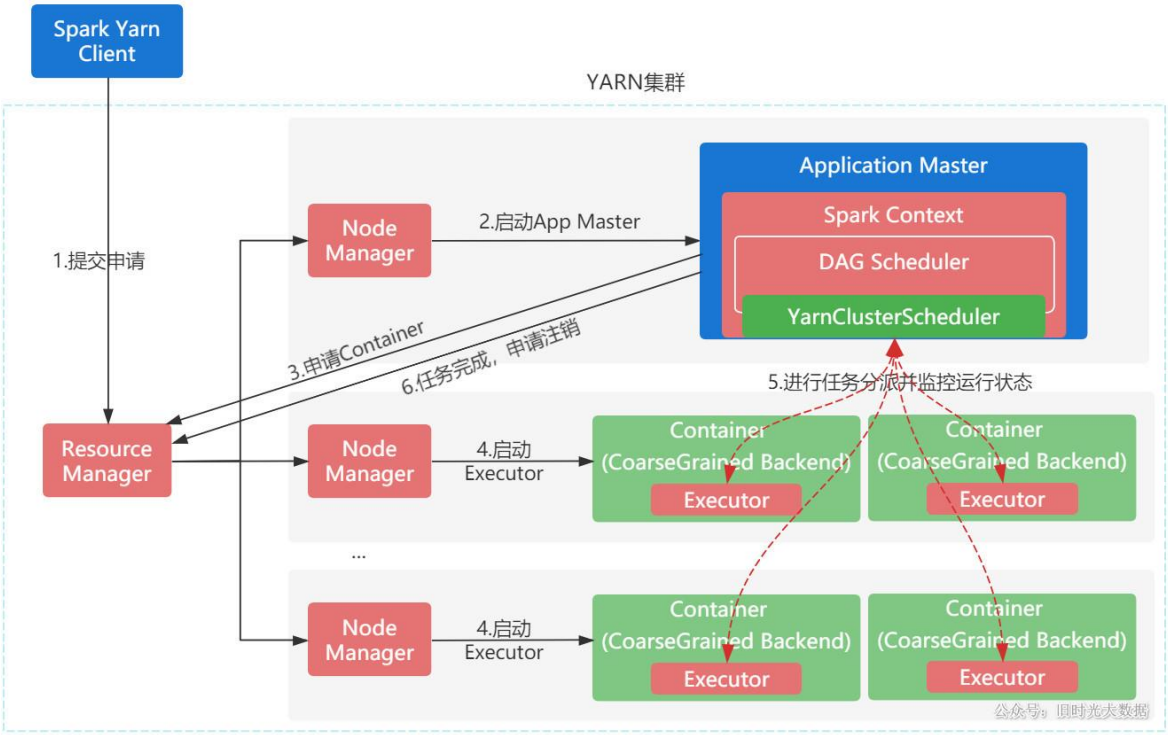
4）一旦ApplicationMaster申请到资源（也就是Container）后，便与对应的NodeManager通信，要求它在获得的Container中启动启动CoarseGrainedExecutorBackend，CoarseGrainedExecutorBackend启动后会向Client中的SparkContext注册并申请Task；

5）Client中的SparkContext分配Task给CoarseGrainedExecutorBackend执行，CoarseGrainedExecutorBackend运行Task并向Driver汇报运行的状态和进度，以让Client随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务；

6）应用程序运行完成后，Client的SparkContext向ResourceManager申请注销并关闭自己。

YARN-Cluster模式

在YARN-Cluster模式中，当用户向YARN中提交一个应用程序后，YARN将分两个阶段运行该应用程序：第一个阶段是把Spark的Driver作为一个ApplicationMaster在YARN集群中先启动；第二个阶段是由ApplicationMaster创建应用程序，然后为它向ResourceManager申请资源，并启动Executor来运行Task，同时监控它的整个运行过程，直到运行完成。



YARN-cluster的工作流程分为以下几个步骤：

1）Spark Yarn Client向YARN中提交应用程序，包括ApplicationMaster程序、启动ApplicationMaster的命令、需要在Executor中运行的程序等；

2）ResourceManager收到请求后，在集群中选择一个NodeManager，为该应用程序分配第一个Container，要求它在这个Container中启动应用程序的ApplicationMaster，其中ApplicationMaster进行SparkContext等的初始化；

3）ApplicationMaster向ResourceManager注册，这样用户可以直接通过ResourceManage查看应用程序的运行状态，然后它将采用轮询的方式通过RPC协议为各个任务申请资源，并监控它们的运行状态直到运行结束；

公众号：旧时光大数据

4）一旦ApplicationMaster申请到资源（也就是Container）后，便与对应的NodeManager通信，要求它在获得的Container中启动启动CoarseGrainedExecutorBackend，CoarseGrainedExecutorBackend启动后会向ApplicationMaster中的SparkContext注册并申请Task。这一点和Standalone模式一样，只不过SparkContext在Spark Application中初始化时，使用CoarseGrainedSchedulerBackend配合YarnClusterScheduler进行任务的调度，其中YarnClusterScheduler只是对TaskSchedulerImpl的一个简单包装，增加了对Executor的等待逻辑等；

5）ApplicationMaster中的SparkContext分配Task给CoarseGrainedExecutorBackend执行，CoarseGrainedExecutorBackend运行Task并向ApplicationMaster汇报运行的状态和进度，以让ApplicationMaster随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务；

6）应用程序运行完成后，ApplicationMaster向ResourceManager申请注销并关闭自己。

3、YARN-Client 与 YARN-Cluster 区别

理解YARN-Client和YARN-Cluster深层次的区别之前先清楚一个概念：Application Master。在YARN中，每个Application实例都有一个ApplicationMaster进程，它是Application启动的第一个容器。它负责和ResourceManager打交道并请求资源，获取资源之后告诉NodeManager为其启动Container。从深层次的含义讲YARN-Cluster和YARN-Client模式的区别其实就是ApplicationMaster进程的区别。

YARN-Cluster模式下，Driver运行在AM(Application Master)中，它负责向YARN申请资源，并监督作业的运行状况。当用户提交了作业之后，就可以关掉Client，作业会继续在YARN上运行，因而YARN-Cluster模式不适合运行交互类型的作业；

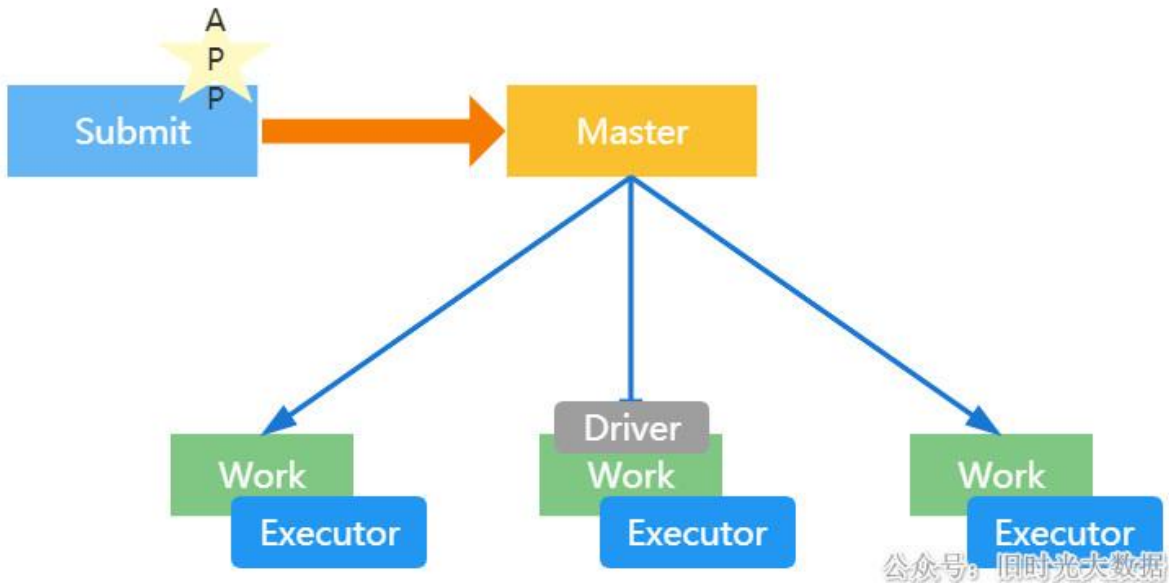
YARN-Client模式下，Application Master仅仅向YARN请求Executor，Client会和请求的Container通信来调度他们工作，也就是说Client不能离开。

Spark的yarn-cluster涉及的参数有哪些?

问过的一些公司：字节

参考答案：

Spark On Yarn的Cluster模式指的是Driver程序运行在Yarn集群上



公众号：旧时光大数据

```
--class org.apache.spark.examples.SparkPi \
--master yarn \
--deploy-mode cluster \
--driver-memory 1g \
--executor-memory 1g \
--executor-cores 2 \
--queue default \

```

参数	解释
--class	程序的main方法所在的类
--master	指定 Master 的地址
--deploy-mode	指定运行模式（client/cluster）
--driver-memory	Driver运行所需要的内存, 默认1g
--executor-memory	指定每个 executor 可用内存为 2g， 默认1g
--executor-cores	指定每一个 executor 可用的核数
--queue	指定任务的对列

Spark提交job的流程

可回答：Saprk的job调度、 Task调度

问过的一些公司：字节，快手，作业帮，蔚来(2021.09)x2，欢聚(2021.09)，腾讯(2021.10)

参考答案：

Job提交运行的总流程，大致分为两个阶段：

1、Stage划分与提交

- Job按照RDD之间的依赖关系是否为宽依赖，由DAGScheduler划分为一个个Stage，并将每个Stage提交给TaskScheduler；
- Stage随后被提交，并由TaskScheduler将每个stage转化为一个TaskSet。

2、Task调度与执行：由TaskScheduler负责将TaskSet中的Task调度到Worker节点的Executor上执行

对于**第一阶段Stage划分与提交**，又主要分为三个阶段：

1、Job的调度模型与运行反馈

1）首先由DAGScheduler负责将Job提交到事件队列eventProcessLoop中，等待调度执行

该事件队列为DAGSchedulerEventProcessLoop类型，内部封装了一个BlockingQueue阻塞队列，并由一个后台线程eventThread不断的调用onReceive()方法处理其中的事件

2）创建一个JobWaiter对象并返回给客户端

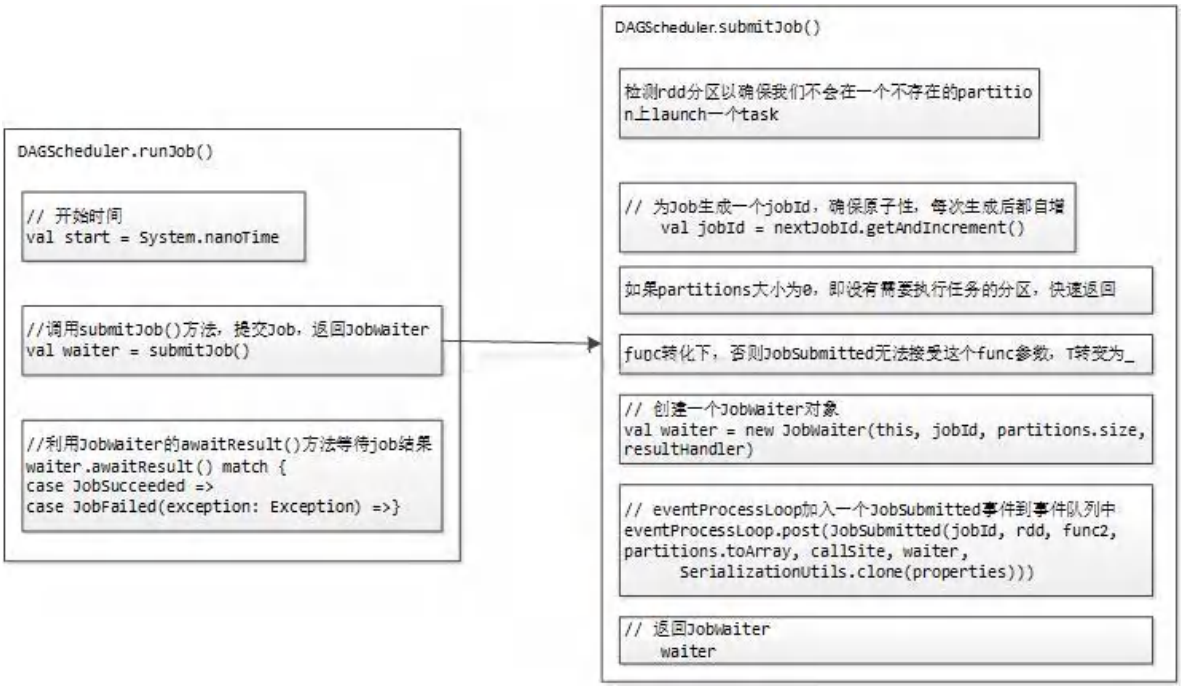
利用这个JobWaiter对象的awaitResult()方法对Job进行监控与运行反馈，并获得JobSucceeded和JobFailed两种Job运行结果

3）DAGSchedulerEventProcessLoop的onReceive()方法处理事件

公众号：旧时光大数据

onReceive()方法继续调用doOnReceive(event)方法，然后根据传入的事件类型DAGSchedulerEvent决定调用哪个方法处理事件，这里传入的是JobSubmitted事件，调用的是DAGScheduler的handleJobSubmitted()方法，继而进入下一个阶段。

整个处理流程如下图所示：



2、Stage划分

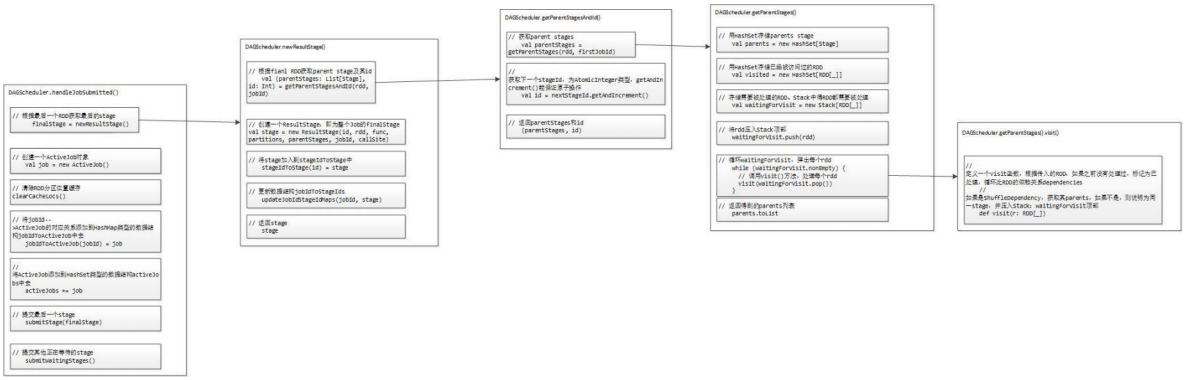
在第一阶段将JobSubmitted事件提交到事件队列后，DAGScheduler的handleJobSubmitted()方法就开始了Stage的划分。

1）根据finalRDD获取其Parent Stages，即ShuffleMapStage列表

2）利用finalRDD生成最后一个Stage，即ResultStage

3）生成ActiveJob对象，并维护各种stage、job等数据结构

整个处理流程如下图所示：



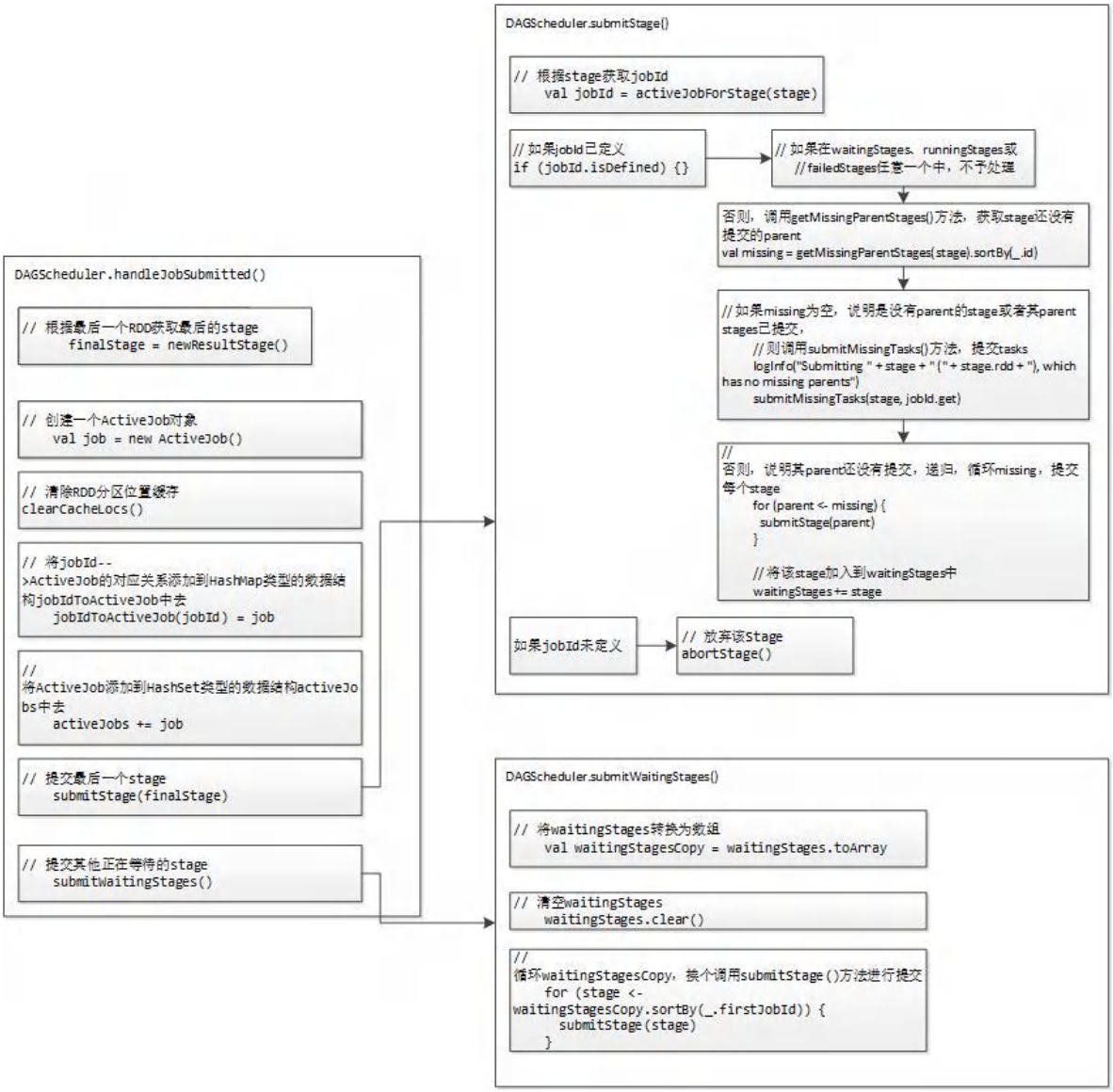
3、Stage提交：对应TaskSet的生成

1）提交finalStage

2）提交其parent Stage，如果对应parent Stage还存在尚未提交的parent Stage，提交之

3）对于没有parent Stage的Stage，根据stage中rdd的分区，生成tasks，即TaskSet，创建TaskSetManager，并由SchedulerBackend申请资源

整个处理流程如下图所示：



Spark的阶段划分

问过的一些公司：京东

参考答案：

这里说下Driver的工作流程，Driver线程主要是初始化SparkContext对象，准备运行所需的上下文，然后一方面保持与ApplicationMaster的RPC连接，通过ApplicationMaster申请资源，另一方面根据用户业务逻辑开始调度任务，将任务下发到已有的空闲Executor上。

当ResourceManager向ApplicationMaster返回Container资源时，ApplicationMaster就尝试在对应的Container上启动Executor进程，Executor进程起来后，会向Driver反向注册，注册成功后保持与Driver的心跳，同时等待Driver分发任务，当分发的任务执行完毕后，将任务状态上报给Driver。

当Driver起来后，Driver则会根据用户程序逻辑准备任务，并根据Executor资源情况逐步分发任务。

一个RDD任务分为：Application、Job、Stage 和 Task。

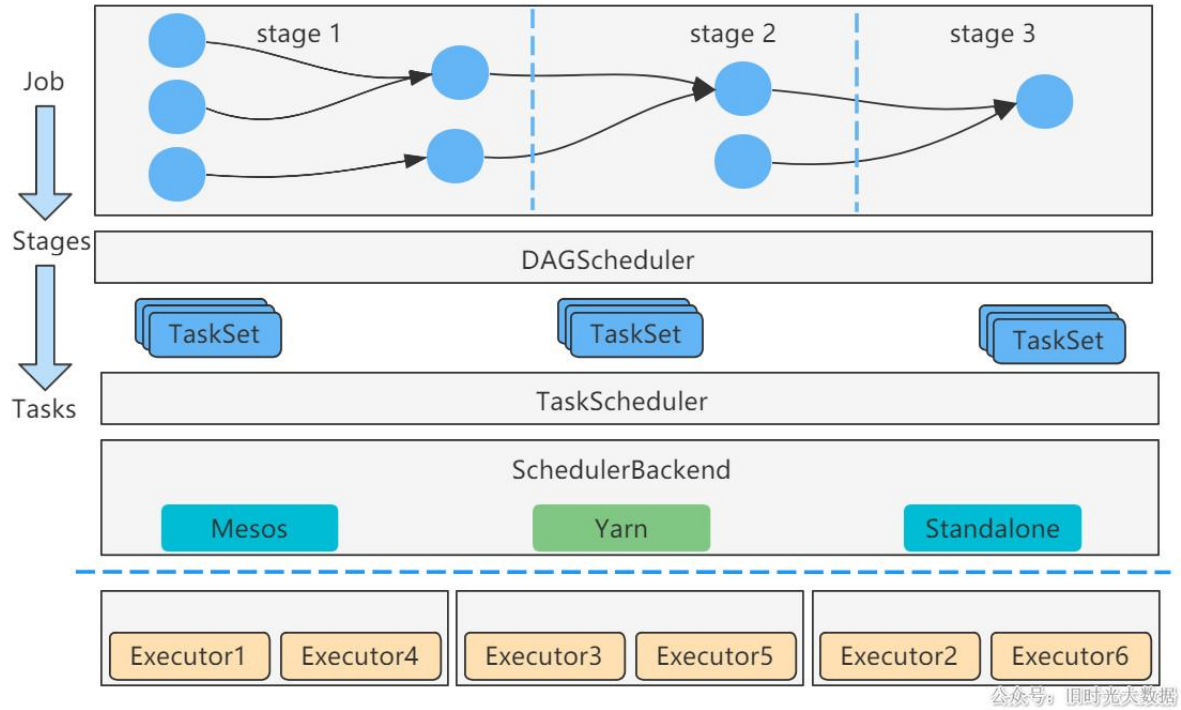
Application：初始化一个 SparkContext 就生成一个 Application；

Job：以Action方法为界，遇到一个Action方法则触发一个Job；

Stage：Job的子集，以RDD宽依赖(即Shuffle)为界，遇到Shuffle做一次划分；

Task：Stage的子集，以并行度(分区数)来衡量，分区数是多少，则有多少个task。

公众号：旧时光大数据



Spark RDD通过其Transactions操作，形成了RDD血缘（依赖）关系图，即DAG，最后通过Action的调用，触发Job并调度执行，执行过程中会创建两个调度器：DAGScheduler和TaskScheduler。

- DAGScheduler负责Stage级的调度，主要是将job切分成若干Stages，并将每个Stage打包成TaskSet交给TaskScheduler调度。
- TaskScheduler负责Task级的调度，将DAGScheduler给过来的TaskSet按照指定的调度策略分发到Executor上执行，调度过程中SchedulerBackend负责提供可用资源，其中SchedulerBackend有多种实现，分别对接不同的资源管理系统。

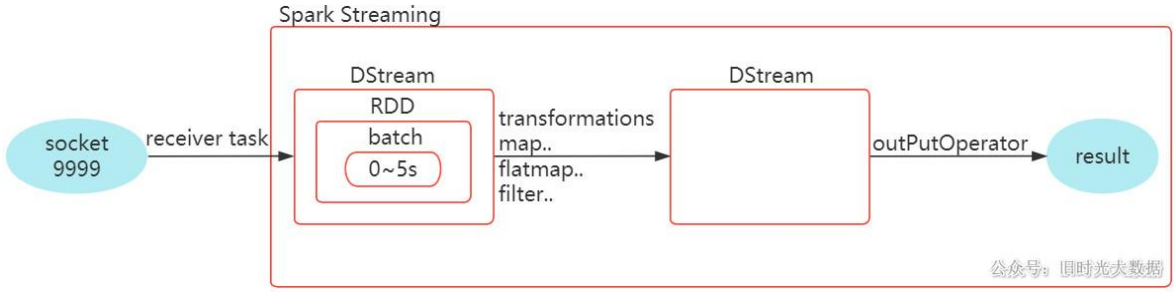
Driver初始化SparkContext过程中，会分别初始化DAGScheduler、TaskScheduler、SchedulerBackend以及HeartbeatReceiver，并启动SchedulerBackend以及HeartbeatReceiver。SchedulerBackend通过ApplicationMaster申请资源，并不断从TaskScheduler中拿到合适的Task分发到Executor执行。HeartbeatReceiver负责接收Executor的心跳信息，监控Executor的存活状况，并通知到TaskScheduler。

Spark处理数据的具体流程

问过的一些公司：趋势科技

参考答案：

Spark Streaming处理数据流程



Spark Streaming是Spark中处理流式数据的模块

Spark Streaming启动之后，启动一个job一直接收数据，将接收来的数据每隔一段时间封装到一个batch中，这里的一段时间就是batchInterval。batch又被封装到RDD中，RDD封装到DStream中，DStream有自己的算子对数据进行处理，Transformation类算子可以对DStream进行转换（懒执行），outputOperator类对DStream进行触发执行。

假设batchInterval=5s，集群处理一批次数据时间是3s：

0-5s：集群接收数据，5-8s：一边接收数据一边处理数据，8-10s：只是接收数据，10-13s...集群处理一批次有休息，不能充分利用集群的资源。

假设batchInterval=5s，集群处理一批次数据时间是8s：

0-5s：集群接收数据，5-10s：一边接收数据一边处理数据，10-13s：一边接收数据一边处理数据，13-15s：一边接收数据一边处理数据...集群处理数据，有任务的堆积，如果接收来的数据存放内存不够，要溢写磁盘，加大数据处理的延迟。

batchInterval生成一批次数据，集群处理一批次数据时间也是batchInterval，集群既不会资源不能充分利用，也不会有任务堆积。

说下Spark join的分类

可回答：1）Spark join过程？2）Spark的三种join

问过的一些公司：腾讯，网易云音乐，字节x2，字节(2021.10)，爱奇艺，有赞，京东(2021.07)

参考答案：

hash join是传统数据库中的单机join算法，在分布式环境下需要经过一定的分布式改造，说到底就是尽可能利用分布式计算资源进行并行化计算，提高总体效率。hash join分布式改造一般有两种经典方案：

broadcast hash join：将其中一张小表广播分发到另一张大表所在的分区节点上，分别并发地与其上的分区记录进行hash join。broadcast适用于小表很小，可以直接广播的场景。

shuffler hash join：一旦小表数据量较大，此时就不再适合进行广播分发。这种情况下，可以根据join key相同必然分区相同的原理，将两张表分别按照join key进行重新组织分区，这样就可以将join分而治之，划分为很多小join，充分利用集群资源并行化。

Spark join方式主要有以下三种：

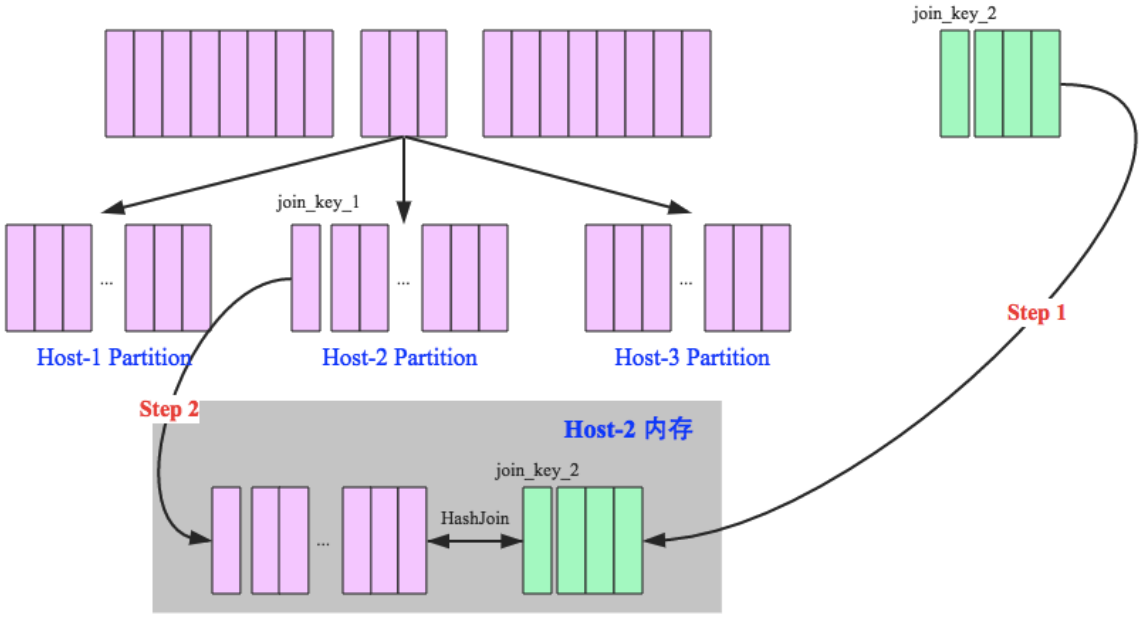
1、Broadcast Hash Join

broadcast hash join可以分为两步：

1）broadcast阶段：将小表广播分发到大表所在的所有主机。广播算法可以有很多，最简单的是先发给driver，driver再统一分发给所有executor；要不就是基于bittorrete的p2p思路；

2）hash join阶段：在每个executor上执行单机版hash join，小表映射，大表试探。

公众号：旧时光大数据



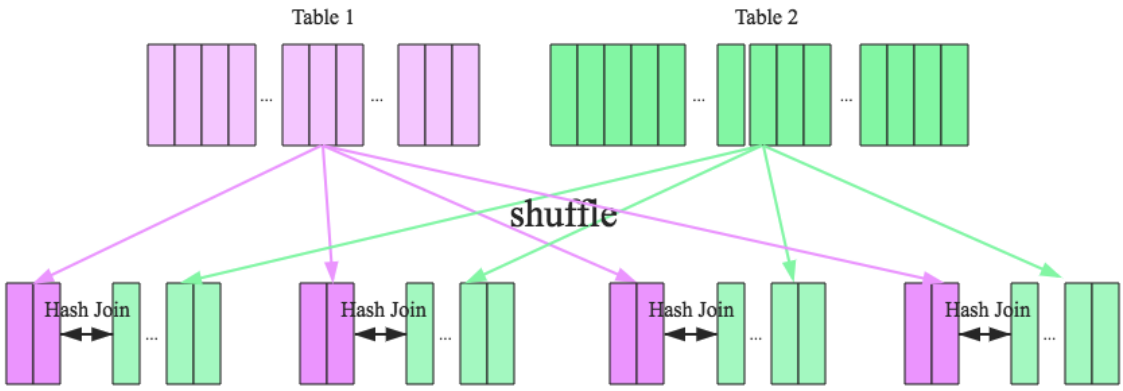
适用于小表与大表的join，SparkSQL规定broadcast hash join执行的基本条件为被广播小表必须小于参数spark.sql.autoBroadcastJoinThreshold，默认为10M，本质上是去用空间换时间，

2、Shuffle Hash Join

在大数据条件下如果一张表很小，执行join操作最优的选择无疑是broadcast hash join，效率最高。但是一旦小表数据量增大，广播所需内存、带宽等资源必然就会太大，broadcast hash join就不再是最优方案。此时可以按照join key进行分区，根据key相同必然分区相同的原理，就可以将大表join分而治之，划分为很多小表的join，充分利用集群资源并行化。如下图所示，shuffle hash join也可以分为两步：

1）shuffle阶段：分别将两个表按照join key进行分区，将相同join key的记录重分布到同一节点，两张表的数据会被重分布到集群中所有节点。这个过程称为shuffle

2）hash join阶段：每个分区节点上的数据单独执行单机hash join算法。

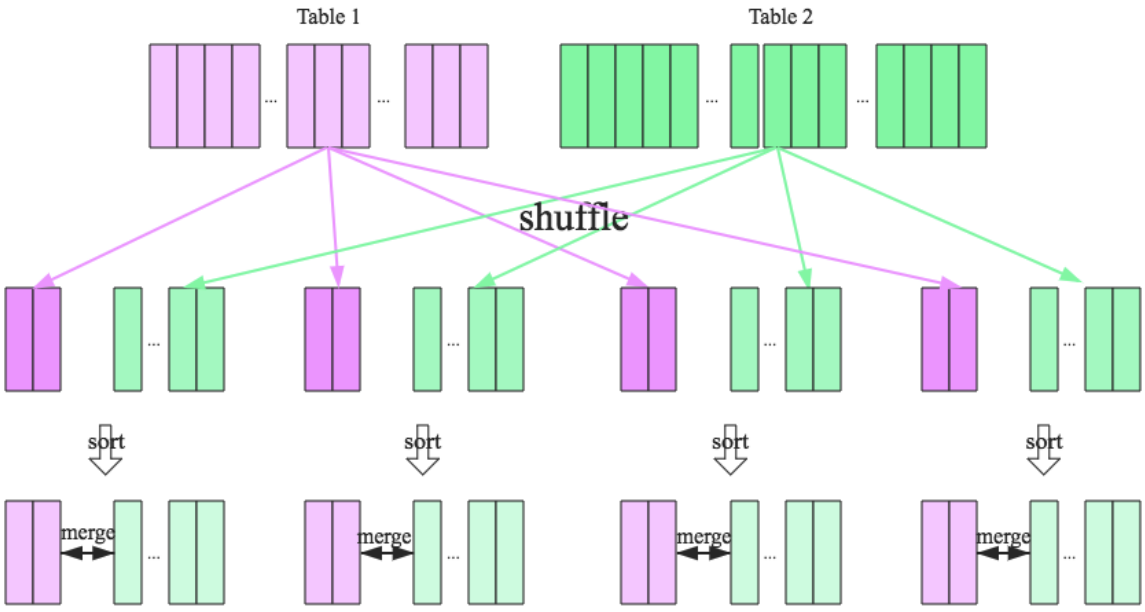


适合于没有特别小的两个表进行关联的时候，默认设置的shuffle partition的个数是200，也就是分了200个区，然后两张表的key值分别去基于200做hash取余然后散步在每个区域中了，这样的思想先把相近的合并在一个区内，再在每个分区内去做比较key值的等值比较，就避免了大范围的遍历比较，节省了时间和内存。

3、Sort-Merge Join

SparkSQL对两张大表join采用了全新的算法：sort-merge join，如下图所示，整个过程分为三个步骤：

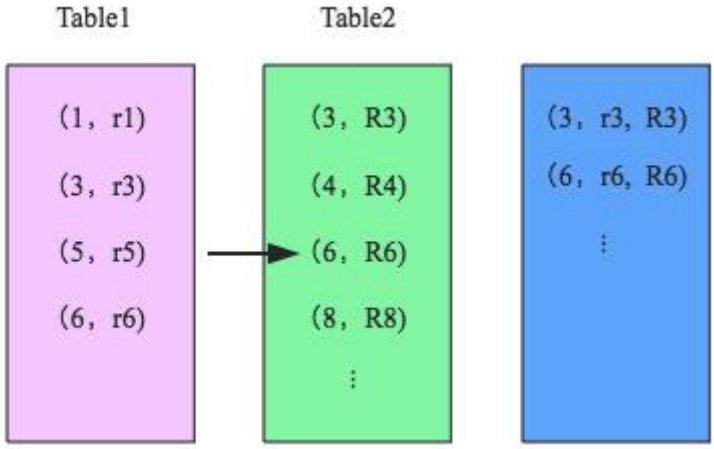
公众号：旧时光大数据



1）shuffle阶段：将两张大表根据join key进行重新分区，两张表数据会分布到整个集群，以便分布式并行处理；

2）sort阶段：对单个分区节点的两表数据，分别进行排序；

3）merge阶段：对排好序的两张分区表数据执行join操作。join操作很简单，分别遍历两个有序序列，碰到相同join key就merge输出，否则取更小一边，见下图示意：



这种**适用于关联的两张表都特别大时**，使用上述的两种方法加载到内存的时候对于内存的压力都非常大时，因此在2方法的基础上，hash取余之后还要分别对两张表的key值进行排序，这样去做等值比较的时候就不需要将某一方的全部数据都加载到内存进行计算了，只需要取一部分就能知道是否有相等的（比如按升序排列，某个值明显比它大了，后面肯定就不会有相等的，就不用继续比较了，节省了时间和内存），也就是在进行等值比较的时候即用即丢的。这个方法在前面进行排序的时候可能会消耗点时间，但相对于后面的时间来说，总体是大大节省了时间。

Spark map join的实现原理

问过的一些公司：爱奇艺，网易云音乐

参考答案：

Map-Join适用于有一份数据较小的连接情况。做法是把该小份数据直接全部加载到内存当中，按Join关键字建立索引。然后大份数据就作为MapTask的输入，对map()方法的每次输入都去内存当中直接进行匹配连接。然后把连接结果按key输出。这种方法要使用Spark中的广播(Broadcast)功能把小份数据分放到各个计算节点，每个maptask执行任务的节点都需要加载该数据到内存。由于Join是在Map阶段进行的，

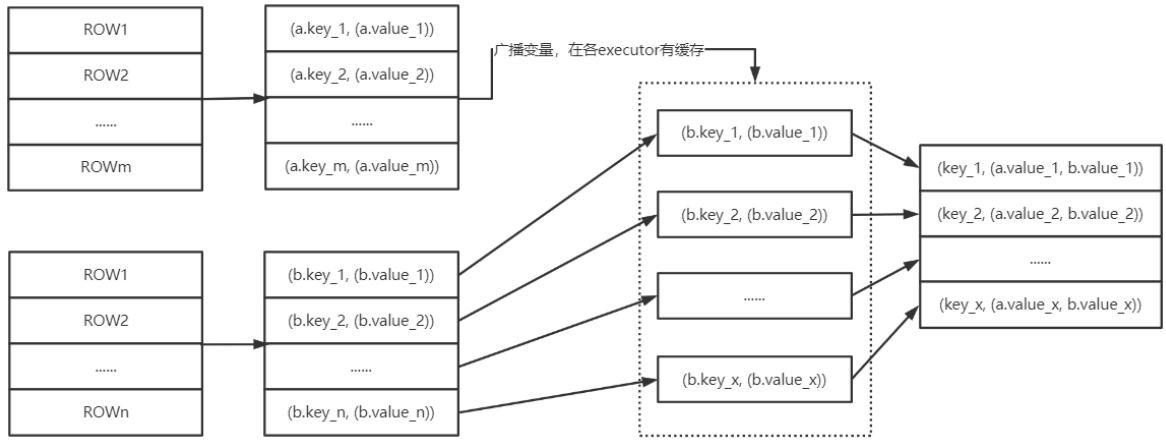
公众号：旧时光大数据

故称为Map-Join 。

缺点如下：

需要将小表建立索引，常用方式是建立Map表，在 Spark 中，可以通过 rdd.collectAsMap() 算子实现。但是 collectAsMap() 在 key 重复时，后面的 value 会覆盖前面的，所以对于存在重复 key 的表，需做其他处理。另外，在 collectAsMap() 过程中，由于需要在 Driver 节点进行 collect ，所以需要保证 Driver 节点内存充足，可在 spark-commit 提交执行任务时，通过设置 driver-memory 调节 Driver 节点大小。

Map-Join 同时需要将经过 Map 广播到不同的 executor ,供 Task 获取数据进行连接并输出结果。所以 executor 也需要保证内存充足，可在 spark-commit 提交执行任务时，通过设置 --executor-cores 调节 Driver 节点大小。



介绍下Spark Shuffle及其优缺点

可回答：1）Spark的Shuffle机制，如何划分Shuffle？2）Spark的shuffle过程

问过的一些公司：字节 x 7，字节(2021.08)-(2021.10)，爱奇艺，头条，第四范式，陌陌 x 3，蘑菇街，美团 x 6，美团(2021.08)，海康，网易云音乐，阿里，商汤科技，作业帮社招，京东，携程(2021.09)，腾讯云(2021.10)

参考答案：

Spark Shuffle 分为两种：一种是基于 Hash 的 Shuffle；另一种是基于 Sort 的 Shuffle。

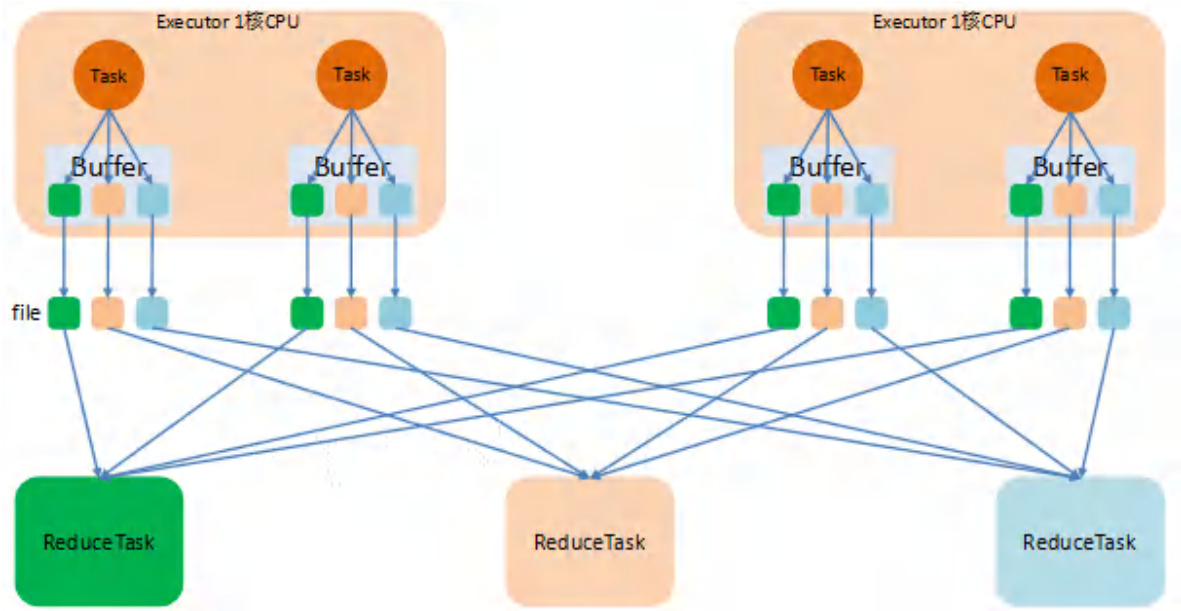
对于下面的内容，先明确一个假设前提：每个Executor只有1个CPU core，也就是说，无论这个Executor上分配多少个task线程，同一时间都只能执行一个task线程。

1、Hash Shuffle

1）未优化的HashShuffle

如下图中有3个 Reducer，从Task 开始那边各自把自己进行 Hash 计算(分区器：hash/numreduce取模)，分类出3个不同的类别，每个 Task 都分成3种类别的数据，想把不同的数据汇聚然后计算出最终的结果，所以Reducer 会在每个 Task 中把属于自己类别的数据收集过来，汇聚成一个同类别的大集合，每1个 Task 输出3份本地文件，这里有4个 Mapper Tasks，所以总共输出了4个 Tasks x 3个分类文件 = 12个本地小文件。

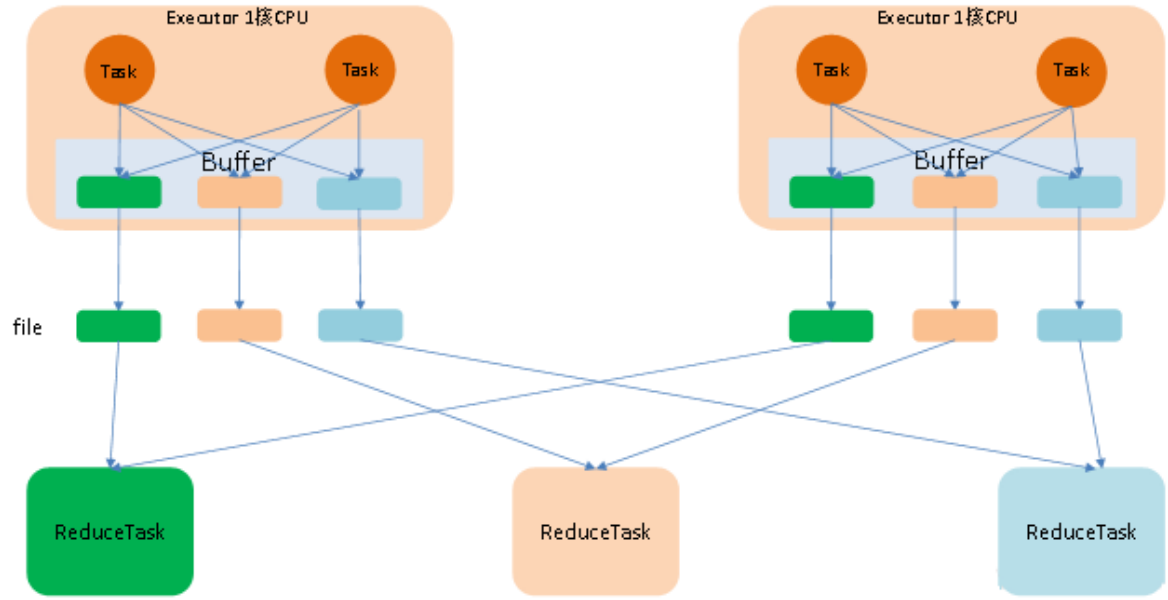
公众号：旧时光大数据



2）优化后的HashShuffle

优化的HashShuffle过程就是**启用合并机制**，合并机制就是复用buffer，开启合并机制的配置是`spark.shuffle.consolidateFiles`。该参数默认值为false，将其设置为true即可开启优化机制。通常来说，如果我们使用HashShuffleManager，那么都建议开启这个选项。

这里还是有4个Tasks，数据类别还是分成3种类型，因为Hash算法会根据你的 Key 进行分类，在同一个进程中，无论是有多少过Task，都会把同样的Key放在同一个Buffer里，然后把Buffer中的数据写入以Core数量为单位的本地文件中，(一个Core只有一种类型的Key的数据)，每1个Task所在的进程中，分别写入共同进程中的3份本地文件，这里有4个Mapper Tasks，所以总共输出是 2个Cores x 3个分类文件 = 6个本地小文件。



基于 Hash 的 Shuffle 机制的优缺点

优点：

- 可以省略不必要的排序开销。
- 避免了排序所需的内存开销。

缺点：

- 生产的文件过多，会对文件系统造成压力。
- 大量小文件的随机读写带来一定的磁盘开销。
- 数据块写入时所需的缓存空间也会随之增加，对内存造成压力。

公众号：旧时光大数据

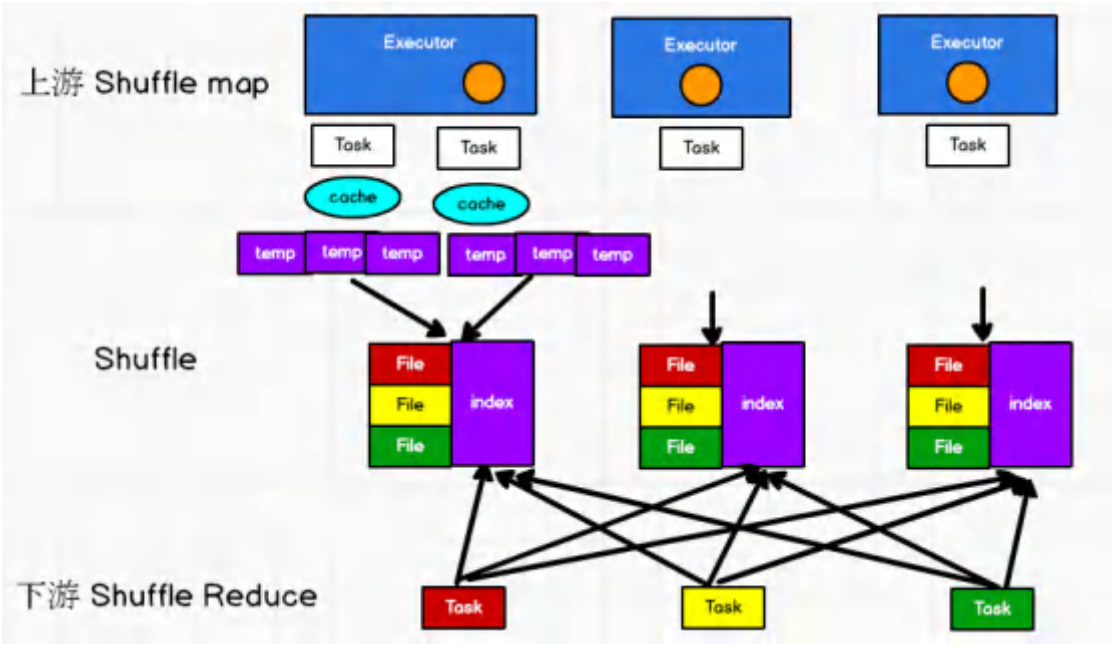
2、Sort Shuffle

1）普通SortShuffle

在该模式下，数据会先写入一个数据结构，reduceByKey写入Map，一边通过Map局部聚合，一遍写入内存。Join算子写入ArrayList直接写入内存中。然后需要判断是否达到阈值，如果达到就会将内存数据结构的数据写入到磁盘，清空内存数据结构。

在溢写磁盘前，先根据key进行排序，排序过后的数据，会分批写入到磁盘文件中。默认批次为10000条，数据会以每批一万条写入到磁盘文件。写入磁盘文件通过缓冲区溢写的方式，每次溢写都会产生一个磁盘文件，也就是说一个Task过程会产生多个临时文件。

最后在每个Task中，将所有的临时文件合并，这就是merge过程，此过程将所有临时文件读取出来，一次写入到最终文件。意味着一个Task的所有数据都在这一个文件中。同时单独写一份索引文件，标识下游各个Task的数据在文件中的索引，start offset和end offset。



2）bypass SortShuffle

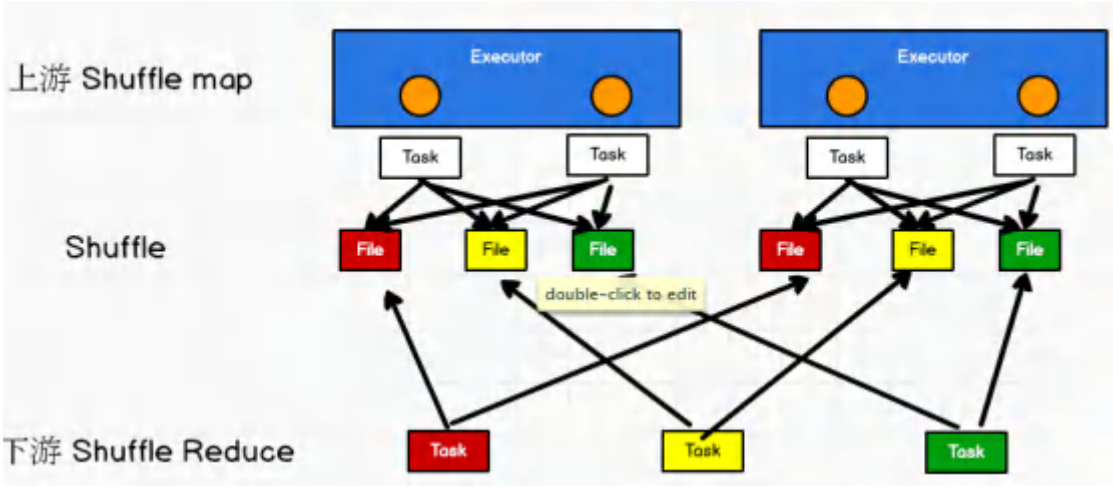
bypass运行机制的触发条件如下：

- shuffle reduce task数量小于spark.shuffle.sort.bypassMergeThreshold参数的值，默认为200
- 不是聚合类的shuffle算子（比如reduceByKey）

此时task会为每个reduce端的task都创建一个临时磁盘文件，并将数据按key进行hash然后根据key的hash值，将key写入对应的磁盘文件之中。当然，写入磁盘文件时也是先写入内存缓冲，缓冲写满之后再溢写到磁盘文件的。最后，同样会将所有临时磁盘文件都合并成一个磁盘文件，并创建一个单独的索引文件。

该过程的磁盘写机制其实跟未经优化的HashShuffleManager是一模一样的，因为都要创建数量惊人的磁盘文件，只是在最后会做一个磁盘文件的合并而已。因此少量的最终磁盘文件，也让该机制相对未经优化的HashShuffleManager来说，shuffle read的性能会更好。

而该机制与普通SortShuffleManager运行机制的不同在于：不会进行排序。也就是说，启用该机制的最大好处在于，shuffle write过程中，不需要进行数据的排序操作，也就节省掉了这部分的性能开销。



3）Tungsten Sort Shuffle 运行机制

基于 Tungsten Sort 的 Shuffle 实现机制主要是借助 Tungsten 项目所做的优化来高效处理 Shuffle。

Spark 提供了配置属性，用于选择具体的 Shuffle 实现机制，但需要说明的是，虽然默认情况下 Spark 默认开启的是基于 SortShuffle 实现机制，但实际上，参考 Shuffle 的框架内核部分可知基于 SortShuffle 的实现机制与基于 Tungsten Sort Shuffle 实现机制都是使用 SortShuffleManager，而内部使用的具体的实现机制，是通过提供的两个方法进行判断的：

对应非基于 Tungsten Sort 时，通过 SortShuffleWriter.shouldBypassMergeSort方法判断是否需要回退到 Hash 风格的 Shuffle 实现机制，当该方法返回的条件不满足时，则通过 SortShuffleManager.canUseSerializedShuffle方法判断是否需要采用基于 Tungsten Sort Shuffle 实现机制，而当这两个方法返回都为 false，即都不满足对应的条件时，会自动采用普通运行机制。

因此，当设置了spark.shuffle.manager=tungsten-sort时，也不能保证就一定采用基于 Tungsten Sort 的 Shuffle 实现机制。

要实现 Tungsten Sort Shuffle 机制需要满足以下条件：

- Shuffle 依赖中不带聚合操作或没有对输出进行排序的要求。
- Shuffle 的序列化器支持序列化值的重定位（当前仅支持 KryoSerializer Spark SQL 框架自定义的序列化器）。
- Shuffle 过程中的输出分区个数少于 16777216 个。

实际上，使用过程中还有其他一些限制，如引入 Page 形式的内存管理模型后，内部单条记录的长度不能超过 128 MB （具体内存模型可以参考 PackedRecordPointer 类）。另外，分区个数的限制也是该内存模型导致的。

所以，目前使用基于 Tungsten Sort Shuffle 实现机制条件还是比较苛刻的。

基于 Sort 的 Shuffle 机制的优缺点

优点：

- 小文件的数量大量减少，Mapper 端的内存占用变少；
- Spark 不仅可以处理小规模的数据，即使处理大规模的数据，也不会很容易达到性能瓶颈。

缺点：

- 如果 Mapper 中 Task 的数量过大，依旧会产生很多小文件，此时在 Shuffle 传数据的过程中到 Reducer 端， Reducer 会需要同时大量地记录进行反序列化，导致大量内存消耗和 GC 负担巨大，造成系统缓慢，甚至崩溃；
- 强制了在 Mapper 端必须要排序，即使数据本身并不需要排序；
- 它要基于记录本身进行排序，这就是 Sort-Based Shuffle 最致命的性能消耗。

公众号：旧时光大数据

什么情况下会产生Spark Shuffle?

问过的一些公司：妙盈科技，腾讯

参考答案：

spark中会导致shuffle操作的有以下几种算子：

- repartition类的操作：比如repartition、repartitionAndSortWithinPartitions、coalesce等
- byKey类的操作：比如reduceByKey、groupByKey、sortByKey等
- join类的操作：比如join、cogroup等

重分区: 一般会shuffle，因为需要在整个集群中，对之前所有的分区的数据进行随机，均匀的打乱，然后把数据放入下游新的指定数量的分区内。

byKey类的操作：因为你要对一个key，进行聚合操作，那么肯定要保证集群中，所有节点上的，相同的key，一定是到同一个节点上进行处理。

join类的操作：两个rdd进行join，就必须将相同join key的数据，shuffle到同一个节点上，然后进行相同key的两个rdd数据的笛卡尔乘积。

为什么要Spark Shuffle?

问过的一些公司：蘑菇街

参考答案：

Shuffle就是对数据进行重组，由于分布式计算的特性和要求，在实现细节上更加繁琐和复杂。

在DAG调度的过程中，**Stage阶段的划分是根据是否有shuffle过程**，也就是存在ShuffleDependency宽依赖的时候，需要进行shuffle，这时候会将作业job划分成多个Stage；并且在划分Stage的时候，构建ShuffleDependency的时候进行shuffle注册，获取后续数据读取所需要的ShuffleHandle，最终每一个job提交后都会生成一个ResultStage和若干个ShuffleMapStage，其中ResultStage表示生成作业的最终结果所在的Stage。ResultStage与ShuffleMapStage中的task分别对应着ResultTask与ShuffleMapTask。

RDD 的 Transformation 函数中，又分为窄依赖(narrow dependency)和宽依赖(wide dependency)的操作。窄依赖跟宽依赖的区别是是否发生shuffle(洗牌) 操作。宽依赖会发生shuffle操作。窄依赖是子RDD的各个分片(partition)不依赖于其他分片，能够独立计算得到结果，宽依赖指子RDD的各个分片会依赖于父RDD的多个分片，所以会造成父 RDD 的各个分片在集群中重新分片。

Spark为什么快?

问过的一些公司：字节，字节(2021.07)，华为精英计划(2021.07)

参考答案：

1）基于内存计算，减少低效的磁盘交互

2）基于DAG，高效的调度算法

3）容错机制Linage

4）缓存方式：persist、cache

Spark为什么适合迭代处理?

问过的一些公司：阿里零售供应链

参考答案：

先来看下**什么是迭代**：

迭代是重复反馈过程的活动，其目的通常是为了逼近所需目标或结果。每一次对过程的重复称为一次“迭代”，而每一次迭代得到的结果会作为下一次迭代的初始值。

重复执行一系列运算步骤，从前面的量依次求出后面的量的过程。此过程的每一次结果，都是由对前一次所得结果施行相同的运算步骤得到的。例如利用迭代法求某一数学问题的解。

对计算机特定程序中需要反复执行的子程序(一组指令)，进行一次重复，即重复执行程序中的循环，直到满足某条件为止，亦称为迭代。

Spark为什么适合迭代处理：

Spark迭代运算，采用内存存储中间计算结果，减少了迭代运算的磁盘IO，并通过并行计算DAG图的优化，减少不同任务之间的依赖，降低延迟等待时间。

相比于MapReduce，Spark在处理数据方面，可以一步接一步直接处理，而MapReduce每一个阶段都要将数据存入磁盘中，相比而言，Spark会更有优势，也更适合迭代处理。

Spark数据倾斜问题，如何定位，解决方案

可回答：Spark的OOM问题怎么产生的以及解决方案

问过的一些公司：字节跳动 x 7，字节(2021.07)，阿里，百度，百度(2021.09)，顺丰，腾讯，网易云音乐 x 2，小米，小米(2021.08)，安恒信息， 今日头条， 祖龙娱乐，商汤科技，阿里云，米哈游，快手，百度社招，触宝，多益，贝壳，ebay x 2，京东，嘉云数据，端点数据(2021.07)，Shopee(2021.07)，58同城(2021.08)，网易严选(2021.09)，美团(2021.09)，快手(2021.09)，蔚来(2021.09)x2，唯品会(2021.10)，360(2021.10)，阿里(2021.10)

新版数据倾斜及其解决方案

1、数据倾斜

Spark中的数据倾斜问题主要指shuffle过程中出现的数据倾斜问题，是由于不同的key对应的数据量不同导致的不同task所处理的数据量不同的问题

例如，reduce点一共要处理100万条数据，第一个和第二个task分别被分配到了1万条数据，计算5分钟内完成，第三个task分配到了98万数据，此时第三个task可能需要10个小时完成，这使得整个Spark作业需要10个小时才能运行完成，这就是数据倾斜所带来的后果

数据倾斜俩大直接致命后果

1）数据倾斜直接会导致一种情况：Out Of Memory

2）运行速度慢

注意，要区分开数据倾斜与数据量过量这两种情况，数据倾斜是指少数task被分配了绝大多数的数据，因此少数task运行缓慢；数据过量是指所有task被分配的数据量都很大，相差不多，所有task都运行缓慢

数据倾斜的表现：

1）Spark作业的大部分task都执行迅速，只有有限的几个task执行的非常慢，此时可能出现了数据倾斜，作业可以运行，但是运行得非常慢

2）Spark作业的大部分task都执行迅速，但是有的task在运行过程中会突然报出OOM，反复执行几次都在某一个task报出OOM错误，此时可能出现了数据倾斜，作业无法正常运行

公众号：旧时光大数据

定位数据倾斜问题：

1）查阅代码中的shuffle算子，例如reduceByKey、countByKey、groupByKey、join等算子，根据代码逻辑判断此处是否会出现数据倾斜

2）查看Spark作业的log文件，log文件对于错误的记录会精确到代码的某一行，可以根据异常定位到的代码位置来明确错误发生在第几个stage，对应的shuffle算子是哪一个

2、解决方案一：聚合元数据

1）避免shuffle过程

绝大多数情况下，Spark作业的数据来源都是Hive表，这些Hive表基本都是经过ETL之后的昨天的数据

为了避免数据倾斜，我们可以考虑避免shuffle过程，如果避免了shuffle过程，那么从根本上就消除了发生数据倾斜问题的可能

如果Spark作业的数据来源于Hive表，那么可以先在Hive表中对数据进行聚合，例如按照key进行分组，将同一key对应的所有value用一种特殊的格式拼接到一个字符串里去，这样，一个key就只有一条数据了；之后，对一个key的所有value进行处理时，只需要进行map操作即可，无需再进行任何的shuffle操作。通过上述方式就避免了执行shuffle操作，也就不可能会发生任何的数据倾斜问题。

对于Hive表中数据的操作，不一定是拼接成一个字符串，也可以是直接对key的每一条数据进行累计计算

要区分开，处理的数据量大和数据倾斜的区别

2）缩小key粒度（增大数据倾斜可能性，降低每个task的数据量）

key的数量增加，可能使数据倾斜更严重

3）增大key粒度（减小数据倾斜可能性，增大每个task的数据量）

如果没有办法对每个key聚合出来一条数据，在特定场景下，可以考虑扩大key的聚合粒度

例如，目前有10万条用户数据，当前key的粒度是（省，城市，区，日期），现在我们考虑扩大粒度，将key的粒度扩大为（省，城市，日期），这样的话，key的数量会减少，key之间的数据量差异也有可能会减少，由此可以减轻数据倾斜的现象和问题。（此方法只针对特定类型的数据有效，当应用场景不适宜时，会加重数据倾斜）

3、解决方案二：过滤导致倾斜的key

如果在Spark作业中允许丢弃某些数据，那么可以考虑将可能导致数据倾斜的key进行过滤，滤除可能导致数据倾斜的key对应的数据，这样，在Spark作业中就不会发生数据倾斜了

4、解决方案三：提高shuffle操作中的reduce并行度

当方案一和方案二对于数据倾斜的处理没有很好的效果时，可以考虑提高shuffle过程中的reduce端并行度，reduce端并行度的提高就增加了reduce端task的数量，那么每个task分配到的数据量就会相应减少，由此缓解数据倾斜问题

1）reduce端并行度的设置

在大部分的shuffle算子中，都可以传入一个并行度的设置参数，比如reduceByKey(500)，这个参数会决定shuffle过程中reduce端的并行度，在进行shuffle操作的时候，就会对应着创建指定数量的reduce task。对于Spark SQL中的shuffle类语句，比如group by、join等，需要设置一个参数，即spark.sql.shuffle.partitions，该参数代表了shuffle read task的并行度，该值默认是200，对于很多场景来说都有点过小

增加shuffle read task的数量，可以让原本分配给一个task的多个key分配给多个task，从而让每个task处理比原来更少的数据。举例来说，如果原本有5个key，每个key对应10条数据，这5个key都是分配给一个task的，那么这个task就要处理50条数据。而增加了shuffle read task以后，每个task就分配到一个key，即每个task就处理10条数据，那么自然每个task的执行时间都会变短了

公众号：旧时光大数据

2）reduce端并行度设置存在的缺陷

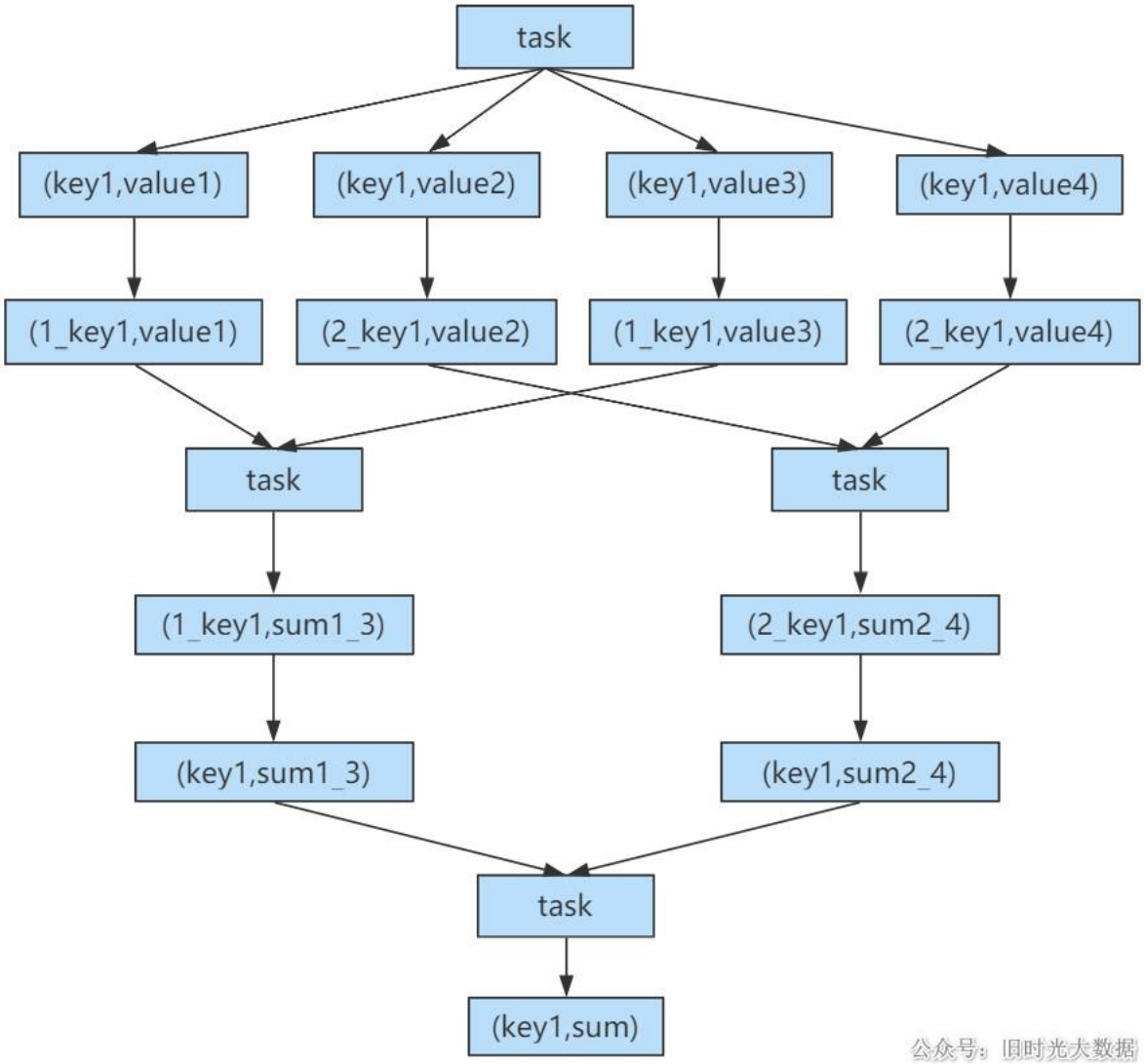
提高reduce端并行度并没有从根本上改变数据倾斜的本质和问题（方案一和方案二从根本上避免了数据倾斜的发生），只是尽可能地去缓解和减轻shuffle reduce task的数据压力，以及数据倾斜的问题，适用于有较多key对应的数据量都比较大的情况

该方案通常无法彻底解决数据倾斜，因为如果出现一些极端情况，比如某个key对应的数据量有100万，那么无论你的task数量增加到多少，这个对应着100万数据的key肯定还是会分配到一个task中去处理，因此注定还是会发生数据倾斜的。所以这种方案只能说是在发现数据倾斜时尝试使用的第一种手段，尝试去用嘴简单的方法缓解数据倾斜而已，或者是和其他方案结合起来使用

在理想情况下，reduce端并行度提升后，会在一定程度上减轻数据倾斜的问题，甚至基本消除数据倾斜；但是，在一些情况下，只会让原来由于数据倾斜而运行缓慢的task运行速度稍有提升，或者避免了某些task的OOM问题，但是，仍然运行缓慢，此时，要及时放弃方案三，开始尝试后面的方案

5、解决方案四：使用随机key实现双重聚合

当使用了类似于groupByKey、reduceByKey这样的算子时，可以考虑使用随机key实现双重聚合，如下图所示



首先，通过map算子给每个数据的key添加随机数前缀，对key进行打散，将原先一样的key变成不一样的key，然后进行第一次聚合，这样就可以让原本被一个task处理的数据分散到多个task上去做局部聚合；随后，去除掉每个key的前缀，再次进行聚合

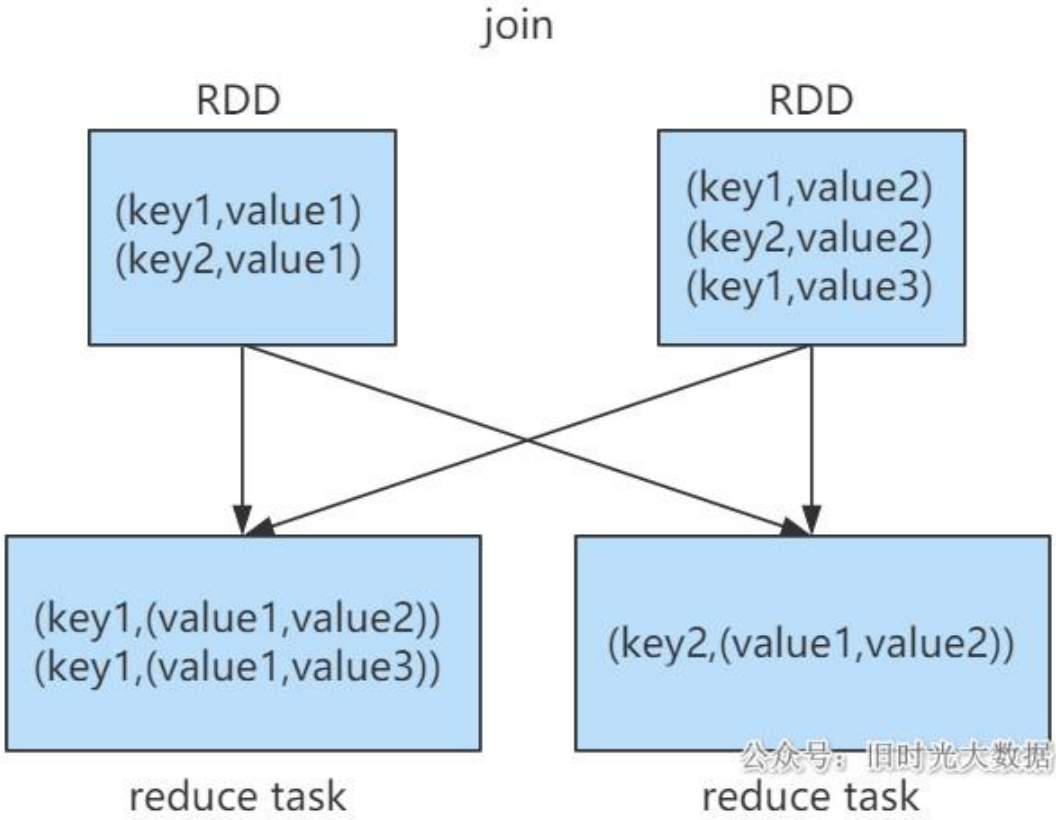
此方法对于由groupByKey、reduceByKey这类算子造成的数据倾斜由比较好的效果，仅仅适用于聚合类的shuffle操作，适用范围相对较窄。如果是join类的shuffle操作，还得用其他的解决方案

此方法也是前几种方案没有比较好的效果时要尝试的解决方案

公众号：旧时光大数据

6、解决方案五：将reduce join转换为map join

正常情况下，join操作都会执行shuffle过程，并且执行的是reduce join，也就是先将所有相同的key和对应的value汇聚到一个reduce task中，然后再进行join。普通join的过程如下图所示



普通的join是会走shuffle过程的，而一旦shuffle，就相当于会将相同key的数据拉取到一个shuffle read task中再进行join，此时就是reduce join。但是如果一个RDD是比较小的，则可以采用广播小RDD全量数据+map算子来实现与join同样的效果，也就是map join，此时就不会发生shuffle操作，也就不会发生数据倾斜

注意，RDD是并不能进行广播的，只能将RDD内部的数据通过collect拉取到Driver内存然后再进行广播

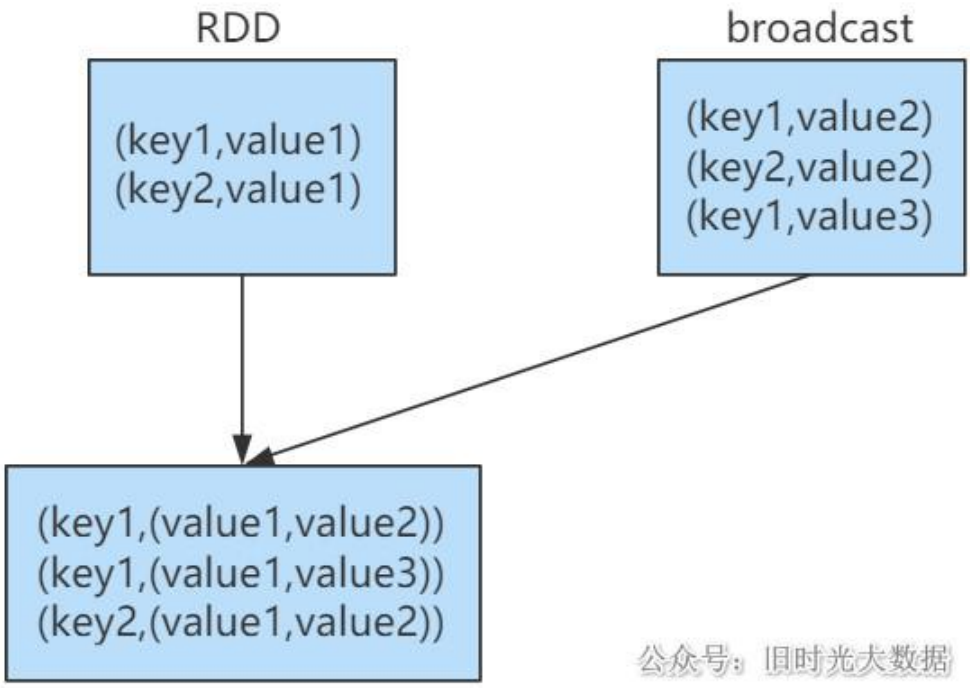
1）核心思路

不使用join算子进行连接操作，而使用Broadcast变量与map类算子实现join操作，进而完全规避掉shuffle类的操作，彻底避免数据倾斜的发生和出现。将较小RDD中的数据直接通过collect算子拉取到Driver端的内存中来，然后对其创建一个Broadcast变量；接着对另外一个RDD执行map类算子，在算子函数内，从Broadcast变量中获取较小RDD的全量数据，与当前RDD的每一条数据按照连接key进行比对，如果连接key相同的话，那么就将两个RDD的数据用你需要的方式连接起来

根据上述思路，根本不会发生shuffle操作，从根本上杜绝了join操作可能导致的数据倾斜问题

当join操作有数据倾斜问题并且其中一个RDD的数据量较小时，可以优先考虑这种方式，效果非常好。map join的过程如下图所示

公众号：旧时光大数据



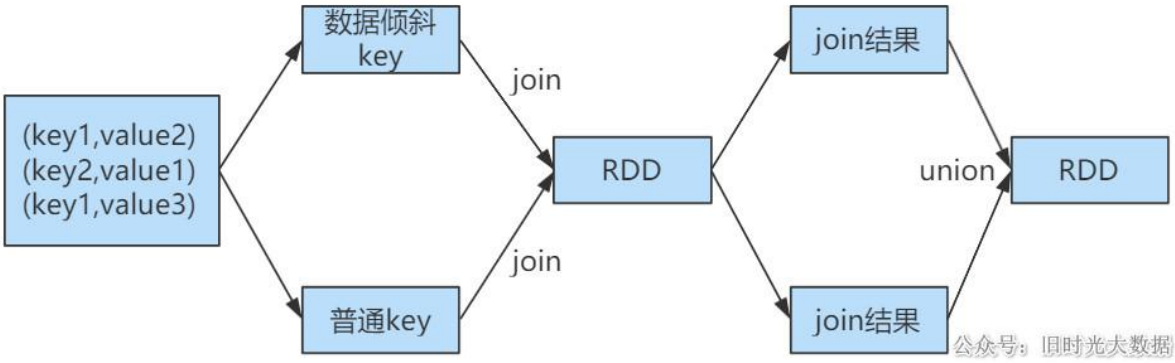
2）不使用场景分析

由于Spark的广播变量是在每个Executor中保存一个副本，如果两个RDD数据量都比较大，那么如果将一个数据量比较大的RDD做成广播变量，那么很有可能会造成内存溢出

7、解决方案六：sample采样对倾斜key单独进行join

在Spark中，如果某个RDD只有一个key，那么在shuffle过程中会默认将此key对应的数据打散，由不同的reduce端task进行处理

当由单个key导致数据倾斜时，可有将发生数据倾斜的key单独提取出来，组成一个RDD，然后用这个原本会导致倾斜的key组成的RDD根其他RDD单独join，此时，根据Spark的运行机制，此RDD中的数据会在shuffle阶段被分散到多个task中去进行join操作。倾斜key单独join的流程如下图所示



1）适用场景分析

对于RDD中的数据，可以将其转换为一个中间表，或者是直接使用countByKey()的方式，看一个这个RDD中各个key对应的数据量，此时如果你发现整个RDD就一个key的数据量特别多，那么就可以考虑使用这种方法

当数据量非常大时，可以考虑使用sample采样获取10%的数据，然后分析这10%的数据中哪个key可能会导致数据倾斜，然后将这个key对应的数据单独提取出来

2）不适用场景分析

如果一个RDD中导致数据倾斜的key很多，那么此方案不适用

8、解决方案七：使用随机数以及扩容进行join

公众号：旧时光大数据

如果在进行join操作时，RDD中有大量的key导致数据倾斜，那么进行分拆key也没什么意义，此时就只能使用最后一种方案来解决问题了，对于join操作，我们可以考虑对其中一个RDD数据进行扩容，另一个RDD进行稀释后再join

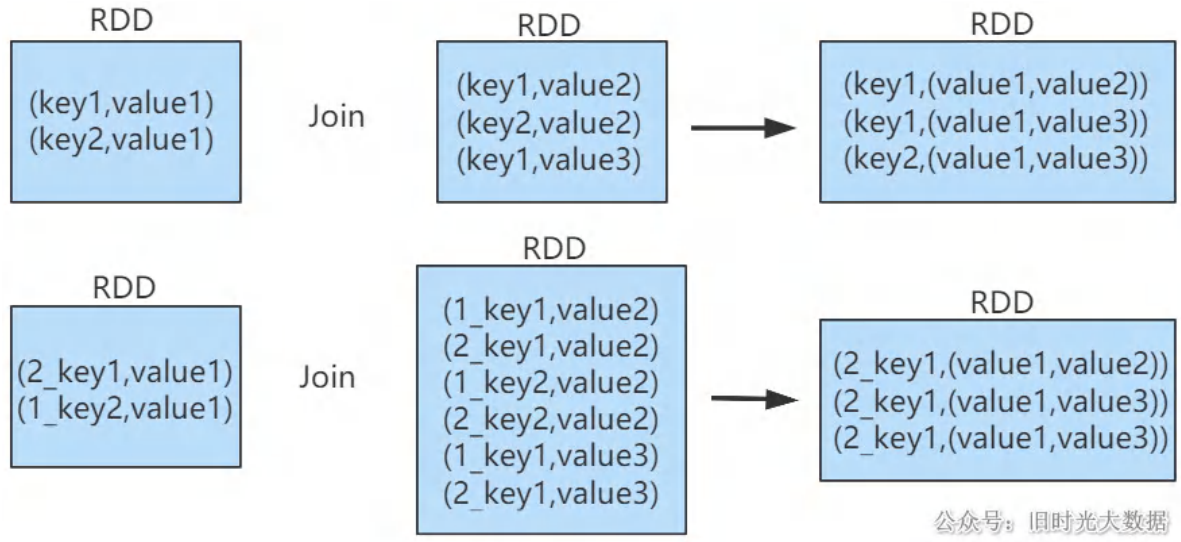
我们会将原先一样的key通过附加随机前缀变成不一样的key，然后就可以将这些处理后的“不同key”分散到多个task中去处理，而不是让一个task处理大量的相同key。这一种方案是针对有大量倾斜key的情况，没法将部分key拆分出来进行单独处理，需要对整个RDD进行数据扩容，对内存资源要求很高

1）核心思想

选择一个RDD，使用flatMap进行扩容，对每条数据的key添加数值前缀（1~N的数值），将一条数据映射为多条数据；（扩容）

选择另外一个RDD，进行map映射操作，每条数据的key都打上一个随机数作为前缀（1~N的随机数）；（稀释）

将两个处理后的RDD，进行join操作



2）局限性

如果两个RDD都很大，那么将RDD进行N倍的扩容显然行不通

使用扩容的方式只能缓解数据倾斜，不能彻底解决数据倾斜问题

3）使用方案七对方案六进一步优化分析

当RDD中有几个key导致数据倾斜时，方案六不再适用，而方案七又非常消耗资源，此时可以引入方案七的思想完善方案六

（1）对包含少数几个数据量过大的key的那个RDD，通过sample算子采样出一份样本来，然后统计一下每个key的数量，计算出来数据量最大的是哪几个key

（2）然后将这几个key对应的数据从原来的RDD中拆分出来，形成一个单独的RDD，并给每个key都打上n以内的随机数作为前缀，而不会导致倾斜的大部分key形成另外一个RDD

（3）接着将需要join的另一个RDD，也过滤出来那几个倾斜key对应的数据并形成一个单独的RDD，将每条数据膨胀成n条数据，这n条数据都按顺序附加一个0~n的前缀，不会导致倾斜的大部分key也形成另外一个RDD

（4）再将附加了随机前缀的独立RDD与另一个膨胀n倍的独立RDD进行join，此时就可以将原先相同的key打散成n份，分散到多个task中去进行join了

（5）而另外两个普通的RDD就照常join即可

（6）最后将两次join的结果使用union算子合并起来即可，就是最终的join结果

公众号：旧时光大数据

旧版数据倾斜及其解决方案

1、数据倾斜

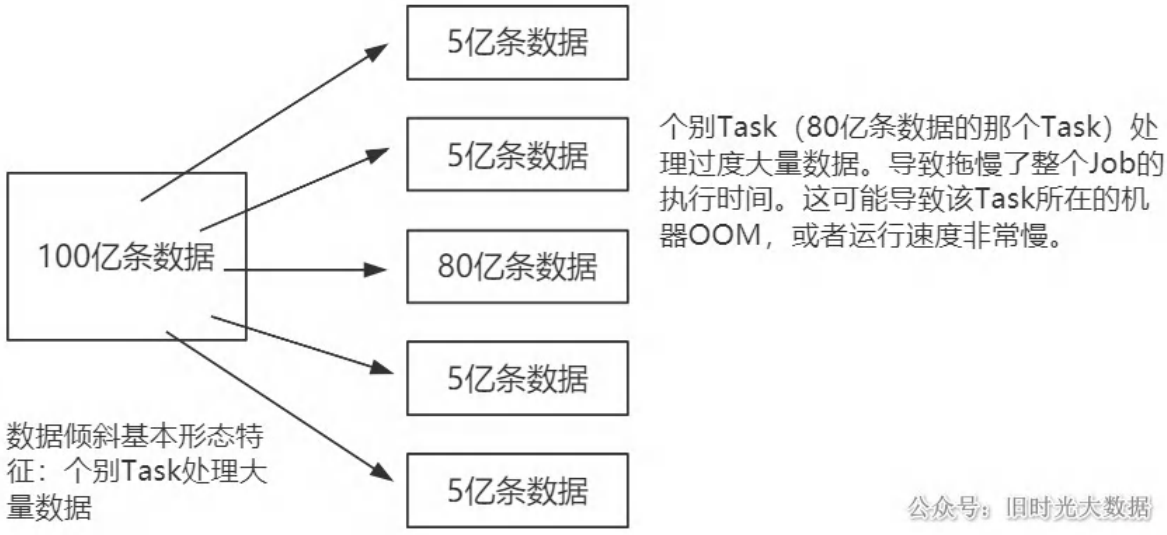
数据倾斜指的是，并行处理的数据集中，某一部分（如Spark或Kafka的一个Partition）的数据显著多于其它部分，从而使得该部分的处理速度成为整个数据集处理的瓶颈

数据倾斜俩大直接致命后果

1）数据倾斜直接会导致一种情况：Out Of Memory

2）运行速度慢

主要是发生在Shuffle阶段。同样Key的数据条数太多了。导致了某个key(下图中的80亿条)所在的Task数据量太大了。远远超过其他Task所处理的数据量



一个经验结论是：一般情况下，OOM的原因都是数据倾斜

2、如何定位数据倾斜

1）是不是有OOM情况出现，一般是少数出现内存溢出的问题

2）是不是应用运行时间差异很大，总体时间很长

3）你需要了解你所处理的数据key的分布情况，如果有些key具有大量的条数，那么就要小心数据倾斜的问题

4）一般需要通过Spark Web UI和其它一些监控方式中出现的异常来综合判断

5）数据倾斜一般会发生在shuffle过程中。很大程度上是你使用了可能会触发shuffle操作的算子：distinct、groupByKey、reduceByKey、aggregateByKey、join、cogroup、repartition等

原因： 查看任务-》查看Stage-》查看代码

某个task执行特别慢的情况

某个task莫名其妙内存溢出的情况

查看导致数据倾斜的key的数据分布情况

85	154	0	SUCCESS	PROCESS_LOCAL	3 / rz-data-hdp-dn0912.rz.sankuai.com	2016/01/29 13:42:02	3.2 min	0.9 s	807.7 KB / 8691
86	155	0	SUCCESS	PROCESS_LOCAL	46 / rz-data-hdp-dn0890.rz.sankuai.com	2016/01/29 13:42:02	49 s	0.5 s	531.4 KB / 5309
87	156	0	SUCCESS	PROCESS_LOCAL	92 / rz-data-hdp-dn1275.rz.sankuai.com	2016/01/29 13:42:02	31 s	0.6 s	360.7 KB / 3696
88	157	0	SUCCESS	PROCESS_LOCAL	64 / rz-data-hdp-dn0121.rz.sankuai.com	2016/01/29 13:42:02	27 s	0.4 s	406.1 KB / 4104
89	158	0	SUCCESS	PROCESS_LOCAL	13 / rz-data-hdp-dn1184.rz.sankuai.com	2016/01/29 13:42:02	14 s	0.4 s	347.3 KB / 3561
90	159	0	SUCCESS	PROCESS_LOCAL	5 / rz-data-hdp-dn0912.rz.sankuai.com	2016/01/29 13:42:02	13 s	0.3 s	351.4 KB / 3622
91	160	0	RUNNING	PROCESS_LOCAL	90 / rz-data-hdp-dn0059.rz.sankuai.com	2016/01/29 13:42:02	3.9 min	0.8 s	1617.0 KB / 18545
92	161	0	SUCCESS	PROCESS_LOCAL	87 / rz-data-hdp-dn0879.rz.sankuai.com	2016/01/29 13:42:02	26 s	0.4 s	318.1 KB / 3081
93	162	0	SUCCESS	PROCESS_LOCAL	55 / rz-data-hdp-dn0875.rz.sankuai.com	2016/01/29 13:42:02	19 s	0.5 s	359.6 KB / 3574
94	163	0	RUNNING	PROCESS_LOCAL	82 / rz-data-hdp-dn0430.rz.sankuai.com	2016/01/29 13:42:02	3.9 min	0.8 s	2023.4 KB / 22812
95	164	0	SUCCESS	PROCESS_LOCAL	99 / rz-data-hdp-dn0817.rz.sankuai.com	2016/01/29 13:42:02	5 s	0.2 s	188.1 KB / 1426
96	165	0	SUCCESS	PROCESS_LOCAL	56 / rz-data-hdp-dn0875.rz.sankuai.com	2016/01/29 13:42:02	10 s	0.3 s	214.5 KB / 1683
97	166	0	SUCCESS	PROCESS_LOCAL	71 / rz-data-hdp-dn0576.rz.sankuai.com	2016/01/29 13:42:02	2.9 min	0.4 s	673.8 KB / 6932
98	167	0	SUCCESS	PROCESS_LOCAL	77 / rz-data-hdp-dn0242.rz.sankuai.com	2016/01/29 13:42:02	13 s	0.3 s	276.3 KB / 2349
99	168	0	RUNNING	PROCESS_LOCAL	58 / rz-data-hdp-dn0491.rz.sankuai.com	2016/01/29 13:42:02	3.9 min	1 s	1321.0 KB / 14508

3、数据倾斜的集中典型情况

1）数据源中的数据分布不均匀，Spark需要频繁交互

2）数据集中的不同Key由于分区方式，导致数据倾斜

3）JOIN操作中，一个数据集中的数据分布不均匀，另一个数据集较小（主要）

4）聚合操作中，数据集中的数据分布不均匀（主要）

5）JOIN操作中，两个数据集都比较大，其中只有几个Key的数据分布不均匀

6）JOIN操作中，两个数据集都比较大，有很多Key的数据分布不均匀

7）数据集中少数几个key数据量很大，不重要，其他数据均匀

注意：

1）需要处理的数据倾斜问题就是Shuffle后数据的分布是否均匀问题

2）只要保证最后的结果是正确的，可以采用任何方式来处理数据倾斜，只要保证在处理过程中不发生数据倾斜就可以

4、数据倾斜的处理方法

1）数据源中的数据分布不均匀，Spark需要频繁交互

解决方案：避免数据源的数据倾斜

实现原理：通过在Hive中对倾斜的数据进行预处理，以及在进行kafka数据分发时尽量进行平均分配。这种方案从根源上解决了数据倾斜，彻底避免了在Spark中执行shuffle类算子，那么肯定就不会有数据倾斜的问题了

方案优点：实现起来简单便捷，效果还非常好，完全规避掉了数据倾斜，Spark作业的性能会大幅度提高

方案缺点：治标不治本，Hive或者Kafka中还是会发生数据倾斜

适用情况：在一些Java系统与Spark结合使用的项目中，会出现Java代码频繁调用Spark作业的场景，而且对Spark作业的执行性能要求很高，就比较适合使用这种方案。将数据倾斜提前到上游的Hive ETL，每天仅执行一次，只有那一次是比较慢的，而之后每次Java调用Spark作业时，执行速度都会很快，能够提供更好的用户体验

总结：前台的Java系统和Spark有很频繁的交互，这个时候如果Spark能够在最短的时间内处理数据，往往会给前端有非常好的体验。这个时候可以将数据倾斜的问题抛给数据源端，在数据源端进行数据倾斜的处理。但是这种方案没有真正的处理数据倾斜问题

2）数据集中的不同Key由于分区方式，导致数据倾斜

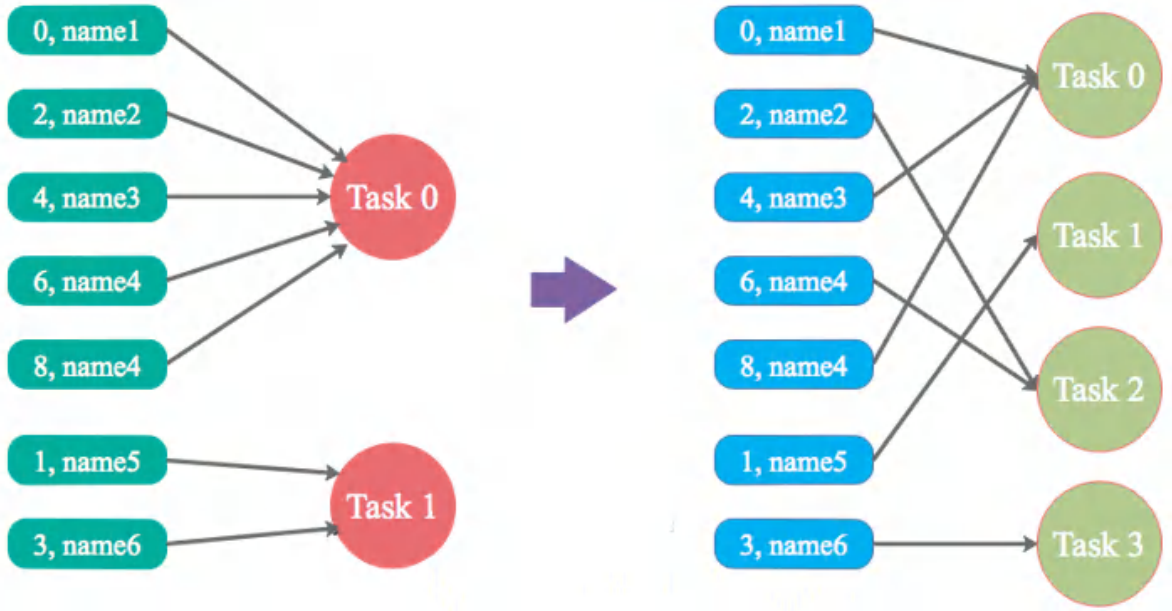
解决方案：调整并行度

实现原理：增加shuffle read task的数量，可以让原本分配给一个task的多个key分配给多个task，从而让每个task处理比原来更少的数据

方案优点：实现起来比较简单，可以有效缓解和减轻数据倾斜的影响

方案缺点：只是缓解了数据倾斜而已，没有彻底根除问题，根据实践经验来看，其效果有限

实践经验：该方案通常无法彻底解决数据倾斜，因为如果出现一些极端情况，比如某个key对应的数据量有100万，那么无论你的task数量增加到多少，都无法处理



总结：调整并行度：适合于有大量key由于分区算法或者分区数的问题，将key进行了不均匀分区，可以通过调大或者调小分区数来试试是否有效

缓解数据倾斜（自定义Partitioner）

适用场景：大量不同的Key被分配到了相同的Task造成该Task数据量过大。

解决方案： 使用自定义的Partitioner实现类代替默认的HashPartitioner，尽量将所有不同的Key均匀分配到不同的Task中

优势： 不影响原有的并行度设计。如果改变并行度，后续Stage的并行度也会默认改变，可能会影响后续Stage

劣势： 适用场景有限，只能将不同Key分散开，对于同一Key对应数据集非常大的场景不适用。效果与调整并行度类似，只能缓解数据倾斜而不能完全消除数据倾斜。而且需要根据数据特点自定义专用的Partitioner，不够灵活

3）JOIN操作中，一个数据集中的数据分布不均匀，另一个数据集较小（主要）

解决方案：Reduce side Join转变为Map side Join

方案适用场景：在对RDD使用join类操作，或者是在Spark SQL中使用join语句时，而且join操作中的一个RDD或表的数据量比较小（比如几百M），比较适用此方案

方案实现原理：普通的join是会走shuffle过程的，而一旦shuffle，就相当于会将相同key的数据拉取到一个shuffle read task中再进行join，此时就是reduce join。但是如果一个RDD是比较小的，则可以采用广播小RDD全量数据+map算子来实现与join同样的效果，也就是map join，此时就不会发生shuffle操作，也就不会发生数据倾斜

方案优点：对join操作导致的数据倾斜，效果非常好，因为根本就不会发生shuffle，也就根本不会发生数据倾斜

方案缺点：适用场景较少，因为这个方案只适用于一个大表和一个小表的情况

4）聚合操作中，数据集中的数据分布不均匀（主要）

解决方案：两阶段聚合（局部聚合+全局聚合）

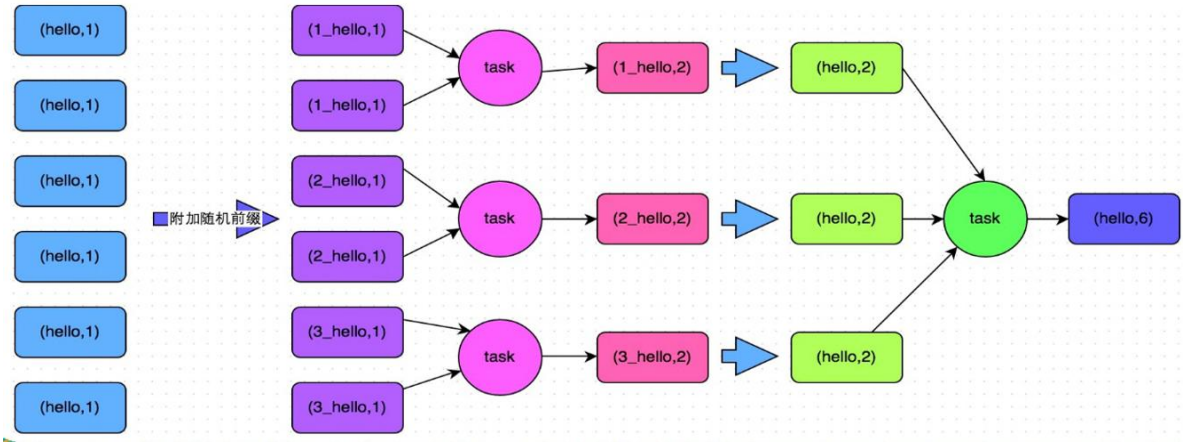
适用场景：对RDD执行reduceByKey等聚合类shuffle算子或者在Spark SQL中使用group by语句进行分组聚合时，比较适用这种方案

实现原理：将原本相同的key通过附加随机前缀的方式，变成多个不同的key，就可以让原本被一个task处理的数据分散到多个task上去做局部聚合，进而解决单个task处理数据量过多的问题。接着去除掉随机前缀，再次进行全局聚合，就可以得到最终的结果。具体原理见下图

优点：对于聚合类的shuffle操作导致的数据倾斜，效果是非常不错的。通常都可以解决掉数据倾斜，或者至少是大幅度缓解数据倾斜，将Spark作业的性能提升数倍以上

缺点：仅仅适用于聚合类的shuffle操作，适用范围相对较窄。如果是join类的shuffle操作，还得用其它的解决方案

将相同key的数据分拆处理



最后全局聚合即可

5）JOIN操作中，两个数据集都比较大，其中只有几个Key的数据分布不均匀

解决方案：为倾斜key增加随机前/后缀

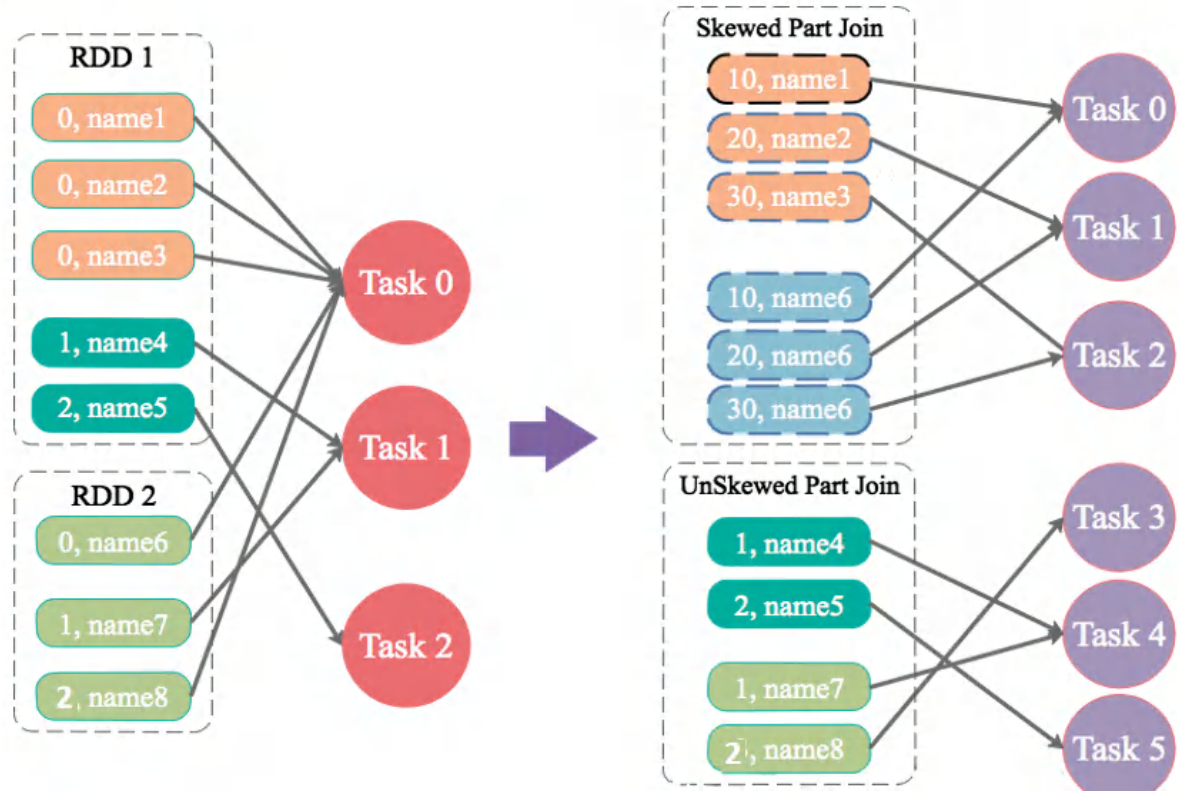
适用场景：两张表都比较大，无法使用Map侧Join。其中一个RDD有少数几个Key的数据量过大，另外一个RDD的Key分布较为均匀

实现原理：将有数据倾斜的RDD中倾斜Key对应的数据集单独抽取出来加上随机前缀，另外一个RDD每条数据分别与随机前缀结合形成新的RDD（笛卡尔积，相当于将其数据增到到原来的N倍，N即为随机前缀的总个数），然后将二者Join后去掉前缀。然后将不包含倾斜Key的剩余数据进行Join。最后将两次Join的结果集通过union合并，即可得到全部Join结果

优势：相对于Map侧Join，更能适应大数据集的Join。如果资源充足，倾斜部分数据集与非倾斜部分数据集可并行进行，效率提升明显。且只针对倾斜部分的数据做数据扩展，增加的资源消耗有限

劣势：如果倾斜Key非常多，则另一侧数据膨胀非常大，此方案不适用。而且此时对倾斜Key与非倾斜Key分开处理，需要扫描数据集两遍，增加了开销

注意：具有倾斜Key的RDD数据集中，key的数量比较少



6）JOIN操作中，两个数据集都比较大，有很多Key的数据分布不均匀

解决方案：随机前缀和扩容RDD进行join

适用场景：如果在进行join操作时，RDD中有大量的key导致数据倾斜，那么进行分拆key也没什么意义

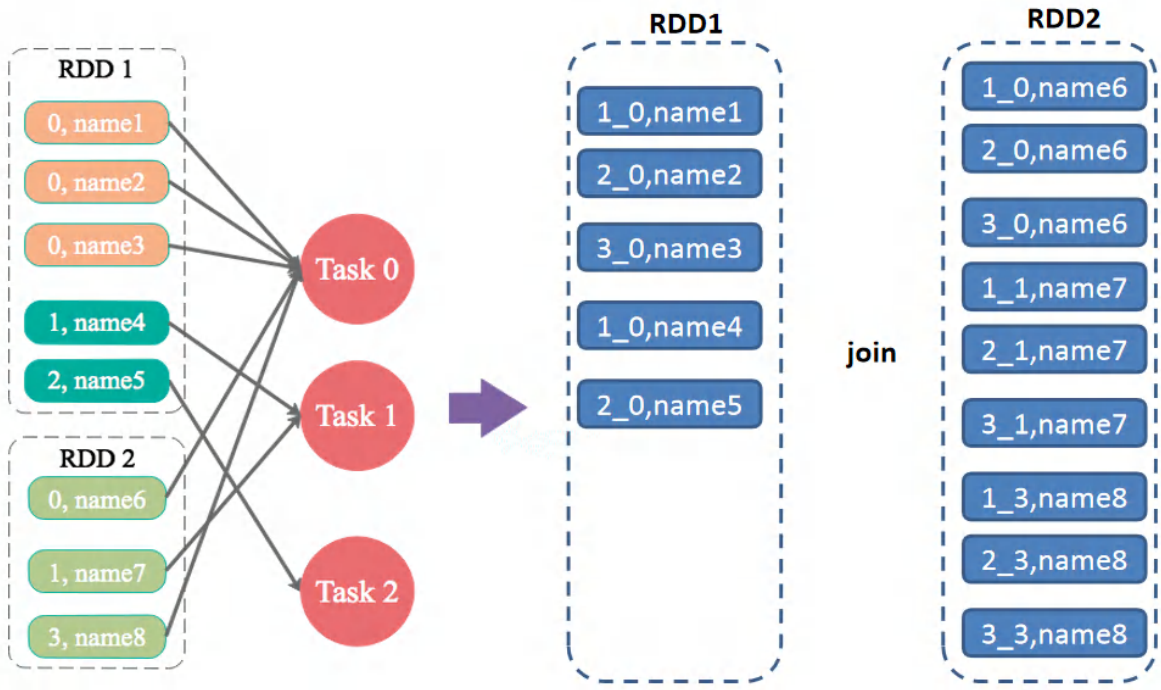
实现思路：将该RDD的每条数据都打上一个n以内的随机前缀。同时对另外一个正常的RDD进行扩容，将每条数据都扩容成n条数据，扩容出来的每条数据都依次打上一个0~n的前缀。最后将两个处理后的RDD进行join即可。和上一种方案是尽量只对少数倾斜key对应的数据进行特殊处理，由于处理过程需要扩容RDD，因此上一种方案扩容RDD后对内存的占用并不大；而这一种方案是针对有大量倾斜key的情况，没法将部分key拆分出来进行单独处理，因此只能对整个RDD进行数据扩容，对内存资源要求很高

优点：对join类型的数据倾斜基本都可以处理，而且效果也相对比较显著，性能提升效果非常不错

缺点：该方案更多的是缓解数据倾斜，而不是彻底避免数据倾斜。而且需要对整个RDD进行扩容，对内存资源要求很高

实践经验：曾经开发一个数据需求的时候，发现一个join导致了数据倾斜。优化之前，作业的执行时间大约是60分钟左右；使用该方案优化之后，执行时间缩短到10分钟左右，性能提升了6倍。注意：将倾斜Key添加1-N的随机前缀，并将被Join的数据集相应的扩大N倍（需要将1-N数字添加到每一条数据上作为前缀）

公众号：旧时光大数据



7）数据集中少数几个key数据量很大，不重要，其他数据均匀

解决方案：过滤少数倾斜Key

适用场景：如果发现导致倾斜的key就少数几个，而且对计算本身的影响并不大的话，那么很适合使用这种方案。比如99%的key就对应10条数据，但是只有一个key对应了100万数据，从而导致了数据倾斜

优点：实现简单，而且效果也很好，可以完全规避掉数据倾斜

缺点：适用场景不多，大多数情况下，导致倾斜的key还是很多的，并不是只有少数几个

实践经验：在项目中我们也采用过这种方案解决数据倾斜。有一次发现某一天Spark作业在运行的时候突然OOM了，追查之后发现，是Hive表中的某一个key在那天数据异常，导致数据量暴增。因此就采取每次执行前先进行采样，计算出样本中数据量最大的几个key之后，直接在程序中将那些key给过滤掉

Spark的stage如何划分？在源码中是怎么判断属于Shuffle Map Stage或Result Stage的？

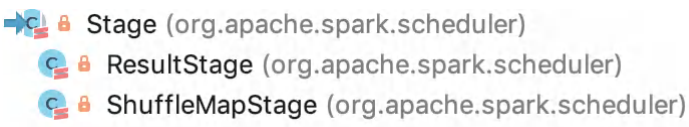
可回答：Stage划分的依据是什么？

问过的一些公司：字节跳动，安恒信息，抖音，美团，字节，陌陌，端点数据(2021.07)，网易严选(2021.09)

参考答案：

Job按照RDD之间的依赖关系是否为宽依赖，由DAGScheduler划分为一个Stage。

Satge有两个子类：



Result Stage

ResultStage在RDD的某些分区上应用函数来计算action操作的结果。 对于诸如first（）和lookup（）之类的操作，某些stage可能无法在RDD的所有分区上运行。

公众号：旧时光大数据

简言之，ResultStage是应用action操作在action上进而得出计算结果。

源码如下：

```
private[spark] class ResultStage(
    id: Int,
    rdd: RDD[_],
    val func: (TaskContext, Iterator[_]) => _,
    val partitions: Array[Int],
    parents: List[Stage],
    firstJobId: Int,
    callSite: CallSite)
  extends Stage(id, rdd, partitions.length, parents, firstJobId, callSite) {

  /**
   * The active job for this result stage. Will be empty if the job has already finished
   * (e.g., because the job was cancelled).
   */
  private[this] var _activeJob: Option[ActiveJob] = None

  def activeJob: Option[ActiveJob] = _activeJob

  def setActiveJob(job: ActiveJob): Unit = {
    _activeJob = Option(job)
  }

  def removeActiveJob(): Unit = {
    _activeJob = None
  }

  /**
   * Returns the sequence of partition ids that are missing (i.e. needs to be computed).
   *
   * This can only be called when there is an active job.
   */
  override def findMissingPartitions(): Seq[Int] = {
    val job = activeJob.get
    (0 until job.numPartitions).filter(id => !job.finished(id))
  }

  override def toString: String = "ResultStage " + id
}
```

Shuffle Map Stage

ShuffleMapStage 是中间的stage，为shuffle生产数据。它们在shuffle之前出现。当执行完毕之后，结果数据被保存，以便reduce 任务可以获取到。

源码如下：

```
private[spark] class ShuffleMapStage(
    id: Int,
    rdd: RDD[_],
    numTasks: Int,
    parents: List[Stage],
    firstJobId: Int,
    callSite: CallSite,
    val shuffleDep: ShuffleDependency[_, _, _],
    mapOutputTrackerMaster: MapOutputTrackerMaster)
  extends Stage(id, rdd, numTasks, parents, firstJobId, callSite)
```

Spark join在什么情况下会变成窄依赖?

问过的一些公司：小米

参考答案：

公众号：旧时光大数据

如果需要join的两个表，本身已经有分区器，且分区的数目相同，此时，相同的key在同一个分区内，就是窄依赖。反之，如果两个需要join的表中没有分区器或者分区数量不同，在join的时候需要shuffle，那么就是宽依赖。

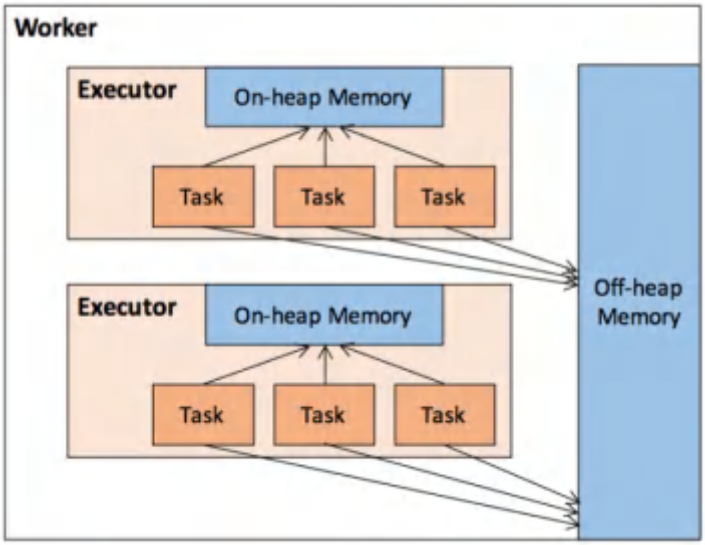
Spark的内存模型?

问过的一些公司：阿里，360，小米，安恒信息，京东(2021.07)

参考答案：

1、堆内和堆外内存规划

作为一个JVM 进程，Executor 的内存管理建立在JVM的内存管理之上，Spark对 JVM的堆内（On-heap）空间进行了更为详细的分配，以充分利用内存。同时，Spark引入了堆外（Off-heap）内存，使之可以直接在工作节点的系统内存中开辟空间，进一步优化了内存的使用。堆内内存受到JVM统一管理，堆外内存是直接向操作系统进行内存的申请和释放。



1）堆内内存

默认情况下，Spark 仅仅使用了堆内内存。Executor 端的堆内内存区域大致可以分为以下四大块：

- **Execution 内存**：主要用于存放 Shuffle、Join、Sort、Aggregation 等计算过程中的临时数据
- **Storage 内存**：主要用于存储 spark 的 cache 数据，例如RDD的缓存、unroll数据；
- **用户内存（User Memory）**：主要用于存储 RDD 转换操作所需要的数据，例如 RDD 依赖等信息。
- **预留内存（Reserved Memory）**：系统预留内存，会用来存储Spark内部对象。

堆内内存的大小，由Spark应用程序启动时的 –executor-memory 或 spark.executor.memory 参数配置。Executor 内运行的并发任务共享 JVM 堆内内存，这些任务在缓存 RDD 数据和广播（Broadcast）数据时占用的内存被规划为存储（Storage）内存，而这些任务在执行 Shuffle 时占用的内存被规划为执行（Execution）内存，剩余的部分不做特殊规划，那些Spark内部的对象实例，或者用户定义的 Spark 应用程序中的对象实例，均占用剩余的空间。不同的管理模式下，这三部分占用的空间大小各不相同。

Spark对堆内内存的管理是一种逻辑上的"规划式"的管理，因为对象实例占用内存的申请和释放都由JVM完成，Spark只能在申请后和释放前记录这些内存，我们来看其具体流程：

申请内存流程如下：

- Spark 在代码中 new 一个对象实例；
- JVM 从堆内内存分配空间，创建对象并返回对象引用；
- Spark 保存该对象的引用，记录该对象占用的内存。

公众号：旧时光大数据

释放内存流程如下：

- Spark记录该对象释放的内存，删除该对象的引用；
- 等待JVM的垃圾回收机制释放该对象占用的堆内内存。

2）堆外内存

为了进一步优化内存的使用以及提高Shuffle时排序的效率，Spark引入了堆外（Off-heap）内存，使之可以直接在工作节点的系统内存中开辟空间，存储经过序列化的二进制数据。

堆外内存意味着把内存对象分配在Java虚拟机的堆以外的内存，这些内存直接受操作系统管理（而不是虚拟机）。这样做的结果就是能保持一个较小的堆，以减少垃圾收集对应用的影响。

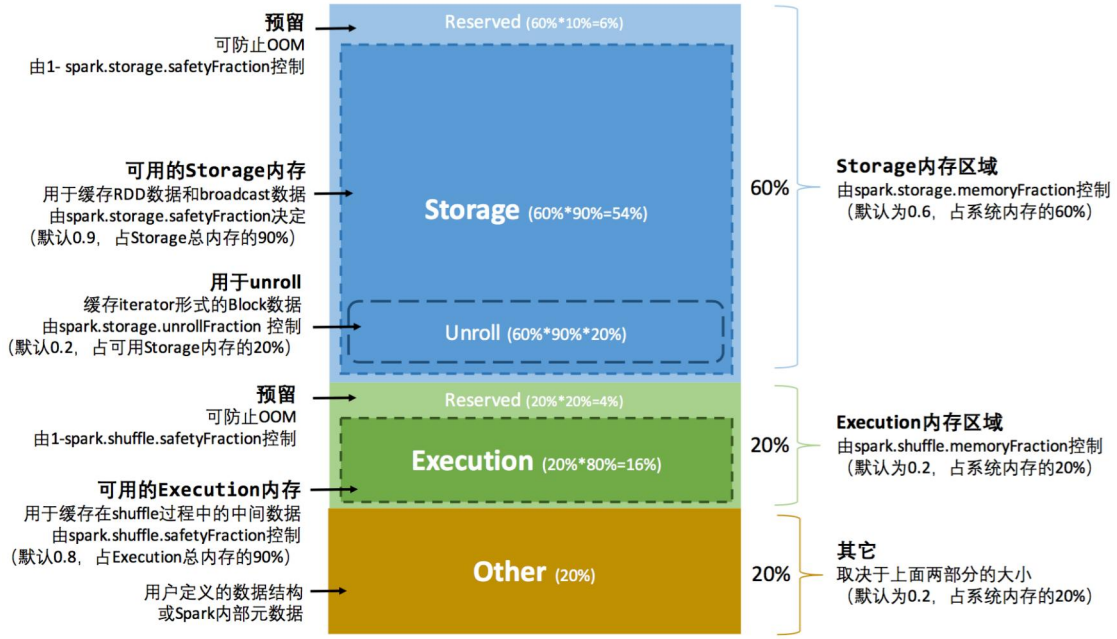
利用JDK Unsafe API（从Spark 2.0开始，在管理堆外的存储内存时不再基于Tachyon，而是与堆外的执行内存一样，基于 JDK Unsafe API 实现），Spark 可以直接操作系统堆外内存，减少了不必要的内存开销，以及频繁的 GC 扫描和回收，提升了处理性能。堆外内存可以被精确地申请和释放（堆外内存之所以能够被精确的申请和释放，是由于内存的申请和释放不再通过JVM机制，而是直接向操作系统申请，JVM对于内存的清理是无法准确指定时间点的，因此无法实现精确的释放），而且序列化的数据占用的空间可以被精确计算，所以相比堆内内存来说降低了管理的难度，也降低了误差。

在默认情况下堆外内存并不启用，可通过配置 spark.memory.offHeap.enabled 参数启用，并由 spark.memory.offHeap.size 参数设定堆外空间的大小。除了没有 other 空间，堆外内存与堆内内存的划分方式相同，所有运行中的并发任务共享存储内存和执行内存。

2、内存空间分配

1）静态内存管理

在Spark最初采用的静态内存管理机制下，存储内存、执行内存和其他内存的大小在Spark应用程序运行期间均为固定的，但用户可以应用程序启动前进行配置，堆内内存的分配如图所示：



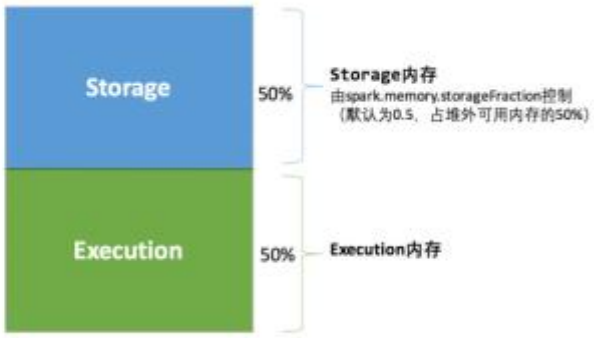
可以看到，可用的堆内内存的大小需要按照下列方式计算：

```
可用的存储内存 = systemMaxMemory * spark.storage.memoryFraction *
spark.storage.safety Fraction
可用的执行内存 = systemMaxMemory * spark.shuffle.memoryFraction *
spark.shuffle.safety Fraction

```

其中systemMaxMemory取决于当前JVM堆内内存的大小，最后可用的执行内存或者存储内存要在此基础上与各自的memoryFraction 参数和safetyFraction 参数相乘得出。上述计算公式中的两个 safetyFraction 参数，其意义在于在逻辑上预留出 1-safetyFraction 这么一块保险区域，降低因实际内存超出当前预设范围而导致 OOM 的风险（上文提到，对于非序列化对象的内存采样估算会产生误差）。值得注意的是，这个预留的保险区域仅仅是一种逻辑上的规划，在具体使用时 Spark 并没有区别对待，和"其它内存"一样交给了 JVM 去管理。

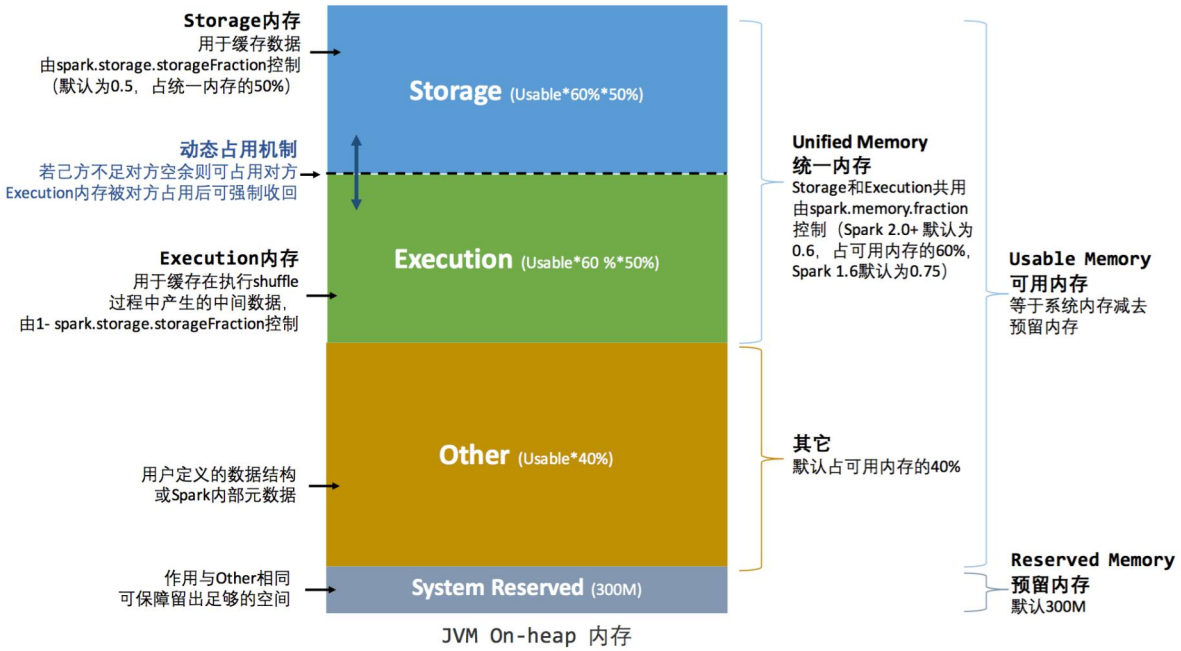
Storage内存和Execution内存都有预留空间，目的是防止OOM，因为Spark堆内内存大小的记录是不准确的，需要留出保险区域。

堆外的空间分配较为简单，只有存储内存和执行内存，如下图所示。可用的执行内存和存储内存占用的空间大小直接由参数spark.memory.storageFraction 决定，由于堆外内存占用的空间可以被精确计算，所以无需再设定保险区域。



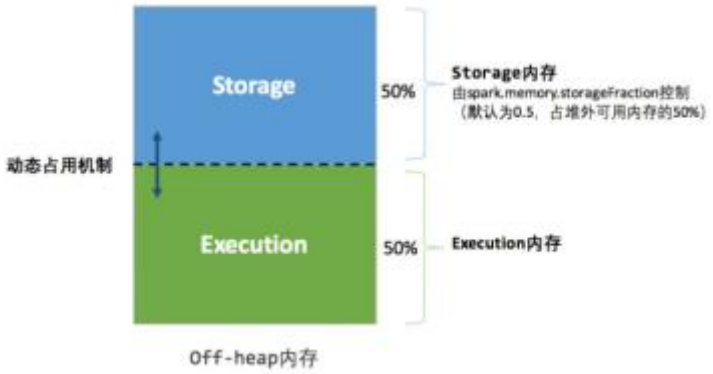
2）统一内存管理

Spark1.6 之后引入的统一内存管理机制，与静态内存管理的区别在于存储内存和执行内存共享同一块空间，可以动态占用对方的空闲区域，统一内存管理的堆内内存结构如图所示：



统一内存管理的堆外内存结构如下图所示：

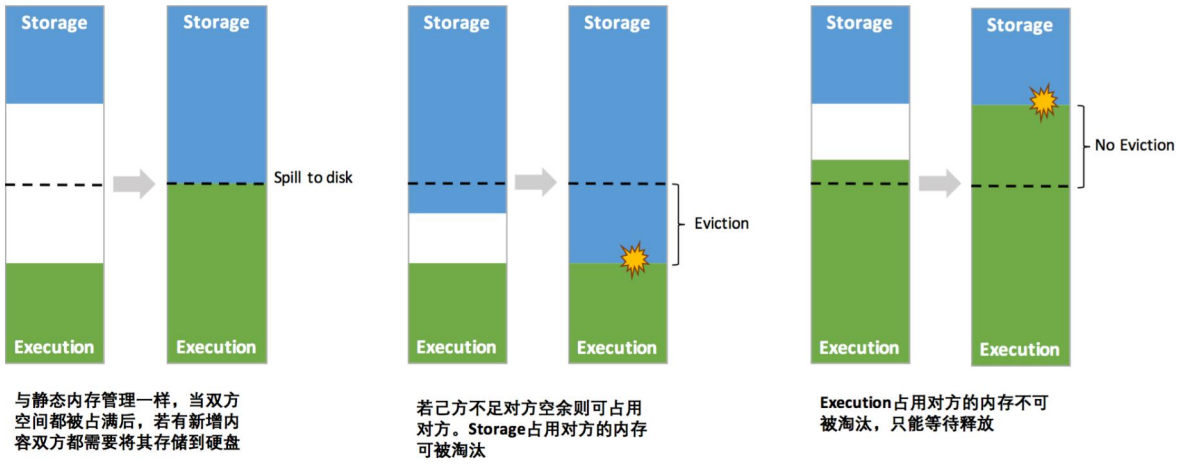
公众号：旧时光大数据



其中最重要的优化在于动态占用机制，其规则如下：

- 设定基本的存储内存和执行内存区域（spark.storage.storageFraction参数），该设定确定了双方各自拥有的空间的范围；
- 双方的空间都不足时，则存储到硬盘；若己方空间不足而对方空余时，可借用对方的空间;（存储空间不足是指不足以放下一个完整的Block）
- 执行内存的空间被对方占用后，可让对方将占用的部分转存到硬盘，然后"归还"借用的空间；
- 存储内存的空间被对方占用后，无法让对方"归还"，因为需要考虑 Shuffle过程中的很多因素，实现起来较为复杂。

统一内存管理的动态占用机制如图所示：



凭借统一内存管理机制，Spark 在一定程度上提高了堆内和堆外内存资源的利用率，降低了开发者维护 Spark 内存的难度，但并不意味着开发者可以高枕无忧。譬如，所以如果存储内存的空间太大或者说缓存的数据过多，反而会导致频繁的全量垃圾回收，降低任务执行时的性能，因为缓存的 RDD 数据通常都是长期驻留内存的。所以要想充分发挥 Spark 的性能，需要开发者进一步了解存储内存和执行内存各自的管理方式和实现原理。

3、存储内存管理

1）RDD的持久化机制

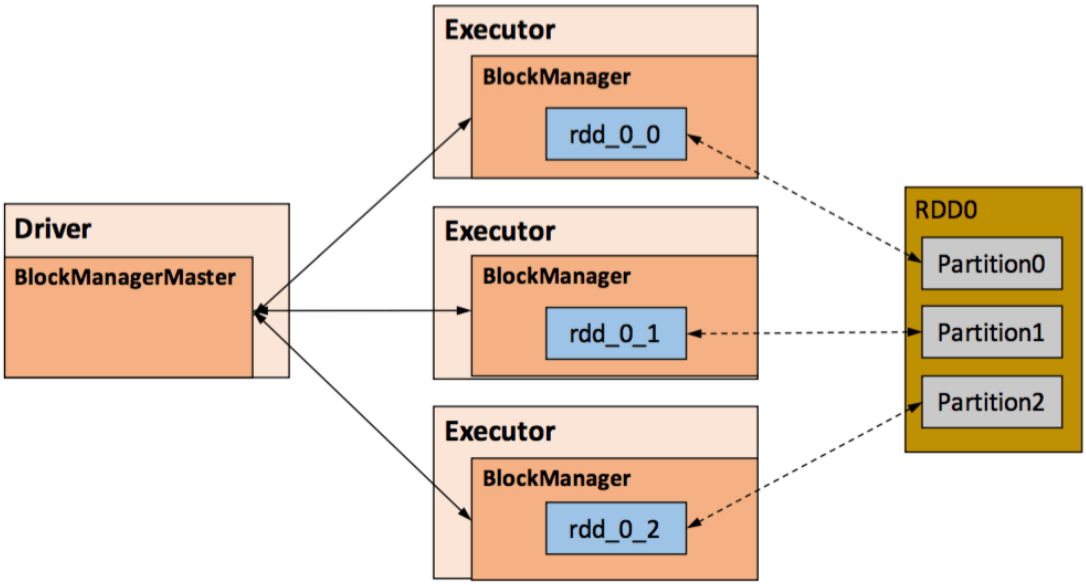
弹性分布式数据集（RDD）作为 Spark 最根本的数据抽象，是只读的分区记录（Partition）的集合，只能基于在稳定物理存储中的数据集上创建，或者在其他已有的RDD上执行转换（Transformation）操作产生一个新的RDD。转换后的RDD与原始的RDD之间产生的依赖关系，构成了血统（Lineage）。凭借血统，Spark 保证了每一个RDD都可以被重新恢复。但RDD的所有转换都是惰性的，即只有当一个返回结果给Driver的行动（Action）发生时，Spark才会创建任务读取RDD，然后真正触发转换的执行。

Task在启动之初读取一个分区时，会先判断这个分区是否已经被持久化，如果没有则需要检查 Checkpoint 或按照血统重新计算。所以如果一个 RDD 上要执行多次行动，可以在第一次行动中使用 persist或cache 方法，在内存或磁盘中持久化或缓存这个RDD，从而在后面的行动时提升计算速度。

公众号：旧时光大数据

事实上，cache 方法是使用默认的 MEMORY_ONLY 的存储级别将 RDD 持久化到内存，故缓存是一种特殊的持久化。 堆内和堆外存储内存的设计，便可以对缓存RDD时使用的内存做统一的规划和管理。

RDD的持久化由 Spark的Storage模块负责，实现了RDD与物理存储的解耦合。Storage模块负责管理Spark在计算过程中产生的数据，将那些在内存或磁盘、在本地或远程存取数据的功能封装了起来。在具体实现时Driver端和 Executor 端的Storage模块构成了主从式的架构，即Driver端的BlockManager为Master，Executor端的BlockManager 为 Slave。Storage模块在逻辑上以Block为基本存储单位，RDD的每个Partition经过处理后唯一对应一个 Block（BlockId 的格式为rdd_RDD-ID_PARTITION-ID ）。Driver端的Master负责整个Spark应用程序的Block的元数据信息的管理和维护，而Executor端的Slave需要将Block的更新等状态上报到Master，同时接收Master 的命令，例如新增或删除一个RDD。



在对RDD持久化时，Spark规定了MEMORY_ONLY、MEMORY_AND_DISK 等7种不同的存储级别，而存储级别是以下5个变量的组合：

```
class StorageLevel private(
private var _useDisk: Boolean, //磁盘
private var _useMemory: Boolean, //这里其实是指堆内内存
private var _useOffHeap: Boolean, //堆外内存
private var _deserialized: Boolean, //是否为非序列化
private var _replication: Int = 1 //副本个数
)

```

Spark中7种存储级别如下：

公众号：旧时光大数据

持久化级别	含义
MEMORY_ONLY	以非序列化的Java对象的方式持久化在JVM内存中。如果内存无法完全存储RDD所有的partition，那么那些没有持久化的partition就会在下一次需要使用它们的时候，重新被计算
MEMORY_AND_DISK	同上，但是当某些partition无法存储在内存中时，会持久化到磁盘中。下次需要使用这些partition时，需要从磁盘上读取
MEMORY_ONLY_SER	同MEMORY_ONLY，但是会使用Java序列化方式，将Java对象序列化后进行持久化。可以减少内存开销，但是需要进行反序列化，因此会加大CPU开销
MEMORY_AND_DISK_SER	同MEMORY_AND_DISK，但是使用序列化方式持久化Java对象
DISK_ONLY	使用非序列化Java对象的方式持久化，完全存储到磁盘上
MEMORY_ONLY_2 MEMORY_AND_DISK_2等等	如果是尾部加了2的持久化级别，表示将持久化数据复用一份，保存到其他节点，从而在数据丢失时，不需要再次计算，只需要使用备份数据即可

通过对数据结构的分析，可以看出存储级别从三个维度定义了RDD的 Partition（同时也就是Block）的存储方式：

- 存储位置：磁盘 / 堆内内存 / 堆外内存。如MEMORY_AND_DISK是同时在磁盘和堆内内存上存储，实现了冗余备份。OFF_HEAP 则是只在堆外内存存储，目前选择堆外内存时不能同时存储到其他位置。
- 存储形式：Block 缓存到存储内存后，是否为非序列化的形式。如 MEMORY_ONLY是非序列化方式存储，OFF_HEAP 是序列化方式存储。
- 副本数量：大于1时需要远程冗余备份到其他节点。如DISK_ONLY_2需要远程备份1个副本。

2）RDD的缓存过程

RDD 在缓存到存储内存之前，Partition中的数据一般以迭代器（Iterator）的数据结构来访问，这是Scala语言中一种遍历数据集合的方法。通过Iterator可以获取分区中每一条序列化或者非序列化的数据项(Record)，这些Record的对象实例在逻辑上占用了JVM堆内内存的other部分的空间，同一Partition的不同 Record 的存储空间并不连续。

RDD 在缓存到存储内存之后，Partition 被转换成Block，Record在堆内或堆外存储内存中占用一块连续的空间。将Partition由不连续的存储空间转换为连续存储空间的过程，Spark称之为"展开"（Unroll）。

Block 有序列化和非序列化两种存储格式，具体以哪种方式取决于该 RDD 的存储级别。非序列化的Block以一种 DeserializedMemoryEntry 的数据结构定义，用一个数组存储所有的对象实例，序列化的Block则以SerializedMemoryEntry的数据结构定义，用字节缓冲区（ByteBuffer）来存储二进制数据。每个 Executor 的 Storage模块用一个链式Map结构（LinkedHashMap）来管理堆内和堆外存储内存中所有的Block对象的实例，对这个LinkedHashMap新增和删除间接记录了内存的申请和释放。

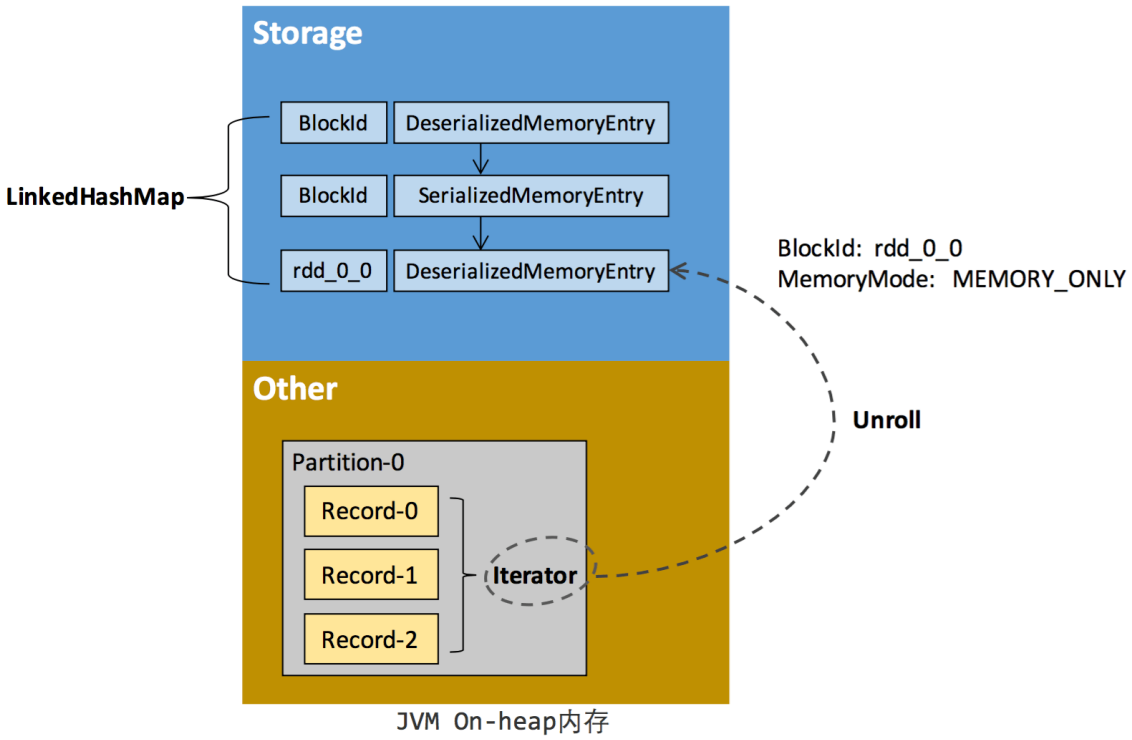
因为不能保证存储空间可以一次容纳 Iterator 中的所有数据，当前的计算任务在 Unroll 时要向 MemoryManager 申请足够的Unroll空间来临时占位，空间不足则Unroll失败，空间足够时可以继续进行。

对于序列化的Partition，其所需的Unroll空间可以直接累加计算，一次申请。

对于非序列化的 Partition 则要在遍历 Record 的过程中依次申请，即每读取一条 Record，采样估算其所需的Unroll空间并进行申请，空间不足时可以中断，释放已占用的Unroll空间。

如果最终Unroll成功，当前Partition所占用的Unroll空间被转换为正常的缓存 RDD的存储空间，如下图所示。

公众号：旧时光大数据



在静态内存管理时，Spark 在存储内存中专门划分了一块 Unroll 空间，其大小是固定的，统一内存管理时则没有对 Unroll 空间进行特别区分，当存储空间不足时会根据动态占用机制进行处理。

3）淘汰与落盘

由于同一个Executor的所有的计算任务共享有限的存储内存空间，当有新的 Block 需要缓存但是剩余空间不足且无法动态占用时，就要对LinkedHashMap中的旧Block进行淘汰（Eviction），而被淘汰的Block如果其存储级别中同时包含存储到磁盘的要求，则要对其进行落盘（Drop），否则直接删除该Block。

存储内存的淘汰规则为：

- 被淘汰的旧Block要与新Block的MemoryMode相同，即同属于堆外或堆内内存；
- 新旧Block不能属于同一个RDD，避免循环淘汰；
- 旧Block所属RDD不能处于被读状态，避免引发一致性问题；
- 遍历LinkedHashMap中Block，按照最近最少使用（LRU）的顺序淘汰，直到满足新Block所需的空间。其中LRU是LinkedHashMap的特性。

落盘的流程则比较简单，如果其存储级别符合 *useDisk为true的条件，再根据其* deserialized判断是否是非序列化的形式，若是则对其进行序列化，最后将数据存储到磁盘，在Storage模块中更新其信息。

4、执行内存管理

执行内存主要用来存储任务在执行Shuffle时占用的内存，Shuffle是按照一定规则对RDD数据重新分区的过程，我们来看Shuffle的Write和Read两阶段对执行内存的使用：

1）Shuffle Write

若在map端选择普通的排序方式，会采用ExternalSorter进行外排，在内存中存储数据时主要占用堆内执行空间。

若在map端选择 Tungsten 的排序方式，则采用ShuffleExternalSorter直接对以序列化形式存储的数据排序，在内存中存储数据时可以占用堆外或堆内执行空间，取决于用户是否开启了堆外内存以及堆外执行内存是否足够。

2）Shuffle Read

在对reduce端的数据进行聚合时，要将数据交给Aggregator处理，在内存中存储数据时占用堆内执行空间。

如果需要进行最终结果排序，则要将再次将数据交给ExternalSorter 处理，占用堆内执行空间。

在ExternalSorter和Aggregator中，Spark会使用一种叫AppendOnlyMap的哈希表在堆内执行内存中存储数据，但在 Shuffle 过程中所有数据并不能都保存到该哈希表中，当这个哈希表占用的内存会进行周期性地采样估算，当其大到一定程度，无法再从MemoryManager 申请到新的执行内存时，Spark就会将其全部内容存储到磁盘文件中，这个过程被称为溢存(Spill)，溢存到磁盘的文件最后会被归并(Merge)。

Shuffle Write 阶段中用到的Tungsten是Databricks公司提出的对Spark优化内存和CPU使用的计划（钨丝计划），解决了一些JVM在性能上的限制和弊端。Spark会根据Shuffle的情况来自动选择是否采用Tungsten排序。

Tungsten 采用的页式内存管理机制建立在MemoryManager之上，即 Tungsten 对执行内存的使用进行了一步的抽象，这样在 Shuffle 过程中无需关心数据具体存储在堆内还是堆外。

每个内存页用一个MemoryBlock来定义，并用 Object obj 和 long offset 这两个变量统一标识一个内存页在系统内存中的地址。

堆内的MemoryBlock是以long型数组的形式分配的内存，其obj的值为是这个数组的对象引用，offset是long型数组的在JVM中的初始偏移地址，两者配合使用可以定位这个数组在堆内的绝对地址；堆外的MemoryBlock是直接申请到的内存块，其obj为null，offset是这个内存块在系统内存中的64位绝对地址。Spark用MemoryBlock巧妙地将堆内和堆外内存页统一抽象封装，并用页表(pageTable)管理每个Task申请到的内存页。

Tungsten 页式管理下的所有内存用64位的逻辑地址表示，由页号和页内偏移量组成：

- 页号：占13位，唯一标识一个内存页，Spark在申请内存页之前要先申请空闲页号。
- 页内偏移量：占51位，是在使用内存页存储数据时，数据在页内的偏移地址。

有了统一的寻址方式，Spark 可以用64位逻辑地址的指针定位到堆内或堆外的内存，整个Shuffle Write排序的过程只需要对指针进行排序，并且无需反序列化，整个过程非常高效，对于内存访问效率和CPU使用效率带来了明显的提升。

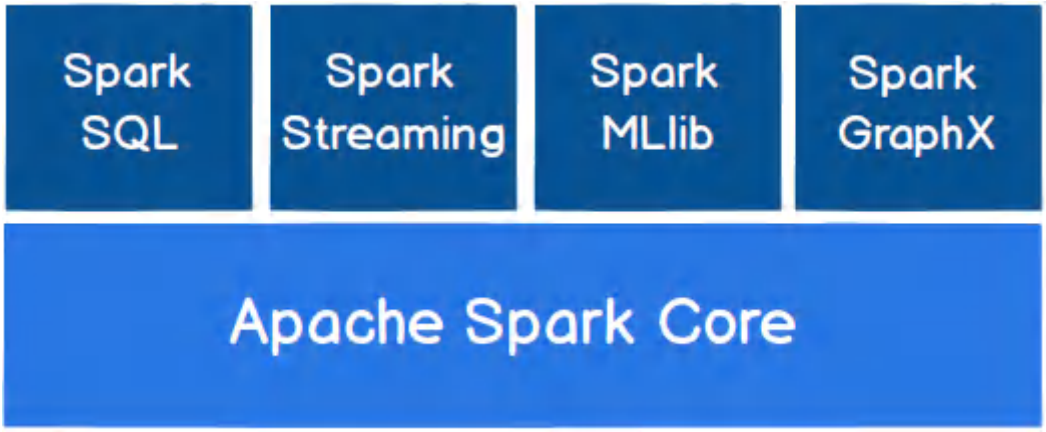
Spark的存储内存和执行内存有着截然不同的管理方式：对于存储内存来说，Spark用一个LinkedHashMap来集中管理所有的Block，Block由需要缓存的 RDD的Partition转化而成；而对于执行内存，Spark用AppendOnlyMap来存储 Shuffle过程中的数据，在Tungsten排序中甚至抽象成为页式内存管理，开辟了全新的JVM内存管理机制。

Spark分哪几个部分（模块）？分别有什么作用（做什么，自己用过哪些，做过什么）？

可回答： Spark的组件有哪些?

问过的一些公司：字节，美团，映客直播

参考答案：



Spark Core

Spark Core 中提供了 Spark 最基础与最核心的功能，Spark 其他的功能如：Spark SQL， Spark Streaming，GraphX, MLlib 都是在 Spark Core 的基础上进行扩展的

Spark SQL

Spark SQL 是Spark 用来操作结构化数据的组件。通过 Spark SQL，用户可以使用 SQL或者Apache Hive 版本的 SQL 方言（HQL）来查询数据。

Spark Streaming

Spark Streaming 是 Spark 平台上针对实时数据进行流式计算的组件，提供了丰富的处理数据流的API。

Spark MLlib

MLlib 是 Spark 提供的一个机器学习算法库。MLlib 不仅提供了模型评估、数据导入等额外的功能，还提供了一些更底层的机器学习原语。

Spark GraphX

GraphX 是 Spark 面向图计算提供的框架与算法库。

RDD的宽依赖和窄依赖，举例一些算子

可回答：Spark的宽窄算子

问过的一些公司：今日头条 x 2，字节 x 7，快手x3，美团 x 2，妙盈科技，网易云音乐x2，蘑菇街，阿里 x 4，阿里蚂蚁(2021.08)x2，阿里(2021.09)x2，京东x3，海康，抖音，米哈游x2，顺丰 x 2，小米 x 5，360，拼多多，腾讯x2，作业帮社招，猿辅导，ebay，网易，蔚来(2021.09)，快手(2021.09)

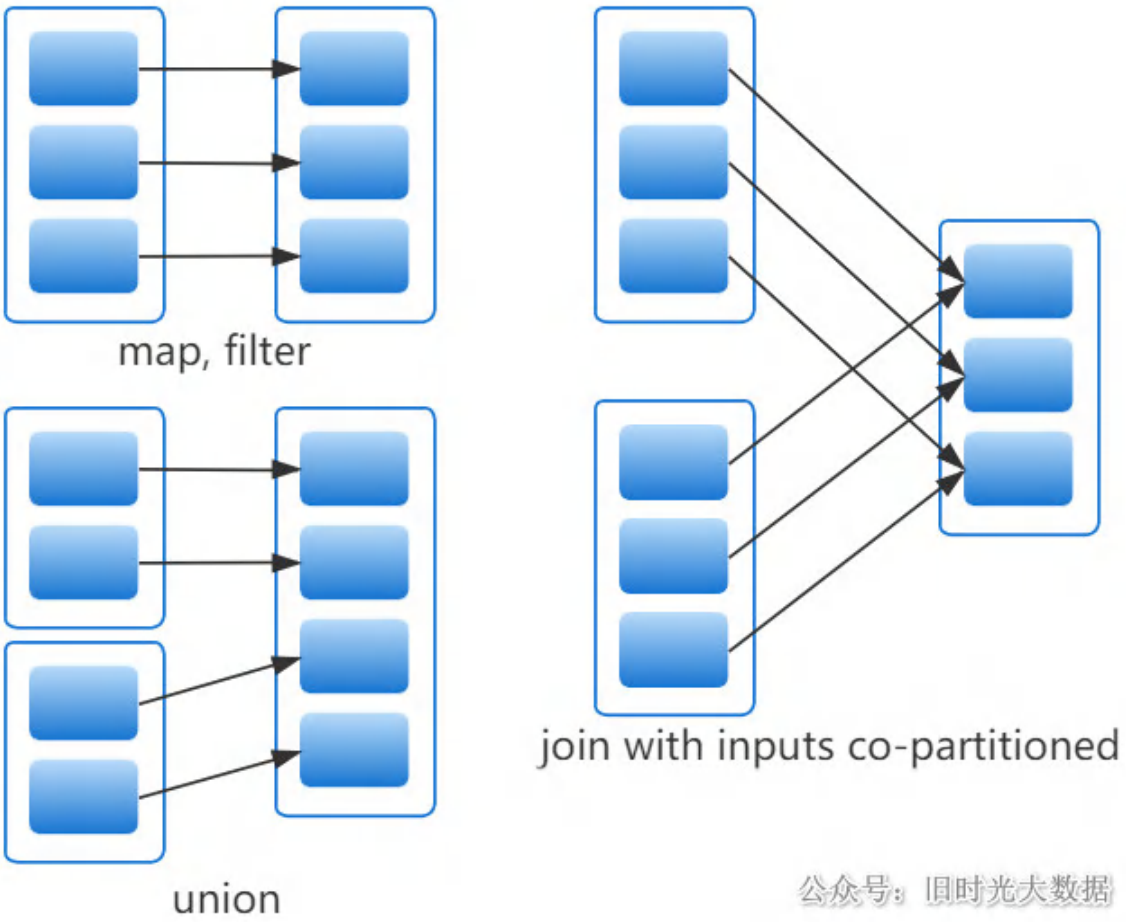
参考答案：

RDD和它依赖的parent RDD(s)的关系有两种不同的类型，即窄依赖（narrow dependency）和宽依赖（wide dependency）

1、窄依赖

窄依赖指的是每一个parent RDD的Partition最多被子RDD的一个Partition使用

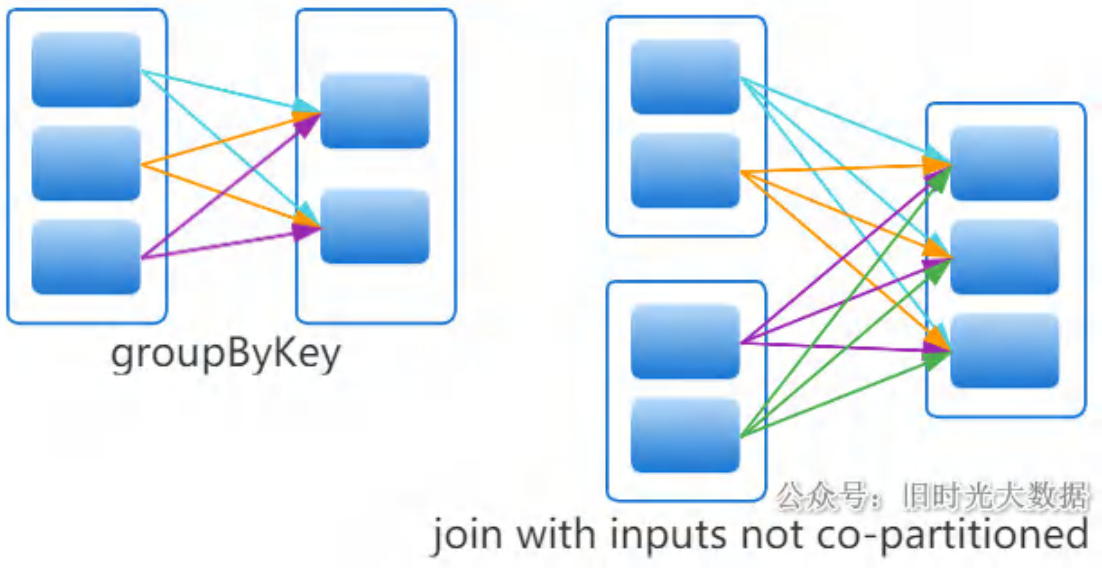
公众号：旧时光大数据



比如map，filter，union属于窄依赖

2、宽依赖

宽依赖指的是多个子RDD的Partition会依赖同一个parent RDD的Partition



具有宽依赖的 transformations 包括：sort，reduceByKey，groupByKey，join，和调用rePartition函数的任何操作

Spark SQL的GroupBy会造成窄依赖吗?

问过的一些公司：字节

参考答案：

公众号：旧时光大数据

不会，它属于宽依赖

GroupBy是行动算子吗

问过的一些公司：阿里(2021.09)

参考答案：

GroupBy在Spark中是Transformation，产生shuffle

Spark的宽依赖和窄依赖，为什么要这么划分？

可回答：RDD为什么要设计宽窄依赖

问过的一些公司：有赞，网易x2

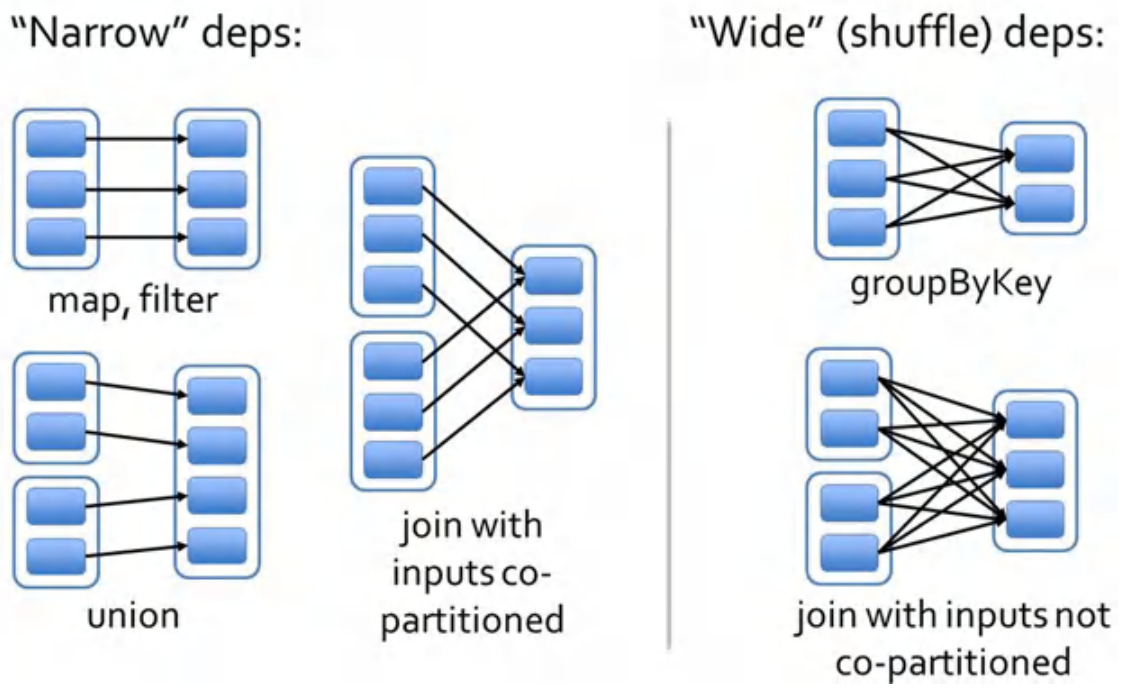
参考答案：

Spark中RDD的高效与DAG图有着莫大的关系，在DAG调度中需要对计算过程划分stage，而划分依据就是RDD之间的依赖关系。针对不同的转换函数，RDD之间的依赖关系分类窄依赖（narrow dependency）和宽依赖（wide dependency, 也称 shuffle dependency）。

1、宽依赖与窄依赖

注意：也可参考前面的题目

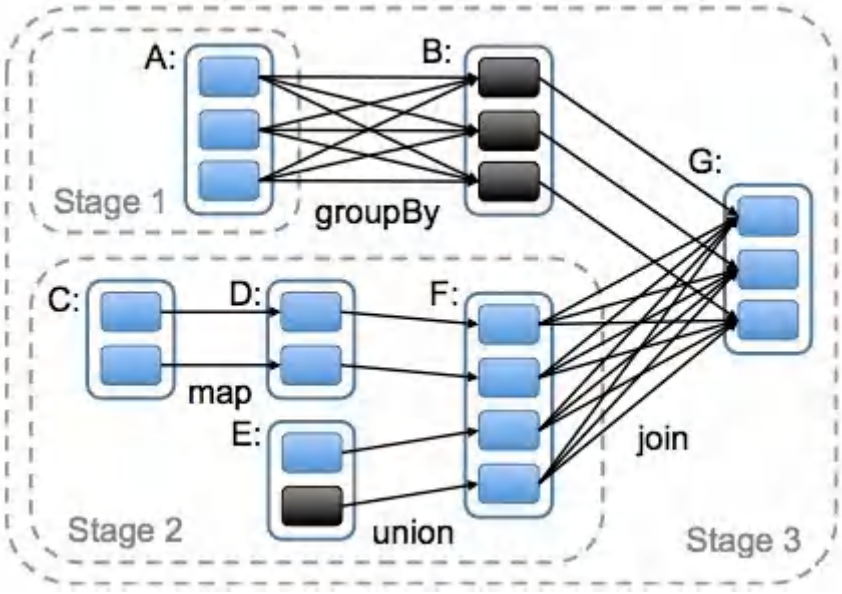
- 窄依赖是指父RDD的每个分区只被子RDD的一个分区所使用，**子RDD分区通常对应常数个父RDD分区**(O(1)，与数据规模无关)
- 相应的，宽依赖是指父RDD的每个分区都可能被多个子RDD分区所使用，**子RDD分区通常对应所有的父RDD分区**(O(n)，与数据规模有关)



2、划分宽窄依赖原由

需要从宽窄依赖和容错性方面考虑。

Spark基于lineage的容错性是指，如果一个RDD出错，那么可以从它的所有父RDD重新计算所得，如果一个RDD仅有一个父RDD（即窄依赖），那么这种重新计算的代价会非常小。



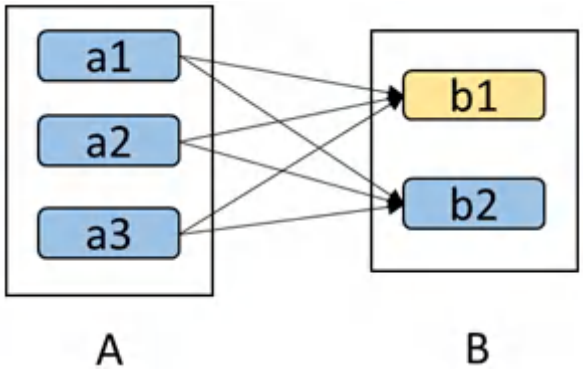
对于Spark基于Checkpoint（物化）的容错机制，在上图中，宽依赖得到的结果（经历过Shuffle过程）是很昂贵的，因此，Spark将此结果物化到磁盘上了，以备后面使用。

对于join操作有两种情况，如果join操作的每个partition仅仅和已知的Partition进行join，此时的join操作就是窄依赖；其他情况的join操作就是宽依赖；因为是确定的Partition数量的依赖关系，所以就是窄依赖，得出一个推论，窄依赖不仅包含一对一的窄依赖，还包含一对固定个数的窄依赖（也就是说对父RDD的依赖的Partition的数量不会随着RDD数据规模的改变而改变）。

如果不进行宽窄依赖的划分，对于一些算子随便使用，可以使用窄依赖算子时也使用宽依赖算子，就会造成资源浪费，导致效率低下；同理，当需要使用宽依赖算子时，却使用窄依赖算子，则会导致我们得不到需要的结果。

相比于宽依赖，窄依赖对优化还有以下几点优势：

- 宽依赖往往对应着shuffle操作，需要在运行过程中将同一个父RDD的分区传入到不同的子RDD分区中，中间可能涉及多个节点之间的数据传输；而窄依赖的每个父RDD的分区只会传入到一个子RDD分区中，通常可以在一个节点内完成转换。
- 当RDD分区丢失时（某个节点故障），spark会对数据进行重算。
    - 对于窄依赖，由于父RDD的一个分区只对应一个子RDD分区，这样只需要重算和子RDD分区对应的父RDD分区即可，所以这个重算对数据的利用率是100%的；
    - 对于宽依赖，重算的父RDD分区对应多个子RDD分区，这样实际上父RDD 中只有一部分的数据是被用于恢复这个丢失的子RDD分区的，另一部分对应子RDD的其它未丢失分区，这就造成了多余的计算；更一般的，宽依赖中子RDD分区通常来自多个父RDD分区，极端情况下，所有的父RDD分区都要进行重新计算。
    - 如下图所示，b1分区丢失，则需要重新计算a1,a2和a3，这就产生了冗余计算(a1,a2,a3中对应b2的数据)。



公众号：旧时光大数据

说下Spark中的Transform和Action，为什么Spark要把操作分为Transform和Action？常用的列举一些，说下算子原理

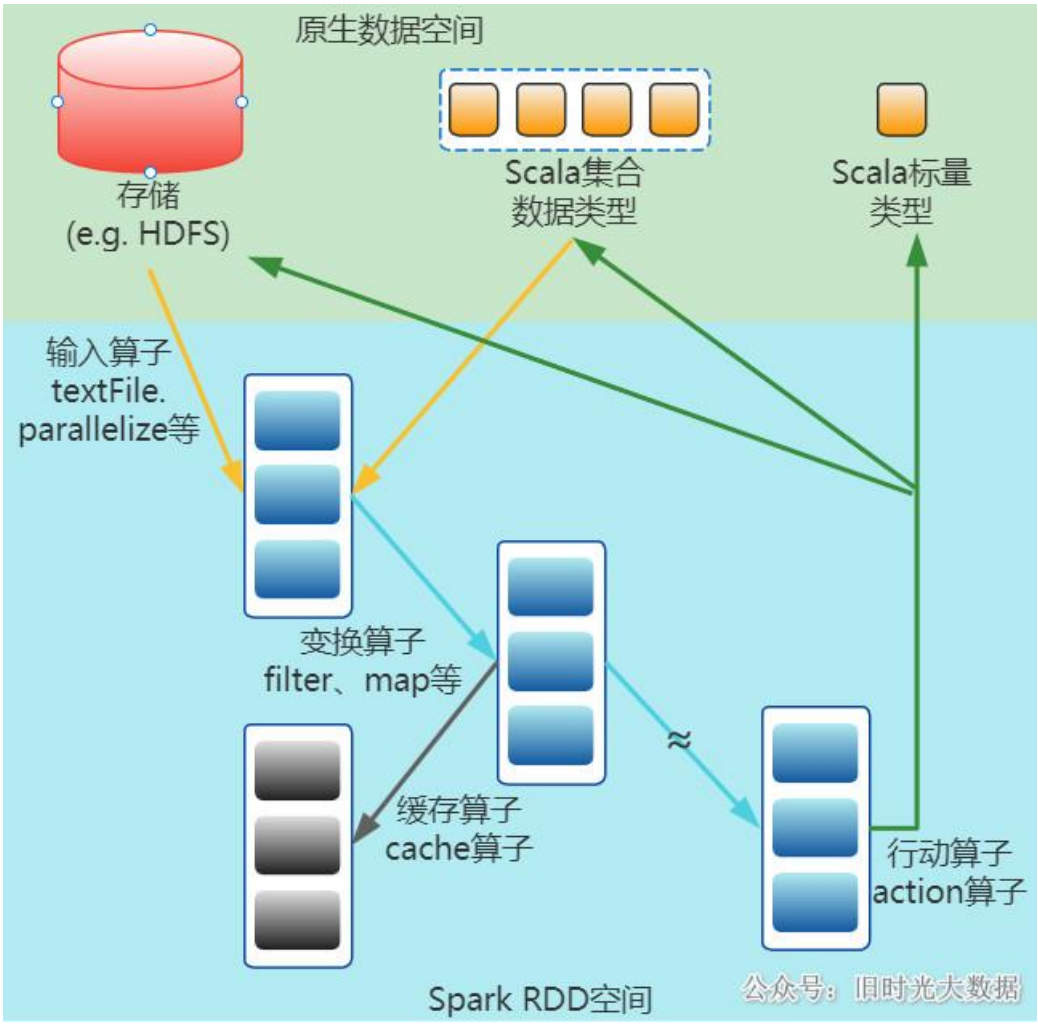
可回答：Spark常见的算子介绍一下

问过的一些公司：字节x3，字节(2021.08)，小米x3，阿里，猿辅导，ebay，美团x2，腾讯x2，端点数据(2021.07)

参考答案：

我们先来看下Spark算子的作用：

下图描述了Spark在运行转换中通过算子对RDD进行转换。 算子是RDD中定义的函数，可以对RDD中的数据进行转换和操作。



输入：在Spark程序运行中，数据从外部数据空间（如分布式存储：textFile读取HDFS等，parallelize方法输入Scala集合或数据）输入Spark，数据进入Spark运行时数据空间，转化为Spark中的数据块，通过BlockManager进行管理。

运行：在Spark数据输入形成RDD后便可以通过变换算子，如filter等，对数据进行操作并将RDD转化为新的RDD，通过Action算子，触发Spark提交作业。 如果数据需要复用，可以通过Cache算子，将数据缓存到内存。

输出：程序运行结束数据会输出Spark运行时空间，存储到分布式存储中（如saveAsTextFile输出到HDFS），或Scala数据或集合中（collect输出到Scala集合，count返回Scala int型数据）。

1、Transform和Action

Transformation是得到一个新的RDD，方式很多，比如从数据源生成一个新的RDD，从RDD生成一个新的RDD

公众号：旧时光大数据

Action是得到一个值，或者一个结果（直接将RDD cache到内存中）

因为所有的Transformation都是采用的懒策略，就是如果只是将Transformation提交是不会执行计算的，计算只有在Action被提交的时候才被触发。这样有利于减少内存消耗，提高了执行效率。

2、算子原理

1）Transformation

map(func)：返回一个新的分布式数据集，由每个原元素经过func函数转换后组成。

filter(func)：返回一个新的数据集，由经过func函数后返回值为true的原元素组成。

flatMap(func)：类似于map，但是每一个输入元素，会被映射为0到多个输出元素（因此，func函数的返回值是一个Seq，而不是单一元素）。

union(otherDataset)：返回一个新的数据集，由原数据集和参数联合而成。

groupByKey([numTasks])：在一个由（K,V）对组成的数据集上调用，返回一个（K，Seq[V])对的数据集。注意：默认情况下，使用8个并行任务进行分组，你可以传入numTask可选参数，根据数据量设置不同数目的Task。

reduceByKey(func, [numTasks])：在一个（K，V)对的数据集上使用，返回一个（K，V）对的数据集，key相同的值，都被使用指定的reduce函数聚合到一起。和groupbykey类似，任务的个数是可以通过第二个可选参数来配置的。

join(otherDataset, [numTasks])：在类型为（K,V)和（K,W)类型的数据集上调用，返回一个（K,(V,W))对，每个key中的所有元素都在一起的数据集。

2）Action

reduce(func)：通过函数func聚集数据集中的所有元素。Func函数接受2个参数，返回一个值。这个函数必须是关联性的，确保可以被正确的并发执行。

collect()：在Driver的程序中，以数组的形式，返回数据集的所有元素。这通常会在使用filter或者其它操作后，返回一个足够小的数据子集再使用，直接将整个RDD集Collect返回，很可能会让Driver程序OOM。

count()：返回数据集的元素个数。

foreach(func): 在数据集的每一个元素上，运行函数func。这通常用于更新一个累加器变量，或者和外部存储系统做交互。

Spark的哪些算子会有shuffle过程?

可回答：说下会产生Shuffle的算子，没有Shuffle的算子

问过的一些公司：字节x2，作业帮，海康，触宝(2021.07)

参考答案：

spark中会导致shuffle操作的有以下几种算子：

1、重分区类操作

比如repartition、repartitionAndSortWithinPartitions、coalesce(shuffle=true)等。重分区一般会shuffle，因为需要在整个集群中，对之前所有的分区的数据进行随机，均匀的打乱，然后把数据放入下游新的指定数量的分区内。

2、聚合、bykey类操作

公众号：旧时光大数据

比如reduceByKey、groupByKey、sortByKey等。byKey类的操作要对一个key，进行聚合操作，那么肯定要保证集群中，所有节点上的相同的key，移动到同一个节点上进行处理。

3、集合/表间交互操作

比如join、cogroup等。两个rdd进行join，就必须将相同join key的数据，shuffle到同一个节点上，然后进行相同key的两个rdd数据的笛卡尔乘积。

4、去重类操作

如distinct。

5、排序类操作

如sortByKey。

无shuffle操作的一些算子：

如map，filter，union等。

Spark有了RDD，为什么还要有Dataform和DataSet?

问过的一些公司：字节

参考答案：

RDD叫做弹性分布式数据集，与RDD类似，DataFrame是一个分布式数据容器，但是DataFrame不是类型安全的。
DataSet是DataFrame API的一个扩展，是Spark最新的数据抽象，结合了RDD和DataFrame的优点。

Spark的RDD、 DataFrame、 DataSet、 DataStream区别?

问过的一些公司：阿里，平安，小米，海康，京东，字节，米哈游，趋势科技

参考答案：

RDD

RDD（Resilient Distributed Dataset）叫做分布式数据集，是Spark中最基本的数据抽象。代码中是一个抽象类，它代表一个不可变、可分区、里面的元素可并行计算的集合。

DataFrame

DataFrame是一种以RDD为基础的分布式数据集，类似于传统数据库中的二维表格。DataFrame与RDD的主要区别在于，前者带有schema元信息，即DataFrame所表示的二维表数据集的每一列都带有名称和类型。

DataSet

1）是Dataframe API的一个扩展，是Spark最新的数据抽象。它提供了RDD的优势（强类型，使用强大的lambda函数的能力）以及Spark SQL优化执行引擎的优点。

2）用户友好的API风格，既具有类型安全检查也具有Dataframe的查询优化特性。

3）Dataset支持编解码器，当需要访问非堆上的数据时可以避免反序列化整个对象，提高了效率。

4）样例类被用来在Dataset中定义数据的结构信息，样例类中每个属性的名称直接映射到DataSet中的字段名称。

公众号：旧时光大数据

5） Dataframe是Dataset的特列，DataFrame=Dataset[Row] ，所以可以通过as方法将Dataframe转换为Dataset。Row是一个类型，跟Car、Person这些的类型一样，所有的表结构信息我都用Row来表示。

6）DataSet是强类型的。比如可以有Dataset[Car]，Dataset[Person]。

7）DataFrame只是知道字段，但是不知道字段的类型，所以在执行这些操作的时候是没办法在编译的时候检查是否类型失败的，比如你可以对一个String进行减法操作，在执行的时候才报错，而DataSet不仅仅知道字段，而且知道字段类型，所以有更严格的错误检查。就跟JSON对象和类对象之间的类比。

DataStream

DStream是随时间推移而收到的数据的序列。在内部，每个时间区间收到的数据都作为RDD存在，而DStream是由这些RDD所组成的序列(因此得名“离散化”)。所以简单来将，DStream就是对RDD在实时数据处理场景的一种封装。

Spark的Job、Stage、Task分别介绍下，如何划分?

问过的一些公司：字节x6，腾讯，小米，360，阿里x2，流利说，vivo，快手x2，祖龙娱乐，一点咨询，陌陌，拼多多，星环科技，京东x2，猿辅导，顺丰，字节(2021.08)

参考答案：

一个job含有多个stage，一个stage含有多个task。action 的触发会生成一个job，Job会提交给DAGScheduler，分解成Stage。

Job

- 包含很多task的并行计算，可以认为**是Spark RDD 里面的action**,每个action的计算会生成一个job。
- 用户提交的Job会提交给DAGScheduler，Job会被分解成Stage和Task。

Stage

- DAGScheduler根据shuffle将job划分为不同的stage，同一个stage中包含多个task，这些tasks有相同的shuffle dependencies。
- 一个Job会被拆分为多组Task，每组任务被称为一个Stage，就像Map Stage，Reduce Stage。

Task

- 即stage下的一个任务执行单元，一般来说，一个rdd有多少个partition，就会有多少个task，因为每一个task只是处理一个partition上的数据。
- 每个executor执行的task的数目，可以由submit时，--num-executors(on yarn)来指定。

Job->Stage->Task每一层都是1对n的关系。

Application 、job、Stage、task之间的关系

参考答案：

Application是spark的一个应用程序，它包含了客户端写好的代码以及任务运行的时候需要的资源信息。

后期一个application中有很多个action操作，一个action操作就是一个job，一个job会存在大量的宽依赖，后期会按照宽依赖进行stage的划分，一个job又产生很多个stage，每一个stage内部有很多可以并行的task。

Stage内部逻辑

公众号：旧时光大数据

参考答案：

1）每一个stage中按照RDD的分区划分了很多个可以并行运行的task

2）把每一个stage中这些可以并行运行的task都封装到一个taskSet集合中

3）前面stage中task的输出结果数据，是后面stage中task输入数据

为什么要根据宽依赖划分Stage?

问过的一些公司：陌陌

参考答案：

DAG（Directed Acyclic Graph）有向无环图是由点和线组成的拓扑图形，该图形具有方向，不会闭环。原始的RDD通过一系列的转换就形成了DAG，根据RDD之间的依赖关系的不同将DAG划分成不同的Stage，对于窄依赖，partition的转换处理在Stage中完成计算。对于宽依赖，由于有Shuffle的存在，只能在parent RDD处理完成后，才能开始接下来的计算，因此宽依赖是划分Stage的依据。

为什么要划分Stage

问过的一些公司：陌陌(2021.07)，字节(2021.10)

参考答案：

由于一个job任务中可能有大量的宽窄依赖，窄依赖不会产生shuffle，宽依赖会产生shuffle。后期划分完stage之后，在同一个stage中只有窄依赖，并没有宽依赖，这些窄依赖对应的task就可以相互独立的取运行。划分完stage之后，它内部是有很多可以并行运行task。

Stage的数量等于什么

问过的一些公司：端点数据(2021.07)

参考答案：

等于宽依赖数量+1

对RDD、 DAG 和Task的理解

问过的一些公司：美团

参考答案：

1、RDD

RDD（Resilient Distributed Dataset）叫做分布式数据集，是Spark中最基本的数据抽象。代码中是一个抽象类，它代表一个弹性的、不可变、可分区、里面的元素可并行计算的集合。

RDD特点：

1）弹性

存储的弹性：内存和磁盘的自动切换；

公众号：旧时光大数据

容错的弹性：数据丢失可以自动回复；

计算的弹性：计算出错重试机制；

分片的弹性：可根据需要重新分片。

2）分布式

数据存储在大数据集群的不同节点上

3）数据集

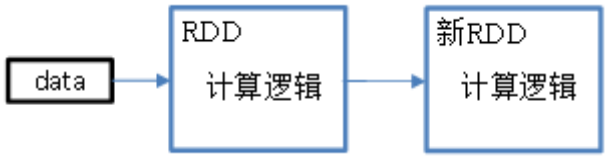
RDD封装了计算逻辑，并不保存数据

4）数据抽象

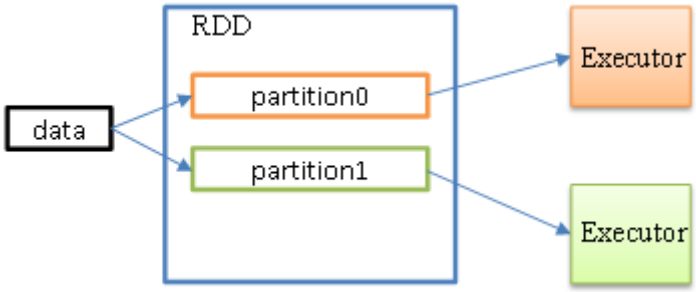
RDD是一个抽象类，需要子类具体实现

5）不可变

RDD封装了计算逻辑，是不可以改变的，想要改变，只能产生新的RDD，在新的RDD里面封装计算逻辑



6）可分区、并行计算



RDD属性：

1）一组分区（Partition），即是数据集的基本组成单位；

2）一个计算每个分区的函数；

3）RDD之间的依赖关系；

4）一个Partitioner，即RDD的分片函数；控制分区的数据流向（键值对）；

5）一个列表，存储存取每个Partition的优先位置（preferredlocation）。移动数据不如移动资源，除非资源不够。

2、DAG

DAG（Directed Acyclic Graph）有向无环图是由点和线组成的拓扑图形，该图形具有方向，不会闭环。原始的RDD通过一系列的转换就形成了DAG，根据RDD之间的依赖关系的不同将DAG划分成不同的Stage，对于窄依赖，partition的转换处理在Stage中完成计算。对于宽依赖，由于有Shuffle的存在，只能在parent RDD处理完成后，才能开始接下来的计算，宽依赖是划分Stage的依据。

3、Task

在RDD中，RDD任务切分中间分为：Application、Job、Stage和Task。Task是stage下的一个任务执行单元，一般来说，一个rdd有多少个partition，就会有多少个task，因为每一个task只是处理一个partition上的数据。

公众号：旧时光大数据

一个Stage阶段中，最后一个RDD的分区个数就是Task的个数。

每个executor执行的task的数目， 可以由submit时，--num-executors(on yarn) 来指定。

DAG为什么适合Spark?

问过的一些公司：美团

参考答案：

DAG，全称 Directed Acyclic Graph， 中文为：有向无环图。在 Spark 中， 使用 DAG 来描述我们的计算逻辑。

1、什么是DAG

DAG 是一组顶点和边的组合。顶点代表了 RDD， 边代表了对 RDD 的一系列操作。

DAG Scheduler 会根据 RDD 的 transformation 动作，将 DAG 分为不同的 stage，每个 stage 中分为多个 task，这些 task 可以并行运行。

2、DAG解决了什么问题

DAG 的出现主要是为了解决 Hadoop MapReduce 框架的局限性。

MapReduce的局限性主要有两个：

- 每个MapReduce操作都是相互独立的，Hadoop不知道接下来会有哪些Map Reduce。
- 每一步的输出结果，都会持久化到硬盘或者HDFS上。

当以上两个特点结合之后，我们就可以想象，如果在某些迭代的场景下，MapReduce 框架会对硬盘和 HDFS 的读写造成大量浪费。

而且每一步都是堵塞在上一步中，所以当我们处理复杂计算时，会需要很长时间，但是数据量却不大。

所以**Spark 中引入了 DAG，它可以优化计算计划，比如减少 shuffle 数据。**

介绍下Spark的DAG以及它的生成过程

可回答：介绍DAG有向无环图

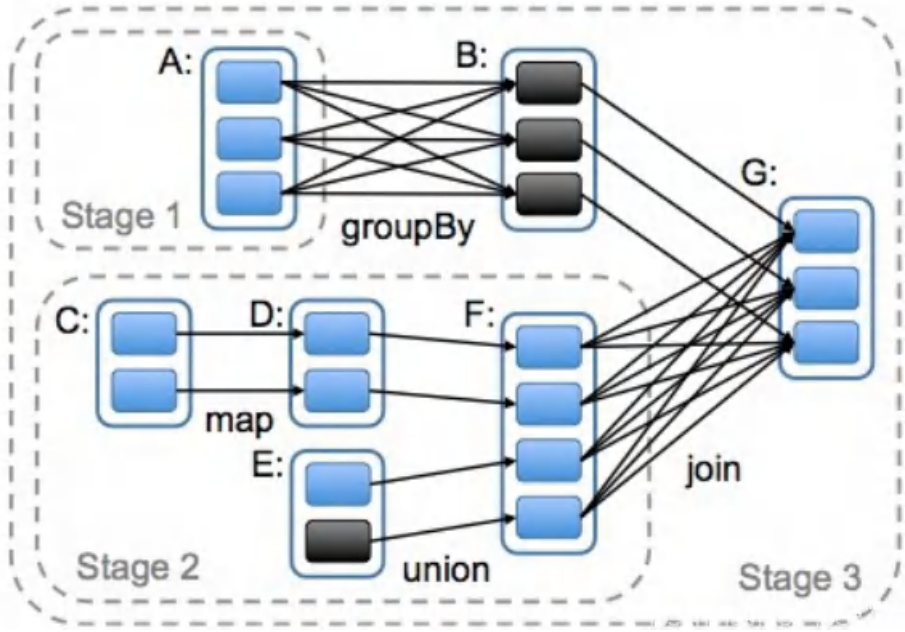
问过的一些公司：字节，美团，端点数据(2021.07)

参考答案：

DAG(Directed Acyclic Graph)叫做有向无环图，原始的RDD通过一系列的转换就形成了DAG，根据RDD之间依赖关系的不同将DAG划分成不同的Stage(调度阶段)。

对于窄依赖，partition的转换处理在一个Stage中完成计算。

对于宽依赖，由于有Shuffle的存在，只能在parent RDD处理完成后，才能开始接下来的计算，因此宽依赖是划分Stage的依据。



DAG的边界：

- 开始：通过SparkContext创建的RDD
- 触发Action，**一旦触发Action就形成了一个完整的DAG**

一个Spark的Application应用中一个或者多个DAG(也就是一个Job)，取决于触发了多少次Action。

在Spark中，DAG生成的流程关键在于回溯，在程序提交后，高层调度器将所有的RDD看成是一个Stage，然后对此Stage进行从后往前的回溯，遇到Shuffle就断开，遇到窄依赖，则归并到同一个Stage。等到所有的步骤回溯完成，便生成一个DAG图。

DAGScheduler如何划分？干了什么活?

问过的一些公司：字节，快手

参考答案：

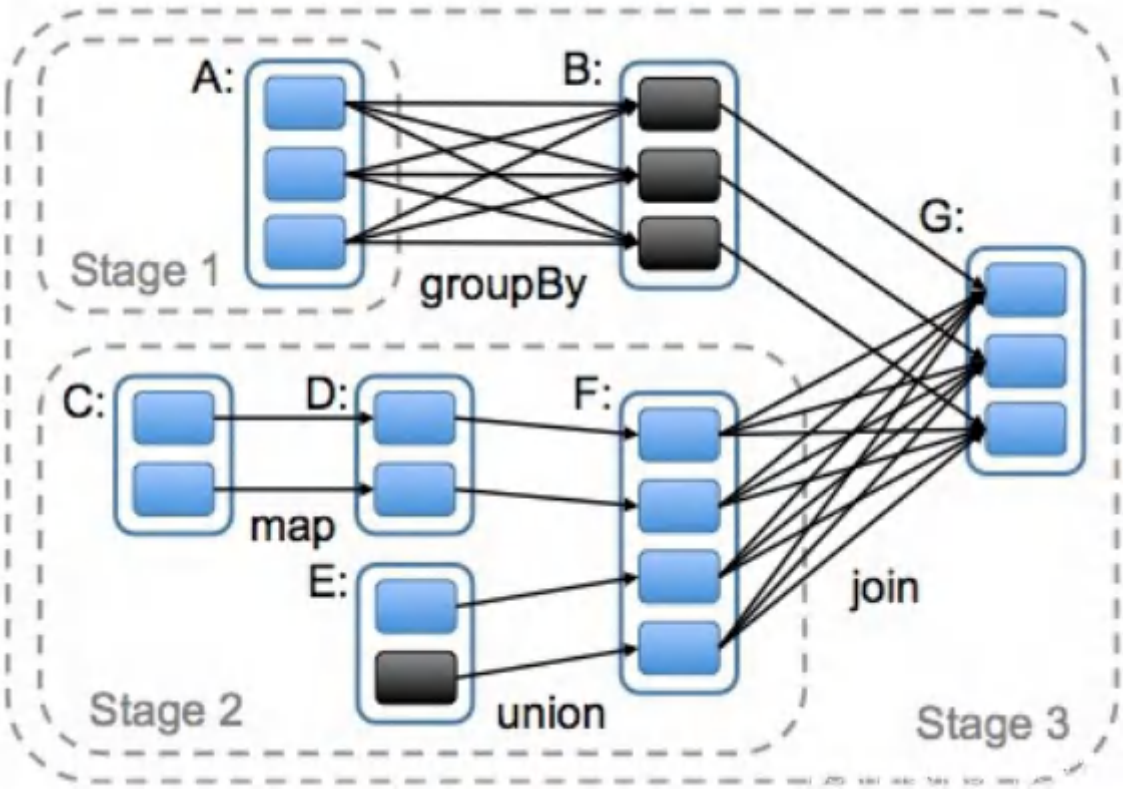
DAGScheduler是面向Stage的高层级的调度器，**DAGScheduler把DAG拆分成很多的Tasks**，每组的Tasks都是一个Stage，解析时是以Shuffle为边界反向解析构建Stage，每当遇到Shuffle，就会产生新的Stage，然后以一个个TaskSet（每个Stage封装一个TaskSet）的形式提交给底层调度器TaskScheduler。
DAGScheduler需要记录哪些RDD被存入磁盘等物化动作，同时要寻求Task的最优化调度，如在Stage内部数据的本地性等。DAGScheduler还需要监视因为Shuffle跨节点输出可能导致的失败，如果发现这个Stage失败，可能就要重新提交该Stage。

在Spark源码中，**DAGScheduler是整个Spark Application的入口，即在SparkContext中声明并实例化**。在实例化DAGScheduler之前，已经实例化了SchedulerBackend和底层调度器TaskScheduler，而SchedulerBackend和TaskScheduler是通过SparkContext的方法createTaskScheduler实例化的。
DAGScheduler在提交TaskSet给底层调度器的时候是面向TaskScheduler接口的，这符合面向对象中依赖抽象，而不依赖具体实现的原则，带来底层资源调度器的可插拔性，以至于Spark可以运行在众多的部署模式上，如Standalone、Yarn、Mesos、Local及其他自定义的部署模式。

DAGScheduler划分Stage的原理

Spark将数据在分布式环境下分区，然后将作业转化为DAG，并分阶段进行DAG的调度和任务的分布式并行处理。DAG将调度提交给DAGScheduler，DAGScheduler调度时会根据是否需要经过Shuffle过程将Job划分为多个Stage。

DAG划分Stage及Stage并行计算示意图如下图所示。



其中，实线圆角方框标识的是RDD，方框中的矩形块为RDD的分区。

在上图中，RDD A到RDD B之间，以及RDD F到RDD G之间的数据需要经过Shuffle过程，因此RDD A和RDD F分别是Stage 1跟Stage 3和Stage 2跟Stage 3的划分点。而RDD B到RDD G之间，以及RDD C到RDD D到RDD F和RDD E到RDD F之间的数据不需要经过Shuffle过程，因此，RDD G和RDD B的依赖是窄依赖，RDD B和RDD G划分到同一个Stage 3，RDD F和RDD D和RDD E的依赖以及RDD D和RDD C的依赖是窄依赖，RDD C、RDD D、RDD E和RDD F划分到同一个Stage 2。Stage 1和Stage 2是相互独立的，可以并发执行。而由于Stage 3依赖Stage 1和Stage 2的计算结果，所以Stage 3最后执行计算。

Spark容错机制?

可回答： Spark容错方法?

问过的一些公司：头条， 字节，阿里 x 3，腾讯，竞技世界

参考答案：

1、容错方式

容错指的是一个系统在部分模块出现故障时还能否持续的对外提供服务，一个高可用的系统应该具有很高的容错性；对于一个大的集群系统来说，机器故障、网络异常等都是很常见的，Spark这样的大型分布式计算集群提供了很多的容错机制来提高整个系统的可用性。

一般来说，分布式数据集的容错性有两种方式：**数据检查点和记录数据的更新。**

面向大规模数据分析，数据检查点操作成本很高，需要通过数据中心的网络连接在机器之间复制庞大的数据集，而网络带宽往往比内存带宽低得多，同时还需要消耗更多的存储资源。

因此，**Spark选择记录更新的方式。**但是，如果更新粒度太细太多，那么记录更新成本也不低。因此，RDD只支持粗粒度转换，即只记录单个块上执行的单个操作，然后将创建RDD的一系列变换序列（每个RDD都包含了他是如何由其他RDD变换过来的以及如何重建某一块数据的信息。因此RDD的容错机制又称“血统(Lineage)”容错）记录下来，以便恢复丢失的分区。

Lineage本质上很类似于数据库中的重做日志（Redo Log），只不过这个重做日志粒度很大，是对全局数据做同样的重做进而恢复数据。

公众号：旧时光大数据

2、Lineage机制

Lineage简介

相比其他系统的细颗粒度的内存数据更新级别的备份或者LOG机制，RDD的Lineage记录的是粗颗粒度的特定数据Transformation操作（如filter、map、join等）行为。当这个RDD的部分分区数据丢失时，它可以通过Lineage获取足够的信息来重新运算和恢复丢失的数据分区。因为这种粗颗粒的数据模型，限制了Spark的运用场合，所以Spark并不适用于所有高性能要求的场景，但同时相比细颗粒度的数据模型，也带来了性能的提升。

两种依赖关系

RDD在Lineage依赖方面分为两种：窄依赖(Narrow Dependencies)与宽依赖(Wide Dependencies，源码中称为Shuffle
Dependencies)，用来解决数据容错的高效性。

窄依赖是指父RDD的每一个分区最多被一个子RDD的分区所用，表现为一个父RDD的分区对应于一个子RDD的分区
或多个父RDD的分区对应于一个子RDD的分区，也就是说一个父RDD的一个分区不可能对应一个子RDD的多个分区。 1个父RDD分区对应1个子RDD分区，这其中又分两种情况：1个子RDD分区对应1个父RDD分区（如map、filter等算子），1个子RDD分区对应N个父RDD分区（如co-paritioned（协同划分）过的Join）。

宽依赖是指子RDD的分区依赖于父RDD的多个分区或所有分区，即存在一个父RDD的一个分区对应一个子RDD的多个分区。 1个父RDD分区对应多个子RDD分区，这其中又分两种情况：1个父RDD对应所有子RDD分区（未经协同划分的Join）或者1个父RDD对应非全部的多个RDD分区（如groupByKey）。

Spark依赖的实现：

```
abstract class NarrowDependency[T](_rdd: RDD[T]) extends Dependency[T] {
    //返回子RDD的partitionId依赖的所有的parent RDD的Partition(s)
    def getParents(partitionId: Int): Seq[Int]
    override def rdd: RDD[T] = _rdd
}

```

1）窄依赖是有两种具体实现，分别如下：

一种是一对一的依赖，即OneToOneDependency

```
class OneToOneDependency[T](rdd: RDD[T]) extends NarrowDependency[T](rdd) {
    override def getParents(partitionId: Int) = List(partitionId)
}

```

通过getParents的实现不难看出，RDD仅仅依赖于parent RDD相同ID的Partition。

还有一个是范围的依赖，即RangeDependency，它仅仅被org.apache.spark.rdd.UnionRDD使用。UnionRDD是把多个RDD合成一个RDD，这些RDD是被拼接而成，即每个parent RDD的Partition的相对顺序不会变，只不过每个parent RDD在UnionRDD中的Partition的起始位置不同。因此它的getPartents如下：

```
override def getParents(partitionId: Int) = {
    if(partitionId >= outStart && partitionId < outStart + length) {
       List(partitionId - outStart + inStart)
    } else {
       Nil
    }
}

```

公众号：旧时光大数据

其中，inStart是parent RDD中Partition的起始位置，outStart是在UnionRDD中的起始位置，length就是parent RDD中Partition的数量。

2）宽依赖的实现

宽依赖的实现只有一种：ShuffleDependency。子RDD依赖于parent RDD的所有Partition，因此需要Shuffle过程：

```
class ShuffleDependency[K, V, C](
    @transient _rdd: RDD[_ <: Product2[K, V]],
    val partitioner: Partitioner,
    val serializer: Option[Serializer] = None,
    val keyOrdering: Option[Ordering[K]] = None,
    val aggregator: Option[Aggregator[K, V, C]] = None,
    val mapSideCombine: Boolean = false)
extends Dependency[Product2[K, V]] {

override def rdd = _rdd.asInstanceOf[RDD[Product2[K, V]]]
//获取新的shuffleId
val shuffleId: Int = _rdd.context.newShuffleId()
//向ShuffleManager注册Shuffle的信息
val shuffleHandle: ShuffleHandle =
_rdd.context.env.shuffleManager.registerShuffle(
    shuffleId, _rdd.partitions.size, this)

    _rdd.sparkContext.cleaner.foreach(_.registerShuffleForCleanup(this))
}

```

注意：宽依赖支持两种Shuffle Manager。 即org.apache.spark.shuffle.hash.HashShuffleManager（基于Hash的Shuffle机制）和org.apache.spark.shuffle.sort.SortShuffleManager（基于排序的Shuffle机制）。

本质理解：根据父RDD分区是对应1个还是多个子RDD分区来区分窄依赖（父分区对应一个子分区）和宽依赖（父分区对应多个子分区）。如果对应多个，则当容错重算分区时，因为父分区数据只有一部分是需要重算子分区的，其余数据重算就造成了冗余计算。

对于宽依赖，Stage计算的输入和输出在不同的节点上，对于输入节点完好，而输出节点死机的情况，通过重新计算恢复数据这种情况下，这种方法容错是有效的，否则无效，因为无法重试，需要向上追溯其祖先看是否可以重试（这就是lineage，血统的意思），窄依赖对于数据的重算开销要远小于宽依赖的数据重算开销。

窄依赖和宽依赖的概念主要用在两个地方：一个是容错中相当于Redo日志的功能；另一个是在调度中构建DAG作为不同Stage的划分点。

依赖关系的特性

第一，窄依赖可以在某个计算节点上直接通过计算父RDD的某块数据计算得到子RDD对应的某块数据；宽依赖则要等到父RDD所有数据都计算完成之后，并且父RDD的计算结果进行hash并传到对应节点上之后才能计算子RDD。
第二，数据丢失时，对于窄依赖只需要重新计算丢失的那一块数据来恢复；对于宽依赖则要将祖先RDD中的所有数据块全部重新计算来恢复。所以在长“血统”链特别是有宽依赖的时候，需要在适当的时机设置数据检查点。也是这两个特性要求对于不同依赖关系要采取不同的任务调度机制和容错恢复机制。

容错原理

在容错机制中，如果一个节点死机了，而且运算窄依赖，则只要把丢失的父RDD分区重算即可，不依赖于其他节点。而宽依赖需要父RDD的所有分区都存在，重算就很昂贵了。可以这样理解开销的经济与否：在窄依赖中，在子RDD的分区丢失、重算父RDD分区时，父RDD相应分区的所有数据都是子RDD分区的数据，并不存在冗余计算。在宽依赖情况下，丢失一个子RDD分区重算的每个父RDD的每个分区的所

公众号：旧时光大数据

有数据并不是都给丢失的子RDD分区用的，会有一部分数据相当于对应的是未丢失的子RDD分区中需要的数据，这样就会产生冗余计算开销，这也是宽依赖开销更大的原因。

3、Checkpoint机制

checkpoint就是把内存中的变化刷新到持久存储，斩断依赖链在存储中 checkpoint 是一个很常见的概念， 举几个例子：

- 数据库 checkpoint 过程中一般把内存中的变化进行持久化到物理页， 这时候就可以斩断依赖链，就可以把 redo 日志删掉了， 然后更新下检查点。
- hdfs namenode 的元数据 editlog， Secondary namenode 会把 edit log 应用到 fsimage， 然后刷到磁盘上， 也相当于做了一次 checkpoint， 就可以把老的 edit log 删除了。
- spark streaming 中对于一些 有状态的操作， 这在某些 stateful 转换中是需要的，在这种转换中，生成 RDD 需要依赖前面的 batches，会导致依赖链随着时间而变长。为了避免这种没有尽头的变长，要定期将中间生成的 RDDs 保存到可靠存储来切断依赖链， 必须隔一段时间进行一次进行一次 checkpoint。

cache 和 checkpoint 是有显著区别的， 缓存把 RDD 计算出来然后放在内存中， 但是RDD 的依赖链（相当于数据库中的redo 日志）， 也不能丢掉， 当某个点某个 executor 宕了， 上面cache 的RDD就会丢掉， 需要通过 依赖链重放计算出来， 不同的是， checkpoint 是把 RDD 保存在 HDFS中， 是多副本可靠存储，所以依赖链就可以丢掉了，就斩断了依赖链， 是通过复制实现的高容错。但是有一点要注意，因为checkpoint是需要把 job 重新从头算一遍， 最好先cache一下， checkpoint就可以直接保存缓存中的 RDD 了， 就不需要重头计算一遍了， 对性能有极大的提升。

checkpoint 的正确使用姿势

```
val data = sc.textFile("/tmp/spark/1.data").cache() // 注意要cache
sc.setCheckpointDir("/tmp/spark/checkpoint")
data.checkpoint
data.count

```

使用很简单，就是设置一下 checkpoint 目录，然后再rdd上调用 checkpoint 方法，action 的时候就对数据进行了 checkpoint。

checkpoint写流程

RDD checkpoint 过程中会经过以下几个状态

```
Initialized -> marked for checkpointing -> checkpointing in progress ->
checkpointed

```

我们看下状态转换流程

- 首先 driver program 需要使用 rdd.checkpoint() 去设定哪些 rdd 需要 checkpoint，设定后，该 rdd 就接受 RDDCheckpointData 管理。用户还要设定 checkpoint 的存储路径，一般在 HDFS 上。
- marked for checkpointing：初始化后，RDDCheckpointData 会将 rdd 标记为 MarkedForCheckpoint。
- checkpointing in progress：每个 job 运行结束后会调用 finalRdd.doCheckpoint()，finalRdd 会顺着 computing chain 回溯扫描，碰到要 checkpoint 的 RDD 就将其标记为 CheckpointingInProgress，然后将写磁盘（比如写 HDFS）需要的配置文件（如 core-site.xml 等）broadcast 到其他 worker 节点上的 blockManager。完成以后，启动一个 job 来完成 checkpoint（使用 rdd.context.runJob(rdd, CheckpointRDD.writeToFile(path.toString, broadcastedConf))）。
- checkpointed：job 完成 checkpoint 后，将该 rdd 的 dependency 全部清掉，并设定该 rdd 状态为 checkpointed。然后，为该 rdd 强加一个依赖，设置该 rdd 的 parent rdd 为 CheckpointRDD，该 CheckpointRDD 负责以后读取在文件系统上的 checkpoint 文件，生成该 rdd 的 partition。

checkpoint读流程

如果一个RDD 我们已经 checkpoint了那么是什么时候用呢，checkpoint 将 RDD 持久化到 HDFS 或本地文件夹，如果不被手动 remove 掉，是一直存在的，也就是说可以被下一个 driver program 使用。 比如 spark streaming 挂掉了，重启后就可以使用之前 checkpoint 的数据进行 recover （这个流程我们在下面一篇文章会讲到） ， 当然在同一个 driver program 也可以使用。 我们讲下在同一个 driver program 中是怎么使用 checkpoint 数据的。

如果 一个 RDD 被checkpoint了，如果这个 RDD 上有 action 操作时候，或者回溯的这个 RDD 的时候,这个 RDD 进行计算的时候，里面判断如果已经 checkpoint 过, 对分区和依赖的处理都是使用的 RDD 内部的 checkpointRDD 变量。

具体细节如下：

如果一个RDD被checkpoint了，那么这个 RDD 中对分区和依赖的处理都是使用的RDD内部的 checkpointRDD变量，具体实现是 ReliableCheckpointRDD 类型。这个是在 checkpoint 写流程中创建的。依赖和获取分区方法中先判断是否已经checkpoint，如果已经checkpoint了，就斩断依赖，使用 ReliableCheckpointRDD，来处理依赖和获取分区。

如果没有，才往前回溯依赖。依赖就是没有依赖，因为已经斩断了依赖，获取分区数据就是读取 checkpoint 到 hdfs目录中不同分区保存下来的文件。

整个 checkpoint 读流程就完了。

在以下两种情况下，RDD需要加检查点。

- DAG中的Lineage过长，如果重算，则开销太大（如在PageRank中）。
- 在宽依赖上做Checkpoint获得的收益更大。

由于RDD是只读的，所以Spark的RDD计算中一致性不是主要关心的内容，内存相对容易管理，这也是设计者很有远见的地方，这样减少了框架的复杂性，提升了性能和可扩展性，为以后上层框架的丰富奠定了强有力的基础。

在RDD计算中，通过检查点机制进行容错，传统做检查点有两种方式：通过冗余数据和日志记录更新操作。在RDD中的doCheckPoint方法相当于通过冗余数据来缓存数据，而之前介绍的血统就是通过相当粗粒度的记录更新操作来实现容错的。

检查点（本质是通过将RDD写入Disk做检查点）是为了通过lineage做容错的辅助，lineage过长会造成容错成本过高，这样就不如在中间阶段做检查点容错，如果之后有节点出现问题而丢失分区，从做检查点的RDD开始重做Lineage，就会减少开销。

RDD的容错

问过的一些公司：端点数据(2021.07)

参考答案：

1、Lineage机制（简称血统机制）

RDD的Lineage机制是将粗颗粒度相同的操作（map flatmap filter之类的）记录下来，万一RDD信息丢失可通过Lineage获取信息从新获得原来丢失的RDD

缺点：lineage如果过长 影响性能，开销太大

2、checkpoint机制

将RDD的结果存入HDFS，以文件的形式存在

3、cache机制

将结果存入内存

公众号：旧时光大数据

4、persist机制

将结果存入磁盘

checkpoint 和 cache 和 persist的异同

相同点：都要在action算子之后触发

不同点：

checkpoint：

- 是存在HDFS上的，更加安全可靠，并且checkpoint会改变依赖关系，并且之后会触发一个job操作

persist和cache

- 不改变依赖关系，运行完成后缓存数据自动消失

Executor内存分配?

可回答：Spark的内存管理机制

问过的一些公司：流利说，陌陌，商汤科技，阿里

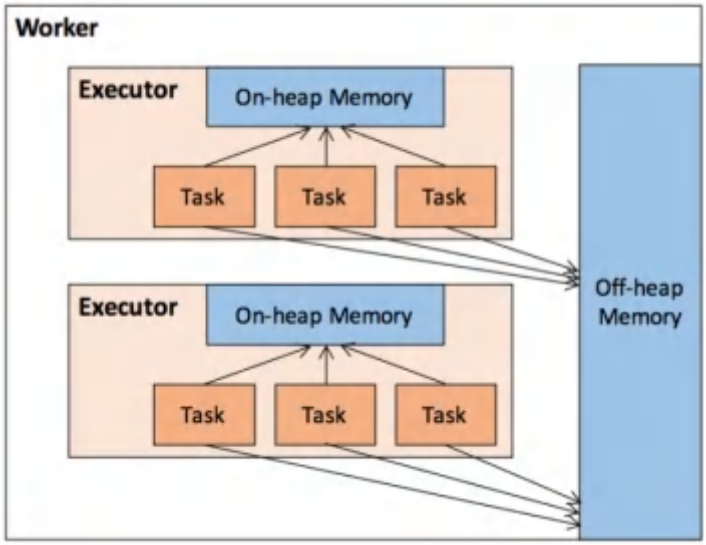
参考答案：

在执行Spark 的应用程序时，Spark 集群会启动 Driver 和 Executor 两种 JVM 进程，前者为主控进程，负责创建 Spark 上下文，提交 Spark 作业（Job），并将作业转化为计算任务（Task），在各个 Executor 进程间协调任务的调度，后者负责在工作节点上执行具体的计算任务，并将结果返回给 Driver，同时为需要持久化的 RDD 提供存储功能。**下方内容中的 Spark 内存均特指 Executor 的内存。**

1、堆内和堆外内存规划

作为一个 JVM 进程，Executor 的内存管理建立在 JVM 的内存管理之上，Spark 对 JVM 的堆内（On-heap）空间进行了更为详细的分配，以充分利用内存。同时，Spark 引入了堆外（Off-heap）内存，使之可以直接在工作节点的系统内存中开辟空间，进一步优化了内存的使用。

堆内内存受到JVM统一管理，堆外内存是直接向操作系统进行内存的申请和释放。



1）堆内内存

公众号：旧时光大数据

堆内内存的大小，由 Spark 应用程序启动时的 –executor-memory 或 spark.executor.memory 参数配置。Executor 内运行的并发任务共享 JVM 堆内内存，这些任务在缓存 RDD 数据和广播（Broadcast）数据时占用的内存被规划为存储（Storage）内存，而这些任务在执行 Shuffle 时占用的内存被规划为执行（Execution）内存，剩余的部分不做特殊规划，那些 Spark 内部的对象实例，或者用户定义的 Spark 应用程序中的对象实例，均占用剩余的空间。不同的管理模式下，这三部分占用的空间大小各不相同。

Spark 对堆内内存的管理是一种逻辑上的"**规划式**"的管理，因为对象实例占用内存的申请和释放都由 JVM 完成，Spark 只能在申请后和释放前记录这些内存，我们来看其具体流程：

申请内存流程如下：

- Spark 在代码中 new 一个对象实例；
- JVM 从堆内内存分配空间，创建对象并返回对象引用；
- Spark 保存该对象的引用，记录该对象占用的内存。

释放内存流程如下：

- Spark记录该对象释放的内存，删除该对象的引用；
- 等待JVM的垃圾回收机制释放该对象占用的堆内内存。

我们知道，JVM 的对象可以以序列化的方式存储，序列化的过程是将对象转换为二进制字节流，本质上可以理解为将非连续空间的链式存储转化为连续空间或块存储，在访问时则需要进行序列化的逆过程——反序列化，将字节流转化为对象，序列化的方式可以节省存储空间，但增加了存储和读取时候的计算开销。

对于 Spark 中序列化的对象，由于是字节流的形式，其占用的内存大小可直接计算，而对于非序列化的对象，其占用的内存是通过周期性地采样近似估算而得，即并不是每次新增的数据项都会计算一次占用的内存大小，这种方法降低了时间开销但是有可能误差较大，导致某一时刻的实际内存有可能远远超出预期。此外，在被 Spark 标记为释放的对象实例，很有可能在实际上并没有被 JVM 回收，导致实际可用的内存小于 Spark 记录的可用内存。所以 Spark 并不能准确记录实际可用的堆内内存，从而也就无法完全避免内存溢出（OOM, Out of Memory）的异常。

虽然不能精准控制堆内内存的申请和释放，但 Spark 通过对存储内存和执行内存各自独立的规划管理，可以决定是否要在存储内存里缓存新的 RDD，以及是否为新的任务分配执行内存，在一定程度上可以提升内存的利用率，减少异常的出现。

2）堆外内存

为了进一步优化内存的使用以及提高 Shuffle 时排序的效率，Spark 引入了堆外（Off-heap）内存，使之可以直接在工作节点的系统内存中开辟空间，存储经过序列化的二进制数据。

堆外内存意味着把内存对象分配在Java虚拟机的堆以外的内存，这些内存直接受操作系统管理（而不是虚拟机）。这样做的结果就是能保持一个较小的堆，以减少垃圾收集对应用的影响。

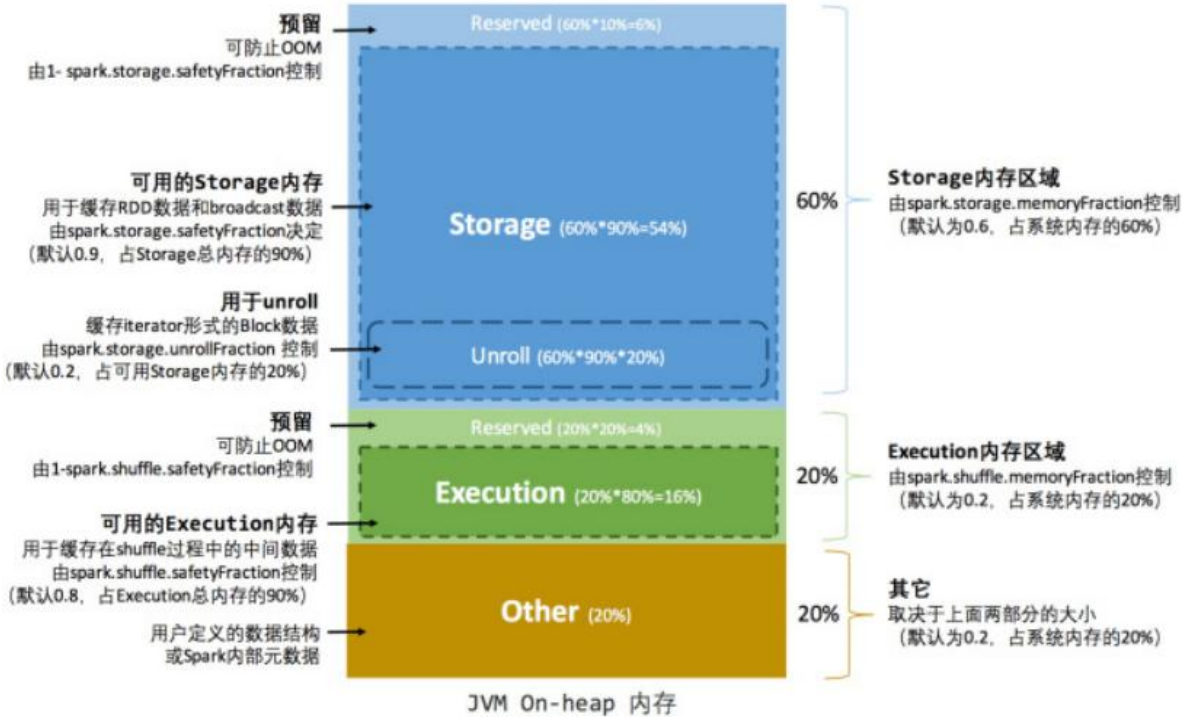
利用 JDK Unsafe API（从 Spark 2.0 开始，在管理堆外的存储内存时不再基于 Tachyon，而是与堆外的执行内存一样，基于 JDK Unsafe API 实现），**Spark 可以直接操作系统堆外内存，减少了不必要的内存开销，以及频繁的 GC 扫描和回收，提升了处理性能。**堆外内存可以被精确地申请和释放（堆外内存之所以能够被精确的申请和释放，是由于内存的申请和释放不再通过JVM机制，而是直接向操作系统申请，JVM对于内存的清理是无法准确指定时间点的，因此无法实现精确的释放），**而且序列化的数据占用的空间可以被精确计算，所以相比堆内内存来说降低了管理的难度，也降低了误差。**

在默认情况下堆外内存并不启用，可通过配置 spark.memory.offHeap.enabled 参数启用，并由 spark.memory.offHeap.size 参数设定堆外空间的大小。**除了没有 other 空间，堆外内存与堆内内存的划分方式相同，所有运行中的并发任务共享存储内存和执行内存。**

2、内存空间分配

1）静态内存管理

在 Spark 最初采用的静态内存管理机制下，存储内存、执行内存和其他内存的大小在 Spark 应用程序运行期间均为固定的，但用户可以应用程序启动前进行配置，堆内内存的分配如下图所示：



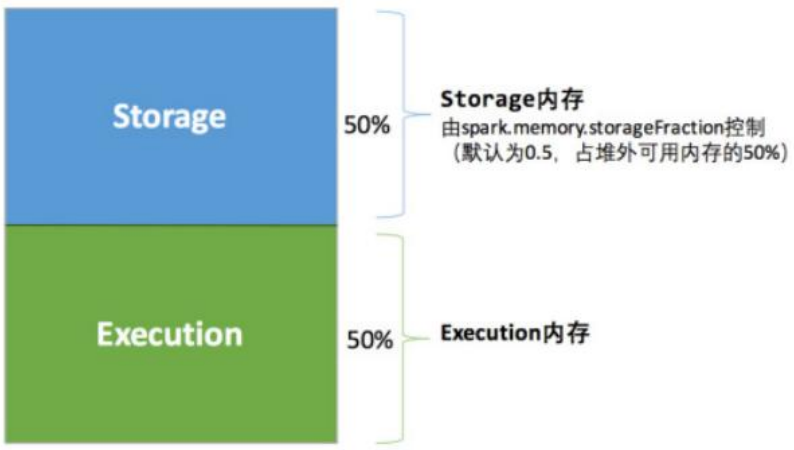
可以看到，可用的堆内内存的大小需要按照下方代码清单的方式计算：

```
1    可用的存储内存 = systemMaxMemory * spark.storage.memoryFraction *
     spark.storage.safety Fraction
2    可用的执行内存 = systemMaxMemory * spark.shuffle.memoryFraction *
     spark.shuffle.safety Fraction
3
```

其中 systemMaxMemory 取决于当前 JVM 堆内内存的大小，最后可用的执行内存或者存储内存要在此基础上与各自的 memoryFraction 参数和 safetyFraction 参数相乘得出。上述计算公式中的两个 safetyFraction 参数，其意义在于在逻辑上预留出 1-safetyFraction 这么一块保险区域，降低因实际内存超出当前预设范围而导致 OOM 的风险（上文提到，对于非序列化对象的内存采样估算会产生误差）。值得注意的是，这个预留的保险区域仅仅是一种逻辑上的规划，在具体使用时 Spark 并没有区别对待，和”其它内存”一样交给了 JVM 去管理。

Storage内存和Execution内存都有预留空间，目的是防止OOM，因为Spark堆内内存大小的记录是不准确的，需要留出保险区域。

堆外的空间分配较为简单，只有存储内存和执行内存，如下入所示。可用的执行内存和存储内存占用的空间大小直接由参数spark.memory.storageFraction 决定，由于堆外内存占用的空间可以被精确计算，所以无需再设定保险区域。

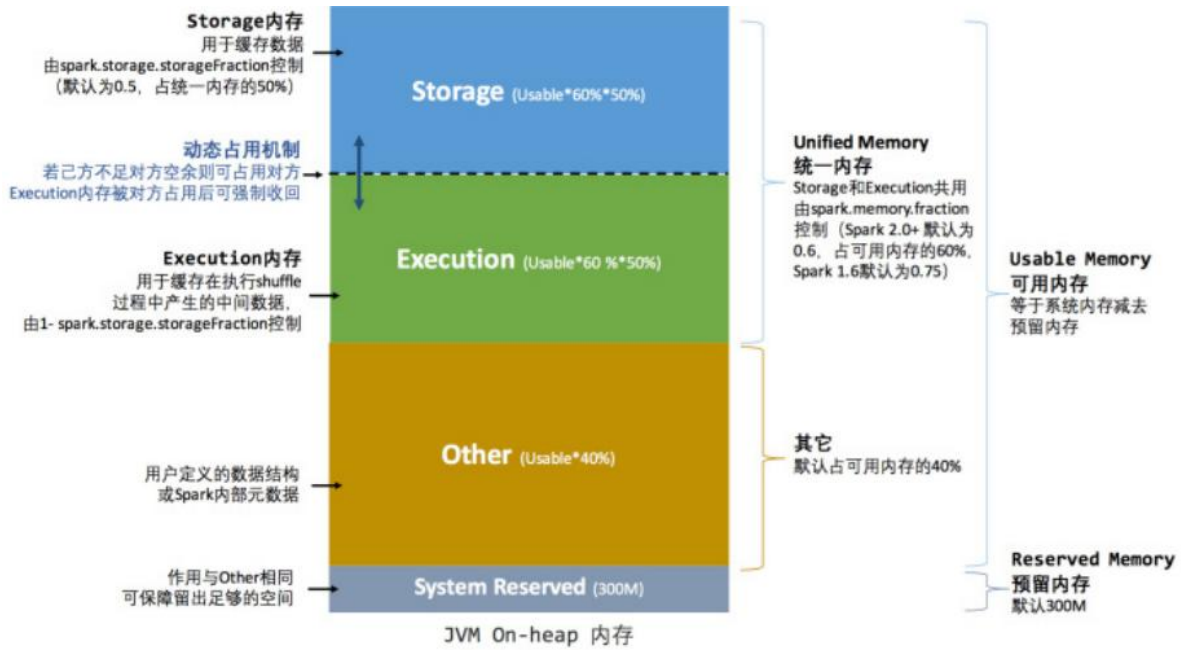


公众号：旧时光大数据

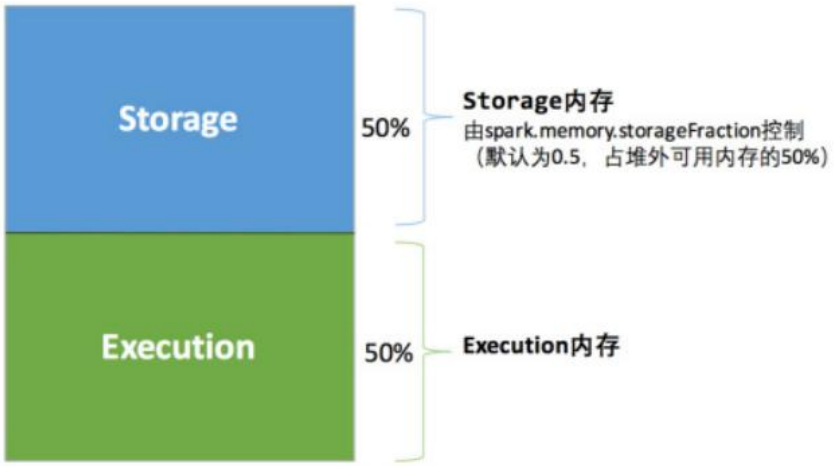
静态内存管理机制实现起来较为简单，但如果用户不熟悉 Spark 的存储机制，或没有根据具体的数据规模和计算任务或做相应的配置，很容易造成”一半海水，一半火焰”的局面，即存储内存和执行内存中的一方剩余大量的空间，而另一方却早早被占满，不得不淘汰或移出旧的内容以存储新的内容。由于新的内存管理机制的出现，这种方式目前已经很少有开发者使用，出于兼容旧版本的应用程序的目的，Spark 仍然保留了它的实现。

2）统一内存管理

Spark 1.6 之后引入的统一内存管理机制，与静态内存管理的区别在于存储内存和执行内存共享同一块空间，可以动态占用对方的空闲区域，统一内存管理的堆内内存结构如下图所示：



统一内存管理的堆外内存结构如下图所示：

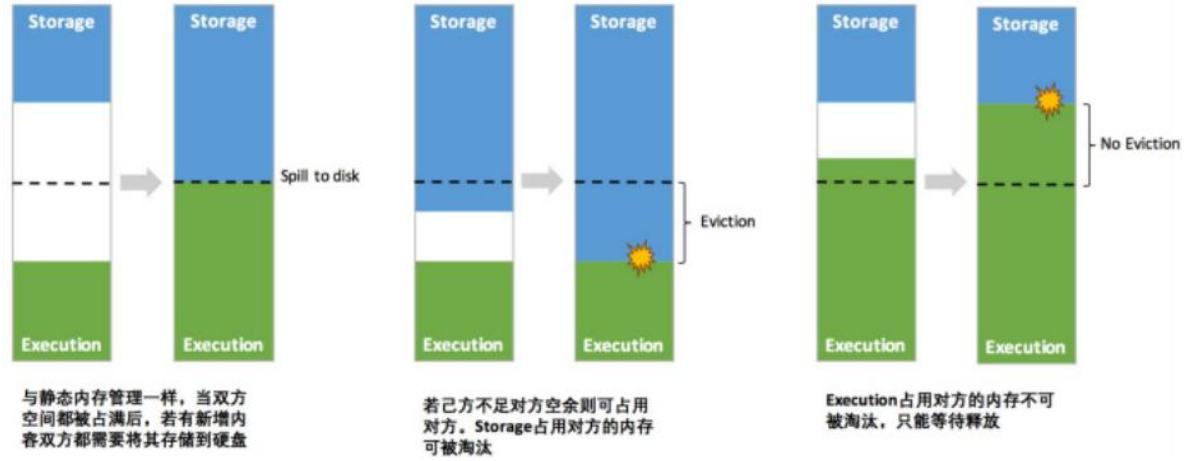


其中最重要的优化在于动态占用机制，其规则如下：

- 设定基本的存储内存和执行内存区域（spark.storage.storageFraction 参数），该设定确定了双方各自拥有的空间的范围；
- 双方的空间都不足时，则存储到硬盘；若己方空间不足而对方空余时，可借用对方的空间;（存储空间不足是指不足以放下一个完整的 Block）
- 执行内存的空间被对方占用后，可让对方将占用的部分转存到硬盘，然后”归还”借用的空间；
- 存储内存的空间被对方占用后，无法让对方”归还”，因为需要考虑 Shuffle 过程中的很多因素，实现起来较为复杂。

统一内存管理的动态占用机制如下图所示：

公众号：旧时光大数据



凭借统一内存管理机制，Spark 在一定程度上提高了堆内和堆外内存资源的利用率，降低了开发者维护 Spark 内存的难度，但并不意味着开发者可以高枕无忧。如果存储内存的空间太大或者说缓存的数据过多，反而会导致频繁的全量垃圾回收，降低任务执行时的性能，因为缓存的 RDD 数据通常都是长期驻留内存的。所以要想充分发挥 Spark 的性能，需要开发者进一步了解存储内存和执行内存各自的管理方式和实现原理。

Spark的batchsize，怎么解决小文件合并问题?

可回答：Spark小文件问题

问过的一些公司：vivo，陌陌(2021.07)

参考答案：

1、相关问题描述

当我们使用spark sql执行etl时候出现了，可能最终结果大小只有几百k，但是小文件一个分区有上千的情况。

这样就会导致以下的一些危害：

- hdfs有最大文件数限制；
- 浪费磁盘资源（可能存在空文件）；
- hive中进行统计,计算的时候,会产生很多个map,影响计算的速度。

2、解决方案

1） 方法一：通过spark的coalesce()方法和repartition()方法

```
1   val rdd2 = rdd1.coalesce(8, true) // true表示是否shuffle
2   val rdd3 = rdd1.repartition(8)
3
```

coalesce：coalesce()方法的作用是返回指定一个新的指定分区的Rdd，如果是生成一个窄依赖的结果，那么可以不发生shuffle，分区的数量发生激烈的变化，计算节点不足，不设置true可能会出错。

repartition：coalesce()方法shuffle为true的情况。

2）方法二：降低spark并行度，即调节spark.sql.shuffle.partitions

比如之前设置的为100，按理说应该生成的文件数为100；但是由于业务比较特殊，采用的大量的union all，且union all在spark中属于窄依赖，不会进行shuffle，所以导致最终会生成（union all数量+1）*100的文件数。如有10个union all，会生成1100个小文件。这样导致降低并行度为10之后，执行时长大大增加，且文件数依旧有110个，效果有，但是不理想。

公众号：旧时光大数据

3）方法三：新增一个并行度=1任务，专门合并小文件。

先将原来的任务数据写到一个临时分区（如tmp）；再起一个并行度为1的任务，类似：

```
insert overwrite 目标表 select * from 临时分区
```

但是结果小文件数还是没有减少，原因：‘select * from 临时分区’这个任务在spark中属于窄依赖；并且spark DAG中分为宽依赖和窄依赖，只有宽依赖会进行shuffle；故并行度shuffle，spark.sql.shuffle.partitions=1也就没有起到作用；由于数据量本身不是特别大，所以可以直接采用group by（在spark中属于宽依赖）的方式，类似：

```
insert overwrite 目标表 select * from 临时分区 group by *
```

先运行原任务，写到tmp分区，‘dfs -count’查看文件数，1100个，运行加上group by的临时任务（spark.sql.shuffle.partitions=1），查看结果目录，文件数=1，成功。

最后又加了个删除tmp分区的任务。

3、总结

1）方便的话，可以采用coalesce()方法和repartition()方法。

2）如果任务逻辑简单，数据量少，可以直接降低并行度。

3）任务逻辑复杂，数据量很大，原任务大并行度计算写到临时分区，再加两个任务：一个用来将临时分区的文件用小并行度（加宽依赖）合并成少量文件到实际分区；另一个删除临时分区。

4）hive任务减少小文件相对比较简单，可以直接设置参数，如：

Map-only的任务结束时合并小文件：

```
sethive.merge.mapfiles = true
```

在Map-Reduce的任务结束时合并小文件：

```
sethive.merge.mapredfiles= true
```

当输出文件的平均大小小于1GB时，启动一个独立的map-reduce任务进行文件merge：

```
sethive.merge.smallfiles.avgsize=1024000000
```

Spark参数（性能）调优

可回答：1）Spark有关资源分配的参数有哪些？这些参数如何自动分配？2）driver、exectuor参数优化；3）怎么优化Spark；4）Spark提交任务时有哪些参数可以调整

问过的一些公司：字节 x 4，字节(2021.10)-(2021.07)，百度，阿里 x 2，阿里(2021.10)，深信服，猿辅导，小米，京东(2021.07)，美团(2021.09)，蔚来(2021.09)，360(2021.10)，快手(2021.10)，网易严选(2021.09)

参考答案：

1、常规性能调优

公众号：旧时光大数据

1）常规性能调优一：最优资源配置

Spark性能调优的第一步，就是为任务分配更多的资源，在一定范围内，增加资源的分配与性能的提升是成正比的，实现了最优的资源配置后，在此基础上再考虑进行后面论述的性能调优策略。

资源的分配在使用脚本提交Spark任务时进行指定，标准的Spark任务提交脚本如以下代码所示：

```
/usr/opt/modules/spark/bin/spark-submit \
--class com.atguigu.spark.Analysis \
--num-executors 80 \
--driver-memory 6g \
--executor-memory 6g \
--executor-cores 3 \
/usr/opt/modules/spark/jar/spark.jar \

```

可以进行分配的资源如下表所示：

名称	说明
--num-executors	配置Executor的数量
--driver-memory	配置Driver内存（影响不大）
--executor-memory	配置每个Executor的内存大小
--executor-cores	配置每个Executor的CPU core数量

调节原则：尽量将任务分配的资源调节到可以使用的资源的最大限度。

对于具体资源的分配，我们分别讨论Spark的两种Cluster运行模式：

- 第一种是Spark Standalone模式，你在提交任务前，一定知道或者可以从运维部门获取到你可以使用的资源情况，在编写submit脚本的时候，就根据可用的资源情况进行资源的分配，比如说集群有15台机器，每台机器为8G内存，2个CPU core，那么就指定15个Executor，每个Executor分配8G内存，2个CPU core。
- 第二种是Spark Yarn模式，由于Yarn使用资源队列进行资源的分配和调度，在表写submit脚本的时候，就根据Spark作业要提交到的资源队列，进行资源的分配，比如资源队列有400G内存，100个CPU core，那么指定50个Executor，每个Executor分配8G内存，2个CPU core。

对各项资源进行了调节后，得到的性能提升会有如下表现：

公众号：旧时光大数据

名称	解析
增加Executor·个数	在资源允许的情况下，增加Executor的个数可以提高执行task的并行度。比如有4个Executor，每个Executor有2个CPU core，那么可以并行执行8个task，如果将Executor的个数增加到8个（资源允许的情况下），那么可以并行执行16个task，此时的并行能力提升了一倍。
增加每个Executor的CPU core个数	在资源允许的情况下，增加每个Executor的Cpu core个数，可以提高执行task的并行度。比如有4个Executor，每个Executor有2个CPU core，那么可以并行执行8个task，如果将每个Executor的CPU core个数增加到4个（资源允许的情况下），那么可以并行执行16个task，此时的并行能力提升了一倍。
增加每个Executor的内存量	在资源允许的情况下，增加每个Executor的内存量以后，对性能的提升有三点：1. 可以缓存更多的数据（即对RDD进行cache），写入磁盘的数据相应减少，甚至可以不写入磁盘，减少了可能的磁盘IO；2. 可以为shuffle操作提供更多内存，即有更多空间来存放reduce端拉取的数据，写入磁盘的数据相应减少，甚至可以不写入磁盘，减少了可能的磁盘IO；3. 可以为task的执行提供更多内存，在task的执行过程中可能创建很多对象，内存较小时会引发频繁的GC，增加内存后，可以避免频繁的GC，提升整体性能。

补充：生产环境Spark submit脚本配置

```
bin/spark-submit \
--class com.atguigu.spark.WordCount \
--master yarn\
--deploy-mode cluster\
--num-executors 80 \
--driver-memory 6g \
--executor-memory 6g \
--executor-cores 3 \
--queue root.default \
--conf spark.yarn.executor.memoryOverhead=2048 \
--conf spark.core.connection.ack.wait.timeout=300 \
/usr/local/spark/spark.jar

```

参数配置参考值：

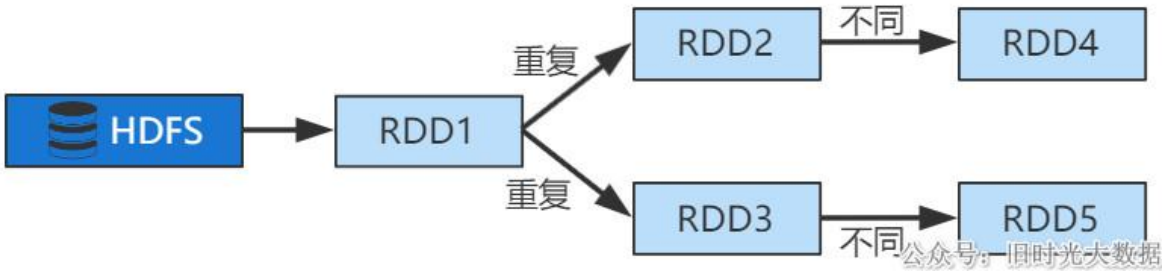
- --num-executors：50~100
- --driver-memory：1G~5G
- --executor-memory：6G~10G
- --executor-cores：3
- --master：实际生产环境一定使用yarn

2）常规性能调优二：RDD优化

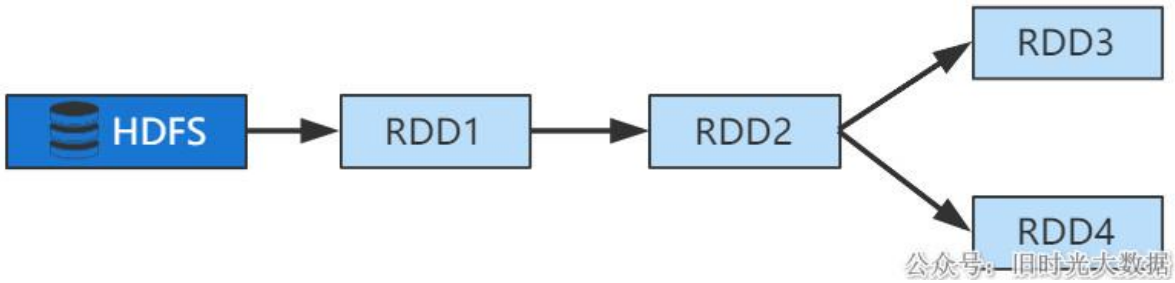
（1）RDD复用

在对RDD进行算子时，要避免相同的算子和计算逻辑之下对RDD进行重复的计算

公众号：旧时光大数据



对上图中的RDD计算架构进行修改，得到如下图所示的优化结果：



（2）RDD持久化

在Spark中，当多次对同一个RDD执行算子操作时，每一次都会对这个RDD以之前的父RDD重新计算一次，这种情况是必须要避免的，对同一个RDD的重复计算是对资源的极大浪费，因此，必须对多次使用的RDD进行持久化，通过持久化将公共RDD的数据缓存到内存/磁盘中，之后对于公共RDD的计算都会从内存/磁盘中直接获取RDD数据。

对于RDD的持久化，有两点需要说明：

- RDD的持久化是可以进行序列化的，当内存无法将RDD的数据完整的进行存放的时候，可以考虑使用序列化的方式减小数据体积，将数据完整存储在内存中。
- 如果对于数据的可靠性要求很高，并且内存充足，可以使用副本机制，对RDD数据进行持久化。当持久化启用了复本机制时，对于持久化的每个数据单元都存储一个副本，放在其他节点上面，由此实现数据的容错，一旦一个副本数据丢失，不需要重新计算，还可以使用另外一个副本。

（3）RDD尽可能早的filter操作

获取到初始RDD后，应该考虑尽早地过滤掉不需要的数据，进而减少对内存的占用，从而提升Spark作业的运行效率。

3）常规性能调优三：并行度调节

Spark作业中的并行度指各个stage的task的数量。

如果并行度设置不合理而导致并行度过低，会导致资源的极大浪费，例如，20个Executor，每个Executor分配3个CPU core，而Spark作业有40个task，这样每个Executor分配到的task个数是2个，这就使得每个Executor有一个CPU core空闲，导致资源的浪费。

理想的并行度设置，应该是让并行度与资源相匹配，简单来说就是在资源允许的前提下，并行度要设置的尽可能大，达到可以充分利用集群资源。合理的设置并行度，可以提升整个Spark作业的性能和运行速度。

Spark官方推荐，task数量应该设置为Spark作业总CPU core数量的2~3倍。之所以没有推荐task数量与CPU core总数相等，是因为task的执行时间不同，有的task执行速度快而有的task执行速度慢，如果task数量与CPU core总数相等，那么执行快的task执行完成后，会出现CPU core空闲的情况。如果task数量设置为CPU core总数的2~3倍，那么一个task执行完毕后，CPU core会立刻执行下一个task，降低了资源的浪费，同时提升了Spark作业运行的效率。

Spark作业并行度的设置如下所示：

公众号：旧时光大数据

```
val conf = new SparkConf().set("spark.default.parallelism", "500")

```

4）常规性能调优四：广播大变量

默认情况下，task中的算子中如果使用了外部的变量，每个task都会获取一份变量的复本，这就造成了内存的极大消耗。一方面，如果后续对RDD进行持久化，可能就无法将RDD数据存入内存，只能写入磁盘，磁盘IO将会严重消耗性能；另一方面，task在创建对象的时候，也许会发现堆内存无法存放新创建的对象，这就会导致频繁的GC，GC会导致工作线程停止，进而导致Spark暂停工作一段时间，严重影响Spark性能。

假设当前任务配置了20个Executor，指定500个task，有一个20M的变量被所有task共用，此时会在500个task中产生500个副本，耗费集群10G的内存，如果使用了广播变量， 那么每个Executor保存一个副本，一共消耗400M内存，内存消耗减少了5倍。

广播变量在每个Executor保存一个副本，此Executor的所有task共用此广播变量，这让变量产生的副本数量大大减少。

在初始阶段，广播变量只在Driver中有一份副本。task在运行的时候，想要使用广播变量中的数据，此时首先会在自己本地的Executor对应的BlockManager中尝试获取变量，如果本地没有，BlockManager就会从Driver或者其他节点的BlockManager上远程拉取变量的复本，并由本地的BlockManager进行管理；之后此Executor的所有task都会直接从本地的BlockManager中获取变量。

5）常规性能调优五：Kryo序列化

默认情况下，Spark使用Java的序列化机制。Java的序列化机制使用方便，不需要额外的配置，在算子中使用的变量实现Serializable接口即可，但是，Java序列化机制的效率不高，序列化速度慢并且序列化后的数据所占用的空间依然较大。

Kryo序列化机制比Java序列化机制性能提高10倍左右，Spark之所以没有默认使用Kryo作为序列化类库，是因为它不支持所有对象的序列化，同时Kryo需要用户在使用前注册需要序列化的类型，不够方便，但从Spark 2.0.0版本开始，**简单类型、简单类型数组、字符串类型的Shuffling RDDs 已经默认使用Kryo序列化方式了。**

```
public class MyKryoRegistrator implements KryoRegistrator
{
  @Override
  public void registerClasses(Kryo kryo)
  {
    kryo.register(StartupReportLogs.class);
  }
}

```

配置Kryo序列化方式的实例代码：

```
//创建SparkConf对象
val conf = new SparkConf().setMaster(…).setAppName(…)
//使用Kryo序列化库，如果要使用Java序列化库，需要把该行屏蔽掉
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
//在Kryo序列化库中注册自定义的类集合，如果要使用Java序列化库，需要把该行屏蔽掉
conf.set("spark.kryo.registrator", "ag.com.MyKryoRegistrator");

```

6）常规性能调优六：调节本地化等待时长

公众号：旧时光大数据

Spark作业运行过程中，Driver会对每一个stage的task进行分配。根据Spark的task分配算法，Spark希望task能够运行在它要计算的数据算在的节点（数据本地化思想），这样就可以避免数据的网络传输。通常来说，task可能不会被分配到它处理的数据所在的节点，因为这些节点可用的资源可能已经用尽，此时，Spark会等待一段时间，默认3s，如果等待指定时间后仍然无法在指定节点运行，那么会自动降级，尝试将task分配到比较差的本地化级别所对应的节点上，比如将task分配到离它要计算的数据比较近的一个节点，然后进行计算，如果当前级别仍然不行，那么继续降级。

当task要处理的数据不在task所在节点上时，会发生数据的传输。task会通过所在节点的BlockManager获取数据，BlockManager发现数据不在本地时，户通过网络传输组件从数据所在节点的BlockManager处获取数据。

网络传输数据的情况是我们不愿意看到的，大量的网络传输会严重影响性能，因此，我们希望通过调节本地化等待时长，如果在等待时长这段时间内，目标节点处理完成了一部分task，那么当前的task将有机会得到执行，这样就能够改善Spark作业的整体性能。

Spark的本地化等级如表所示：

名称	解析
PROCESS_LOCAL	进程本地化，task和数据在同一个Executor中，性能最好。
NODE_LOCAL	节点本地化，task和数据在同一个节点中，但是task和数据不在同一个Executor中，数据需要在进程间进行传输。
RACK_LOCAL	机架本地化，task和数据在同一个机架的两个节点上，数据需要通过网络在节点之间进行传输。
NO_PREF	对于task来说，从哪里获取都一样，没有好坏之分。
ANY	task和数据可以在集群的任何地方，而且不在一个机架中，性能最差。

在Spark项目开发阶段，可以使用client模式对程序进行测试，此时，可以在本地看到比较全的日志信息，日志信息中有明确的task数据本地化的级别，如果大部分都是PROCESS_LOCAL，那么就无需进行调节，但是如果发现很多的级别都是NODE_LOCAL、ANY，那么需要对本地化的等待时长进行调节，通过延长本地化等待时长，看看task的本地化级别有没有提升，并观察Spark作业的运行时间有没有缩短。

注意，过犹不及，不要将本地化等待时长延长地过长，导致因为大量的等待时长，使得Spark作业的运行时间反而增加了。

Spark本地化等待时长的设置如代码所示：

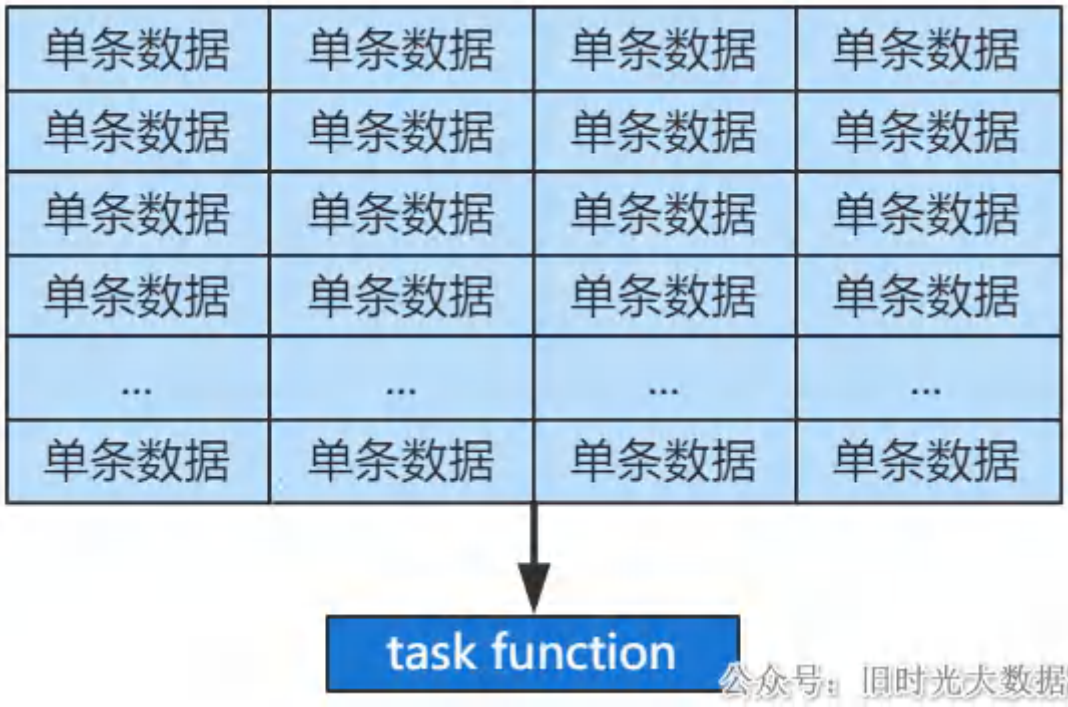
```
1    val conf = new SparkConf().set("spark.locality.wait", "6")
2
```

2、算子调优

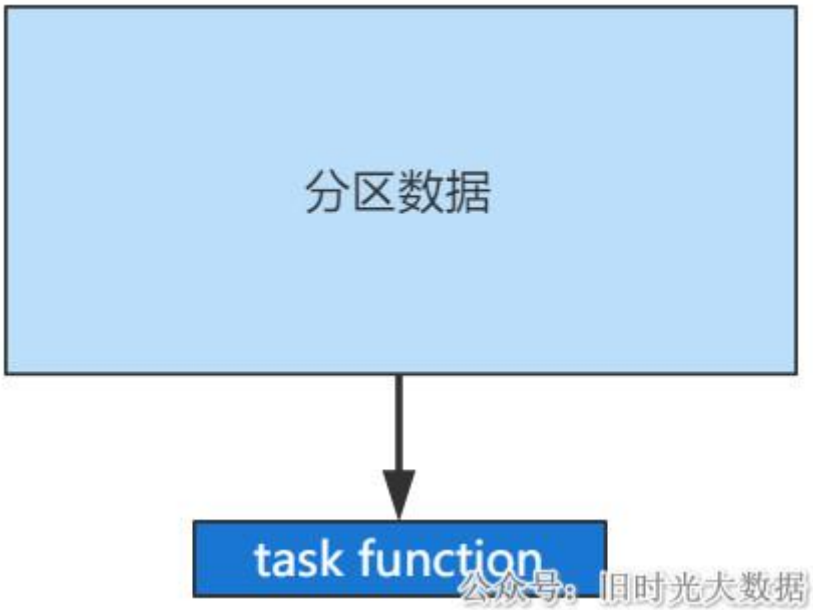
1）算子调优一：mapPartitions

普通的map算子对RDD中的每一个元素进行操作，而mapPartitions算子对RDD中每一个分区进行操作。如果是普通的map算子，假设一个partition有1万条数据，那么map算子中的function要执行1万次，也就是对每个元素进行操作。

公众号：旧时光大数据



如果是mapPartition算子，由于一个task处理一个RDD的partition，那么一个task只会执行一次function，function一次接收所有的partition数据，效率比较高。



比如，当要把RDD中的所有数据通过JDBC写入数据，如果使用map算子，那么需要对RDD中的每一个元素都创建一个数据库连接，这样对资源的消耗很大，如果使用mapPartitions算子，那么针对一个分区的数据，只需要建立一个数据库连接。

mapPartitions算子也存在一些缺点：对于普通的map操作，一次处理一条数据，如果在处理了2000条数据后内存不足，那么可以将已经处理完的2000条数据从内存中垃圾回收掉；但是如果使用mapPartitions算子，但数据量非常大时，function一次处理一个分区的数据，如果一旦内存不足，此时无法回收内存，就可能会OOM，即内存溢出。

因此，mapPartitions算子适用于数据量不是特别大的时候，此时使用mapPartitions算子对性能的提升效果还是不错的。（当数据量很大的时候，一旦使用mapPartitions算子，就会直接OOM）

在项目中，应该首先估算一下RDD的数据量、每个partition的数据量，以及分配给每个Executor的内存资源，如果资源允许，可以考虑使用mapPartitions算子代替map。

2）算子调优二：foreachPartition优化数据库操作

公众号：旧时光大数据

在生产环境中，通常使用foreachPartition算子来完成数据库的写入，通过foreachPartition算子的特性，可以优化写数据库的性能。

如果使用foreach算子完成数据库的操作，由于foreach算子是遍历RDD的每条数据，因此，每条数据都会建立一个数据库连接，这是对资源的极大浪费，因此，对于写数据库操作，我们应当使用foreachPartition算子。

与mapPartitions算子非常相似，foreachPartition是将RDD的每个分区作为遍历对象，一次处理一个分区的数据，也就是说，如果涉及数据库的相关操作，一个分区的数据只需要创建一次数据库连接，如图所示：

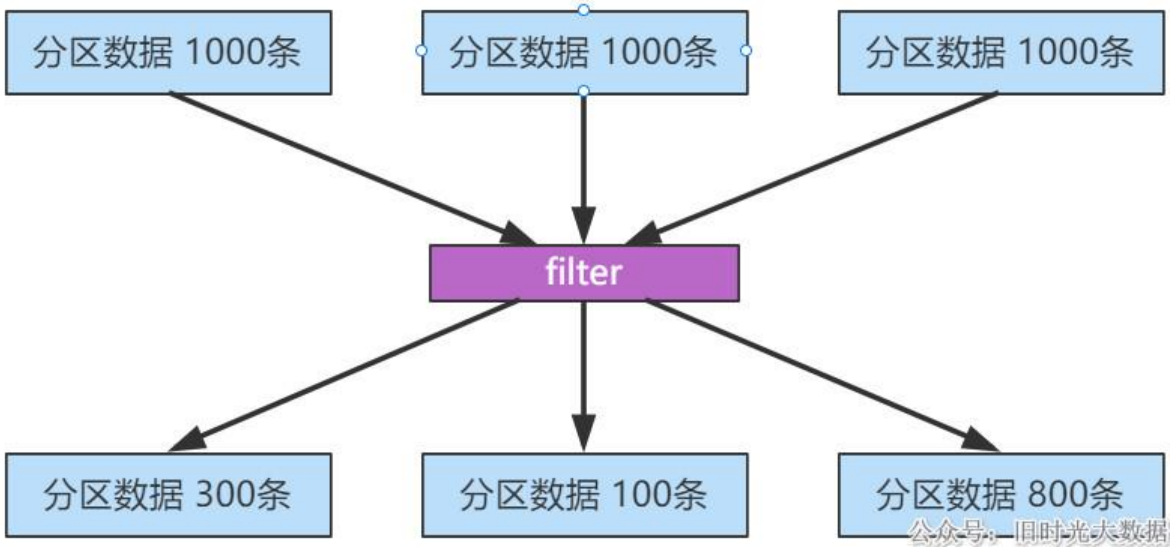
使用了foreachPartition算子后，可以获得以下的性能提升：

- 对于我们写的function函数，一次处理一整个分区的数据；
- 对于一个分区内的数据，创建唯一的数据库连接；
- 只需要向数据库发送一次SQL语句和多组参数；

在生产环境中，全部都会使用foreachPartition算子完成数据库操作。foreachPartition算子存在一个问题，与mapPartitions算子类似，如果一个分区的数据量特别大，可能会造成OOM，即内存溢出。

3）算子调优三：filter与coalesce的配合使用

在Spark任务中我们经常会使用filter算子完成RDD中数据的过滤，在任务初始阶段，从各个分区中加载到的数据量是相近的，但是一旦进过filter过滤后，每个分区的数据量有可能会存在较大差异，如图所示：



根据图中信息我们可以发现两个问题：

- 每个partition的数据量变小了，如果还按照之前与partition相等的task个数去处理当前数据，有点浪费task的计算资源；
- 每个partition的数据量不一样，会导致后面的每个task处理每个partition数据的时候，每个task要处理的数据量不同，这很有可能导致数据倾斜问题。

如上图所示，第二个分区的数据过滤后只剩100条，而第三个分区的数据过滤后剩下800条，在相同的处理逻辑下，第二个分区对应的task处理的数据量与第三个分区对应的task处理的数据量差距达到了8倍，这也会导致运行速度可能存在数倍的差距，这也就是数据倾斜问题。

针对上述的两个问题，我们分别进行分析：

- 针对第一个问题，既然分区的数据量变小了，我们希望可以对分区数据进行重新分配，比如将原来4个分区的数据转化到2个分区中，这样只需要用后面的两个task进行处理即可，避免了资源的浪

公众号：旧时光大数据

费。

- 针对第二个问题，解决方法和第一个问题的解决方法非常相似，对分区数据重新分配，让每个partition中的数据量差不多，这就避免了数据倾斜问题。

那么具体应该如何实现上面的解决思路？我们需要coalesce算子。

repartition与coalesce都可以用来进行重分区，其中repartition只是coalesce接口中shuffle为true的简易实现，coalesce默认情况下不进行shuffle，但是可以通过参数进行设置。

假设我们希望将原本的分区个数A通过重新分区变为B，那么有以下几种情况：

A > B（多数分区合并为少数分区）

a）A与B相差值不大

此时使用coalesce即可，无需shuffle过程。

b）A与B相差值很大

此时可以使用coalesce并且不启用shuffle过程，但是会导致合并过程性能低下，所以推荐设置coalesce的第二个参数为true，即启动shuffle过程。

A < B（少数分区分解为多数分区）

此时使用repartition即可，如果使用coalesce需要将shuffle设置为true，否则coalesce无效。

我们可以在filter操作之后，使用coalesce算子针对每个partition的数据量各不相同的情况，压缩partition的数量，而且让每个partition的数据量尽量均匀紧凑，以便于后面的task进行计算操作，在某种程度上能够在一定程度上提升性能。

注意：local模式是进程内模拟集群运行，已经对并行度和分区数量有了一定的内部优化，因此不用去设置并行度和分区数量。

4）算子调优四：repartition解决SparkSQL低并行度问题

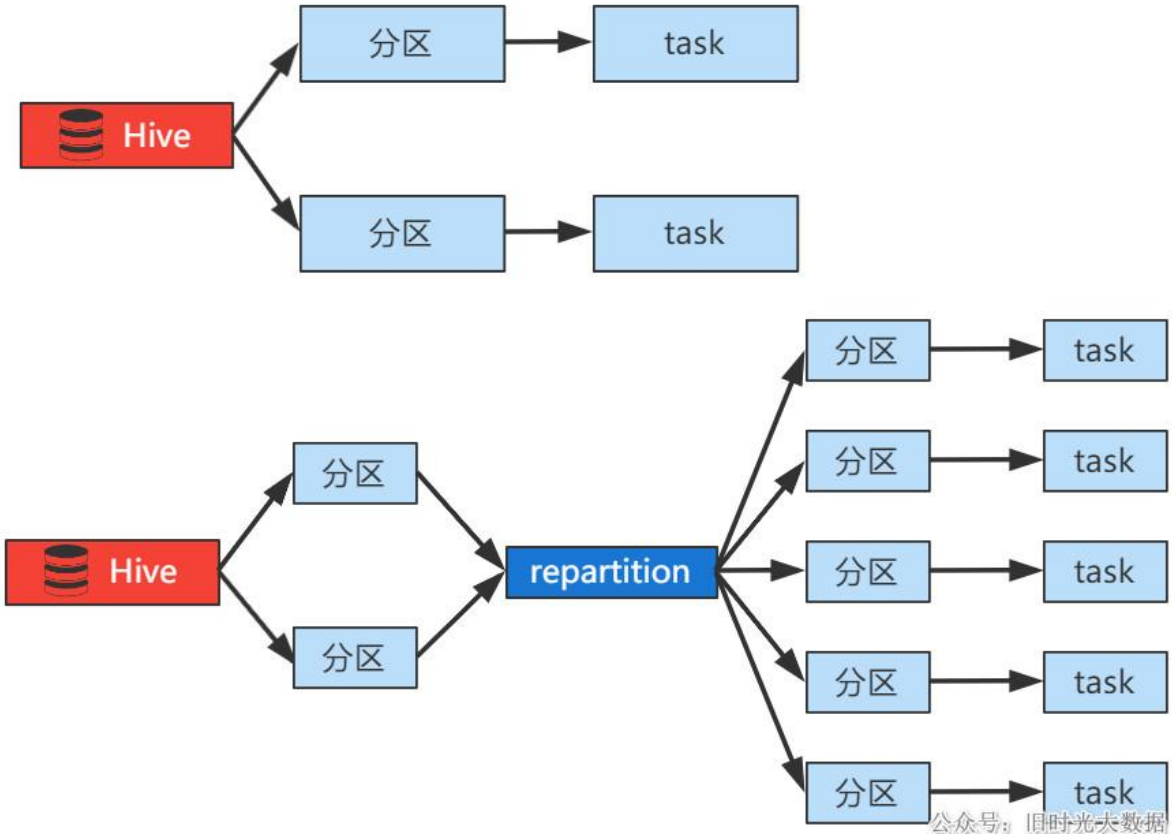
在前面常规性能调优中我们说了并行度的调节策略，但是，并行度的设置对于Spark SQL是不生效的，用户设置的并行度只对于Spark SQL以外的所有Spark的stage生效。

Spark SQL的并行度不允许用户自己指定，Spark SQL自己会默认根据hive表对应的HDFS文件的split个数自动设置Spark SQL所在的那个stage的并行度，用户自己通spark.default.parallelism参数指定的并行度，只会在没Spark SQL的stage中生效。

由于Spark SQL所在stage的并行度无法手动设置，如果数据量较大，并且此stage中后续的transformation操作有着复杂的业务逻辑，而Spark SQL自动设置的task数量很少，这就意味着每个task要处理为数不少的数据量，然后还要执行非常复杂的处理逻辑，这就可能表现为第一个有Spark SQL的stage速度很慢，而后续的没有Spark SQL的stage运行速度非常快。

为了解决Spark SQL无法设置并行度和task数量的问题，我们可以使用repartition算子。

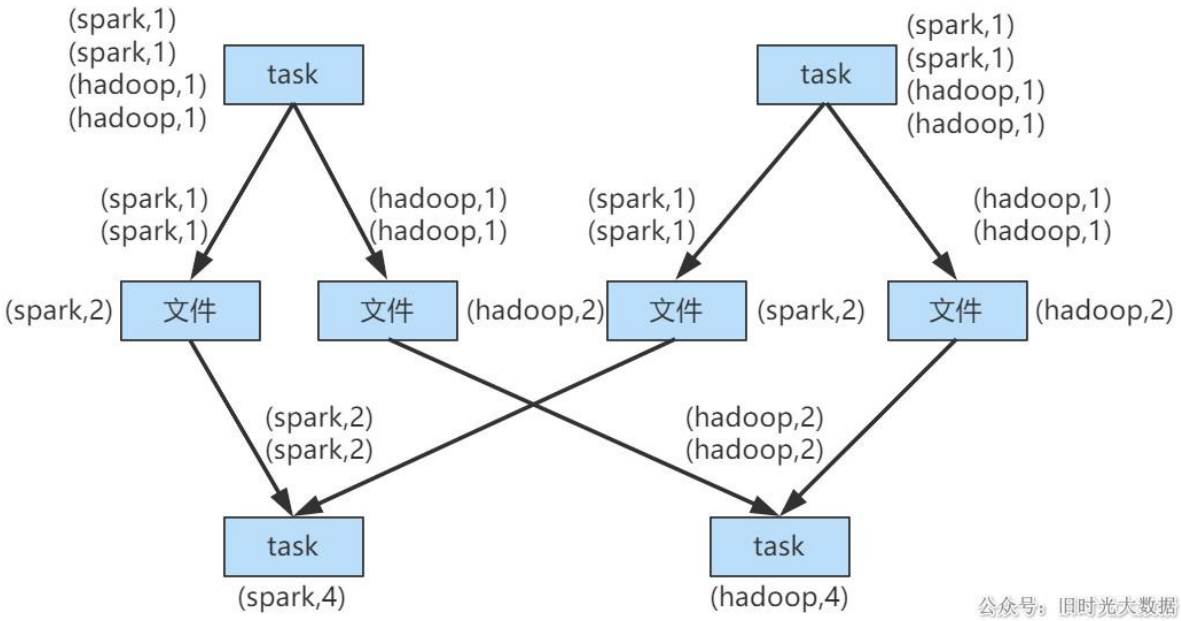
公众号：旧时光大数据



Spark SQL这一步的并行度和task数量肯定是没有办法去改变了，但是，对于Spark SQL查询出来的RDD，立即使用repartition算子，去重新进行分区，这样可以重新分区为多个partition，从repartition之后的RDD操作，由于不再设计Spark SQL，因此stage的并行度就会等于你手动设置的值，这样就避免了Spark SQL所在的stage只能用少量的task去处理大量数据并执行复杂的算法逻辑。

5）算子调优五：reduceByKey预聚合

reduceByKey相较于普通的shuffle操作一个显著的特点就是会进行map端的本地聚合，map端会先对本地的数据进行combine操作，然后将数据写入给下个stage的每个task创建的文件中，也就是在map端，对每一个key对应的value，执行reduceByKey算子函数。reduceByKey算子的执行过程如图所示：



使用reduceByKey对性能的提升如下：

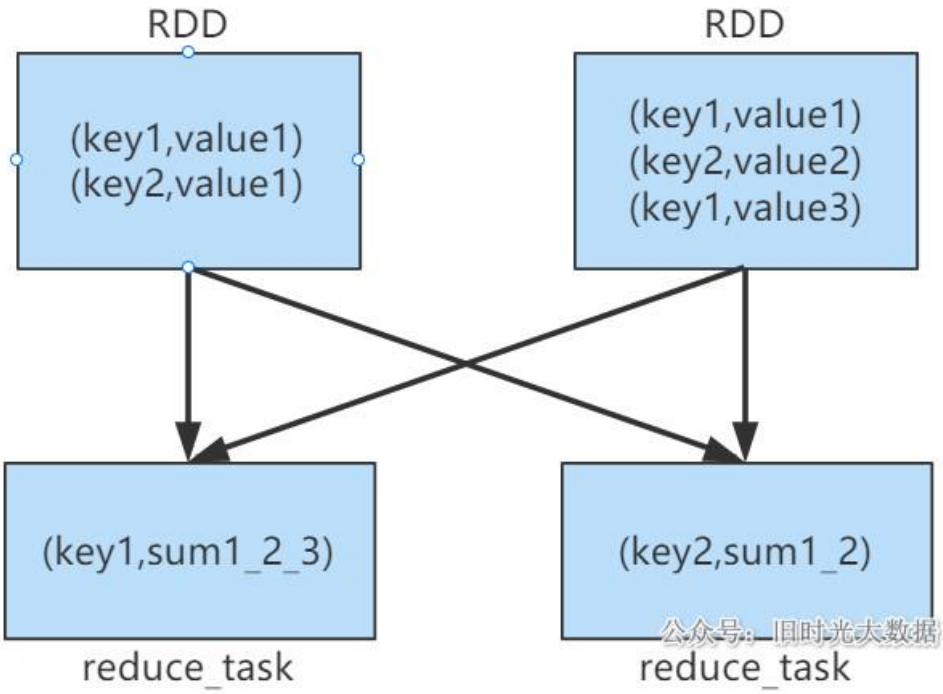
- 本地聚合后，在map端的数据量变少，减少了磁盘IO，也减少了对磁盘空间的占用；
- 本地聚合后，下一个stage拉取的数据量变少，减少了网络传输的数据量；
- 本地聚合后，在reduce端进行数据缓存的内存占用减少；

公众号：旧时光大数据

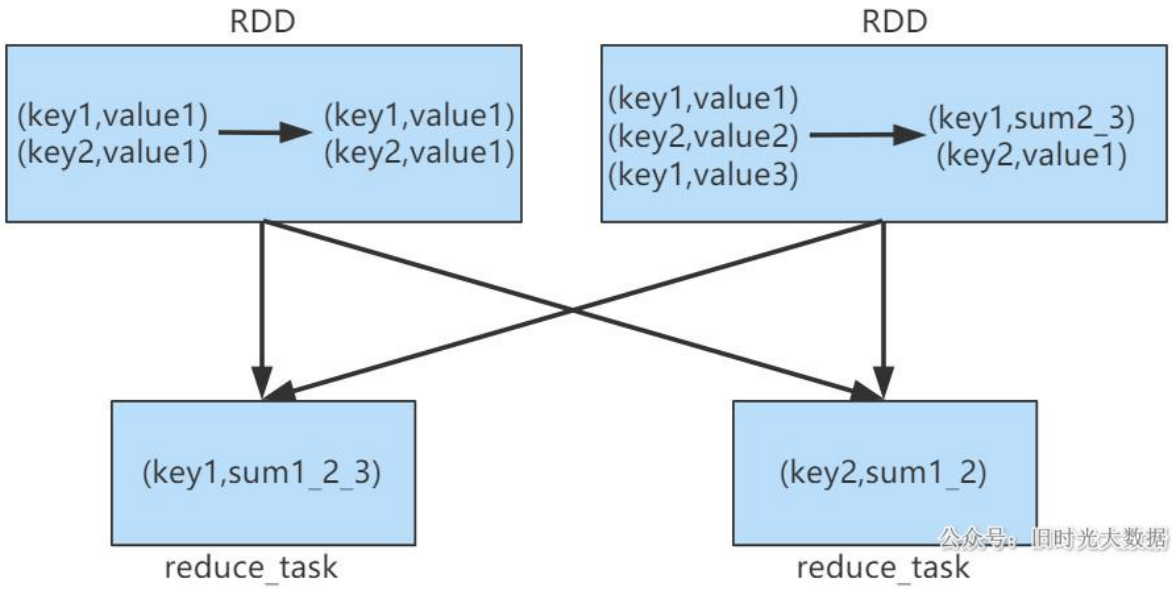
- 本地聚合后，在reduce端进行聚合的数据量减少。

基于reduceByKey的本地聚合特征，我们应该考虑使用reduceByKey代替其他的shuffle算子，例如groupByKey。reduceByKey与groupByKey的运行原理如图所示：

groupByKey的运行原理：



reduceByKey的运行原理：



根据上图可知，groupByKey不会进行map端的聚合，而是将所有map端的数据shuffle到reduce端，然后在reduce端进行数据的聚合操作。由于reduceByKey有map端聚合的特性，使得网络传输的数据量减小，因此效率要明显高于groupByKey。

4、Shuffle调优

1）Shuffle调优一：调节map端缓冲区大小

在Spark任务运行过程中，如果shuffle的map端处理的数据量比较大，但是map端缓冲的大小是固定的，可能会出现map端缓冲数据频繁spill溢写到磁盘文件中的情况，使得性能非常低下，通过调节map端缓冲的大小，可以避免频繁的磁盘IO操作，进而提升Spark任务的整体性能。

公众号：旧时光大数据

map端缓冲的默认配置是32KB，如果每个task处理640KB的数据，那么会发生640/32 = 20次溢写，如果每个task处理64000KB的数据，机会发生64000/32=2000此溢写，这对于性能的影响是非常严重的。

map端缓冲的配置方法如代码清单所示：

```
1    val conf = new SparkConf().set("spark.shuffle.file.buffer", "64")
2
```

2）Shuffle调优二：调节reduce端拉取数据缓冲区大小

Spark Shuffle过程中，shuffle reduce task的buffer缓冲区大小决定了reduce task每次能够缓冲的数据量，也就是每次能够拉取的数据量，如果内存资源较为充足，适当增加拉取数据缓冲区的大小，可以减少拉取数据的次数，也就可以减少网络传输的次数，进而提升性能。

reduce端数据拉取缓冲区的大小可以通过spark.reducer.maxSizeInFlight参数进行设置，默认为48MB，该参数的设置方法如代码清单所示：

```
1    val conf = new SparkConf().set("spark.reducer.maxSizeInFlight", "96")
2
```

3）Shuffle调优三：调节reduce端拉取数据重试次数

Spark Shuffle过程中，reduce task拉取属于自己的数据时，如果因为网络异常等原因导致失败会自动进行重试。对于那些包含了特别耗时的shuffle操作的作业，建议增加重试最大次数（比如60次），以避免由于JVM的full gc或者网络不稳定等因素导致的数据拉取失败。在实践中发现，对于针对超大数据量（数十亿~上百亿）的shuffle过程，调节该参数可以大幅度提升稳定性。

reduce端拉取数据重试次数可以通过spark.shuffle.io.maxRetries参数进行设置，该参数就代表了可以重试的最大次数。如果在指定次数之内拉取还是没有成功，就可能会导致作业执行失败，默认为3，该参数的设置方法如代码清单所示：

```
1    val conf = new SparkConf().set("spark.shuffle.io.maxRetries", "6")
2
```

4）Shuffle调优四：调节reduce端拉取数据等待间隔

Spark Shuffle过程中，reduce task拉取属于自己的数据时，如果因为网络异常等原因导致失败会自动进行重试，在一次失败后，会等待一定的时间间隔再进行重试，可以通过加大间隔时长（比如60s），以增加shuffle操作的稳定性。

reduce端拉取数据等待间隔可以通过spark.shuffle.io.retryWait参数进行设置，默认值为5s，该参数的设置方法如代码清单所示：

```
1    val conf = new SparkConf().set("spark.shuffle.io.retryWait", "60s")
2
```

5）Shuffle调优五：调节SortShuffle排序操作阈值

对于SortShuffleManager，如果shuffle reduce task的数量小于某一阈值则shuffle write过程中不会进行排序操作，而是直接按照未经优化的HashShuffleManager的方式去写数据，但是最后会将每个task产生的所有临时磁盘文件都合并成一个文件，并会创建单独的索引文件。

当你使用SortShuffleManager时，如果的确不需要排序操作，那么建议将这个参数调大一些，大于shuffle read task的数量，那么此时map-side就不会进行排序了，减少了排序的性能开销，但是这种方式下，依然会产生大量的磁盘文件，因此shuffle write性能有待提高。

SortShuffleManager排序操作阈值的设置可以通过spark.shuffle.sort. bypassMergeThreshold这一参数进行设置，默认值为200，该参数的设置方法如代码清单所示：

公众号：旧时光大数据

```
val conf = new SparkConf().set("spark.shuffle.sort.bypassMergeThreshold", "400")
```

4、JVM调优

对于JVM调优，首先应该明确，full gc/minor gc，都会导致JVM的工作线程停止工作，即stop the world。

1）JVM调优一：降低cache操作的内存占比

（1）静态内存管理机制

根据Spark静态内存管理机制，堆内存被划分为了两块，Storage和Execution。Storage主要用于缓存RDD数据和broadcast数据，Execution主要用于缓存在shuffle过程中产生的中间数据，Storage占系统内存的60%，Execution占系统内存的20%，并且两者完全独立。

在一般情况下，Storage的内存都提供给了cache操作，但是如果在某些情况下cache操作内存不是很紧张，而task的算子中创建的对象很多，Execution内存又相对较小，这回导致频繁的minor gc，甚至于频繁的full gc，进而导致Spark频繁的停止工作，性能影响会很大。

在Spark UI中可以查看每个stage的运行情况，包括每个task的运行时间、gc时间等等，如果发现gc太频繁，时间太长，就可以考虑调节Storage的内存占比，让task执行算子函数式，有更多的内存可以使用。

Storage内存区域可以通过spark.storage.memoryFraction参数进行指定，默认为0.6，即60%，可以逐级向下递减，如代码清单所示：

```
val conf = new SparkConf().set("spark.storage.memoryFraction", "0.4")
```

（2）统一内存管理机制

根据Spark统一内存管理机制，堆内存被划分为了两块，Storage和Execution。Storage主要用于缓存数据，Execution主要用于缓存在shuffle过程中产生的中间数据，两者所组成的内存部分称为统一内存，Storage和Execution各占统一内存的50%，由于动态占用机制的实现，shuffle过程需要的内存过大时，会自动占用Storage的内存区域，因此无需手动进行调节。

2）JVM调优二：调节Executor堆外内存

Executor的堆外内存主要用于程序的共享库、Perm Space、 线程Stack和一些Memory mapping等, 或者类C方式allocate object。

有时，如果你的Spark作业处理的数据量非常大，达到几亿的数据量，此时运行Spark作业会时不时地报错，例如shuffle output file cannot find，executor lost，task lost，out of memory等，这可能是Executor的堆外内存不太够用，导致Executor在运行的过程中内存溢出。

stage的task在运行的时候，可能要从一些Executor中去拉取shuffle map output文件，但是Executor可能已经由于内存溢出挂掉了，其关联的BlockManager也没有了，这就可能会报出shuffle output file cannot find，executor lost，task lost，out of memory等错误，此时，就可以考虑调节一下Executor的堆外内存，也就可以避免报错，与此同时，堆外内存调节的比较大的时候，对于性能来讲，也会带来一定的提升。

默认情况下，Executor堆外内存上限大概为300多MB，在实际的生产环境下，对海量数据进行处理的时候，这里都会出现问题，导致Spark作业反复崩溃，无法运行，此时就会去调节这个参数，到至少1G，甚至于2G、4G。

Executor堆外内存的配置需要在spark-submit脚本里配置，如代码清单所示：

```
--conf spark.yarn.executor.memoryOverhead=2048
```

以上参数配置完成后，会避免掉某些JVM OOM的异常问题，同时，可以提升整体Spark作业的性能。

公众号：旧时光大数据

3）JVM调优三：调节连接等待时长

在Spark作业运行过程中，Executor优先从自己本地关联的BlockManager中获取某份数据，如果本地BlockManager没有的话，会通过TransferService远程连接其他节点上Executor的BlockManager来获取数据。

如果task在运行过程中创建大量对象或者创建的对象较大，会占用大量的内存，这回导致频繁的垃圾回收，但是垃圾回收会导致工作现场全部停止，也就是说，垃圾回收一旦执行，Spark的Executor进程就会停止工作，无法提供相应，此时，由于没有响应，无法建立网络连接，会导致网络连接超时。

在生产环境下，有时会遇到file not found、file lost这类错误，在这种情况下，很有可能是Executor的BlockManager在拉取数据的时候，无法建立连接，然后超过默认的连接等待时长60s后，宣告数据拉取失败，如果反复尝试都拉取不到数据，可能会导致Spark作业的崩溃。这种情况也可能会导致DAGScheduler反复提交几次stage，TaskScheduler返回提交几次task，大大延长了我们的Spark作业的运行时间。

此时，可以考虑调节连接的超时时长，连接等待时长需要在spark-submit脚本中进行设置，设置方式如代码清单所示：

```
1   --conf spark.core.connection.ack.wait.timeout=300
2
```

调节连接等待时长后，通常可以避免部分的XX文件拉取失败、XX文件lost等报错。

介绍一下Spark怎么基于内存计算的

问过的一些公司：端点数据(2021.07)

参考答案：

主要了是基于RDD

说下什么是RDD（对RDD的理解）？RDD有哪些特点？说下知道的RDD算子

问过的一些公司：字节x2，小米x2，平安，阿里x 2，海康，美团x2，一点咨询，趋势科技，快手x2，阿里云社招，祖龙娱乐，360，字节社招，端点数据(2021.07)，大华(2021.07)，触宝(2021.07)x2，携程(2021.09)

参考答案：

RDD（Resilient Distributed Dataset）叫做弹性分布式数据集，是 Spark 中最基本的数据处理模型。代码中是一个抽象类，它代表一个弹性的、不可变、可分区、里面的元素可并行计算的集合。

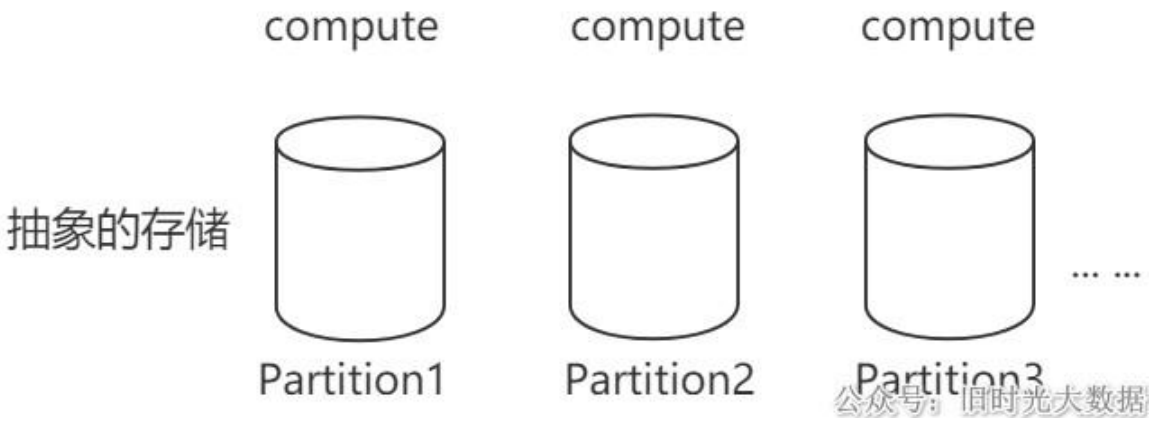
RDD特点

RDD表示只读的分区的数据集，对RDD进行改动，只能通过RDD的转换操作，由一个RDD得到一个新的RDD，新的RDD包含了从其他RDD衍生所必需的信息。RDDs之间存在依赖，RDD的执行是按照血缘关系延时计算的。如果血缘关系较长，可以通过持久化RDD来切断血缘关系。

1、分区

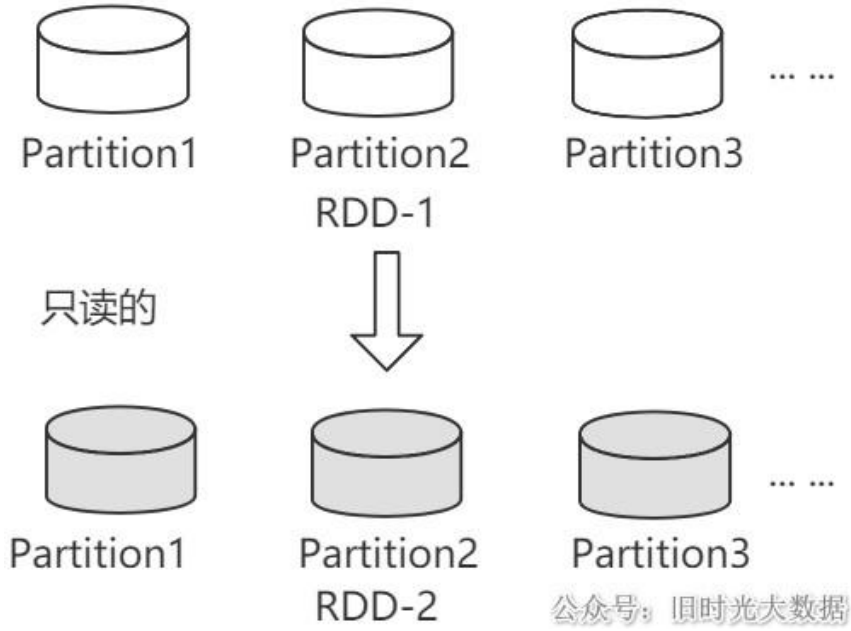
RDD逻辑上是分区的，每个分区的数据是抽象存在的，计算的时候会通过一个compute函数得到每个分区的数据。如果RDD是通过已有的文件系统构建，则compute函数是读取指定文件系统中的数据，如果RDD是通过其他RDD转换而来，则compute函数是执行转换逻辑将其他RDD的数据进行转换。

公众号：旧时光大数据



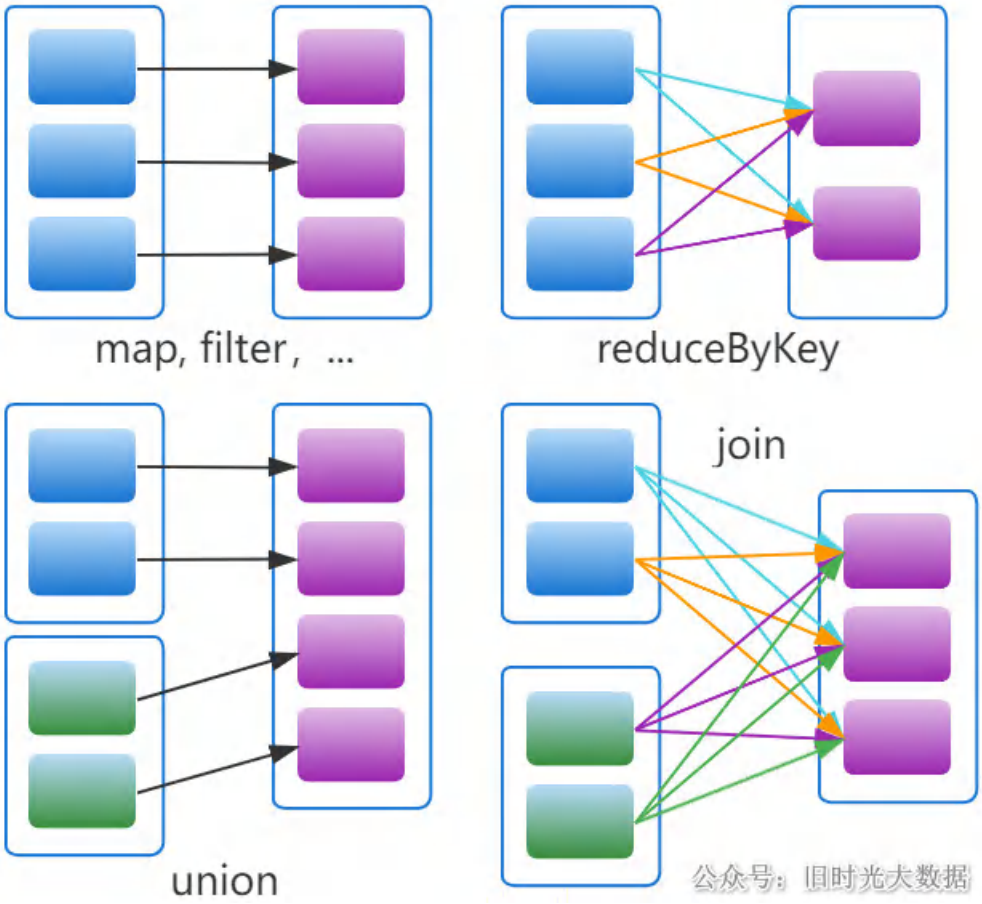
2、只读

如下图所示，RDD是只读的，要想改变RDD中的数据，只能在现有的RDD基础上创建新的RDD。



由一个RDD转换到另一个RDD，可以通过丰富的操作算子实现，不再像MapReduce那样只能写map和reduce了，如下图所示。

公众号：旧时光大数据

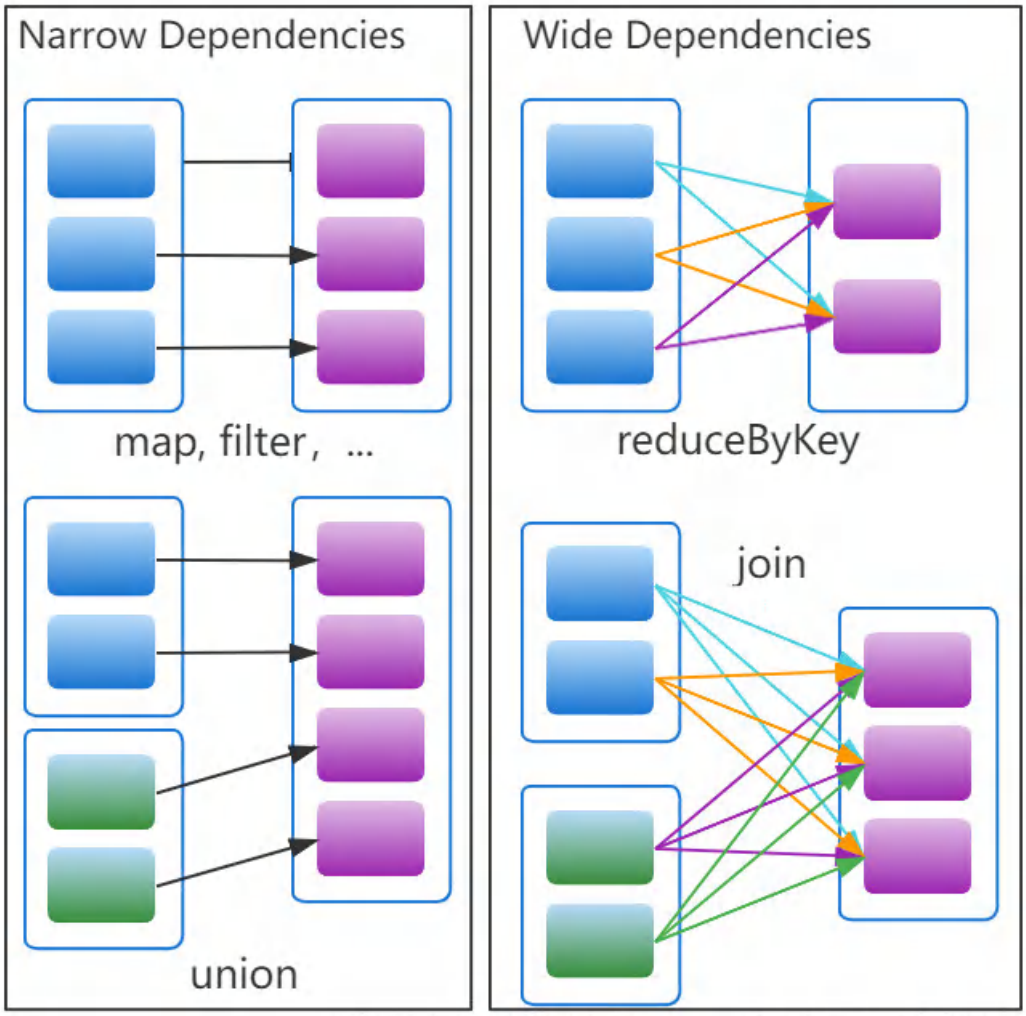


RDD的操作算子包括两类，一类叫做transformations，它是用来将RDD进行转化，构建RDD的血缘关系；另一类叫做actions，它是用来触发RDD的计算，得到RDD的相关计算结果或者将RDD保存的文件系统中。下图是RDD所支持的操作算子列表。

3、依赖

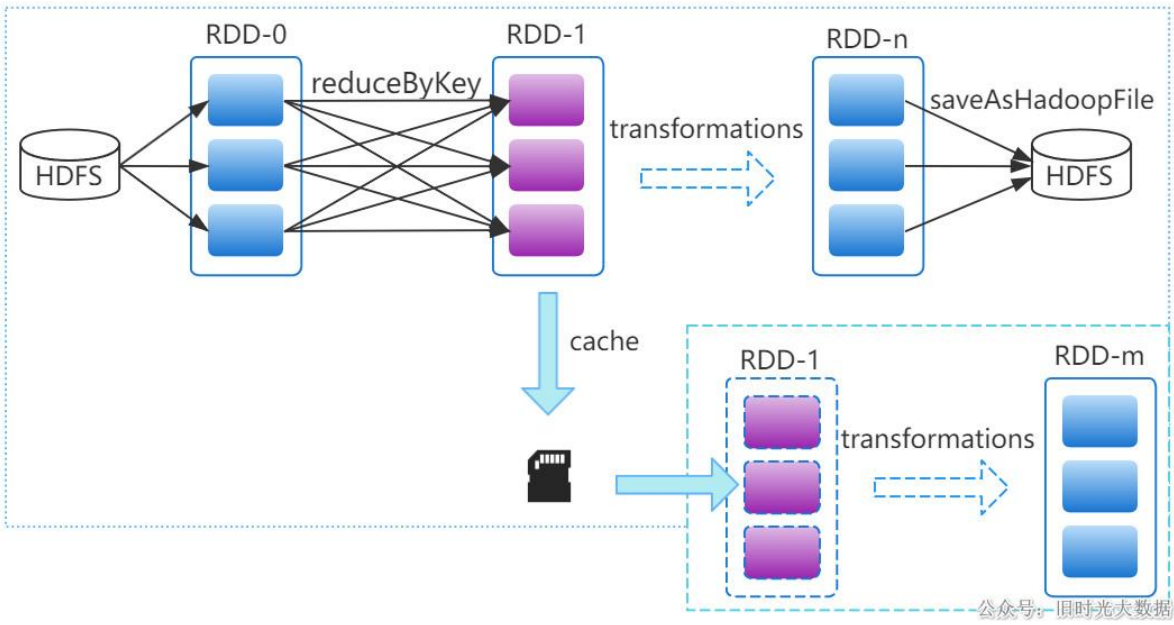
RDDs通过操作算子进行转换，转换得到的新RDD包含了从其他RDDs衍生所必需的信息，RDDs之间维护着这种血缘关系，也称之为依赖。如下图所示，依赖包括两种，一种是窄依赖，RDDs之间分区是一一对应的，另一种是宽依赖，下游RDD的每个分区与上游RDD(也称之为父RDD)的每个分区都有关，是多对多的关系。

公众号：旧时光大数据



4、缓存

如果在应用程序中多次使用同一个RDD，可以将该RDD缓存起来，该RDD只有在第一次计算的时候会根据血缘关系得到分区的数据，在后续其他地方用到该RDD的时候，会直接从缓存处取而不用再根据血缘关系计算，这样就加速后期的重用。如下图所示，RDD-1经过一系列的转换后得到RDD-n并保存到hdfs，RDD-1在这一过程中会有个中间结果，如果将其缓存到内存，那么在随后的RDD-1转换到RDD-m这一过程中，就不会计算其之前的RDD-0了。



5、CheckPoint

虽然RDD的血缘关系天然地可以实现容错，当RDD的某个分区数据失败或丢失，可以通过血缘关系重建。但是对于长时间迭代型应用来说，随着迭代的进行，RDDs之间的血缘关系会越来越长，一旦在后续迭代过程中出错，则需要通过非常长的血缘关系去重建，势必影响性能。为此，RDD支持checkpoint将数据保存到持久化的存储中，这样就可以切断之前的血缘关系，因为checkpoint后的RDD不需要知道它的父RDDs了，它可以从checkpoint处拿到数据。

RDD算子

RDD整体上分为Value类型和Key-Value类型

比如map、flatmap等这些，回答几个，讲一下原理就差不多了

map：遍历RDD,将函数f应用于每一个元素，返回新的RDD(transformation算子)。

foreach：遍历RDD,将函数f应用于每一个元素，无返回值(action算子)。

mapPartitions：遍历操作RDD中的每一个分区，返回新的RDD（transformation算子）。

foreachPartition：遍历操作RDD中的每一个分区。无返回值(action算子)。

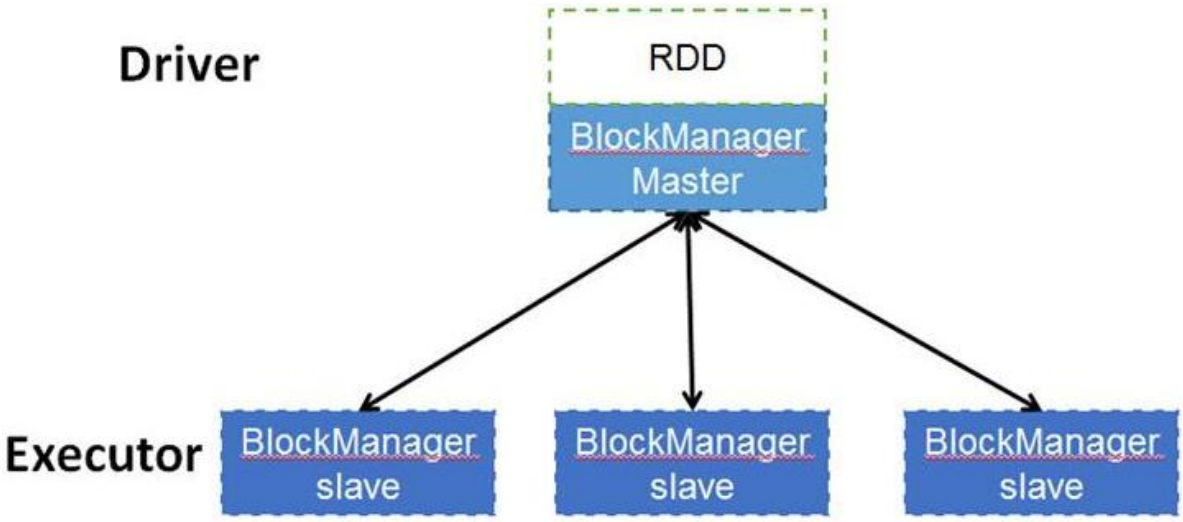
RDD底层原理

可回答：RDD的五个特性

问过的一些公司：携程，端点数据(2021.07)

参考答案：

RDD是一个分布式数据集，顾名思义，其数据应该分部存储于多台机器上。事实上，每个RDD的数据都以Block的形式存储于多台机器上，下图是Spark的RDD存储架构图，其中每个Executor会启动一个BlockManagerSlave，并管理一部分Block；而Block的元数据由Driver节点的BlockManagerMaster保存。BlockManagerSlave生成Block后向BlockManagerMaster注册该Block，BlockManagerMaster管理RDD与Block的关系，当RDD不再需要存储的时候，将向BlockManagerSlave发送指令删除相应的Block。



RDD属性

问过的一些公司：字节，端点数据(2021.07)

参考答案：

1）一组分区（Partition），即数据集的基本组成单位

公众号：旧时光大数据

RDD数据结构中存在分区列表，用于执行任务时并行计算，是实现分布式计算的重要属性。

2）一个计算每个分区的函数

Spark在计算时，是使用分区函数对每一个分区进行计算

3）RDD之间的依赖关系

RDD是计算模型的封装，当需求中需要将多个计算模型进行组合时，就需要将多个RDD建立依赖关系

4）一个Partitioner，即RDD的分片函数

当数据为KV类型数据时，可以通过设定分区器自定义数据的分区

5）一个列表，存储存取每个Partition的优先位置（preferred location）

计算数据时，可以根据计算节点的状态选择不同的节点位置进行计算

RDD的缓存级别?

问过的一些公司：快手

参考答案：

在Spark持久化时，Spark规定了MEMORY_ONLY、MEMORY_AND_DISK等七种不同的存储级别，而存储级别是以下5个变量的组合：

```
class StorageLevel private{
    private var _useDisk：Boolean,  //磁盘
    private var _useMemory：Boolean,  //这里指的是堆内内存
    private var _useOffHeap：Boolean,  //堆外内存
    private var _deserialized：Boolean,  //是否为非序列化
    private var _replication：Int = 1  //副本个数
}

```

Spark中7种存储级别如下：

持久化级别	含义
MEMORY_ONLY	以非序列化的Java对象的方式持久化在JVM内存中。如果内存无法完全存储RDD所有的partition，那么那些没有持久化的partition就会在下一次需要使用它们的时候，重新被计算
MEMORY_AND_DISK	同上，但是当某些partition无法存储在内存中时，会持久化到磁盘中。下次需要使用这些partition时，需要从磁盘中读取
MEMORY_ONLY_SER	同MEMORY_ONLY，但是会使用Java序列化方式，将Java对象序列化后进行持久化。可以减少内存开销，但是需要进行反序列化，因此会加大CPU开销
MEMORY_AND_DISK_SER	同MEMORY_AND_DISK，但是使用序列化方式持久化Java对象
DISK_ONLY	使用非序列化Java对象的方式持久化，完全存储到磁盘上
MEMORY_ONLY_2、MEMORY_AND_DISK_2等等	如果是尾部加了2的持久化级别，表示将持久化数据复用一份，保存到其它节点，从而在数据丢失时，不需要再次计算，只需要使用备份数据即可

公众号：旧时光大数据

通过对数据结构的分析，可以看出存储级别从三个维度定义了RDD的Partition（同时也就是Block）的存储方式：

- 存储位置：磁盘/堆内内存/堆外内存。如MEMORY_AND_DISK是同时在磁盘和堆内内存上存储，实现了冗余备份。OFF_HEAP则是只在堆外内存存储，目前选择堆外内存时不能存储到其他位置。
- 存储形式：Block缓存到内存后，是否为非序列化的形式。如MEMORY_ONLY是非序列化方式存储，OFF_HEAP是序列化方式存储。
- 副本数量：大于1时需要冗余备份到其他节点。如DISK_ONLY_2需要远程备份1个副本。

Spark广播变量的实现和原理?

可回答：Spark广播变量在什么情况下使用

问过的一些公司：拼多多，腾讯 x 2，小米，祖龙娱乐，滴滴，京东(2021.07)，陌陌(2021.07)，百度(2021.08)

参考答案：

Spark官方对广播变量的说明如下：

广播变量可以让我们在每台计算机上保留一个只读变量，而不是为每个任务复制一份副本。例如，可以使用他们以高效的方式为每个计算节点提供大型输入数据集的副本。Spark也尽量使用有效的广播算法来分发广播变量，以降低通信成本。

另外，Spark action操作会被划分成一系列的stage来执行，这些stage根据是否产生shuffle操作来进行划分的。Spark会自动广播每个stage任务需要的通用数据。这些被广播的数据以序列化的形式缓存起来，然后在任务运行前进行反序列化。

也就是说，在以下两种情况下显示的创建广播变量才有用：

1）当任务跨多个stage并且需要同样的数据时；

2）当以反序列化的形式来缓存数据时。

从以上官方定义我们可以得出Spark广播变量的一些特性：

1）广播变量会在每个worker节点上保留一份副本，而不是为每个Task保留一份副本。这样有什么好处？可以想象，在一个worker有时同时会运行若干的Task，若把一个包含较大数据的变量为Task都复制一份，而且还需要通过网络传输，应用的处理效率一定会受到很大影响。

2）Spark会通过某种广播算法来进行广播变量的分发，这样可以减少通信成本。Spark使用了类似于BitTorrent协议的数据分发算法来进行广播变量的数据分发，该分发算法会在后面进行分析。

3）广播变量有一定的适用场景：当任务跨多个stage，且需要同样的数据时，或以反序列化的形式来缓存数据时。

1、广播变量的创建和使用

假设你有一个变量: v。要使用该变量来创建一个广播变量时，非常简单，只需要调用SparkContext的broadcast(v)函数即可。在spark-shell下代码如下：

```
scala> val v = Array(1,2,3,4,5,6)
v: Array[Int] = Array(1, 2, 3, 4, 5, 6)

scala> val bv = sc.broadcast(v)
res1: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(1)

scala> bv
res9: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(2)
```

公众号：旧时光大数据

```
// 获取广播变量的值
scala> bv.value
res10: Array[Int] = Array(1, 2, 3, 4, 5, 6)

// 销毁广播变量
scala> bv.destory
```

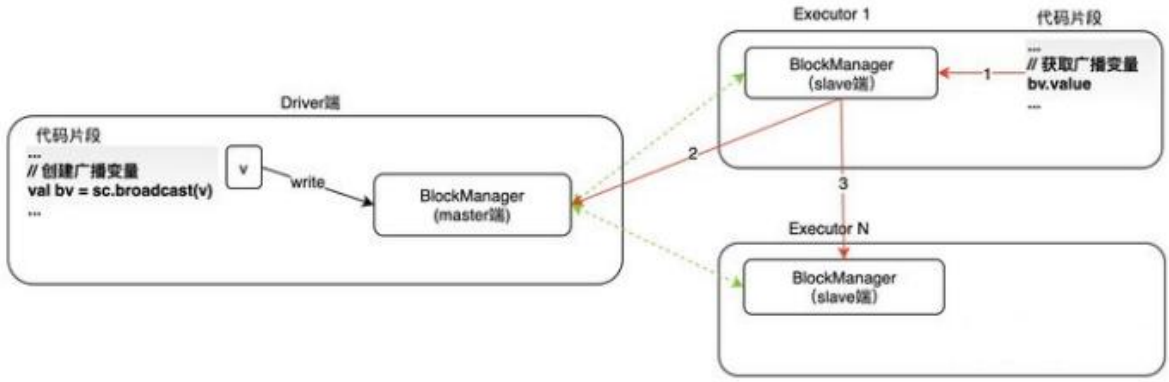
我们把一个变量v（一个普通数组）转换成了一个广播变量bv。通过查看bv的类型，可以看出bv是一个Array[Int]类型的广播变量。我们可以通过bv.value来获取广播变量的值。这样，广播变量bv就可以用到以后的数据计算中了。

注意，在创建广播变量时，**广播变量的值必须是本地的可序列化的值，不能是RDD。**

另外，广播变量一旦创建就不应该再修改，这样可以保证所以的worker节点上的值是一致的。这是因为，现有worker将看不到更新的值，新的worker才可能会看到新的值。

2、广播变量的实现原理

根据广播变量的创建和使用流程来分析广播变量的实现。广播变量的实现过程如下图所示：



注：BlockManager是Spark数据块管理模块

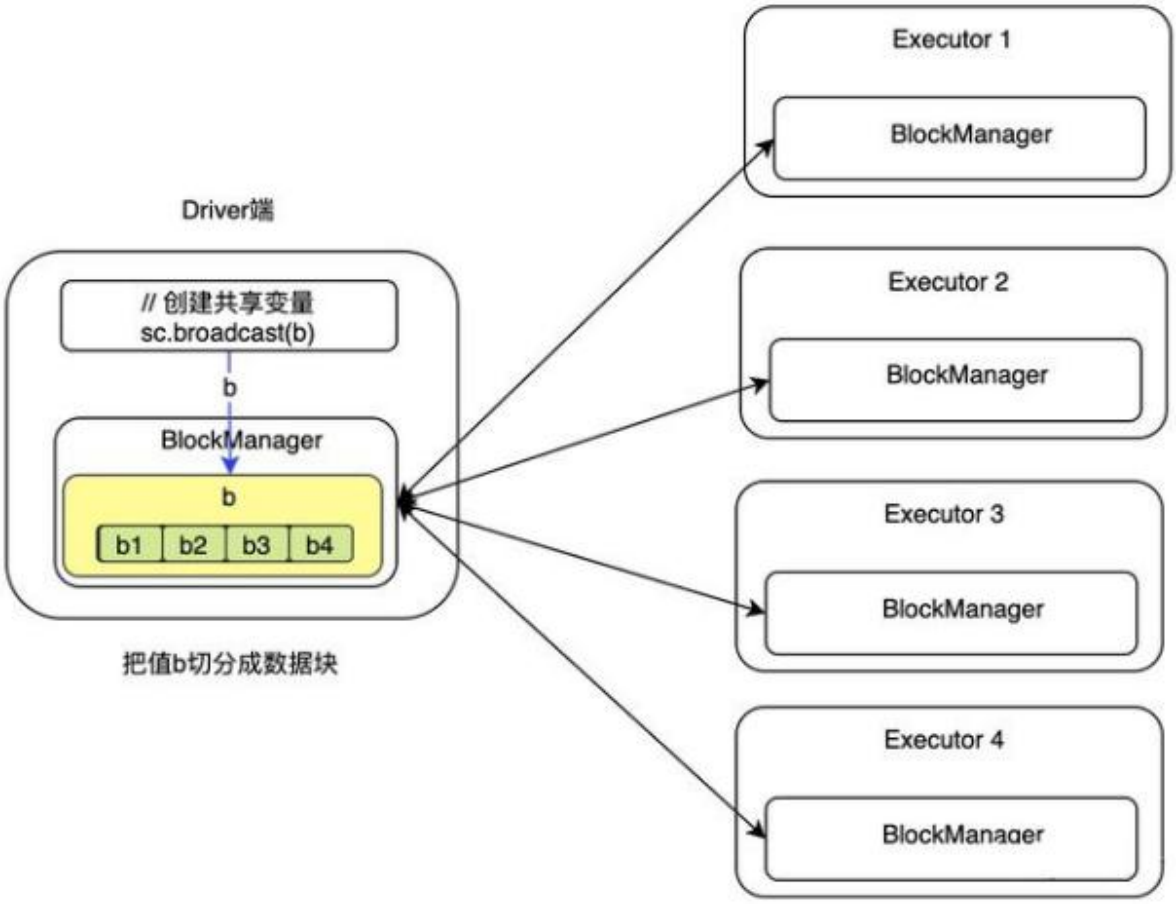
3、广播变量的创建

广播变量的创建发生在Driver端，如图2所示，当调用SparkContext#broadcast来创建广播变量时，会把该变量的数据切分成多个数据块，保存到driver端的BlockManger中，使用的存储级别是：MEMORY_AND_DISK_SER。

所以，广播变量的读取也是懒评价的，只有在Executor端需要获取广播变量时才会去获取。此时广播变量的数据只在Driver端存在。

此时的状态如下图所示：

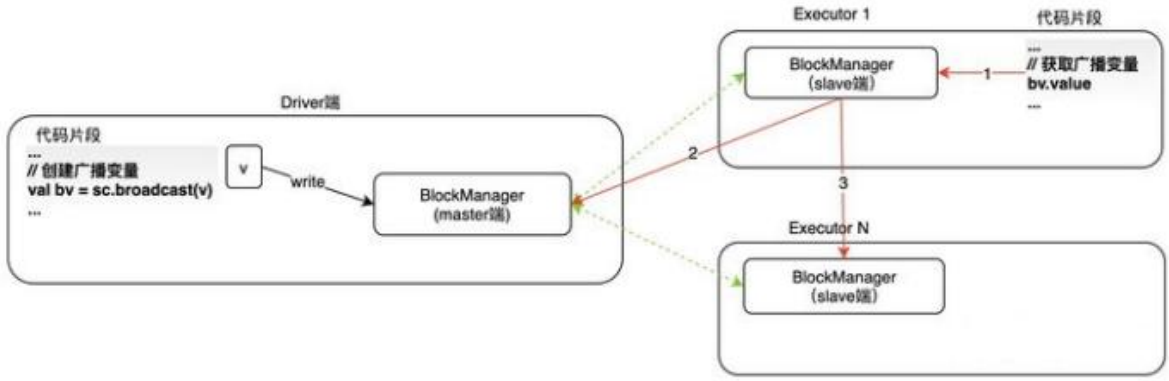
公众号：旧时光大数据



从上图可以看出，此时广播变量被保存在本地，并会把广播变量的值切分成多个数据块进行保存。广播变量数据块的默认大小是4M，数据块太大或太小都不利于数据的传输。

4、广播变量的读取

当要使用广播变量时，需要先获取广播变量的值，其实现流程如下图所示。获取广播变量调用的是bv.value。



其实现逻辑如下：

1）第1步（红色线1）：首先从Executor本地的BlockManager中读取广播变量的数据，若存在就直接获取，并返回。不在执行第2步和第3步。若不存在，则执行第2步。

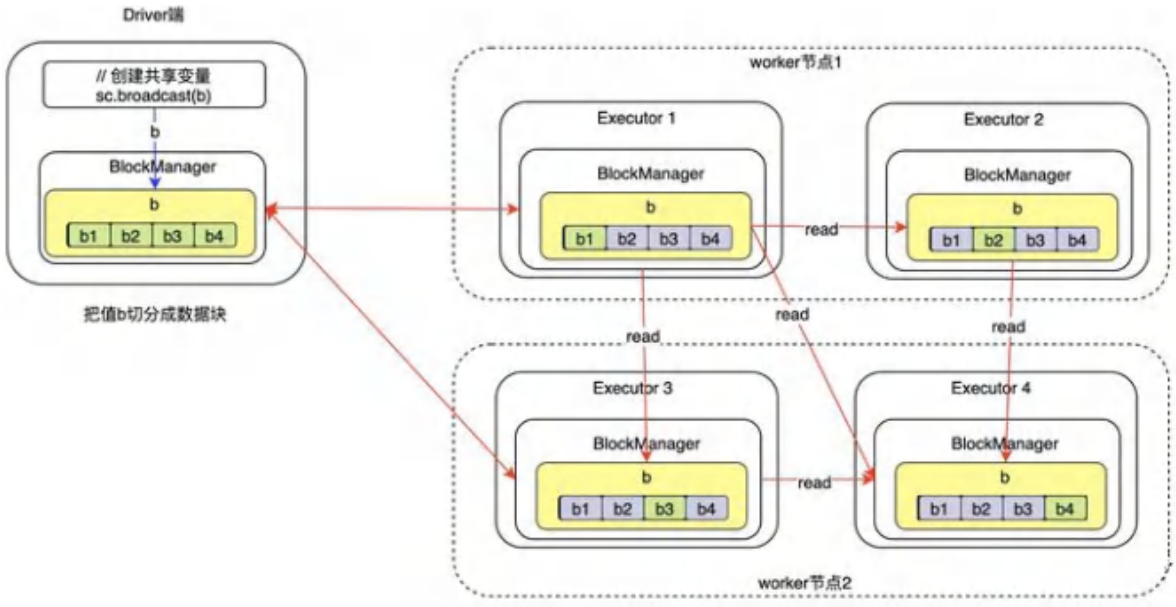
2）第2步（红色线2）：从Driver端获取广播变量的状态和位置信息（由于所有的BlockManager slave端都会向Master端汇报数据块状态）。

3）第3步：优先从本地目录（数据块就在本地），或者相同主机的其他Executor中读取广播变量数据块。若在本Executor和同主机其他Executor中都不存在，则只能从远端获取数据。从远端获取数据的原则是：先从同一个机架(rack)的主机的Executor端获取。若不能从其他Executor中获取广播变量，则会直接从Driver端获取。

从以上获取流程可以看出，在执行spark应用时，只要有一个worker节点的Executor从Driver端获取到了广播变量的数据，则其他的Executor就不需要从Driver端获取了。

当某个Executor上的某个数据块被删除，可以从其他Executor直接获取该数据块，然后把数据块保存到自己的Executor的BlockManager中。

这个读取的协议类似于：BitTorrent数据传输协议。其读取过程的示意下图所示：



从上图可以看出，Executor4中的任务需要使用广播变量，但它只有该变量的b4数据块。此时，它首先从同主机（worker2节点）的中获取数据，获取到数据块b3；然后分别从不同主机的Executor1和Executor2中读取数据块b1和b2。此时，Executor4就获取到变量b的全部数据块了，然后把这些数据块在自己的BlockManager中保存一份。此时，其他Executor就可以从Executor4中读取数据了。

当完成这些操作后，各个Executor端的BlockManager（slave端）会向Driver端的BlockManager（master端）汇报数据块的状态。

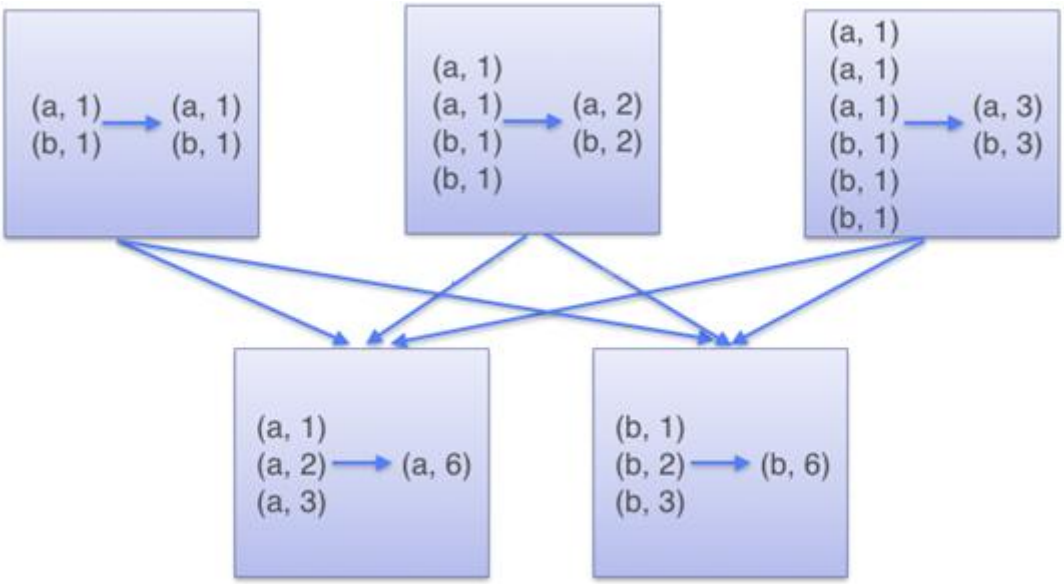
reduceByKey和groupByKey的区别和作用?

问过的一些公司：妙盈科技，字节社招，字节(2021.08)，小米x2，祖龙娱乐，米哈游 x 2，好未来，科大讯飞(2021.08)，携程(2021.09)，科大讯飞(2021.08)，蔚来(2021.09)

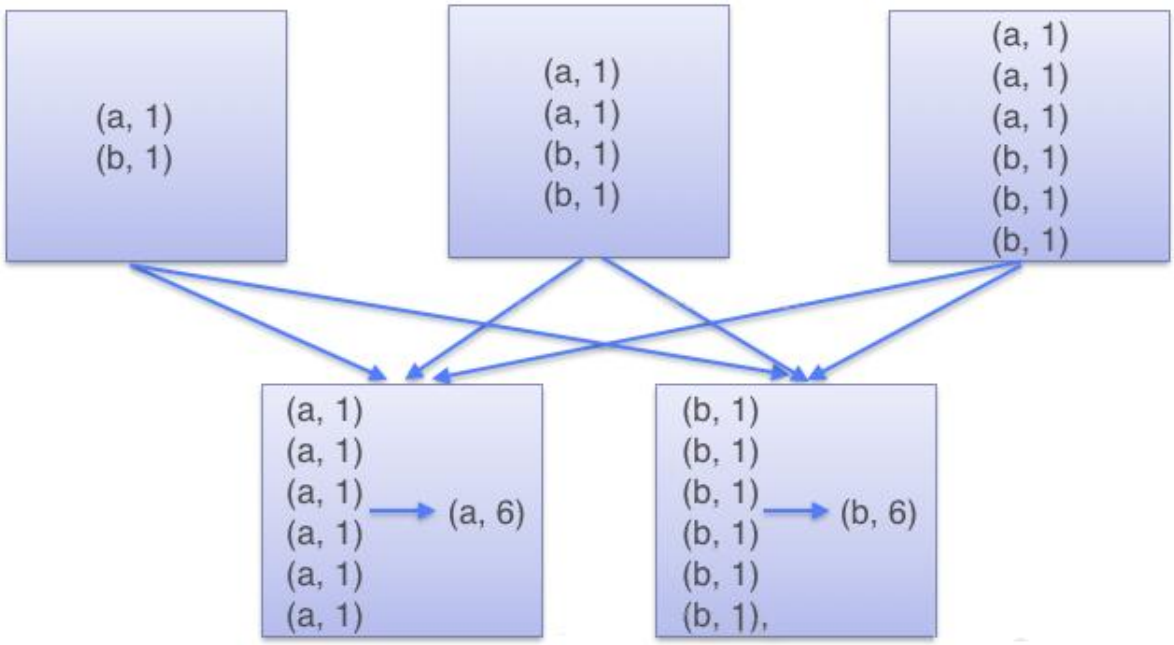
参考答案：

reduceByKey：用于对每个 key 对应的多个 value 进行 merge 操作，最重要的是它能够在本地先进行 merge 操作，并且 merge 操作可以通过函数自定义。

公众号：旧时光大数据



groupByKey：也是对每个 key 进行操作，但只生成一个 sequence， groupByKey 本身不能自定义函数，需要先用 groupByKey 生成 RDD ，然后才能对此 RDD 通过 map 进行自定义函数操作。



比较发现，使用groupByKey时，spark会将所有的键值对进行移动，不会进行局部merge，会导致集群节点之间的开销很大，导致传输延时。

reduceByKey和reduce的区别?

问过的一些公司：阿里

参考答案：

reduce：是用于一元组，遍历一元组的数据，进行处理。

公众号：旧时光大数据

```
List<Integer> data = Arrays.asList(1,2,3,4,5,6);
JavaRDD<Integer> parallelizeRdd = jsc.parallelize(data);
Integer reduceSum = parallelizeRdd.reduce(new Function2<Integer, Integer,
Integer>() {
    @Override
    public Integer call(Integer v1, Integer v2) throws Exception {
        return v1 + v2;
    }
});
System.out.println(reduceSum);

```

call一次处理为： 1+2=3 ； 3+3=9 ； 9+4=13 ； 13+5 =18 ； 18+6 =24

reduceByKey：是用于二元组When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs，可以通过function，对一个相同的key的value进行操作，得到一个U类型的值。

```
List<String> listKyes = Arrays.asList("k1", "k2", "k3","k1","k1","k2");
JavaRDD<String> keysRDD = jsc.parallelize(listKyes);
keysRDD.mapToPair(new PairFunction<String, String, Integer>() {
    @Override
    public Tuple2<String, Integer> call(String s) throws Exception {
        return new  Tuple2(s,1);
    }
}).reduceByKey(new Function2<Integer, Integer, Integer>() {
    @Override
    public Integer call(Integer v1, Integer v2) throws Exception {
        return v1 + v2;
    }
}).sortByKey(false).foreach(new VoidFunction<Tuple2<String, Integer>>() {
 //sortByKey(false) 升序
    @Override
    public void call(Tuple2<String, Integer> wordCount) throws Exception {
        System.out.println(wordCount._1 + "=====" + wordCount._2);
    }
});

```

reduceByKey的call方法对传入的是相同的key的value进行执行；

```
k1:1+1=2, 2+1 =3
k2:1+1=2
k3:1
```

使用reduceByKey出现数据倾斜怎么办?

问过的一些公司：字节

参考答案：

1）自定义分区（继承Partitioner）

2）使用累加器

3）reduceByKey本身有分区内聚合操作，出现数据倾斜可能性较小

公众号：旧时光大数据

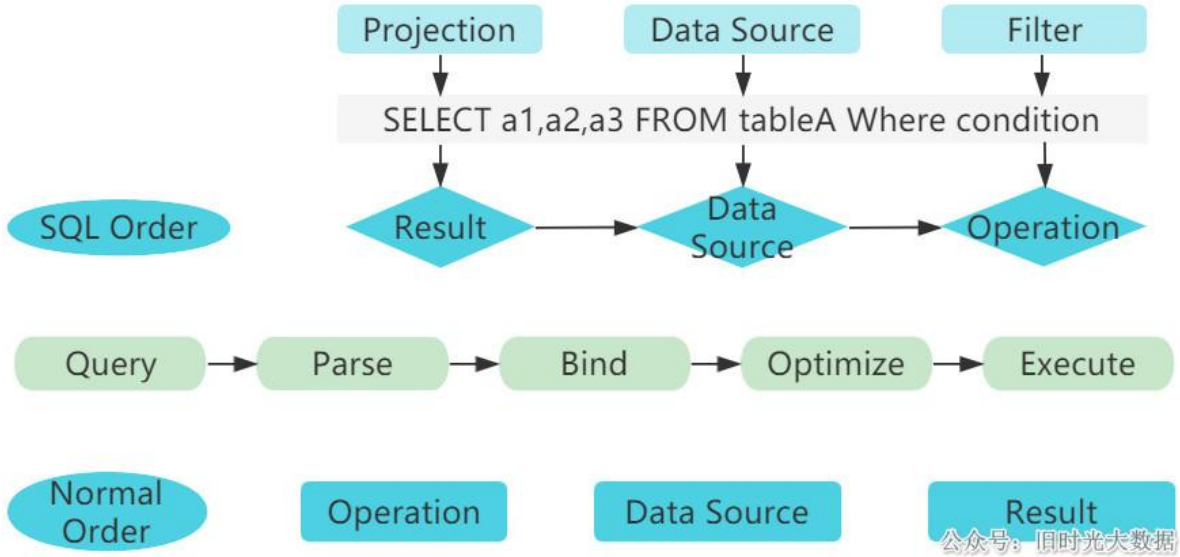
Spark SQL的执行原理?

可回答：1）Spark SQL的执行流程；2）Spark sql解析过程

问过的一些公司：阿里，顺丰，有赞(2021.08)

参考答案：

近似于关系型数据库，见下图，Spark SQL语句由Projection（a1，a2，a3）、Data Source（tableA）、Filter（condition）组成，分别对应SQL查询过程中的Result、Data Source、Operation，也就是说，SQL语句按指定次序来描述，如Result->Data Source->Operation。



执行Spark SQL语句的顺序如下：

1）对读入的SQL语句进行解析（Parse），分辨出SQL语句中的关键词（如SELECT、FROM、Where）、表达式、Projection、Data Source等，从而判断SQL语句是否规范。

2）将SQL语句和数据库的数据字典（列、表、视图等）进行绑定（Bind），如果相关的Projection、Data Source等都存在的话，就表示这个SQL语句是可以执行的。

3）选择最优计划。一般的数据库会提供几个执行计划，这些计划一般都有运行统计数据，数据库会在这些计划中选择一个最优计划（Optimize）。

4）计划执行（Execute）。计划执行案Operation->Data Source->Result的次序来执行，在执行过程中有时候甚至不需要读取物理表就可以返回结果，如重新运行执行过的SQL语句，可直接从数据库的缓冲池中获取返回结果。

Spark SQL的优化?

问过的一些公司：美团

参考答案：

1、在数据统计的时候选择高性能算子

例如Dataframe使用foreachPartitions将数据写入数据库，不要每个record都去拿一次数据库连接。通常写法是每个partition拿一次数据库连接。

```
/**
  * 将统计结果写入MySQL中
  * 代码优化：
  * 在进行数据库操作的时候，不要每个record都去操作一次数据库
```

公众号：旧时光大数据

```
    * 通常写法是每个partition操作一次数据库
    **/
  try {
    videoLogTopNDF.foreachPartition(partitionOfRecords => {
      val list = new ListBuffer[DayVideoAccessStat]
      partitionOfRecords.foreach(info => {
        val day = info.getAs[String]("day")
        val cmsId = info.getAs[Long]("cmsId")
        val times = info.getAs[Long]("times")

        list.append(DayVideoAccessStat(day, cmsId, times))
      })
      StatDao.insertDayVideoTopN(list)
    })
  }catch{
    case e:Exception =>e.printStackTrace()
  }

```

2、写数据库的时候，关闭自动提交，不要每条提交一次，自己手动每个批次提交一次

```
  var connection:Connection = null
     var pstmt : PreparedStatement = null
     try{
       connection = MySQLUtils.getConnection()

       connection.setAutoCommit(false)//关闭自动提交
       val sql = "insert into day_video_access_topn_stat(day,cms_id,times)
  values(?,?,?)"
       pstmt = connection.prepareStatement(sql)

       for(ele <- list){
         pstmt.setString(1,ele.day)
         pstmt.setLong(2,ele.cmsId)
         pstmt.setLong(3,ele.times)
         //加入到批次中，后续再执行批量处理 这样性能会好很多
         pstmt.addBatch()
       }
       //执行批量处理
       pstmt.executeBatch()

       connection.commit() //手工提交

     }catch {
       case e :Exception =>e.printStackTrace()
     }finally {
       MySQLUtils.release(connection,pstmt)
     }

```

3、复用已有的数据

三个统计方法都是只要当天的视频数据，所以在调用方法前过滤出当天视频数据，缓存到内存中。然后传到三个统计方法中使用。不要在每个统计方法都去做一次相同的过滤。

```
  val logDF = spark.read.format("parquet")
        .load("file:///F:\\mc\\SparkSQL\\data\\afterclean")
     val day = "20170511"
```

```
4
5
6       /**
7         * 代码优化：复用已有数据
8         * 既然每次统计都是统计的当天的视频，
9         * 先把该数据拿出来，然后直接传到每个具体的统计方法中
10        * 不要在每个具体的统计方法中都执行一次同样的过滤
11        *
12        * 用$列名得到列值，需要隐式转换 import spark.implicits._
13        * */
14      import spark.implicits._
15      val dayVideoDF = logDF.filter($"day" ===day&&$"cmsType"==="video")
16      /**
17        * 将这个在后文中会复用多次的dataframe缓存到内存中
18        * 这样后文在复用的时候会快很多
19        *
20        * default storage level (`MEMORY_AND_DISK`).
21        * */
22      dayVideoDF.cache()
23
24      //logDF.printSchema()
25      //logDF.show()
26
27      StatDao.deletaDataByDay(day)
28
29      //统计每天最受欢迎(访问次数)的TopN视频产品
30      videoAccessTopNStatDFAPI(spark,dayVideoDF)
31
32      //按照地势统计每天最受欢迎(访问次数)TopN视频产品 每个地市只要最后欢迎的前三个
33      cityAccessTopNStat(spark,dayVideoDF)
34
35      //统计每天最受欢迎(流量)TopN视频产品
36      videoTrafficsTopNStat(spark,dayVideoDF)
37
38      //清除缓存
39      dayVideoDF.unpersist(true)
40
```

4、根据实际情况调整并行度：spark.sql.shuffle.partitions

5、不必要的情况下，关闭分区字段类型自动推导

说下Spark checkpoint

可回答：Spark Streaming的checkpoint

问过的一些公司：顺丰，竞技世界，哈啰

参考答案：

1、先看下checkpoint到底是什么？

1）Spark在生产环境下经常会面临Tranformations的RDD非常多（例如，一个Job中包含10 000个RDD）或者具体Tranformation产生的RDD本身计算特别复杂和耗时（例如，计算时常超过1h），此时我们必须考虑对计算结果数据的持久化。

公众号：旧时光大数据

2）Spark擅长多步骤迭代，同时擅长基于Job的复用，这时如果能够对曾经计算的过程产生的数据进行复用，就可以极大地提升效率。

3）如果采用persist把数据放在内存中，虽然是最快速的，但是也是最不可靠的。如果放在磁盘上，也不是完全可靠的。例如，磁盘会损坏，管理员可能清空磁盘等。

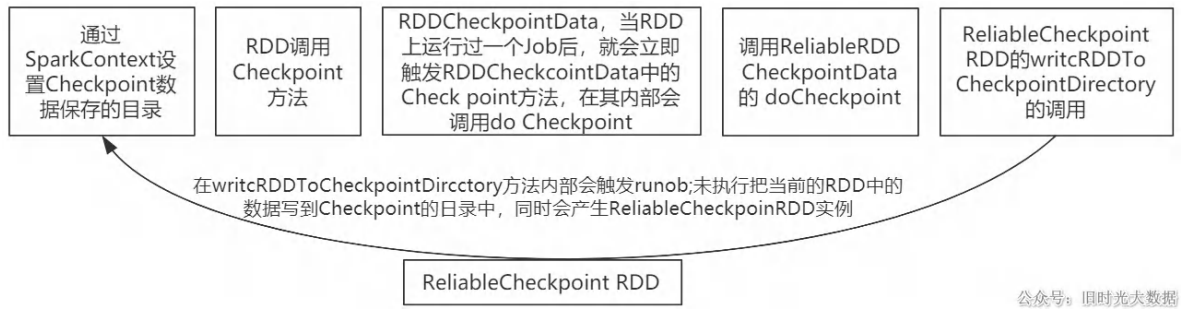
4）checkpoint的产生就是为了相对更加可靠地持久化数据，checkpoint可以指定把数据放在本地并且是多副本的方式，但是在正常的生产情况下是放在HDFS，这就自然地借助HDFS高容错、高可靠的特征完成了最大化的、可靠的持久化数据的方式。

5）为确保RDD复用计算的可靠性，checkpoint把数据持久化到HDFS中，保证数据最大程度的安全性。

6）checkpoint就是针对整个RDD计算链条中特别需要数据持久化的环节（后面会反复使用当前环节的RDD）开始基于HDFS等的数据持久化复用策略，通过对RDD启动checkpoint机制来实现容错和高可用。

至于Spark中**checkpoint源码详解**可以参考《Spark大数据商业实战三部曲》9.2小结，讲的很详细

2、checkpoint运行流程图如下图所示



通过SparkContext设置Checkpoint数据保存的目录，RDD调用checkpoint方法，生产RDDCheckpointData，当RDD上运行一个Job后，就会立即触发RDDCheckpointData中的checkpoint方法，在其内部会调用doCheckpoint；然后调用ReliableRDDCheckpointData的doCheckpoint；ReliableCheckpointRDD的writeRDDToCheckpointDirectory的调用；在writeRDDToCheckpointDirectory方法内部会触发runJob，来执行把当前的RDD中的数据写到Checkpoint的目录中，同时会产生ReliableCheckpointRDD实例。

checkpoint保存在HDFS中，具有多个副本；persist保存在内存中或者磁盘中。在Job作业调度的时候，checkpoint沿着finalRDD的“血统”关系lineage从后往前回溯向上查找，查找哪些RDD曾标记为要进行checkpoint，标记为checkpointInProgress；一旦进行checkpoint，RDD所有父RDD就被清空。

Spark SQL与DataFrame的使用?

问过的一些公司：腾讯

参考答案：

Spark SQL

官网描述：Spark SQL is Apache Spark's module for working with structured data.

也就是说Spark SQL是Spark用于结构化数据（structured data）处理的模块。它能让SQL查询跟Spark程序无缝连接使用。

数据兼容方面：Spark SQL不但兼容Hive，还可以从RDD、parquet文件、JSON文件中获取数据，未来版本甚至支持获取RDBMS数据以及cassandra等NOSQL数据；

性能优化方面：除了采取In-Memory Columnar Storage、byte-code generation等优化技术外、将会引进Cost Model对查询进行动态评估、获取最佳物理计划等等；

公众号：旧时光大数据

组件扩展方面：无论是SQL的语法解析器、分析器还是优化器都可以重新定义，进行扩展。

对于开发人员来讲，Spark SQL可以简化RDD的开发，提高开发效率，且执行效率非常快，所以实际工作中，基本上采用的就是Spark SQL。Spark SQL为了简化RDD的开发，提高开发效率，提供了2个编程抽象：DataFrame和DataSet。

DataFrame

在Spark中，DataFrame是一种以RDD为基础的分布式数据集，类似于传统数据库中的二维表格。DataFrame与RDD的主要区别在于，前者带有schema元信息，即DataFrame所表示的二维表数据集的每一列都带有名称和类型。这**使得Spark SQL得以洞察更多的结构信息，从而对藏于DataFrame背后的数据源以及作用于DataFrame之上的变换进行了针对性的优化，最终达到大幅提升运行时效率的目标。**反观RDD，由于无从得知所存数据元素的具体内部结构，Spark Core只能在stage层面进行简单、通用的流水线优化。

同时，与Hive类似，DataFrame也支持嵌套数据类型（struct、array和map）。从API易用性的角度上看，DataFrame API提供的是一套高层的关系操作，比函数式的RDD API要更加友好，门槛更低。

Spark SQL的DataFrame API允许我们使用DataFrame而不用必须去注册临时表或者生成SQL表达式。DataFrame API既有transformation操作也有action操作。

DataFrame优于RDD，因为它提供了内存管理和优化的执行计划。总结为一下两点：

1）自定义内存管理：当数据以二进制格式存储在堆外内存时，会节省大量内存。除此之外，没有垃圾回收（GC）开销。还避免了昂贵的Java序列化。因为数据是以二进制格式存储的，并且内存的schema是已知的。

2）优化执行计划：这也称为查询优化器。可以为查询的执行创建一个优化的执行计划。优化执行计划完成后最终将在RDD上运行执行。

DataSet

DataSet是分布式数据集合。DataSet是Spark1.6中添加的一个新抽象，是DataFrame的一个扩展。它提供了RDD的优势（强类型，使用强大的lambda函数的能力）以及SparkSQL优化执行引擎的优点。DataSet也可以使用功能性的转换（操作map，flatMap，filter等等）。

Spark sql自定义函数？怎么创建DataFrame？

问过的一些公司：腾讯

参考答案：

1、自定义函数

UDF(User-Defined-Function)，即最基本的自定义函数，类似to_char,to_date等

UDAF（User- Defined Aggregation Funcation），用户自定义聚合函数，类似在group by之后使用的sum,avg等

UDTF(User-Defined Table-Generating Functions),用户自定义生成函数，有点像stream里面的flatMap

2、自定义函数的使用——UDF

1）定义case class

```
case class
Emp(empno:Int,ename:String,job:String,mgr:String,hiredate:String,sal:Int,
comm:String,deptno:Int)

```

公众号：旧时光大数据

2）导入emp.csv的文件

```
val lineRDD = sc.textFile("/emp.csv").map(_.split(","))

```

3）生成DataFrame

```
val allEmp =
lineRDD.map(x=>Emp(x(0).toInt,x(1),x(2),x(3),x(4),x(5).toInt,x(6),x(7).toInt))
val empDF = allEmp.toDF

```

4）注册成一个临时视图

```
empDF.createOrReplaceTempView("emp")

```

5）自定义一个函数，拼加字符串

```
spark.sqlContext.udf.register("concatstr",(s1:String,s2:String)=>s1+"***"+s2)

```

6）调用自定义函数，将ename和job这两个字段拼接在一起

```
spark.sql("select concatstr(ename,job) from emp").show

```

3、自定义函数的使用——UDAF

强类型的Dataset和弱类型的DataFrame都提供了相关的聚合函数， 如 count()， countDistinct()， avg()，max()，min()。除此之外，用户可以设定自己的自定义聚合函数。通过继承UserDefinedAggregateFunction来实现用户自定义弱类型聚合函数。从Spark3.0版本后，UserDefinedAggregateFunction已经不推荐使用了。可以统一采用强类型聚合函数Aggregator。

这里以计算平均工资为例，多种实现方式

1）RDD方式

```
val conf: SparkConf = new SparkConf().setAppName("app").setMaster("local[*]")
val sc: SparkContext = new SparkContext(conf)
val res: (Int, Int) = sc.makeRDD(List(("zhangsan", 20), ("lisi", 30), ("wangw",
40))).map {
  case (name, age) => {
    (age, 1)
  }
}.reduce {
  (t1, t2) => {
    (t1._1 + t2._1, t1._2 + t2._2)
  }
}
println(res._1/res._2)
// 关闭连接
sc.stop()

```

2）累加器

```
class MyAC extends AccumulatorV2[Int,Int]{
  var sum:Int = 0
```

公众号：旧时光大数据

```
  var count:Int = 0
  override def isZero: Boolean = {
    return sum ==0 && count == 0
  }

  override def copy(): AccumulatorV2[Int, Int] = {
    val newMyAc = new MyAC
    newMyAc.sum = this.sum
    newMyAc.count = this.count
    newMyAc
  }

  override def reset(): Unit = {
    sum =0
    count = 0
  }

  override def add(v: Int): Unit = {
    sum += v
    count += 1
  }

  override def merge(other: AccumulatorV2[Int, Int]): Unit = {
    other match {
      case o:MyAC=>{
        sum += o.sum
        count += o.count
      }
      case _=>
    }

  }

  override def value: Int = sum/count
}

```

3）UDAF-弱类型

```
/*
定义类继承UserDefinedAggregateFunction，并重写其中方法
*/
class MyAveragUDAF extends UserDefinedAggregateFunction {

  // 聚合函数输入参数的数据类型
  def inputSchema: StructType =
StructType(Array(StructField("age",IntegerType)))

  // 聚合函数缓冲区中值的数据类型(age,count)
  def bufferSchema: StructType = {
    StructType(Array(StructField("sum",LongType),StructField("count",LongType)))
  }

  // 函数返回值的数据类型
  def dataType: DataType = DoubleType

  // 稳定性：对于相同的输入是否一直返回相同的输出。
  def deterministic: Boolean = true
```

公众号：旧时光大数据

```
19
20    // 函数缓冲区初始化
21    def initialize(buffer: MutableAggregationBuffer): Unit = {
22      // 存年龄的总和
23      buffer(0) = 0L
24      // 存年龄的个数
25      buffer(1) = 0L
26    }
27
28    // 更新缓冲区中的数据
29    def update(buffer: MutableAggregationBuffer,input: Row): Unit = {
30      if (!input.isNullAt(0)) {
31        buffer(0) = buffer.getLong(0) + input.getInt(0)
32        buffer(1) = buffer.getLong(1) + 1
33      }
34    }
35
36    // 合并缓冲区
37    def merge(buffer1: MutableAggregationBuffer,buffer2: Row): Unit = {
38      buffer1(0) = buffer1.getLong(0) + buffer2.getLong(0)
39      buffer1(1) = buffer1.getLong(1) + buffer2.getLong(1)
40    }
41
42    // 计算最终结果
43    def evaluate(buffer: Row): Double = buffer.getLong(0).toDouble /
    buffer.getLong(1)
44  }
45
46  。。。
47
48  //创建聚合函数
49  var myAverage = new MyAveragUDAF
50
51  //在spark中注册聚合函数
52  spark.udf.register("avgAge",myAverage)
53
54  spark.sql("select avgAge(age) from user").show()
55
```

4）UDAF-强类型

```
1   //输入数据类型
2   case class User01(username:String,age:Long)
3   //缓存类型
4   case class AgeBuffer(var sum:Long,var count:Long)
5
6   /**
7     * 定义类继承org.apache.spark.sql.expressions.Aggregator
8     * 重写类中的方法
9     */
10  class MyAveragUDAF1 extends Aggregator[User01,AgeBuffer,Double]{
11    override def zero: AgeBuffer = {
12      AgeBuffer(0L,0L)
13    }
14
15    override def reduce(b: AgeBuffer, a: User01): AgeBuffer = {
16      b.sum = b.sum + a.age
17      b.count = b.count + 1
```

公众号：旧时光大数据

```
    b
  }

  override def merge(b1: AgeBuffer, b2: AgeBuffer): AgeBuffer = {
    b1.sum = b1.sum + b2.sum
    b1.count = b1.count + b2.count
    b1
  }

  override def finish(buff: AgeBuffer): Double = {
    buff.sum.toDouble/buff.count
  }
  //DataSet默认额编解码器，用于序列化，固定写法
  //自定义类型就是product自带类型根据类型选择
  override def bufferEncoder: Encoder[AgeBuffer] = {
    Encoders.product
  }

  override def outputEncoder: Encoder[Double] = {
    Encoders.scalaDouble
  }
}

。。。

//封装为DataSet
val ds: Dataset[User01] = df.as[User01]

//创建聚合函数
var myAgeUdaf1 = new MyAveragUDAF1
//将聚合函数转换为查询的列
val col: TypedColumn[User01, Double] = myAgeUdaf1.toColumn

//查询
ds.select(col).show()

```

Spark3.0版本可以采用强类型的Aggregate方式代替UserDefinedAggregateFunction

```
// TODO 创建UDAF函数
val udaf = new MyAvgAgeUDAF
// TODO 注册到SparkSQL中
spark.udf.register("avgAge", functions.udaf(udaf))
// TODO 在SQL中使用聚合函数

// 定义用户的自定义聚合函数
spark.sql("select avgAge(age) from user").show

// ************************************************

case class Buff( var sum:Long, var cnt:Long )
// totalage, count
class MyAvgAgeUDAF extends Aggregator[Long, Buff, Double]{
    override def zero: Buff = Buff(0,0)

    override def reduce(b: Buff, a: Long): Buff = {
        b.sum += a
        b.cnt += 1
```

公众号：旧时光大数据

```
        b
    }

    override def merge(b1: Buff, b2: Buff): Buff = {
        b1.sum += b2.sum
        b1.cnt += b2.cnt
        b1
    }

    override def finish(reduction: Buff): Double = {
        reduction.sum.toDouble/reduction.cnt
    }

    override def bufferEncoder: Encoder[Buff] = Encoders.product

    override def outputEncoder: Encoder[Double] = Encoders.scalaDouble
}

```

4、DataFrame创建方式

方法一：Spark中使用 toDF 函数创建DataFrame

通过导入(importing)Spark sql implicits, 就可以将本地序列(seq), 数组或者RDD转为DataFrame。只要这些数据的内容能指定数据类型即可。

本地seq + toDF创建DataFrame示例：

```
import sqlContext.implicits._
val df = Seq(
  (1, "First Value", java.sql.Date.valueOf("2010-01-01")),
  (2, "Second Value", java.sql.Date.valueOf("2010-02-01"))
).toDF("int_column", "string_column", "date_column")

```

注意：如果直接用toDF()而不指定列名字，那么默认列名为"*1*", " 2", ...

通过case class + toDF创建DataFrame的示例

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// this is used to implicitly convert an RDD to a DataFrame.
import sqlContext.implicits._

// Define the schema using a case class.
// Note: Case classes in Scala 2.10 can support only up to 22 fields. To work around this limit,
// you can use custom classes that implement the Product interface.
case class Person(name: String, age: Int)

// Create an RDD of Person objects and register it as a table.
val people = sc.textFile("examples/src/main/resources/people.txt").map(_.split(",")).map(p => Person(p(0), p(1).trim.toInt)).toDF()
people.registerTempTable("people")

// 使用 sqlContext 执行 sql 语句.
val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")
```

公众号：旧时光大数据

```
17
18  // 注：sql()函数的执行结果也是DataFrame，支持各种常用的RDD操作.
19  teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
20
```

方法二：Spark中使用 createDataFrame 函数创建DataFrame

在 SqlContext 中使用createDataFrame也可以创建DataFrame。跟 toDF 一样，这里创建DataFrame的数据形态也可以是本地数组或者RDD。

通过row+schema创建示例

```
1   import org.apache.spark.sql.types._
2   val schema = StructType(List(
3       StructField("integer_column", IntegerType, nullable = false),
4       StructField("string_column", StringType, nullable = true),
5       StructField("date_column", DateType, nullable = true)
6   ))
7
8   val rdd = sc.parallelize(Seq(
9     Row(1, "First Value", java.sql.Date.valueOf("2010-01-01")),
10    Row(2, "Second Value", java.sql.Date.valueOf("2010-02-01"))
11  ))
12  val df = sqlContext.createDataFrame(rdd, schema)
13
```

方法三：通过文件直接创建DataFrame

使用parquet文件创建

```
1   val df = sqlContext.read.parquet("hdfs:/path/to/file")
2
```

使用json文件创建

```
1   val df = spark.read.json("examples/src/main/resources/people.json")
2
3   df.show()
4   // +----+-------+
5   // | age|   name|
6   // +----+-------+
7   // |null|Michael|
8   // |  30|   Andy|
9   // |  19| Justin|
10  // +----+-------+
11
```

使用csv文件，spark2.0+后的版本就可以使用

```
1   //首先初始化一个SparkSession对象
2   val spark = org.apache.spark.sql.SparkSession.builder
3       .master("local")
4       .appName("Spark CSV Reader")
5       .getOrCreate;
6
7   //然后使用SparkSessions对象加载CSV成为DataFrame
8   val df = spark.read
9       .format("com.databricks.spark.csv")
10      .option("header", "true") //reading the headers
```

公众号：旧时光大数据

```
        .option("mode", "DROPMALFORMED")
        .load("csv/file/path"); //.csv("csv/file/path") //spark 2.0 api

    df.show()

```

也可以简介回答下面三点：

1）从Spark数据源进行创建

2）从RDD进行转换

3）从Hive Table进行查询返回

HashPartitioner和RangePartitioner的实现

问过的一些公司：网易云音乐

参考答案：

1、Spark分区器

在Spark中分区器直接决定了RDD中分区的个数，RDD中每条数据经过Shuffle过程属于哪个分区以及Reduce的个数。只有Key-Value类型的RDD才有分区的，非Key-Value类型的RDD分区的值是None的。

在Spark中，存在两类分区函数：HashPartitioner和RangePartitioner，它们都是继承自Partitioner，主要提供了每个RDD有几个分区（numPartitions）以及对于给定的值返回一个分区ID（0~numPartitions-1），也就是决定这个值是属于那个分区的。

2、HashPartitioner分区

HashPartitioner分区的原理很简单，对于给定的key，计算其hashCode，并除于分区的个数取余，最后返回的值就是这个key所属的分区ID。

3、RangePartitioner分区

从HashPartitioner分区的实现原理可以看出，其结果可能导致每个分区中数据量的不均匀。而RangePartitioner分区则尽量保证每个分区中数据量的均匀，而且分区与分区之间是有序的，但是分区内的元素是不能保证顺序的。sortByKey底层就是RangePartitioner分区器。

首先了解蓄水池抽样(Reservoir Sampling)，它能够在O(n)时间内对n个数据进行等概率随机抽取。首先构建一个可放k个元素的蓄水池，将序列的前k个元素放入蓄水池中。然后从第k+1个元素开始，以k/n的概率来替换掉蓄水池中国的某个元素即可。当遍历完所有元素之后，就可以得到随机挑选出的k个元素，复杂度为O(n)。

RangePartitioner分区器的主要作用就是将一定范围内的数映射到某一个分区内。该分区器的实现方式主要是通过两个步骤来实现的，第一步，先从整个RDD中抽取出样本数据，将样本数据排序，计算出每个分区的最大key值，形成一个Array[KEY]类型的数组变量rangeBounds；第二步，判断key在rangeBounds中所处的范围，给出该key的分区ID。

RangePartitioner的重点是在于构建rangeBounds数组对象，主要步骤是：

1）计算总体的数据抽样大小sampleSize，计算规则是：(math.min(20.0 * partitions, 1e6))，至少每个分区抽取20个数据或者最多1M的数据量；

2）根据sampleSize和分区数量计算每个分区的数据抽样样本数量sampleSizePrePartition(math.ceil(3.0 * sampleSize / rdd.partitions.length).toInt)，即每个分区抽取的数据量一般会比之前计算的大一点)；

3）调用RangePartitioner的sketch函数进行数据抽样，计算出每个分区的样本；

公众号：旧时光大数据

4）计算样本的整体占比以及数据量过多的数据分区，防止数据倾斜；

5）对于数据量比较多的RDD分区调用RDD的sample函数API重新进行数据抽取；

6）将最终的样本数据通过RangePartitoner的determineBounds函数进行数据排序分配，计算出rangeBounds。

RangePartitioner的sketch函数的作用是对RDD中的数据按照需要的样本数据量进行数据抽取，主要调用SamplingUtils类的reservoirSampleAndCount方法对每个分区进行数据抽取，抽取后计算出整体所有分区的数据量大小；reservoirSampleAndCount方法的抽取方式是先从迭代器中获取样本数量个数据(顺序获取), 然后对剩余的数据进行判断，替换之前的样本数据，最终达到数据抽样的效果。RangePartitioner的determineBounds函数的作用是根据样本数据记忆权重大小确定数据边界。

4、自定义分区器

可以通过扩展Spark中的默认Partitioner类来自定义我们需要的分区数以及应该存储在这些分区中的内容。然后通过partitionBy()在RDD上应用自定义分区逻辑

Spark的水塘抽样

问过的一些公司：网易云音乐

参考答案：

问题定义可以简化如下：在不知道文件总行数的情况下，如何从文件中随机的抽取一行？

首先想到的是我们做过类似的题目吗?当然，在知道文件行数的情况下，我们可以很容易的用C运行库的rand函数随机的获得一个行数，从而随机的取出一行，但是，当前的情况是不知道行数，这样如何求呢？我们需要一个概念来帮助我们做出猜想，来使得对每一行取出的概率相等，也即随机。这个概念即蓄水池抽样（Reservoir Sampling）。

水塘抽样算法（Reservoir Sampling）思想：

在序列流中取一个数，如何确保随机性，即取出某个数据的概率为：1/(已读取数据个数)

假设已经读取n个数，现在保留的数是Ax，取到Ax的概率为(1/n)。

对于第n+1个数An+1，以1/(n+1)的概率取An+1，否则仍然取Ax。依次类推，可以保证取到数据的随机性。

数学归纳法证明如下：当n=1时，显然，取A1。取A1的概率为1/1。

```
1    假设当n=k时，取到的数据Ax。取Ax的概率为1/k。
2    当n=k+1时，以1/(k+1)的概率取An+1，否则仍然取Ax。
3
```

1）如果取Ak+1，则概率为1/(k+1)；

2）如果仍然取Ax，则概率为(1/k)*(k/(k+1))=1/(k+1)

所以，对于之后的第n+1个数An+1，以1/(n+1)的概率取An+1，否则仍然取Ax。依次类推，可以保证取到数据的随机性。

在序列流中取k个数，如何确保随机性，即取出某个数据的概率为：k/(已读取数据个数)

建立一个数组，将序列流里的前k个数，保存在数组中。(也就是所谓的”蓄水池”)

对于第n个数An，以k/n的概率取An并以1/k的概率随机替换“蓄水池”中的某个元素；否则“蓄水池”数组不变。依次类推，可以保证取到数据的随机性。

公众号：旧时光大数据

数学归纳法证明如下：

当n=k是，显然“蓄水池”中任何一个数都满足，保留这个数的概率为k/k。

假设当n=m(m>k)时，“蓄水池”中任何一个数都满足，保留这个数的概率为k/m。

当n=m+1时，以k/(m+1)的概率取An，并以1/k的概率，随机替换“蓄水池”中的某个元素，否则“蓄水池”数组不变。则数组中保留下来的数的概率为：

所以，对于第n个数An，以k/n的概率取An并以1/k的概率随机替换“蓄水池”中的某个元素；否则“蓄水池”数组不变。依次类推，可以保证取到数据的随机性。

DAGScheduler、TaskScheduler、SchedulerBackend实现原理

问过的一些公司：网易云音乐

参考答案：

Scheduler任务调度器模块作为Spark的核心部件，涉及三个重要的类：

- org.apache.spark.scheduler.DAGScheduler
- org.apache.spark.scheduler.SchedulerBackend
- org.apache.spark.scheduler.TaskScheduler

1、DAGScheduler

DAGScheduler是面向Stage的高层级的调度器，DAGScheduler把DAG拆分成很多的Tasks，每组的Tasks都是一个Stage，解析时是以Shuffle为边界反向解析构建Stage，每当遇到Shuffle，就会产生新的Stage，然后以一个个TaskSet（每个Stage封装一个TaskSet）的形式提交给底层调度器TaskScheduler。
DAGScheduler需要记录哪些RDD被存入磁盘等物化动作，同时要寻求Task的最优化调度，如在Stage内部数据的本地性等。DAGScheduler还需要监视因为Shuffle跨节点输出可能导致的失败，如果发现这个Stage失败，可能就要重新提交该Stage。

2、TaskScheduler

TaskScheduler的核心任务是提交TaskSet到集群运算并汇报结果。

1）为TaskSet创建和维护一个TaskSetManager，并追踪任务的本地性以及错误信息。

2）遇到Straggle任务时，会放到其他节点进行重试。

3）向DAGScheduler汇报执行情况，包括在Shuffle输出丢失的时候报告fetch failed错误等信息。

DAGScheduler将划分的一系列的Stage（每个Stage封装一个TaskSet），按照Stage的先后顺序依次提交给底层的TaskScheduler去执行。

DAGScheduler在SparkContext中实例化的时候，TaskScheduler以及SchedulerBackend就已经先在SparkContext的createTaskScheduler创建出实例对象了。

虽然Spark支持多种部署模式（包括Local、Standalone、YARN、Mesos等），但是底层调度器TaskScheduler接口的实现类都是TaskSchedulerImpl。

这里只对Standalone部署模式下的具体实现StandaloneSchedulerBackend来作分析。

TaskSchedulerImpl在createTaskScheduler方法中实例化后，就立即调用自己的initialize方法把StandaloneSchedulerBackend的实例对象传进来，从而赋值给TaskSchedulerImpl的backend。在TaskSchedulerImpl的initialize方法中，根据调度模式的配置创建实现了SchedulerBuilder接口的相应的实例对象，并且创建的对象会立即调用buildPools创建相应数量的Pool存放和管理TaskSetManager的实例对象。实现SchedulerBuilder接口的具体类都是SchedulerBuilder的内部类。

公众号：旧时光大数据

1）FIFOSchedulableBuilder：调度模式是SchedulingMode.FIFO，使用先进先出策略调度。先进先出（FIFO）为默认模式。在该模式下只有一个TaskSetManager池。

2）FairSchedulableBuilder：调度模式是SchedulingMode.FAIR，使用公平策略调度。

在createTaskScheduler方法返回后，TaskSchedulerImpl通过DAGScheduler的实例化过程设置DAGScheduler的实例对象。然后调用自己的start方法。在TaskSchedulerImpl调用start方法的时候，会调用StandaloneSchedulerBackend的start方法，在StandaloneSchedulerBackend的start方法中，会最终注册应用程序AppClient。TaskSchedulerImpl的start方法中还会根据配置判断是否周期性地检查任务的推测执行。

TaskSchedulerImpl启动后，就可以接收DAGScheduler的submitMissingTasks方法提交过来的TaskSet进行进一步处理。TaskSchedulerImpl在submitTasks中初始化一个TaskSetManager，对其生命周期进行管理，当TaskSchedulerImpl得到Worker节点上的Executor计算资源的时候，会通过TaskSetManager发送具体的Task到Executor上执行计算。

如果Task执行过程中有错误导致失败，会调用TaskSetManager来处理Task失败的情况，进而通知DAGScheduler结束当前的Task。TaskSetManager会将失败的Task再次添加到待执行Task队列中。Spark Task允许失败的次数默认是4次，在TaskSchedulerImpl初始化的时候，通过spark.task.maxFailures设置该值。

如果Task执行完毕，执行的结果会反馈给TaskSetManager，由TaskSetManager通知DAGScheduler，DAGScheduler根据是否还存在待执行的Stage，继续迭代提交对应的TaskSet给TaskScheduler去执行，或者输出Job的结果。

通过下面的调度链，Executor把Task执行的结果返回给调度器（Scheduler）。

（1）Executor.run。

（2）CoarseGrainedExecutorBackend.statusUpdate（发送StatusUpdate消息）。

（3）CoarseGrainedSchedulerBackend.receive（处理StatusUpdate消息）。

（4）TaskSchedulerImpl.statusUpdate。

（5）TaskResultGetter.enqueueSuccessfulTask或者enqueueFailedTask。

（6）TaskSchedulerImpl.handleSuccessfulTask或者handleFailedTask。

（7）TaskSetManager.handleSuccessfulTask或者handleFailedTask。

（8）DAGScheduler.taskEnded。

（9）DAGScheduler.handleTaskCompletion。

在上面的调度链中值得关注的是：第（7）步中，TaskSetManager的handleFailedTask方法会将失败的Task再次添加到待执行Task队列中。在第（6）步中，TaskSchedulerImpl的handleFailedTask方法在TaskSetManager的handleFailedTask方法返回后，会调用CoarseGrainedSchedulerBackend的reviveOffers方法给重新执行的Task获取资源。

3、SchedulerBackend

这里以Spark Standalone部署方式为例，StandaloneSchedulerBackend在启动的时候构造了StandaloneAppClient实例，并在该实例start的时候启动了ClientEndpoint消息循环体，ClientEndpoint在启动的时候会向Master注册当前程序。而StandaloneSchedulerBackend的父类CoarseGrainedSchedulerBackend在start的时候会实例化类型为DriverEndPoint（这就是程序运行时的经典的对象Driver）的消息循环体，StandaloneSchedulerBackend专门负责收集Worker上的资源信息，当ExecutorBackend启动的时候，会发送RegisteredExecutor信息向DriverEndpoint注册，此时StandaloneSchedulerBackend就掌握了当前应用程序拥有的计算资源，TaskScheduler就是通过StandaloneSchedulerBackend拥有的计算资源来具体运行Task的。

介绍下Spark client提交application后，接下来的流程?

可回答：1）Spark on Yarn流程；2）介绍下Spark yarn模式下的cluster模式和client模式，它们有什么区别？3）基于YARN集群的任务提交过程

问过的一些公司：字节，字节(2021.10)，陌陌，华为x2，网易x2，美团，欢聚(2021.09)

参考答案：

1、Spark on Standalone

1）spark集群启动后，Worker向Master注册信息；

```
    val workers = new HashSet[WorkerInfo]

```

2）spark-submit命令提交程序后，driver和application也会向Master注册信息；

```
    val waitingApps = new ArrayBuffer[ApplicationInfo]
    private val drivers = new HashSet[DriverInfo]

```

3）创建SparkContext对象：主要的对象包含DAGScheduler和TaskScheduler；

4）Driver把Application信息注册给Master后，Master会根据App信息去Worker节点启动Executor；

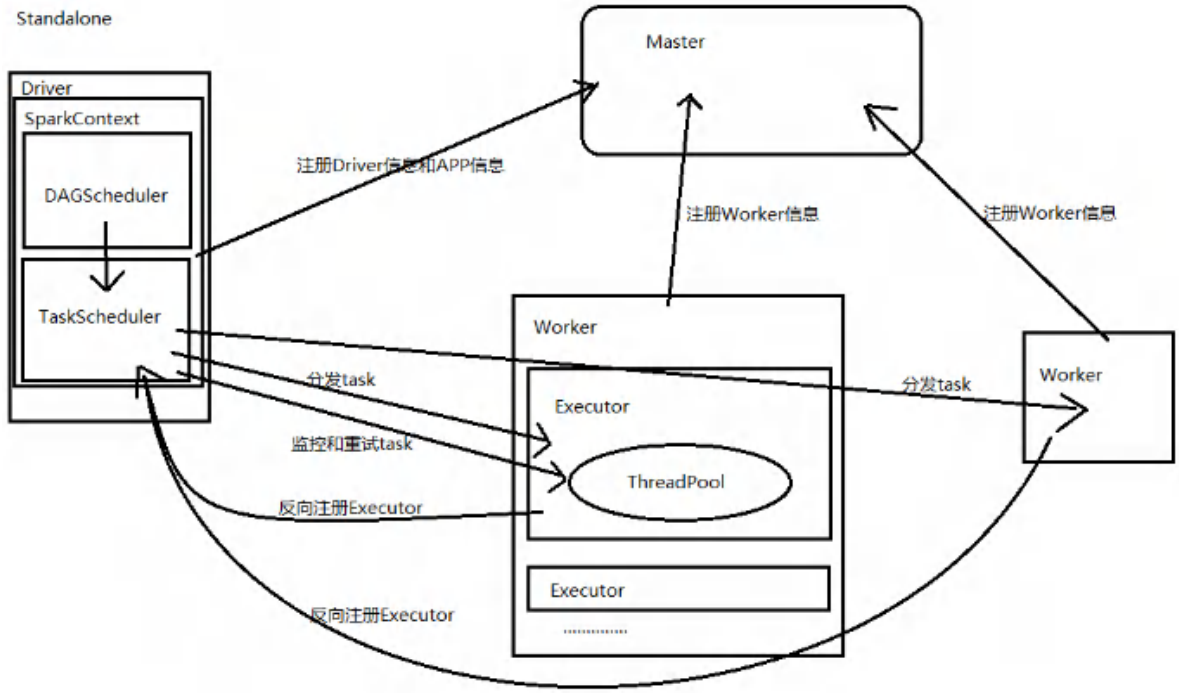
5）Executor内部会创建运行task的线程池，然后把启动的Executor反向注册给Dirver；

6）DAGScheduler：负责把Spark作业转换成Stage的DAG（Directed Acyclic Graph有向无环图），根据宽窄依赖切分Stage，然后把Stage封装成TaskSet的形式发送个TaskScheduler；同时DAGScheduler还会处理由于Shuffle数据丢失导致的失败；

7）TaskScheduler：维护所有TaskSet，分发Task给各个节点的Executor（根据数据本地化策略分发Task），监控task的运行状态，负责重试失败的task；

8）所有task运行完成后，SparkContext向Master注销，释放资源。

注：job的失败不会重试

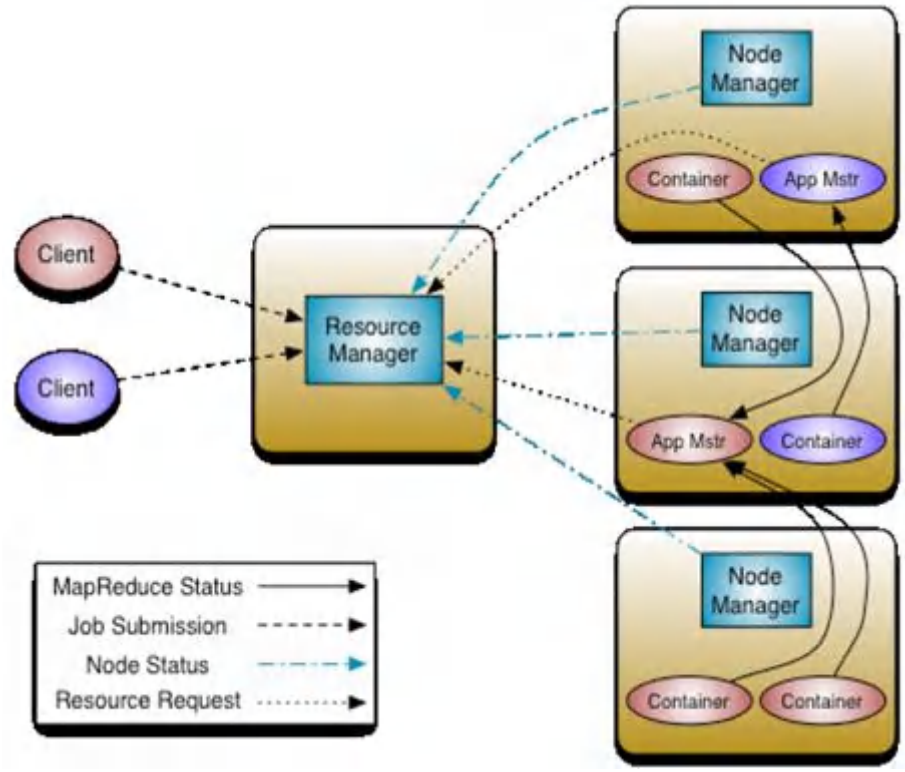


公众号：旧时光大数据

2、Spark on Yarn

yarn是一种统一的资源管理机制，可以通过队列的方式，管理运行多套计算框架。Spark on Yarn模式根据Dirver在集群中的位置分为两种模式。一种是Yarn-Client模式，另一种是Yarn-Cluster模式。

Yarn框架的基本运行流程图

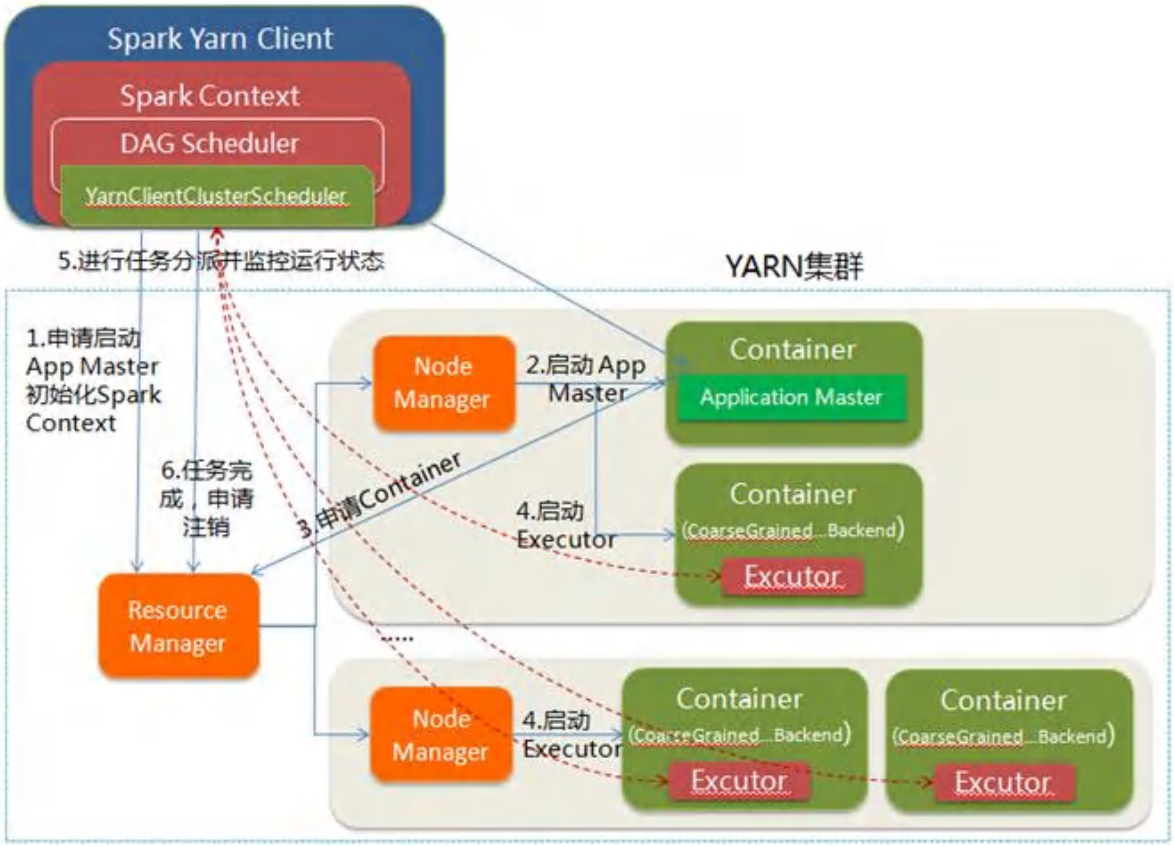


ResourceManager：负责将集群的资源分配给各个应用使用，而资源分配和调度的基本单位是Container，其中封装了集群资源（CPU、内存、磁盘等），每个任务只能在Container中运行，并且只使用Container中的资源；

NodeManager：是一个个计算节点，负责启动Application所需的Container，并监控资源的使用情况汇报给ResourceManager；

ApplicationMaster：主要负责向ResourceManager申请Application的资源，获取Container并跟踪这些Container的运行状态和执行进度，执行完后通知ResourceManager注销ApplicationMaster，ApplicationMaster也是运行在Container中。

yarn-client模式，Dirver运行在本地的客户端上。



1）client向ResouceManager申请启动ApplicationMaster，同时在SparkContext初始化中创建DAGScheduler和TaskScheduler；

2）ResouceManager收到请求后，在一台NodeManager中启动第一个Container运行ApplicationMaster；

3）Dirver中的SparkContext初始化完成后与ApplicationMaster建立通讯，ApplicationMaster向ResourceManager申请Application的资源；

4）一旦ApplicationMaster申请到资源，便与之对应的NodeManager通讯，启动Executor，并把Executor信息反向注册给Dirver；

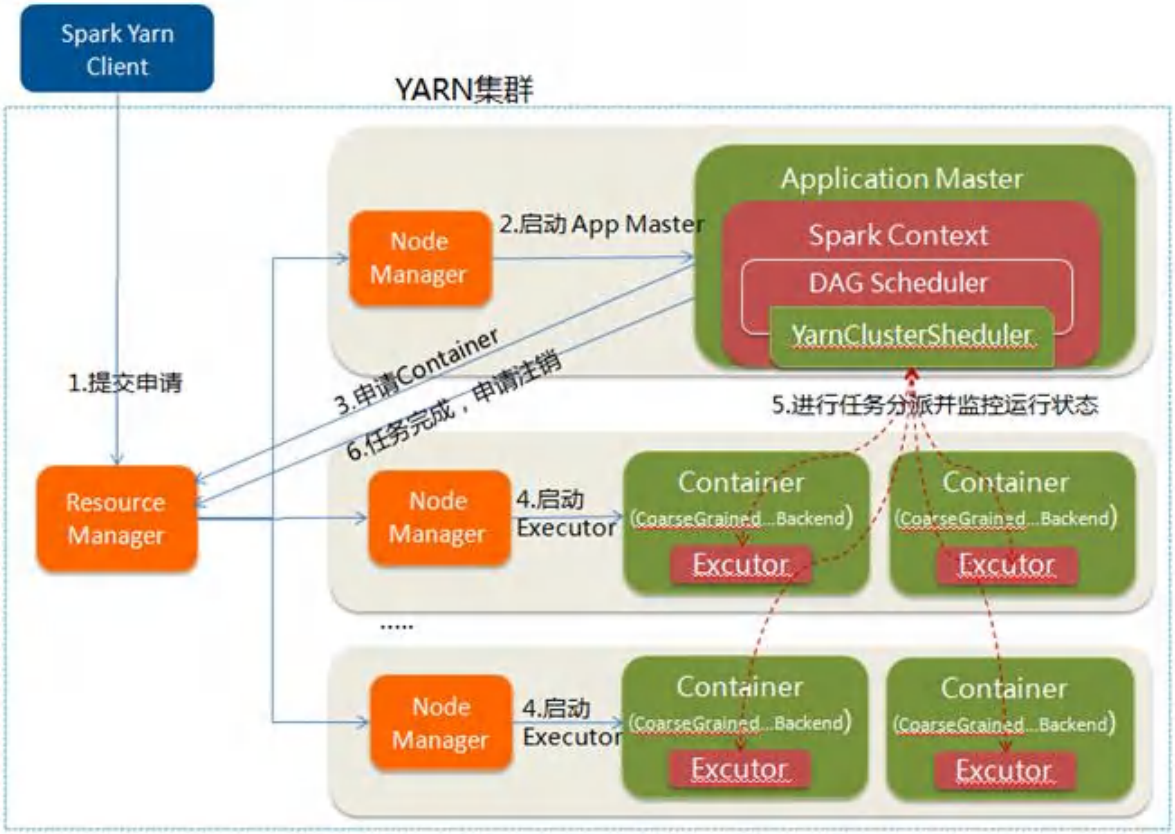
5）Dirver分发task，并监控Executor的运行状态，负责重试失败的task；

6）运行完成后，Client的SparkContext向ResourceManager申请注销并关闭自己。

yarn-cluster模式中，当用户向yarn提交应用程序后，yarn将分为两阶段运行该应用程序：

第一个阶段是把Spark的Dirver作为一个ApplicationMaster在yarn中启动；

第二个阶段是ApplicationMaster向ResourceManager申请资源，并启动Executor来运行task，同时监控task整个运行流程并重试失败的task。



Yarn-client和Yarn-cluster的区别：

yarn-cluster模式下，Dirver运行在ApplicationMaster中，负责申请资源并监控task运行状态和重试失败的task，当用户提交了作业之后就可以关掉client，作业会继续在yarn中运行；

yarn-client模式下，Dirver运行在本地客户端，client不能离开。

Spark的几种部署方式

可回答：1）Spark的提交模式；2）Spark的standlone模式；3）Spark各种运行模式的区别？

问过的一些公司：字节x2，百度，字节，京东，蔚来(2021.09)，恒生(2021.09)

参考答案：

1、本地模式

Spark Local模式被称为Local[N]模式，是用单机的多个线程来模拟Spark分布式计算，直接运行在本地，便于调试，通常用来验证开发出来的应用程序逻辑上有没有问题，其中N代表可以使用N个线程，每个线程拥有一个core。

将Spark应用以多线程的方式直接运行在本地，一般都是为了方便调试，本地模式分三类

- local：只启动一个executor
- local[k]：启动k个executor
- local[*]：启动跟cpu数目相同的executor

通常，Local模式用于完成开发出来的分布式程序的测试工作，并不用于实际生产。

2、Standalone模式

Standalone模式是Spark实现的资源调度框架，其自带完整的服务，可单独部署到一个集群中，无需依赖任何其他资源管理系统。主要的节点有Client节点、Master节点和Worker节点。其中Driver既可以运行在Master节点上中，也可以运行在本地Client端。当用spark-shell交互式工具提交Spark的Job时，Driver在Master节点上运行；当使用spark-submit工具提交Job或者在Eclipse、IDEA等开发平台上使用new SparkConf().setMaster("spark://master:7077")方式运行Spark任务时，Driver是运行在本地Client端上的。

Spark Standalone运行流程：

1）SparkContext连接到Master，向Master注册并申请资源（CPU Core 和Memory）；

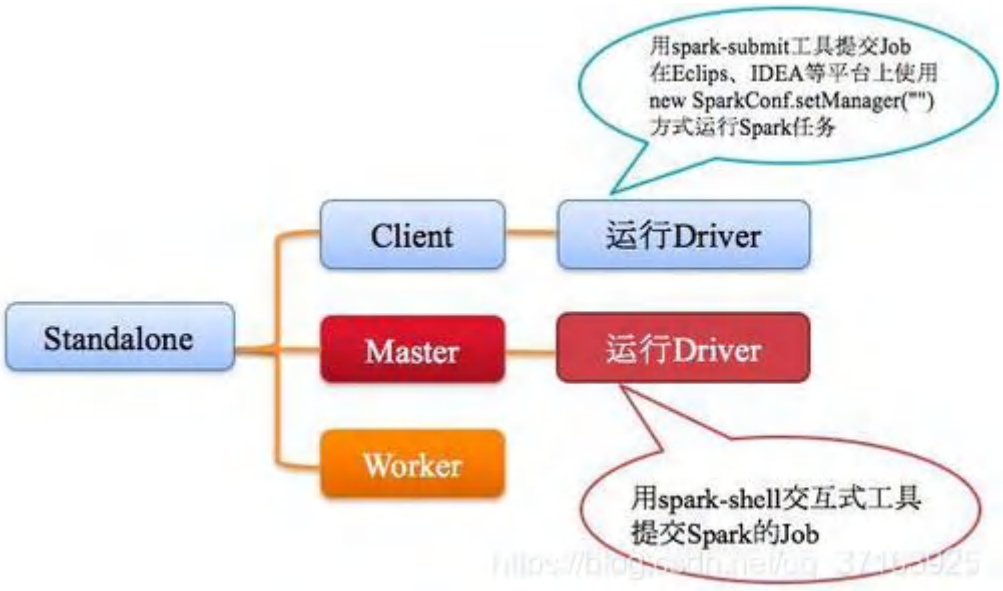
2）Master根据SparkContext的资源申请要求和Worker心跳周期内报告的信息决定在哪个Worker上分配资源，然后在该Worker上获取资源，然后启动StandaloneExecutorBackend；

3）StandaloneExecutorBackend向SparkContext注册；

4）SparkContext将Applicaiton代码发送给StandaloneExecutorBackend；并且SparkContext解析Applicaiton代码，构建DAG图，并提交给DAG Scheduler分解成Stage（当碰到Action操作时，就会催生Job；每个Job中含有1个或多个Stage，Stage一般在获取外部数据和shuffle之前产生），DAG Scheduler将TaskSet提交给Task Scheduler，Task Scheduler负责将Task分配到相应的Worker，最后提交给StandaloneExecutorBackend执行；

5）StandaloneExecutorBackend会建立Executor线程池，开始执行Task，并向SparkContext报告，直至Task完成。

6）所有Task完成后，SparkContext向Master注销，释放资源。



3、Spark On Mesos模式

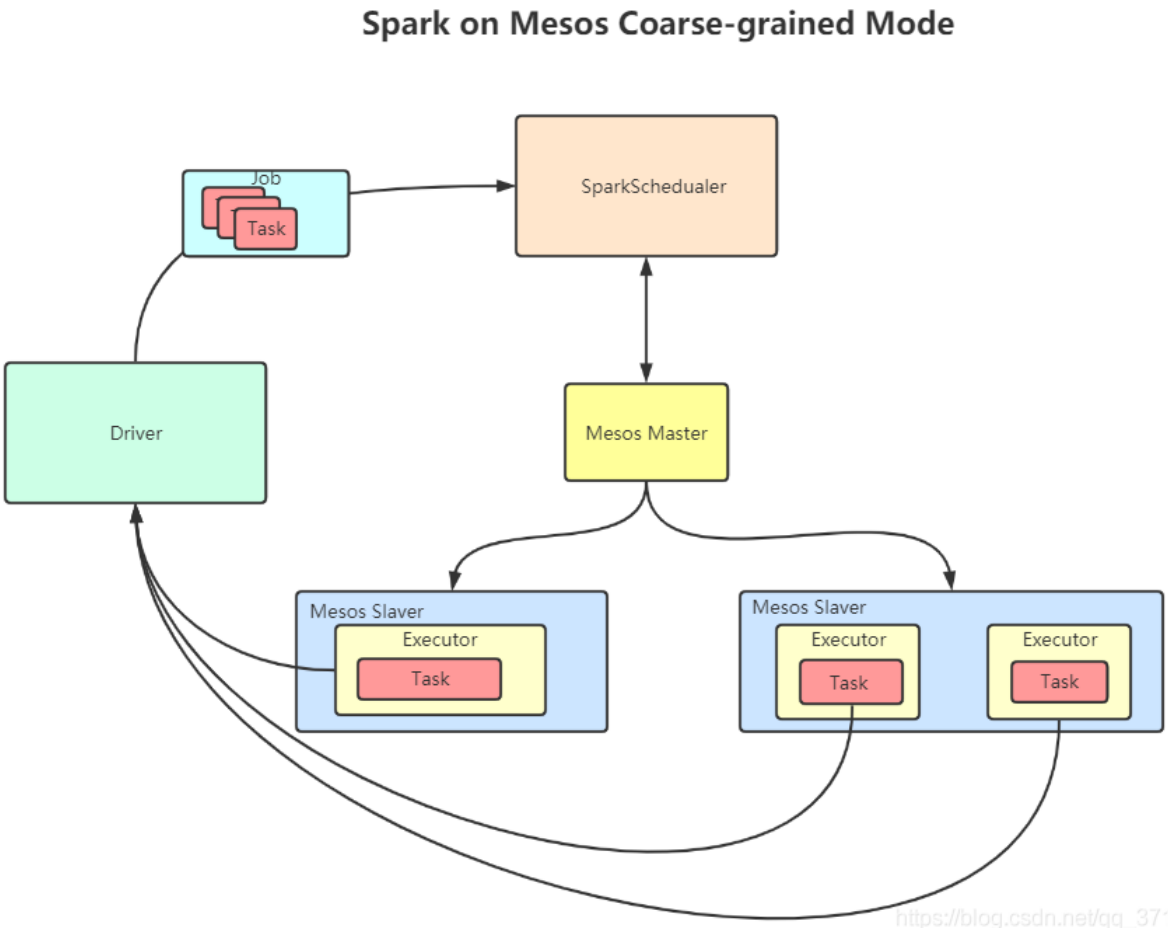
Spark On Mesos模式是很多公司采用的模式，Spark运行在Mesos上会比运行在YARN上更加灵活，更加自然。在Spark On Mesos环境中，用户可选择粗粒度模式和细粒度模式两种调度模式之一运行自己的应用程序。**其中细粒度模式于Spark2.0.0版本开始不再使用。**

目前在Spark On Mesos环境中，用户可选择两种调度模式之一运行自己的应用程序（可参考Andrew Xia的"Mesos Scheduling Mode on Spark"）：

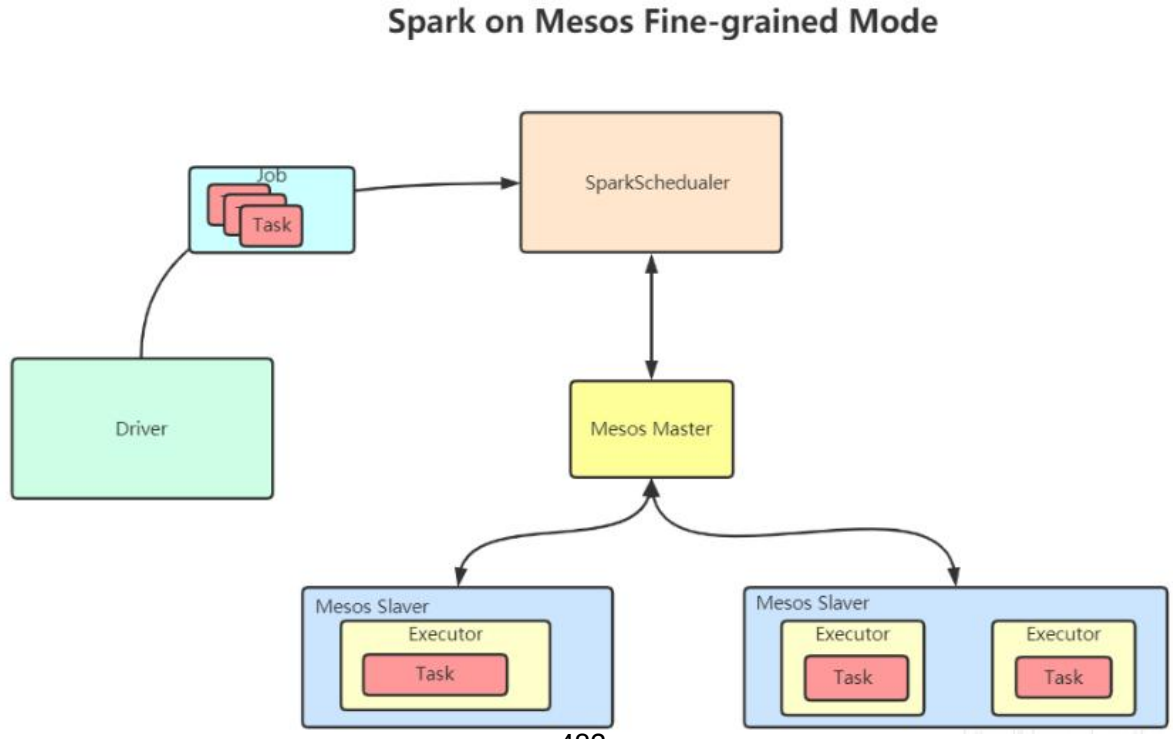
1）粗粒度模式（Coarse-grained Mode）：每个应用程序的运行环境由一个Dirver和若干个Executor组成，其中，每个Executor占用若干资源，内部可运行多个Task（对应多少个"slot"）。应用程序的各个任务正式运行之前，需要将运行环境中的资源全部申请好，且运行过程中要一直占用这些资源，即使不用，最后程序运行结束后，回收这些资源。举个例子，比如你提交应用程序时，指定使用5个executor运行你的应用程序，每个executor占用5GB内存和5个CPU，每个executor内部设置了5个slot，则Mesos需要先为executor分配资源并启动它们，之后开始调度任务。另外，在程序运行过程中，mesos的master和

公众号：旧时光大数据

slave并不知道executor内部各个task的运行情况，executor直接将任务状态通过内部的通信机制汇报给Driver，从一定程度上可以认为，每个应用程序利用mesos搭建了一个虚拟集群自己使用。



2）细粒度模式（Fine-grained Mode）：鉴于粗粒度模式会造成大量资源浪费，Spark On Mesos还提供了另外一种调度模式：细粒度模式，这种模式类似于现在的云计算，思想是按需分配。与粗粒度模式一样，应用程序启动时，先会启动executor，但每个executor占用资源仅仅是自己运行所需的资源，不需要考虑将来要运行的任务，之后，mesos会为每个executor动态分配资源，每分配一些，便可以运行一个新任务，单个Task运行完之后可以马上释放对应的资源。每个Task会汇报状态给Mesos slave和Mesos Master，便于更加细粒度管理和容错，这种调度模式类似于MapReduce调度模式，每个Task完全独立，优点是便于资源控制和隔离，但缺点也很明显，短作业运行延迟大。



公众号：旧时光大数据

4、Spark On YARN模式

这是一种很有前景的部署模式。但限于YARN自身的发展，目前仅支持粗粒度模式（Coarse-grained Mode）。这是由于YARN上的Container资源是不可以动态伸缩的，一旦Container启动之后，可使用的资源不能再发生变化，不过这个已经在YARN计划中了。

Spark on YARN的支持两种模式：

- yarn-cluster：适用于生产环境
- yarn-client：适用于交互、调试，希望立即看到app的输出

yarn-cluster和yarn-client的区别在于yarn appMaster，每个yarn app实例有一个appMaster进程，是为app启动的第一个container；负责从ResourceManager请求资源，获取到资源后，告诉NodeManager为其启动container。yarn-cluster和yarn-client模式内部实现还是有很大的区别。如果你需要用于生产环境，那么请选择yarn-cluster；而如果你仅仅是Debug程序，可以选择yarn-client。

总结：

这几种分布式部署方式各有利弊，通常需要根据实际情况决定采用哪种方案。进行方案选择时，往往要考虑公司的技术路线（采用Hadoop生态系统还是其他生态系统）、相关技术人才储备等。上面涉及到Spark的许多部署模式，究竟哪种模式好这个很难说，需要根据你的需求，如果你只是测试Spark Application，你可以选择local模式。而如果你数据量不是很多，Standalone 是个不错的选择。当你需要统一管理集群资源（Hadoop、Spark等），那么你可以选择Yarn或者mesos，但是这样维护成本就会变高。

- 从对比上看，mesos似乎是Spark更好的选择，也是被官方推荐的。
- 但如果你同时运行hadoop和Spark，从兼容性上考虑，Yarn是更好的选择。
- 如果你不仅运行了hadoop、spark。还在资源管理上运行了docker，Mesos更加通用。
- Standalone对于小规模计算集群更适合。

在Yarn-client情况下，Driver此时在哪

问过的一些公司：字节

参考答案：

在哪提交Driver就在哪

Spark的cluster模式有什么好处

问过的一些公司：字节(2021.10)

参考答案：

在yarn-cluster模式下，Dirver运行在ApplicationMaster中，也就是说Driver是运行在其它节点的，当用户提交了作业之后就可以关掉client，此时作业会继续在yarn中运行；

而yarn-client模式下，Dirver运行在本地客户端，client不能离开。

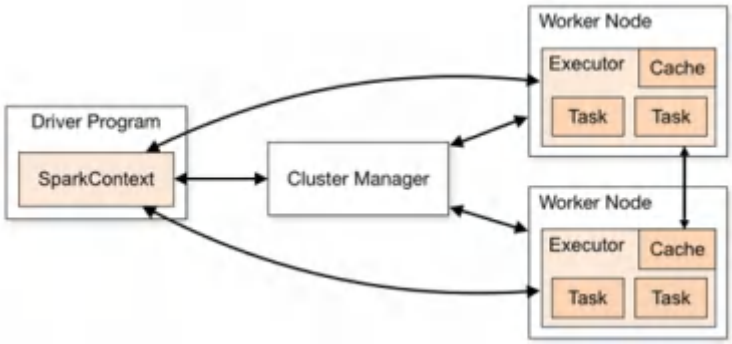
Driver怎么管理executor

问过的一些公司：字节

参考答案：

Spark框架的核心是一个计算引擎，整体来说，它采用了标准 master-slave 的结构。

如下图所示，它展示了一个 Spark执行时的基本结构。图形中的Driver表示master，负责管理整个集群中的作业任务调度。图形中的Executor 则是 slave，负责实际执行任务。



Driver是Spark驱动器节点，用于执行Spark任务中的main方法，负责实际代码的执行工作。

Spark Application的main方法（于SparkContext相关的代码）运行在Driver上，当用于计算的RDD触发Action动作之后，会提交Job，那么RDD就会向前追溯每一个transformation操作，直到初始的RDD开始，这之间的代码运行在Executor。

Driver在Spark作业执行时主要负责：

- 将用户程序转化为作业（job）
- 在Executor之间调度任务(task)
- 跟踪Executor的执行情况
- 通过UI展示查询运行情况

实际上，我们无法准确地描述Driver的定义，因为在整个的编程过程中没有看到任何有关Driver的字眼。所以简单理解，所谓的Driver就是驱使整个应用运行起来的程序，也称之为Driver类。

Spark的map和flatmap的区别?

问过的一些公司：小米，网易，米哈游，美团

参考答案：

flatMap返回的是**迭代器**中的元素。

```
In [15]: sc
Out[15]: <pyspark.context.SparkContext at 0xb602fb8c>

In [16]: rdd=sc.parallelize([1,2,3])

In [17]: rdd1=rdd.map(lambda x : x+1)

In [18]: rdd1.collect()
Out[18]: [2, 3, 4]

In [19]: rdd2=rdd.flatMap(lambda x: x+1)

In [20]: rdd2.collect()
16/03/17 00:03:19 ERROR Executor: Exception in task 1.0 in stage 6.0 (TID 13)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/qiangang/program/spark-1.6.0-bin-hadoop2.4/python/lib/pyspark.zip/pyspark/worker.py", line 11
1, in main
    process()
  File "/home/qiangang/program/spark-1.6.0-bin-hadoop2.4/python/lib/pyspark.zip/pyspark/worker.py", line 10
6, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/qiangang/program/spark-1.6.0-bin-hadoop2.4/python/lib/pyspark.zip/pyspark/serializers.py", li
ne 263, in dump_stream
    vs = list(itertools.islice(iterator, batch))
TypeError: 'int' object is not iterable
```

上面的例子说明对于传递给flatMap的函数返回的类型是一个可迭代的类型（例如list）。

现在比较一下map和flatMap接收返回值为可迭代类型的函数的区别：

```
In [29]: rdd.collect()
Out[29]: [1, 2, 3]

In [30]: rdd
Out[30]: ParallelCollectionRDD[5] at parallelize at PythonRDD.scala:423

In [31]: rdd1=rdd.map(lambda x:range(x,3))

In [32]: rdd1.collect()
Out[32]: [[1, 2], [2], []]

In [33]: rdd2=rdd.flatMap(lambda x:range(x,3))

In [34]: rdd2.collect()
Out[34]: [1, 2, 2]
```

上例说明对于返回可迭代类型的函数map与flatMap的区别在于：

map函数会对每一条输入进行指定的操作，然后为每一条输入返回一个对象；而flatMap函数则是两个操作的集合——正是“先映射后扁平化”：

- 操作1：同map函数一样：对每一条输入进行指定的操作，然后为每一条输入返回一个对象
- 操作2：最后将所有对象合并为一个对象

map：List里有小的List

flatmap：是先flat再map，只能压一次，形成一个新的List集合，把原元素放进新的集合里面

map(func)函数会对每一条输入进行指定的func操作，然后为每一条输入返回一个对象；而flatMap(func)也会对每一条输入进行执行的func操作，然后每一条输入返回一个相对，但是最后会将所有的对象再合成为一个对象；从返回的结果的数量上来讲，map返回的数据对象的个数和原来的输入数据是相同的，而flatMap返回的个数则是不同的。

map和mapPartition的区别

问过的一些公司：端点数据(2021.07)

参考答案：

Spark中的map和mapPartitions算子，在处理数据上，一些区别如下：

主要区别：

map是对rdd中的每一个元素进行操作；

mapPartitions则是对rdd中的每个分区的迭代器进行操作。

mapPartitions的优点：

如果是普通的map，比如一个partition中有1万条数据。那么function要执行和计算1万次。

使用MapPartitions操作之后，一个task仅仅会执行一次function，function一次接收所有的partition数据。只要执行一次就可以了，性能比较高。如果在map过程中需要频繁创建额外的对象（例如将rdd中的数据通过jdbc写入数据库，map需要为每个元素创建一个链接而mapPartition为每个partition创建一个链接），则mapPartitions效率比map高的多。

SparkSql或DataFrame默认会对程序进行mapPartition的优化。

总结： **MapPartiton 的性能较高**

mapPartitions的缺点：

公众号：旧时光大数据

如果是普通的map操作，一次function的执行就处理一条数据；那么如果内存不够用的情况下，比如处理了1千条数据了，那么这个时候内存不够了，那么就可以将已经处理完的1千条数据从内存里面垃圾回收掉，或者用其他方法，腾出空间来吧。所以说普通的map操作通常不会导致内存的OOM异常。

但是MapPartitions操作，对于大量数据来说，比如甚至一个partition，100万数据，一次传入一个function以后，那么可能一下子内存不够，但是又没有办法去腾出内存空间来，可能就OOM，内存溢出。

总结：**MapPartition 有可能会导致内存溢出，数据一次获取过多！！**

Spark的cache和persist的区别？它们是transformaiton算子还是action算子?

问过的一些公司：字节，作业帮社招，360-大数据开发

参考答案：

cache和persist都是用于将一个RDD进行缓存的，这样在之后使用的过程中就不需要重新计算了，可以大大节省程序运行时间。

先来看下源码：

```
1    def cache(): this.type = persist()
2
```

说明是cache()调用了persist()，再看一下persist函数：

```
1    def persist(): this.type = persist(StorageLevel.MEMORY_ONLY)
2
```

可以看到persist()内部调用了persist(StorageLevel.MEMORY_ONLY)

```
1    def persist(newLevel: StorageLevel): this.type = {
2      // TODO: Handle changes of StorageLevel
3      if (storageLevel != StorageLevel.NONE && newLevel != storageLevel) {
4        throw new UnsupportedOperationException(
5          "Cannot change storage level of an RDD after it was already assigned a
   level")
6      }
7      sc.persistRDD(this)
8      // Register the RDD with the ContextCleaner for automatic GC-based cleanup
9      sc.cleaner.foreach(_.registerRDDForCleanup(this))
10     storageLevel = newLevel
11     this
12   }
13
```

可以看出来persist有一个 StorageLevel 类型的参数，这个表示的是RDD的缓存级别。

至此便可得出cache和persist的区别了：**cache只有一个默认的缓存级别MEMORY_ONLY，而persist可以根据情况设置其它的缓存级别。**

Saprk Streaming从Kafka中读取数据两种方式?

可回答：Spark是怎么对接Kafka的数据流？怎么获取Kafka的数据？

公众号：旧时光大数据

问过的一些公司：美团，腾讯，京东 x 2，知乎，百度，字节，阿里云

参考答案：

1、Recive

Receive是使用的高级API，需要消费者连接Zookeeper来读取数据。是由Zookeeper来维护偏移量，不用我们来手动维护，这样的话就比较简单一些，减少了代码量。但是天下没有免费的午餐，它也有很多缺点：

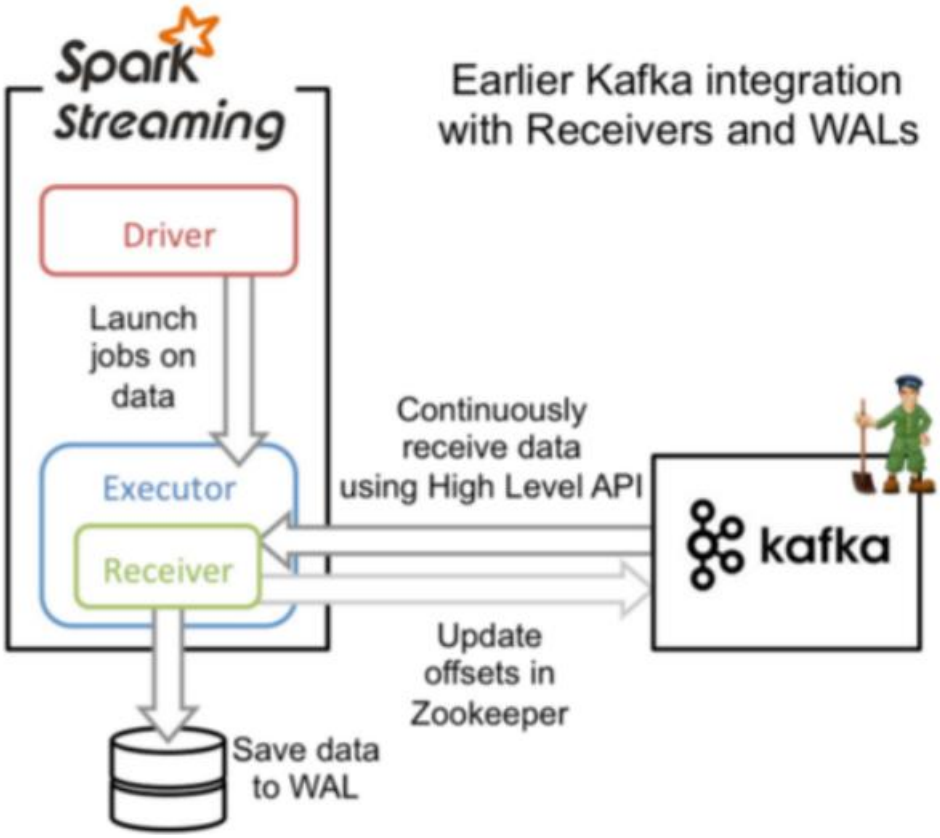
1）导致丢失数据。它是由Executor内的Receive来拉取数据并存放在内存中，再由Driver端提交的job来处理数据。这样的话，如果底层节点出现错误，就会发生数据丢失。

2）浪费资源。可以采取WALs方式将数据同步到高可用数据存储平台上（HDFS，S3），那么如果再发生错误，就可以从中再次读取数据。但是这样会导致同样的数据存储了两份，浪费了资源。

3）可能会导致重复读取数据。对于公司来说，一些数据宁可丢失了一小小部分也不能重复读取，但是这种由Zookeeper来记录偏移量的方式，可能会因为Spark和Zookeeper不同步，导致一份数据读取了两次。

4）效率低。因为是分批次执行的，它是接收数据，直到达到了设定的时间间隔，才可是进行计算。而且我们在KafkaUtils.createStream()中设定的partition数量，只会增加receive的数量，不能提高并行计算的效率，但我们可以设定不同的Group和Topic创建DStream，然后再用Union合并DStream，提高并行效率。

官网Receive的架构图如下：



2、Direct

Direct方式则采用的是低层次的API，直接连接kafka服务器上读取数据。需要我们自己去手动维护偏移量，代码量稍微大些。不过这种方式的优点有：

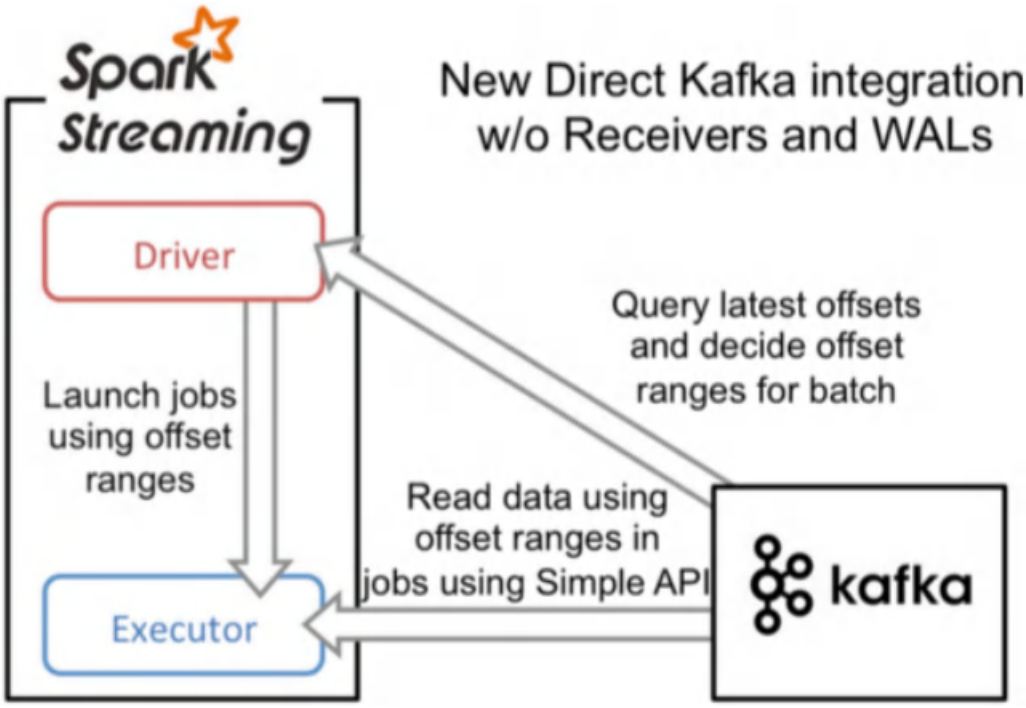
1）当我们读取Topic下的数据时，它会自动对应Topic下的Partition生成相对应数量的RDD Partition，提高了计算时的并行度，提高了效率。

公众号：旧时光大数据

2）它不需要通过WAL来维持数据的完整性。采取Direct直连方式时，当数据发生丢失，只要kafka上的数据进行了复制，就可以根据副本来进行数据重新拉取。

3）它保证了数据只消费一次。因为我们将偏移量保存在一个地方，当我们读取数据时，从这里拿到数据的起始偏移量和读取偏移量确定读取范围，通过这些我们可以读取数据，当读取完成后会更新偏移量，这就保证了数据只消费一次。

官方的Direct架构图如下：



3、总结

在spark1.3以后的版本中，direct方式取代了receive方式，当然在公司中，使用的都是direct方式。从上面对比也能看出receive方式的效率低，而且数据完整性也很让人担忧，当我们采取direct方式时，完全不用为这两点所担忧，可以根据自己想读取的范围进行读取，底层失败后也能通过副本来进行数据恢复。

Spark Streaming的工作原理?

可回答：1）Spark Streaming的（底层）实现原理；2）Spark Streaming的工作机制

问过的一些公司：爱奇艺，哈啰，阿里，作业帮，快手，腾讯

参考答案：

1、Spark Streaming介绍

Spark Streaming是一个流式数据处理（Stream Processing）的框架，**要处理的数据就像流水一样源源不断的产生，就需要实时处理。**

在Spark Streaming中，对于Spark Core进行了API的封装和扩展，**将流式的数据切分为小批次（batch，称之为微批，按照时间间隔切分）进行处理，可以用于进行大规模、高吞吐量、容错的实时数据流的处理。同Storm相比：Storm是来一条数据处理一条数据，是真正意义上的实时处理**

支持从很多种数据源中读取数据，使用算子来进行数据处理，处理后的数据可以被保存到文件系统、数据库等存储中。

相关概念：

公众号：旧时光大数据

DStream：离散流，相当于是一个数据的集合

StreamingContext：在创建StreamingContext的时候，会自动的创建SparkContext对象

对于电商来说，每时每刻都会产生数据（如订单，网页的浏览数据等），这些数据就需要实时的数据处理，将源源不断产生的数据实时收集并实时计算，尽可能快的得到计算结果并展示。

2、Spark Streaming组成部分

1）数据源

大多情况从Kafka中获取数据，还可以从Flume中直接获取，还能从TCP Socket中获取数据（一般用于开发测试）

2）数据处理

主要通过DStream针对不同的业务需求使用不同的方法（算子）对数据进行相关操作。

企业中最多的两种类型统计：实时累加统计（如统计某电商销售额）会用到DStream中的算子updateStateBykey、实时统计某段时间内的数据（如对趋势进行统计分析，实时查看最近20分钟内各个省份用户点击广告的流量统计）会用到reduceByKeyAndWindow这个算子。

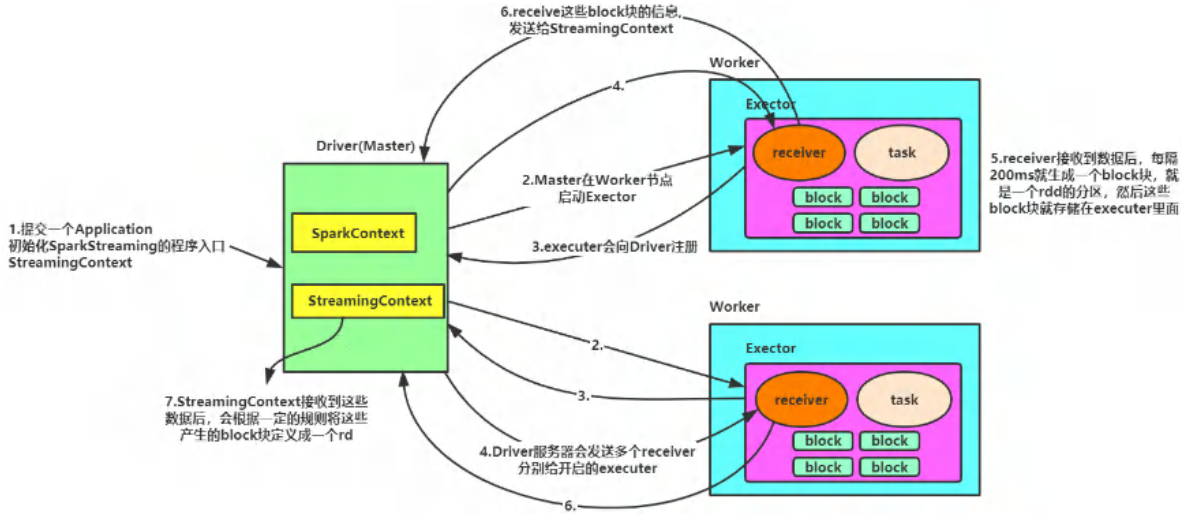
3）存储结果

调用RDD中的API将数据进行存储，因为Spark Streaming是将数据分为微批处理的，所以每一批次就相当于一个RDD。

可以把结果存储到Console（控制台打印，开发测试）、Redis（基于内存的分布式Key-Value数据库）、HBase（分布式列式数据库）、RDBMS（关系型数据库，如MySQL，通过JDBC）

3、SparkStreaming的运行流程

1）运行流程图



2）运行流程

（1）我们在集群中的其中一台机器上提交我们的Application Jar，然后就会产生一个Application，开启一个Driver，然后初始化SparkStreaming的程序入口StreamingContext；

（2）Master（Driver是spark作业的Master）会为这个Application的运行分配资源，在集群中的一台或者多台Worker上面开启Executer，executer会向Driver注册；

公众号：旧时光大数据

（3）Driver服务器会发送多个receiver给开启的executer，（receiver是一个接收器，是用来接收消息的，在excuter里面运行的时候，其实就相当于一个task任务）。每个作业包含多个Executor，每个Executor以线程的方式运行task，Spark Streaming至少包含一个receiver task；

（4）receiver接收到数据后，每隔200ms就生成一个block块，就是一个rdd的分区，然后这些block块就存储在executer里面，block块的存储级别是Memory_And_Disk_2；

（5）receiver产生了这些block块后会把这些block块的信息发送给StreamingContext；

（6）StreamingContext接收到这些数据后，会根据一定的规则将这些产生的block块定义成一个rdd；

4、Spark Streaming工作原理

接收实时输入数据流，然后将数据拆分成多个batch，比如每收集1s的数据封装为一个batch，然后将每个batch交给Spark的计算引擎进行处理，最后会生产出一个结果数据流，其中的数据，也是一个个的batch所组成的。

其中，一个batchInterval累加读取到的数据对应一个RDD的数据

BlockInterval：200ms

生成block块的依据，多久内的数据生成一个block块，默认值200ms生成一个block块，官网最小推荐值50ms。

BatchInterval：1s

我们将每秒的数据抽象为一个RDD。那么这个RDD里面包含了多个block（1s则是50个RDD），这些block是分散的存储在各个节点上的。

5、Spark Streaming和Storm对比

对比点	Storm	Spark Streaming
实时计算模型	纯实时，来一条数据，处理一条数据	准实时，对一个时间段内的数据收集起来，作为一个RDD，再处理
实时计算延迟度	毫秒级	秒级
吞吐量	低	高
事务机制	支持完善	支持，但不够完善
健壮性 / 容错性	ZooKeeper，Acker，非常强	Checkpoint，WAL，一般
动态调整并行度	支持	不支持

1）对比优势

公众号：旧时光大数据

Storm在实时延迟度上，比Spark Streaming就好多了，Storm是纯实时，Spark Streaming是准实时；而且Storm的事务机制，健壮性/容错性、动态调整并行度等特性，都要比Spark Streaming更加优秀。

Spark Streaming的真正优势（Storm绝对比不上的），是它属于Spark生态技术栈中，因此Spark Streaming可以和Spark Core、Spark SQL无缝整合，而这也就意味着，我们可以对实时处理出来的中间数据，立即在程序中无缝进行延迟批处理、交互式查询等操作，这个特点大大增强了Spark Streaming的优势和功能。

2）应用场景

Storm应用场景：

- 建议在那种需要纯实时，不能忍受1s以上延迟的场景下使用，比如金融系统，要求纯实时进行金融交易和分析；
- 如果对于实时计算的功能中，要求可靠的事务机制和可靠性机制，即数据的处理完全精准，一条也不能多，一条也不能少，也可以考虑使用Strom；
- 如果需要针对高峰低峰时间段，动态调整实时计算程序的并行度，以最大限度利用集群资源，也可以考虑用Storm；
- 如果一个大数据应用系统，它就是纯粹的实时计算，不需要在中间执行SQL交互式查询、复杂的transformation算子等，那么使用Storm是比较好的选择

Spark Streaming应用场景：

- 如果对上述适用于Storm的三点，一条都不满足的实时场景，即，不要求纯实时，不要求强大可靠的事务机制，不要求动态调整并行度，那么可以考虑使用Spark Streaming；
- 考虑使用Spark Streaming最主要的一个因素，应该是针对整个项目进行宏观的考虑，即，如果一个项目除了实时计算之外，还包括了离线批处理、交互式查询等业务功能，而且实时计算中，可能还会牵扯到高延迟批处理、交互式查询等功能，那么就应该首选Spark生态，用Spark Core开发离线批处理，用Spark SQL开发交互式查询，用Spark Streaming开发实时计算，三者可以无缝整合，给系统提供非常高的可扩展性。

Spark Streaming的DStream和DStreamGraph的区别?

问过的一些公司：深信服

参考答案：

Spark Streaming 是基于Spark Core将流式计算分解成一系列的小批处理任务来执行。

在Spark Streaming里，总体负责任务的动态调度是 `JobScheduler` ，而 `JobScheduler` 有两个很重要的成员： `JobGenerator` 和 `ReceiverTracker` 。 `JobGenerator` 负责将每个 batch 生成具体的 RDD DAG ，而 `ReceiverTracker` 负责数据的来源。

Spark Streaming里的 `DStream` 可以看成是Spark Core里的RDD的模板， `DStreamGraph` 是RDD DAG的模板。

来看下案例：

DStream 也和 RDD 一样有着转换（transformation）和 输出（output）操作，通过 `transformation` 操作会产生新的 `DStream` ，典型的 `transformation` 操作有map(), filter(), reduce(), join()等。RDD的输出操作会触发action，而DStream的输出操作也会新建一个 `ForeachDStream` ，用一个函数func来记录所需要做的操作。

```
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
val lines = ssc.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
wordCounts.print()
ssc.start()
ssc.awaitTermination()

```

在创建 `StreamingContext` 的时候实创建了graph：DStreamGraph：

```
private[streaming] val graph: DStreamGraph = {
    if (isCheckpointPresent) {
      _cp.graph.setContext(this)
      _cp.graph.restoreCheckpointData()
      _cp.graph
    } else {
      require(_batchDur != null, "Batch duration for StreamingContext cannot be null")
      val newGraph = new DStreamGraph()
      newGraph.setBatchDuration(_batchDur)
      newGraph
    }
  }

```

若 `checkpoint` 可用，会优先从 checkpoint 恢复 graph，否则新建一个。graph用来动态的创建RDD DAG， `DStreamGraph` 有两个重要的成员： `inputStreams` 和 `outputStreams` 。

```
private val inputStreams = new ArrayBuffer[InputDStream[_]]()
private val outputStreams = new ArrayBuffer[DStream[_]]()

```

Spark Streaming记录DStream DAG 的方式就是通过 `DStreamGraph` 实例记录所有的 `outputStreams` ，因为 `outputStream` 会通过依赖 `dependencies` 来和parent DStream形成依赖链，通过 `outputStreams` 向前追溯遍历就可以得到所有上游的DStream，另外， `DStreamGraph` 还会记录所有的 `inputStreams` ，避免每次为查找 input stream 而对 output steam 进行 BFS 的消耗。

继续回到例子，这里通过ssc.socketTextStream 创建了一个 `ReceiverInputDStream` ，在其父类 InputDStream 中会将该 `ReceiverInputDStream` 添加到 `inputStream` 里。

接着调用了flatMap方法：

```
def flatMap[U: ClassTag](flatMapFunc: T => TraversableOnce[U]): DStream[U] = ssc.withScope {
    new FlatMappedDStream(this, context.sparkContext.clean(flatMapFunc))
  }

private[streaming]class FlatMappedDStream[T: ClassTag, U: ClassTag](
    parent: DStream[T],    flatMapFunc: T => TraversableOnce[U]
  ) extends DStream[U](parent.ssc) {

  override def dependencies: List[DStream[_]] = List(parent)

  override def slideDuration: Duration = parent.slideDuration

```

公众号：旧时光大数据

```
12
13      override def compute(validTime: Time): Option[RDD[U]] = {
14        parent.getOrCompute(validTime).map(_.flatMap(flatMapFunc))
15      }
16    }
17
```

创建了一个 `FlatMappedDStream` ，而该类的compute方法是在父 DStream（ReceiverInputDStream）在对应batch时间的RDD上调用了flatMap方法，也就是构造了 `rdd.flatMap(func)` 这样的代码，后面的操作类似，随后形成的是 `rdd.flatMap(func1).map(func2).reduceByKey(func3).take()` ，这不就是我们spark core里的东西吗。另外其 `dependencies` 是直接指向了其构造参数parent，也就是刚才的 `ReceiverInputDStream` ，每个新建的DStream的dependencies都是指向了其父DStream，这样就构成了一个依赖链，也就是形成了DStream DAG。

这里我们再看看最后的 print() 操作：

```
1    def print(num: Int): Unit = ssc.withScope {
2        def foreachFunc: (RDD[T], Time) => Unit = {
3          (rdd: RDD[T], time: Time) => {
4            val firstNum = rdd.take(num + 1)        // scalastyle:off println
5            println("-------------------------------------------")
6            println(s"Time: $time")
7            println("-------------------------------------------")
8            firstNum.take(num).foreach(println)        if (firstNum.length > num)
   println("...")
9            println()        // scalastyle:on println
10         }
11       }
12       foreachRDD(context.sparkContext.clean(foreachFunc), displayInnerRDDOps =
   false)
13     }
14    private def foreachRDD(
15         foreachFunc: (RDD[T], Time) => Unit,
16         displayInnerRDDOps: Boolean): Unit = {    new ForEachDStream(this,
17         context.sparkContext.clean(foreachFunc, false),
   displayInnerRDDOps).register()
18     }
19    def generateJob(time: Time): Option[Job] = {    parent.getOrCompute(time) match
   {      case Some(rdd) =>
20           val jobFunc = () => createRDDWithLocalProperties(time,
   displayInnerRDDOps) {
21             foreachFunc(rdd, time)
22           }
23           Some(new Job(time, jobFunc))      case None => None
24       }
25     }
26
```

在print() 方法里构建了一个foreachFunc方法：对一个rdd进行了take操作并打印（spark core中的action操作）。随后创建了ForEachDStream实例并调用了register()方法：

```
1    private[streaming] def register(): DStream[T] = {
2        ssc.graph.addOutputStream(this)
3        this
4    }
5
```

公众号：旧时光大数据

将 OutputStream 添加到 `DStreamGraph` 的 `outputStreams` 里。可以看到刚才构建的 foreachFunc 方法最终用在了 `ForEachDStream` 实例的 `generateJob` 方法里，并创建了一个Streaming 中的Job，在job中的run方法中会调用这个方法，也就是会触发action操作。

注意这里Spark Streaming的Job和Spark Core里的Job是不一样的，Streaming的Job执行的是前面构造的方法，方法里面是Core里的Job，方法可以定义多个core里的Job，也可以一个core里的job都没有。

Spark输出文件的个数，如何合并小文件?

问过的一些公司：美团，北京元安物联社招

参考答案：

当使用spark sql执行etl时候出现了，最终结果大小只有几百k，但是小文件一个分区可能就有上千的情况。

小文件过多的一些危害如下：

- hdfs有最大文件数限制
- 浪费磁盘资源（可能存在空文件）
- hive中进行统计，计算的时候，会产生很多个map，影响计算的速度。

解决方案如下：

方法一：通过spark的coalesce()方法和repartition()方法

```
1    val rdd2 = rdd1.coalesce(8, true)  //（true表示是否shuffle）
2    val rdd3 = rdd1.repartition(8)
3
```

coalesce：coalesce()方法的作用是返回指定一个新的指定分区的Rdd，如果是生成一个窄依赖的结果，那么可以不发生shuffle，分区的数量发生激烈的变化，计算节点不足，不设置true可能会出错。

repartition：coalesce()方法shuffle为true的情况。

方法二：降低spark并行度，即调节spark.sql.shuffle.partitions

比如之前设置的为100，按理说应该生成的文件数为100；但是由于业务比较特殊，采用的大量的union all，且union all在spark中属于窄依赖，不会进行shuffle，所以导致最终会生成（union all数量+1）*100的文件数。如有10个union all，会生成1100个小文件。这样导致降低并行度为10之后，执行时长大大增加，且文件数依旧有110个，效果有，但是不理想。

方法三：新增一个并行度=1任务，专门合并小文件

先将原来的任务数据写到一个临时分区（如tmp）；再起一个并行度为1的任务，类似：

```
1    insert overwrite 目标表 select * from 临时分区
2
```

结果小文件数还是没有减少，经过多次测后发现原因：‘select * from 临时分区’这个任务在spark中属于窄依赖；并且spark DAG中分为宽依赖和窄依赖，只有宽依赖会进行shuffle；故并行度shuffle，spark.sql.shuffle.partitions=1也就没有起到作用；

由于数据量本身不是特别大，所以直接采用了group by（在spark中属于宽依赖）的方式，类似：

```
1    insert overwrite 目标表 select * from 临时分区 group by *
2
```

公众号：旧时光大数据

先运行原任务，写到tmp分区，‘dfs -count’查看文件数，1100个，运行加上group by的临时任务（spark.sql.shuffle.partitions=1），查看结果目录，文件数=1，成功。

总结：

1）方便的话，可以采用coalesce()方法和repartition()方法

2）如果任务逻辑简单，数据量少，可以直接降低并行度

3）任务逻辑复杂，数据量很大，原任务大并行度计算写到临时分区，再加两个任务：

- 一个用来将临时分区的文件用小并行度（加宽依赖）合并成少量文件到实际分区
- 另一个删除临时分区

4）hive任务减少小文件相对比较简单，可以直接设置参数，如：

Map-only的任务结束时合并小文件：

```
1    sethive.merge.mapfiles = true
2
```

在Map-Reduce的任务结束时合并小文件：

```
1   sethive.merge.mapredfiles= true
2
```

当输出文件的平均大小小于1GB时，启动一个独立的map-reduce任务进行文件merge：

```
1   sethive.merge.smallfiles.avgsize=1024000000
2
```

Spark的driver是怎么驱动作业流程的?

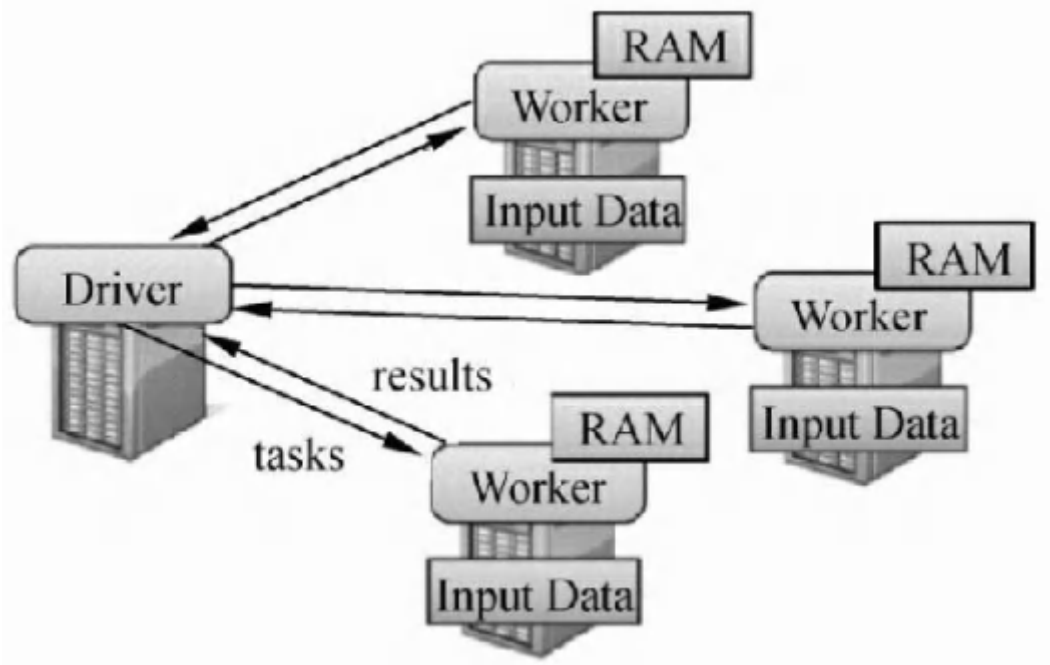
问过的一些公司：平安

参考答案：

SparkContext是通往Spark集群的唯一入口，是整个Application运行调度的核心。

Spark Driver Program（以下简称Driver）是运行Application的main函数并且新建SparkContext实例的程序。其实，初始化SparkContext是为了准备Spark应用程序的运行环境，在Spark中，由SparkContext负责与集群进行通信、资源的申请、任务的分配和监控等。当Worker节点中的Executor运行完毕Task后，Driver同时负责将SparkContext关闭。通常也可以使用SparkContext来代表驱动程序（Driver）。

Driver（SparkContext）整体架构图如下图所示。



Spark SQL的劣势?

问过的一些公司：蘑菇街

参考答案：

1）复杂分析，SQL嵌套较多：试想一下3层嵌套的 SQL维护起来应该挺力不从心的吧

2）机器学习较难：试想一下如果使用SQL来实现机器学习算法也挺为难的吧

介绍下Spark Streaming和Structed Streaming

问过的一些公司：腾讯

参考答案：

Spark在2.*版本后加入Structed Streaming模块，与流处理引擎Spark Streaming一样，用于处理流数据。但二者又有许多不同之处。

Spark Streaming首次引入在0.*版本，其核心思想是利用spark批处理框架，以microbatch（以一段时间的流作为一个batch）的方式，完成对流数据的处理。

Structed Streaming诞生于2.*版本，主要用于处理结构化流数据，除了与Spark Streaming类似的microbatch的处理流数据方式，也实现了long-running的task，可以"不停的"循环从数据源获取数据并处理，从而实现真正的流处理。以dataset为代表的带有结构化（schema信息）的数据处理由于钨丝计划的完成，表现出更优越的性能。同时Structed Streaming可以从数据中获取时间（eventTime），从而可以针对流数据的生产时间而非收到数据的时间进行处理。

Spark Streaming与Structed Streaming对比

公众号：旧时光大数据

流处理模式	SparkStreaming	Structed streaming
执行模式	Micro Batch	Micro batch / Streaming
API	Dstream/streamingContext	Dataset/DataFrame,SparkSession
Job 生成方式	Timer定时器定时生成job	Trigger触发
支持数据源	Socket,filstream,kafka,zeroMq,flume,kinesis	Socket,filstream,kafka,ratesource
executed-based	Executed based on dstream api	Executed based on sparksql
Time based	Processing Time	ProcessingTime & eventTIme
UI	Built-in	No

执行模式

Spark Streaming以micro-batch的模式

- 以固定的时间间隔来划分每次处理的数据，在批次内以采用的是批处理的模式完成计算

Structed streaming有两种模式：

1）Micro-batch模式

- 一种同样以micro-batch的模式完成批处理，处理模式类似sparkStreaming的批处理，可以定期（以固定间隔）处理，也可以处理完一个批次后，立刻进入下一批次的处理

2）Continuous Processing模式

- 一种启动长时运行的线程从数据源获取数据，worker线程长时运行，实时处理消息。放入queue中，启动long-running的worker线程从queue中读取数据并处理。该模式下，当前只能支持简单的projection式（如map,filter,mappartitions等）的操作

API

Spark Streaming :

Sparkstreaming框架基于RDD开发，自实现一套API封装，程序入口是StreamingContext，数据模型是Dstream，数据的转换操作通过Dstream的api完成，真正的实现依然是通过调用rdd的api完成。

Structed Streaming：

Structed Streaming 基于sql开发，入口是sparksession，使用的统一Dataset数据集，数据的操作会使用sql自带的优化策略实现。

Job生成与调度

Spark Streaming Job生成：

```
1    // job通过定时器根据batch duration定期生成Streaming的
     Job（org.apache.spark.streaming.scheduler.Job）该Job是对Spark core的job的封装，在run
     方法中会完成对sparkcontext.runJob的调用
2    // 通过timer定时器定期发生generatorJobs的消息
3    timer = new RecurringTimer(clock, ssc.graph.batchDuration.milliseconds,
4    longTime => eventLoop.post(GenerateJobs(new Time(longTime))), "JobGenerator")//
     调用eventLoop的processEvent方法处理根据消息类型处理消息
5        private def processEvent(event: JobGeneratorEvent) {
6        logDebug("Got event " + event)
7        event match {      case GenerateJobs(time) => generateJobs(time)      case
     ClearMetadata(time) => clearMetadata(time)      case DoCheckpoint(time,
     clearCheckpointDataLater) =>
8            doCheckpoint(time, clearCheckpointDataLater)      case
     ClearCheckpointData(time) => clearCheckpointData(time)
9        }
10   }
11
```

Spark Streaming的Job的调度：

```
1    //生成Job之后，通过调用JobScheduler的submitJobSet方法提交job，submitJobSet通过
     jobExecutor完成调用job的run方法，完成spark Core job的调用
2    def submitJobSet(jobSet: JobSet) {
3        if (jobSet.jobs.isEmpty) {
4          logInfo("No jobs added for time " + jobSet.time)
5        } else {
6          listenerBus.post(StreamingListenerBatchSubmitted(jobSet.toBatchInfo))
7          jobSets.put(jobSet.time, jobSet)
8          jobSet.jobs.foreach(job => jobExecutor.execute(new JobHandler(job)))
9          logInfo("Added jobs for time " + jobSet.time)
10       }
11   }
12       //其中JobExecutor是一个有numCorrentJob（由spark.streaming.concurrentJobs决定，
     默认为1）个线程的线程池，定义如下：
13       jobExecutor = ThreadUtils.newDaemonFixedThreadPool(numConcurrentJobs,
     "streaming-job-executor")
14
```

Structed Streaming job生成与调度：

Structed Streaming通过trigger完成对job的生成和调度。

支持数据源

SparkStreaming出现较早，支持的数据源较为丰富：Socket,filstream,kafka,zeroMq,flume,kinesis

Structed Streaming 支持的数据源：Socket,filstream,kafka,ratesource

executed-based

SparkStreaming在执行过程中调用的是dstream api，其核心执行仍然调用rdd的接口

Structed Streaming底层的数据结构为dataset/dataframe，在执行过程中可以使用的钨丝计划，自动代码生成等优化策略，提升性能

Time Based

Sparkstreaming的处理逻辑是根据应用运行时的时间（Processing Time）进行处理，不能根据消息中自带的时间戳完成一些特定的处理逻辑

公众号：旧时光大数据

Structed streaming，加入了event Time，weatermark等概念，在处理消息时，可以考虑消息本身的时间属性。同时，也支持基于运行时时间的处理方式。

UI

Sparkstreaming 提供了内置的界面化的UI操作，便于观察应用运行，批次处理时间，消息速率，是否延迟等信息。

Structed streaming 暂时没有直观的UI

Spark为什么比Hadoop速度快?

问过的一些公司：腾讯，小米

参考答案：

Spark SQL比Hadoop Hive快，是有一定条件的，而且不是Spark SQL的引擎比Hive的引擎快，相反，Hive的HQL引擎还比Spark SQL的引擎更快。

其实，关键还是在于Spark本身快。

1）Spark是基于内存的计算，而Hadoop是基于磁盘的计算；Spark是一种内存计算技术。

所谓的内存计算技术也就是缓存技术，把数据放到缓存中，减少cpu磁盘消耗。Spark和Hadoop的根本差异是多个任务之间的数据通信问题：Spark多个任务之间数据通信是基于内存，而Hadoop是基于磁盘。Hadoop每次shuffle操作后，必须写到磁盘，而Spark在shuffle后不一定落盘，可以cache到内存中，以便迭代时使用。如果操作复杂，很多的shufle操作，那么Hadoop的读写IO时间会大大增加。多个任务之间的操作也就是shuffle过程，因为要把不同task的相同信息集合到一起，这样内存的速度要明显大于磁盘了。

2）JVM的优化

Hadoop每次MapReduce操作，启动一个Task便会启动一次JVM，基于进程的操作。而Spark每次MapReduce操作是基于线程的，只在启动Executor是启动一次JVM，内存的Task操作是在线程复用的。

每次启动JVM的时间可能就需要几秒甚至十几秒，那么当Task多了，这个时间Hadoop不知道比Spark慢了多少。

考虑一种极端查询：Select month_id,sum(sales) from T group by month_id;

这个查询只有一次shuffle操作，此时，也许Hive HQL的运行时间也许比Spark还快。

总之，Spark快不是绝对的，但是绝大多数，Spark都比Hadoop计算要快。这主要得益于其对mapreduce操作的优化以及对JVM使用的优化。

Spark比MapReduce更快的各种原因：

1. 在处理过程中，Spark使用RAM存储中间数据，而MapReduce使用磁盘存储中间数据。
2. Spark非常有效地使用底层硬件缓存。除了RAM外，Spark还能有效地使用L1、L2和L3缓存，这些缓存比RAM更接近CPU。
3. Spark使用内部Catalyst Optimizer来优化查询物理和逻辑计划。
4. Spark使用Predicate Pushdown。
5. Spark拥有自己的垃圾收集机制，而不是在内置的JAVA垃圾收集器中使用。
6. Spark使用其自己的字节码生成器，而不是内置的JAVA生成器。最终的字节码始终在JRE环境中执行。
7. Spark使用RDD，Transformation和Action进行数据转换，而MapReduce不使用。
8. Spark有自己的专用序列化器和反序列化器，使其更高效。

公众号：旧时光大数据

DAG划分Spark源码实现?

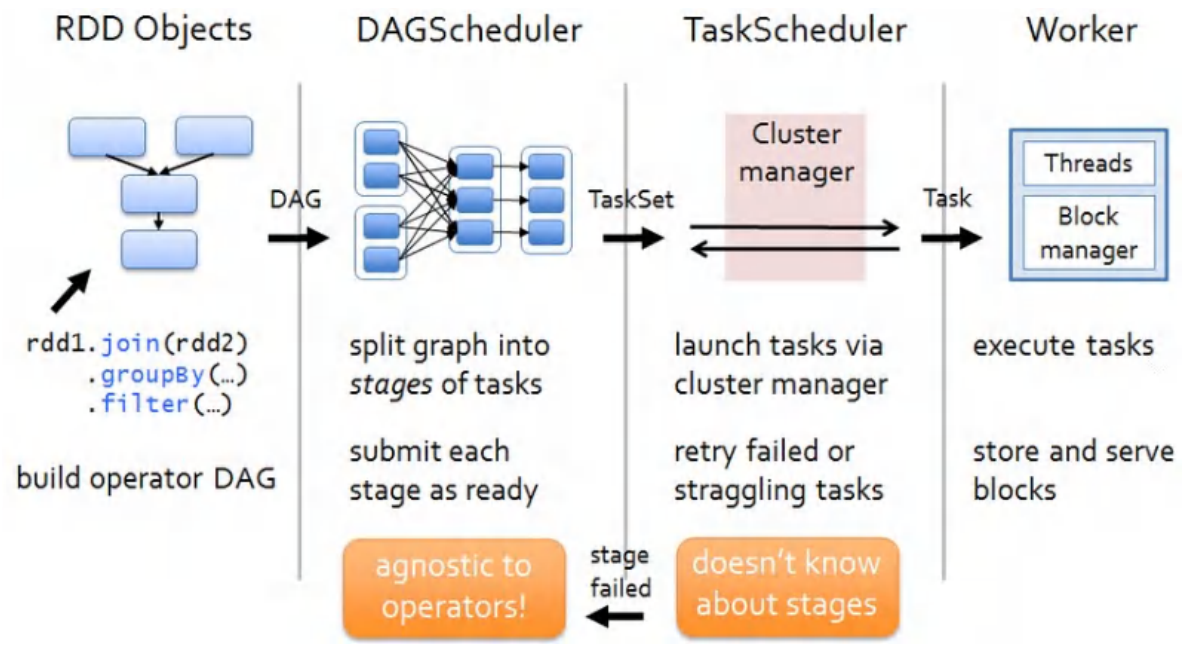
问过的一些公司：腾讯

参考答案：

1、Spark运行架构

Spark 运行架构如下图：

各个RDD之间存在着依赖关系，这些依赖关系形成有向无环图DAG，DAGScheduler对这些依赖关系形成的DAG，进行Stage划分，划分的规则很简单，从后往前回溯，遇到窄依赖加入本stage，遇见宽依赖进行Stage切分。完成了Stage的划分，DAGScheduler基于每个Stage生成TaskSet,并将TaskSet提交给TaskScheduler。TaskScheduler 负责具体的task调度，在Worker节点上启动task。



2、源码解析：DAGScheduler中的DAG划分

当RDD触发一个Action操作（如：colllect）后，导致SparkContext.runJob的执行。而在SparkContext的run方法中会调用DAGScheduler的run方法最终调用了DAGScheduler的submit方法：

```
  def submitJob[T, U](
      rdd: RDD[T],
      func: (TaskContext, Iterator[T]) => U,
      partitions: Seq[Int],
      callSite: CallSite,
      resultHandler: (Int, U) => Unit,
      properties: Properties): JobWaiter[U] = {
    // Check to make sure we are not launching a task on a partition that does not exist.
    val maxPartitions = rdd.partitions.length
    partitions.find(p => p >= maxPartitions || p < 0).foreach { p =>
      throw new IllegalArgumentException(
        "Attempting to access a non-existent partition: " + p + ". " +
          "Total number of partitions: " + maxPartitions)
    }
    val jobId = nextJobId.getAndIncrement()
    if (partitions.size == 0) {
      // Return immediately if the job is running 0 tasks
      return new JobWaiter[U](this, jobId, 0, resultHandler)
    }
    assert(partitions.size > 0)
```

公众号：旧时光大数据

```
21        val func2 = func.asInstanceOf[(TaskContext, Iterator[_]) => _]
22        val waiter = new JobWaiter(this, jobId, partitions.size, resultHandler)
23        //给eventProcessLoop发送JobSubmitted消息
24        eventProcessLoop.post(JobSubmitted(
25          jobId, rdd, func2, partitions.toArray, callSite, waiter,
26          SerializationUtils.clone(properties)))
27        waiter
28      }
29
```

DAGScheduler的submit方法中，像eventProcessLoop对象发送了JobSubmitted消息。eventProcessLoop是DAGSchedulerEventProcessLoop 类的对象

```
1    private[scheduler] val eventProcessLoop = new DAGSchedulerEventProcessLoop(this)
2
```

DAGSchedulerEventProcessLoop，接收各种消息并进行处理，处理的逻辑在其doOnReceive方法中：

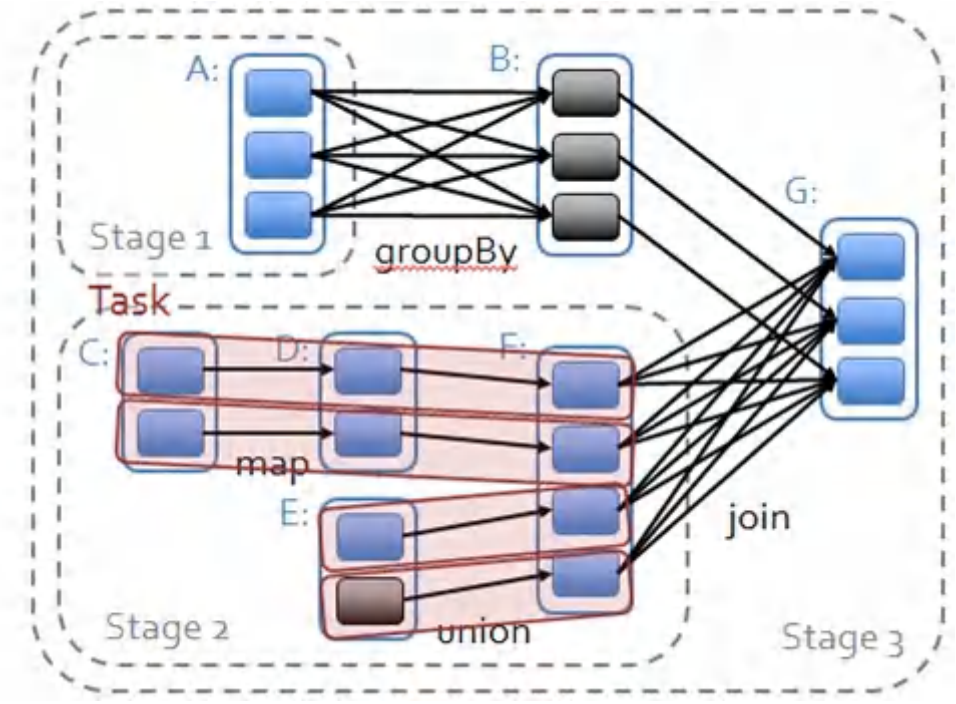
```
1     private def doOnReceive(event: DAGSchedulerEvent): Unit = event match {
2        //Job提交
3        case JobSubmitted(jobId, rdd, func, partitions, callSite, listener,
    properties) =>
4          dagScheduler.handleJobSubmitted(jobId, rdd, func, partitions, callSite,
    listener, properties)
5        case MapStageSubmitted(jobId, dependency, callSite, listener, properties)
    =>
6          dagScheduler.handleMapStageSubmitted(jobId, dependency, callSite,
    listener, properties)
7        case StageCancelled(stageId) =>
8          dagScheduler.handleStageCancellation(stageId)
9        case JobCancelled(jobId) =>
10         dagScheduler.handleJobCancellation(jobId)
11       case JobGroupCancelled(groupId) =>
12         dagScheduler.handleJobGroupCancelled(groupId)
13       case AllJobsCancelled =>
14         dagScheduler.doCancelAllJobs()
15       case ExecutorAdded(execId, host) =>
16         dagScheduler.handleExecutorAdded(execId, host)
17       case ExecutorLost(execId) =>
18         dagScheduler.handleExecutorLost(execId, fetchFailed = false)
19       case BeginEvent(task, taskInfo) =>
20         dagScheduler.handleBeginEvent(task, taskInfo)
21       case GettingResultEvent(taskInfo) =>
22         dagScheduler.handleGetTaskResult(taskInfo)
23       case completion: CompletionEvent =>
24         dagScheduler.handleTaskCompletion(completion)
25       case TaskSetFailed(taskSet, reason, exception) =>
26         dagScheduler.handleTaskSetFailed(taskSet, reason, exception)
27       case ResubmitFailedStages =>
28         dagScheduler.resubmitFailedStages()
29     }
30
```

可以把 DAGSchedulerEventProcessLoop 理解成DAGScheduler的对外的功能接口。它对外隐藏了自己内部实现的细节。无论是内部还是外部消息，DAGScheduler可以共用同一消息处理代码，逻辑清晰，处理方式统一。

接下来分析 DAGScheduler的Stage划分，handleJobSubmitted 方法首先创建ResultStage

```
    try {
          //创建新stage可能出现异常，比如job运行依赖hdfs文文件被删除
          finalStage = newResultStage(finalRDD, func, partitions, jobId, callSite)
        } catch {
          case e: Exception =>
            logWarning("Creating new stage failed due to exception - job: " +
    jobId, e)
            listener.jobFailed(e)
            return
        }

```

然后调用submitStage方法，进行stage的划分。



首先由finalRDD获取它的父RDD依赖，判断依赖类型，如果是窄依赖，则将父RDD压入栈中，如果是宽依赖，则作为父Stage。

看一下源码的具体过程：

```
  private def getMissingParentStages(stage: Stage): List[Stage] = {
      val missing = new HashSet[Stage] //存储需要返回的父Stage
      val visited = new HashSet[RDD[_]] //存储访问过的RDD
      //自己建立栈，以免函数的递归调用导致
      val waitingForVisit = new Stack[RDD[_]]
      def visit(rdd: RDD[_]) {
        if (!visited(rdd)) {
          visited += rdd
          val rddHasUncachedPartitions = getCacheLocs(rdd).contains(Nil)
          if (rddHasUncachedPartitions) {
            for (dep <- rdd.dependencies) {
              dep match {
                case shufDep: ShuffleDependency[_, _, _] =>
                  val mapStage = getShuffleMapStage(shufDep, stage.firstJobId)
                  if (!mapStage.isAvailable) {
                    missing += mapStage //遇到宽依赖，加入父stage
                  }
                case narrowDep: NarrowDependency[_] =>
                  waitingForVisit.push(narrowDep.rdd) //窄依赖入栈，
```

```
            }
          }
        }
      }
    //回溯的起始RDD入栈
    waitingForVisit.push(stage.rdd)
     while (waitingForVisit.nonEmpty) {
       visit(waitingForVisit.pop())
     }
     missing.toList
   }

```

getMissingParentStages方法是由当前stage，返回他的父stage，父stage的创建由getShuffleMapStage返回，最终会调用 newOrUsedShuffleStage 方法返回ShuffleMapStage

```
 private def newOrUsedShuffleStage(
       shuffleDep: ShuffleDependency[_, _, _],
       firstJobId: Int): ShuffleMapStage = {
     val rdd = shuffleDep.rdd
     val numTasks = rdd.partitions.length
     val stage = newShuffleMapStage(rdd, numTasks, shuffleDep, firstJobId,
rdd.creationSite)
     if (mapOutputTracker.containsShuffle(shuffleDep.shuffleId)) {
      //Stage已经被计算过，从MapOutputTracker中获取计算结果
       val serLocs =
mapOutputTracker.getSerializedMapOutputStatuses(shuffleDep.shuffleId)
       val locs = MapOutputTracker.deserializeMapStatuses(serLocs)
       (0 until locs.length).foreach { i =>
         if (locs(i) ne null) {
           // locs(i) will be null if missing
           stage.addOutputLoc(i, locs(i))
         }
       }
     } else {
       // Kind of ugly: need to register RDDs with the cache and map output
tracker here
       // since we can't do it in the RDD constructor because # of partitions is
unknown
       logInfo("Registering RDD " + rdd.id + " (" + rdd.getCreationSite + ")")
       mapOutputTracker.registerShuffle(shuffleDep.shuffleId,
rdd.partitions.length)
     }
     stage
   }

```

现在父Stage已经划分好，下面看看Stage的提交逻辑

```
  /** Submits stage, but first recursively submits any missing parents. */
   private def submitStage(stage: Stage) {
     val jobId = activeJobForStage(stage)
     if (jobId.isDefined) {
       logDebug("submitStage(" + stage + ")")
       if (!waitingStages(stage) && !runningStages(stage) &&
!failedStages(stage)) {
         val missing = getMissingParentStages(stage).sortBy(_.id)
```

公众号：旧时光大数据

```
8             logDebug("missing: " + missing)
9             if (missing.isEmpty) {
10              logInfo("Submitting " + stage + " (" + stage.rdd + "), which has no
    missing parents")
11              //如果没有父stage，则提交当前stage
12              submitMissingTasks(stage, jobId.get)
13            } else {
14              for (parent <- missing) {
15                //如果有父stage，则递归提交父stage
16                submitStage(parent)
17              }
18              waitingStages += stage
19            }
20          }
21        } else {
22          abortStage(stage, "No active job for stage " + stage.id, None)
23        }
24      }
25
```

提交的过程很简单，首先当前stage获取父stage，如果父stage为空，则当前Stage为起始stage，交给submitMissingTasks处理，如果当前stage不为空，则递归调用submitStage进行提交。

Spark Streaming的双流join的过程，怎么做的?

问过的一些公司：快手

参考答案：

在构建实时数仓过程中，有时需要将两个实时数据源进行关联，生成大宽表数据，这时就不得不用到双流join。

1、场景介绍

比如有这样的场景，订单实时数据源，和订单物品实时数据源。订单数据源有订单id，下单时间，订单金额，收货人，收货地址等信息，订单商品数据源有订。单号id，商品名称，商品品牌，商品价格，商品成交价格，商品所属商家。订单和订单商品一般是一对多的关系。生成订单大宽表数据，需要将订单的每一件商品信息都关联上订单详细信息。

2、可能面临问题

我们使用流数据join，不像静态数据那么简单。我们知道静态数据join，两份数据都是确定的，没关联上就是没关联上，关联上了就是关联上了。此刻关联上的，将来再运行，也必然是关联上的，此刻没关联上的，将来运行，也是关联不上的。而在流数据join时，由于数据是动态的，两个原本能关联的数据进入程序的时机未必一致，此时就有可能造成关联数据失去关联，当然没关联上的也有可能是真的关联不上(join的一方数据缺漏)。

3、解决方案

由于流数据join的特殊性，我们须采取必要的处理方案，使得两份join的流数据尽量都能够关联上，真正完成流数据join的使命。

订单数据和订单商品数据是一对多的关系，也就是一条订单数据对应着一条或者多条订单商品数据，而一条订单商品数据只能对应着一条订单数据。所以在join的时候，订单数据无论是否关联上订单商品数据，都应当保存下来，留作下次继续join使用。而一条订单商品数据，一旦关联上订单数据，即可被废弃，否则就应当将失配的订单商品数据保存下来，直至下次关联成功。

公众号：旧时光大数据

4、详细说明

为方便说明，假定订单数据A, B, 订单商品数据a1,a2, a3. A可关联a1, a2, a3. B则没有订单商品数据与之关联。下面讨论某一时刻join的各种情况及其应对办法。

1）(A, None), 该情况说明某一时刻，订单数据A到了，能与之关联的a系列数据一个也没到，此时有2种情况，

- a系列数据从未进入过程序, 对于这种情况，我们将A数据保存到redis中，key的结构为 order_info:<order_id>. 使得下次a系列的订单商品数据来了之后，有机会进行关联；
- 还有一种情况是a系列数据先到了，已经保存到了redis中，如情况3的第 2) 种情况所述, 去redis中将a系列数据查出(可能存在多个订单商品数据), 完成匹配，然后将匹配完的a系列数据，从redis中删除，以免下次join又去redis查询，重复匹配。接着，还需将A数据保存到redis中，因为将来还有可能有a系列的数据需要匹配。

2）(A, a1), 该情况说明某一时刻，订单数据A到了，能与之关联的a1订单商品数据也到了，正好进行join。此时需要注意，由于订单数据和订单商品数据是一对多的关系，后续是否还存在能关联A的a系列订单商品数据，我们并不知道，极有可能是存在的。因此，尽管已经形成了关联，还需要将A保存到redis中，如1所述，而a1已经完成匹配，不可能再有匹配的机会，无需保存. 另外，还需要注意的是，有可能此时redis中还存有a系列的商品订单数据，因此还需要像情况1的情形 2) 那样，去redis中寻找a系列的商品订单数据，找到后进行关联，关联完要将redis中的a系列商品订单数据删除，避免下次重复匹配。我们注意到，情况2和情况1的过程几乎差不多，就一个进入程序的A和a1不用查redis就可关联之外，其他过程几乎一样，因此在程序中，可以将这2种情况合并

3）(None, a2)，该情况说明，订单商品数据a2到了，能与之关联的A订单数据却不在。此时我们应当这样处理，先去redis查找A订单数据.

- 如果找到，则可完成匹配，不用保存a2数据；
- 如果没有找到A数据，说明A数据还没进入过程序，a2数据应当保存到redis中, key结构为 order_product:<order_id>:<produce_id>, 这样a2数据可以在redis中等待下一次A数据的到来，然后完成匹配

4）(B, None) 其实是和情况1相同，不过实际数据中，不存在能与之关联的订单商品数据，但在程序中我们并不知道这一情况，我们一律将B存到redis中，订单流数据是源源不断的进来，会造成redis数据不断膨胀，因此我们不能订单数据永久地存于redis中，订单和订单商品是强关联的，关联匹配的两方数据不可能会相隔太久进入程序。因此我们在redis中设置订单数据的存活时间

需要说明的是，对于没有匹配的订单商品数据，保存到redis中，一旦下次完成匹配，须立刻将其删除，以免下次join又将其查询出来重复匹配。还有，如果订单商品数据一直没有匹配的订单数据，过一段时间也应该将其删除，因此同样地，需要在redis中设置订单商品的存活时间。

Spark的Block管理

问过的一些公司：快手社招

参考答案：

Apache Spark中，对Block的查询、存储管理，是通过唯一的Block ID来进行区分的。所以，了解Block ID的生成规则，能够帮助我们了解Block查询、存储过程中是如何定位Block以及如何处理互斥存储/读取同一个Block的。

可以想到，同一个Spark Application，以及多个运行的Application之间，对应的Block都具有唯一的ID，通过代码可以看到，BlockID包括：RDDBlockId、ShuffleBlockId、ShuffleDataBlockId、ShuffleIndexBlockId、BroadcastBlockId、TaskResultBlockId、TempLocalBlockId、TempShuffleBlockId这8种ID，可以详见如下代码定义：

公众号：旧时光大数据

```
@DeveloperApi
case class RDDBlockId(rddId: Int, splitIndex: Int) extends BlockId {
  override def name: String = "rdd_" + rddId + "_" + splitIndex
}

// Format of the shuffle block ids (including data and index) should be kept in sync with
//
org.apache.spark.network.shuffle.ExternalShuffleBlockResolver#getBlockData().
@DeveloperApi
case class ShuffleBlockId(shuffleId: Int, mapId: Int, reduceId: Int) extends BlockId {
  override def name: String = "shuffle_" + shuffleId + "_" + mapId + "_" + reduceId
}

@DeveloperApi
case class ShuffleDataBlockId(shuffleId: Int, mapId: Int, reduceId: Int) extends BlockId {
  override def name: String = "shuffle_" + shuffleId + "_" + mapId + "_" + reduceId + ".data"
}

@DeveloperApi
case class ShuffleIndexBlockId(shuffleId: Int, mapId: Int, reduceId: Int) extends BlockId {
  override def name: String = "shuffle_" + shuffleId + "_" + mapId + "_" + reduceId + ".index"
}

@DeveloperApi
case class BroadcastBlockId(broadcastId: Long, field: String = "") extends BlockId {
  override def name: String = "broadcast_" + broadcastId + (if (field == "") "" else "_" + field)
}

@DeveloperApi
case class TaskResultBlockId(taskId: Long) extends BlockId {
  override def name: String = "taskresult_" + taskId
}

@DeveloperApi
case class StreamBlockId(streamId: Int, uniqueId: Long) extends BlockId {
  override def name: String = "input-" + streamId + "-" + uniqueId
}

/** Id associated with temporary local data managed as blocks. Not serializable. */
private[spark] case class TempLocalBlockId(id: UUID) extends BlockId {
  override def name: String = "temp_local_" + id
}

/** Id associated with temporary shuffle data managed as blocks. Not serializable. */
private[spark] case class TempShuffleBlockId(id: UUID) extends BlockId {
  override def name: String = "temp_shuffle_" + id
}
```

公众号：旧时光大数据

```
47
```

我们以RDDBlockId的生成规则为例，它是以前缀字符串“rdd ”为前缀、分配的全局RDD ID、下划线“ ”、Partition ID这4部分拼接而成，因为RDD ID是唯一的，所以最终构造好的RDDBlockId对应的字符串就是唯一的。如果该Block存在，查询可以唯一定位到该Block，存储也不会出现覆盖其他RDDBlockId的问题。

下面通过分析MemoryStore、DiskStore、BlockManager、BlockInfoManager这4个最核心的与Block管理相关的实现类，来理解Spark对Block的管理。主要针对RDDBlockId对应的Block数据的处理、存储、查询、读取，来分析Block的管理。

1、MemoryStore

先说明一下MemoryStore，它主要用来在内存中存储Block数据，可以避免重复计算同一个RDD的Partition数据。一个Block对应着一个RDD的一个Partition的数据。当StorageLevel设置为如下值时，都会可能会需要使用MemoryStore来存储数据：

```
1   MEMORY_ONLY
2   MEMORY_ONLY_2
3   MEMORY_ONLY_SER
4   MEMORY_ONLY_SER_2
5   MEMORY_AND_DISK
6   MEMORY_AND_DISK_2
7   MEMORY_AND_DISK_SER
8   MEMORY_AND_DISK_SER_2
9   OFF_HEAP
10
```

所以，MemoryStore提供对Block数据的存储、读取等操作API，MemoryStore也提供了多种存储方式，下面详细说明每种方式。

1）以序列化格式保存Block数据

```
1   def putBytes[T: ClassTag](
2       blockId: BlockId,
3       size: Long,
4       memoryMode: MemoryMode,
5       _bytes: () => ChunkedByteBuffer): Boolean
6
```

首先，通过MemoryManager来申请Storage内存，调用putBytes方法，会根据size大小去申请Storage内存，如果申请成功，则会将blockId对应的Block数据保存在内部的LinkedHashMap[BlockId, MemoryEntry[_]]映射表中，然后以SerializedMemoryEntry这种序列化的格式存储，实际SerializedMemoryEntry就是简单指向Buffer中数据的引用对象：

```
1   private case class SerializedMemoryEntry[T](
2       buffer: ChunkedByteBuffer,
3       memoryMode: MemoryMode,
4       classTag: ClassTag[T]) extends MemoryEntry[T] {
5     def size: Long = buffer.size
6   }
7
```

如果无法申请到size大小的Storage内存，则存储失败，对于出现这种失败的情况，需要使用MemoryStore存储API的调用者去处理异常情况。

2）基于记录迭代器，以反序列化Java对象形式保存Block数据

```
  private[storage] def putIteratorAsValues[T](
      blockId: BlockId,
      values: Iterator[T],
      classTag: ClassTag[T]): Either[PartiallyUnrolledIterator[T], Long]

```

这种方式，调用者希望将Block数据记录以反序列化的方式保存在内存中，如果内存中能放得下，则返回最终Block数据记录的大小，否则返回一个PartiallyUnrolledIterator[T]迭代器，其中对应如下2种情况：

第一种，Block数据记录能够完全放到内存中：和前面的方式类似，能够全部放到内存，但是不同的是，这种方式对应的数据格式是反序列化的Java对象格式，对应实现类DeserializedMemoryEntry[T]，它也会被直接存放到MemoryStore内部的LinkedHashMap[BlockId, MemoryEntry[_]]映射表中。DeserializedMemoryEntry[T]类定义如下所示：

```
   private case class DeserializedMemoryEntry[T](
       value: Array[T],
       size: Long,
       classTag: ClassTag[T]) extends MemoryEntry[T] {
     val memoryMode: MemoryMode = MemoryMode.ON_HEAP
   }

```

它与SerializedMemoryEntry都是MemoryEntry[T]的子类，所有被放到同一个映射表LinkedHashMap[BlockId, MemoryEntry[_]] entries中。

另外，也存在这种可能，通过MemoryManager申请的Unroll内存大小大于该Block打开需要的内存，则会返回如下结果对象：

```
   Left(new PartiallyUnrolledIterator( this,  unrollMemoryUsedByThisBlock,
     unrolled = arrayValues.toIterator,  rest = Iterator.empty))

```

上面unrolled = arrayValues.toIterator，rest = Iterator.empty，表示在内存中可以打开迭代器中全部的数据记录，打开对象类型为DeserializedMemoryEntry[T]。

第二种，Block数据记录只能部分放到内存中：也就是说Driver或Executor上的内存有限，只可以放得下部分记录，另一部分记录内存中放不下。values记录迭代器对应的全部记录数据无法完全放在内存中，所以为了保证不发生OOM异常，首选会调用MemoryManager的acquireUnrollMemory方法去申请Unroll内存，如果可以申请到，在迭代values的过程中，需要累加计算打开（Unroll）的记录对象大小之和，使其大小不能大于申请到的Unroll内存，直到还有一部分记录无法放到申请的Unroll内存中。 最后，返回的结果对象如下所示：

```
   Left(new PartiallyUnrolledIterator(
     this, unrollMemoryUsedByThisBlock, unrolled = vector.iterator, rest = values))

```

上面的PartiallyUnrolledIterator中rest对应的values就是putIteratorAsValues方法传进来的迭代器参数值，该迭代器已经迭代出部分记录，放到了内存中，调用者可以继续迭代该迭代器去处理未打开（Unroll）的记录，而unrolled对应一个打开记录的迭代器。这里，PartiallyUnrolledIterator迭代器包装了vector.iterator和一个迭代出部分记录的values迭代器，调用者对PartiallyUnrolledIterator进行统一迭代能够获取到全部记录，里面包含两种类型的记录：DeserializedMemoryEntry[T]和T。

3）基于记录迭代器，以序列化二进制格式保存Block数据

公众号：旧时光大数据

```
  private[storage] def putIteratorAsBytes[T](
      blockId: BlockId,
      values: Iterator[T],
      classTag: ClassTag[T],
      memoryMode: MemoryMode): Either[PartiallySerializedBlock[T], Long]

```

这种方式，调用这种希望将Block数据记录以二进制的格式保存在内存中。如果内存中能放得下，则返回最终的大小，否则返回一个PartiallySerializedBlock[T]迭代器。

如果Block数据记录能够完全放到内存中，则以SerializedMemoryEntry[T]格式放到内存的映射表中。如果Block数据记录只能部分放到内存中，则返回如下对象：

```
  Left(
    new PartiallySerializedBlock(
      this,
      serializerManager,
      blockId,
      serializationStream,
      redirectableStream,
      unrollMemoryUsedByThisBlock,
      memoryMode,
      bbos.toChunkedByteBuffer,
      values,
      classTag))

```

类似地，返回结果对象对调用者保持统一的迭代API视图。

2、DiskStore

DiskStore提供了将Block数据写入到磁盘的基本操作，它是通过DiskBlockManager来管理逻辑上Block到物理磁盘上Block文件路径的映射关系。当StorageLevel设置为如下值时，都可能会需要使用DiskStore来存储数据：

```
  DISK_ONLY
  DISK_ONLY_2
  MEMORY_AND_DISK
  MEMORY_AND_DISK_2
  MEMORY_AND_DISK_SER
  MEMORY_AND_DISK_SER_2
  OFF_HEAP

```

DiskBlockManager管理了每个Block数据存储位置的信息，包括从Block ID到磁盘上文件的映射关系。DiskBlockManager主要有如下几个功能：

- 负责创建一个本地节点上的指定磁盘目录，用来存储Block数据到指定文件中
- 如果Block数据想要落盘，需要通过调用getFile方法来分配一个唯一的文件路径
- 如果想要查询一个Block是否在磁盘上，通过调用containsBlock方法来查询
- 查询当前节点上管理的全部Block文件
- 通过调用createTempLocalBlock方法，生成一个唯一Block ID，并创建一个唯一的临时文件，用来存储中间结果数据
- 通过调用createTempShuffleBlock方法，生成一个唯一Block ID，并创建一个唯一的临时文件，用来存储Shuffle过程的中间结果数据

DiskStore提供的基本操作接口，与MemoryStore类似，比较简单，如下所示：

公众号：旧时光大数据

1）通过文件流写Block数据

该种方式对应的接口方法，如下所示：

```
1    def put(blockId: BlockId)(writeFunc: FileOutputStream => Unit): Unit
2
```

参数指定Block ID，还有一个写Block数据到打开的文件流的函数，在调用put方法时，首先会从DiskBlockManager分配一个Block ID对应的磁盘文件路径，然后将数据写入到该文件中。

2）将二进制Block数据写入文件

putBytes方法实现了，将一个Buffer中的Block数据写入指定的Block ID对应的文件中，方法定义如下所示：

```
1   def putBytes(blockId: BlockId, bytes: ChunkedByteBuffer): Unit
2
```

实际上，它是调用上面的put方法，将bytes中的Block二进制数据写入到Block文件中。

3）从磁盘文件读取Block数据

对应方法如下所示：

```
1    def getBytes(blockId: BlockId): ChunkedByteBuffer
2
```

通过给定的blockId，获取磁盘上对应的Block文件的数据，以ChunkedByteBuffer的形式返回。

4）删除Block文件

对应的删除方法定义，如下所示：

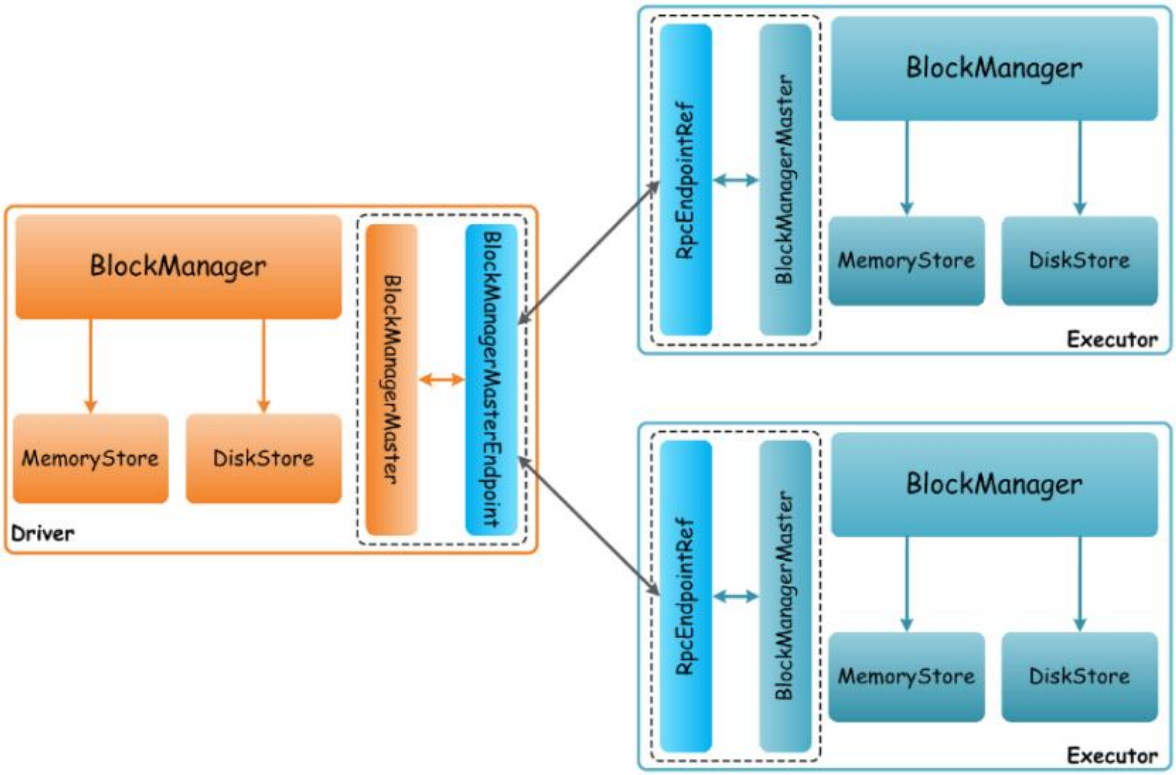
```
1    def remove(blockId: BlockId): Boolean1
2
```

通过DiskBlockManager查找到blockId对应的Block文件，然后删除掉。

3、BlockManager

谈到Spark中的Block数据存储，我们很容易能够想到BlockManager，他负责管理在每个Dirver和Executor上的Block数据，可能是本地或者远程的。具体操作包括查询Block、将Block保存在指定的存储中，如内存、磁盘、堆外（Off-heap）。而BlockManager依赖的后端，对Block数据进行内存、磁盘存储访问，都是基于前面讲到的MemoryStore、DiskStore。

在Spark集群中，当提交一个Application执行时，该Application对应的Driver以及所有的Executor上，都存在一个BlockManager、BlockManagerMaster，而BlockManagerMaster是负责管理各个BlockManager之间通信，这个BlockManager管理集群，如下图所示：



关于一个Application运行过程中Block的管理，主要是基于该Application所关联的一个Driver和多个Executor构建了一个Block管理集群：Driver上的(BlockManagerMaster, BlockManagerMasterEndpoint)是集群的Master角色，所有Executor上的(BlockManagerMaster, RpcEndpointRef)作为集群的Slave角色。当Executor上的Task运行时，会查询对应的RDD的某个Partition对应的Block数据是否处理过，这个过程中会触发多个BlockManager之间的通信交互。我们以ShuffleMapTask的运行为例，对应代码如下所示：

```
 override def runTask(context: TaskContext): MapStatus = {
   // Deserialize the RDD using the broadcast variable.
   val deserializeStartTime = System.currentTimeMillis()
   val ser = SparkEnv.get.closureSerializer.newInstance()
   val (rdd, dep) = ser.deserialize[(RDD[_], ShuffleDependency[_, _, _])](
     ByteBuffer.wrap(taskBinary.value),
Thread.currentThread.getContextClassLoader)
   _executorDeserializeTime = System.currentTimeMillis() - deserializeStartTime

   var writer: ShuffleWriter[Any, Any] = null
   try {
     val manager = SparkEnv.get.shuffleManager
     writer = manager.getWriter[Any, Any](dep.shuffleHandle, partitionId,
context)
     writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[_ <:
Product2[Any, Any]]]) // 处理RDD的Partition的数据
     writer.stop(success = true).get
   } catch {
     case e: Exception =>
       try {
         if (writer != null) {
           writer.stop(success = false)
         }
       } catch {
         case e: Exception =>
           log.debug("Could not stop writer", e)
       }
       throw e
   }
```

公众号：旧时光大数据

```
27      }
28
```

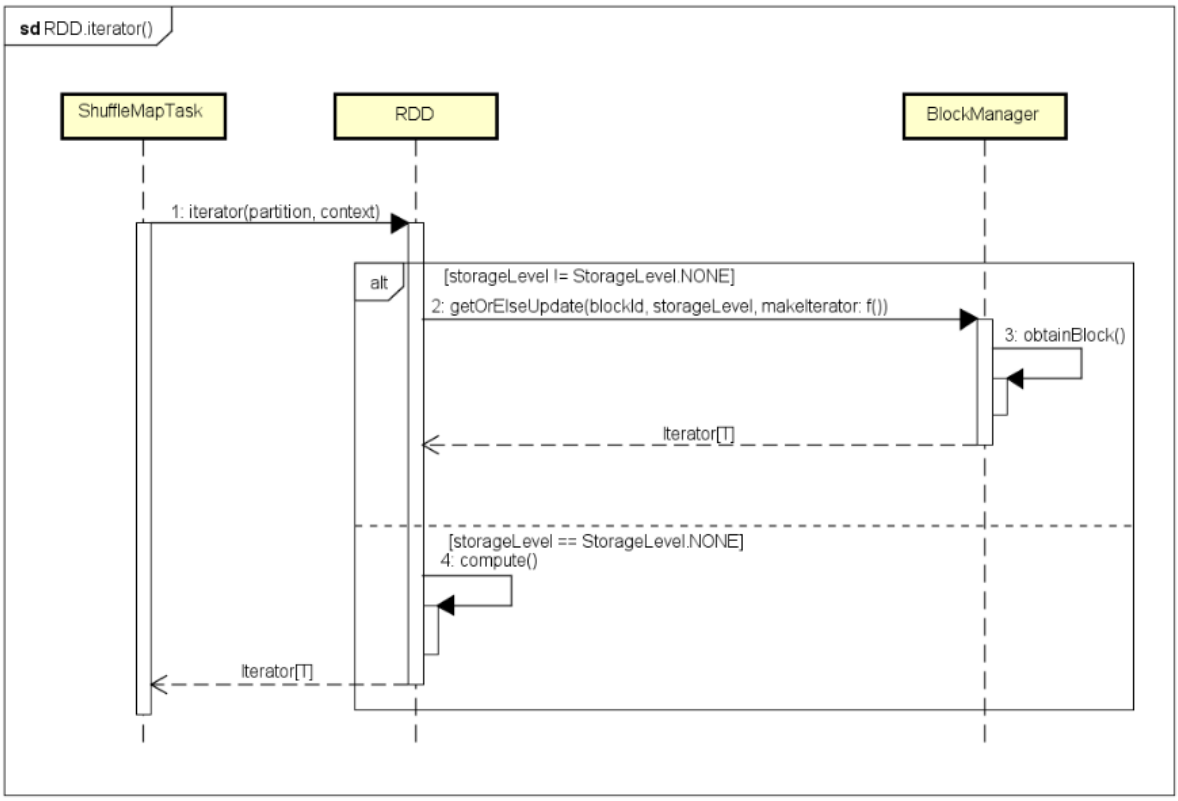
一个RDD的Partition对应一个ShuffleMapTask，一个ShuffleMapTask会在一个Executor上运行，它负责处理RDD的一个Partition对应的数据，基本处理流程，如下所示：

- 根据该Partition的数据，创建一个RDDBlockId（由RDD ID和Partition Index组成），即得到一个稳定的blockId（如果该Partition数据被处理过，则可能本地或者远程Executor存储了对应的Block数据）。
- 先从BlockManager获取该blockId对应的数据是否存在，如果本地存在（已经处理过），则直接返回Block结果（BlockResult）；否则，查询远程的Executor是否已经处理过该Block，处理过则直接通过网络传输到当前Executor本地，并根据StorageLevel设置，保存Block数据到本地MemoryStore或DiskStore，同时通过BlockManagerMaster上报Block数据状态（通知Driver当前的Block状态，亦即，该Block数据存储在哪个BlockManager中）。
- 如果本地及远程Executor都没有处理过该Partition对应的Block数据，则调用RDD的compute方法进行计算处理，并将处理的Block数据，根据StorageLevel设置，存储到本地MemoryStore或DiskStore。
- 根据ShuffleManager对应的ShuffleWriter，将返回的该Partition的Block数据进行Shuffle写入操作

下面，我们基于上面逻辑，详细分析在这个处理过程中重要的交互逻辑：

1）根据RDD获取一个Partition对应数据的记录迭代器

用户提交的Spark Application程序，会设置对应的StorageLevel，所以设置与不设置对该处理逻辑有一定影响，具有两种情况，如下图所示：

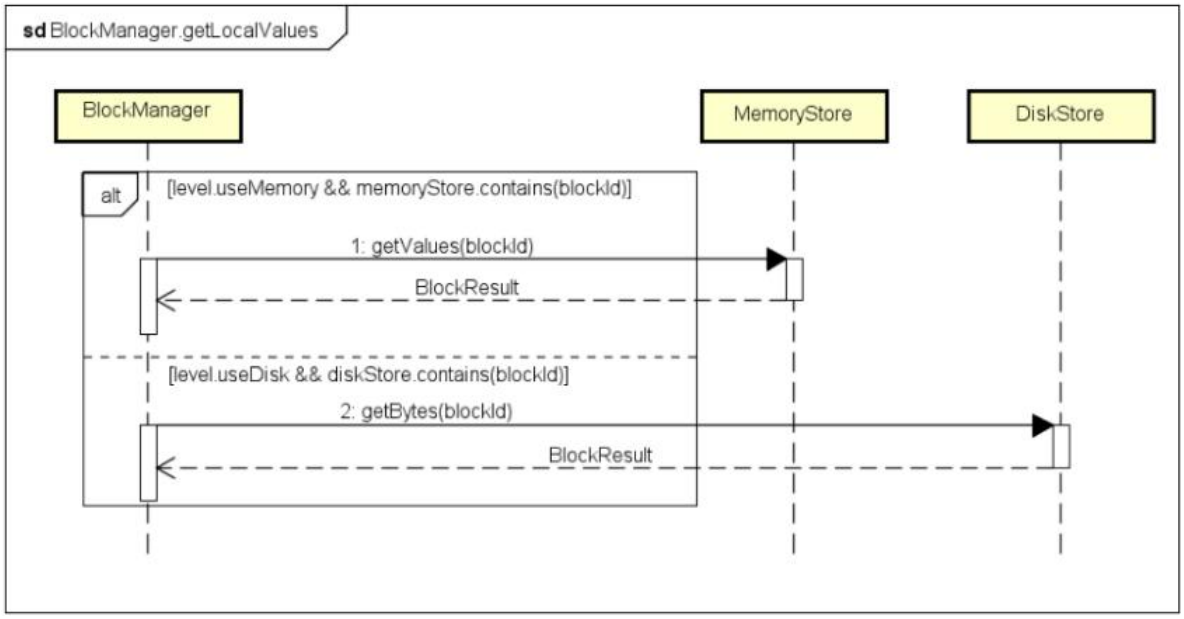


如果用户程序设置了StorageLevel，可能该Partition的数据已经处理过，那么对应的处理结果Block数据可能已经存储。一般设置的StorageLevel，或者将Block存储在内存中，或者存储在磁盘上，这里会尝试调用getOrElseUpdate()方法获取对应的Block数据，如果存在则直接返回Block对应的记录的迭代器实例，就不需要重新计算了，如果没有找到对应的已经处理过的Block数据，则调用RDD的compute()方法进行处理，处理结果根据StorageLevel设置，将Block数据存储在内存或磁盘上，缓存供后续Task重复使用。

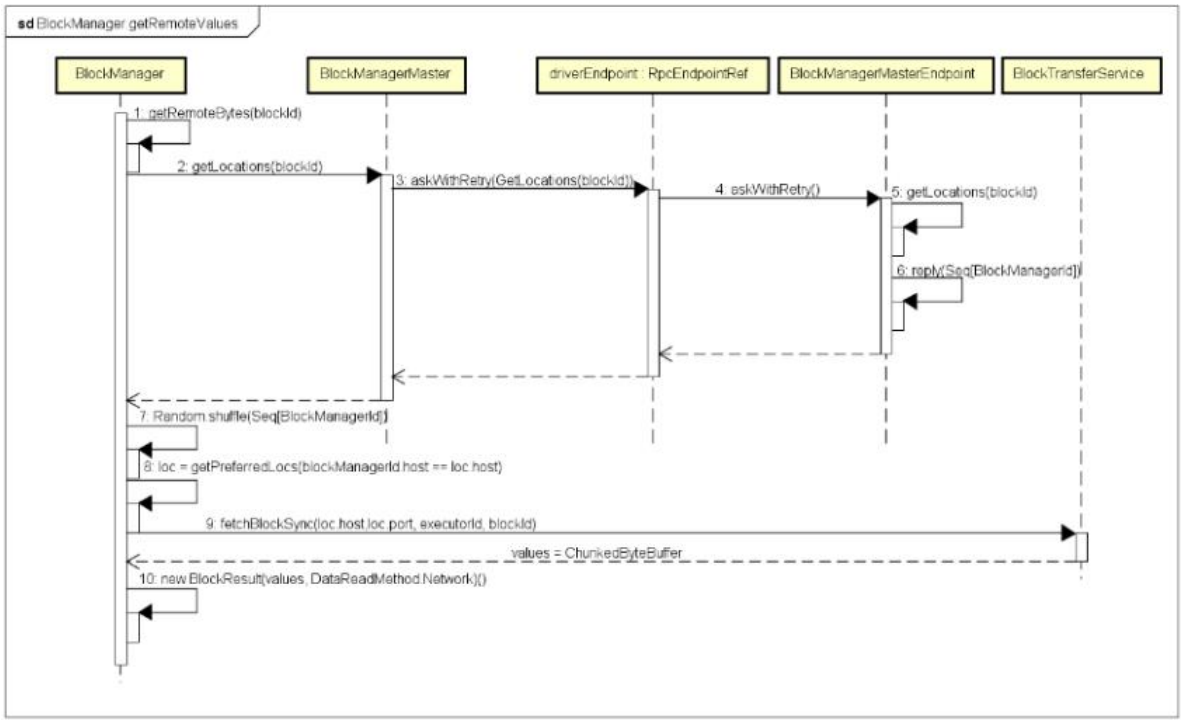
如果用户程序没有设置StorageLevel，那么RDD对应的该Partition的数据一定没有进行处理过，即使处理过，如果没有进行Checkpointing，也需要重新计算（如果进行了Checkpointing，可以直接从缓存中获取），直接调用RDD的compute()方法进行处理。

2）从BlockManager查询获取Block数据

每个Executor上都有一个BlockManager实例，负责管理用户提交的该Application计算过程中产生的Block。很有可能当前Executor上存储在RDD对应Partition的经过处理后得到的Block数据，也有可能当前Executor上没有，但是其他Executor上已经处理过并缓存了Block数据，所以对应着本地获取、远程获取两种可能，本地获取交互逻辑如下图所示：



从本地获取，根据StorageLevel设置，如果是存储在内存中，则从本地的MemoryStore中查询，存在则读取并返回；如果是存储在磁盘上，则从本地的DiskStore中查询，存在则读取并返回。本地不存在，则会从远程的Executor读取，对应的组件交互逻辑，如下图所示：



远程获取交互逻辑相对比较复杂：当前Executor上的BlockManager通过BlockManagerMaster，向远程的Driver上的BlockManagerMasterEndpoint查询对应Block ID，有哪些Executor已经保存了该Block数据，Dirver返回一个包含了该Block数据的Location列表，如果对应的Location信息与当前ShuffleMapTask执行所在Executor在同一台节点上，则会优先使用该Location，因为同一节点上的多个Executor之间传输Block数据效率更高。

这里需要说明的是，如果对应的Block数据的StorageLevel设置为写磁盘，通过前面我们知道，DiskStore是通过DiskBlockManager进行管理存储到磁盘上的Block数据文件的，在同一个节点上的多个Executor共享相同的磁盘文件路径，相同的Block数据文件也就会被同一个节点上的多个Executor所共享。而对应MemoryStore，因为每个Executor对应独立的JVM实例，从而具有独立的Storage/Execution内存管理，所以使用MemoryStore不能共享同一个Block数据，但是同一个节点上的多个Executor之间的MemoryStore之间拷贝数据，比跨网络传输要高效的多。

4、BlockInfoManager

用户提交一个Spark Application程序，如果程序对应的DAG图相对复杂，其中很多Task计算的结果Block数据都有可能被重复使用，这种情况下如何去控制某个Executor上的Task线程去读写Block数据呢？其实，BlockInfoManager就是用来控制Block数据读写操作，并且跟踪Task读写了哪些Block数据的映射关系，这样如果两个Task都想去处理同一个RDD的同一个Partition数据，如果没有锁来控制，很可能两个Task都会计算并写同一个Block数据，从而造成混乱。我们分析每种情况下，BlockInfoManager是如何管理Block数据（同一个RDD的同一个Partition）读写的：

1）第一个Task请求写Block数据

这种情况下，没有其他Task写Block数据，第一个Task直接获取到写锁，并启动写Block数据到本地MemoryStore或DiskStore。如果其他写Block数据的Task也请求写锁，则该Task会阻塞，等待第一个获取写锁的Task完成写Block数据，直到第一个Task写完成，并通知其他阻塞的Task，然后其他Task需要再次获取到读锁来读取该Block数据。

2）第一个Task正在写Block数据，其他Task请求读Block数据

这种情况，Block数据没有完成写操作，其他读Block数据的Task只能阻塞，等待写Block的Task完成并通知读Task去读取Block数据。

3）Task请求读取Block数据

如果该Block数据不存在，则直接返回空，表示当前RDD的该Partition并没有被处理过。如果当前Block数据存在，并且没有其他Task在写，表示已经完成了些Block数据操作，则该Task直接读取该Block数据。

Spark怎么保证数据不丢失

问过的一些公司：京东

参考答案：

在Spark Streaming的生产实践中，要做到数据零丢失，需要满足以下几个先决条件：

1、输入的数据源是可靠的/数据接收器是可靠的

对于一些输入数据源（比如Kafka），Spark Streaming可以对已经接收的数据进行确认。输入的数据首先被接收器（receivers）所接收，然后存储到Spark中（默认情况下，数据保存到2个执行器中以便进行容错）。数据一旦存储到Spark中，接收器可以对它进行确认（比如，如果消费Kafka里面的数据时可以更新Zookeeper里面的偏移量）。这种机制保证了在接收器突然挂掉的情况下也不会丢失数据：因为数据虽然被接收，但是没有被持久化的情况下是不会发送确认消息的。所以在接收器恢复的时候，数据可以被原端重新发送。

2、应用程序的metadata被application的driver持久化了(checkpointed)

可靠的数据源和接收器可以让我们从接收器挂掉的情况下恢复（或者是接收器运行的Exectuor和服务器挂掉都可以）。但是更棘手的问题是，如果Driver挂掉如何恢复？对此引入了很多技术来让Driver从失败中恢复。其中一个就是对应用程序的元数据进行Checkpoint。利用这个特性，Driver可以将应用程序的重要元数据持久化到可靠的存储中，比如HDFS；然后Driver可以利用这些持久化的数据进行恢复。

经过上面两点还是可能存在数据丢失的场景如下：

公众号：旧时光大数据

1）两个Exectuor已经从接收器中接收到输入数据，并将它缓存到Exectuor的内存中；

2）接收器通知输入源数据已经接收；

3）Exectuor根据应用程序的代码开始处理已经缓存的数据；

4）这时候Driver突然挂掉了；

5）从设计的角度看，一旦Driver挂掉之后，它维护的Exectuor也将全部被kill；

6）既然所有的Exectuor被kill了，所以缓存到它们内存中的数据也将被丢失。结果，这些已经通知数据源但是还没有处理的缓存数据就丢失了；

7）缓存的时候不可能恢复，因为它们是缓存在Exectuor的内存中，所以数据被丢失了。

为了解决上面提到的糟糕场景，Spark Streaming 1.2开始引入了下面的WAL机制。

3、启用了WAL特性（Write ahead log）

但如果Exectuor已经接收并缓存了数据,这个时候挂掉了,这个时候数据是在Exectuor中,数据还是会丢失,启用了WAL机制，所以已经接收的数据被接收器写入到容错存储中，比如HDFS。由于采用了WAl机制，Driver可以从失败的点重新读取数据，即使Exectuor中内存的数据已经丢失了。在这个简单的方法下，Spark Streaming提供了一种即使是Driver挂掉也可以避免数据丢失的机制。

虽然WAL可以确保数据不丢失，它并不能对所有的数据源保证exactly-once语义。想象一下可能发生在Spark Streaming整合Kafka的糟糕场景。

1）接收器接收到输入数据，并把它存储到WAL中；

2）接收器在更新Zookeeper中Kafka的偏移量之前突然挂掉了；

3）Spark Streaming假设输入数据已成功收到（因为它已经写入到WAL中），然而Kafka认为数据被没有被消费，因为相应的偏移量并没有在Zookeeper中更新；

4）过了一会，接收器从失败中恢复；

5）那些被保存到WAL中但未被处理的数据被重新读取；

6）一旦从WAL中读取所有的数据之后，接收器开始从Kafka中消费数据。因为接收器是采用Kafka的High-Level Consumer API实现的，它开始从Zookeeper当前记录的偏移量开始读取数据，但是因为接收器挂掉的时候偏移量并没有更新到Zookeeper中，所有有一些数据被处理了2次。

除了上面描述的场景，**WAL还有其他两个不可忽略的缺点**：

1）WAL减少了接收器的吞吐量，因为接受到的数据必须保存到可靠的分布式文件系统中。

2）对于一些输入源来说，它会重复相同的数据。比如当从Kafka中读取数据，你需要在Kafka的brokers中保存一份数据，而且你还得在Spark Streaming中保存一份。

Kafka direct API

为了解决由WAL引入的性能损失，并且保证 exactly-once 语义，Spark Streaming 1.3中引入了名为Kafka direct API。

这个想法对于这个特性是非常明智的。Spark driver只需要简单地计算下一个batch需要处理Kafka中偏移量的范围，然后命令Spark Exectuor直接从Kafka相应Topic的分区中消费数据。换句话说，这种方法把Kafka当作成一个文件系统，然后像读文件一样来消费Topic中的数据。

在这个简单但强大的设计中:

- 不再需要Kafka接收器，Exectuor直接采用Simple Consumer API从Kafka中消费数据。
- 不再需要WAL机制，我们仍然可以从失败恢复之后从Kafka中重新消费数据；
- exactly-once语义得以保存，我们不再从WAL中读取重复的数据。

公众号：旧时光大数据

Spark SQL如何使用UDF?

问过的一些公司：小米

参考答案：

将代码打包成udf.jar，存放在/home/Hadoop/lib下

方式一：在启动spark-sql时通过--jars指定

```
    cd $SPARK_HOME/bin
    spark-sql --jars /home/hadoop/lib/udf.jar
    CREATE TEMPORARY FUNCTION hello AS 'com.luogankun.udf.HelloUDF';
    select hello(url) from page_views limit 1;

```

方式二：先启动spark-sql后add jar

```
    cd $SPARK_HOME/bin
    spark-sql
    add jar /home/hadoop/lib/udf.jar;
    CREATE TEMPORARY FUNCTION hello AS 'com.luogankun.udf.HelloUDF';

    select hello(url) from page_views limit 1;

```

在测试过程中发现并不支持该种方式，会报java.lang.ClassNotFoundException: com.luogankun.udf.HelloUDF

解决方案如下：

1）需要先将udf.jar的路径配置到spark-env.sh的SPARK_CLASSPATH中，形如：

```
    export SPARK_CLASSPATH=$SPARK_CLASSPATH:/home/hadoop/software/mysql-connector-java-5.1.27-bin.jar:/home/hadoop/lib/udf.jar

```

2）再启动spark-sql，直接CREATE TEMPORARY FUNCTION即可；

```
    cd $SPARK_HOME/bin
    spark-sql
    CREATE TEMPORARY FUNCTION hello AS 'com.luogankun.udf.HelloUDF';

    select hello(url) from page_views limit 1;

```

方式三：Thrift JDBC Server中使用UDF

在beeline命令行中执行：

```
    add jar /home/hadoop/lib/udf.jar;
    CREATE TEMPORARY FUNCTION hello AS 'com.luogankun.udf.HelloUDF';

    select hello(url) from page_views limit 1;

```

公众号：旧时光大数据

Spark温度二次排序

问过的一些公司：

参考答案：

1、输入数据格式

```
2020    5   30  21
2020    6   6   29
2020    4   16  26
2020    4   16  24

```

2、预期结果

按月份排序，再按温度排序

```
(2020-4,25)
(2020-4,25)
(2020-4,25)
(2020-4,26)

```

3、代码

```
 package com.spark

 import org.apache.spark.rdd.RDD
 import org.apache.spark.{SparkConf, SparkContext}

 class UDFSort(val first: Int, val second: Int) extends Ordered[UDFSort] with
Serializable {
   override def compare(that: UDFSort): Int = {
     if (this.first - that.first != 0) {
       //第一个值不相等的时候，直接返回大小
       this.first - that.first //返回值
     }
     else {
       //第一个值相等的时候,比较第二个值
       this.second - that.second
     }
   }
 }

 object SecondSort {

   def main(args: Array[String]): Unit = {

     //创建SparkConf对象
     val config: SparkConf = new
SparkConf().setMaster("local[*]").setAppName("SecondSort")
     //创建Spark上下文对象
     val sc = new SparkContext(config)

     //文件输入输出路径
     val inputPath = "D:\\IDEA2018\\Scala\\spark-study\\src\\test.txt"
     val outputPath = "D:\\IDEA2018\\Scala\\spark-study\\src\\test1.txt"

```

公众号：旧时光大数据

```
32        //读取文件 将文件内容一行一行的读取出来
33        val input: RDD[String] = sc.textFile(inputPath)
34
35        val mapRDD: RDD[(UDFSort, (String, String))] = input.map(line => {
36          val fields: Array[String] = line.split("\t")
37          (new UDFSort(fields(1).toInt, fields(3).toInt), (fields(0) + "-" +
    fields(1), fields(3)))
38        })
39        val sortRDD: RDD[(String, String)] = mapRDD.sortByKey(true, 1).map(x => {
40          (x._2._1, x._2._2)
41        }).reduceByKey(_ + "\t" + _)
42
43        sortRDD.foreach(println(_))
44
45      }
46
47    }
48
```

Spark实现wordcount

可回答：wordcount用了哪些算子

问过的一些公司：字节，作业帮，金山云，携程(2021.09)x2

参考答案：

1、WordCount案例

需求：使用netcat工具向9999端口不断的发送数据，通过SparkStreaming读取端口数据并统计不同单词出现的次数

1）添加依赖

```
1    <dependency>
2        <groupId>org.apache.spark</groupId>
3        <artifactId>spark-streaming_2.12</artifactId>
4        <version>3.0.0</version>
5    </dependency>
6
```

2）代码

```
1     object StreamWordCount {
2
3       def main(args: Array[String]): Unit = {
4
5         //1.初始化Spark配置信息
6         val sparkConf = new
    SparkConf().setMaster("local[*]").setAppName("StreamWordCount")
7
8         //2.初始化SparkStreamingContext
9         val ssc = new StreamingContext(sparkConf, Seconds(3))
10
11        //3.通过监控端口创建DStream，读进来的数据为一行行
12        val lineStreams = ssc.socketTextStream("linux1", 9999)
13
```

公众号：旧时光大数据

```
        //将每一行数据做切分，形成一个个单词
        val wordStreams = lineStreams.flatMap(_.split(" "))

        //将单词映射成元组（word,1）
        val wordAndOneStreams = wordStreams.map((_, 1))

        //将相同的单词次数做统计
        val wordAndCountStreams = wordAndOneStreams.reduceByKey(_+_)

        //打印
        wordAndCountStreams.print()

        //启动SparkStreamingContext
        ssc.start()
        ssc.awaitTermination()
      }
    }

```

3）启动程序并通过netcat发送数据：

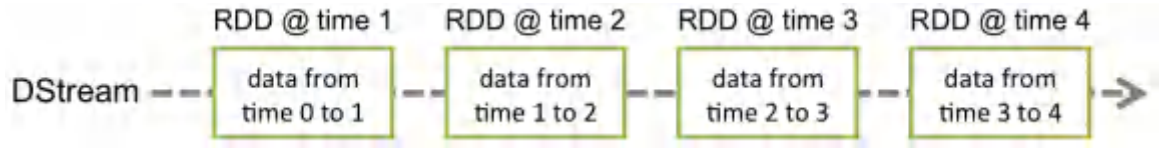
```
    nc -lk 9999
    hello spark

```

简洁版：

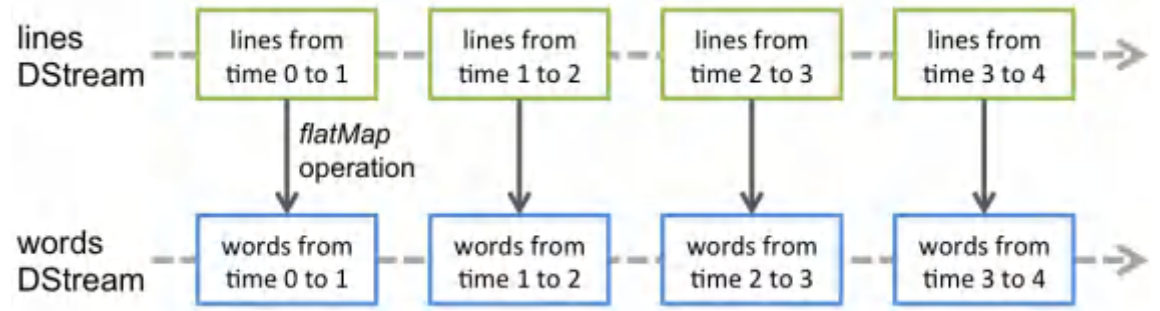
```
  object WordCount {
      def main(args: Array[String]): Unit = {
          val lines = List("兰 亭 临 帖", "行 书 如 行 云 流 水")
          println(lines.flatMap(_.split(" ")).map((_,1)).groupBy(_._1).map(x =>
  (x._1, x._2.size)).toList.sortBy(_._2))
      }
  }

```

2、WordCount解析

Discretized Stream是Spark Streaming的基础抽象，代表持续性的数据流和经过各种Spark原语操作后的结果数据流。在内部实现上，DStream是一系列连续的RDD来表示。每个RDD含有一段时间间隔内的数据。



对数据的操作也是按照RDD为单位来进行的



计算过程由Spark Engine来完成

Spark Streaming怎么实现数据持久化保存?

问过的一些公司：作业帮

参考答案：

1、缓存与持久化机制

与RDD类似，Spark Streaming也可以让开发人员手动控制，将数据流中的数据持久化到内存中。对DStream调用persist()方法，就可以让Spark Streaming自动将该数据流中的所有产生的RDD，都持久化到内存中。如果要对一个DStream多次执行操作，那么，对DStream持久化是非常有用的。因为多次操作，可以共享使用内存中的一份缓存数据。

对于基于窗口的操作，比如reduceByWindow、reduceByKeyAndWindow，以及基于状态的操作，比如updateStateByKey，默认就隐式开启了持久化机制。即Spark Streaming默认就会将上述操作产生的Dstream中的数据，缓存到内存中，不需要开发人员手动调用persist()方法。

对于通过网络接收数据的输入流，比如socket、Kafka、Flume等，默认的持久化级别，是将数据复制一份，以便于容错。相当于是，用的是类似MEMORY_ONLY_SER_2。

与RDD不同的是，默认的持久化级别，统一都是要序列化的。

2、Checkpoint机制

Checkpoint机制概述

每一个Spark Streaming应用，正常来说，都是要7 * 24小时运转的，这就是实时计算程序的特点。因为要持续不断的对数据进行计算。因此，对实时计算应用的要求，应该是必须要能够对与应用程序逻辑无关的失败，进行容错。

如果要实现这个目标，Spark Streaming程序就必须将足够的信息checkpoint到容错的存储系统上，从而让它能够从失败中进行恢复。有两种数据需要被进行checkpoint：

1）元数据checkpoint——将定义了流式计算逻辑的信息，保存到容错的存储系统上，比如HDFS。当运行Spark Streaming应用程序的Driver进程所在节点失败时，该信息可以用于进行恢复。元数据信息包括了：

- 配置信息——创建Spark Streaming应用程序的配置信息，比如SparkConf中的信息。
- DStream的操作信息——定义了Spark Stream应用程序的计算逻辑的DStream操作信息。
- 未处理的batch信息——那些job正在排队，还没处理的batch信息。

2）数据checkpoint——将实时计算过程中产生的RDD的数据保存到可靠的存储系统中。

对于一些将多个batch的数据进行聚合的，有状态的transformation操作，这是非常有用的。在这种transformation操作中，生成的RDD是依赖于之前的batch的RDD的，这会导致随着时间的推移，RDD的依赖链条变得越来越长。

要避免由于依赖链条越来越长，导致的一起变得越来越长的失败恢复时间，有状态的transformation操作执行过程中间产生的RDD，会定期地被checkpoint到可靠的存储系统上，比如HDFS。从而削减RDD的依赖链条，进而缩短失败恢复时，RDD的恢复时间。

一句话概括，元数据checkpoint主要是为了从driver失败中进行恢复；而RDD checkpoint主要是为了，使用到有状态的transformation操作时，能够在其生产出的数据丢失时，进行快速的失败恢复。

公众号：旧时光大数据

何时启用Checkpoint机制?

1）使用了有状态的transformation操作——比如updateStateByKey，或者reduceByKeyAndWindow操作，被使用了，那么checkpoint目录要求是必须提供的，也就是必须开启checkpoint机制，从而进行周期性的RDD checkpoint。

2）要保证可以从Driver失败中进行恢复——元数据checkpoint需要启用，来进行这种情况的恢复。

要注意的是，并不是说，所有的Spark Streaming应用程序，都要启用checkpoint机制，如果即不强制要求从Driver失败中自动进行恢复，又没使用有状态的transformation操作，那么就不需要启用checkpoint。事实上，这么做反而是有助于提升性能的。

如何启用Checkpoint机制?

1）对于有状态的transformation操作，启用checkpoint机制，定期将其生产的RDD数据checkpoint，是比较简单的。

可以通过配置一个容错的、可靠的文件系统（比如HDFS）的目录，来启用checkpoint机制，checkpoint数据就会写入该目录。使用StreamingContext的checkpoint()方法即可。然后，你就可以放心使用有状态的transformation操作了。

2）如果为了要从Driver失败中进行恢复，那么启用checkpoint机制，是比较复杂的。需要改写Spark Streaming应用程序。

当应用程序第一次启动的时候，需要创建一个新的StreamingContext，并且调用其start()方法，进行启动。当Driver从失败中恢复过来时，需要从checkpoint目录中记录的元数据中，恢复出来一个StreamingContext。

Checkpoint的说明

将RDD checkpoint到可靠的存储系统上，会耗费很多性能。当RDD被checkpoint时，会导致这些batch的处理时间增加。因此，checkpoint的间隔，需要谨慎的设置。

对于那些间隔很多的batch，比如1秒，如果还要执行checkpoint操作，则会大幅度削减吞吐量。而另外一方面，如果checkpoint操作执行的太不频繁，那就会导致RDD的lineage变长，又会有失败恢复时间过长的风险。

对于那些要求checkpoint的有状态的transformation操作，默认的checkpoint间隔通常是batch间隔的数倍，至少是10秒。使用DStream的checkpoint()方法，可以设置这个DStream的checkpoint的间隔时长。通常来说，将checkpoint间隔设置为窗口操作的滑动间隔的5~10倍，是个不错的选择。

Spark SQL读取文件，内存不够使用，如何处理?

问过的一些公司：腾讯

参考答案：

增加驱动程序内存

Spark的lazy体现在哪里?

问过的一些公司：字节

参考答案：

Spark通过lazy特性，可以进行底层的spark应用执行的优化。

Spark中的并行度等于什么

问过的一些公司：端点数据(2021.07)

参考答案：

spark作业中，各个stage的task的数量，也就代表了spark作业在各个阶段stage的并行度。也可以说等于RDD的一个分区数。

Spark运行时并行度的设置

问过的一些公司：北京元安物联社招

参考答案：

1、Spark的并行度指的是什么?

spark作业中，各个stage的task的数量，也就代表了spark作业在各个阶段stage的并行度。

当分配完所能分配的最大资源了，然后对应资源去调节程序的并行度，如果并行度没有与资源相匹配，那么导致你分配下去的资源都浪费掉了。同时并行运行，还可以让每个task要处理的数量变少（很简单的原理。合理设置并行度，可以充分利用集群资源，减少每个task处理数据量，而增加性能加快运行速度）。

假如，现在已经在spark-submit 脚本里面，给我们的spark作业分配了足够多的资源，比如**50个executor**，每个**executor 有10G内存**，**每个executor有3个cpu core** 。 基本已经达到了集群或者yarn队列的资源上限。

task没有设置，或者设置的很少，比如就设置了，100个task 。**50个executor，每个executor 有3个core，也就是说**** Application 任何一个stage运行的时候，都有总数150个cpu core，可以并行运行。但是，你现在只有100个task，平均分配一下，每个executor 分配到2个task，ok，那么同时在运行的task，只有100个task，每个executor 只会并行运行 2个task。 每个executor 剩下的一个cpu core 就浪费掉了！你的资源，虽然分配充足了，但是问题是， 并行度没有与资源相匹配，导致你分配下去的资源都浪费掉了。合理的并行度的设置，应该要设置的足够大，大到可以完全合理的利用你的集群资源；** 比如上面的例子，总共集群有150个cpu core，可以并行运行150个task。那么你就应该将你的Application 的并行度，至少设置成150个，才能完全有效的利用你的集群资源，让150个task，并行执行，而且task增加到150个以后，即可以同时并行运行，还可以让每个task要处理的数量变少；比如总共 150G 的数据要处理，如果是100个task，每个task 要计算1.5G的数据。 现在增加到150个task，每个task只要处理1G数据。

2、如何去提高并行度

1）**task数量，至少设置成与spark Application 的总cpu core 数量相同**（最理性情况，150个core，分配150task，一起运行，差不多同一时间运行完毕）**官方推荐，task数量，设置成spark Application 总cpu core数量的2~3倍，比如150个cpu core，基本设置 task数量为 300~ 500.** 与理性情况不同的，有些task会运行快一点，比如50s 就完了，有些task 可能会慢一点，要一分半才运行完，所以如果你的task数量，刚好设置的跟cpu core 数量相同，可能会导致资源的浪费，因为 比如150task，10个先运行完了，剩余140个还在运行，但是这个时候，就有10个cpu core空闲出来了，导致浪费。如果设置2~3倍，那么一个task运行完以后，另外一个task马上补上来，尽量让cpu core不要空闲。同时尽量提升spark运行效率和速度。提升性能。

2）如何设置一个Spark Application的并行度?

公众号：旧时光大数据

spark.defalut.parallelism：默认是没有值的，如果设置了值比如说10，是在shuffle的过程才会起作用（val rdd2 = rdd1.reduceByKey(+) //rdd2的分区数就是10，rdd1的分区数不受这个参数的影响）

```
    new SparkConf().set("spark.defalut.parallelism",""500)

```

3）如果读取的数据在HDFS上，增加block数，默认情况下split与block是一对一的，而split又与RDD中的partition对应，所以增加了block数，也就提高了并行度。

4）RDD.repartition，给RDD重新设置partition的数量

5）reduceByKey的算子指定partition的数量

```
    val rdd2 = rdd1.reduceByKey(_+_,10)  val rdd3 = rdd2.map.filter.reduceByKey(_+_)

```

6）val rdd3 = rdd1.join（rdd2）

rdd3里面partiiton的数量是由父RDD中最多的partition数量来决定，因此使用join算子的时候，增加父RDD中partition的数量。

7）设置spark sql中shuffle过程中partitions的数量

```
    spark.sql.shuffle.partitions  //spark sql中shuffle过程中partitions的数量

```

Spark SQL的数据倾斜

问过的一些公司：网易

参考答案：

有些可以参考前面的数据倾斜部分的答案

Spark SQL的数据倾斜解决方案：

- 聚合源数据：Spark Core和Spark SQL没有任何区别
- 过滤导致倾斜的key：在sql中用where条件
- 提高shuffle并行度：groupByKey(1000)，spark.sql.shuffle.partitions（默认是200）
- 双重groupBy：改写SQL，两次groupBy
- reduce join转换为map join：spark.sql.autoBroadcastJoinThreshold（默认是10485760）；可以自己将表做成RDD，自己手动去实现map join；SparkSQL内置的map join，默认如果有一个10M以内的小表，会将该表进行broadcast，然后执行map join；调节这个阈值，比如调节到20M、50M、甚至1G。
- 采样倾斜key并单独进行join：纯Spark Core的一种方式，sample、filter等算子
- 随机key与扩容表：Spark SQL+Spark Core

Spark的exactly-once

问过的一些公司：触宝(2021.07)

参考答案：

exactly once指的是在处理数据的过程中，系统有很好的容错性(fault-tolerance)，能够保证数据处理不重不丢，每一条数据仅被处理一次。

公众号：旧时光大数据

分布式系统Exactly-Once的一致性保障，不是依靠某个环节的强一致性，而是要求系统的全流程均保持Exactly-Once一致性！！

Spark具备很好的机制来保证exactly once的语义，具体体现在数据源的可重放性、计算过程中的容错性、以及写入存储介质时的幂等性或者事务性。

实时场景下，Spark在整个流式处理中如何保证Exactly-Once一致性是重中之重。这需要整个系统的各环节均保持强一致性，包括**可靠的数据源端**(数据可重复读取、不丢失) 、**可靠的消费端**(Spark内部精确一次消费)、**可靠的输出端**(幂等性、事务)。

1、数据源端

支持可靠的数据源接入(例如Kafka)，源数据可重读 。

Spark Streaming内置的Kafka Direct API (KafkaUtils.createDirectStream)。实现**精确Exactly-Once一致性语义。**

- Spark Streaming 自己管理offset（手动提交offset)，并保持到checkpoint中

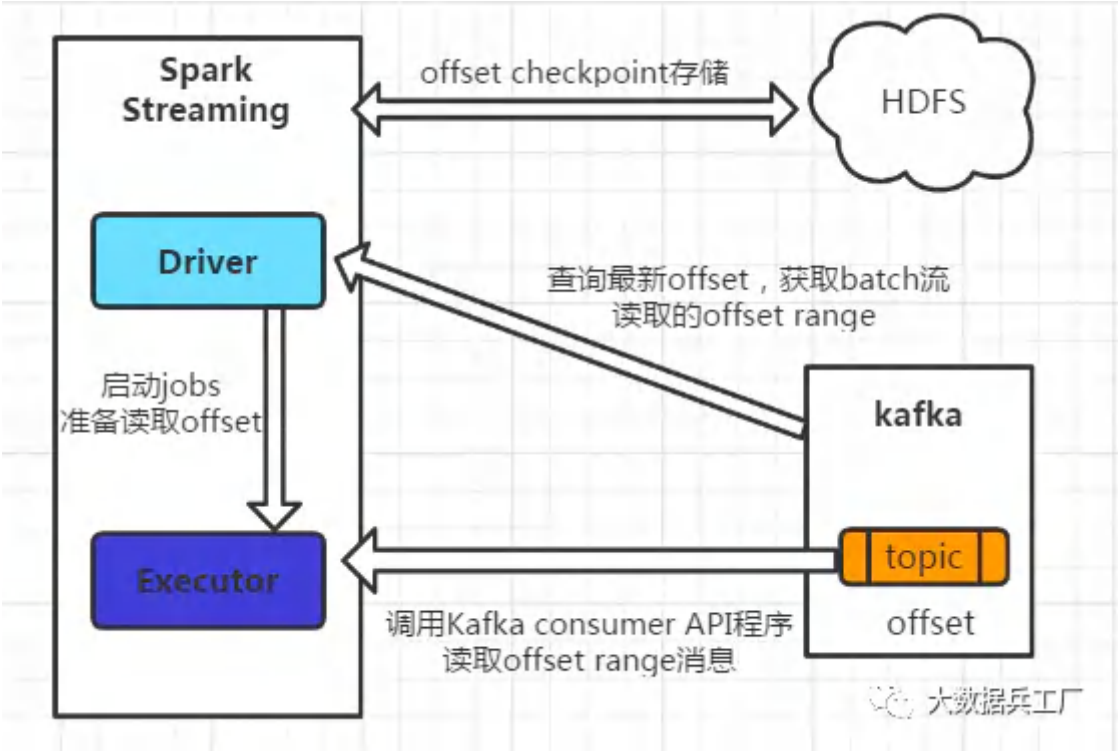
Spark Streaming此时直接调用Kafka Consumer的API，自己管理维护offset(包括同步提交offset、保存checkpoint)，所以即使在重启情况下数据也不会重复。

```
1   val ssc = new StreamingContext(sc, Seconds(5))
2
3   val kafkaParams = Map[String,String]{
4     "bootstrap.servers" -> "m1:9092,m2:9092,m3:9092",
5     "group.id" -> "spark",
6     "auto.offset.reset" -> "smallest"
7   }
8   // 直连方式拉取数据，这种方式不会修改数据的偏移量，需要手动的更新
9   val lines = kafkaUtils.createDirectStream[String, String, StringDecoder,
    StringDecoder](ssc, kafkaParams)
10
```

Driver进程保持与Kafka通信，定期获取最新offset range范围，Executor进程根据offset range拉取kafka消息。因为Kafka本身offset就具有唯一特性，且Spark Streaming此时作为唯一的消费者，故**全过程保持Exactly-once的一致性状态。**

注意：如果程序崩溃，整个流可能会从earliest/latest处恢复重读，需考虑其他后续处理。

公众号：旧时光大数据



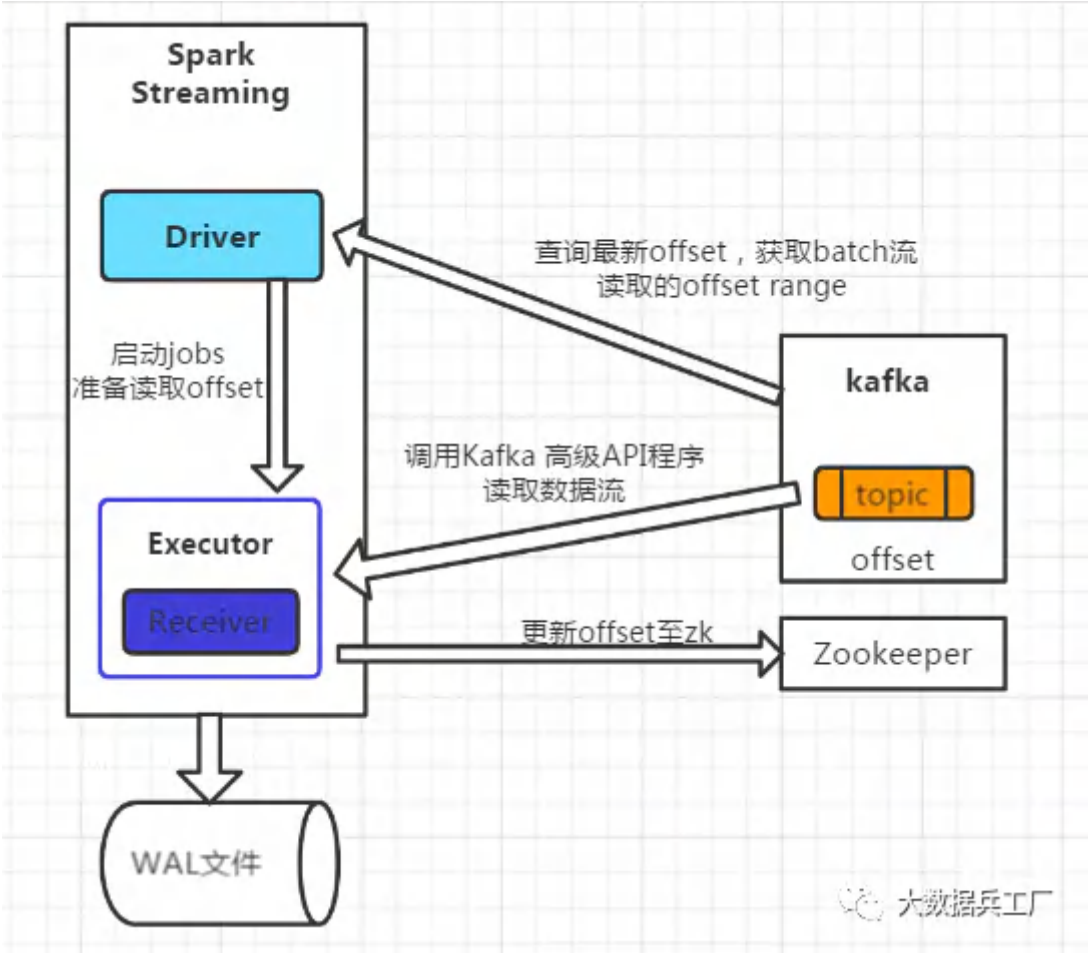
Spark Streaming 基于Receiver的Kafka高级API，**实现At least Once语义。**

- 基于Spark Streaming的Receiver模式，在Executor持续拉取kafka数据流
- kafka数据存储到Executor内存和WAL(预写日志)中
- WAL(预先日志)写入完成后，自动更新offset至zookeeper上

利用Spark本身的Receivers线程接收数据，内部调用Kafka高级消费API，不断触发batch消息拉取。获取的kafka数据在Executor本地存储，也可以启用WAL预写文件，将数据存储到第三方介质(HDFS)中。

```
1   val sparkConf = new SparkConf().setAppName("kafkaWordCount")
2   val ssc = new StreamingContext(sparkConf, Seconds(2))
3   ssc.checkpoint("checkpoint")
4
5   val topicMap = topics.split(",").map((_, numThreamds, toInt)).toMap
6   val lines = KafkaUtils.createStream(ssc, zkQuorum, group, topicMap).map(_._2)
7
```

此过程**仅可实现At least once（至少一次）**，也就是说数据可能会被重复读取。即使理论上WAL机制可确保数据不丢失，但是会存在消息写入WAL完成，但因其他原因无法及时更新offset至zookeeper的情况。此时kafka会重新发送offset，造成数据在Executor中多存储一份。



综上所述：

1）高级消费者API需要启用Receiver线程消费Kafka数据，相较于第一种增加了开销，且无法直接实现并行读取，需要使用多个Kafka Dtstream 消费同一组然后union。

2）高级消费API在Executor本地和WAL存储两份数据<开启WAL不丢失机制>，而第一种Direct API仅在Executor中存储数据<offset存储到checkpoint中>

3）**基于Kafka Direct API的方式，因Spark集成Kafka API直接管理offset，同时依托于Kafka自身特性，实现了Exactly-Once一致性语义。因此在生产中建议使用此种方式。**

2、Spark消费端

park的基本数据单元是一种被称作是RDD(分布式弹性数据集)的数据结构，Spark内部程序通过对RDD的进行一系列的transform和action操作，完成数据的分析处理。

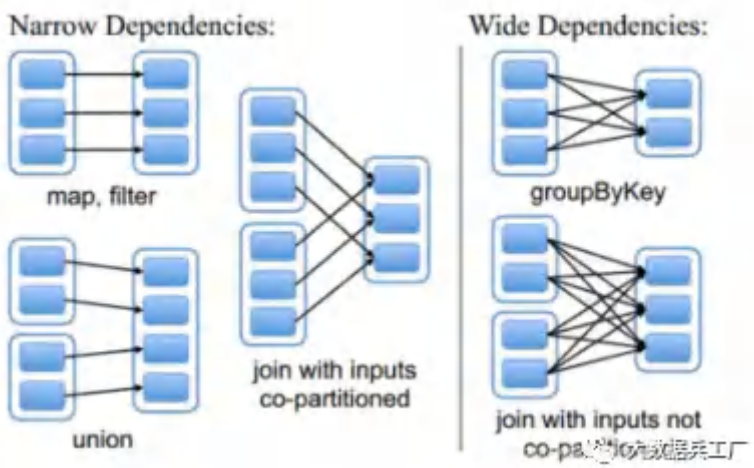
基于RDD内存模型，启用多种一致性策略，实现Exactly-Once一致性。

1）RDD特性

RDD是**分布式、容错、不可变**的数据集。其本身是只读的，不存储真实的数据，当结构更新或者丢失时可对RDD进行重建，RDD不会发生变化。

每个RDD都会有自己的Dependency RDD，即**RDD的血脉机制**。在开启 Checkpoint机制下，可以将RDD依赖保存到HDFS中。当RDD丢失或者程序出现问题，可以快速从血缘关系中恢复。因为记录了RDD的所有依赖过程，**通过血脉追溯可重构计算过程且保证多次计算结果相同。**

公众号：旧时光大数据



2） Checkpoint持久化机制 + WAL机制

Spark的Checkpoint机制会在当前job执行完成后，再重新启动一个job，将程序中需要Checkpoint的RDD标记为MarkedForCheckpoint RDD, 且重新执行一遍RDD前面的依赖，完成后将结果保存到checkpoint中，并删除原先Dependency RDD依赖的血缘关系。同时可以将此次Checkpoint结果持久化到缓存中，便于后期快速恢复。利用**Checkpoint的特性和高可用存储，保证RDD数据结果不丢失。**

启用WAL预写文件机制。如果存在Driver或者Executor异常挂掉的场景，RDD结果或者jobs信息就会丢失，因此很有必要将此类信息持久化到WAL预写日志中，通过对元数据和中间数据存储备份，**WAL机制可以防止数据丢失且提供数据恢复功能。**

3）程序代码去重

如果实时流进入到Spark消费端已经存在重复数据，可以编写**Spark程序代码进行去重**操作，实现Exactly-Once一致性。

- 内存去重。采用Hashset等数据结构，读取数据中类似主键等唯一性标识字段，在内存中存储并进行去重判断。
- 使用Redis Key去重。借助Redis的Hset等特殊数据类型，自动完成Key去重。
- DataFrame/SQL场景，使用group by/ over() window开窗等SQL函数去重
- 利用groupByKey等聚合算子去重
- 其他方法。

3、输出端

输出端保持Exactly-Once一致性，其输出源需要满足一定条件：

支持幂等写入、事务写入机制

1）幂等写入

定义：“**幂等是一个数学与计算机学概念，常见于抽象代数**”中。在编程中一个幂等操作的特点是其任意多次执行所产生的影响均与一次执行的影响相同。

结合语义可知，**幂等写入就是多次写入会产生相同的结果，结果具有不可变性。**在Spark中saveAsTextFile算子就是一种比较典型的幂等写入，也经常被用来作为数据的输出源。

此类型的写入方式，如果在消息中包含唯一主键，那么即使源头存在多条重复数据，在主键约束条件下也不会重复写入，从而实现Exactly-Once语义。

2）事务写入

公众号：旧时光大数据

大家对事务的概念都不陌生，在一个处理过程中的所有操作均需要满足一致性，即要不都发生，要不都不发生，常见于业务性、安全性要求比较高的场景，例如银行卡账户金额存取行为等，具有**原子性、一致性、隔离性、持久性**等四大特征。Spark读取Kafka数据需满足输出端的事务写入，则一般需生成一个唯一ID(可由批次号、时间、分区、offset等组合)，之后将该ID结合计算结果在同一个事务中写入目标源，提交和写入操作保持原子性，实现输出端的Exactly-Once语义。

Spark的RDD和partition的联系

问过的一些公司：触宝(2021.07)

参考答案：

RDD主要由Dependency、Partition、Partitioner组成，Partition是其中之一。

一份待处理的原始数据会被按照相应的逻辑（例如jdbc和hdfs的split逻辑）切分成n份，每份数据对应到RDD中的一个Partition，Partition的数量决定了task的数量，影响着程序的并行度。

Partition的源码如下：

```
trait Partition extends Serializable {
  def index: Int
  override def hashCode(): Int = index
  override def equals(other: Any): Boolean = super.equals(other)
}


```

Partition和RDD是伴生的，即每一种RDD都有其对应的Partition实现，分析Partition主要是分析其子类，两个常用的子类：JdbcPartition和HadoopPartition。

1、决定Partition数量的因素

Partition数量可以在初始化RDD时指定（如JdbcPartition例子），不指定的话，则读取spark.default.parallelism配置，不同类型资源管理器取值不同。

2、Partition数量的影响

Partition数量太少：资源不能充分利用，例如local模式下，有16core，但是Partition数量仅为8的话，有一半的core没利用到。

Partition数量太多：资源利用没问题，但是导致task过多，task的序列化和传输的时间开销增大。

3、Partition调整

1）repartition

reparation是coalesce(numPartitions, shuffle = true)，repartition不仅会调整Partition数，也会将Partitioner修改为hashPartitioner，产生shuffle操作。

2）coalesce

coalesce函数可以控制是否shuffle，但当shuffle为false时，只能减小Partition数，无法增大。

Spark 3.0特性

可回答：1）迁移Spark3有什么收益；2）为什么要乔伊到Spark 3

问过的一些公司：字节(2021.08)，百度(2021.10)，腾讯(2021.10)

参考答案：

Spark 3.0增加的新特性包括**动态分区修剪**（Dynamic Partition Pruning）、**自适应查询执行**（Adaptive Query Execution）、**加速器感知调度**（Accelerator-aware Scheduling）、**支持 Catalog 的数据源 API**（Data Source API with Catalog Supports）、**SparkR 中的向量化**（Vectorization in SparkR）、**支持 Hadoop 3/JDK 11/Scala 2.12** 等等。

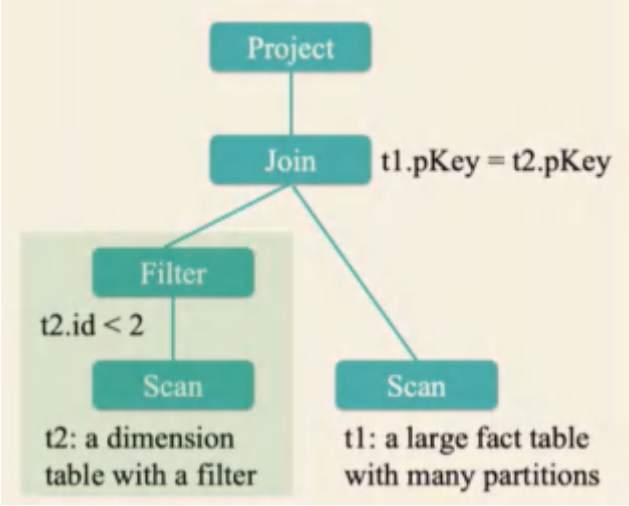
1、动态分区裁剪（Dynamic Partition Pruning）

在3.0以前，spark是不支持动态分区的，**所谓动态分区就是针对分区表中多个表进行join的时候基于运行时（runtime）推断出来的信息，在on后面的条件语句满足一定的要求后就会进行自动动态分区裁减优化。**

比如：

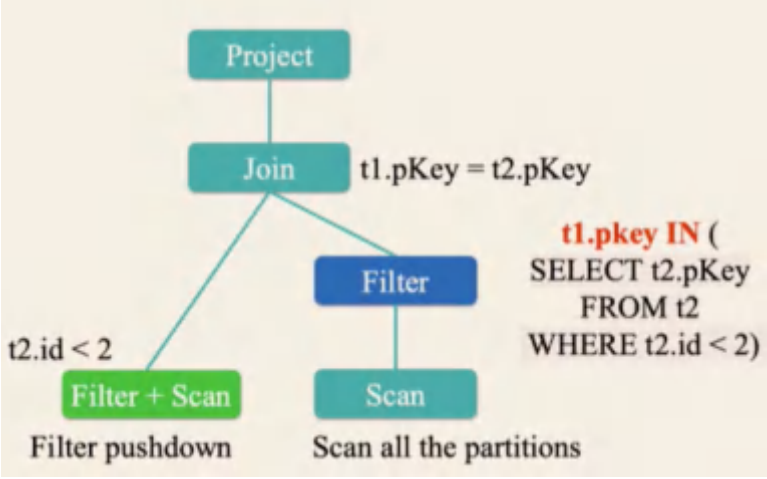
```
SELECT t1.id,t2.pKey
FROM t1
JOIN t2
ON t1.pKey = t2.pKey
AND t2.id < 2;

```

上面这条SQL语句在没有使用动态分区的情况下执行过程如下图所示：

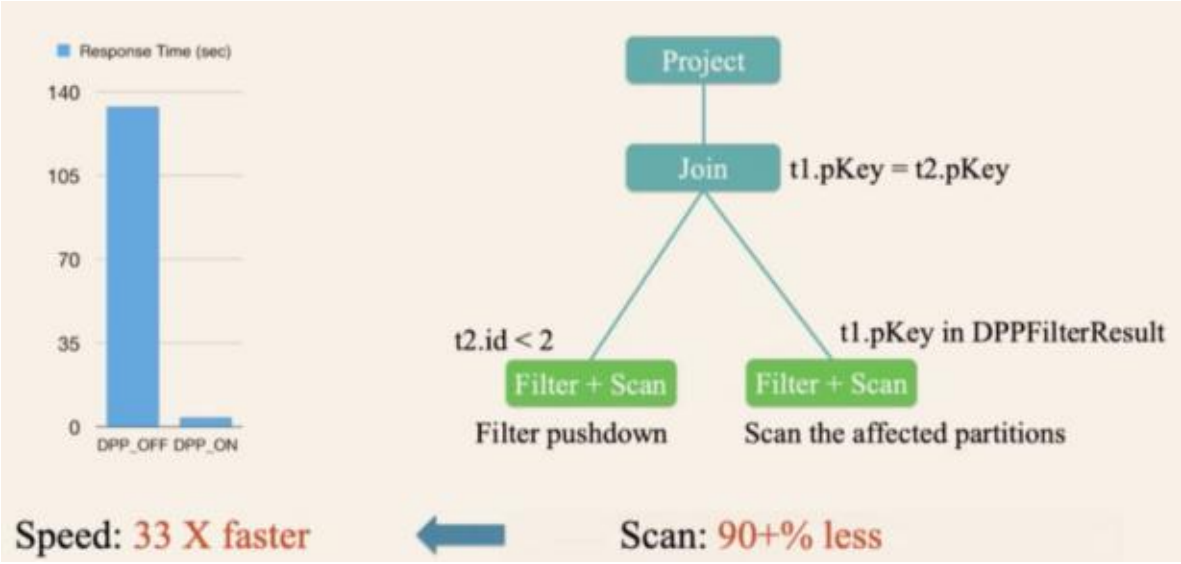


由于之前版本的 Spark 无法进行动态计算代价，所以可能会导致t1表扫描出大量无效的数据。

在使用了动态分区以后执行过程变成了下图：



可以看到之前进行join的时候对t1表中满足on条件的所有数据进行扫描，然后再把两张表进行join，在加入动态分区之后，我们在运行的时候过滤掉了t1表中的无用数据，经过这个优化，查询扫描的数据大大减少，也减少了join时的内存计算开销，在实际环境中同样的代码，性能提升了有33倍！



2、自适应查询执行（Adaptive Query Execution）

自适应查询是指对执行计划按照实际数据分布和组织情况，评估其执行所消耗的时间和资源，从而选择代价最小的计划去执行。一般数据库的优化器有两种，一种是基于规则的优化器(RBO)，一种是基于代价的优化器(CBO),自适应查询指的就是对CBO的优化，Spark SQL的运行流程主要有：

1）将SQL语句通过词法和语法解析生成未绑定的逻辑执行计划；

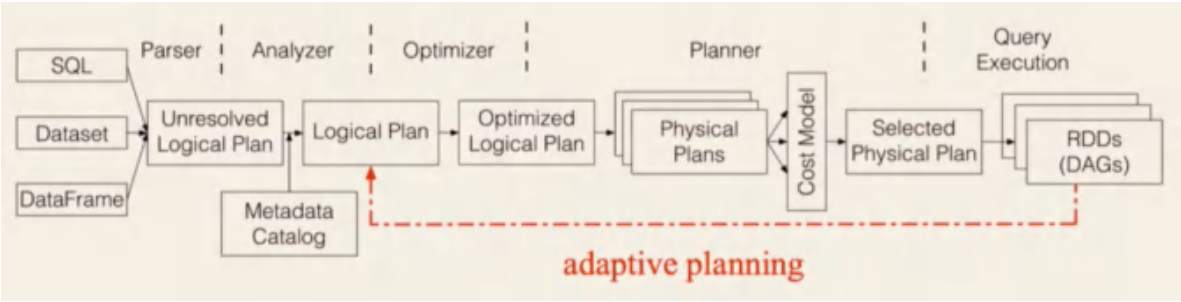
2）分析器配合元数据使用分析规则，完善未绑定的逻辑计划属性转换成绑定的逻辑计划；

3）优化器使用优化规则将绑定的逻辑计划进行合并、裁减、下推生成优化的逻辑计划；

4）规划器使用规划策略把优化的逻辑计划转换成可执行的物理计划；

5）进行preparations规则处理，构建RDD的DAG图执行查询计划。

Spark以前的调度规则是执行计划一旦确定，即使发现后续执行计划可以优化，也不可更改，而自适应查询功能则是在执行查询计划的同时，基于表和列的统计信息，对各个算子产生的中间结果集大小进行估算，根据估算结果来动态地选择最优执行计划，如下图：



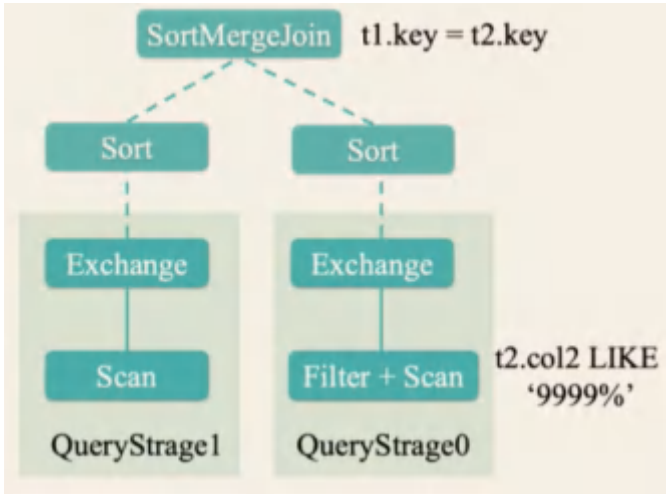
上图是一个经典的Spark的流程，从Parser、Analyzer、Optimizer、Planner到Query的执行。该版本中，AQE指的是图中的红线部分。当某种condition满足的情况下可以进行动态自适应规划。

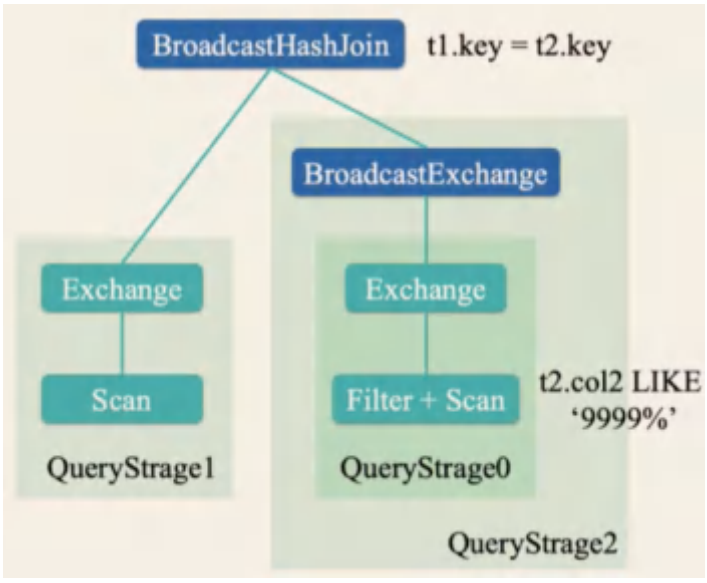
下面举一个简单的例子，执行的是两个表之间的join查询。包含一个key和Filter，如t2.col2 LIKE '9999%'。

在基于cost的模型中是不可能准确的知道Filter能排除多少行的，这种情况下Spark通过谓词下推，将各个条件先应用到对应的数据上，而不是根据写入的顺序执行，就可以先过滤掉部分数据，降低join等一系列操作的数据量级。

在没有动态实时运行信息的时候，保守估计判断只能用SortMergeJoin。

当收集到运行时信息后会发现某个Filter事实上已经去掉了表中绝大多数行，完全可以采用BroadcastHashJoin，如果上层parent也有这种情况，就可以大大提升查询效率。





3、加速器感知调度（Accelerator-aware Scheduling）

如今大数据和机器学习已经有了很大的结合，在机器学习里面，因为计算迭代的时间可能会很长，开发人员一般会选择使用 GPU、FPGA 或 TPU 来加速计算。

在 Apache Hadoop 3.1 版本里面已经开始内置原生支持 GPU 和 FPGA 了。作为通用计算引擎的 Spark 肯定也不甘落后，来自 Databricks、NVIDIA、Google 以及阿里巴巴的工程师们正在为 Apache Spark 添加原生的 GPU 调度支持，该方案填补了 Spark 在 GPU 资源的任务调度方面的空白，有机地融合了大数据处理和 AI 应用，扩展了 Spark 在深度学习、信号处理和各大数据应用的应用场景。

为了让 Spark 也支持 GPUs，在技术层面上需要做出两个主要改变：

1）**在 cluster manager 层面上**，需要升级 cluster managers 来支持 GPU。并且给用户提供相关 API，使得用户可以控制 GPU 资源的使用和分配。

2）在 Spark 内部，需要**在 scheduler 层面**做出修改，使得 scheduler 可以在用户 task 请求中识别 GPU 的需求，然后根据 executor 上的 GPU 供给来完成分配。

因为让 Apache Spark 支持 GPU 是一个比较大的特性，所以项目分为了几个阶段。

1）在 Apache Spark 3.0 版本，将支持在 standalone、 YARN 以及 Kubernetes 资源管理器下支持 GPU，并且对现有正常的作业基本没影响。

2）对于 TPU 的支持、Mesos 资源管理器中 GPU 的支持、以及 Windows 平台的 GPU 支持将不是这个版本的目标。

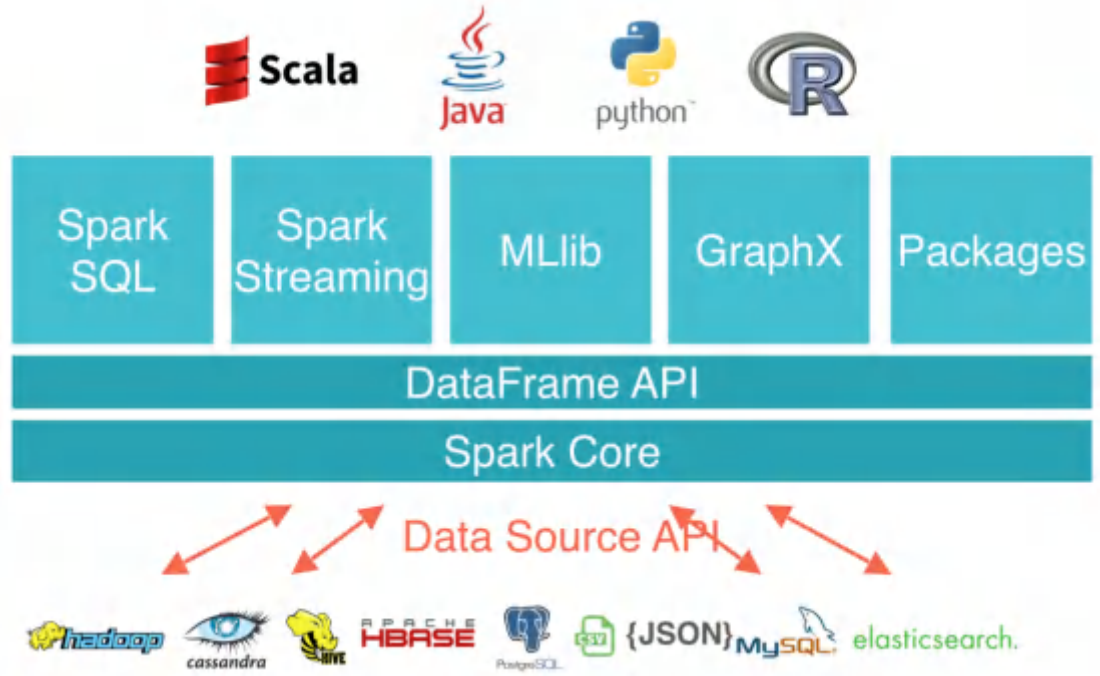
3）而且对于一张 GPU 卡内的细粒度调度也不会在这个版本支持；Spark 3.0 版本将把一张 GPU 卡和其内存作为不可分割的单元。

公众号：旧时光大数据

4、更好的API扩展（BetterAPI-Extensions-DataSourceV2）

Data Source API 定义如何从存储系统进行读写的相关 API 接口，比如 Hadoop 的 InputFormat/OutputFormat，Hive 的 Serde 等。这些 API 非常适合用户在 Spark 中使用 RDD 编程的时候使用。使用这些 API 进行编程虽然能够解决我们的问题，但是对用户来说使用成本还是挺高的，而且 Spark 也不能对其进行优化。

为了解决这些问题，Spark 1.3 版本开始引入了 Data Source API V1，通过这个 API 我们可以很方便的读取各种来源的数据，而且 Spark 使用 SQL 组件的一些优化引擎对数据源的读取进行优化，比如列裁剪、过滤下推等等。



Data Source API V1 为我们抽象了一系列的接口，使用这些接口可以实现大部分的场景。但是随着使用的用户增多，逐渐显现出一些问题：

1）部分接口还是太过于依赖SQLContext和DataFrame

2）缺乏对列式数据库存储的读取支持

3）写操作不支持事务

4）不支持流式处理，不能进行增量跌打

5）缺乏分区和排序信息

6）扩展能力有限，难以下推其它算子

为了解决 Data Source V1 的一些问题，从 Apache Spark 2.3.0 版本开始，社区引入了 Data Source API V2，在保留原有的功能之外，还解决了 Data Source API V1 存在的一些问题，比如不再依赖上层 API，扩展能力增强。虽然这个功能在 Apache Spark 2.x 版本就出现了，但是不是很稳定，所以社区对 Spark DataSource API V2 的稳定性工作以及新功能分别开了两个 ISSUE：SPARK-25186 以及 SPARK-22386。

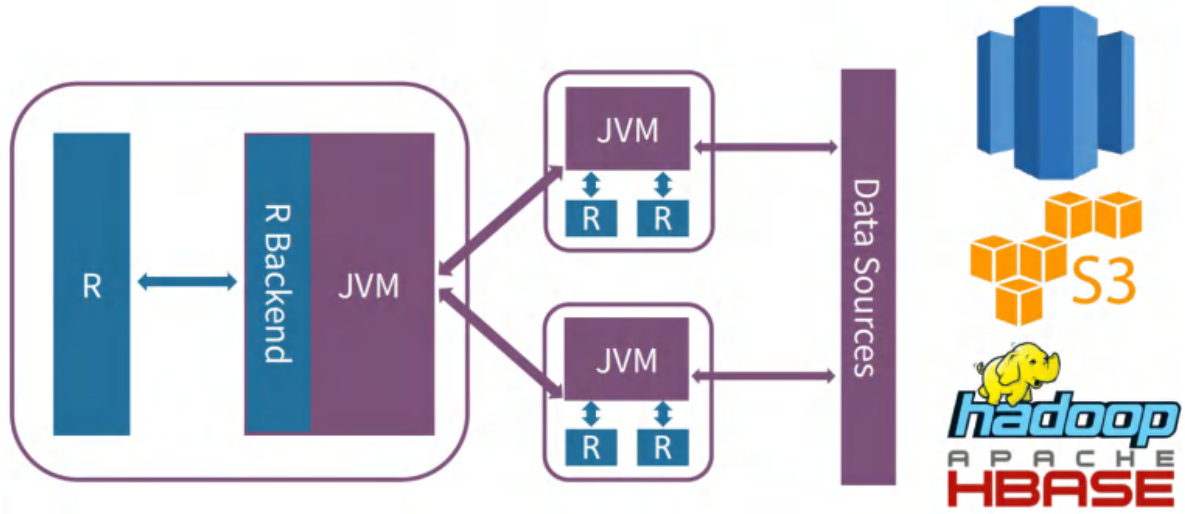
Spark DataSource API V2 最终稳定版以及新功能将会随着年底和 Apache Spark 3.0.0 版本一起发布，其也算是Spark 3.0版本的一大新功能。

5、更好的ANSI-SQL兼容（ANSI SQL Compatible）

因为Spark原来的SQL语法和函数跟ANSI标准还是存在一些差异，因此这次版本更新将缩小和ANSI标准之间的差异，包括增加一些ANSI SQL的函数、区分SQL保留关键字以及内置函数等。

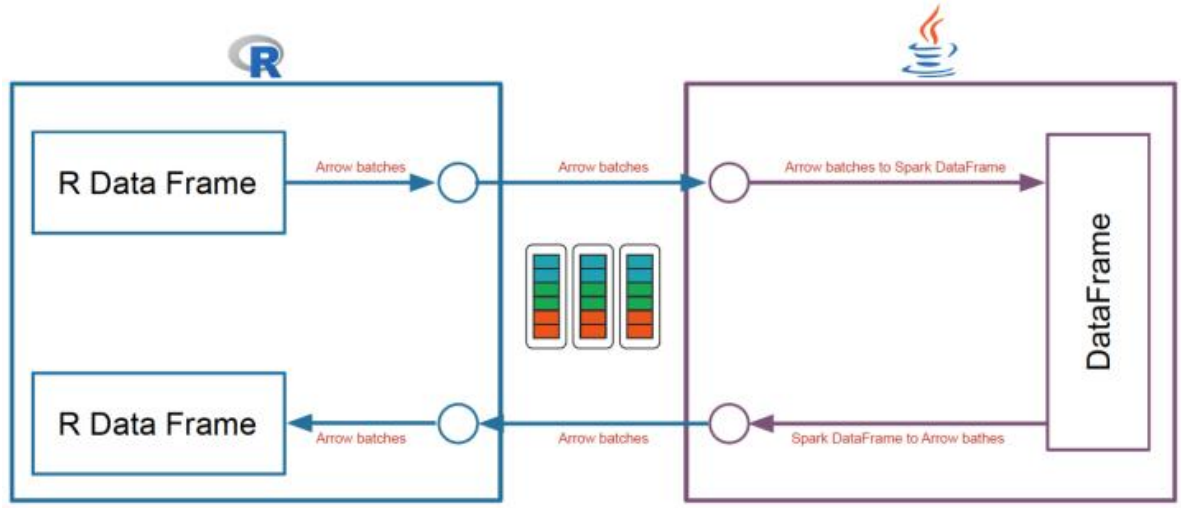
6、SparkR 中的向量化（Vectorization in SparkR）

Spark 是从 1.4 版本开始支持 R 语言的，但是那时候 Spark 和 R 进行交互的架构图如下：



每当使用 R 语言和 Spark 集群进行交互，需要经过JVM ，这也就无法避免数据的序列化和反序列化操作，这在数据量很大的情况下性能是十分低下的！而Spark 已经在许多操作中进行了向量化优化（vectorization optimization），例如，内部列式格式（columnar format）、Parquet/ORC 向量化读取、Pandas UDFs 等。向量化可以大大提高性能。

SparkR 向量化允许用户按原样使用现有代码，但是当他们执行 R 本地函数或将 Spark DataFrame 与 R DataFrame 互相转换时，可以将性能提高大约数千倍。这项工作可以看下 Spark-26759。新的架构如下:



可以看出，SparkR 向量化是利用 Apache Arrow，其使得系统之间数据的交互变得很高效，而且避免了数据的序列化和反序列化的消耗，所以采用了这个之后，SparkR 和 Spark 交互的性能得到极大提升。

Spark计算的灵活性体现在哪里

问过的一些公司：阿里蚂蚁(2021.08)

参考答案：

Spark提供了不同层面的灵活性。

在**实现层**，它完美演绎了Scala trait动态混入（mixin）策略（如可更换的集群调度器、序列化库）。

在**原语（Primitive）层**，它允许扩展新的数据算子（operator）、新的数据源（如HDFS之外支持DynamoDB）、新的language bindings（Java和Python）。

在**范式（Paradigm）层**，Spark支持内存计算、多迭代批量处理、即席查询、流处理和图计算等多种范式。

公众号：旧时光大数据

Flink面试题

Flink架构

问过的一些公司：美团，字节

参考答案：

1、Flink运行时的架构组件

Flink运行时架构主要包括四个不同的组件，它们会在运行流处理应用程序时协同工作：作业管理器（JobManager）、资源管理器（ResourceManager）、任务管理器（TaskManager），以及分发器（Dispatcher）。因为Flink是用Java和Scala实现的，所以所有组件都会运行在Java虚拟机上。每个组件的职责如下：

1）作业管理器（JobManager）

控制一个应用程序执行的主进程，也就是说，每个应用程序都会被一个不同的JobManager所控制执行。JobManager会先接收到要执行的应用程序，这个应用程序会包括：作业图（JobGraph）、逻辑数据流图（logical dataflow graph）和打包了所有的类、库和其它资源的JAR包。JobManager会把JobGraph转换成一个物理层面的数据流图，这个图被叫做“执行图”（ExecutionGraph），包含了所有可以并发执行的任务。JobManager会向资源管理器（ResourceManager）请求执行任务必要的资源，也就是任务管理器（TaskManager）上的插槽（slot）。一旦它获取到了足够的资源，就会将执行图分发到真正运行它们的TaskManager上。而在运行过程中，JobManager会负责所有需要中央协调的操作，比如说检查点（checkpoints）的协调。

2）资源管理器（ResourceManager）

主要负责管理任务管理器（TaskManager）的插槽（slot），TaskManger插槽是Flink中定义的处理资源单元。Flink为不同的环境和资源管理工具提供了不同资源管理器，比如YARN、Mesos、K8s，以及standalone部署。当JobManager申请插槽资源时，ResourceManager会将有空闲插槽的TaskManager分配给JobManager。如果ResourceManager没有足够的插槽来满足JobManager的请求，它还可以向资源提供平台发起会话，以提供启动TaskManager进程的容器。另外，ResourceManager还负责终止空闲的TaskManager，释放计算资源。

3）任务管理器（TaskManager）

Flink中的工作进程。通常在Flink中会有多个TaskManager运行，每一个TaskManager都包含了一定数量的插槽（slots）。插槽的数量限制了TaskManager能够执行的任务数量。启动之后，TaskManager会向资源管理器注册它的插槽；收到资源管理器的指令后，TaskManager就会将一个或者多个插槽提供给JobManager调用。JobManager就可以向插槽分配任务（tasks）来执行了。在执行过程中，一个TaskManager可以跟其它运行同一应用程序的TaskManager交换数据。

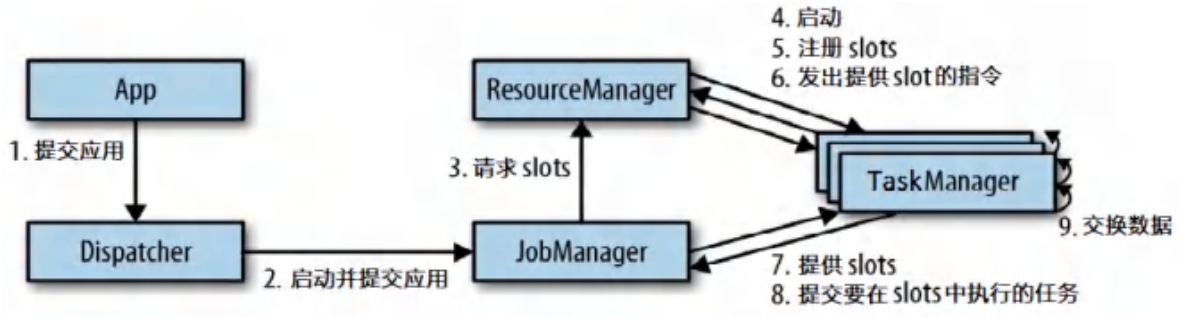
4）分发器（Dispatcher）

可以跨作业运行，它为应用提交提供了REST接口。当一个应用被提交执行时，分发器就会启动并将应用移交给一个JobManager。由于是REST接口，所以Dispatcher可以作为集群的一个HTTP接入点，这样就能够不受防火墙阻挡。Dispatcher也会启动一个Web UI，用来方便地展示和监控作业执行的信息。Dispatcher在架构中可能并不是必需的，这取决于应用提交运行的方式。

2、任务提交流程

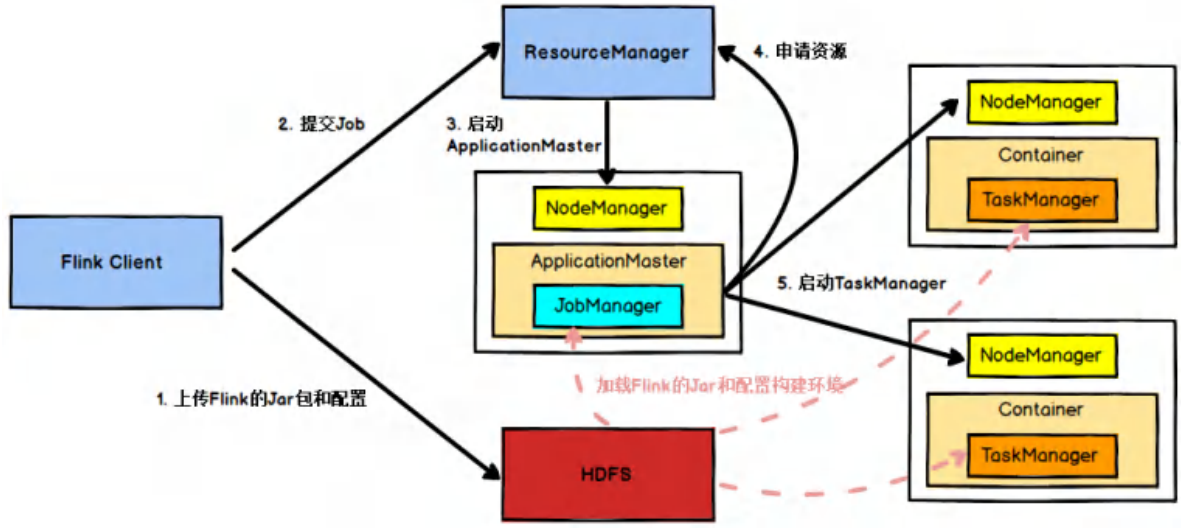
我们来看看当一个应用提交执行时，Flink的各个组件是如何交互协作的：

公众号：旧时光大数据



上图是从一个较为高层级的视角，来看应用中各组件的交互协作。如果部署的集群环境不同（例如YARN，Mesos，Kubernetes，standalone等），其中一些步骤可以被省略，或是有些组件会运行在同一个JVM进程中。

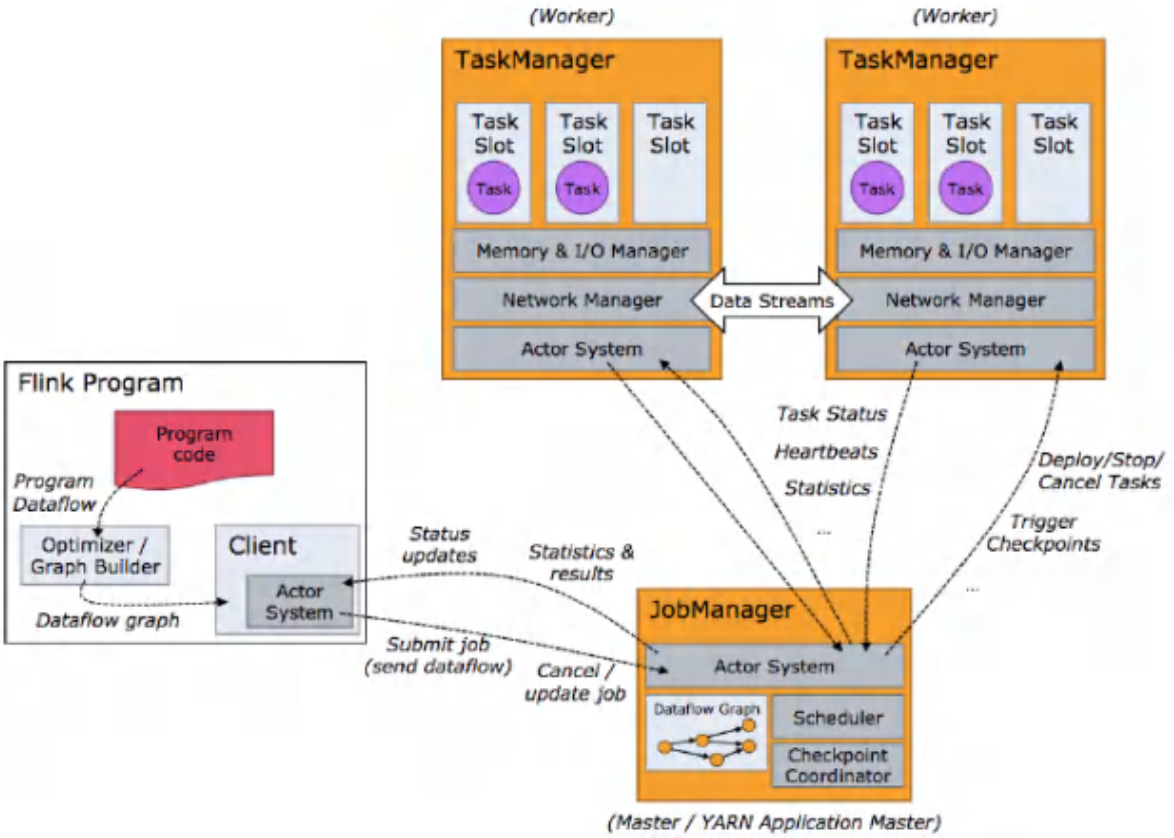
具体地，如果我们将Flink集群部署到YARN上，那么就会有如下的提交流程：



Flink任务提交后，Client向HDFS上传Flink的Jar包和配置，之后向Yarn ResourceManager提交任务，ResourceManager分配Container资源并通知对应的NodeManager启动ApplicationMaster，ApplicationMaster启动后加载Flink的Jar包和配置构建环境，然后启动JobManager，之后ApplicationMaster向ResourceManager申请资源启动TaskManager，ResourceManager分配Container资源后，由ApplicationMaster通知资源所在节点的NodeManager启动TaskManager，NodeManager加载Flink的Jar包和配置构建环境并启动TaskManager，TaskManager启动后向JobManager发送心跳包，并等待JobManager向其分配任务。

3、架构组成（任务调度原理）

公众号：旧时光大数据



客户端不是运行时和程序执行的一部分，但它用于准备并发送dataflow(JobGraph)给Master(JobManager)，然后，客户端断开连接或者维持连接以等待接收计算结果。

当 Flink 集群启动后，首先会启动一个 JobManger 和一个或多个的 TaskManager。由 Client 提交任务给 JobManager，JobManager 再调度任务到各个 TaskManager 去执行，然后 TaskManager 将心跳和统计信息汇报给 JobManager。TaskManager 之间以流的形式进行数据的传输。上述三者均为独立的 JVM 进程。

Client：为提交 Job 的客户端，可以是运行在任何机器上（与 JobManager 环境连通即可）。提交 Job 后，Client 可以结束进程（Streaming的任务），也可以不结束并等待结果返回。

JobManager：主要负责调度 Job 并协调 Task 做 checkpoint，职责上很像 Storm 的 Nimbus。从 Client 处接收到 Job 和 JAR 包等资源后，会生成优化后的执行计划，并以 Task 的单元调度到各个 TaskManager 去执行。

TaskManager：在启动的时候就设置好了槽位数（Slot），每个 slot 能启动一个 Task，Task 为线程。从 JobManager 处接收需要部署的 Task，部署启动后，与自己的上游建立 Netty 连接，接收数据并处理。

Flink的窗口了解哪些，都有什么区别，有哪几种？如何定义？

可回答：1）Flink窗口机制（后面还有另一种参考，差不多）；2）Flink的时间窗口中，简介下会话窗口

问过的一些公司：安恒信息，嘉云，字节(2021.07)

参考答案：

1、Window概述

streaming流式计算是一种被设计用于处理无限数据集的数据处理引擎，而无限数据集是指一种不断增长的本质上无限的数据集，而window是一种**切割无限数据为有限块进行处理**的手段。

Window是无限数据流处理的核心，Window将一个无限的stream拆分成有限大小的“buckets”桶，我们可以在这些桶上做计算操作。

公众号：旧时光大数据

2、Window类型

Window可以分成两类：

- CountWindow：按照指定的数据条数生成一个Window，与时间无关。
- TimeWindow：按照时间生成Window。

对于TimeWindow，可以根据窗口实现原理的不同分成三类：滚动窗口（Tumbling Window）、滑动窗口（Sliding Window）和会话窗口（Session Window）。

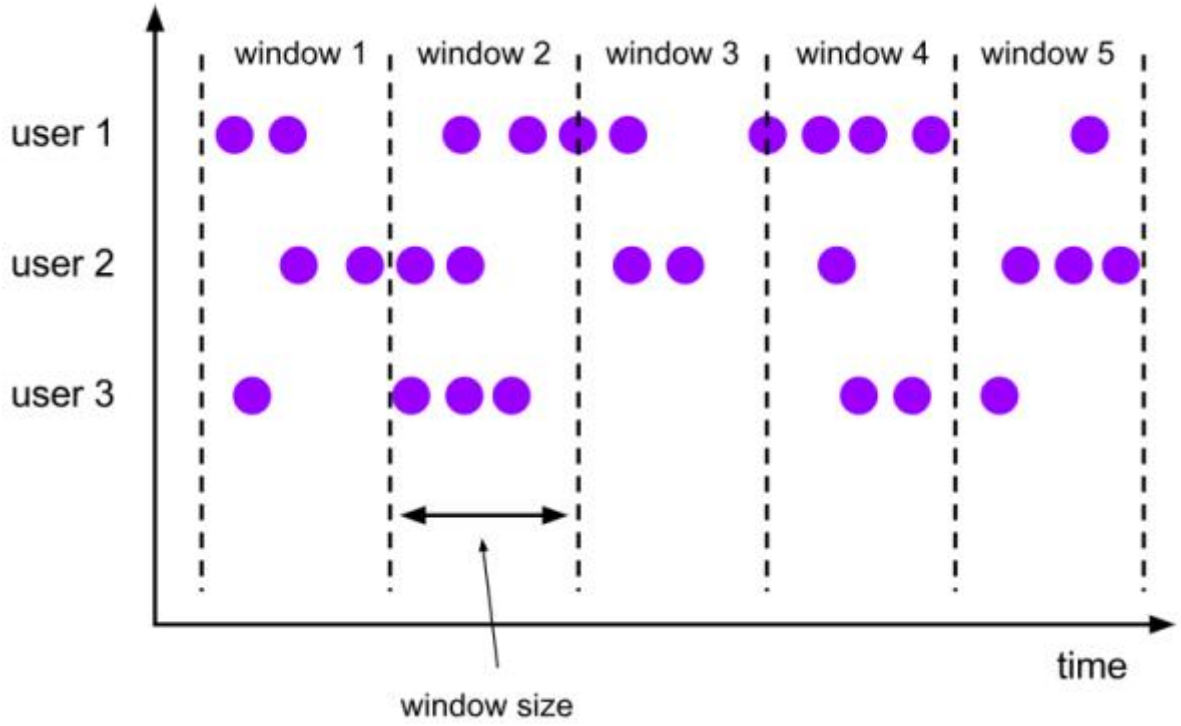
1）滚动窗口（Tumbling Windows）

将数据依据固定的窗口长度对数据进行切片。

特点：时间对齐，窗口长度固定，没有重叠。

滚动窗口分配器将每个元素分配到一个指定窗口大小的窗口中，滚动窗口有一个固定的大小，并且不会出现重叠。

例如：如果你指定了一个5分钟大小的滚动窗口，窗口的创建如下图所示：



适用场景：适合做BI统计等（做每个时间段的聚合计算）。

2）滑动窗口（Sliding Windows）

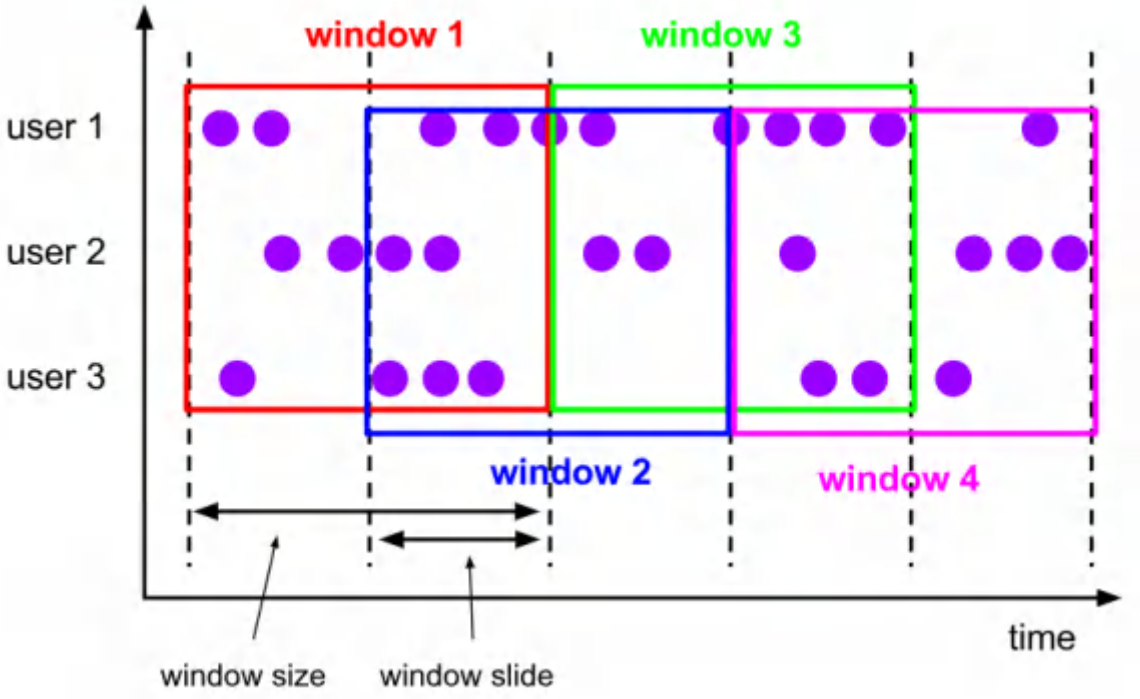
滑动窗口是固定窗口的更广义的一种形式，滑动窗口由固定的窗口长度和滑动间隔组成。

特点：时间对齐，窗口长度固定，可以有重叠。

滑动窗口分配器将元素分配到固定长度的窗口中，与滚动窗口类似，窗口的大小由窗口大小参数来配置，另一个窗口滑动参数控制滑动窗口开始的频率。因此，滑动窗口如果滑动参数小于窗口大小的话，窗口是可以重叠的，在这种情况下元素会被分配到多个窗口中。

例如，你有10分钟的窗口和5分钟的滑动，那么每个窗口中5分钟的窗口里包含着上个10分钟产生的数据，如下图所示：

公众号：旧时光大数据



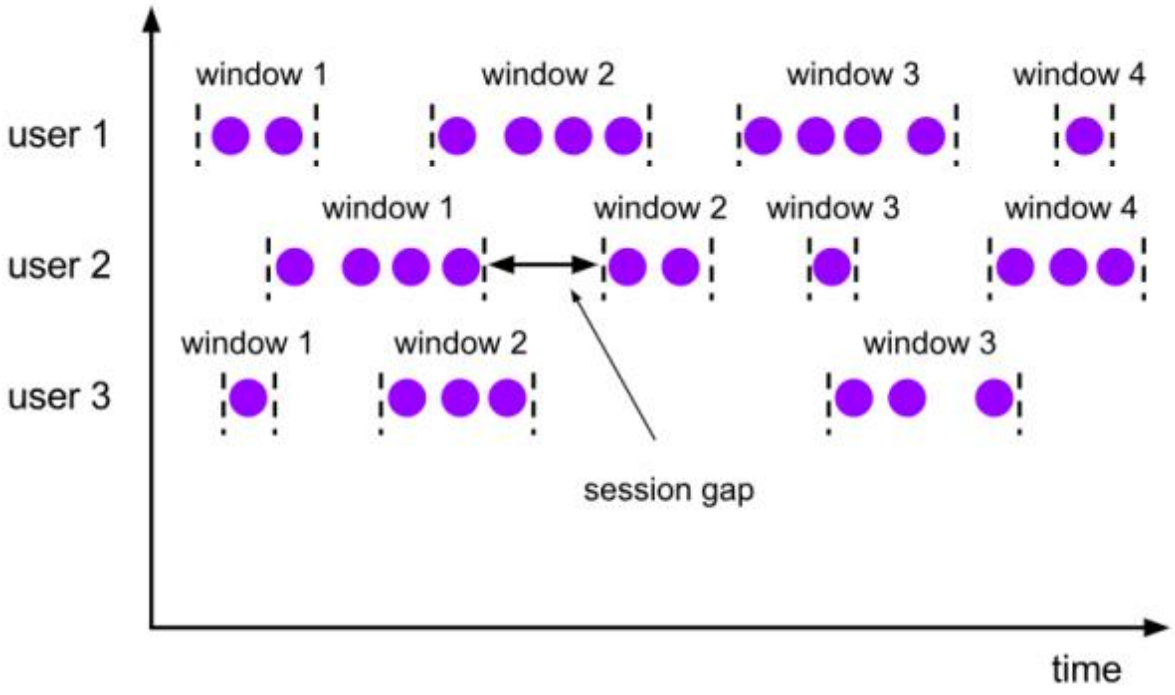
适用场景：对最近一个时间段内的统计（求某接口最近5min的失败率来决定是否要报警）。

3）会话窗口（Session Window）

由一系列事件组合一个指定时间长度的timeout间隙组成，类似于web应用的session，也就是一段时间没有接收到新数据就会生成新的窗口。

特点：时间无对齐。

session窗口分配器通过session活动来对元素进行分组，session窗口跟滚动窗口和滑动窗口相比，不会有重叠和固定的开始时间和结束时间的情况，相反，当它在一个固定的时间周期内不再收到元素，即非活动间隔产生，那个这个窗口就会关闭。一个session窗口通过一个session间隔来配置，这个session间隔定义了非活跃周期的长度，当这个非活跃周期产生，那么当前的session将关闭并且后续的元素将被分配到新的session窗口中去。



3、Window API

1）TimeWindow

公众号：旧时光大数据

TimeWindow是将指定时间范围内的所有数据组成一个window，一次对一个window里面的所有数据进行计算。

（1）滚动窗口

Flink 默认的时间窗口根据 Processing Time 进行窗口的划分，将 Flink 获取到的数据根据进入 Flink 的时间划分到不同的窗口中。

```
val minTempPerWindow = dataStream
      .map(r => (r.id, r.temperature))
      .keyBy(_._1)
      .timeWindow(Time.seconds(15))
      .reduce((r1, r2) => (r1._1, r1._2.min(r2._2)))
```

时间间隔可以通过 Time.milliseconds(x)，Time.seconds(x)，Time.minutes(x)等其中的一个来指定。

（2）滑动窗口

滑动窗口和滚动窗口的函数名是完全一致的，只是在传参数时需要传入两个参数，一个是 window_size，一个是 sliding_size。

下面代码中的 sliding_size 设置为了 5s，也就是说，窗口每 5s 就计算一次，每一次计算的 window 范围是 15s 内的所有元素。

```
val minTempPerWindow: DataStream[(String, Double)] = dataStream
      .map(r => (r.id, r.temperature))
      .keyBy(_._1)
      .timeWindow(Time.seconds(15), Time.seconds(5))
      .reduce((r1, r2) => (r1._1, r1._2.min(r2._2)))
     .window(EventTimeSessionWindows.withGap(Time.minutes(10))
```

时间间隔可以通过 Time.milliseconds(x)，Time.seconds(x)，Time.minutes(x)等其中的一个来指定。

2）CountWindow

CountWindow根据窗口中相同 key 元素的数量来触发执行，执行时只计算元素数量达到窗口大小的 key 对应的结果。

注意：CountWindow 的 window_size 指的是相同 Key 的元素的个数，不是输入的所有元素的总数。

（1）滚动窗口

默认的 CountWindow 是一个滚动窗口，只需要指定窗口大小即可，当元素数量达到窗口大小时，就会触发窗口的执行。

```
val minTempPerWindow: DataStream[(String, Double)] = dataStream
  .map(r => (r.id, r.temperature))
  .keyBy(_._1)
  .countWindow(5)
   .reduce((r1, r2) => (r1._1, r1._2.max(r2._2)))
```

（2）滑动窗口

滑动窗口和滚动窗口的函数名是完全一致的，只是在传参数时需要传入两个参数，一个是 window_size，一个是 sliding_size。

下面代码中的 sliding_size 设置为了 2，也就是说，每收到两个相同 key 的数据就计算一次，每一次计算的 window 范围是 5 个元素。

公众号：旧时光大数据

```
val keyedStream: KeyedStream[(String, Int), Tuple] = dataStream.map(r => (r.id, r.temperature)).keyBy(0)

//每当某一个 key 的个数达到 2 的时候,触发计算，计算最近该 key 最近 10 个元素的内容
val windowedStream: WindowedStream[(String, Int), Tuple, GlobalWindow] = keyedStream.countWindow(10,2)

val sumDstream: DataStream[(String, Int)] = windowedStream.sum(1)
```

4、窗口函数

window function 定义了要对窗口中收集的数据做的计算操作，主要可以分为两类：

增量聚合函数（incremental aggregation functions）

- 每条数据到来就进行计算，保持一个简单的状态。典型的增量聚合函数有ReduceFunction，AggregateFunction。

全窗口函数（full window functions）

- 先把窗口所有数据收集起来，等到计算的时候会遍历所有数据。ProcessWindowFunction 就是一个全窗口函数。

5、其它可选API

trigger() —— 触发器

- 定义 window 什么时候关闭，触发计算并输出结果

evitor() —— 移除器

- 定义移除某些数据的逻辑

allowedLateness() —— 允许处理迟到的数据

sideOutputLateData() —— 将迟到的数据放入侧输出流

getSideOutput() —— 获取侧输出流

Flink窗口函数，时间语义相关的问题

可回答：Flink窗口函数介绍下

问过的一些公司：网易，美团，网易严选(2021.09)

参考答案：

1、窗口函数（window function）

window function 定义了要对窗口中收集的数据做的计算操作，主要可以分为两类：

增量聚合函数（incremental aggregation functions）

- 每条数据到来就进行计算，保持一个简单的状态。典型的增量聚合函数有ReduceFunction，AggregateFunction。

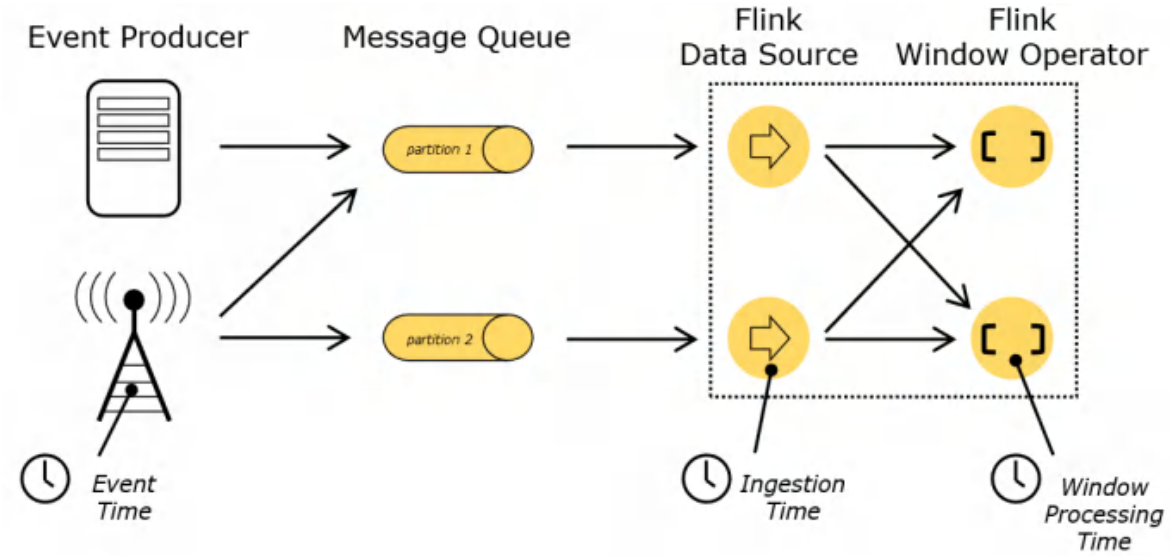
全窗口函数（full window functions）

- 先把窗口所有数据收集起来，等到计算的时候会遍历所有数据。ProcessWindowFunction 就是一个全窗口函数。

2、时间语义

公众号：旧时光大数据

在Flink的流式处理中，会涉及到时间的不同概念，如下图所示：

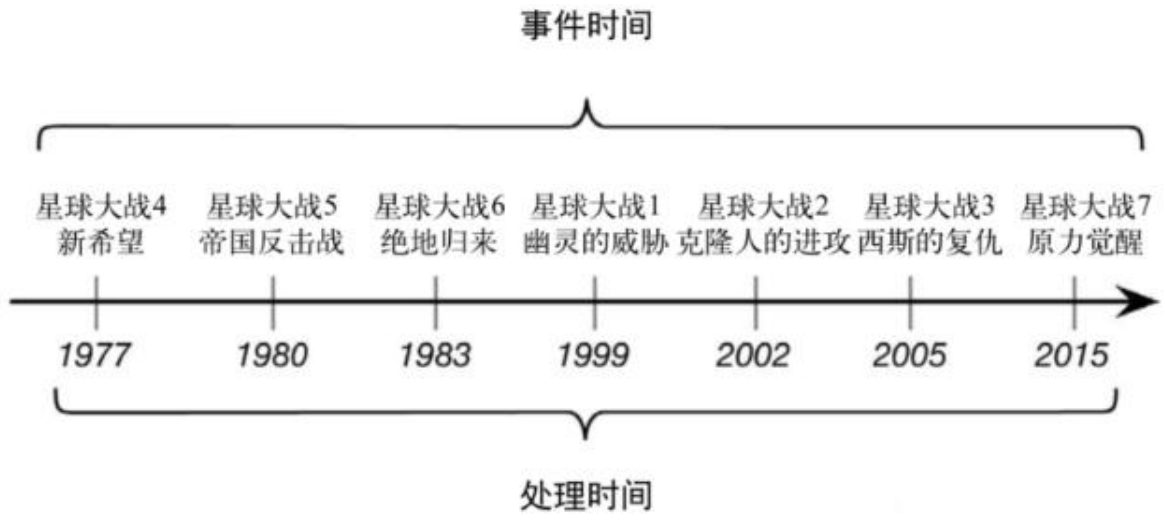


Event Time：是事件创建的时间。它通常由事件中的时间戳描述，例如采集的日志数据中，每一条日志都会记录自己的生成时间，Flink通过时间戳分配器访问事件时间戳。

Ingestion Time：是数据进入Flink的时间。

Processing Time：是每一个执行基于时间操作的算子的本地系统时间，与机器相关，默认的时间属性就是Processing Time。

一个例子——电影《星球大战》：



例如，一条日志进入Flink的时间为2017-11-12 10:00:00.123，到达Window的系统时间为2017-11-12 10:00:01.234，日志的内容如下：

```
2017-11-02 18:37:15.624 INFO Fail over to rm2
```

对于业务来说，要统计1min内的故障日志个数，哪个时间是最有意义的？—— eventTime，因为我们要根据日志的生成时间进行统计。

1）EventTime的引入

在Flink的流式处理中，绝大部分的业务都会使用eventTime，一般只在eventTime无法使用时，才会被迫使用ProcessingTime或者IngestionTime。

如果要使用EventTime，那么需要引入EventTime的时间属性，引入方式如下所示：

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
// 从调用时刻开始给env创建的每一个stream追加时间特征
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
```

介绍下Flink的watermark（水位线），watermark需要实现哪个实现类，在何处定义？有什么作用？

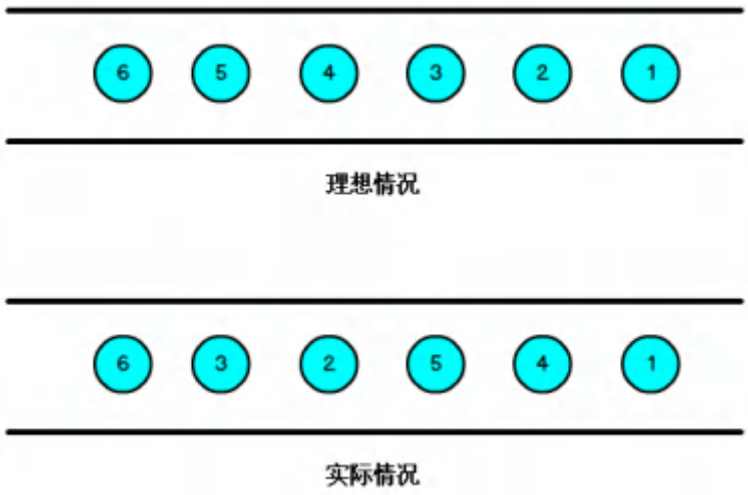
可回答：1）Flink水位线有什么作用；2）Flink的watermark机制

问过的一些公司：字节，蘑菇街，安恒信息，字节社招，一点咨询，触宝(2021.07)，网易严选(2021.09)

参考答案：

1、Watermark介绍及作用

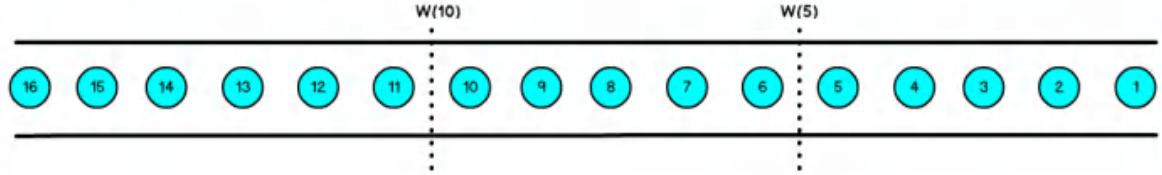
我们知道，流处理从事件产生，到流经source，再到operator，中间是有一个过程和时间的，虽然大部分情况下，流到operator的数据都是按照事件产生的时间顺序来的，但是也不排除由于网络、分布式等原因，导致乱序的产生，所谓乱序，就是指Flink接收到的事件的先后顺序不是严格按照事件的Event Time顺序排列的。



那么此时出现一个问题，一旦出现乱序，如果只根据eventTime决定window的运行，我们不能明确数据是否全部到位，但又不能无限期的等下去，此时必须要有个机制来保证一个特定的时间后，必须触发window去进行计算了，这个特别的机制，就是Watermark。

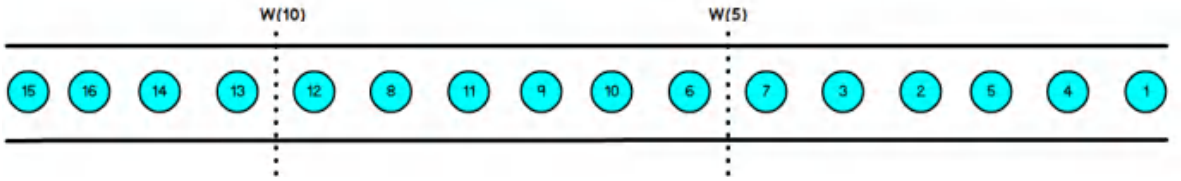
- Watermark是一种衡量Event Time进展的机制。
- **Watermark是用于处理乱序事件的**，而正确的处理乱序事件，通常用Watermark机制结合window来实现。
- 数据流中的Watermark用于表示timestamp小于Watermark的数据，都已经到达了，因此，window的执行也是由Watermark触发的。
- Watermark可以理解成一个延迟触发机制，我们可以设置Watermark的延时时长t，每次系统会校验已经到达的数据中最大的maxEventTime，然后认定eventTime小于maxEventTime - t的所有数据都已经到达，如果有窗口的停止时间等于maxEventTime – t，那么这个窗口被触发执行。

有序流的Watermarker如下图所示：（Watermark设置为0）



公众号：旧时光大数据

乱序流的Watermarker如下图所示：（Watermark设置为2）



当Flink接收到数据时，会按照一定的规则去生成Watermark，这条Watermark就等于当前所有到达数据中的maxEventTime - 延迟时长，也就是说，Watermark是基于数据携带的时间戳生成的，一旦Watermark比当前未触发的窗口的停止时间要晚，那么就会触发相应窗口的执行。由于event time是由数据携带的，因此，如果运行过程中无法获取新的数据，那么没有被触发的窗口将永远都不被触发。

上图中，我们设置的允许最大延迟到达时间为2s，所以时间戳为7s的事件对应的Watermark是5s，时间戳为12s的事件的Watermark是10s，如果我们的窗口1是1s~5s，窗口2是6s~10s，那么时间戳为7s的事件到达时的Watermarker恰好触发窗口1，时间戳为12s的事件到达时的Watermark恰好触发窗口2。

Watermark 就是触发前一窗口的“关窗时间”，一旦触发关门那么以当前时刻为准在窗口范围内的所有所有数据都会收入窗中。

只要没有达到水位那么不管现实中的时间推进了多久都不会触发关窗。

2、Watermark的使用

Watermark的两种生成方式

1）SourceFunction中产生，将Timestamp的分配(也就是上文提到的离散化)和watermark的生成放在上游，同时sourceFunction中也有两个方法生成watermark

通过collectwithTimestamp方法发送数据，和调用emitWatermark产生watermark,我们可以看到，调用collectwithTimestamp需要传入两个参数，第一个参数就是数据，第二次参数就是数据对应的时间戳，这样就完成了timestamp的分配，调用emitWatermark生成watermark。

```
override def run(ctx: SourceContext[MyType]): Unit = {
    while (/* condition */) {
        val next: MyType = getNext()
        ctx.collectWithTimestamp(next, next.eventTimestamp)

        if (next.hasWatermarkTime) {
            ctx.emitWatermark(new Watermark(next.getWatermarkTime))
        }
    }
}
```

2）DataStream API指定，调用assignTimestampsAndWatermarks方法，用于某些sourceFunction不支持的情况，它能够接收不同的timestamp和watermark生成器，说白了就是函数里面参数不同。

定期生成：

```
val resultData = logData.assignTimestampsAndWatermarks(new
AssignerWithPeriodicWatermarks[(Long, String, Long)] {

  val maxOutOfOrderness = 10000L

  var currentMaxTimestamp: Long = _

  override def getCurrentWatermark: Watermark = {
    new Watermark(currentMaxTimestamp - maxOutOfOrderness)
  }

```

公众号：旧时光大数据

```
    // 根据数据本身的 Event time 来获取
    override def extractTimestamp(element: (Long, String, Long),
previousElementTimestamp: Long): Long = {
      val timestamp = element._1
      currentMaxTimestamp = Math.max(timestamp, currentMaxTimestamp)
      timestamp
    }
  })
```

标记生成：

```
class PunctuatedAssigner extends AssignerWithPunctuatedWatermarks[SensorReading]
{

  // 1 min in ms
  val bound: Long = 60 * 1000

  override def checkAndGetNextWatermark(r: SensorReading, extractedTS: Long):
Watermark = {
    if (r.id == "sensor_1") {
      // emit watermark if reading is from sensor_1
      new Watermark(extractedTS - bound)
    } else {
      // do not emit a watermark
      null
    }
  }

  override def extractTimestamp(r: SensorReading, previousTS: Long): Long = {
    // assign record timestamp
    r.timestamp
  }
}
```

区别：定期指的是定时调用逻辑生成watermark，而标记不是根据时间，而是看到特殊记录表示接下来的数据可能发不过来了，分配timestamp 调用用户实现的watermark方法。

建议：越靠近源端处理更容易进行判断。

Flink的窗口（实现）机制

问过的一些公司：蘑菇街，星环科技，一点咨询

参考答案：可以结合前面的题目

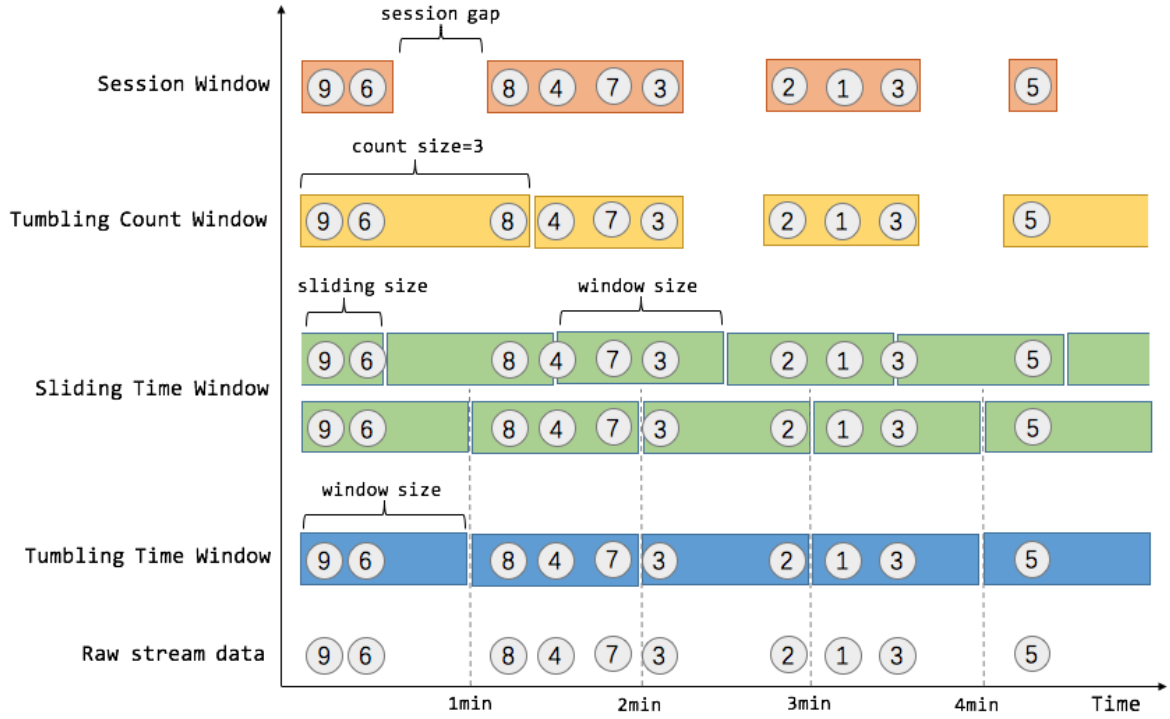
Flink 认为 Batch 是 Streaming 的一个特例，所以 Flink 底层引擎是一个流式引擎，在上面实现了流处理和批处理。而窗口（window）就是从 Streaming 到 Batch 的一个桥梁。Flink 提供了非常完善的窗口机制，这是Flink最大的亮点之一（其他的亮点包括消息乱序处理，和 checkpoint 机制）。

1、什么是Window

在流处理应用中，数据是连续不断的，因此我们不可能等到所有数据都到了才开始处理。当然我们可以每来一个消息就处理一次，但是有时我们需要做一些聚合类的处理，例如：在过去的1分钟内有多少用户点击了我们的网页。在这种情况下，我们必须定义一个窗口，用来收集最近一分钟内的数据，并对这个窗口内的数据进行计算。

窗口可以是时间驱动的（Time Window，例如：每30秒钟），也可以是数据驱动的（Count Window，例如：每一百个元素）。一种经典的窗口分类可以分成：滚动窗口（Tumbling Window，无重叠），滑动窗口（Sliding Window，有重叠），和会话窗口（Session Window，活动间隙）。

我们举个具体的场景来形象地理解不同窗口的概念。假设，淘宝网会记录每个用户每次购买的商品个数，我们要做的是统计不同窗口中用户购买商品的总数。下图给出了几种经典的窗口切分概述图：



上图中，raw data stream 代表用户的购买行为流，圈中的数字代表该用户本次购买的商品个数，事件是按时间分布的，所以可以看出事件之间是有time gap的。

2、Time Window

Time Window 是根据时间对数据流进行分组的。我们只需要知道 Flink 提出了三种时间的概念，分别是 event time（事件时间：事件发生时的时间），ingestion time（摄取时间：事件进入流处理系统的时间），processing time（处理时间：消息被计算处理的时间）。Flink 中窗口机制和时间类型是完全解耦的，也就是说当需要改变时间类型时不需要更改窗口逻辑相关的代码。

1）Tumbling Time Window

如上图，我们需要统计每一分钟中用户购买的商品的总数，需要将用户的行为事件按每一分钟进行切分，这种切分被成为翻滚时间窗口（Tumbling Time Window）。翻滚窗口能将数据流切分成不重叠的窗口，每一个事件只能属于一个窗口。通过使用 DataStream API，我们可以这样实现：

```
// Stream of (userId, buyCnt)
val buyCnts: DataStream[(Int, Int)] = ...

val tumblingCnts: DataStream[(Int, Int)] = buyCnts
  // key stream by userId
  .keyBy(0)
  // tumbling time window of 1 minute length
  .timeWindow(Time.minutes(1))
  // compute sum over buyCnt
  .sum(1)
```

2）Sliding Time Window

公众号：旧时光大数据

对于某些应用，它们需要的窗口是不间断的，需要平滑地进行窗口聚合。比如，我们可以每30秒计算一次最近一分钟用户购买的商品总数。这种窗口我们称为滑动时间窗口（Sliding Time Window）。在滑窗中，一个元素可以对应多个窗口。通过使用 DataStream API，我们可以这样实现：

```
val slidingCnts: DataStream[(Int, Int)] = buyCnts
  .keyBy(0)
  // sliding time window of 1 minute length and 30 secs trigger interval
  .timeWindow(Time.minutes(1), Time.seconds(30))
  .sum(1)
```

3、Count Window

Count Window 是根据元素个数对数据流进行分组的。

1）Tumbling Count Window

当我们想要每100个用户购买行为事件统计购买总数，那么每当窗口中填满100个元素了，就会对窗口进行计算，这种窗口我们称之为翻滚计数窗口（Tumbling Count Window），上图所示窗口大小为3个。通过使用 DataStream API，我们可以这样实现：

```
// Stream of (userId, buyCnts)
val buyCnts: DataStream[(Int, Int)] = ...

val tumblingCnts: DataStream[(Int, Int)] = buyCnts
  // key stream by sensorId
  .keyBy(0)
  // tumbling count window of 100 elements size
  .countWindow(100)
  // compute the buyCnt sum
  .sum(1)
```

2）Sliding Count Window

当然Count Window 也支持 Sliding Window，虽在上图中未描述出来，但和Sliding Time Window含义是类似的，例如计算每10个元素计算一次最近100个元素的总和，代码示例如下。

```
val slidingCnts: DataStream[(Int, Int)] = vehicleCnts
  .keyBy(0)
  // sliding count window of 100 elements size and 10 elements trigger interval
  .countWindow(100, 10)
  .sum(1)
```

4、Session Window

在这种用户交互事件流中，我们首先想到的是将事件聚合到会话窗口中（一段用户持续活跃的周期），由非活跃的间隙分隔开。如上图所示，就是需要计算每个用户在活跃期间总共购买的商品数量，如果用户30秒没有活动则视为会话断开（假设raw data stream是单个用户的购买行为流）。Session Window 的示例代码如下：

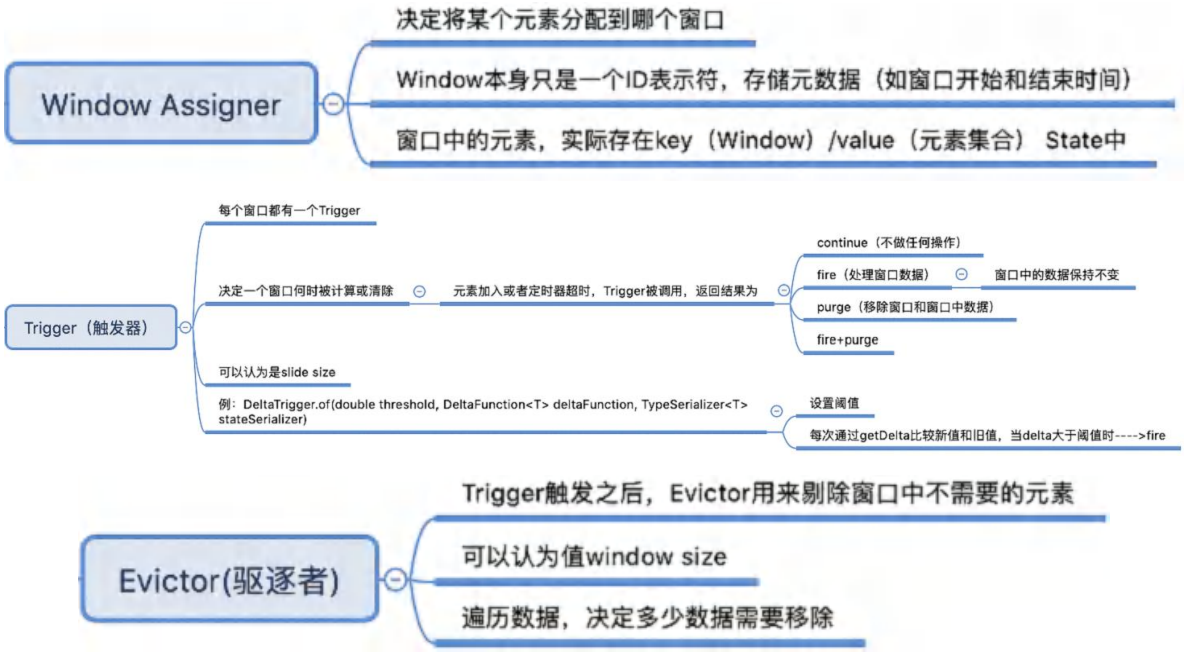
```
// Stream of (userId, buyCnts)
val buyCnts: DataStream[(Int, Int)] = ...

val sessionCnts: DataStream[(Int, Int)] = vehicleCnts
  .keyBy(0)
  // session window based on a 30 seconds session gap interval
  .window(ProcessingTimeSessionWindows.withGap(Time.seconds(30)))
  .sum(1)
```

公众号：旧时光大数据

一般而言，window 是在无限的流上定义了一个有限的元素集合。这个集合可以是基于时间的，元素个数的，时间和个数结合的，会话间隙的，或者是自定义的。Flink 的 DataStream API 提供了简洁的算子来满足常用的窗口操作，同时提供了通用的窗口机制来允许用户自己定义窗口分配逻辑。下面我们会对 Flink 窗口相关的 API 进行剖析。

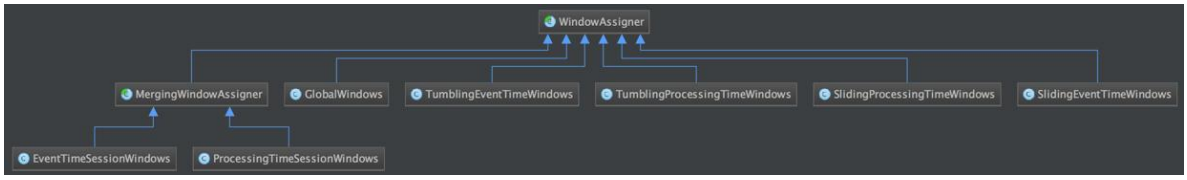
5、剖析 Window API

得益于 Flink Window API 松耦合设计，我们可以非常灵活地定义符合特定业务的窗口。Flink 中定义一个窗口主要需要以下三个组件。



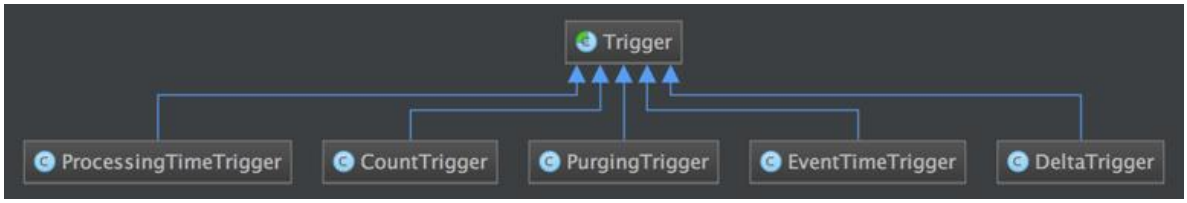
1）**Window Assigner**：用来决定某个元素被分配到哪个/哪些窗口中去。

如下类图展示了目前内置实现的 Window Assigners：



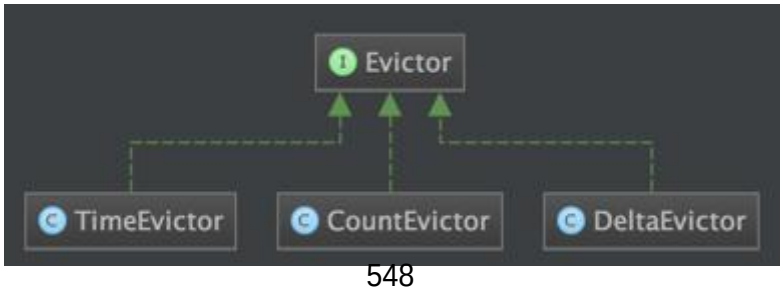
2）**Trigger**：触发器。决定了一个窗口何时能够被计算或清除，每个窗口都会拥有一个自己的Trigger。

如下类图展示了目前内置实现的 Triggers：



3）**Evictor**：可以译为“驱逐者”。在Trigger触发之后，在窗口被处理之前，Evictor（如果有Evictor的话）会用来剔除窗口中不需要的元素，相当于一个filter。

如下类图展示了目前内置实现的 Evictors：

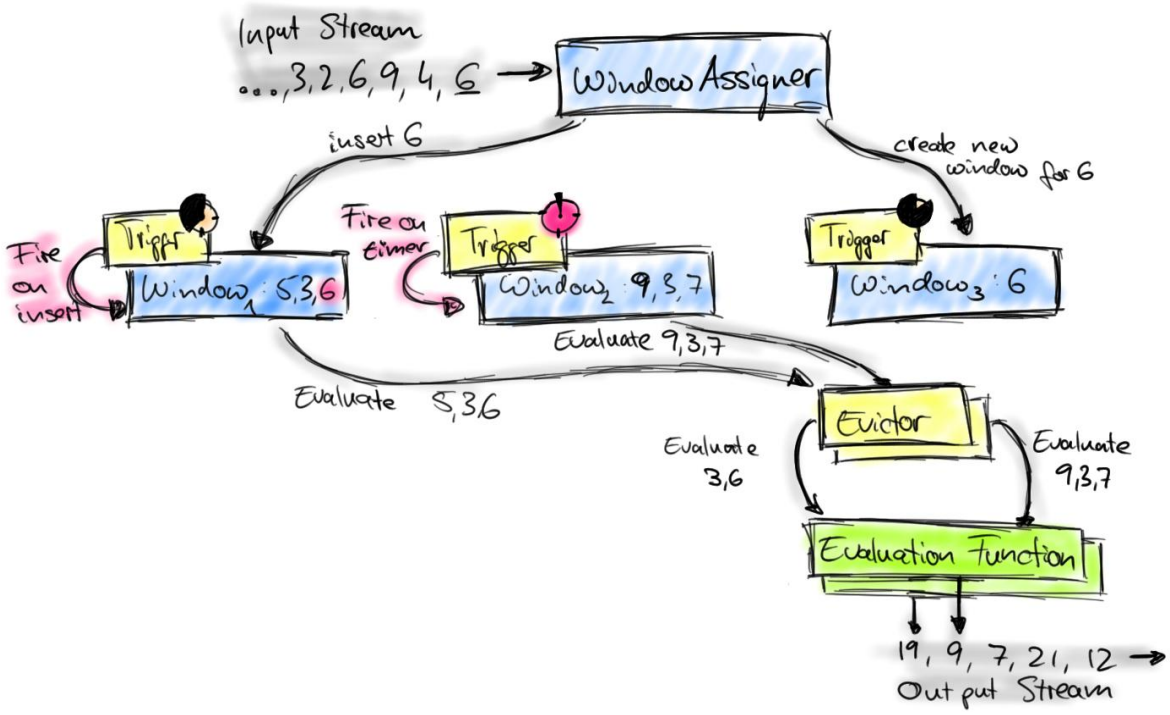


公众号：旧时光大数据

上述三个组件的不同实现的不同组合，可以定义出非常复杂的窗口。Flink 中内置的窗口也都是基于这三个组件构成的，当然内置窗口有时候无法解决用户特殊的需求，所以 Flink 也暴露了这些窗口机制的内部接口供用户实现自定义的窗口。下面我们将基于这三者探讨窗口的实现机制。

6、Window 的实现

下图描述了 Flink 的窗口机制以及各组件之间是如何相互工作的。



首先上图中的组件都位于一个算子（window operator）中，数据流源源不断地进入算子，每一个到达的元素都会被交给 WindowAssigner。WindowAssigner 会决定元素被放到哪个或哪些窗口（window），可能会创建新窗口。因为一个元素可以被放入多个窗口中，所以同时存在多个窗口是可能的。注意，`Window` 本身只是一个ID标识符，其内部可能存储了一些元数据，如 `TimeWindow` 中有开始和结束时间，但是并不会存储窗口中的元素。窗口中的元素实际存储在 Key/Value State 中，key为 `Window`，value为元素集合（或聚合值）。为了保证窗口的容错性，该实现依赖了 Flink 的 State 机制。

每一个窗口都拥有一个属于自己的 Trigger，Trigger上会有定时器，用来决定一个窗口何时能够被计算或清除。每当有元素加入到该窗口，或者之前注册的定时器超时了，那么Trigger都会被调用。Trigger的返回结果可以是 continue（不做任何操作），fire（处理窗口数据），purge（移除窗口和窗口中的数据），或者 fire + purge。一个Trigger的调用结果只是fire的话，那么会计算窗口并保留窗口原样，也就是说窗口中的数据仍然保留不变，等待下次Trigger fire的时候再次执行计算。一个窗口可以被重复计算多次知道它被 purge 了。在purge之前，窗口会一直占用着内存。

当Trigger fire了，窗口中的元素集合就会交给 `Evictor`（如果指定了的话）。Evictor 主要用来遍历窗口中的元素列表，并决定最先进入窗口的多少个元素需要被移除。剩余的元素会交给用户指定的函数进行窗口的计算。如果没有 Evictor 的话，窗口中的所有元素会一起交给函数进行计算。

计算函数收到了窗口的元素（可能经过了 Evictor 的过滤），并计算出窗口的结果值，并发送给下游。窗口的结果值可以是一个也可以是多个。DataStream API 上可以接收不同类型的计算函数，包括预定义的 `sum()`，`min()`，`max()`，还有 `ReduceFunction`，`FoldFunction`，还有 `WindowFunction`。WindowFunction 是最通用的计算函数，其他的预定义的函数基本都是基于该函数实现的。

Flink 对于一些聚合类的窗口计算（如sum,min）做了优化，因为聚合类的计算不需要将窗口中的所有数据都保存下来，只需要保存一个result值就可以了。每个进入窗口的元素都会执行一次聚合函数并修改result值。这样可以大大降低内存的消耗并提升性能。但是如果用户定义了 Evictor，则不会启用对聚合窗口的优化，因为 Evictor 需要遍历窗口中的所有元素，必须要将窗口中所有元素都存下来。

7、源码分析

公众号：旧时光大数据

1）Count Window实现

Count Window 是使用三组件的典范，我们可以在 `KeyedStream` 上创建 Count Window，其源码如下所示：

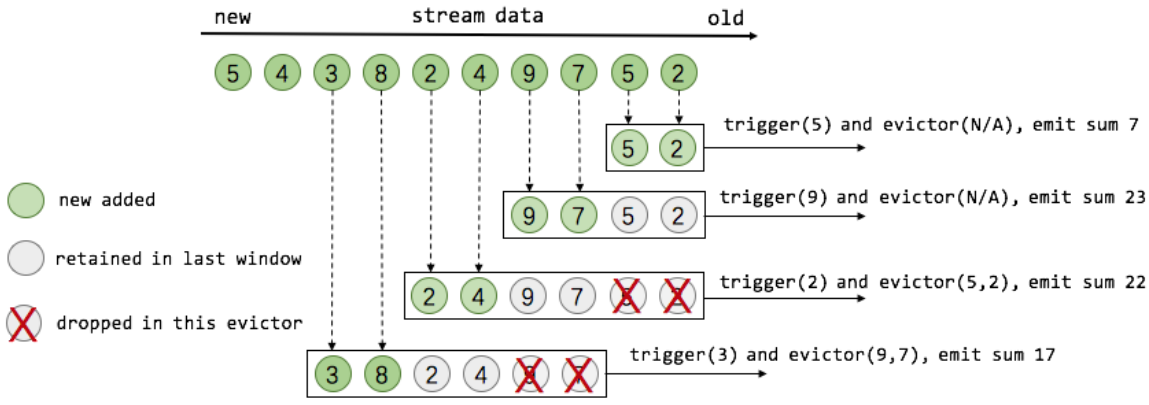
```
// tumbling count window
public WindowedStream<T, KEY, GlobalWindow> countWindow(long size) {
  return window(GlobalWindows.create())  // create window stream using GlobalWindows
        .trigger(PurgingTrigger.of(CountTrigger.of(size))); // trigger is window size
}
// sliding count window
public WindowedStream<T, KEY, GlobalWindow> countWindow(long size, long slide) {
  return window(GlobalWindows.create())
    .evictor(CountEvictor.of(size))  // evictor is window size
    .trigger(CountTrigger.of(slide)); // trigger is slide size
}
```

第一个函数是申请翻滚计数窗口，参数为窗口大小。第二个函数是申请滑动计数窗口，参数分别为窗口大小和滑动大小。它们都是基于 `GlobalWindows` 这个 WindowAssigner 来创建的窗口，该assigner会将所有元素都分配到同一个global window中，所有 `GlobalWindows` 的返回值一直是 `GlobalWindow` 单例。基本上自定义的窗口都会基于该assigner实现。

翻滚计数窗口并不带evictor，只注册了一个trigger。该trigger是带purge功能的 CountTrigger。也就是说每当窗口中的元素数量达到了 window-size，trigger就会返回fire+purge，窗口就会执行计算并清空窗口中的所有元素，再接着储备新的元素。从而实现了tumbling的窗口之间无重叠。

滑动计数窗口的各窗口之间是有重叠的，但我们用的 GlobalWindows assinger 从始至终只有一个窗口，不像 sliding time assigner 可以同时存在多个窗口。所以trigger结果不能带purge，也就是说计算完窗口后窗口中的数据要保留下来（供下个滑窗使用）。另外，trigger的间隔是slide-size，evictor的保留的元素个数是window-size。也就是说，每个滑动间隔就触发一次窗口计算，并保留下最新进入窗口的window-size个元素，剔除旧元素。

假设有一个滑动计数窗口，每2个元素计算一次最近4个元素的总和，那么窗口工作示意图如下所示：



图中所示的各个窗口逻辑上是不同的窗口，但在物理上是同一个窗口。该滑动计数窗口，trigger的触发条件是元素个数达到2个（每进入2个元素就会触发一次），evictor保留的元素个数是4个，每次计算完窗口总和后会保留剩余的元素。所以第一次触发trigger是当元素5进入，第三次触发trigger是当元素2进入，并驱逐5和2，计算剩余的4个元素的总和（22）并发送出去，保留下2,4,9,7元素供下个逻辑窗口使用。

2）Time Window 实现

同样的，我们也可以在 `KeyedStream` 上申请 Time Window，其源码如下所示：

公众号：旧时光大数据

```
// tumbling time window
public WindowedStream<T, KEY, TimeWindow> timeWindow(Time size) {
  if (environment.getStreamTimeCharacteristic() ==
TimeCharacteristic.ProcessingTime) {
    return window(TumblingProcessingTimeWindows.of(size));
  } else {
    return window(TumblingEventTimeWindows.of(size));
  }
}
// sliding time window
public WindowedStream<T, KEY, TimeWindow> timeWindow(Time size, Time slide) {
  if (environment.getStreamTimeCharacteristic() ==
TimeCharacteristic.ProcessingTime) {
    return window(SlidingProcessingTimeWindows.of(size, slide));
  } else {
    return window(SlidingEventTimeWindows.of(size, slide));
  }
}
```

在方法体内部会根据当前环境注册的时间类型，使用不同的WindowAssigner创建window。可以看到，EventTime和IngestTime都使用了 `XXXEventTimeWindows` 这个assigner，因为EventTime和IngestTime在底层的实现上只是在Source处为Record打时间戳的实现不同，在window operator中的处理逻辑是一样的。

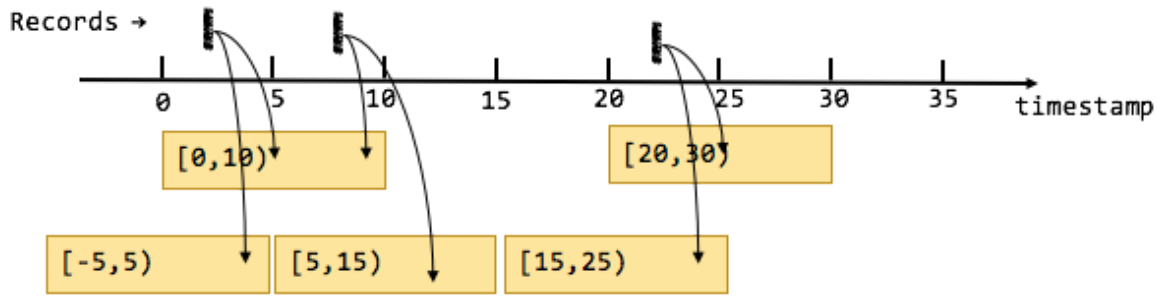
这里我们分析sliding process time window，如下是相关源码：

```
public class SlidingProcessingTimeWindows extends WindowAssigner<Object,
TimeWindow> {
  private static final long serialVersionUID = 1L;

  private final long size;

  private final long slide;

  private SlidingProcessingTimeWindows(long size, long slide) {
    this.size = size;
    this.slide = slide;
  }

  @Override
  public Collection<TimeWindow> assignWindows(Object element, long timestamp) {
    timestamp = System.currentTimeMillis();
    List<TimeWindow> windows = new ArrayList<>((int) (size / slide));
    // 对齐时间戳
    long lastStart = timestamp - timestamp % slide;
    for (long start = lastStart;
      start > timestamp - size;
      start -= slide) {
      // 当前时间戳对应了多个window
      windows.add(new TimeWindow(start, start + size));
    }
    return windows;
  }
  ...
}

public class ProcessingTimeTrigger extends Trigger<Object, TimeWindow> {
  @Override
```

公众号：旧时光大数据

```
32      // 每个元素进入窗口都会调用该方法
33      public TriggerResult onElement(Object element, long timestamp, TimeWindow
    window, TriggerContext ctx) {
34          // 注册定时器，当系统时间到达window end timestamp时会回调该trigger的
    onProcessingTime方法
35          ctx.registerProcessingTimeTimer(window.getEnd());
36          return TriggerResult.CONTINUE;
37      }
38
39      @Override
40      // 返回结果表示执行窗口计算并清空窗口
41      public TriggerResult onProcessingTime(long time, TimeWindow window,
    TriggerContext ctx) {
42          return TriggerResult.FIRE_AND_PURGE;
43      }
44      ...
45  }
46
```

首先，`SlidingProcessingTimeWindows` 会对每个进入窗口的元素根据系统时间分配到 `(size / slide)` 个不同的窗口，并会在每个窗口上根据窗口结束时间注册一个定时器（相同窗口只会注册一份），当定时器超时时意味着该窗口完成了，这时会回调对应窗口的Trigger的 `onProcessingTime` 方法，返回FIRE_AND_PURGE，也就是会执行窗口计算并清空窗口。整个过程示意图如下：



如上图所示横轴代表时间戳（为简化问题，时间戳从0开始），第一条record会被分配到[-5,5)和[0,10)两个窗口中，当系统时间到5时，就会计算[-5,5)窗口中的数据，并将结果发送出去，最后清空窗口中的数据，释放该窗口资源。

说下Flink的CEP

问过的一些公司：安恒信息

参考答案：

1、CEP概述

复合事件处理（Complex Event Processing，CEP）是一种基于动态环境中事件流的分析技术，事件在这里通常是有意义的状态变化，通过分析事件间的关系，利用过滤、关联、聚合等技术，根据事件间的时序关系和聚合关系制定检测规则，持续地从事件流中查询出符合要求的事件序列，最终分析得到更复杂的复合事件。

2、特征

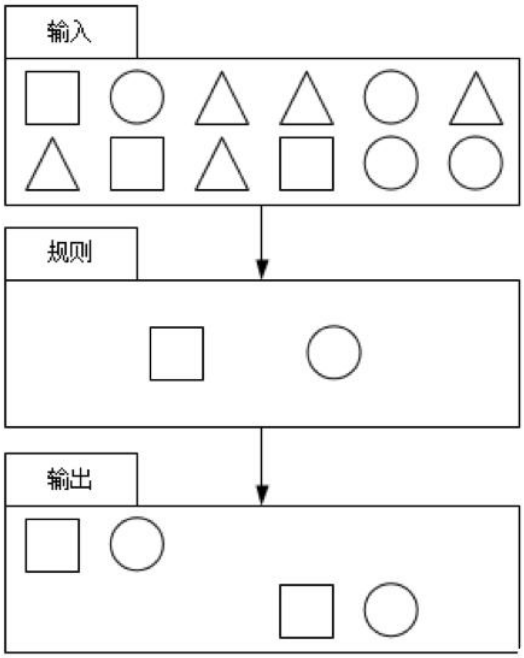
目标：从有序的简单事件流中发现一些高阶特征；

输入：一个或多个简单事件构成的事件流；

处理：识别简单事件之间的内在联系，多个符合一定规则的简单事件构成复杂事件；

公众号：旧时光大数据

输出：满足规则的复杂事件。



3、功能

CEP用于分析低延迟、频繁产生的不同来源的事件流。CEP可以帮助在复杂的、不相关的时间流中找出有意义的模式和复杂的关系，以接近实时或准实时的获得通知或组织一些行为。

CEP支持在流上进行模式匹配，根据模式的条件不同，分为连续的条件或不连续的条件；模式的条件允许有时间的限制，当条件范围内没有达到满足的条件时，会导致模式匹配超时。

看起来很简单，但是它有很多不同的功能：

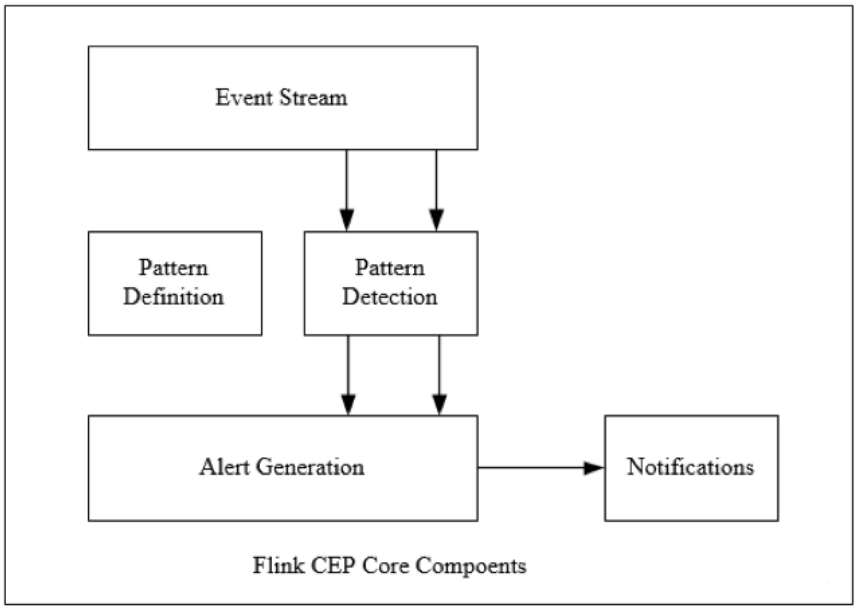
- 输入的流数据，尽快产生结果；
- 在2个事件流上，基于时间进行聚合类的计算；
- 提供实时/准实时的警告和通知；
- 在多样的数据源中产生关联分析模式；
- 高吞吐、低延迟的处理。

市场上有多种CEP的解决方案，例如Spark、Samza、Beam等，但他们都没有提供专门的库支持。然而，Flink提供了专门的CEP库。

4、主要组件

Flink为CEP提供了专门的Flink CEP library，它包含如下组件：Event Stream、Pattern定义、Pattern检测和生成Alert。首先，开发人员要在DataStream流上定义出模式条件，之后Flink CEP引擎进行模式检测，必要时生成警告。

公众号：旧时光大数据



5、Pattern API

处理事件的规则，被叫作模式（Pattern）。

Flink CEP提供了Pattern API用于对输入流数据进行复杂事件规则定义，用来提取符合规则的事件序列。

模式大致分为三类：

个体模式（Individual Patterns）

组成复杂规则的每一个单独的模式定义，就是个体模式。

```
start.times(3).where(_.behavior.startsWith('fav'))

```

组合模式（Combining Patterns，也叫模式序列）

很多个体模式组合起来，就形成了整个的模式序列。

模式序列必须以一个初始模式开始：

```
val start = Pattern.begin('start')

```

模式组（Group of Pattern）

将一个模式序列作为条件嵌套在个体模式里，成为一组模式。

1）个体模式

个体模式包括单例模式和循环模式。单例模式只接收一个事件，而循环模式可以接收多个事件。

（1）量词

可以在一个个体模式后追加量词，也就是指定循环次数。

公众号：旧时光大数据

```
// 匹配出现4次
start.time(4)
// 匹配出现0次或4次
start.time(4).optional
// 匹配出现2、3或4次
start.time(2,4)
// 匹配出现2、3或4次，并且尽可能多地重复匹配
start.time(2,4).greedy
// 匹配出现1次或多次
start.oneOrMore
// 匹配出现0、2或多次，并且尽可能多地重复匹配
start.timesOrMore(2).optional.greedy

```

（2）条件

每个模式都需要指定触发条件，作为模式是否接受事件进入的判断依据。CEP中的个体模式主要通过调用.where()、.or()和.until()来指定条件。按不同的调用方式，可以分成以下几类：

① 简单条件

通过.where()方法对事件中的字段进行判断筛选，决定是否接收该事件

```
start.where(event=>event.getName.startsWith("foo"))

```

② 组合条件

将简单的条件进行合并；or()方法表示或逻辑相连，where的直接组合就相当于与and。

```
Pattern.where(event => …/*some condition*/).or(event => /*or condition*/)

```

③ 终止条件

如果使用了oneOrMore或者oneOrMore.optional，建议使用.until()作为终止条件，以便清理状态。

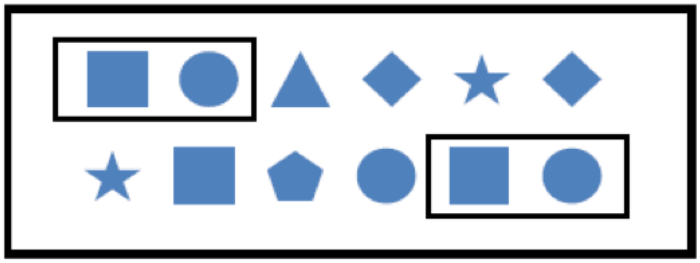
④ 迭代条件

能够对模式之前所有接收事件进行处理；调用.where((value,ctx) => {…})，可以调用ctx.getEventForPattern("name")

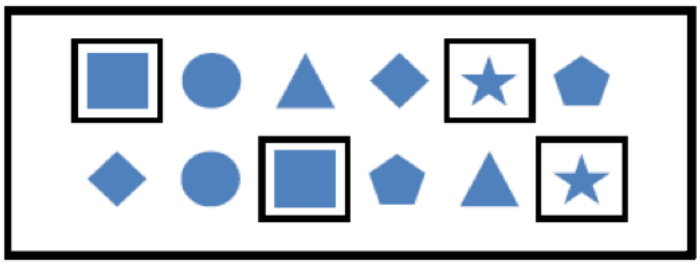
2）模式序列

不同的近邻模式如下图：

公众号：旧时光大数据



严格近邻



宽松近邻

（1）严格近邻

所有事件按照严格的顺序出现，中间没有任何不匹配的事件，由.next()指定。例如对于模式“a next b”，事件序列“a,c,b1,b2”没有匹配。

（2）宽松近邻

允许中间出现不匹配的事件，由.followedBy()指定。例如对于模式“a followedBy b”，事件序列“a,c,b1,b2”匹配为{a,b1}。

（3）非确定性宽松近邻

进一步放宽条件，之前已经匹配过的事件也可以再次使用，由.followedByAny()指定。例如对于模式“a followedByAny b”，事件序列“a,c,b1,b2”匹配为{ab1}，{a,b2}。

除了以上模式序列外，还可以定义“不希望出现某种近邻关系”：

.notNext()：不想让某个事件严格紧邻前一个事件发生。

.notFollowedBy()：不想让某个事件在两个事件之间发生。

需要注意：

①所有模式序列必须以.begin()开始；

②模式序列不能以.notFollowedBy()结束；

③“not”类型的模式不能被optional所修饰；

④可以为模式指定时间约束，用来要求在多长时间内匹配有效。

```
next.within(Time.seconds(10))

```

3）模式的检测

指定要查找的模式序列后，就可以将其应用于输入流以检测潜在匹配。调用CEP.pattern()，给定输入流和模式，就能得到一个PatternStream。

```
val input:DataStream[Event] = …
val pattern:Pattern[Event,_] = …
val patternStream:PatternStream[Event]=CEP.pattern(input,pattern)

```

公众号：旧时光大数据

4）匹配事件的提取

创建PatternStream之后，就可以应用select或者flatSelect方法，从检测到的事件序列中提取事件了。

select()方法需要输入一个select function作为参数，每个成功匹配的事件序列都会调用它。

select()以一个Map[String,Iterable[IN]]来接收匹配到的事件序列，其中key就是每个模式的名称，而value就是所有接收到的事件的Iterable类型。

```
def selectFn(pattern : Map[String,Iterable[IN]]):OUT={
  val startEvent = pattern.get("start").get.next
  val endEvent = pattern.get("end").get.next
  OUT(startEvent, endEvent)
}

```

flatSelect通过实现PatternFlatSelectFunction实现与select相似的功能。唯一的区别就是flatSelect方法可以返回多条记录，它通过一个Collector[OUT]类型的参数来将要输出的数据传递到下游。

5）超时事件的提取

当一个模式通过within关键字定义了检测窗口时间时，部分事件序列可能因为超过窗口长度而被丢弃；为了能够处理这些超时的部分匹配，select和flatSelect API调用允许指定超时处理程序。

6、Flink CEP实战

为了使用Flink CEP，需要导入pom依赖。

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-cep-scala_2.11</artifactId>
    <version>1.7.0</version>
</dependency>

```

LoginLog.csv中的数据格式为：

```
5402,83.149.11.115,success,1558430815
23064,66.249.3.15,fail,1558430826
5692,80.149.25.29,fail,1558430833
7233,86.226.15.75,success,1558430832
5692,80.149.25.29,success,1558430840
29607,66.249.73.135,success,1558430841

```

需求：检测一个用户在3秒内连续登陆失败。

```
// 输入的登录事件样例类
case class LoginEvent( userId: Long, ip: String, eventType: String, eventTime: Long )
// 输出的异常报警信息样例类
case class Warning( userId: Long, firstFailTime: Long, lastFailTime: Long, warningMsg: String)

object LoginFailWithCep {
  def main(args: Array[String]): Unit = {
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    env.setParallelism(1)

```

公众号：旧时光大数据

```
12        // 1. 读取事件数据，创建简单事件流
13        val resource = getClass.getResource("/LoginLog.csv")
14        val loginEventStream = env.readTextFile(resource.getPath)
15          .map( data => {
16            val dataArray = data.split(",")
17            LoginEvent( dataArray(0).trim.toLong, dataArray(1).trim,
    dataArray(2).trim, dataArray(3).trim.toLong )
18          } )
19          .assignTimestampsAndWatermarks( new
    BoundedOutOfOrdernessTimestampExtractor[LoginEvent](Time.seconds(5)) {
20            override def extractTimestamp(element: LoginEvent): Long =
    element.eventTime * 1000L
21          } )
22          .keyBy(_.userId)
23
24        // 2. 定义匹配模式
25        val loginFailPattern = Pattern.begin[LoginEvent]("begin").where(_.eventType
    == "fail")
26          .next("next").where(_.eventType == "fail")
27          .within(Time.seconds(3))
28
29        // 3. 在事件流上应用模式，得到一个pattern stream
30        val patternStream = CEP.pattern(loginEventStream, loginFailPattern)
31
32        // 4. 从pattern stream上应用select function，检出匹配事件序列
33        val loginFailDataStream = patternStream.select( new LoginFailMatch() )
34
35        loginFailDataStream.print()
36
37        env.execute("login fail with cep job")
38      }
39    }
40
41    class LoginFailMatch() extends PatternSelectFunction[LoginEvent, Warning]{
42      override def select(map: util.Map[String, util.List[LoginEvent]]): Warning = {
43        // 从map中按照名称取出对应的事件
44        //    val iter = map.get("begin").iterator()
45        val firstFail = map.get("begin").iterator().next()
46        val lastFail = map.get("next").iterator().next()
47        Warning( firstFail.userId, firstFail.eventTime, lastFail.eventTime, "login
    fail!" )
48      }
49    }
50
```

说一说Flink的Checkpoint机制

可回答：Flink的容错机制

问过的一些公司：网易x2，爱奇艺，竞技世界，美团(2021.08)

参考答案：

`Checkpoint`：某一时刻，Flink中所有的Operator的当前State的全局快照，一般存在磁盘上。

公众号：旧时光大数据

Flink提供了Exactly once特性，是依赖于带有barrier的分布式快照+可部分重发的数据源功能实现的。而分布式快照中，就保存了operator的状态信息。

Flink的失败恢复依赖于**检查点机制 +可部分重发的数据源**。

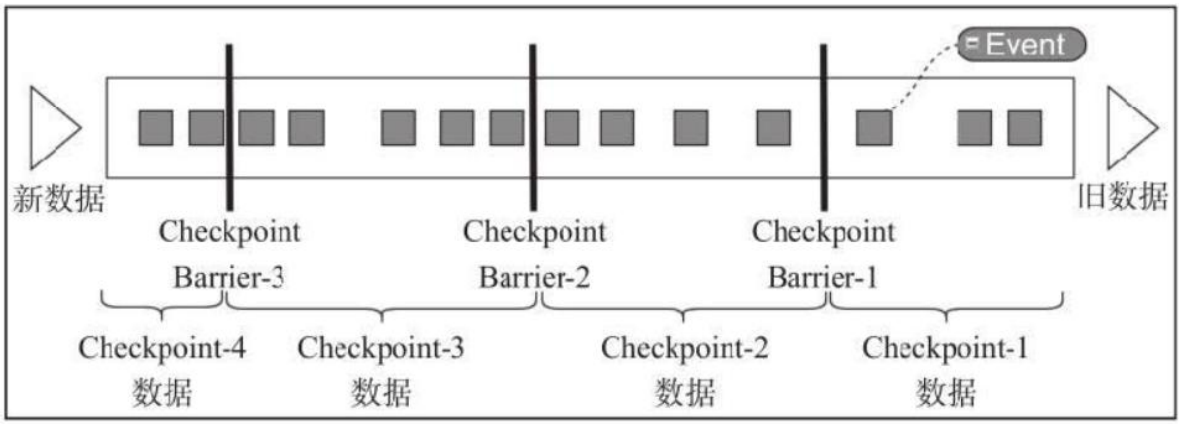
检查点机制机制：checkpoint定期触发，产生快照，快照中记录了：

- 当前检查点开始时数据源（例如Kafka）中消息的offset。
- 记录了所有有状态的operator当前的状态信息（例如sum中的数值）。

可部分重发的数据源：Flink选择最近完成的检查点K，然后系统重放整个分布式的数据流，然后给予每个operator他们在检查点k快照中的状态。数据源被设置为从位置Sk开始重新读取流。例如在Apache Kafka中，那意味着告诉消费者从偏移量Sk开始重新消费。

1、Checkpoint

Flink中基于异步轻量级的分布式快照技术提供了Checkpoints容错机制，**分布式快照可以将同一时间点Task/Operator的状态数据全局统一快照处理，包括Keyed State和Operator State（State后面也接着介绍了）**。如下图所示，Flink会在输入的数据集上间隔性地生成checkpoint barrier，通过栅栏（barrier）将间隔时间段内的数据划分到相应的checkpoint中。当应用出现异常时，Operator就能够从上一次快照中恢复所有算子之前的状态，从而保证数据的一致性。例如在Kafka Consumer算子中维护Offset状态，当系统出现问题无法从Kafka中消费数据时，可以将Offset记录在状态中，当任务重新恢复时就能够从指定的偏移量开始消费数据。对于状态占用空间比较小的应用，快照产生过程非常轻量，高频率创建且对Flink任务性能影响相对较小。checkpoint过程中状态数据一般被保存在一个可配置的环境中，通常是在JobManager节点或HDFS上。



默认情况下Flink不开启检查点的，用户需要在程序中通过调用enable-Checkpointing(n)方法配置和开启检查点，其中n为检查点执行的时间间隔，单位为毫秒。除了配置检查点时间间隔，针对检查点配置还可以调整其他相关参数：

1）Checkpoint开启和时间间隔指定

开启检查点并且指定检查点时间间隔为1000ms，根据实际情况自行选择，如果状态比较大，则建议适当增加该值。

```
1   env.enableCheckpointing(1000);
2
```

2）exactly-ance和at-least-once语义选择

可以选择exactly-once语义保证整个应用内端到端的数据一致性，这种情况比较适合于数据要求比较高，不允许出现丢数据或者数据重复，与此同时，Flink的性能也相对较弱，而at-least-once语义更适合于时廷和吞吐量要求非常高但对数据的一致性要求不高的场景。如下通过setCheckpointingMode()方法来设定语义模式，默认情况下使用的是exactly-once模式。

```
env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);

```

3）Checkpoint超时时间

超时时间指定了每次Checkpoint执行过程中的上限时间范围，一旦Checkpoint执行时间超过该阈值，Flink将会中断Checkpoint过程，并按照超时处理。该指标可以通过setCheckpointTimeout方法设定，默认为10分钟。

```
env.getCheckpointConfig().setCheckpointTimeout(60000);

```

4）检查点之间最小时间间隔

该参数主要目的是设定两个Checkpoint之间的最小时间间隔，防止出现例如状态数据过大而导致Checkpoint执行时间过长，从而导致Checkpoint积压过多，最终Flink应用密集地触发Checkpoint操作，会占用了大量计算资源而影响到整个应用的性能。

```
env.getCheckpointConfig().setMinPauseBetweenCheckpoints(500);

```

5）最大并行执行的检查点数量

通过setMaxConcurrentCheckpoints()方法设定能够最大同时执行的Checkpoint数量。在默认情况下只有一个检查点可以运行，根据用户指定的数量可以同时触发多个Checkpoint，进而提升Checkpoint整体的效率。

```
env.getCheckpointConfig().setMaxConcurrentCheckpoints(1);

```

6）外部检查点
设定周期性的外部检查点，然后将状态数据持久化到外部系统中，使用这种方式不会在任务正常停止的过程中清理掉检查点数据，而是会一直保存在外部系统介质中，另外也可以通过从外部检查点中对任务进行恢复。

```
env.getCheckpointConfig().enableExternalizedCheckpoints(ExternalizedCheckp
ointCleanup.RETAIN_ON_CANCELLATION);

```

7）failOnCheckpointingErrors

failOnCheckpointingErrors参数决定了当Checkpoint执行过程中如果出现失败或者错误时，任务是否同时被关闭，默认值为True。

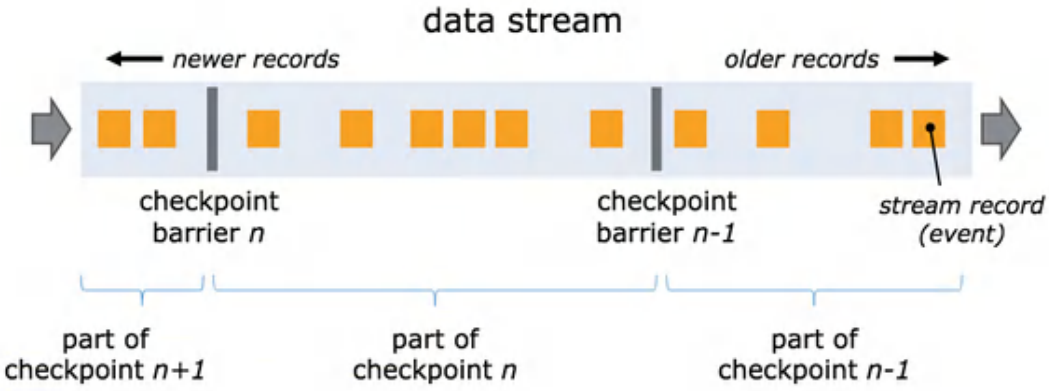
```
env.getCheckpointConfig().setFailOnCheckpointingErrors (false);

```

快照的核心概念之一是barrier。这些barrier被注入数据流并与记录一起作为数据流的一部分向下流动。barriers永远不会超过记录，数据流严格有序，barrier将数据流中的记录隔离成一系列的记录集合，并将一些集合中的数据加入到当前的快照中，而另一些数据加入到下一个快照中。

每个barrier都带有快照的ID，并且barrier之前的记录都进入了该快照。barriers不会中断流处理，非常轻量级。来自不同快照的多个barrier可以同时在流中出现，这意味着多个快照可能并发地发生。

单流的barrier：

公众号：旧时光大数据

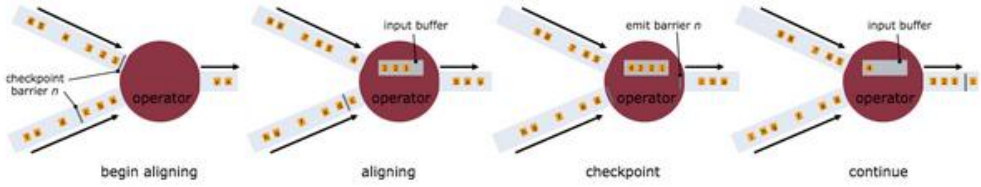


barrier在数据流源处被注入并行数据流中。快照n的barriers被插入的位置（记之为Sn）是快照所包含的数据在数据源中最大位置。例如，在Apache Kafka中，此位置将是分区中最后一条记录的偏移量。 将该位置Sn报告给checkpoint协调器（Flink的JobManager）。

然后barriers向下游流动。当一个中间操作算子从其所有输入流中收到快照n的barriers时，它会为快照n发出barriers进入其所有输出流中。 一旦sink操作算子（流式DAG的末端）从其所有输入流接收到barriers n，它就向checkpoint协调器确认快照n完成。在所有sink确认快照后，意味快照着已完成。

一旦完成快照n，job将永远不再向数据源请求Sn之前的记录，因为此时这些记录（及其后续记录）将已经通过整个数据流拓扑，也即是已经被处理结束。

多流的barrier：



接收多个输入流的运算符需要基于快照barriers上对齐(align)输入流。 上图说明了这一点：

- 一旦操作算子从一个输入流接收到快照barriers n，它就不能处理来自该流的任何记录，直到它从其他输入接收到barriers n为止。 否则，它会搞混属于快照n的记录和属于快照n + 1的记录。
- barriers n所属的流暂时会被搁置。 从这些流接收的记录不会被处理，而是放入输入缓冲区。可以看到1,2,3会一直放在Input buffer，直到另一个输入流的快照到达Operator。
- 一旦从最后一个流接收到barriers n，操作算子就会发出所有挂起的向后传送的记录，然后自己发出快照n的barriers。
- 之后，它恢复处理来自所有输入流的记录，在处理来自流的记录之前优先处理来自输入缓冲区的记录。

2、State

state一般指一个具体的task/operator的状态。Flink中包含两种基础的状态：Keyed State和Operator State。

Keyed State，就是基于KeyedStream上的状态。这个状态是跟特定的key绑定的，对KeyedStream流上的每一个key，可能都对应一个state。

Operator State与Keyed State不同，Operator State跟一个特定operator的一个并发实例绑定，整个operator只对应一个state。相比较而言，在一个operator上，可能会有很多个key，从而对应多个keyed state。

举例来说，Flink中的Kafka Connector，就使用了operator state。它会在每个connector实例中，保存该实例中消费topic的所有(partition, offset)映射。

Keyed State和Operator State，可以以两种形式存在：**原始状态**和**托管状态**(Raw and Managed State)。

托管状态是由Flink框架管理的状态，如ValueState, ListState, MapState等。而raw state即原始状态，由用户自行管理状态具体的数据结构，框架在做checkpoint的时候，使用byte[]来读写状态内容。通常在DataStream上的状态推荐使用托管的状态，当实现一个用户自定义的operator时，会使用到原始状态。

这里重点说说State-Keyed State，基于key/value的状态接口，这些状态只能用于keyedStream之上。keyedStream上的operator操作可以包含window或者map等算子操作。这个状态是跟特定的key绑定的，对KeyedStream流上的每一个key，都对应一个state。

key/value下可用的状态接口：

- ValueState：状态保存的是一个值，可以通过update()来更新，value()获取。
- ListState：状态保存的是一个列表，通过add()添加数据，通过get()方法返回一个Iterable来遍历状态值。
- ReducingState：这种状态通过用户传入的reduceFunction，每次调用add方法添加值的时候，会调用reduceFunction，最后合并到一个单一的状态值。
- MapState：即状态值为一个map。用户通过put或putAll方法添加元素。

以上所述的State对象，仅仅用于与状态进行交互（更新、删除、清空等），而真正的状态值，有可能是存在内存、磁盘、或者其他分布式存储系统中。实际上，这些状态有三种存储方式:HeapStateBackend、MemoryStateBackend、FsStateBackend、RockDBStateBackend。

- MemoryStateBackend: state数据保存在java堆内存中，执行checkpoint的时候，会把state的快照数据保存到jobmanager的内存中。
- FsStateBackend: state数据保存在taskmanager的内存中，执行checkpoint的时候，会把state的快照数据保存到配置的文件系统中，可以使用hdfs等分布式文件系统。
- RocksDBStateBackend: RocksDB跟上面的都略有不同，它会在本地文件系统中维护状态，state会直接写入本地rocksdb中。同时RocksDB需要配置一个远端的filesystem。RocksDB克服了state受内存限制的缺点，同时又能够持久化到远端文件系统中，比较适合在生产中使用。

通过创建一个StateDescriptor，可以得到一个包含特定名称的状态句柄，可以分别创建ValueStateDescriptor、 ListStateDescriptor或ReducingStateDescriptor状态句柄。状态是通过RuntimeContext来访问的，因此只能在RichFunction中访问状态。这就要求UDF时要继承Rich函数，例如RichMapFunction、RichFlatMapFunction等。

Flink的Checkpoint底层如何实现的？savepoint和checkpoint有什么区别？

可回答：Flink的checkpoint是如何创建的

问过的一些公司：科大讯飞，安恒信息，Shopee(2021.08)

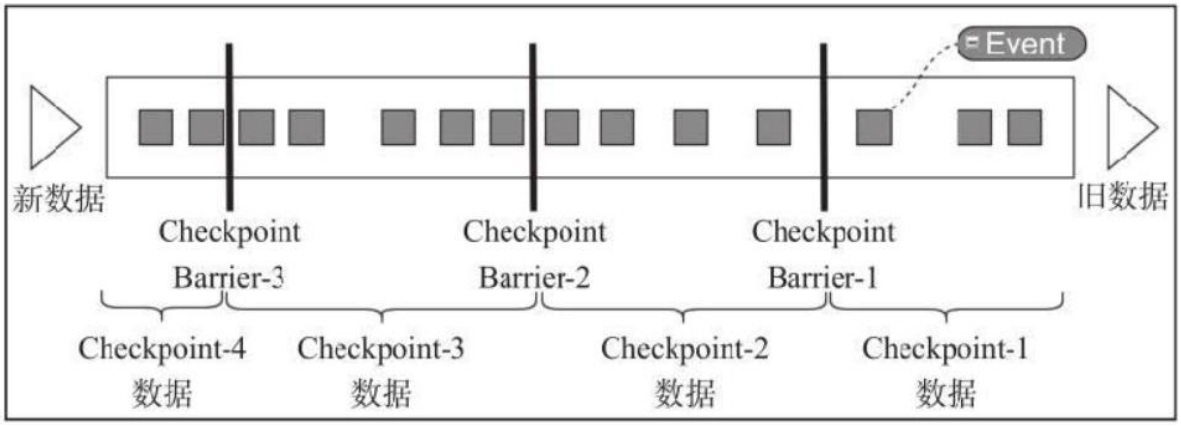
参考答案：

1、Checkpoint

Flink中基于异步轻量级的分布式快照技术提供了Checkpoints容错机制，分布式快照可以将同一时间点Task/Operator的状态数据全局统一快照处理，包括Keyed State和Operator State（State后面也接着介绍了）。如下图所示，Flink会在输入的数据集上间隔性地生成checkpoint barrier，通过栅栏（barrier）将间隔时间段内的数据划分到相应的checkpoint中。当应用出现异常时，Operator就能够从上一次快照中恢复所有算子之前的状态，从而保证数据的一致性。例如在Kafka Consumer算子中维护Offset状态，当系统出现问题无法从Kafka中消费数据时，可以将Offset记录在状态中，当任务重新恢复时就能够从指定的偏移量开始消费数据。对于状态占用空间比较小的应用，快照产生过程非常轻量，高频率创建且对Flink任务性能影响相对较小。checkpoint过程中状态数据一般被保存在一个可配置的环境中，通常是在JobManager节点或HDFS上。

公众号：旧时光大数据

checkpoint原理就是连续绘制分布式的快照，而且非常轻量级，可以连续绘制，并且不会对性能产生太大影响。

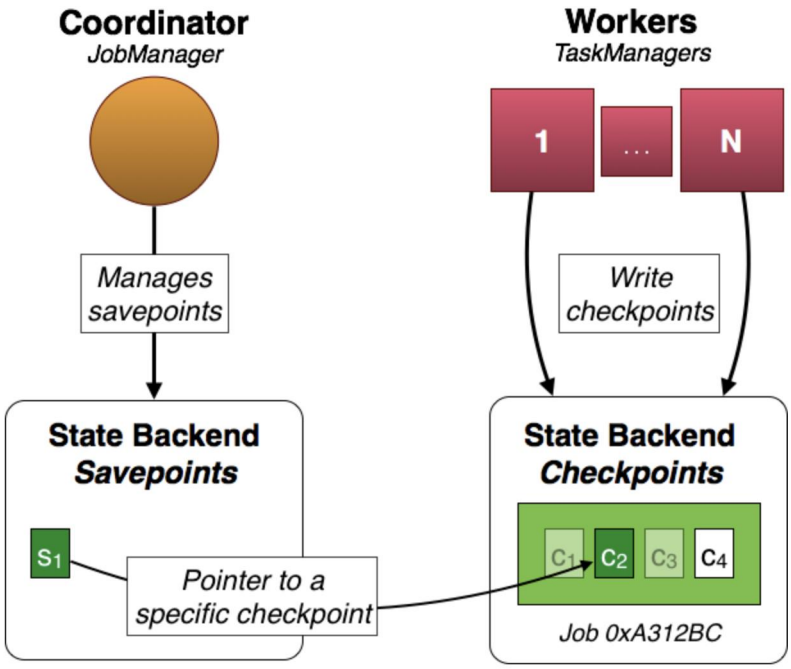


默认情况下Flink不开启检查点的，用户需要在程序中通过调用enable-Checkpointing(n)方法配置和开启检查点，其中n为检查点执行的时间间隔，单位为毫秒。

2、savepoint和checkpoint区别

Savepoints是检查点的一种特殊实现，底层其实也是使用Checkpoints的机制。Savepoints是用户以手工命令的方式触发Checkpoint，并将结果持久化到指定的存储路径中，其主要目的是帮助用户在升级和维护集群过程中保存系统中的状态数据，避免因为停机运维或者升级应用等正常终止应用的操作而导致系统无法恢复到原有的计算状态的情况，从而无法实现端到端的Excatly-Once语义保证。

下面这张来自Flink 1.1版本文档的图示出了checkpoint和savepoint的关系。



总结如下所示：

- checkpoint的侧重点是“容错”，即Flink作业意外失败并重启之后，能够直接从早先打下的checkpoint恢复运行，且不影响作业逻辑的准确性。而savepoint的侧重点是“维护”，即Flink作业需要在人工干预下手动重启、升级、迁移或A/B测试时，先将状态整体写入可靠存储，维护完毕之后再从savepoint恢复现场。
- savepoint是“通过checkpoint机制”创建的，所以savepoint本质上是特殊的checkpoint。
- checkpoint面向Flink Runtime本身，由Flink的各个TaskManager定时触发快照并自动清理，一般不需要用户干预；savepoint面向用户，完全根据用户的需要触发与清理。

公众号：旧时光大数据

- checkpoint的频率往往比较高（因为需要尽可能保证作业恢复的准确度），所以checkpoint的存储格式非常轻量级，但作为trade-off牺牲了一切可移植（portable）的东西，比如不保证改变并行度和升级的兼容性。savepoint则以二进制形式存储所有状态数据和元数据，执行起来比较慢而且“贵”，但是能够保证portability，如并行度改变或代码升级之后，仍然能正常恢复。
- checkpoint是支持增量的（通过RocksDB），特别是对于超大状态的作业而言可以降低写入成本。savepoint并不会连续自动触发，所以savepoint没有必要支持增量。

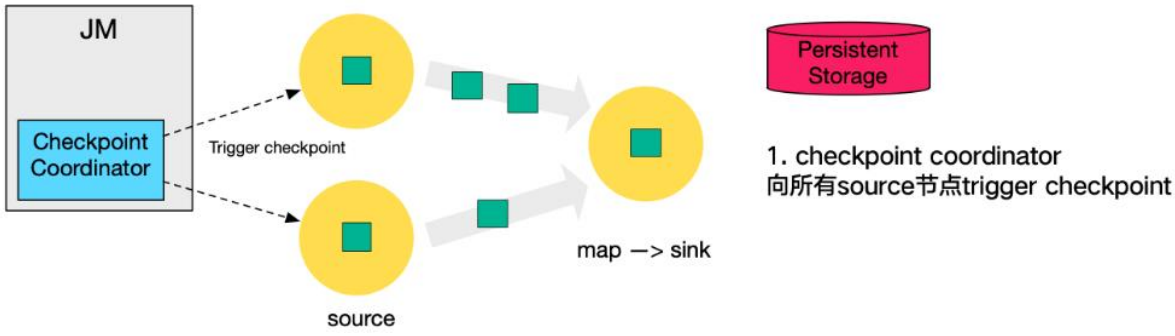
Flink的Checkpoint流程

问过的一些公司：头条，嘉云

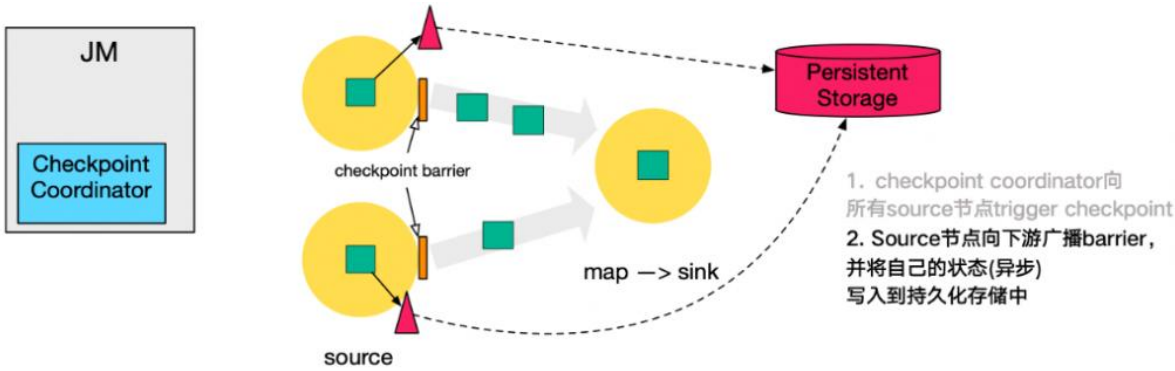
参考答案：

Checkpoint由JM的Checkpoint Coordinator发起

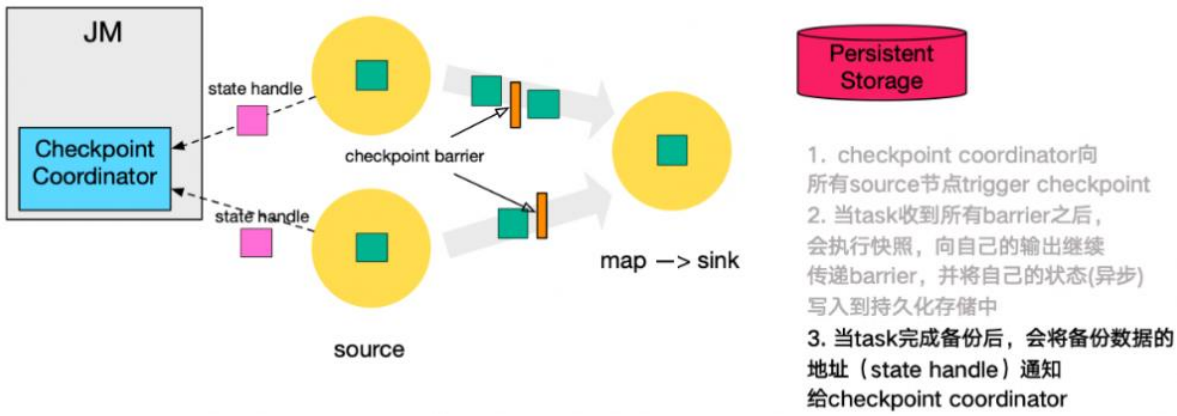
第一步，Checkpoint Coordinator 向所有 source 节点 trigger Checkpoint；



第二步，source 节点向下游广播 barrier，这个 barrier 就是实现 Chandy-Lamport 分布式快照算法的核心，下游的 task 只有收到所有 input 的 barrier 才会执行相应的 Checkpoint。



第三步，当 task 完成 state 备份后，会将备份数据的地址（state handle）通知给 Checkpoint coordinator。



公众号：旧时光大数据

这里分为同步和异步（如果开启的话）两个阶段：

1）**同步阶段**：task执行状态快照，并写入外部存储系统（根据状态后端的选择不同有所区别）

执行快照的过程：

- 对state做深拷贝
- 将写操作封装在异步的FutureTask中

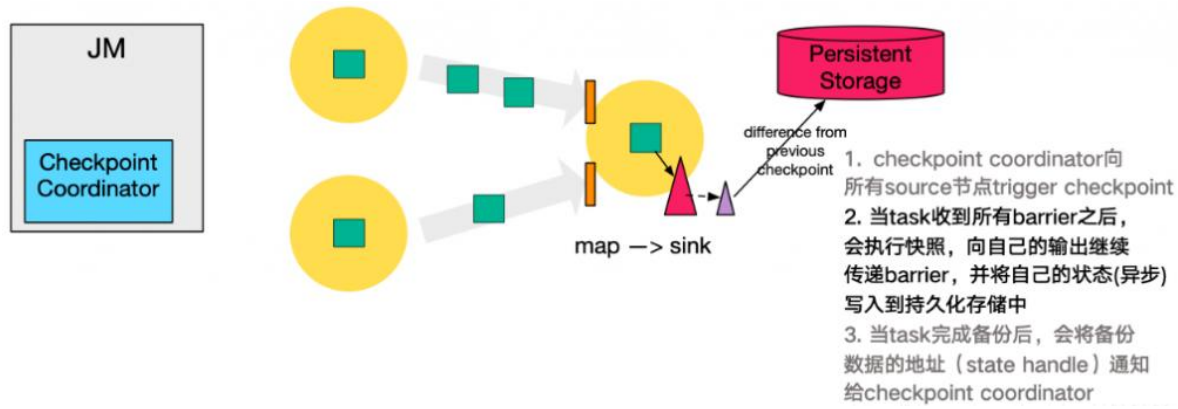
FutureTask的作用包括：

- 打开输入流
- 写入状态的元数据信息
- 写入状态
- 关闭输入流

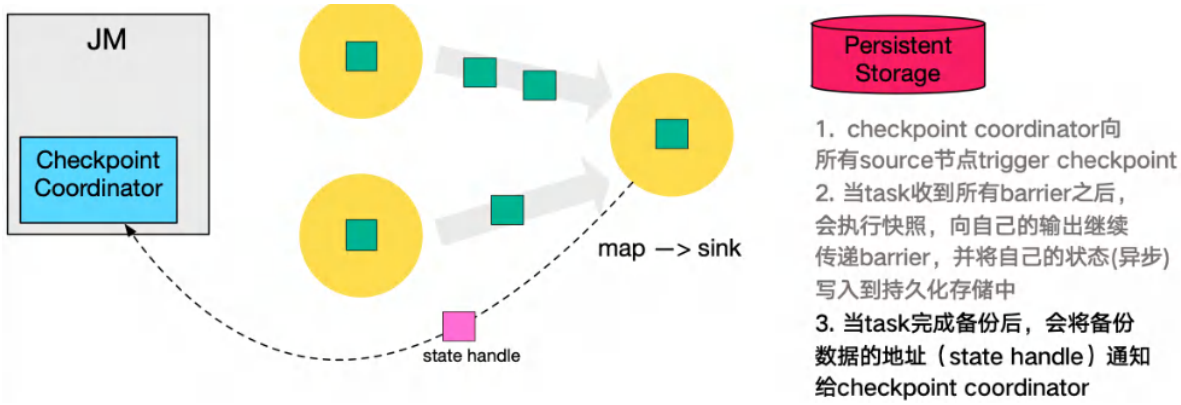
2）**异步阶段**

- 执行同步阶段创建的FutureTask
- 向Checkpoint Coordinator发送ACK响应

第四步，下游的 sink 节点收集齐上游两个 input 的 barrier 之后，会执行本地快照，这里特地展示了 RocksDB incremental Checkpoint 的流程，首先 RocksDB 会全量刷数据到磁盘上（红色大三角表示），然后 Flink 框架会从中选择没有上传的文件进行持久化备份（紫色小三角）。

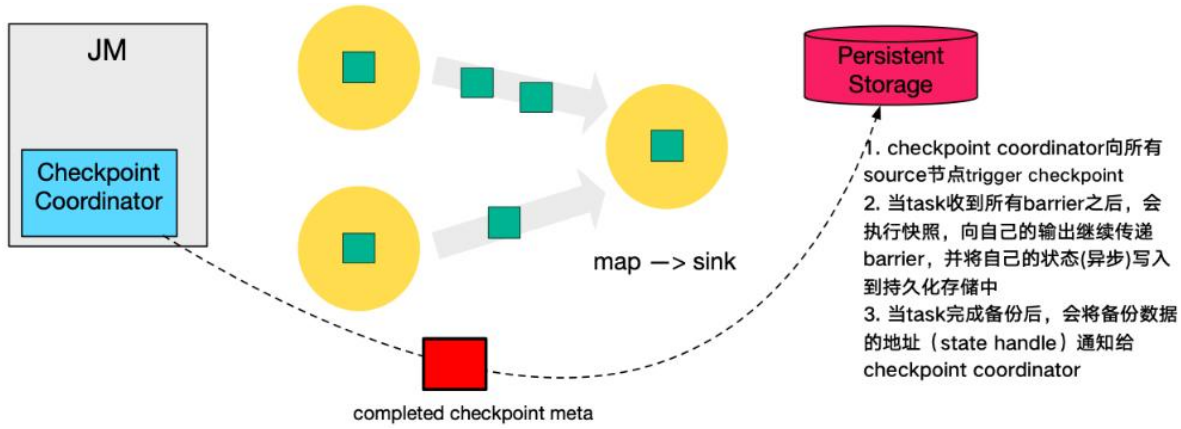


同样的，sink 节点在完成自己的 Checkpoint 之后，会将 state handle 返回通知 Coordinator。



最后，当 Checkpoint coordinator 收集齐所有 task 的 state handle，就认为这一次的 Checkpoint 全局完成了，向持久化存储中再备份一个 Checkpoint meta 文件。

公众号：旧时光大数据



Flink Checkpoint的作用

问过的一些公司：字节(2021.08)，远景智能(2021.08)

参考答案：

`Checkpoint`：某一时刻，Flink中所有的Operator的当前State的全局快照，一般存在磁盘上。

CheckPoint是通过快照（SnapShot）的方式使得将历史某些时刻的状态保存下来，当Flink任务意外挂掉之后，重新运行程序后默认从最近一次保存的完整快照处进行恢复任务。

Flink中的Checkpoint底层使用了`Chandy-Lamport algorithm`分布式快照算法，可以保证数据的在分布式环境下的一致性。

Flink中Checkpoint超时原因

问过的一些公司：

参考答案：

1）**计算量大，CPU密集性，导致TM内线程一直在processElement，而没有时间做Checkpoint**

解决方案：过滤掉部分数据，增大并行度

修改实现逻辑，进行批流分开计算，比如离线数据每半个小时进行一次计算，而实时计算只需要计算最近半小时内的数据即可。总之两个方法，一、减少源数据量，过滤黑名单或者非法ID，window聚合；二、简化处理逻辑，特别是减少遍历。

2）**数据倾斜**

解决方案： 第一，两阶段聚合；第二，重新设置并行度，改变KeyGroup的分布

3）**频繁FULL GC（'Checkpoint Duration (Async)'时间长）**

当StateSize达到200M以上，Async的时间会超过1min。 这种情况比较少见。

4）**出现反压**

包含barrier的event buffer一直不到，subTaskCheckpointCoordinator做不了Checkpoint，就会超时。

Flink的Exactly Once语义怎么保证?

可回答：1）Flink怎么保证精准一次消费？2）Flink怎么实现精准一次语义

公众号：旧时光大数据

问过的一些公司：头条 x 2，一点咨询，字节，微众，社招，陌陌，头条，触宝，网易，贝壳

参考答案：

下级存储支持事务：Flink可以通过实现两阶段提交和状态保存来实现端到端的一致性语义。

分为以下几个步骤：

1）开始事务（beginTransaction）创建一个临时文件夹，来写把数据写入到这个文件夹里面

2）预提交（preCommit）将内存中缓存的数据写入文件并关闭

3）正式提交（commit）将之前写完的临时文件放入目标目录下。这代表着最终的数据会有一些延迟

4）丢弃（abort）丢弃临时文件

5）若失败发生在预提交成功后，正式提交前。可以根据状态来提交预提交的数据，也可删除预提交的数据。

下级存储不支持事务：

- 具体实现是幂等写入，需要下级存储具有幂等性写入特性。

Flink的端到端Exactly Once

问过的一些公司：腾讯社招

参考答案：

1、Flink的Exactly-Once语义

对于Exactly-Once语义，指的是每一个到来的事件仅会影响最终结果一次。就算机器宕机或者软件崩溃，即没有数据重复，也没有数据丢失。

这里有一个关于checkpoint算法的简要介绍，这对于了解更广的主题来说是十分必要的。

一个checkpoint是Flink的一致性快照，它包括：

- 程序当前的状态
- 输入流的位置

Flink通过一个可配置的时间，周期性的生成checkpoint，将它写入到存储中，例如S3或者HDFS。写入到存储的过程是异步的，意味着Flink程序在checkpoint运行的同时还可以处理数据。

在机器或者程序遇到错误重启的时候，Flink程序会使用最新的checkpoint进行恢复。Flink会恢复程序的状态，将输入流回滚到checkpoint保存的位置，然后重新开始运行。这意味着Flink可以像没有发生错误一样计算结果。

在Flink 1.4.0版本之前，Flink仅保证Flink程序内部的Exactly-Once语义，没有扩展到在Flink数据处理完成后存储的外部系统。

Flink程序可以和不同的接收器（sink）交互，开发者需要有能力在一个组件的上下文中维持Exactly-Once语义。

为了提供端到端Exactly-Once语义，除了Flink应用程序本身的状态，Flink写入的外部存储也需要满足这个语义。也就是说，这些外部系统必须提供提交或者回滚的方法，然后通过Flink的checkpoint来协调。

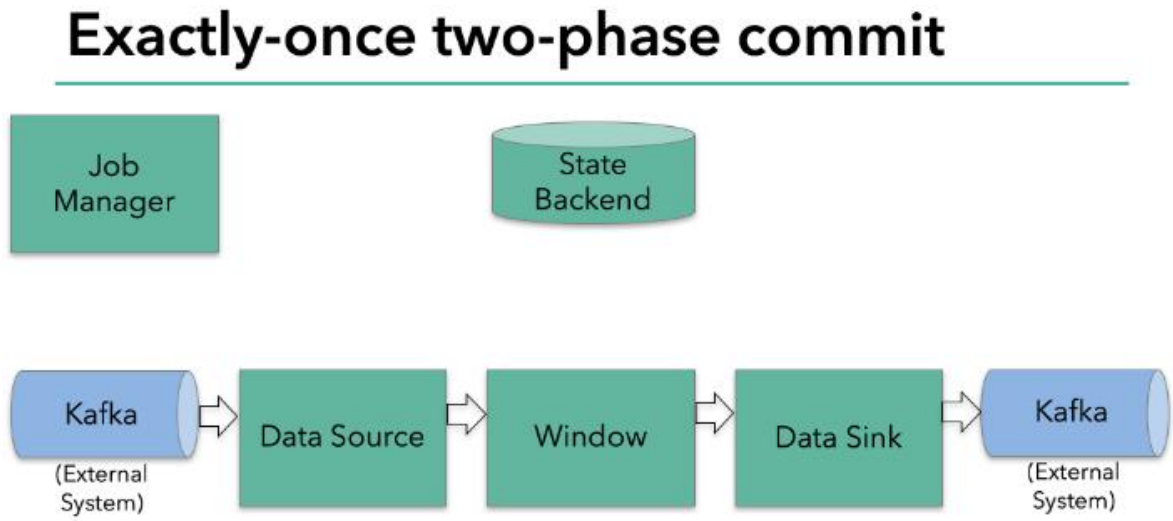
在分布式系统中，协调提交和回滚的通用做法是两阶段提交。**Flink的TwoPhaseCommitSinkFunction使用两阶段提交协议来保证端到端的Exactly-Once语义。**

2、Flink程序端到端的Exactly-Once语义

公众号：旧时光大数据

Kafka是一个流行的消息中间件，经常被拿来和Flink一起使用，Kafka 在最近的0.11版本中添加了对事务的支持。这意味着现在Flink读写Kafka有了必要的支持，使之能提供端到端的Exactly-Once语义。

Flink对端到端的Exactly-Once语义不仅仅局限在Kafka，你可以使用任一输入输出源（source、sink），只要他们提供了必要的协调机制。例如Pravega，来自DELL/EMC的流数据存储系统，通过Flink的TwoPhaseCommitSinkFunction也能支持端到端的Exactly-Once语义。



在这个示例程序中，有：

- 从Kafka读取数据的data source（KafkaConsumer，在Flink中）
- 窗口聚合
- 将数据写回到Kafka的data sink（KafkaProducer，在Flink中）

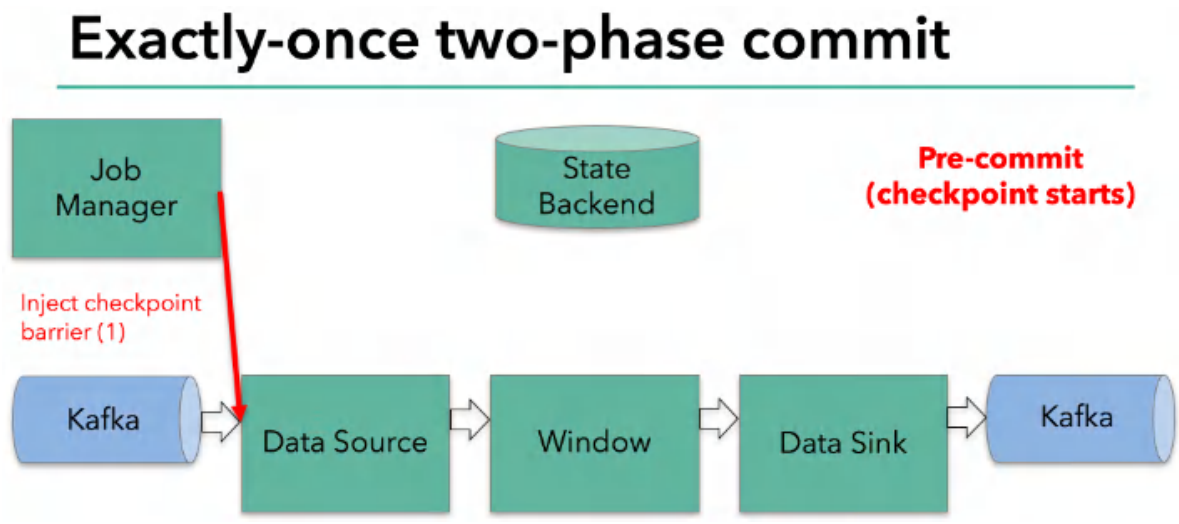
在data sink中要保证Exactly-Once语义，它必须将所有的写入数据通过一个事务提交到Kafka。在两个checkpoint之间，一个提交绑定了所有要写入的数据。

这保证了当出错的时候，写入的数据可以被回滚。

然而在分布式系统中，通常拥有多个并行执行的写入任务，简单的提交和回滚是效率低下的。为了保证一致性，所有的组件必须先达成一致，才能进行提交或者回滚。Flink使用了两阶段提交协议以及预提交阶段来解决这个问题。

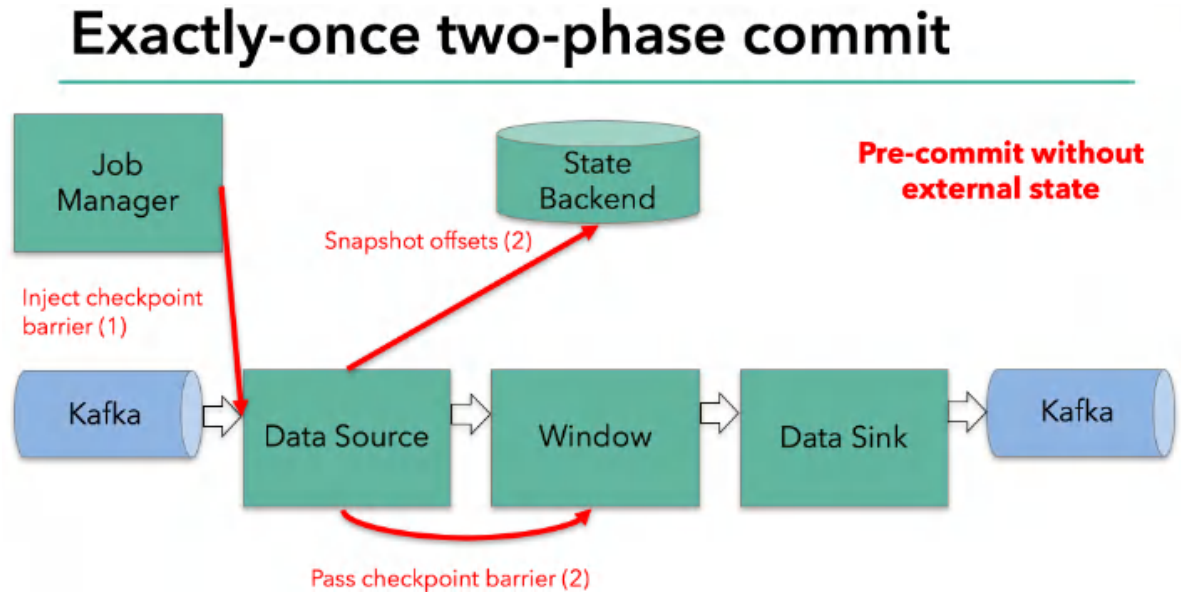
在checkpoint开始的时候，即两阶段提交中的预提交阶段。首先，Flink的JobManager在数据流中注入一个checkpoint屏障（它将数据流中的记录分割开，一些进入到当前的checkpoint，另一些进入下一个checkpoint）。

屏障通过operator传递。对于每一个operator，它将触发operator的状态快照写入到state backend。



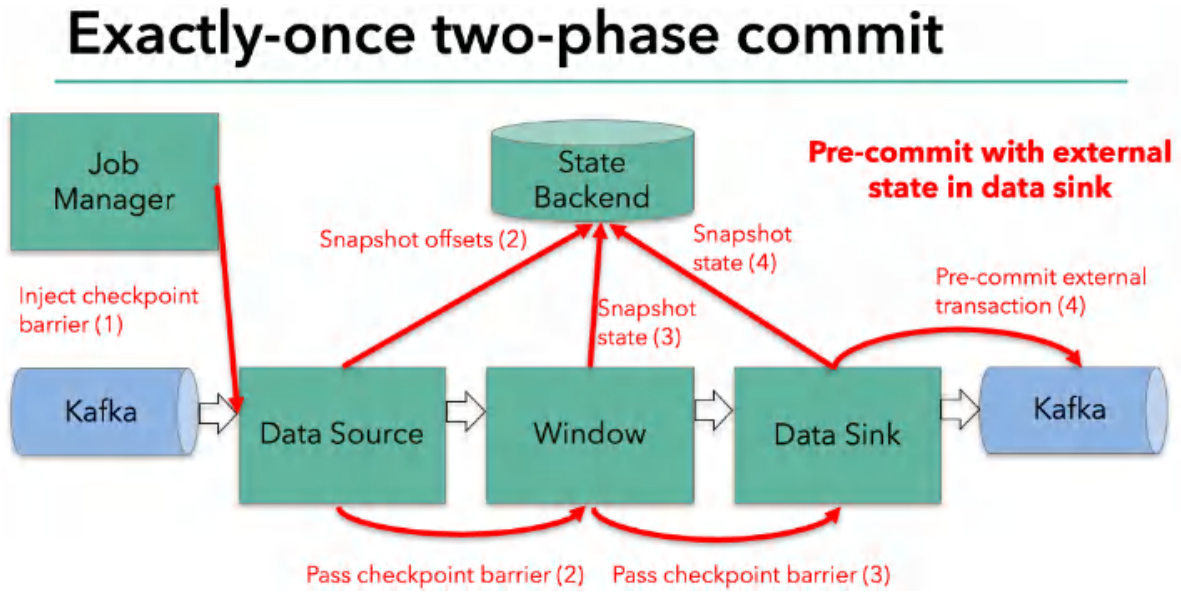
data source保存了Kafka的offset，之后把checkpoint屏障传递到后续的operator。

这种方式仅适用于operator有他的内部状态。内部状态是指，Flink state backends保存和管理的内容-举例来说，第二个operator中window聚合算出来的sum。当一个进程有它的内部状态的时候，除了在checkpoint之前将需要将数据更改写入到state backend，不需要在预提交阶段做其他的动作。在checkpoint成功的时候，Flink会正确的提交这些写入，在checkpoint失败的时候会终止提交。



然而，当一个进程有外部状态的时候，需要用一种不同的方式来处理。外部状态通常由需要写入的外部系统引入，例如Kafka。因此，为了提供Exactly-Once保证，外部系统必须提供事务支持，借此和两阶段提交协议交互。

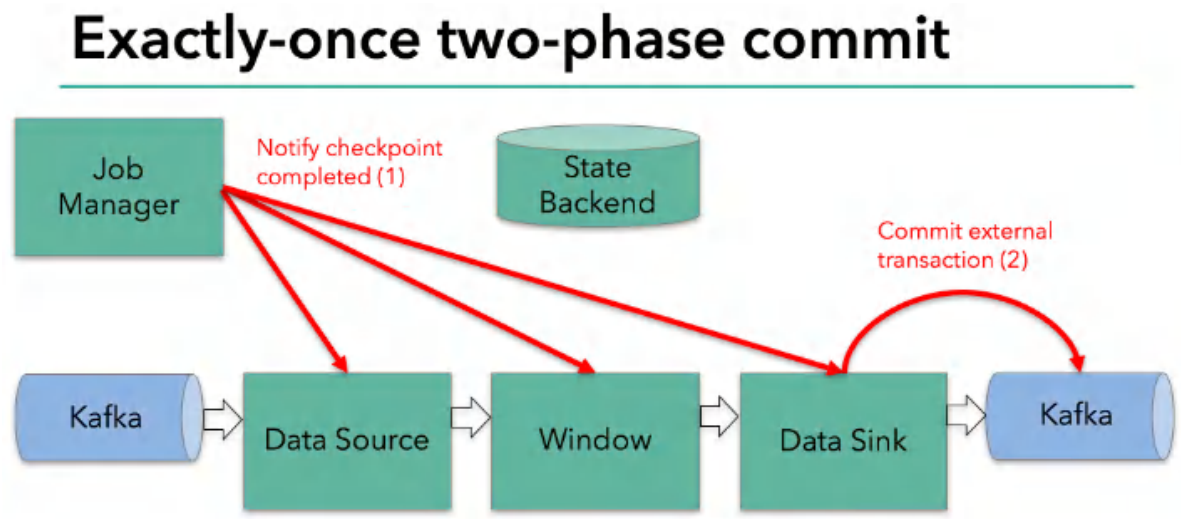
在这个例子中，由于需要将数据写到Kafka，data sink有外部的状态。因此，在预提交阶段，除了将状态写入到state backend之外，data sink必须预提交自己的外部事务。



当checkpoint屏障在所有operator中都传递了一遍，以及它触发的快照写入完成，预提交阶段结束。这个时候，快照成功结束，整个程序的状态，包括预提交的外部状态是一致的。万一出错的时候，我们可以通过checkpoint重新初始化。

下一步是通知所有operator，checkpoint已经成功了。这时两阶段提交中的提交阶段，Jobmanager为程序中的每一个operator发起checkpoint已经完成的回调。data source和window operator没有外部的状态，在提交阶段中，这些operator不会执行任何动作。data sink拥有外部状态，所以通过事务提交外部写入。

公众号：旧时光大数据



对上述的知识点汇总一下：

- 一旦所有的operator完成预提交，就提交一个commit。
- 如果至少有一个预提交失败，其他的都会失败，这时回滚到上一个checkpoint保存的位置。
- 预提交成功后，提交的commit也需要保障最终成功-operator和外部系统需要提供这个保障。如果commit失败了（比如网络中断引起的故障），整个flink程序也因此失败，它会根据用户的重启策略重启，可能还会有一个尝试性的提交。这个过程非常严苛，因为如果提交没有最终生效，会导致数据丢失。

因此，我们可以确定所有的operator同意checkpoint的最终结果：要么都同意提交数据，要么提交被终止然后回滚。

Flink的水印（Watermark），有哪几种?

问过的一些公司：嘉云

参考答案：

水印（Watermark）用于处理乱序事件，而正确地处理乱序事件，通常用Watermark机制结合窗口来实现。
从流处理原始设备产生事件，到Flink读取到数据，再到Flink多个算子处理数据，在这个过程中，会受到网络延迟、数据乱序、背压、Failover等多种情况的影响，导致数据是乱序的。虽然大部分情况下没有问题，但是不得不在设计上考虑此类异常情况，为了保证计算结果的正确性，需要等待数据，这带来了计算的延迟。对于延迟太久的数据，不能无限期地等下去，所以必须有一个机制，来保证特定的时间后一定会触发窗口进行计算，这个触发机制就是Watermark。
在DataStream和Flink Table & SQL模块中，使用了各自的Watermark生成体系。

1、DataStream Watermark生成

通常Watermark在Source Function中生成，如果是并行计算的任务，在多个并行执行的Source Function中，相互独立产生各自的Watermark。而Flink提供了额外的机制，允许在调用DataStream API操作（如map、filter等）之后，根据业务逻辑的需要，使用时间戳和Watermark生成器修改数据记录的时间戳和Watermark。

1）Source Function中生成Watermark

Source Function可以直接为数据元素分配时间戳，同时也会向下游发送Watermark。在Source Function中为数据分配了时间戳

和Watermark就不必在DataStream API中使用了。需要注意的是:如果一个timestamp分配器被使用的话，由源提供的任何Timestamp和Watermark都会被重写。

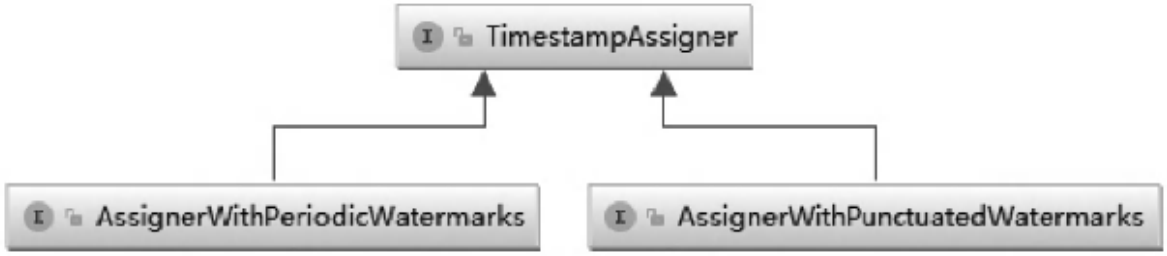
公众号：旧时光大数据

为了通过SourceFunction直接为一个元素分配一个时间戳，SourceFunction需要调用SourceContext中的collectWithTimestamp（...）方法。为了生成Watermark，源需要调用emitWatermark（Watermark）方法，如以下代码所示。

```
@Override
public void run(SourceContext<MyType>ctx)throws Exception {
    while(/* condition */){
        MyType next=getNext();
        //为数据元素赋予时间戳
        ctx.collectwithTimestamp(next,next.getEventTimestamp());
        //生成Watermark并发送给下游
        if(next.hasWatermarkTime()){
            ctx.emitWatermark(new Watermark(next.getWatermarkTime()))
        }
    }
}

```

2）DataStream API中生成Watermark

DataStream API中使用的TimestampAssigner 接口定义了时间戳的提取行为，其有两个不同接口AssignerWithPeriodicWatermarks和AssignerWithPunctuatedWatermarks，分别代表了不同的Watermark生成策略。TimestampAssigner接口体系如下图所示。

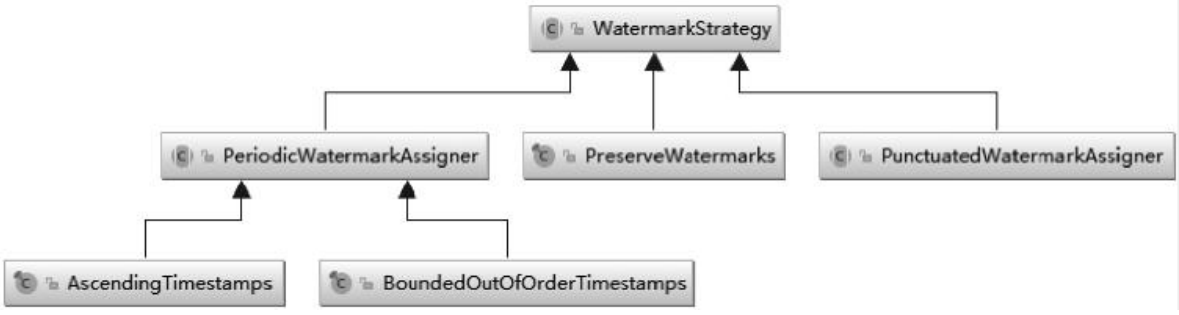


`AssignerWithPeriodicWatermarks` 是周期性生成Watermark策略的顶层抽象接口，该接口的实现类周期性地生成Watermark，而不会针对每一个事件都生成。

`AssignerWithPunctuatedWatermarks` 对每一个事件都会尝试进行Watermark的生成，但是如果生成的Watermark是null或者Watermark小于之前的Watermark，则该Watermark不会发往下游，因为发往下游也不会有任何效果，不会触发任何窗口的执行。

2、Flink SQL Watermark生成

Flink SQL没有DataStram API开发那么灵活，其Watermark的生成主要是在TableSource中完成的，其定义了3类Watermark生成策略。其Watermark生成策略体系如下图所示。



Watermark的生成机制分为如下3类。

1）周期性Watermark策略

公众号：旧时光大数据

周期性Watermark策略在Flink中叫作`PeriodicWatermarkAssigner`，周期性（一定时间间隔或者达到一定的记录条数）地产生一个Watermark。在实际的生产中使用周期性Watermark策略的时候，必须注意时间和数据量，结合时间和积累条数两个维度继续周期性产生Watermark，否则在极端情况下会有很大的延时。

（1）AscendingTimestamps：递增Watermark，作用在Flink SQL中的Rowtime属性上，Watermark=当前收到的数据元素的最大时间戳-1，此处减1的目的是确保有最大时间戳的事件不会被当做迟到数据丢弃。

（2）BoundedOutOfOrderTimestamps：固定延迟Watermark，作用在Flink SQL的Rowtime属性上，Watermark=当前收到的数据元素的最大时间戳-固定延迟。

2）每事件Watermark策略

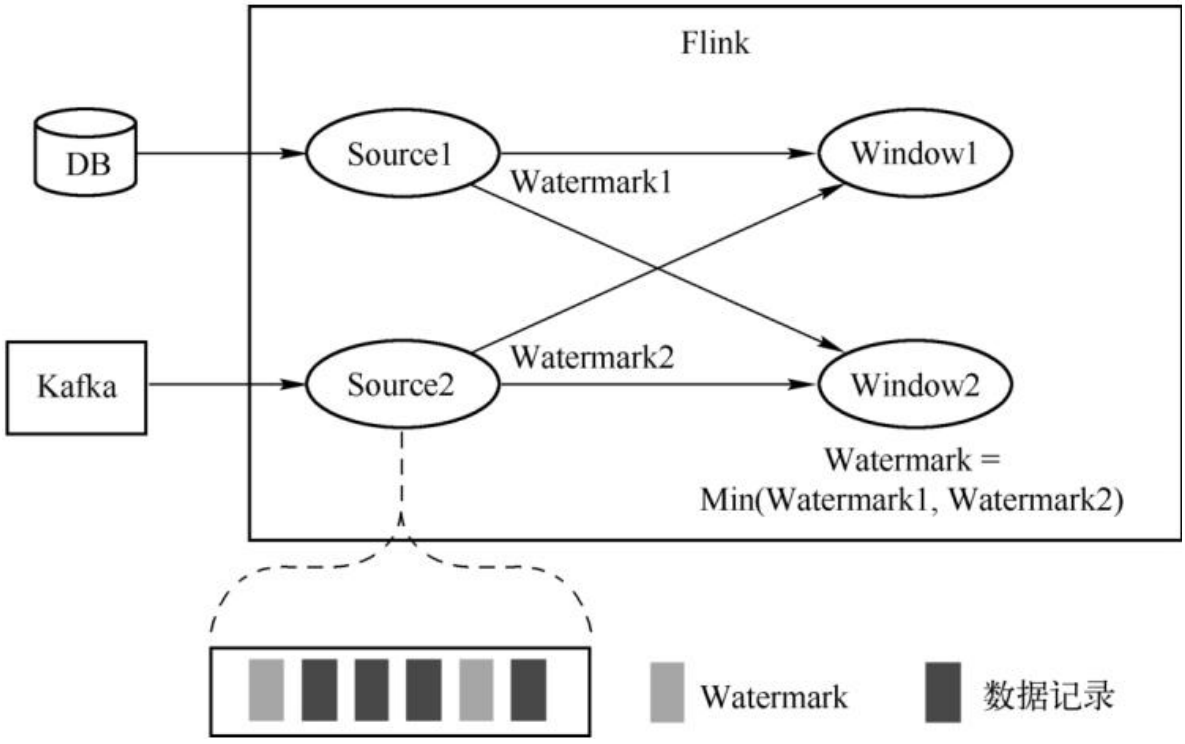
每事件Watermark策略在Flink中叫作`PuntuatedWatamarkAssigner`，数据流中每一个递增的EventTime都会产生一个Watermark。在实际的生产中Punctuated方式在TPS很高的场景下会产生大量的Watermark，在一定程度上会对下游算子造成压力，所以只有在实时性要求非常高的场景下才会选择Punctuated的方式进行Watermark的生成。

3）无为策略

无为策略在Flink中叫作`PreserveWatermark`。在Flink中可以使用DataStream API和Table & SQL混合编程，所以Flink SQL中不设定Watermark策略，使用底层DataStream中的Watermark策略也是可以的，这时Flink SQL的Table Source中不做处理。

3、多流的Watermark

在实际的流计算中一个作业中往往会处理多个Source的数据，对Source的数据进行GroupBy分组，那么来自不同Source的相同key值会shuffle到同一个处理节点，并携带各自的Watermark，Flink内部要保证Watermark保持单调递增，多个Source的Watermark汇聚到一起时可能不是单调自增的，对于这样的情况，Flink的内部处理如下图所示。



Flink内部实现每一个边上只能有一个递增的Watermark，当出现多流携带EventTime汇聚到一起（GroupBy或Union）时，Apache Flink会选择所有流入的EventTime中最小的一个向下游流出，从而保证Watermark的单调递增和数据的完整性。

公众号：旧时光大数据

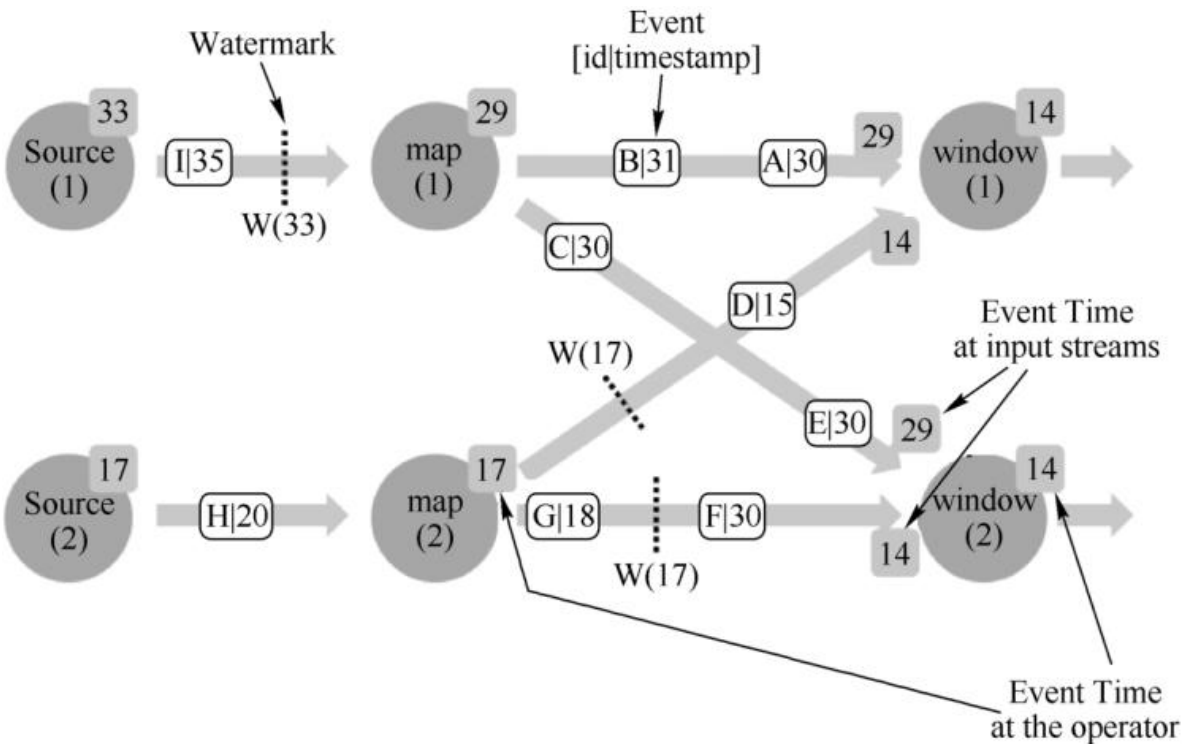
Watermark是在Source Function中生成或者在后续的DataStream API中生成的。Flink作业一般是并行执行的，作业包含多个Task，每个Task运行一个或一组算子（OperatorChain）实例，Task在生成Watermark的时候是相互独立的，也就是说在作业中存在多个并行的Watermark。

Watermark在作业的DAG从上游向下游传递，算子收到上游Watermark后会更新其Watermark。如果新的Watermark大于算子的当前Watermark，则更新算子的Watermark为新Watermark，并发送给下游算子。

某些算子会有多个上游输入，如Union或keyBy、partition之后的算子。在Flink的底层执行模型上，多流输入会被分解为多个双流输入，所以对于多流Watermark的处理也就是双流Watermark的处理，无论是哪一个流的Watermark进入算子，都需要跟另一个流的当前算子进行比较，选择较小的Watermark，即Min(input1Watermark,intput2Watermark)，与算子当前的Watermark比较，如果大于算子当前的Watermark，则更新算子的Watermark为新的Watermark，并发送给下游，如以下代码所示。

```
//AbstractStreamOperator.java
public voidprocessWatermark1(Watermark mark)throws Exception {
    inputlWatermark =mark.getTimestamp();
    longnewMin=Math.min(input1Watermark,input2Watermark);
    if(newMin >combinedWatermark){
        combinedWatermark=newMin;
        processWatermark(new Watermark(combinedwatermark));
    }
}

public voidprocessWatermark2(Watermark mark)throws Exception {
    input2Watermark=mark.getTimestamp();
    longnewMin=Math.min(inputlWatermark,input2Watermark);
    if(newMin>combinedWatermark){
        combinedwatermark=newMin;
        processWatermark(new Watermark(combinedwatermark));
    }
}

```

如下图所示，多流Watermark中使用了事件时间。



公众号：旧时光大数据

在上图中，Source算子产生各自的Watermark，并随着数据流流向下游的map算子，map算子是无状态计算，所以会将Watermark向下透传。window算子收到上游两个输入的Watermark后，选择其中较小的一个发送给下游，window（1）算子比较Watermark 29和Watermark 14，选择Watermark 14作为算子当前Watermark，并将Watermark 14发往下游，window（2）算子也采用相同的逻辑。

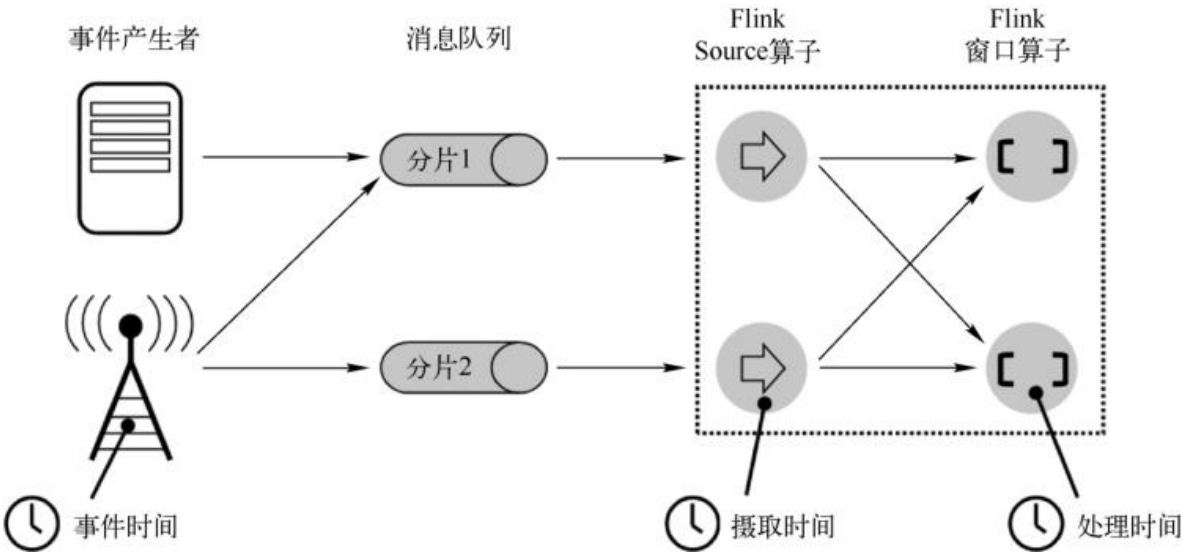
Flink的时间语义

问过的一些公司：电信云计算，网易

参考答案：

在Flink中定义了3种时间类型：事件时间（Event Time）、处理时间（Processing Time）和摄取时间（Ingestion Time）。

3种时间类型如下图所示。



1）事件时间

事件时间指事件发生时的时间，一旦确定之后再也不会改变。例如，事件被记录在日志文件中，日志中记录的时间戳就是事件时间。通过事件时间能够还原出来事件发生的顺序。

使用事件时间的好处是不依赖操作系统的时钟，无论执行多少次，可以保证计算结果是一样的，但计算逻辑稍微复杂，需要从每一条记录中提取时间戳。

2）处理时间

处理时间指消息被计算引擎处理的时间，以各个计算节点的本地时间为准。例如，在物理节点1处理时，处理时间（即当前系统时间）为2019-02-05 12：00：00，然后交给下游的计算节点进程处理，此时的处理时间（即当前系统时间）为2019-02-05 12：00：01。可以看到处理时间是在不停变化的。使用处理时间依赖于操作系统的时钟，重复执行基于窗口的统计作业，结果可能是不同的。处理时间的计算逻辑非常简单，性能好于事件时间，延迟低于事件时间，只需要获取当前系统的时间戳即可。

3）摄取时间

摄取时间指事件进入流处理系统的时间，对于与一个事件来说，使用其被读取的那一刻的时间戳作作为摄取时间。

摄取时间一般使用得较少，从处理机制上来说，其类似于事件时间，在作业异常重启执行的时候，也无法避免使用处理时间的结果不准确的问题。一般来说，若在数据记录中没有记录时间，又想使用事件时间机制来处理记录，会选择使用摄取时间。

公众号：旧时光大数据

在Flink应用中可以使用这3种时间类型，其中最常用的是事件时间和处理时间，如以下使用TimeCharacteristic设置Flink使用的时间类型代码所示。

```
finalStreamExecutionEnvironmentenv=StreamExecutionEnvironment.
getExecutionEnvironment();
//使用处理时间
env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime);
//其他类型的时间
//env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime)
//env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
DataStream<MyEvent>stream =env.addSource(new FlinkKafkaConsumer09<My Event>
(topic,schema, props));
stream
    .keyBy((event)->event.getUser())
    .timeWindow(Time.hours(1))
    .reduce((a, b)->a.add(b))
    .addSink(...);

```

Flink相比于其它流式处理框架的优点?

1、同时支持高吞吐、低延迟、高性能

Flink是目前开源社区中唯一一套集高吞吐、低延迟、高性能三者于一身的分布式流式数据处理框架。Apache Spark也只能兼顾高吞吐和高性能特点，主要是因为Spark Streaming流式计算中无法做到低延迟保障；而流式计算框架Apache Storm只能支持低延迟和高性能特性，但是无法满足高吞吐的要求。而满足高吞吐、低延迟、高性能这三个目标对分布式流式计算框架来说是非常重要的。

2、支持事件时间（Event Time）

在流式计算领域中，窗口计算的地位举足轻重，但目前大多数框架窗口计算采用的都是系统时间（Process Time），也是事件传输到计算框架处理时，系统主机的当前时间。Flink能够支持基于事件时间（Event Time)语义进行窗口计算，也就是使用事件产生的时间，这种基于事件驱动的机制使得事件即使乱序到达，流系统也能够计算出精确的结果，保证了事件原本产生时的时序性，尽可能避免网络传输或硬件系统的影响。

3、支持有状态计算

Flink在1.4版本中实现了状态管理，所谓状态就是在流式计算过程中将算子的中间结果数据保存在内存或文件系统中，等下一个事件进入算子后可以从之前的状态中获取中间结果，计算当前的结果，从而无需每次都基于全部的原始数据来统计结果，这种方式极大地提升了系统的性能，并降低了数据计算过程的资源消耗。对于数据量大且运算逻辑非常复杂的流式计算场景，有状态计算发挥了非常重要的作用。

4、支持高度灵活的窗口（Window）操作

在流处理应用中，数据是连续不断的，需要通过窗口的方式对流数据进行一定范围的聚合计算，例如统计在过去1分钟内有多少用户点击某一网页，在这种情况下，我们必须定义一个窗口，用来收集最近一分钟内的数据，并对这个窗口的数据进行再计算。

Flink将窗口划分为基于Time、Count、Session，以及Data-Driven等类型的窗口操作，窗口可以用灵活的出发条件定制化来达到对复杂的流传输模式的支持，用户可以定义不同的窗口出发机制来满足不同的需求。

5、基于轻量级分布式快照（CheckPoint）实现的容错

公众号：旧时光大数据

Flink能够分布式运行在上千个节点上，将一个大型计算任务的流程拆解成晓得计算过程，然后将task分布到并行节点上处理。在任务执行过程中，能够自动发现事件处理过程中的错误而导致数据不一致的问题，比如：节点宕机、网络传输问题，或是由于用户升级或修复问题而导致计算服务重启等。在这些情况下，通过基于分布式快照技术的Checkpoints，将执行过程中的状态信息进行持久化恢复，以确保数据在处理过程中的一致性（Exactly-Once）。

6、基于JVM实现独立的内存管理

内存管理是所有计算框架需要重点考虑的部分，尤其对于计算量比较大的计算场景，数据在内存中该如何进行管理显得至关重要。针对内存管理，FLink实现了自身管理内存的机制，尽可能减少JVM GC对系统的影响。另外，FLink通过序列化/反序列化方法将所有的数据对象转换成二进制在内存中存储，降低数据存储的大小的同事，能够更加有效地对内存空间进行利用，降低GC带来的性能下降或任务异常的风险，因此Flink较其他分布式处理的框架会显得更加稳定，不会因为JVM GC等问题而影响整个应用的运行

7、Savepoints（保存点）

对于7*24小时运行的流式应用，数据源源不断的接入，在一段时间内应用的终止有可能导致数据的丢失或者极端结果的不准确，例如进行集群版本的升级、停机运维操作等操作。值得一提的是，FLink通过Save Points技术将任务执行的快照保存在存储介质上，当任务重启的时候可以直接从事先保存的Save Points恢复原有的计算状态，是的任务继续按照停机之前的状态运行，Save Points技术可以烫用户更好地管理和运维实时流式应用。

Flink和Spark的区别？什么情况下使用Flink？有什么优点？

几种流式框架的对比：

1、Flink和Spark的区别

数据模型

- Flink基本数据模型是数据流，以及事件序列。
- Spark采用RDD模型，Spark Streaming的DStream实际上也就是一组组小批数据RDD的集合。

运行时架构

- Flink是标准的流执行模式，一个事件在一个节点处理完后可以直接发往下一个节点进行处理。
- Spark是批计算，将DAG划分为不同的Stage，一个完成后才可以计算下一个。

2、Flink应用场景

在实际生产的过程中，大量数据在不断地产生，例如金融交易数据、互联网订单数据、GPS定位数据、传感器信号、移动终端产生的数据、通信信号数据等，以及我们熟悉的网络流量监控、服务器产生的日志数据，这些数据最大的共同点就是实时从不同的数据源中产生，然后再传输到下游的分析系统。针对这些数据类型主要包括实时智能推荐、复杂事件处理、实时欺诈检测、实时数仓与ETL类型、流数据分析类型、实时报表类型等实时业务场景，而Flink对于这些类型的场景都有着非常好的支持。

1）实时智能推荐

智能推荐会根据用户历史的购买行为，通过推荐算法训练模型，预测用户未来可能会购买的物品。对个人来说，推荐系统起着信息过滤的作用，对Web/App服务端来说，推荐系统起着满足用户个性化需求，提升用户满意度的作用。推荐系统本身也在飞速发展，除了算法越来越完善，对时延的要求也越来越苛刻和实时化。利用Flink流计算帮助用户构建更加实时的智能推荐系统，对用户行为指标进行实时计算，对模型进行实时更新，对用户指标进行实时预测，并将预测的信息推送给Wep/App端，帮助用户获取想要的商品信息，另一方面也帮助企业提升销售额，创造更大的商业价值。

2）复杂事件处理

对于复杂事件处理，比较常见的案例主要集中于工业领域，例如对车载传感器、机械设备等实时故障检测，这些业务类型通常数据量都非常大，且对数据处理的时效性要求非常高。通过利用Flink提供的CEP（复杂事件处理）进行事件模式的抽取，同时应用Flink的Sql进行事件数据的转换，在流式系统中构建实时规则引擎，一旦事件触发报警规则，便立即将告警结果传输至下游通知系统，从而实现对设备故障快速预警监测，车辆状态监控等目的。

3）实时欺诈检测

在金融领域的业务中，常常出现各种类型的欺诈行为，例如信用卡欺诈、信贷申请欺诈等，而如何保证用户和公司的资金安全，是来近年来许多金融公司及银行共同面对的挑战。随着不法分子欺诈手段的不断升级，传统的反欺诈手段已经不足以解决目前所面临的问题。以往可能需要几个小时才能通过交易数据计算出用户的行为指标，然后通过规则判别出具有欺诈行为嫌疑的用户，再进行案件调查处理，在这种情况下资金可能早已被不法分子转移，从而给企业和用户造成大量的经济损失。而运用Flink流式计算技术能够在毫秒内就完成对欺诈判断行为指标的计算，然后实时对交易流水进行规则判断或者模型预测，这样一旦检测出交易中存在欺诈嫌疑，则直接对交易进行实时拦截，避免因为处理不及时而导致的经济损失。

4）实时数仓与ETL

结合离线数仓，通过利用流计算诸多优势和SQL灵活的加工能力，对流式数据进行实时清洗、归并、结构化处理，为离线数仓进行补充和优化。另一方面结合实时数据ETL处理能力，利用有状态流式计算技术，可以尽可能降低企业由于在离线数据计算过程中调度逻辑的复杂度，高效快速地处理企业需要的统计结果，帮助企业更好地应用实时数据所分析出来的结果。

5）流数据分析

实时计算各类数据指标，并利用实时结果及时调整在线系统相关策略，在各类内容投放、无线智能推送领域有大量的应用。流式计算技术将数据分析场景实时化，帮助企业做到实时化分析Web应用或者App应用的各项指标，包括App版本分布情况、Crash检测和分布等，同时提供多维度用户行为分析，支持日志自主分析，助力开发者实现基于大数据技术的精细化运营、提升产品质量和体验、增强用户黏性。

6）实时报表分析

实时报表分析是近年来很多公司采用的报表统计方案之一，其中最主要的应用便是实时大屏展示。利用流式计算实时得出的结果直接被推送到前端应用，实时显示出重要指标的变换情况。最典型的案例便是淘宝的双十一活动，每年双十一购物节，除疯狂购物外，最引人注目的就是天猫双十一大屏不停跳跃的成交总额。在整个计算链路中包括从天猫交易下单购买到数据采集、数据计算、数据校验，最终落到双十一大屏上展现的全链路时间压缩在5秒以内，顶峰计算性能高达数三十万笔订单/秒，通过多条链路流计算备份确保万无一失。而在其他行业，企业也在构建自己的实时报表系统，让企业能够依托于自身的业务数据，快速提取出更多的数据价值，从而更好地服务于企业运行过程中。

3、优点

参考上一题

Flink backPressure反压机制，指标监控你是怎么做的?

问过的一些公司：星环科技，蔚来(2021.07)

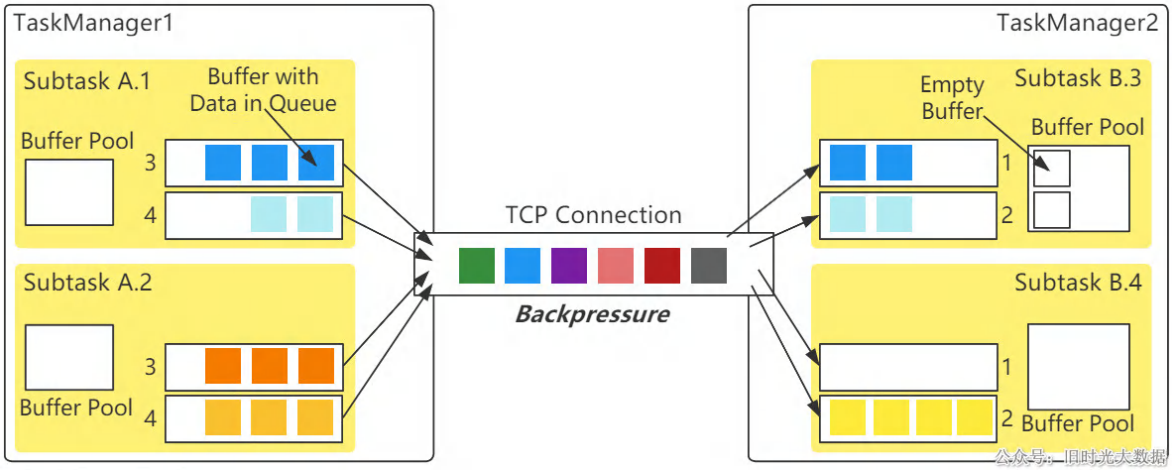
参考答案：

背压是**指系统在一个临时负载峰值期间接收数据的速率大于其处理速率的一种场景**（备注:就是处理速度慢，接收速度快，系统处理不了接收的数据）。许多日常情况都会导致背压。例如，垃圾回收卡顿可能导致流入的数据堆积起来，或者数据源可能出现发送数据过快的峰值。如果处理不当，背压会导致资源耗尽，甚至导致数据丢失。

1、Flink反压

公众号：旧时光大数据

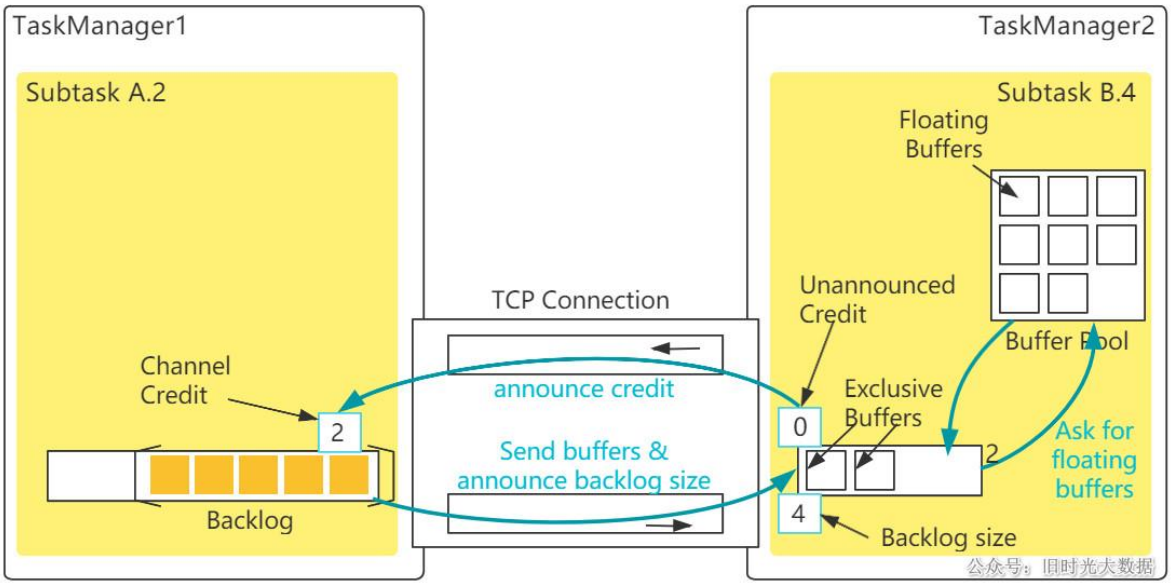
每个子任务都有自己的本地缓存池，收到的数据以及发出的数据，都会序列化之后，放入到缓冲池里。然后，两个TaskManager之间，只会建立一条物理链路（底层使用Netty通讯），所有子任务之间的通讯，都由这条链路承担。



当任何一个子任务的发送缓存（不管是子任务自己的本地缓存，还是底层传输时Netty的发送缓存）耗尽时，发送方就会被阻塞，产生背压；同样，**任何任务接收数据时，如果本地缓存用完了，都会停止从底层Netty那里读取数据，这样很快上游的数据很快就会占满下游的底层接收缓存，从而背压到发送端，形成对上游所有的任务的背压。**

很显然，这种思路有个明显的问题，任何一个下游子任务的产生背压，都会影响整条TaskManager之间的链路，导致全链路所有子任务背压。比如上图的B.3子任务，此时还有处理能力，但也无法收到数据。

Flink - 基于Credit - based Flow Control的背压



为了解决上节的单任务背压影响全链路的问题，在Flink 1.5之后，引入了Credit-based Flow Control，基于信用点的流量控制。

这种方法，首先把每个子任务的本地缓存分为两个部分，独占缓存（Exclusive Buffers）和浮动缓存（Floating Buffers）；

然后，独占缓存的大小作为信用点发给数据发送方，发送方会按照不同的子任务分别记录信用点，并发送尽可能多数据给接收方，发送后则降低对应信用点的大小；

当信用点为0时，则不再发送，起到背压的作用。在发送数据的同时，发送方还会把队列中暂存排队的数据量发给接收方，接收方收到后，根据本地缓存的大小，决定是否去浮动缓存里请求更多的缓存来加速队列的处理，起到动态控制流量的作用。整个过程参考上图。

通过这样的设计，就实现了任务级别的背压：任意一个任务产生背压，只会影响这个任务，并不会对TaskManger上的其它任务造成影响。

2、Flink监控指标

Flink任务提交到集群后，接下来就是对任务进行有效的监控。Flink将任务监控指标主要分为系统指标和用户指标两种：系统指标主要包括Flink集群层面的指标，例如CPU负载，各组件内存使用情况等；用户指标主要包括用户在任务中自定义注册的监控指标，用于获取用户的业务状况等信息。Flink中的监控指标可以通过多种方式获取，例如可以从Flink UI中直接查看，也可以通过Rest Api或Reporter获取。

Flink如何保证一致性?

问过的一些公司：网易，好未来，字节(2021.07)

参考答案：

Flink的检查点和恢复机制定期的会保存应用程序状态的一致性检查点。在故障的情况下，应用程序的状态将会从最近一次完成的检查点恢复，并继续处理。尽管如此，可以使用检查点来重置应用程序的状态无法完全达到令人满意的一致性保证。相反，source和sink的连接器需要和Flink的检查点和恢复机制进行集成才能提供有意义的一致性保证。

为了给应用程序提供恰好处理一次语义的状态一致性保证，应用程序的source连接器需要能够将source的读位置重置到之前保存的检查点位置。当处理一次检查点时，source操作符将会把source的读位置持久化，并在恢复的时候从这些读位置开始重新读取。支持读位置的检查点的source连接器一般来说是基于文件的存储系统，如：文件流或者Kafka source（检查点会持久化某个正在消费的topic的读偏移量）。如果一个应用程序从一个无法存储和重置读位置的source连接器摄入数据，那么当任务出现故障的时候，数据就会丢失。也就是说我们只能提供at-most-once）的一致性保证。

Fink的检查点和恢复机制和可以重置读位置的source连接器结合使用，可以保证应用程序不会丢失任何数据。尽管如此，应用程序可能会发出两次计算结果，因为从上一次检查点恢复的应用程序所计算的结果将会被重新发送一次（一些结果已经发送出去了，这时任务故障，然后从上一次检查点恢复，这些结果将被重新计算一次然后发送出去）。所以，可重置读位置的source和Flink的恢复机制不足以提供端到端的恰好处理一次语义，即使应用程序的状态是恰好处理一次一致性级别。

Flink 中的一个大的特性就是exactly-once的特性，我们在一般的流处理程序中，会有三种处理语义

- AT-MOST-ONCE(最多一次)：当故障发生的时候，什么都不干。就是说每条消息就只消费一次。
- AT-LEAST-ONCE(至少一次)：为了确保数据不丢失，确保每个时间都得到处理，一些时间可能会被处理多次。
- EXACTLY-ONCE(精确一次)：每个时间都精确处理一次

端到端的保证：

- 内部保证--- checkpoint
- source端---可重设数据的读取位置
- sink端---从故障恢复时，数据不会重复写入外部系统

Flink(checkpoint)和source端(Kafka)可以保证不出问题。但一个志在提供端到端恰好处理一次语义一致性的应用程序需要特殊的sink连接器。sink连接器可以在不同的情况下使用两种技术来达到恰好处理一次一致性语义：幂等性写入和事务性写入。

1、幂等性写入

一个幂等操作无论执行多少次都会返回同样的结果。例如，重复的向hashmap中插入同样的key-value对就是幂等操作，因为头一次插入操作之后所有的插入操作都不会改变这个hashmap，因为hashmap已经包含这个key-value对了。另一方面，append操作就不是幂等操作了，因为多次append同一个元素将会导致列表每次都会添加一个元素。在流处理程序中，幂等写入操作是很有意思的，因为幂等写入操作可以执行多次但不改变结果。所以它们可以在某种程度上缓和Flink检查点机制带来的重播计算结果的效应。

需要注意的是，依赖于幂等性sink来达到exactly-once语义的应用程序，必须保证在从检查点恢复以后，它将会覆盖之前已经写入的结果。例如，一个包含有sink操作的应用在sink到一个key-value存储时必须保证它能够确定的计算出将要更新的key值。同时，从Flink程序sink到的key-value存储中读取数据的应用，在Flink从检查点恢复的过程中，可能会看到不想看到的结果。当重播开始时，之前已经发出的计算结果可能会被更早的结果所覆盖（因为在恢复过程中）。所以，一个消费Flink程序输出数据的应用，可能会观察到时间回退，例如读到了比之前小的计数。也就是说，当流处理程序处于恢复过程中时，流处理程序的结果将处于不稳定的状态，因为一些结果被覆盖掉，而另一些结果还没有被覆盖。一旦重播完成，也就是说应用程序已经通过了之前出故障的点，结果将会继续保持一致性。

2、事务性写入

实现端到端的恰好处理一次一致性语义的方法基于事务性写入。其思想是只将最近一次成功保存的检查点之前的计算结果写入到外部系统中去。这样就保证了在任务故障的情况下，端到端恰好处理一次语义。应用将被重置到最近一次的检查点，而在这个检查点之后并没有向外部系统发出任何计算结果。通过只有当检查点保存完成以后再写入数据这种方法，事务性的方法将不会遭受幂等性写入所遭受的重播不一致的问题。尽管如此，事务性写入却带来了延迟，因为只有在检查点完成以后，我们才能看到计算结果。

Flink提供了两种构建模块来实现事务性sink连接器：write-ahead-log（WAL，预写式日志）sink和两阶段提交sink。WAL式sink将会把所有计算结果写入到应用程序的状态中，等接到检查点完成的通知，才会将计算结果发送到sink系统。因为sink操作会把数据都缓存在状态后段，所以WAL可以使用在任何外部sink系统上。尽管如此，WAL还是无法提供刀枪不入的恰好处理一次语义的保证，再加上由于要缓存数据带来的状态后段的状态大小的问题，WAL模型并不十分完美。

与之形成对比的，2PC sink需要sink系统提供事务的支持或者可以模拟出事务特性的模块。对于每一个检查点，sink开始一个事务，然后将所有的接收到的数据都添加到事务中，并将这些数据写入到sink系统，但并没有提交（commit）它们。当事务接收到检查点完成的通知时，事务将被commit，数据将被真正的写入sink系统。这项机制主要依赖于一次sink可以在检查点完成之前开始事务，并在应用程序从一次故障中恢复以后再commit的能力。

2PC协议依赖于Flink的检查点机制。检查点屏障是开始一个新的事务的通知，所有操作符自己的检查点成功的通知是它们可以commit的投票，而作业管理器通知一个检查点成功的消息是commit事务的指令。于WAL sink形成对比的是，2PC sinks依赖于sink系统和sink本身的实现可以实现恰好处理一次语义。更多的，2PC sink不断的将数据写入到sink系统中，而WAL写模型就会有之前所述的问题。

事务写的方式能提供端到端的Exactly-Once一致性，它的代价也是非常明显的，就是牺牲了延迟。输出数据不再是实时写入到外部系统，而是分批次地提交。目前来说，没有完美的故障恢复和Exactly-Once保障机制，对于开发者来说，需要在不同需求之间权衡。

Flink支持JobMaster的HA啊？原理是怎么样的？

问过的一些公司：Shopee

参考答案：

1、JobManager 高可用(HA)

jobManager协调每个flink任务部署。它负责任务调度和资源管理。

公众号：旧时光大数据

默认情况下，每个flink集群只有一个JobManager，这将导致一个单点故障(SPOF)：如果JobManager挂了，则不能提交新的任务，并且运行中的程序也会失败。

使用JobManager HA，集群可以从JobManager故障中恢复，从而避免SPOF(单点故障) 。 用户可以在standalone或 YARN集群 模式下，配置集群高可用。

Standalone集群的高可用

Standalone模式（独立模式）下JobManager的高可用性的基本思想是，任何时候都有一个 Master JobManager ，并且多个Standby JobManagers 。 Standby JobManagers可以在Master JobManager 挂掉的情况下接管集群成为Master JobManager。 这样保证了没有单点故障，一旦某一个Standby JobManager接管集群，程序就可以继续运行。 Standby JobManager和Master JobManager实例之间没有明确区别。每个JobManager都可以成为Master或Standby节点

Yarn 集群高可用

flink on yarn的HA 其实主要是利用yarn自己的job恢复机制

如何确定Flink任务的合理并行度?

问过的一些公司公司：社招

参考答案：

task的parallelism可以在Flink的不同级别上指定。

四种级别是：算子级别、执行环境（ExecutionEnvironment）级别、客户端（命令行）级别、配置文件（flink-conf.yaml）级别。

- 每个operator、data source或者data sink都可以通过调用setParallelism()方法来指定
- 运行环境的默认并发数可以通过调用setParallelism()方法来指定。env.setParallelism(3);运行环境的并发数可以被每个算子确切的并发数配置所覆盖。
- 对于CLI客户端，并发参数可以通过-p来指定
- 影响所有运行环境的系统级别的默认并发度可以在./conf/flink-conf.yaml的parallelism.defaul项中指定。不建议。

当然，也可以设置最大的并行度，通过调用setMaxParallelism()方法来设置最大并发度。

Flink如何确定TaskManager个数：Job的最大并行度除以每个TaskManager分配的任务槽数。

Flink on YARN时，TaskManager的数量就是：max(parallelism) / yarnslots（向上取整）。

例如，一个最大并行度为10，每个TaskManager有两个任务槽的作业，就会启动5个TaskManager

Flink任务如何实现端到端一致?

问过的一些公司：社招

参考答案：

Source端：数据从上游进入Flink，必须保证消息严格一次消费。同时Source 端必须满足可重放（replay）。否则 Flink 计算层收到消息后未计算，却发生 failure 而重启，消息就会丢失。

Flink计算层：利用 Checkpoint 机制，把状态数据定期持久化存储下来，Flink程序一旦发生故障的时候，可以选择状态点恢复，避免数据的丢失、重复。

公众号：旧时光大数据

Sink端： Flink将处理完的数据发送到Sink端时，通过 **两阶段提交协议，** 即TwoPhaseCommitSinkFunction 函数。该 SinkFunction 提取并封装了两阶段提交协议中的公共逻辑，保证Flink 发送Sink端时实现严格一次处理语义。 **同时：** Sink端必须支持**事务机制**，能够进行数据**回滚**或者满足**幂等性。**

回滚机制： 即当作业失败后，能够将部分写入的结果回滚到之前写入的状态。

幂等性： 就是一个相同的操作，无论重复多少次，造成的结果和只操作一次相等。即当作业失败后，写入部分结果，但是当重新写入全部结果时，不会带来负面结果，重复写入不会带来错误结果。

Flink如何处理背（反）压?

可回答：Flink的反压机制，怎么处理反压

问过的一些公司：微众，网易，触宝

参考答案：

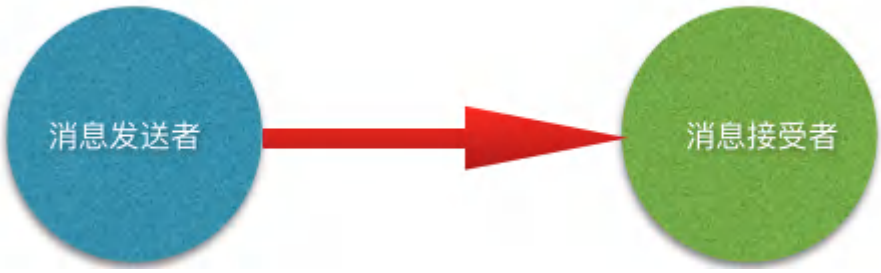
1、什么原因导致背压?

流系统中消息的处理速度跟不上消息的发送速度，导致消息的堆积。如果系统能感知消息堆积，并调整消息发送的速度，使消息的处理速度和发送速度相协调就是有背压感知的系统。

背压如果不能得到正确地处理，可能会导致资源被耗尽或者甚至出现更糟的情况导致数据丢失。flink就是一个有背压感知的基于流的分布式消息处理系统。

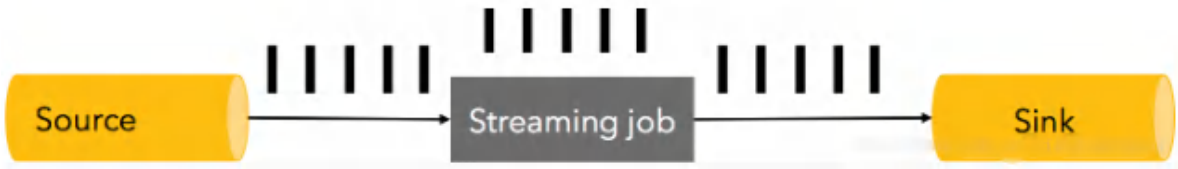
如下图：

- 消息发送的太快，消息接受的太慢，产生消息拥堵。
- 发生消息拥堵后，系统会自动降低消息发送的速度。

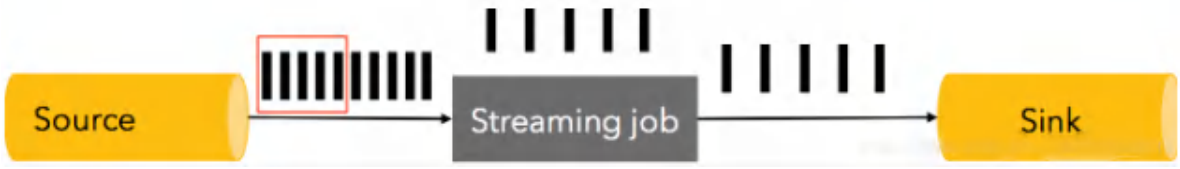


举例说明：

正常情况下：消息处理速度>=消息的发送速度，不会发送消息拥堵，系统运行流畅



异常情况下：消息处理速度<消息的发送速度，发生消息堵塞，系统运行不流畅



消息拥堵可以采用两种方案：

- 将拥堵的消息直接删除，将导致数据丢失，在精确度要求高的场景非常不合适。

公众号：旧时光大数据

- 将拥堵的消息缓存起来，并告知消息发送者减缓消息发送的速度。

将消息缓存起来，并将缓冲区持久化，以方便在处理失败的情况下进行数据重复。有些source本身提供持久化机制，可以优先考虑。例如Kafka就是一个很不错的选择，可以背压从sink到source的整个pipeline，同时对source进行限流来适配整合pipeline中最慢组件的速度，从而获得系统的稳定状态。

2、Flink中的背压

Flink 没有使用任何复杂的机制来解决背压问题，因为根本不需要那样的方案！**它利用自身作为纯数据流引擎的优势来优雅地响应背压问题。**

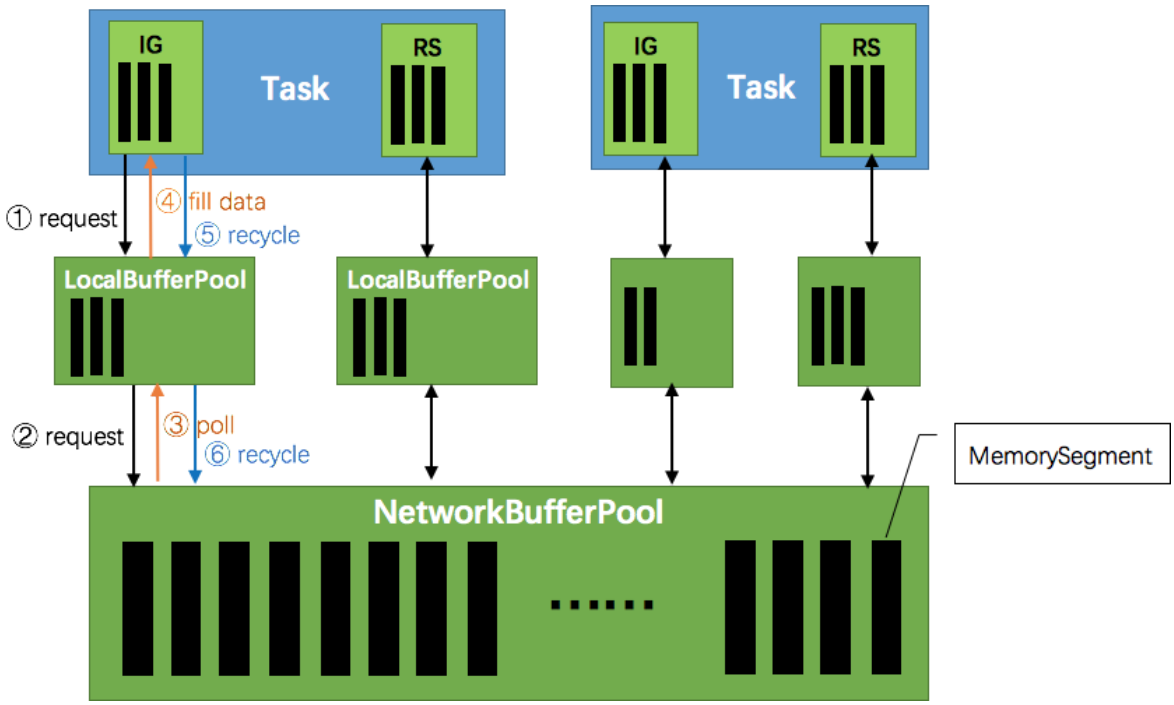
Flink 在运行时主要由 operators 和 streams 两大组件构成。每个 operator 会消费中间态的流，并在流上进行转换，然后生成新的流。对于 Flink 的网络机制一种形象的类比是，Flink 使用了高效有界的分布式阻塞队列，就像 Java 通用的阻塞队列（BlockingQueue）一样。使用 BlockingQueue 的话，一个较慢的接受者会降低发送者的发送速率，因为一旦队列满了（有界队列）发送者会被阻塞。Flink 解决背压的方案就是这种感觉。

在 Flink 中，这些分布式阻塞队列就是这些逻辑流，而队列容量是通过缓冲池来（LocalBufferPool）实现的。每个被生产和被消费的流都会被分配一个缓冲池。缓冲池管理着一组缓冲(Buffer)，缓冲在被消费后可以被回收循环利用。这很好理解：你从池子中拿走一个缓冲，填上数据，在数据消费完之后，又把缓冲还给池子，之后你可以再次使用它。

3、网络传输中的内存管理

在解释 Flink 的反压原理之前，我们必须先对 Flink 中网络传输的内存管理有个了解。

如下图所示展示了 Flink 在网络传输场景下的内存管理。网络上传输的数据会写到 Task 的 InputGate（IG） 中，经过 Task 的处理后，再由 Task 写到 ResultPartition（RS） 中。每个 Task 都包括了输入和输入，输入和输出的数据存在 Buffer 中（都是字节数据）。Buffer 是 MemorySegment 的包装类。



1）TaskManager（TM）在启动时，会先初始化NetworkEnvironment对象，TM 中所有与网络相关的东西都由该类来管理（如 Netty 连接），其中就包括NetworkBufferPool。根据配置，Flink 会在 NetworkBufferPool 中生成一定数量（默认2048）的内存块 MemorySegment，内存块的总数量就代表了网络传输中所有可用的内存。NetworkEnvironment 和 NetworkBufferPool 是 Task 之间共享的，每个 TM 只会实例化一个。

2）Task 线程启动时，会向 NetworkEnvironment 注册，NetworkEnvironment 会为 Task 的 InputGate（IG）和 ResultPartition（RP） 分别创建一个 LocalBufferPool（缓冲池）并设置可申请的 MemorySegment（内存块）数量。IG 对应的缓冲池初始的内存块数量与 IG 中 InputChannel 数量一致，RP 对应的缓冲池初始的内存块数量与 RP 中的 ResultSubpartition 数量一致。不过，每当创建或销毁缓冲池时，NetworkBufferPool 会计算剩余空闲的内存块数量，并平均分配给已创建的缓冲池。注意，这个过程只是指定了缓冲池所能使用的内存块数量，并没有真正分配内存块，只有当需要时才分配。为什么要动态地为缓冲池扩容呢？因为内存越多，意味着系统可以更轻松地应对瞬时压力（如GC），不会频繁地进入反压状态，所以我们要利用起那部分闲置的内存块。

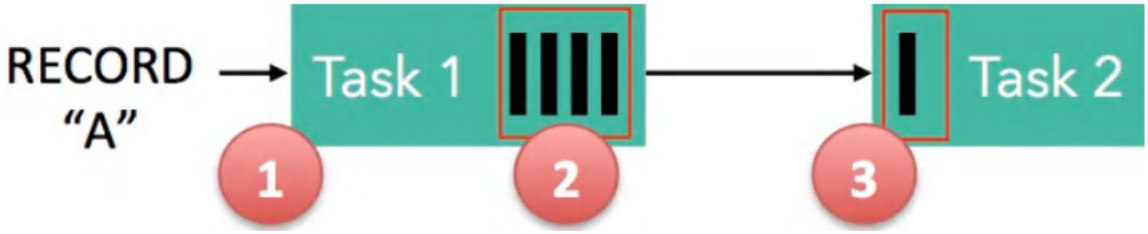
3）在 Task 线程执行过程中，当 Netty 接收端收到数据时，为了将 Netty 中的数据拷贝到 Task 中，InputChannel（实际是 RemoteInputChannel）会向其对应的缓冲池申请内存块（上图中的①）。如果缓冲池中也没有可用的内存块且已申请的数量还没到池子上限，则会向 NetworkBufferPool 申请内存块（上图中的②）并交给 InputChannel 填上数据（上图中的③和④）。如果缓冲池已申请的数量达到上限了呢？或者 NetworkBufferPool 也没有可用内存块了呢？这时候，Task 的 Netty Channel 会暂停读取，上游的发送端会立即响应停止发送，拓扑会进入反压状态。当 Task 线程写数据到 ResultPartition 时，也会向缓冲池请求内存块，如果没有可用内存块时，会阻塞在请求内存块的地方，达到暂停写入的目的。

4）当一个内存块被消费完成之后（在输入端是指内存块中的字节被反序列化成对象了，在输出端是指内存块中的字节写入到 Netty Channel 了），会调用 Buffer.recycle() 方法，会将内存块还给 LocalBufferPool （上图中的⑤）。如果LocalBufferPool中当前申请的数量超过了池子容量（由于上文提到的动态容量，由于新注册的 Task 导致该池子容量变小），则LocalBufferPool会将该内存块回收给 NetworkBufferPool（上图中的⑥）。如果没超过池子容量，则会继续留在池子中，减少反复申请的开销。

4、背压过程

举例说明Flink背压的过程：

下图有一个简单的flow，它由两个task组成



1）记录A进入Flink，然后Task1处理

2）Task1处理后的结果被序列化进缓存区

3）Task2从缓存区内读取一些数据，缓存区内将有更多的空间

4）如果Task2处理的较慢，Task1的缓存区很快填满，发送速度随之下降。

不要忘记记录能被Flink处理的前提：必须有空闲可用的缓存区（Buffer）。

结合上面两张图看：Task 1 在输出端有一个相关联的 LocalBufferPool（称缓冲池1），Task 2 在输入端也有一个相关联的 LocalBufferPool（称缓冲池2）。如果缓冲池1中有空闲可用的 buffer 来序列化记录 “A”，我们就序列化并发送该 buffer。

这里我们需要注意两个场景：

公众号：旧时光大数据

本地传输：如果 Task 1 和 Task 2 运行在同一个 worker 节点（TaskManager），该 buffer 可以直接交给下一个 Task。一旦 Task 2 消费了该 buffer，则该 buffer 会被缓冲池1回收。如果 Task 2 的速度比 1 慢，那么 buffer 回收的速度就会赶不上 Task 1 取 buffer 的速度，导致缓冲池1无可用的 buffer，Task 1 等待在可用的 buffer 上。最终形成 Task 1 的降速。

远程传输：如果 Task 1 和 Task 2 运行在不同的 worker 节点上，那么 buffer 会在发送到网络（TCP Channel）后被回收。在接收端，会从 LocalBufferPool 中申请 buffer，然后拷贝网络中的数据到 buffer 中。如果没有可用的 buffer，会停止从 TCP 连接中读取数据。在输出端，通过 Netty 的水位值机制来保证不往网络中写入太多数据（后面会说）。如果网络中的数据（Netty输出缓冲中的字节数）超过了高水位值，我们会等到其降到低水位值以下才继续写入数据。这保证了网络中不会有太多的数据。如果接收端停止消费网络中的数据（由于接收端缓冲池没有可用 buffer），网络中的缓冲数据就会堆积，那么发送端也会暂停发送。另外，这会使得发送端的缓冲池得不到回收，writer 阻塞在向 LocalBufferPool 请求 buffer，阻塞了 writer 往 ResultSubPartition 写数据。

这种固定大小缓冲池就像阻塞队列一样，保证了 Flink 有一套健壮的反压机制，使得 Task 生产数据的速度不会快于消费的速度。我们上面描述的这个方案可以从两个 Task 之间的数据传输自然地扩展到更复杂的 pipeline 中，保证反压机制可以扩散到整个 pipeline。

Flink解决数据延迟的问题

问过的一些公司：

参考答案：

1、官方解释

迟到数据是指系统的事件时间时钟（由水印指示）在经过延迟元素时间戳之后的时间到达的元素。

2、解决方案

迟到数据可以说是一种特殊的乱序数据，它没有被watermark和Window机制处理，因为是在窗口关闭后才到达的数据。一般这种情况有**三种处理办法**：

- 重新激活已经关闭的窗口并重新计算以修正结果。
- 将迟到数据收集起来另外处理。
- 将迟到数据视为错误消息并丢弃。

Flink默认采用第三种方法，将迟到数据视为错误消息丢弃。想要使用前两种方法需要使用到**sideOutput机制**和**allowedLateness机制**。

sideOutput机制可以将迟到事件单独放入一个数据流分支，这会作为 window 计算结果的副产品，以便用户获取并对其进行特殊处理。

allowedLateness机制允许用户设置一个允许的最大迟到时长。Flink 会在窗口关闭后一直保存窗口的状态直至超过允许迟到时长，这期间的迟到事件不会被丢弃，而是默认会触发窗口重新计算。

所以，如果要设置允许延迟的时间，可以通过DataStream.allowedLateness(lateness: Time)。如果要保存延迟数据，要通过sideOutputLateData(outputTag: OutputTag[T])来保存。而要获取已经保存的延迟数据，则要通过DataStream.getSideOutput(tag: OutputTag[X])。

```
import org.apache.commons.lang.time.FastDateFormat
import org.apache.flink.api.java.tuple.Tuple
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.functions.AssignerWithPeriodicWatermarks
import org.apache.flink.streaming.api.scala.function.WindowFunction
import org.apache.flink.streaming.api.scala.{DataStream, OutputTag,
StreamExecutionEnvironment, WindowedStream}
```

公众号：旧时光大数据

```scala
import org.apache.flink.streaming.api.watermark.Watermark
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time
import org.apache.flink.streaming.api.windowing.windows.TimeWindow
import org.apache.flink.util.Collector
import scala.collection.mutable.ArrayBuffer

object HandleLatenessDemo {
  //定义一个对象，将时间戳转换为时间
  val sdf = FastDateFormat.getInstance("yyyy-MM-dd HH:mm:ss:SSS")

  def main(args: Array[String]): Unit = {
    /**
     * 1、监听某主机的9999端口，读取socket数据(格式为  name:timestamp)
     * 2、给当前进入flink程序的数据加上waterMark，值为eventTime-3s
     * 3、根据name值进行分组，根据窗口大小为5s划分窗口，设置允许迟到时间为2s，依次统计窗口中各name值的数据
     * 4、输出统计结果以及迟到数据
     * 5、启动Job
     */
    // 1.获取执行环境
    val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
    // 1.1 将event time设置为流数据时间类型
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    // 2.获取socket数据源
    // 2.1 socketTextStream可以有hostname/ port/ delimiter/ maxRetry/ 四个参数，这里设置一下前三个，方便后续修改
    val hostname = "node01"
    val port = 9999
    val delimiter = '\n'

    val socketStream: DataStream[String] = env.socketTextStream(hostname, port, delimiter)

    // 3.将输入的数据进行转换
    import org.apache.flink.api.scala._
    val data: DataStream[(String, Long)] = socketStream.map(line => {
      // 比如，输入的数据格式是：name 时间戳
      try {
        val item: Array[String] = line.split(",")
        (item(0).trim, item(1).trim.toLong)
      } catch {
        case _: Exception => println("输入的数据格式不合法")
          ("spark", 0L)
      }
    }).filter(data => data._1.equals("0") && data._2 != 0L)

    // 4.对数据流中的元素分配时间戳，并定期地创建水印(20ms)，监控时间的进度
    val watermarkDataStream: DataStream[(String, Long)] = data.assignTimestampsAndWatermarks(new AssignerWithPeriodicWatermarks[(String, Long)] {
      // 当前最大时间戳
      private var currentMaxTimestamp = 0L
      // 最大延迟时间
      val maxOutOfOrderness = 3000L
```

公众号：旧时光大数据

```
58        var lastEmittedWatermark = Long.MinValue
59
60        /**
61         * 获取当前水印
62         * 该方法会被周期性地执行，20ms一个周期
63         */
64        override def getCurrentWatermark: Watermark = {
65          // 允许最大延迟 3s
66          val potentialWatermark: Long = currentMaxTimestamp - maxOutOfOrderness
67          new Watermark(potentialWatermark)
68        }
69
70        override def extractTimestamp(element: (String, Long),
    previousElementTimestamp: Long): Long = {
71          //将元素的时间字段作为该数据的timestamp
72          val time: Long = element._2
73          if (time > currentMaxTimestamp) {
74            currentMaxTimestamp = time
75          }
76          val outData: String = String.format("key: %s EventTime: %s watermark:
    %s", element._1, sdf.format(time),
    sdf.format(getCurrentWatermark.getTimestamp))
77          println(outData)
78          time
79        }
80      })
81
82      // 5.定义一个侧输出流
83      val lateData: OutputTag[(String, Long)] = new OutputTag[(String, Long)]
    ("late")
84
85      // 6.对水印数据聚合并引入窗口
86      val windowedStream: WindowedStream[(String, Long), Tuple, TimeWindow] =
    watermarkDataStream.keyBy(0).window(TumblingEventTimeWindows.of(Time.seconds(5L
    )))
87
88      // 7.获取带有延迟时间的数据流
89      val result: DataStream[String] =
    windowedStream.allowedLateness(Time.seconds(2)) //允许迟到2s
90        .sideOutputLateData(lateData)
91        .apply(new WindowFunction[(String, Long), String, Tuple, TimeWindow] {
92          override def apply(key: Tuple, window: TimeWindow, input:
    Iterable[(String, Long)], out: Collector[String]): Unit = {
93            val timeArr: ArrayBuffer[String] = ArrayBuffer[String]()
94            val iterator: Iterator[(String, Long)] = input.iterator
95            // 将所有的事件时间放到集合中
96            while (iterator.hasNext) {
97              val tuple: (String, Long) = iterator.next()
98              timeArr.append(sdf.format(tuple._2))
99            }
100           val outData = String.format("key: %s   data: %s   windowStartTime: %s
      windowEndTime: %s",
101             key.toString,
102             timeArr.mkString(","),
103             sdf.format(window.getStart),
104             sdf.format(window.getEnd)
105           )
106           out.collect(outData)
```

```
107             }
108           })
109         result.print("Window的计算结果>>>")
110
111         val late: DataStream[(String, Long)] = result.getSideOutput(lateData)
112         late.print("迟到的数据>>>")
113
114         // 启动执行环境
115         env.execute("HandleLatenessDemo")
116       }
117     }
118
```

Flink消费kafka分区的数据时flink任务并行度之间的关系

问过的一些公司：科大讯飞

参考答案：

从消费端看， 如果source的并行度大于kafka的分区，会导致多余的并行度消费不了数据，进而影响checkpoint，flink做checkpoint失败，所以一定不能大于分区数。

从消费端看， 如果source的并行度小于kafka的分区，会导致一个并行度消费多个kafka分区数据，如果数据量大的时候，吞吐率上不去。

sink到kafka时，如果并行度大于kafka的分区数时，则会轮训把数据插入到kafka分区中，数据不会丢失。

sink到kafka时，如果并行度小于kafka的分区数时，也会轮训把数据插入到kafka分区中，数据不会丢失。 因为如果指定key的情况下，则producer会按照hash规则，把数据hash到相应分区中。也就是说flink sink的时候对kafka并行度影响不大，不会存在数据丢失或者分区没有写入的情况。

这里还有关于Flink和Blink之间的对比：

flink的并行度如果大于kafka的分区，checkpoint可以正常执行成功，但是会存在某个并行度空转的情况。可以正常checkpoint。

blink的并行度如果大于kafka的分区，checkpoint不能正常执行成功，checkpoint会失败，一直没checkpoint。

使用flink-client消费kafka数据还是使用flink-connector消费

问过的一些公司：科大讯飞

参考答案：

Flink 是通过Connector与具体的source 和 sink进行通信的

如何动态修改Flink的配置，前提是Flink不能重启

可回答：Spark 如何实现动态更新作业配置

问过的一些公司：vivo

参考答案：

公众号：旧时光大数据

1、Flink/Spark 如何实现动态更新作业配置

由于实时场景对可用性十分敏感，实时作业通常需要避免频繁重启，因此动态加载作业配置（变量）是实时计算里十分常见的需求，比如通常复杂事件处理 (CEP) 的规则或者在线机器学习的模型。尽管常见，实现起来却并没有那么简单，其中最难点在于如何确保节点状态在变更期间的一致性。目前来说一般有两种实现方式：

- 轮询拉取方式，即作业算子定时检测在外部系统的配置是否有变更，若有则同步配置。
- 控制流方式，即作业除了用于计算的一个或多个普通数据流以外，还有提供一个用于改变作业算子状态的元数据流，也就是控制流。

轮询拉取方式基于 pull 模式，一般实现是用户在 Stateful 算子(比如 RichMap)里实现后台线程定时从外部系统同步变量。这种方式对于一般作业或许足够，但存在两个缺点分别限制了作业的实时性和准确性的进一步提高：首先，轮询总是有一定的延迟，因此变量的变更不能第一时间生效；其次，这种方式依赖于节点本地时间来进行校准。如果在同一时间有的节点已经检测到变更并更新状态，而有的节点还没有检测到或者还未更新，就会造成短时间内的不一致。

控制流方式基于 push 模式，变更的检测和节点更新的一致性都由计算框架负责，从用户视角看只需要定义如何更新算子状态并负责将控制事件丢入控制流，后续工作计算框架会自动处理。控制流不同于其他普通数据流的地方在于控制流是以广播形式流动的，否则在有 Keyby 或者 rebalance 等提高并行度分流的算子的情况下就无法将控制事件传达给所有的算子。

以目前最流行的两个实时计算框架 Spark Streaming 和 Flink 来说，**前者是以类似轮询的方式来实现实时作业的更新，而后者则是基于控制流的方式。**

2、Spark Streaming Broadcast Variable

Spark Streaming 为用户提供了 Broadcast Varialbe，可以用于节点算子状态的初始化和后续更新。Broacast Variable 是一组只读的变量，它在作业初始化时由 Spark Driver 生成并广播到每个 Executor 节点，随后该节点的 Task 可以复用同一份变量。

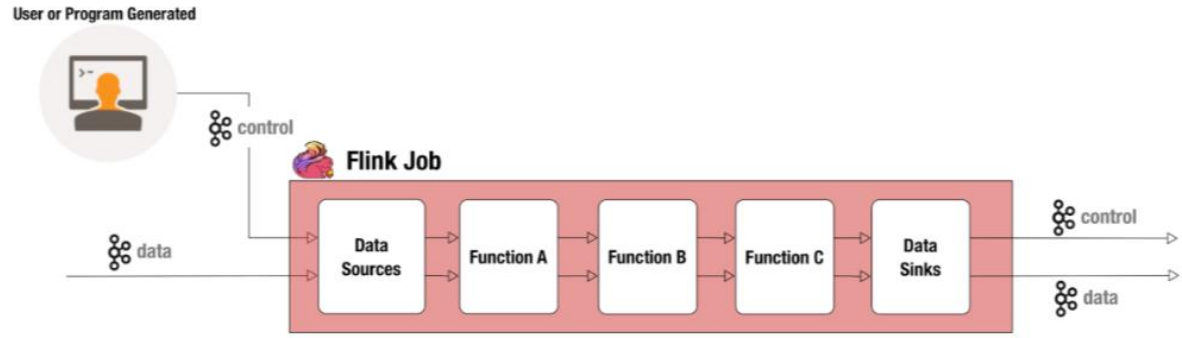
Broadcast Variable 的设计初衷是为了避免大文件，比如 NLP 常用的分词词典，随序列化后的作业对象一起分发，造成重复分发的网络资源浪费和启动时间延长。这类文件的更新频率是相对低的，扮演的角色类似于只读缓存，通过设置 TTL 来定时更新，缓存过期之后 Executor 节点会重新向 Driver 请求最新的变量。

Broadcast Variable 并不是从设计理念上就支持低延迟的作业状态更新，因此用户想出了不少 Hack 的方法，其中最为常见的方式是：一方面在 Driver 实现后台线程不断更新 Broadcast Variavle，另一方面在作业运行时通过显式地删除 Broadcast Variable 来迫使 Executor 重新从 Driver 拉取最新的 Broadcast Variable。这个过程会发生在两个 micro batch 计算之间，以确保每个 micro batch 计算过程中状态是一致的。

比起用户在算子内访问外部系统实现更新变量，这种方式的优点在于一致性更有保证。因为 Broadcast Variable 是统一由 Driver 更新并推到 Executor 的，这就保证不同节点的更新时间是一致的。然而相对地，缺点是会给 Driver 带来比较大的负担，因为需要不断分发全量的 Broadcast Variable (试想下一个巨大的 Map，每次只会更新少数 Entry，却要整个 Map 重新分发)。在 Spark 2.0 版本以后，Broadcast Variable 的分发已经从 Driver 单点改为基于 BitTorrent 的 P2P 分发，这一定程度上缓解了随着集群规模提升 Driver 分发变量的压力，但我个人对这种方式能支持到多大规模的部署还是持怀疑态度。另外一点是重新分发 Broadcast Variable 需要阻塞作业进行，这也会使作业的吞吐量和延迟受到比较大的影响。

3、Flink Broadcast State & Stream

Broadcast Stream 是 Flink 1.5.0 发布的新特性，基于控制流的方式实现了实时作业的状态更新。Broadcast Stream 的创建方式与普通数据流相同，例如从 Kafka Topic 读取，特别之处在于它承载的是控制事件流，会以广播形式将数据发给下游算子的每个实例。Broadcast Stream 需要在作业拓扑的某个节点和普通数据流 (Main Stream) join 到一起。



该节点的算子需要同时处理普通数据流和控制流：一方面它需要读取控制流以更新本地状态 (Broadcast State)，另外一方面需要读取 Main Stream 并根据 Broadcast State 来进行数据转换。由于每个算子实例读到的控制流都是相同的，它们生成的 Broadcast State 也是相同的，从而达到通过控制消息来更新所有算子实例的效果。

目前 Flink 的 Broadcast Stream 从效果上实现了控制流的作业状态更新，不过在编程模型上有点和一般直觉不同。原因主要在于 Flink 对控制流的处理方式和普通数据流保持了一致，最为明显的一点是控制流除了改变本地 State 还可以产生 output，这很大程度上影响了 Broadcast Stream 的使用方式。Broadcast Stream 的使用方式与普通的 DataStream 差别比较大，即需要和 DataStream 连接成为 BroadcastConnectedStream 后，再通过特殊的 BroadcastProcessFunction 来处理，而 BroadcastProcessFunction 目前只支持 类似于 RichCoFlatMap 效果的操作。RichCoFlatMap 可以间接实现对 Main Stream 的 Map 转换（返回一只有一个元素的集合）和 Filter 转换（返回空集合），但无法实现 Window 类计算。这意味着如果用户希望改变 Window 算子的状态，那么需要将状态管理提前到上游的 BroadcastProcessFunction，然后再通过 BroadcastProcessFunction 的输出来将影响下游 Window 算子的行为。

4、总结

实时作业运行时动态加载变量可以令大大提升实时作业的灵活性和适应更多应用场景，目前无论是 Flink 还是 Spark Streaming 对动态加载变量的支持都不是特别完美。Spark Streaming 受限于 Micro Batch 的计算模型（虽然现在 2.3 版本引入 Continuous Streaming 来支持流式处理，但离成熟还需要一定时间），将作业变量作为一致性和实时性要求相对低的节点本地缓存，并不支持低延迟地、低成本地更新作业变量。Flink 将变量更新视为特殊的控制事件流，符合 Even Driven 的流式计算框架定位，目前在业界已有比较成熟的应用。不过美中不足的是编程模型的易用性上有提高空间：控制流目前只能用于和数据流的 join，这意味着下游节点无法继续访问控制流或者需要把控制流数据插入到数据流中（这种方式并不优雅），从而降低了编程模型的灵活性。最好的情况是大部分的算子都可以被拓展为具有 BroadcastOperator，就像 RichFunction 一样，它们可以接收一个数据流和一个至多个控制流，并维护对应的 BroadcastState，这样控制流的接入成本将显著下降。

Flink流批一体解释一下

问过的一些公司：字节

参考答案：

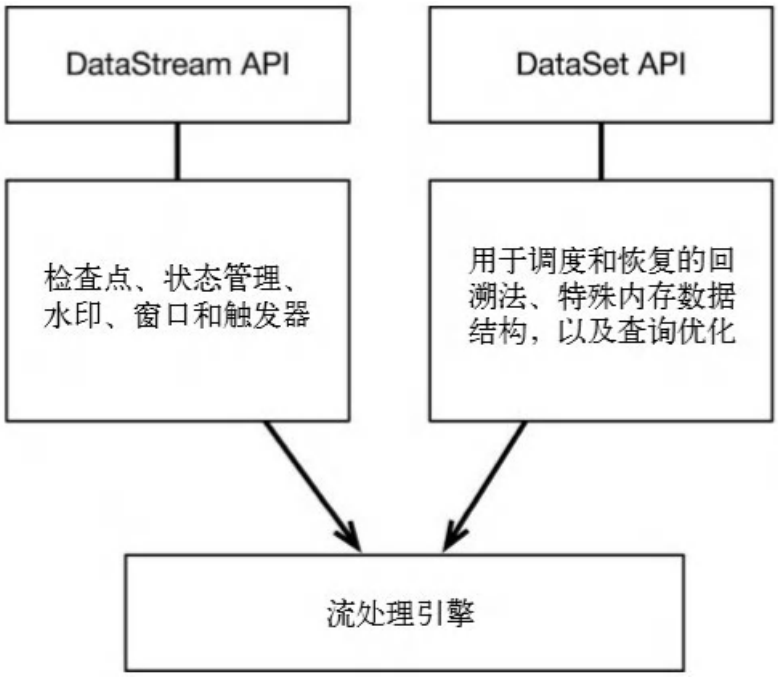
在大数据处理计算领域，有离线计算和实时计算两种模式。一般都是用mapreduce / hive / sparkSQL来处理离线场景，用 sparkStreaming / flink处理实时场景，但是这种lambda架构会导致一个问题：进行更改时要同时更改两套代码，进行同步。

flink流批一体横空处理，为大数据处理带来了一套新的解决方案。

双11中Flink流批一体开始在阿里最核心的数据业务场景崭露头角，并扛住了40亿/秒的实时计算峰值。

其实流批一体的技术里面最早提出于2015年，它的初衷是让大数据开发人员能够用同一套接口实现大数据的流计算和批计算，进而保证处理过程与结果的一致性。spark、flink都陆续提出了自己的解决方案。虽然spark是最早提出流批一体理念的计算引擎之一，但其本质还是用批来实现流，用的是微批次的思想，有秒级的延迟，而且无法正确处理时间语义（数据在分布式传输过程中顺序发生改变，先生产的数据反而后到，导致计算不准确的一种现象），所以难以满足复杂、大规模的实时计算场景，迟迟无法落地。而2019年阿里收购flink后，投入大量研发力量，同时公司也面临离线和实时数据统计口径不一致的问题，影响广告、商务甚至是公司的运行决策，业务的迫切要求，技术力量的不断加入，都促进了flink向流批一体的发展。

Flink 通过一个底层引擎同时支持流处理和批处理



在流处理引擎之上，Flink 有以下机制：

- 检查点机制和状态机制：用于实现容错、有状态的处理；
- 水印机制：用于实现事件时钟；
- 窗口和触发器：用于限制计算范围，并定义呈现结果的时间。

在同一个流处理引擎之上，Flink 还存在另一套机制，用于实现高效的批处理。

- 用于调度和恢复的回溯法：由 Microsoft Dryad 引入，现在几乎用于所有批处理器；
- 用于散列和排序的特殊内存数据结构：可以在需要时，将一部分数据从内存溢出到硬盘上；
- 优化器：尽可能地缩短生成结果的时间。

两套机制分别对应各自的API（DataStream API 和 DataSet API）；在创建 Flink 作业时，并不能通过将两者混合在一起来同时 利用 Flink 的所有功能。

在最新的版本中，Flink 支持两种关系型的 API，**Table API 和 SQL。这两个 API 都是批处理和流处理统一的 API**，这意味着在无边界的实时数据流和有边界的历史记录数据流上，关系型 API 会以相同的语义执行查询，并产生相同的结果。Table API 和 SQL 借助了 Apache Calcite 来进行查询的解析，校验以及优化。它们可以与 DataStream 和 DataSet API 无缝集成，并支持用户自定义的标量函数，聚合函数以及表值函数。

Table API / SQL 正在以流批统一的方式成为分析型用例的主要 API。

DataStream API 是数据驱动应用程序和数据管道的主要API。

从长远来看，DataStream API应该通过有界数据流完全包含DataSet API。

公众号：旧时光大数据

说一下Flink的check和barrier

问过的一些公司：字节，美团(2021.08)

参考答案：

1、Flink Check

Flink Check 是 Apache Flink 的一个基于属性的测试库，它扩展了 ScalaCheck 的线性时序逻辑运算符，适用于测试 Flink 数据流转换。

基于属性的测试(PBT)是一种自动的黑盒测试技术，它通过生成随机输入并检查获得的输出是否满足给定的属性来测试功能。

Flink Check 提供了一个有边界的时间逻辑，用于生成函数的输入和声明属性。这个逻辑是为流媒体系统设计的，它允许用户定义流如何随时间变化，以及哪些属性应该验证相应的输出。

Flink Check随机生成指定数量的有限输入流前缀，并对Flink运行时产生的输出流进行评估。

Flink Check 是基于 sscheck，这是Spark 的一个基于属性的测试库，所以它依赖于 sscheck-core 项目，其中包含了 sscheck 和 Flink Check 共同的代码，特别是系统所基于的 LTLss 逻辑的实现。

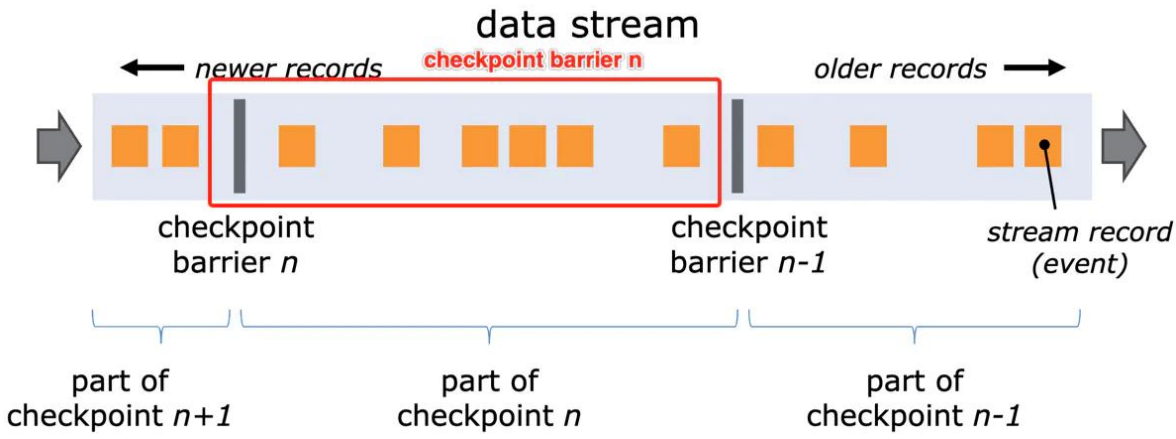
LTLss 是一种有限字的离散时间线性时序逻辑，在 Spark Streaming 的 Property-based testing for Spark Streaming 的论文中有详细介绍。

2、barrier

Flink提供了容错机制，能够在应用失败的时候重新恢复任务。这个机制主要就是通过持续产生快照的方式实现的。Flink快照主要包括两部分数据一部分是数据流的数据，另一部分是operator的状态数据。对应的快照机制的实现有主要两个部分组成，一个是屏障(Barrier），一个是状态(State)。

Flink 分布式快照里面的一个核心的元素就是流屏障（stream barrier）。这些屏障会被插入(injected)到数据流中，并作为数据流的一部分随着数据流动。屏障并不会持有任何数据，而是和数据一样线性的流动。可以看到屏障将数据流分成了两部分数据（实际上是多个连续的部分），一部分是当前快照的数据，一部分下一个快照的数据。每个屏障会带有它的快照ID。这个快照的数据都在这个屏障的前面。从图上看，数据是从左向右移动（右边的先进入系统），那么快照n包含的数据就是右侧到下一个屏障（n-1）截止的数据，图中两个灰色竖线之间的部分，也就是part of checkpoint n。另外屏障并不会打断数的流动,因而屏障是非常轻量的。在同一个时刻，多个快照可以在同一个数据流中，这也就是说多个快照可以同时产生。

屏障示意图如下：



如果是多个输入数据流，多个数据流的屏障会被同时插入到数据流中。快照n的屏障被插入到数据流的点（我们称之为Sn），就是数据流中一直到的某个位置（包含了当前时刻之前时间的所有数据），也就是包含的这部分数据的快照。举例来说，在Kafka中，这个位置就是这个分区的最后一条记录的offset。这个位置Sn就会上报给 checkpoint 的协调器（Flink的 JobManager）。

公众号：旧时光大数据

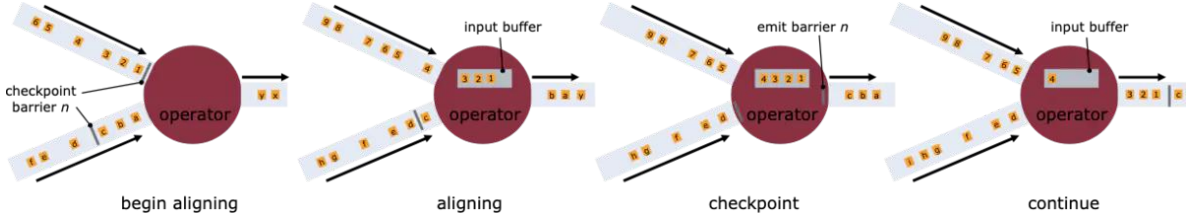
然后屏障开始向下流动。当一个中间的operator收到它的所有输入源的快照n屏障后，它就会向它所有的输出流发射一个快照n的屏障，一旦一个sink的operator收到所有输入数据流的屏障n，它就会向checkpoint的协调器发送快照n确认。当所有的sink都确认了快照n，系统才认为当前快照的数据已经完成。

一旦快照n已经执行完成，任务则不会再请求Sn之前的数据，因为此刻，这些数据都已经完全通过了数据流拓扑图。

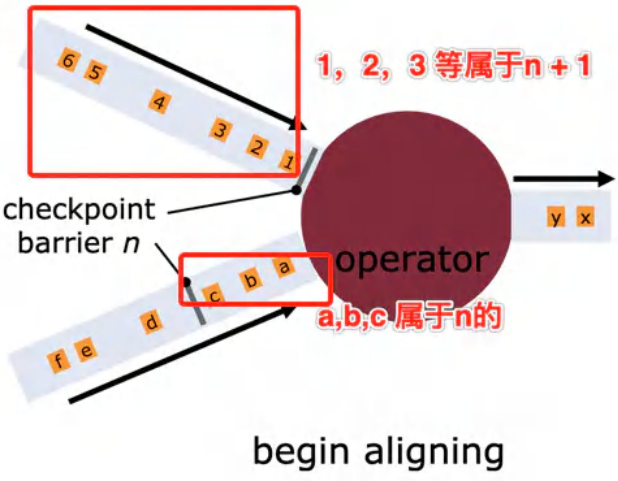
1）对齐机制

接收不止一个数据输入的operator需要基于屏障对齐输入数据流。详述如下：

整个流程图如下所示

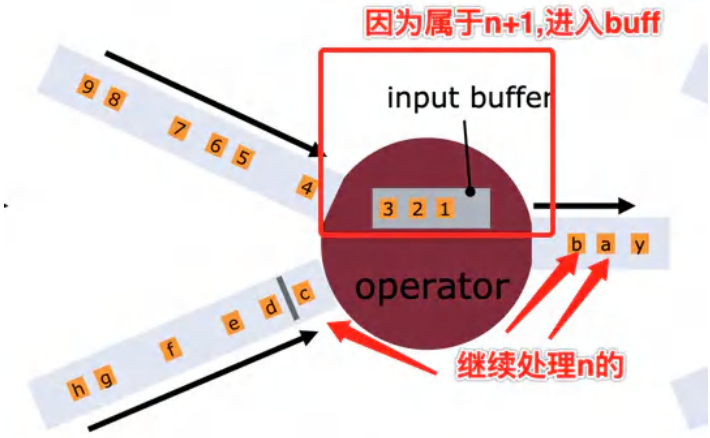


当operator接收到快照的屏障n后并不能直接处理之后的数据，而是需要等待其他输入快照的屏障n。否则话，将会将快照n的数据和快照n+1的数据混在一起。图中第一个所示，operator即将要收到数据流1(上面这个我们当成数据流1（6，5，4，3，2，1），下面的当成数据流2好了)的屏障n，1，2，3在屏障n之后到达operator，这个时候如果数据流1的继续处理，那么operator中就会包含n屏障之后的数据（1，2，3），但是operator中此刻在接收和处理数据流2，数据（a,b,c）就会和数据流1中的（1，2，3）混合。



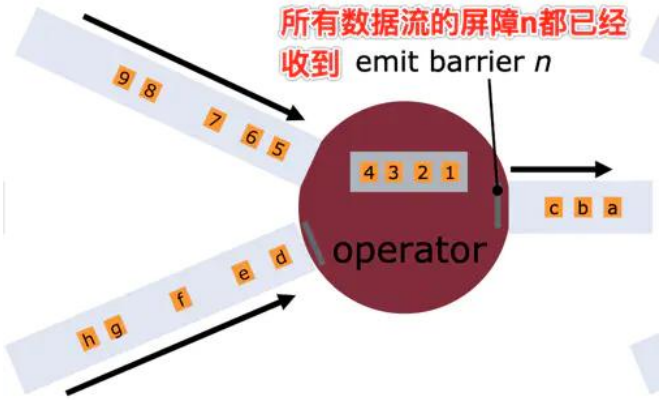
快照n的数据流会被暂时放到一边。从这些数据流中获取到的数据不会被处理，而是存储到一个缓冲中。图中第一个所示，因为数据流2的屏障n还没到，所以operator持续接收1，2，3然而并不做任何处理。但是需要将1，2，3存入到buffer中。此时第二个数据流接到a，b，则直接发送，接到c发送c。

公众号：旧时光大数据



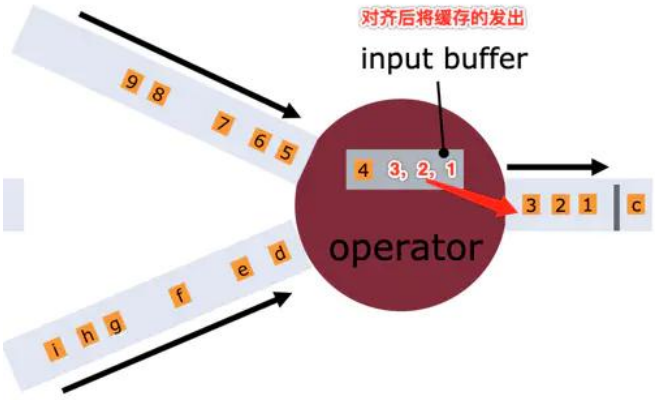
aligning

一旦最后一个数据流收到了快照n，opertor就会将发出所有阻塞的数据，并发出自己的屏障。如图中第三个所示，operator最后收到了另一个数据流的屏障n，然后再发出a,b,c(图中operator中的c,b,a)以后，发出自己的屏障，这个时候buffer中又增加了一个4，变成（4，3，2，1）。



checkpoint

之后operator会重新开始处理所有的输入数据流，先处理buffer中的数据，处理完之后再处理输入数据流的数据。如图第四个所示，先将buffer中的1，2，3，4先处理完，在接收并处理这两个数据源的数据。



continue

说一下Flink状态机制

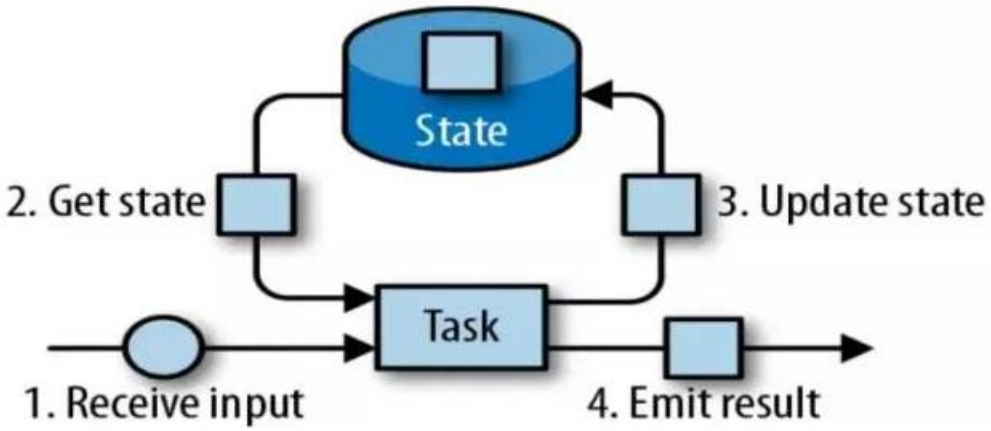
问过的一些公司：字节（2021.07）

公众号：旧时光大数据

参考答案：

1、状态分类

Flink在做计算的过程中经常需要存储中间状态，来避免数据丢失和状态恢复（Flink拥有丰富的状态访问和高效的容错机制）。即可以将中间的计算结果进行保存，并提供给后续的计算使用：



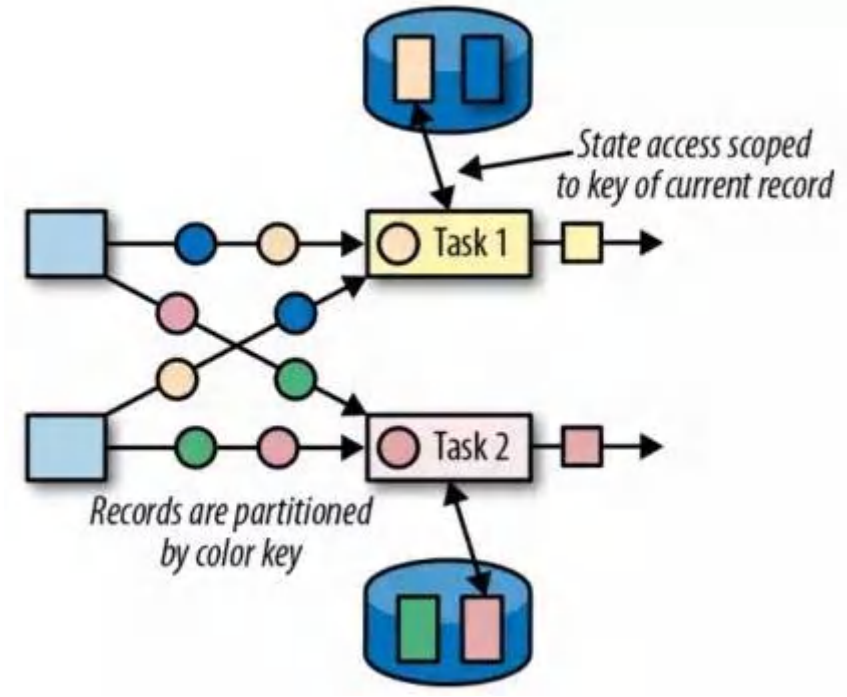
2、Flink中的状态管理

2.1 按照**数据的划分和扩张方式**，Flink中状态 (State) 大致分为2类：

- Keyed States（键控状态）
- Operator States（算子状态）

1）Keyed States（键控状态）

是一种特殊的算子状态，即**状态是根据 key 值进行区分的，Flink会为每类键值维护一个状态实例。**如下图所示，每个颜色代表不同 key 值，对应四个不同的状态实例。需要注意的是键控状态只能在 `KeyedStream` 上进行使用，我们可以通过 `stream.keyBy(...)` 来得到 `KeyedStream` 。



Flink 提供了以下数据格式来管理和存储键控状态（Keyed State）：

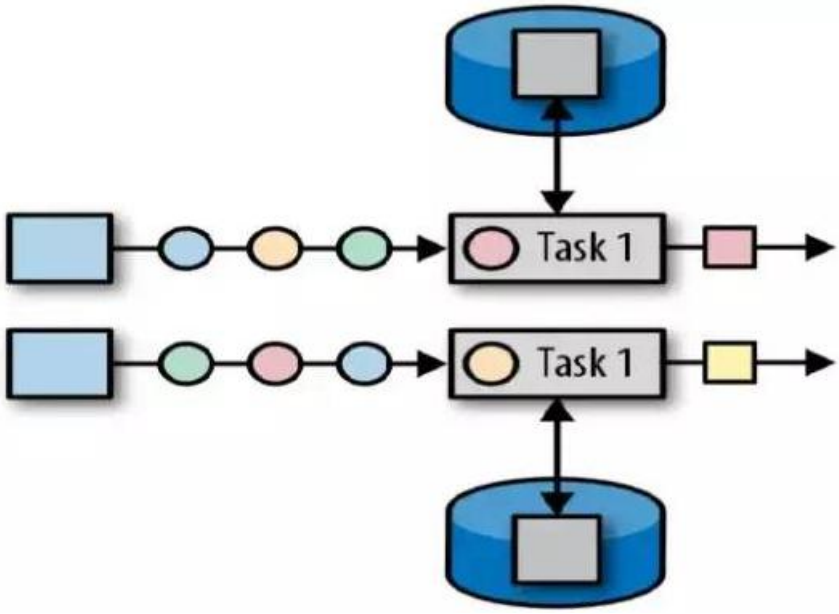
- **ValueState**：存储单值类型的状态。可以使用 `update(T)` 进行更新，并通过 `T value()` 进行检索。
- **ListState**：存储列表类型的状态。可以使用 `add(T)` 或 `addAll(List)` 添加元素；并通过 `get()` 获得整个列表。
- **ReducingState**：用于存储经过 ReduceFunction 计算后的结果，使用 `add(T)` 增加元素。

公众号：旧时光大数据

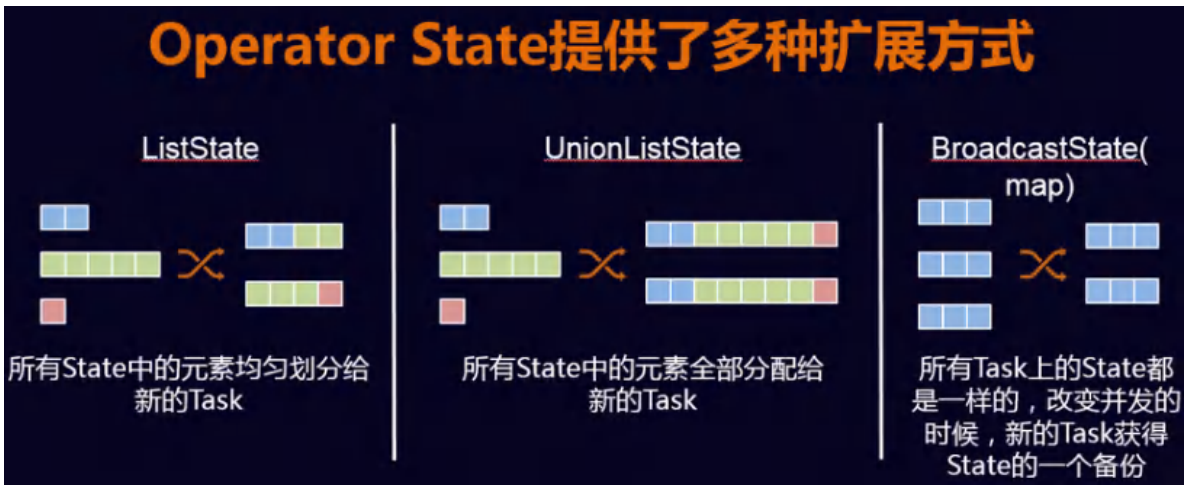
- **AggregatingState**：用于存储经过 AggregatingState 计算后的结果，使用 `add(IN)` 添加元素。
- **FoldingState**：已被标识为废弃，会在未来版本中移除，官方推荐使用 `AggregatingState` 代替。
- **MapState**：维护 Map 类型的状态。

2）Operator States（算子状态）

顾名思义，**状态是和算子进行绑定的，一个算子的状态不能被其他算子所访问到**。官方文档上对 Operator State 的解释是：each operator state is bound to one parallel operator instance，所以更为确切的说**一个算子状态是与一个并发的算子实例所绑定的**，即假设算子的并行度是2，那么其应有两个对应的算子状态：



Operator States多种扩展方式



Operator States的动态扩展是非常灵活的，现提供了3种扩展（支持的存储类型）：

- ListState：并发度在改变的时候，会将并发上的每个List都取出，然后把这些List合并到一个新的List，然后根据元素的个数在均匀分配给新的Task；（存储列表类型的状态）
- UnionListState：相比于ListState更加灵活，把划分的方式交给用户去做，当改变并发的时候，会将原来的List拼接起来。然后不做划分，直接交给用户，具体的划分行为则由用户进行定义；
- BroadcastState：如大表和小表做Join时，小表可以直接广播给大表的分区，在每个并发上的数据都是完全一致的。做的更新也相同，当改变并发的时候，把这些数据copy到新的Task即可。（用于广播的算子状态）

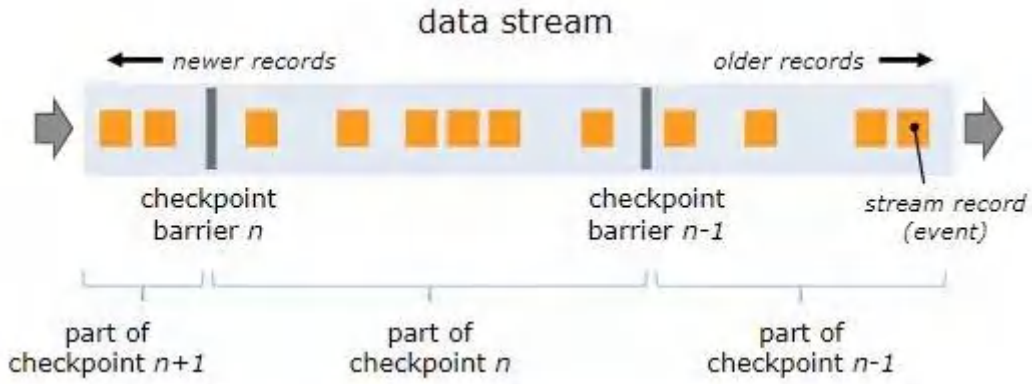
3、检查点机制（这点这里可以不看，放上来有助于理解）

3.1 CheckPoints

公众号：旧时光大数据

使用CheckPoints提高程序的可靠性

为了使 Flink 的状态具有良好的**容错性**，Flink 提供了检查点机制（CheckPoints） 。通过检查点机制，Flink 定期在数据流上生成 checkpoint barrier ，当某个算子收到 barrier 时，即会**基于当前状态生成一份快照**，然后再将该 barrier 传递到下游算子，下游算子接收到该 barrier 后，也基于当前状态生成一份快照，依次传递直至到最后的 Sink 算子上。**当出现异常后，Flink 就可以根据最近的一次的快照数据将所有算子恢复到先前的状态。**

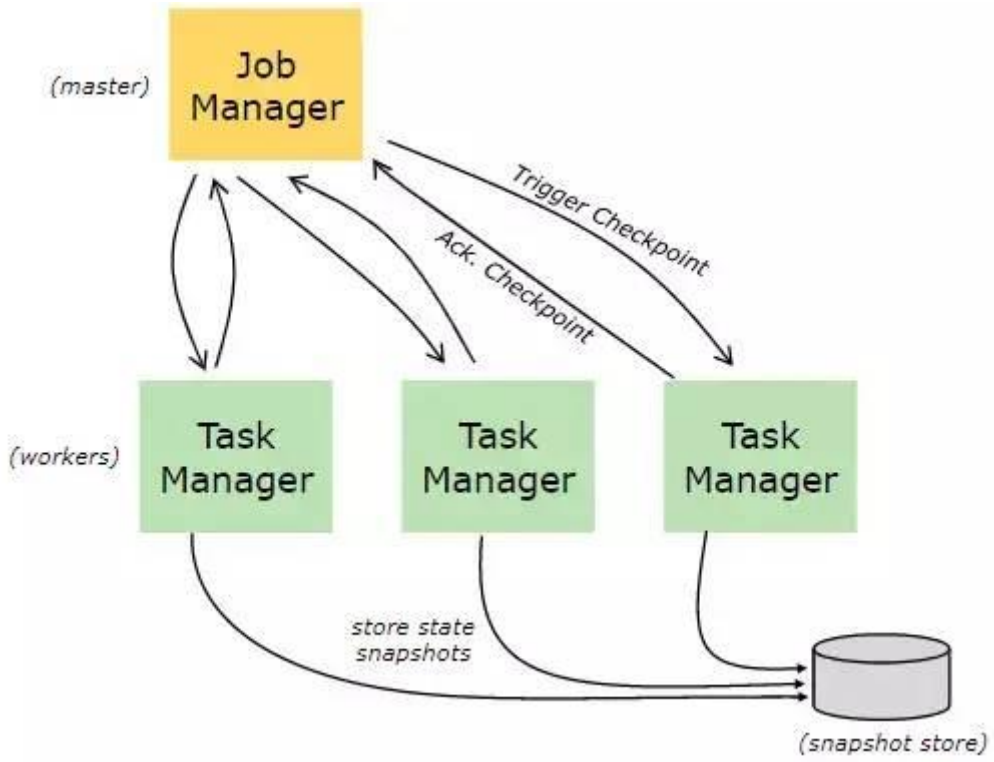


默认情况下，检查点机制是关闭的，需要在程序中进行开启

保存点机制 (Savepoints) 是检查点机制的一种特殊的实现，它允许你通过手工的方式来触发 Checkpoint，并将结果持久化存储到指定路径中，主要用于避免 Flink 集群在重启或升级时导致状态丢失。

4、状态后端（状态存储策略）

默认情况下，所有的状态都存储在 JVM 的堆内存中，在状态数据过多的情况下，这种方式很有可能导致内存溢出，因此 Flink 该提供了其它方式来存储状态数据，这些存储方式统一称为状态后端（或状态管理器）：



4.1 状态管理器（状态存储方式）分类

Flink提供了三种状态存储方式：MemoryStateBackend、FsStateBackend、RocksDBStateBackend。

1）MemoryStateBackend

公众号：旧时光大数据

默认的方式，即基于 JVM 的堆内存进行存储，主要适用于本地开发和调试。

2）FsStateBackend

基于文件系统进行存储，可以是本地文件系统，也可以是 HDFS 等分布式文件系统。需要注意而是虽然选择使用了 FsStateBackend ，但正在进行的数据仍然是存储在 TaskManager 的内存中的，只有在 checkpoint 时，才会将状态快照写入到指定文件系统上。

3）RocksDBStateBackend

RocksDBStateBackend 是 Flink 内置的第三方状态管理器，采用嵌入式的 key-value 型数据库 RocksDB 来存储正在进行的数据。等到 checkpoint 时，再将其中的数据持久化到指定的文件系统中，所以采用 RocksDBStateBackend 时也需要配置持久化存储的文件系统。之所以这样做是因为 RocksDB 作为嵌入式数据库安全性比较低，但比起全文件系统的方式，其读取速率更快；比起全内存的方式，其存储空间更大，因此它是一种比较均衡的方案。

用户可以根据自己的需求选择，如果数据量较小，可以存放到MemoryStateBackend和FsStateBackend中，如果数据量较大，可以放到RocksDBStateBackend中。

4.2 配置方式

Flink 支持使用两种方式来配置后端管理器：

第一种方式：基于代码方式进行配置，只对当前作业生效

```
FsStateBackendenv.setStateBackend(new
FsStateBackend("hdfs://namenode:40010/flink/checkpoints"));
RocksDBStateBackendenv.setStateBackend(new
RocksDBStateBackend("hdfs://namenode:40010/flink/checkpoints"));

```

第二种方式：基于 `flink-conf.yaml` 配置文件的方式进行配置，对所有部署在该集群上的作业都生效

```
state.backend: filesystemstate.checkpoints.dir:
hdfs://namenode:40010/flink/checkpoints

```

Flink广播流

问过的一些公司：字节(2021.07)

参考答案：

其实在上一题中，在Operator States也有介绍广播流（Broadcast State）。

广播状态（Broadcast State）是 **Apache Flink 中支持的第三种类型的Operator State。** Broadcast State使得 Flink 用户能够以**容错、一致、可扩缩容**地将来自广播的低吞吐的事件流数据存储下来，被广播到某个 operator 的所有并发实例中，然后与另一条流数据连接进行计算。

广播状态与其他 operator state 之间有三个主要区别：

- Map 的格式
- 有一条广播的输入流
- operator 可以有多个不同名字的广播状态

1、注意事项

在使用广播状态时要记住以下4个重要事项：

1）使用广播状态，operator task 之间不会相互通信

这也是为什么(Keyed)-BroadcastProcessFunction上只有广播的一边可以修改广播状态的内容。用户必须保证所有 operator 并发实例上对广播状态的修改行为都是一致的。或者说，如果不同的并发实例拥有不同的广播状态内容，将导致不一致的结果。

2）广播状态中事件的顺序在各个并发实例中可能不尽相同

虽然广播流的元素保证了将所有元素（最终）都发给下游所有的并发实例，但是元素的到达的顺序可能在并发实例之间并不相同。因此，对广播状态的修改不能依赖于输入数据的顺序。

3）所有 operator task 都会快照下他们的广播状态

在 checkpoint 时，所有的 task 都会 checkpoint 下它们的广播状态，并不仅仅是其中一个，即使所有 task 在广播状态中存储的元素是一模一样的。这是一个设计倾向，为了避免在恢复期间从单个文件读取而造成热点。然而，随着并发度的增加，checkpoint 的大小也会随之增加，这里会存在一个并发因子p的权衡。Flink保证了在恢复/扩缩容时不会出现重复数据和少数据。在以相同或更小并行度恢复时，每个 task 会读取其对应的检查点状态。在已更大并行度恢复时，每个 task 读取自己的状态，剩余的 task（p_newp_old）会以循环方式（round-robin）读取检查点的状态。

4）RocksDBStateBackend状态后端目前还不支持广播状态

广播状态目前在运行时保存在内存中。因为当前，RocksDB状态后端还不适用于operator state。Flink 用户应该相应地为其应用程序配置足够的内存。

2、广播状态模式的应用

一般来说广播状态的主要应用场景如下：

动态规则：动态规则是一条事件流，要求吞吐量不能太高。例如，当一个报警规则时触发报警信息等。我们将这个规则广播到计算的算子的所有并发实例中。

数据丰富：例如，将用户的详细信息作业广播状态进行广播，对包含用户ID的交易数据流进行数据丰富。

Flink实时topN

问过的一些公司：字节(2021.07)

参考答案：

下面以一个案例来介绍

场景描述： **TopN 是统计报表和大屏非常常见的功能，主要用来实时计算排行榜。**流式的TopN可以使业务方在内存中按照某个统计指标（如出现次数）计算排名并快速出发出更新后的排行榜。我们以统计词频为例展示一下如何快速开发一个计算TopN的Flink程序。

Flink支持各种各样的流数据接口作为数据的数据源，本次案例采用内置的socketTextStream作为数据数据源。

```
1   StreamExecutionEnvironment env =
    StreamExecutionEnvironment.getExecutionEnvironment();
2   env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime); //以
    processtime作为时间语义
3   DataStream<String> text = env.socketTextStream(hostName, port); //监听指定socket端口
    作为输入
4
```

公众号：旧时光大数据

与离线wordcount类似，程序首先需要把输入的整句文字按照分隔符split成一个一个单词，然后按照单词为key实现累加。

```
DataStream<Tuple2<String, Integer>> ds = text.flatMap(new LineSplitter()); //将输入语句split成一个一个单词并初始化count值为1的Tuple2<String, Integer>类型

private static final class LineSplitter implements FlatMapFunction<String, Tuple2<String, Integer>> {
        @Override
        public void flatMap(String value, Collector<Tuple2<String, Integer>> out) {
            // normalize and split the line
            String[] tokens = value.toLowerCase().split("\\W+");

            // emit the pairs
            for (String token : tokens) {
                if (token.length() > 0) {
                    out.collect(new Tuple2<String, Integer>(token, 1));
                }
            }
        }
}

```

```
DataStream<Tuple2<String, Integer>> wcount = ds
                .keyBy(0) //按照Tuple2<String, Integer>的第一个元素为key，也就是单词

.window(SlidingProcessingTimeWindows.of(Time.seconds(600),Time.seconds(20)))
                //key之后的元素进入一个总时间长度为600s,每20s向后滑动一次的滑动窗口
                .sum(1);// 将相同的key的元素第二个count值相加

```

全局TopN

数据流经过前面的处理后会每20s计算一次各个单词的count值并发送到下游窗口

```
DataStream<Tuple2<String, Integer>> ret = wcount
    .windowAll(TumblingProcessingTimeWindows.of(Time.seconds(20))) //所有key元素进入一个20s长的窗口（选20秒是因为上游窗口每20s计算一轮数据，topN窗口一次计算只统计一个窗口时间内的变化）
    .process(new TopNAllFunction(5));//计算该窗口TopN

```

windowAll是一个全局并发为1的特殊操作，也就是所有元素都会进入到一个窗口内进行计算。

```
private static class TopNAllFunction
            extends
            ProcessAllWindowFunction<Tuple2<String, Integer>, Tuple2<String, Integer>, TimeWindow> {

        private int topSize = 10;

        public TopNAllFunction(int topSize) {
            // TODO Auto-generated constructor stub

            this.topSize = topSize;
        }

```

公众号：旧时光大数据

```
        @Override
        public void process(
                ProcessAllWindowFunction<Tuple2<String, Integer>, Tuple2<String, Integer>, TimeWindow>.Context arg0,
                Iterable<Tuple2<String, Integer>> input,
                Collector<Tuple2<String, Integer>> out) throws Exception {
            // TODO Auto-generated method stub

            TreeMap<Integer, Tuple2<String, Integer>> treemap = new TreeMap<Integer, Tuple2<String, Integer>>(
                    new Comparator<Integer>() {

                        @Override
                        public int compare(Integer y, Integer x) {
                            // TODO Auto-generated method stub
                            return (x < y) ? -1 : 1;
                        }

                    }); //treemap按照key降序排列，相同count值不覆盖

            for (Tuple2<String, Integer> element : input) {
                treemap.put(element.f1, element);
                if (treemap.size() > topSize) { //只保留前面TopN个元素
                    treemap.pollLastEntry();
                }
            }

            for (Entry<Integer, Tuple2<String, Integer>> entry : treemap
                    .entrySet()) {
                out.collect(entry.getValue());
            }

        }

    }

```

分组TopN

在部分场景下，用户希望根据不同的分组进行排序，计算出每个分组的一个排行榜。

```
wcount.keyBy(new TupleKeySelectorByStart()) // 按照首字母分组
                .window(TumblingProcessingTimeWindows.of(Time.seconds(20))) //20s窗口统计上游数据
                .process(new TopNFunction(5)) //分组TopN统计

```

```
private static class TupleKeySelectorByStart implements
            KeySelector<Tuple2<String, Integer>, String> {

        @Override
        public String getKey(Tuple2<String, Integer> value) throws Exception {
            // TODO Auto-generated method stub
            return value.f0.substring(0, 1); //取首字母做key
        }

    }

```

公众号：旧时光大数据

```
/**
     *
     *针对keyby window的TopN函数，继承自ProcessWindowFunction
     *
     */
    private static class TopNFunction
            extends
            ProcessWindowFunction<Tuple2<String, Integer>, Tuple2<String,
Integer>, String, TimeWindow> {

        private int topSize = 10;

        public TopNFunction(int topSize) {
            // TODO Auto-generated constructor stub
            this.topSize = topSize;
        }

        @Override
        public void process(
                String arg0,
                ProcessWindowFunction<Tuple2<String, Integer>, Tuple2<String,
Integer>, String, TimeWindow>.Context arg1,
                Iterable<Tuple2<String, Integer>> input,
                Collector<Tuple2<String, Integer>> out) throws Exception {
            // TODO Auto-generated method stub

            TreeMap<Integer, Tuple2<String, Integer>> treemap = new
TreeMap<Integer, Tuple2<String, Integer>>(
                    new Comparator<Integer>() {

                        @Override
                        public int compare(Integer y, Integer x) {
                            // TODO Auto-generated method stub
                            return (x < y) ? -1 : 1;
                        }

                    });

            for (Tuple2<String, Integer> element : input) {
                treemap.put(element.f1, element);
                if (treemap.size() > topSize) {
                    treemap.pollLastEntry();
                }
            }

            for (Entry<Integer, Tuple2<String, Integer>> entry : treemap
                    .entrySet()) {
                out.collect(entry.getValue());
            }

        }

    }

```

上面的代码实现了按照首字母分组，取每组元素count最高的TopN方法。

嵌套TopN

公众号：旧时光大数据

全局topN的缺陷是，由于windowall是一个全局并发为1的操作，所有的数据只能汇集到一个节点进行TopN 的计算，那么计算能力就会受限于单台机器，容易产生数据热点问题。

解决思路就是使用嵌套 TopN，或者说两层 TopN。在原先的 TopN 前面，再加一层 TopN，用于分散热点。例如可以先加一层分组 TopN，第一层会计算出每一组的 TopN，而后在第二层中进行合并汇总，得到最终的全网TopN。第二层虽然仍是单点，但是大量的计算量由第一层分担了，而第一层是可以水平扩展的。

在实习中一般都怎么用Flink

问过的一些公司：Shopee(2021.08)

参考答案：

一般是双流join或者和维度表join，进行逻辑计算，之后写出到数据库。

下面是一个案例：

事实表通常存储在kafka中，维表通常存储在外部设备中（比如MySQL，HBase）。对于每条流式数据，可以关联一个外部维表数据源，为实时计算提供数据关联查询。维表可能是会不断变化的，在维表join时，需指明这条记录关联维表快照的时刻。需要注意是，目前Flink SQL的维表join仅支持对当前时刻维表快照的关联(处理时间语义)，而不支持事实表rowtime所对应的的维表快照。

1、维度表join

1）预加载维表

将维表全量预加载到内存里去做关联，具体的实现方式就是我们定义一个类，去实现 RichFlatMapFunction，然后在 open 函数中读取维度数据库，再将数据全量的加载到内存，然后在 probe 流上使用算子 ，运行时与内存维度数据做关联。

这个方案的优点就是实现起来比较简单，缺点也比较明显，因为我们要把每个维度数据都加载到内存里面，所以它只支持少量的维度数据。同时如果我们要去更新维表的话，还需要重启作业，所以它在维度数据的更新方面代价是有点高的，而且会造成一段时间的延迟。对于预加载维表来说，它适用的场景就是小维表，变更频率诉求不是很高，且对于变更的及时性的要求也比较低的这种场景。

改进：open()新建一个线程定时加载维表，这样就不需要人工的去重启 Job 来让维度数据做更新，可以实现一个周期性的维度数据的更新。

2）distributed cache

通过 Distributed cash 的机制去分发本地的维度文件到 Task Manager后再加载到内存做关联。实现方式可以分为三步：

- 通过 env.registerCached 注册文件。
- 实现 RichFunction，在 open 函数里面通过 RuntimeContext 来获取 Cache 文件。
- 解析和使用这部分文件数据。

因为数据要加载到内存中，所以支持的数据量比较小。而且如果维度数据需要更新，也是需要重启作业的。

那么它适用的场景就是维度数据是文件形式的、数据量比较小、并且更新的频率也比较低的一些场景，比如说我们读一个静态的码表、配置文件等等。

3）热存储关联

公众号：旧时光大数据

把维度数据导入到像 Redis、Tair、HBase 这样的一些热存储中，然后通过异步 IO 去查询，并且叠加使用 Cache 机制，还可以加一些淘汰的机制，最后将维度数据缓存在内存里，来减轻整体对热存储的访问压力。

如上图展示的这样的一个流程。在 Cache 这块的话，比较推荐谷歌的 Guava Cache，它封装了一些关于 Cache 的一些异步的交互，还有 Cache 淘汰的一些机制，用起来是比较方便的。

异步 IO 可以并行发出多个请求，整个吞吐是比较高的，延迟会相对低很多。如果使用异步 IO 的话，它对于外部存储的吞吐量上升以后，会使得外部存储有比较大的压力，有时也会成为我们整个数据处理上延迟的瓶颈。所以引入 Cache 机制是希望通过 Cache来去减少我们对外部存储的访问量。

这个方案的优点就是维度数据不用全量加载到内存中，不受限于内存大小。但是需要依赖热存储资源，再加上cache过期时间，所以最后结果会有一定的延迟。适用于维度数据量比较大，能接受维度更新有一定延迟的情况。

4）广播维表

利用 Broadcast State 将维度数据流广播到下游 Task 做 Join。

- 将维度数据发送到 Kafka 作为广播原始流 S1
- 定义状态描述符 MapStateDescriptor。调用 S1.broadcast()，获得 broadCastStream S2
- 调用非广播流 S3.connect(S2),得到 BroadcastConnectedStream S4
- 在 KeyedBroadcastProcessFunction/BroadcastProcessFunction 实现关联处理逻辑，并作为参数调用 S4.process()

广播维表维度的变更可以及时的更新到结果，但是数据还是需要保存在内存中，因为它是存在 State 里的，所以支持维表数据量仍然不是很大。适用的场景就是我们需要时时的去感知维度的变更，且维度数据又可以转化为实时流。

5）Temporal table function join

首先说明一下什么是 Temporal table？它其实是一个概念：就是能够返回持续变化表的某一时刻数据内容的视图，持续变化表也就是 Changingtable，可以是一个实时的 Changelog 的数据，也可以是放在外部存储上的一个物化的维表。

它的实现是通过 UDTF 去做 probe 流和 Temporal table 的 join，称之 Temporal table function join。这种 Join 的方式，它适用的场景是维度数据为 Changelog 流的形式，而且我们有需要按时间版本去关联的诉求。

在 Changelog 流上面去定义 TemporalTableFunction，这里面有两个关键的参数是必要的。第1个参数就是能够帮我们去识别版本信息的一个 Time attribute，第 2 个参数是需要去做关联的组件。

在 tableEnv 里面去注册 TemporalTableFunction 的名字。

维表join方案对比

公众号：旧时光大数据

	预加载到内存	热存储关联	广播维表	Temporal table function jsoin
实现复杂度	低	中	低	低
维表数据量	低	高	低	高
维表更新频率	低	中	高	高
维表更新实时性	低	中	高	高
维表形式		热存储	实时流	实时流
是否依然外部存储	低	是	否	否

2、双流join

批处理有两种方式处理两个表的Join，**一种是基于排序的Sort-Merge Join，另一种是转化为Hash Table加载到内存里做Hash Join。**

在双流Join的场景中，Join的对象是两个流，数据是不断进入的，所以我们Join的结果也是需要持续更新的。基本思路是将一个无线的数据流，尽可能拆分成有限数据集去做Join。

1）Regular Join

这种 Join 方式需要去保留两个流的状态，持续性地保留并且不会去做清除。两边的数据对于对方的流都是所有可见的，所以数据就需要持续性的存在 State 里面，那么 State 又不能存的过大，因此这个场景的只适合有界数据流。

2）Interval Join

加入了一个时间窗口的限定，要求在两个流做 Join 的时候，其中一个流必须落在另一个流的时间戳的一定时间范围内，并且它们的 Join key 相同才能够完成 Join。加入了时间窗口的限定，就使得我们可以对超出时间范围的数据做一个清理，这样的话就不需要去保留全量的 State。

Interval Join 是同时支持 processing time 和 even time去定义时间的。如果使用的是 processing time，Flink 内部会使用系统时间去划分窗口，并且去做相关的 state 清理。如果使用 even time 就会利用 Watermark 的机制去划分窗口，并且做 State 清理。

3）Window join

将两个流中有相同 key 和处在相同 window 里的元素去做 Join。它的执行的逻辑比较像 Inner Join，必须同时满足 Join key 相同，而且在同一个 Window 里元素才能够在最终结果中输出。

Savepoint知道是什么吗

问过的一些公司：Shopee(2021.08)

参考答案：

Savepoint在Flink中叫做**保存点**，是**基于Flink检查点机制的应用完整快照备份机制**，用来保存状态，可以在另一个集群或者另一个时间点，从保存的状态中将作业恢复回来，适用于应用升级、集群迁移、Flink集群版本更新、A/B测试以及假定场景、暂停和重启、归档等场景。保存点可以视为一个 （算子ID→State）的Map，对于每一个有状态的算子，Key是算子ID，Value是算子的State。

在作业恢复方面，Flink提供了应用自动容错机制，可以减少人为干预，降低运维复杂度。同时为了提高灵活度，也提供了手动恢复。Flink提供了**外部检查点**和**保存点**两种手动作业恢复方式。**这里说下保存点恢复方式。**

保存点恢复方式

用户通过命令触发，由用户手动创建、清理。使用了标准化格式存储，允许作业升级或者配置变更。用户在恢复时需要提供用于恢复作业状态的保存点路径。

公众号：旧时光大数据

其实，从保存点恢复作业并不简单，尤其是在作业变更（如修改逻辑、修复bug）的情况下，需要考虑如下几点。

1）算子的顺序改变

如果对应的UID没变，则可以恢复，如果对应的UID变了则恢复失败。

2）作业中添加了新的算子

如果是无状态算子，没有影响，可以正常恢复，如果是有状态的算子，跟无状态的算子一样处理。

3）从作业中删除了一个有状态的算子

默认需要恢复保存点中所记录的所有算子的状态，如果删除了一个有状态的算子，从保存点恢复的时候被删除的OperatorID找不到，所以会报错，可以通过在命令中添加-allowNonRestoredState （short: -n）跳过无法恢复的算子。

4）添加和删除无状态的算子

如果手动设置了UID，则可以恢复，保存点中不记录无状态的算子，如果是自动分配的UID，那么有状态算子的UID可能会变（Flink使用一个单调递增的计数器生成UID，DAG改版，计数器极有可能会变），很有可能恢复失败。

5）恢复的时候调整并行度

Flink1.2.0及以上版本,如果没有使用作废的API，则没问题；1.2.0以下版本需要首先升级到1.2.0才可以。

为什么用Flink不用别的微批考虑过吗

问过的一些公司：百度(2021.09)

参考答案：

mini-batch模式的处理过程如下：

- 在数据流中收集记录；
- 收集若干记录后，调度一个批处理作业进行数据处理；
- 在批处理运行的同时，收集下一批次的记录。

也就是说**Spark为了处理一个mini-batch，需要调度一个批处理作业，相比于Flink延迟较大，Spark的处理延迟在秒级。**而Flink只需启动一个流计算拓扑，处理持续不断的数据，**Flink的处理延迟在毫秒级别。**如果计算中涉及到多个网络Shuffle，Spark Streaming和Flink之间的延迟差距会进一步拉大.

解释一下啥叫背压

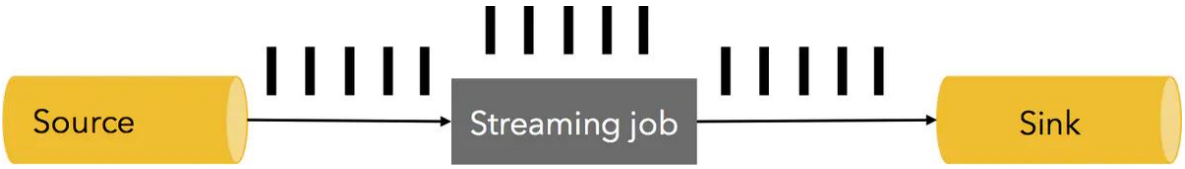
问过的一些公司：美团(2021.08)

参考答案：

背压是**指系统在一个临时负载峰值期间接收数据的速率大于其处理速率的一种场景**（备注:就是处理速度慢，接收速度快，系统处理不了接收的数据）。许多日常情况都会导致背压。例如，垃圾回收卡顿可能导致流入的数据堆积起来，或者数据源可能出现发送数据过快的峰值。如果处理不当，背压会导致资源耗尽，甚至导致数据丢失。

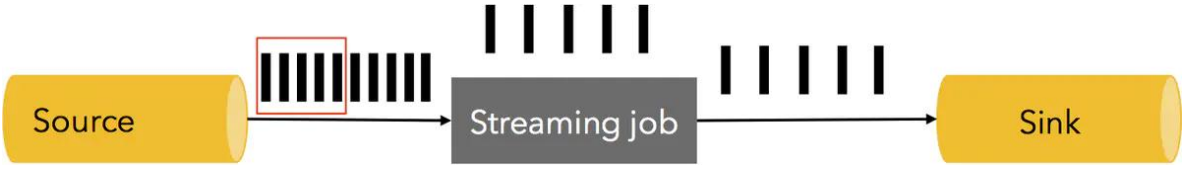
看一个简单的例子。假设数据流 pipeline（抽象为 Source，Streaming job 和 Sink）在稳定状态下以每秒500万个元素的速度处理数据，如下所示正常情况（一个黑色条代表100万个元素，下图表示系统1秒内的快照）：

公众号：旧时光大数据



上述情况没有Back Pressure

如果 Source 发送数据的速度在某个时刻达到了峰值，每秒生成的数据达到了双倍，下游的处理能力不变：



上述情况存在Back Pressure

消息处理速度 < 消息的发送速度，消息拥堵，系统运行不畅。如何处理这种情况？

- 可以去掉这些元素，但是，对于许多流应用程序来说，数据丢失是不可接受的。
- 将拥堵的消息缓存起来，并告知消息发送者减缓消息发送的速度。消息缓存应该是持久的，因为在发生故障的情况下，需要重放这些数据以防止数据丢失。

上述情况是Buffer records

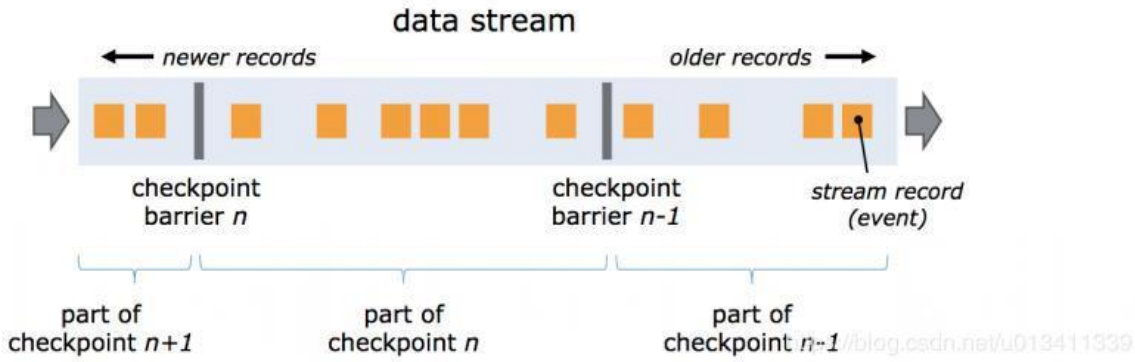
Flink分布式快照

问过的一些公司：美团(2021.08)

参考答案：

分布式快照可以将同一时间点Task/Operator的状态数据全局统一快照处理，包括Keyed State和Operator State。

Flink的分布式快照是根据Chandy-Lamport算法量身定做的。简单来说就是持续创建分布式数据流及其状态的一致快照。



核心思想是在 input source 端插入 barrier，控制 barrier 的同步来实现 snapshot 的备份和 exactly-once 语义。

公众号：旧时光大数据

Flink SQL解析过程

问过的一些公司：网易严选(2021.09)

参考答案：

首先，先了解下Calcite是什么。

1、Apache Calcite是什么

Apache Calcite是一个动**态数据管理框架**，它具备很多典型数据库管理系统的功能，如SQL解析、SQL校验、SQL查询优化、SQL生成以及数据连接查询等，但是又省略了一些关键的功能，如Calcite并不存储相关的元数据和基本数据，不完全包含相关处理数据的算法等。

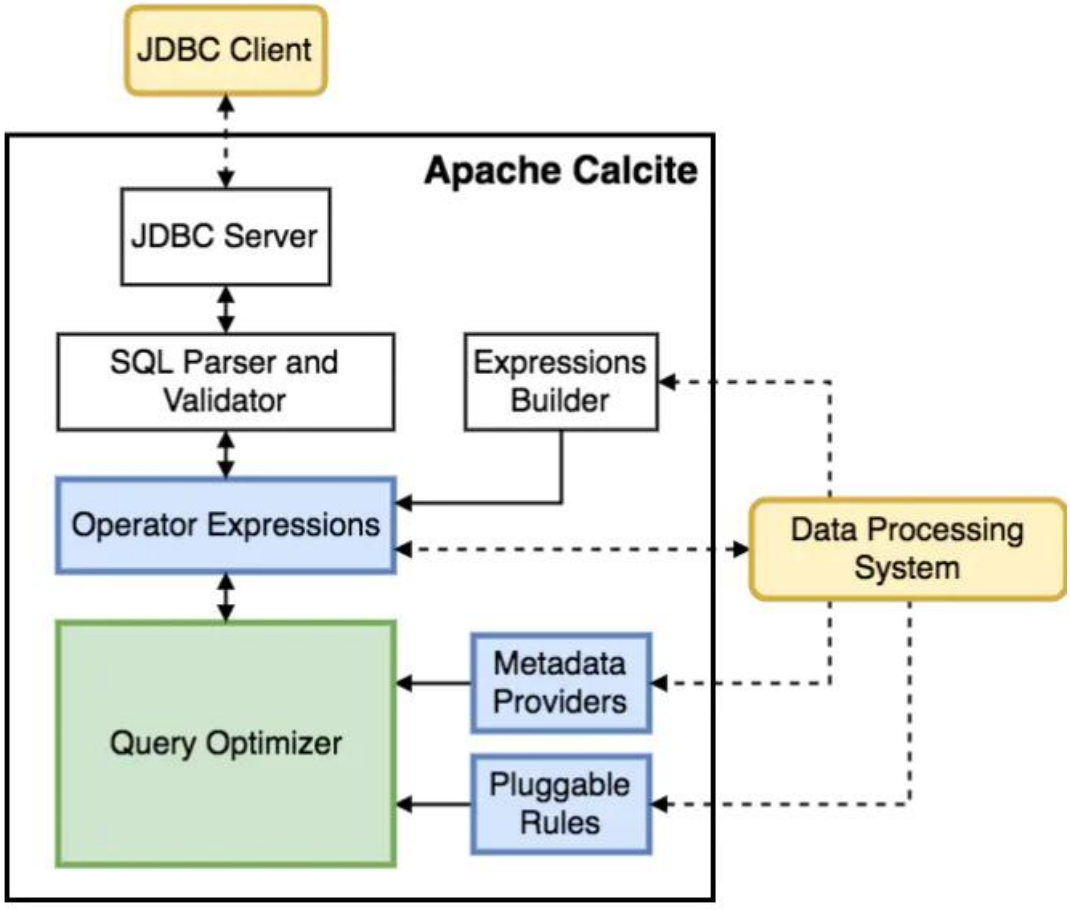
Calcite采用的是业界大数据查询框架的一种通用思路，它的目标是“one size fits all（一种方案适应所有需求场景）”，希望能为不同计算平台和数据源提供统一的查询引擎。

Calcite作为一个强大的SQL计算引擎，**在Flink内部的SQL引擎模块就是基于Calcite。**

2、Calcite的特点

- 支持标准SQL语言；
- 独立于编程语言和数据源，可以支持不同的前端和后端；
- 支持关系代数、可定制的逻辑规则和基于成本模型优化的查询引擎；
- 支持物化视图（materialized view）的管理（创建、丢弃、持久化和自动识别）；
- 基于物化视图的Lattice和Tile机制，以应用于OLAP分析；
- 支持对流数据的查询。

3、Calcite的功能



1）SQL解析

公众号：旧时光大数据

Calcite的SQL解析是通过JavaCC实现的，使用JavaCC编写SQL语法描述文件，将SQL解析成未经校验的AST语法树。

2）SQL效验

校验分两部分：

- 无状态的校验：即验证SQL语句是否符合规范。
- 有状态的校验：即通过与元数据结合验证SQL中的Schema、Field、Function是否存在，输入输出类型是否匹配等。

3）SQL查询优化

对上个步骤的输出（RelNode，逻辑计划树）进行优化，得到优化后的物理执行计划。优化有两种：基于规则的优化和基于代价的优化。

4）SQL生成

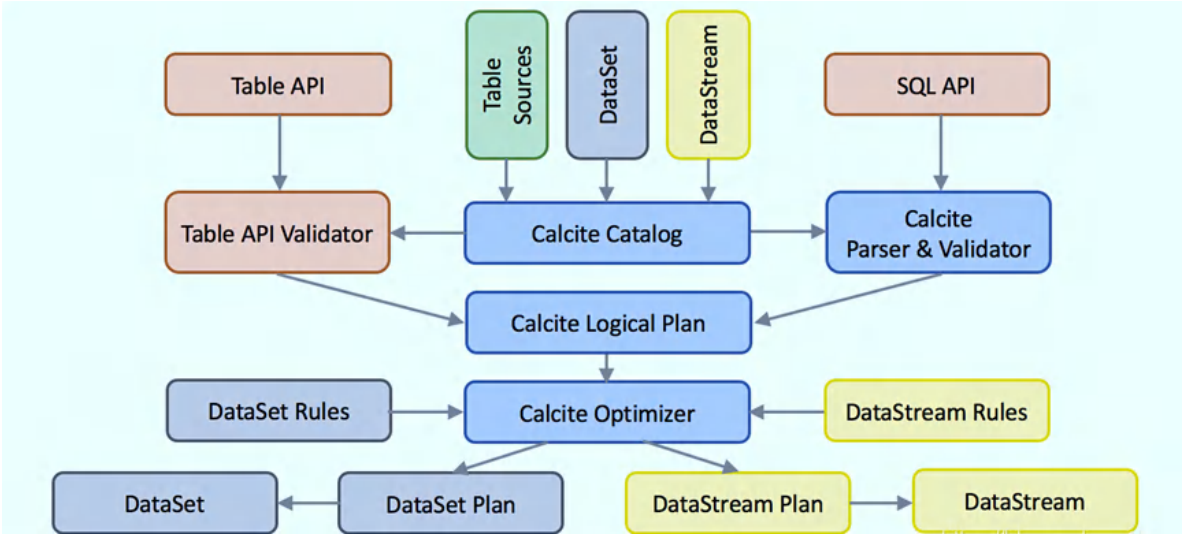
将物理执行计划生成为在特定平台/引擎的可执行程序，如生成符合MySQL或Oracle等不同平台规则的SQL查询语句等。

5）数据连接与执行

通过各个执行平台执行查询，得到输出结果。

在Flink或者其他使用Calcite的大数据引擎中，一般到SQL查询优化即结束，由各个平台结合Calcite的SQL代码生成和平台实现的代码生成，将优化后的物理执行计划组合成可执行的代码，然后在内存中编译执行。

4、Flink SQL结合Calcite（Flink SQL解析）



一条SQL从提交到Calcite解析，优化，到最后的Flink执行，一般分以下过程：

1）Sql Parser： 将sql语句通过java cc解析成AST（语法树），在calcite中用SqlNode表示AST；

2）Sql Validator： 结合数字字典（catalog）去验证sql语法；

3）生成Logical Plan： 将sqlNode表示的AST转换成LogicalPlan， 用relNode表示;

4）生成 optimized LogicalPlan： 先基于calcite rules 去优化logical Plan，基于flink定制的一些优化rules去优化logical Plan；

5）生成Flink PhysicalPlan： 这里也是基于flink里头的rules将，将optimized LogicalPlan转成成Flink的物理执行计划；

6）将物理执行计划转成Flink ExecutionPlan：就是调用相应的tanslateToPlan方法转换和利用CodeGen元编程成Flink的各种算子。

公众号：旧时光大数据

这里再提一下**SQL的优化**：

5、SQL查询优化器

SQL优化的发展，则可以分为两个阶段，即**RBO（基于规则）**，和**CBO（基于代价）**。

逻辑优化使用Calcite的Hep优化器（基于规则），物理优化阶段使用了Calcite的Hep规则优化器和Volcano优化器（基于代价）。

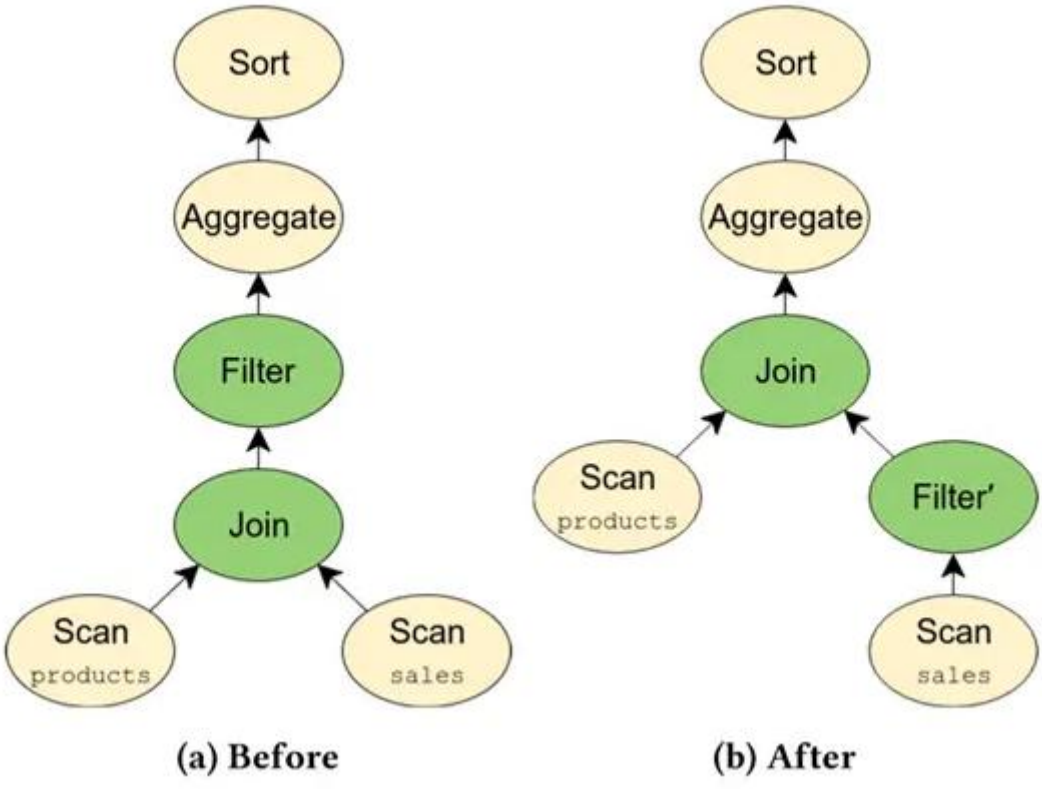
1） **RBO（基于规则的优化器）**会将原有表达式裁剪掉，遍历一系列规则（Rule），只要满足条件就转换，生成最终的执行计划。一些常见的规则包括分区裁剪（Partition Prune）、列裁剪、谓词下推（Predicate Pushdown）、投影下推（Projection Pushdown）、聚合下推、limit下推、sort下推、常量折叠（Constant Folding）、子查询内联转join等。

2） **CBO（基于代价的优化器）**会将原有表达式保留，基于统计信息和代价模型，尝试探索生成等价关系表达式，最终取代价最小的执行计划。CBO的实现有两种模型，Volcano模型，Cascades模型。这两种模型思想很是相似，不同点在于Cascades模型一边遍历SQL逻辑树，一边优化，从而进一步裁剪掉一些执行计划。

看一个案例：

RBO（基于规则）优化

RBO主要是开发人员在使用SQL的过程中，有些发现有些通用的规则，可以显著提高SQL执行的效率，比如最经典的Filter下推：



将Filter下推到Join之前执行，这样做的好处是减少了Join的数量，同时降低了CPU，内存，网络等方面的开销，提高效率。

RBO和CBO的区别大概在于：RBO只为应用提供的rule，而CBO会根据给出的Cost信息，智能应用rule，求出一个Cost最低的执行计划。需要纠正很多人误区的一点是，CBO其实也是基于rule的，接触到RBO和CBO这两个概念的时候，很容易将他们对立起来。但实际上CBO，可以理解为就是加上Cost的RBO。

目前各大数据库和计算引擎倾向于CBO。

公众号：旧时光大数据

Flink on YARN模式

问过的一些公司：米哈游(2021.09)

参考答案：

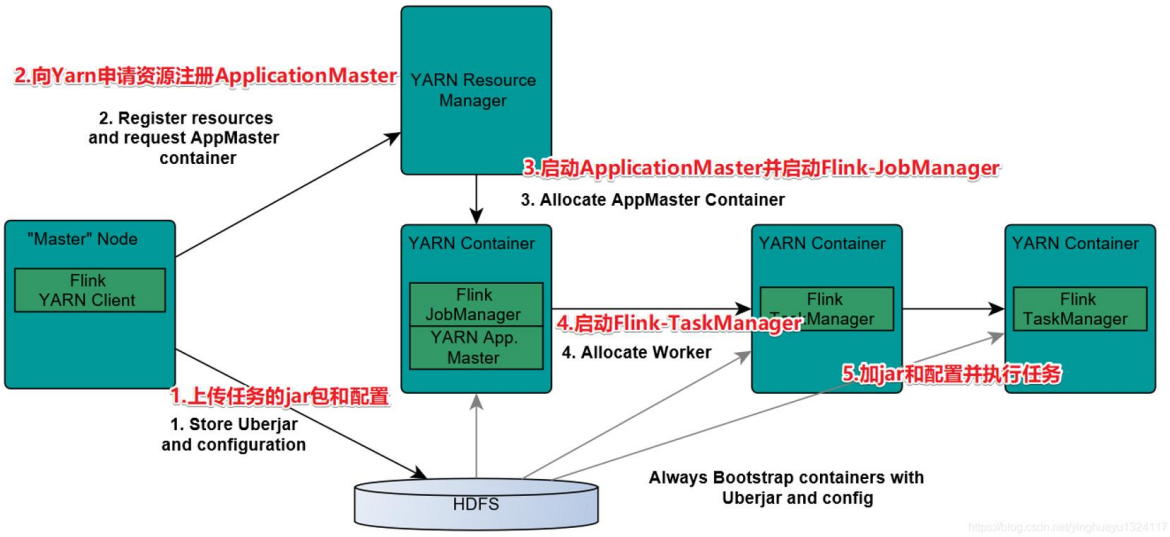
Flink支持多种部署模式：

1）Standalone模式：Flink安装在普通的Linux机器上，或者安装在K8s中，集群的资源由Flink自行管理。

2）Yarn、Mesos、K8s等资源管理集群模式：Flink向资源集群申请资源，创建Flink集群。

3）云上模式：Flink可以在Google、亚马逊云计算平台上轻松部署。

Flink on Yarn交互过程如下：



1）Client上传Flink的jar包和配置文件到HDFS集群上

2）Client向Yarn的ResourceManager提交任务和申请资源

3）ResourceManager分配Container资源并启动ApplicationMaster

4）ApplicationMaster加载Flink的jar包和配置文件构建环境启动Flink-JobManager

5）ApplicationMaster向ResourceManager申请任务资源

6）NodeManager加载Flink的jar包和配置文件构建环境并启动TaskManager

7）TaskManager启动后会向JobManager发送心跳，并等待JobManager向其分配任务

Flink On Yarn模式的两种方式：**Session模式**和**Per-Job模式**

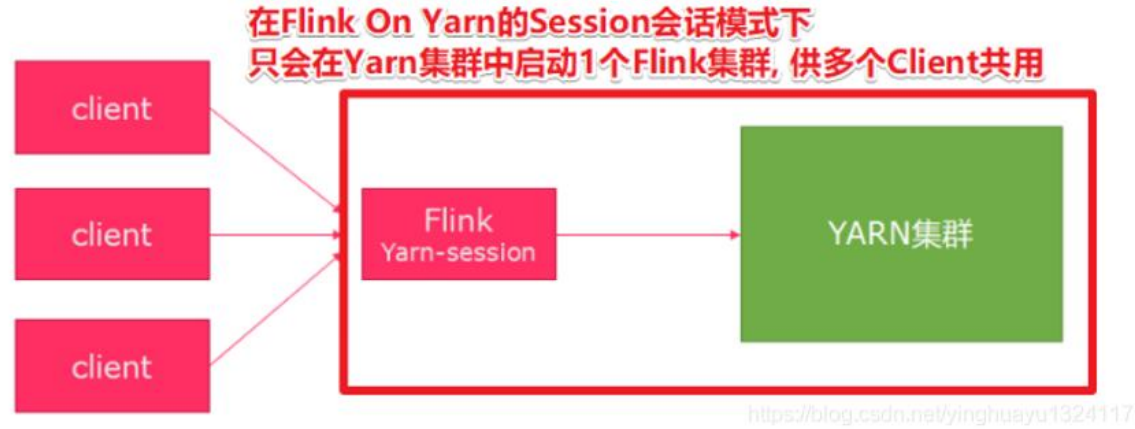
1、Session模式（适合小任务使用）

需要先申请资源，启动JobManager和TaskManager

不需要每次提交任务再去申请资源，而是使用已经申请好的资源，从而提高执行效率

公众号：旧时光大数据

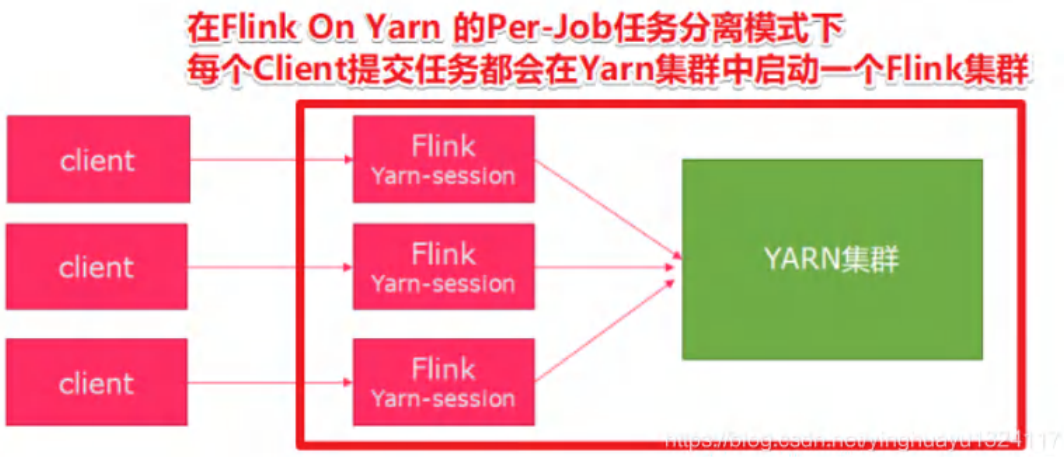
任务提交完资源不会被释放，因此一直会占用资源



2、Per-Job模式：（适合使用大任务，且资源充足）

每次提交任务都需要去申请资源，申请资源需要时间，所有影响执行效率（但是在大数据面前都是小事）

每次执行完任务资源就会立刻被释放，不会占用资源



Flink如何保证数据不丢失

问过的一些公司：京东(2021.09)

参考答案：

Checkpoint（检查点）是Flink实现应用容错的核心机制。

Flink根据配置周期性通知Stream中各个算子的状态来**生成检查点快照**，从而将这些状态数据定期持久化存储下来，**Flink程序一旦意外崩溃，重新运行程序时可以有选择地从这些快照进行恢复，将应用恢复到最后一次快照的状态**，从此刻开始重新执行，**避免数据的丢失、重复。**

默认情况下，如果设置了检查点选项，则Flink只保留最近成功生成的一个检查点，而当Flink程序失败时，可以从最近的这个检查点来进行恢复。但是，如果希望保留多个检查点，并能够根据实际需要选择其中一个进行恢复，会更加灵活。

默认情况下，检查点不会被保留，取消程序时即会删除它们，但是可以通过配置保留定期检查点，根据配置，当作业失败或者取消的时候，不会自动清除这些保留的检查点。

如果想保留检查点，那么Flink也设计了相关实现，可选项如下。

- ExternalizedCheckpointCleanup. RETAIN_ON_CANCELLATION：取消作业时保留检查点。在这种情况下，必须在取消后手动清理检查点状态。

公众号：旧时光大数据

- ExternalizedCheckpointCleanup. DELETE_ON_CANCELLATION：取消作业时删除检查点。只有在作业失败时检查点状态才可用。

数据仓库面试题

介绍下数据仓库

可回答：对数据仓库的理解

问过的一些公司：京东，美团，网易，阿里(2021.09)，网易有道(2021.10)

参考答案：

数据仓库的发展大致经历了这样的三个过程：

简单报表阶段：这个阶段，系统的主要目标是解决一些日常的工作中业务人员需要的报表，以及生成一些简单的能够帮助领导进行决策所 需要的汇总数据。大部分表现形式为数据库和前端报表工具。

数据集市阶段：这个阶段，主要是根据某个业务部门的需要，进行一定的数据的采集，整理，按照业务人员的需要，进行多维报表的展现，能够提供对特定业务指导的数据，并且能够提供特定的领导决策数据。

数据仓库阶段：这个阶段，主要是按照一定的数据模型，对整个企业的数据进行采集，整理，并且能够按照各个业务部门的需要，提供跨部门的，完全一致的业务报表数据，能够通过数据仓库生成对对业务具有指导性的数据，同时，为领导决策提供全面的数据支持。

首先，我们先来看下数据库、数据集市、数据仓库以及数据湖的概念。

1、什么是数据库?

数据库（Database）是按照一定格式和数据结构在计算机保存数据的软件，属于物理层。

最早期是广义上的数据库，这个阶段的数据库结构主要以层次或网状的为主，这是数据库的数据和程序间具备非常强的依赖性，应用有一定局限性。

我们现在所说的数据库一般指的是关系型数据库。关系数据库是指采用了关系模型来组织数据的数据库，其**以行和列的形式存储数据，具有结构化程度高，独立性强，冗余度低等优点。**

关系型数据库主要用于联机事务处理OLTP（On-Line Transaction Processing），主要用于进行基本的、日常的事务处理，例如银行交易等场景。

2、什么是数据集市?

数据集市是一种微型的数据仓库，它通常是有更少的数据，更少的主题区域，以及更少的历史数据，如果数据仓库是企业级的，那数据集市就是部门级的，一般数据集市只能为某个局部范围内的管理人员服务。

3、什么是数据仓库?

数据仓库（Data Warehouse），可简写为DW或DWH。它是为企业所有级别的决策制定过程，提供所有类型数据支持的战略集合。它是单个数据存储，出于分析性报告和决策支持目的而创建。为需要业务智能的企业，提供指导业务流程改进、监视时间、成本、质量以及控制。

数据仓库之父比尔·恩门于1990年提出数据仓库（Data Warehouse），数仓主要是为解决企业的数据集成与分析问题。数据仓库主要功能是将OLTP经年累月所累积的大量数据，通过数据仓库特有的数据储存架构进行OLAP，最终帮助决策者能快速有效地从大量数据中，分析出有价值的信息，提供决策支持。自从数据仓库出现之后，信息产业就开始从以关系型数据库为基础的运营式系统慢慢向决策支持系统发展。

公众号：旧时光大数据

一句话总结：**数据仓库存在的意义在于对企业的所有数据进行汇总，为企业各个部门提供统一的，规范的数据出口。**

数据仓库相比数据库，主要有以下两个特点：

- **数据仓库是面向主题集成的。**数据仓库是为了支撑各种业务而建立的，数据来自于分散的操作型数据。因此需要将所需数据从多个异构的数据源中抽取出来，进行加工与集成，按照主题进行重组，最终进入数据仓库。
- **数据仓库主要用于支撑企业决策分析，所涉及的数据操作主要是数据查询。**因此数据仓库通过表结构优化、存储方式优化等方式提高查询速度、降低开销。

数据仓库与数据库的对比

维度	数据仓库	数据库
应用场景	OLAP	OLTP
数据来源	多数据源	单数据源
数据标准化	非标准化Schema	高度标准化的静态Schema
数据读取优势	针对读操作进行优化	针对写操作进行优化

4、什么是数据湖?

在现在这个时代，数据对于企业而言，已经是一种重要资产。随着企业的不断发展，数据不断堆积，企业希望把生产经营中的所有相关数据都完整保存下来，进行有效管理与集中治理，挖掘和探索数据价值。而数据湖就应运而生。

数据湖是一个集中存储各类结构化和非结构化数据的大型数据仓库，它可以存储来自多个数据源、多种数据类型的原始数据，数据无需经过结构化处理，就可以进行存取、处理、分析和传输。数据湖能帮助企业快速完成异构数据源的联邦分析、挖掘和探索数据价值。

数据湖的本质，是由“**数据存储架构+数据处理工具**”组成的解决方案。

- **数据存储架构**：要有足够的扩展性和可靠性，可以存储海量的任意类型的数据，包括结构化、半结构化和非结构化数据。
- **数据处理工具**，则分为两大类：
    - 第一类工具，**聚焦如何把数据“搬到”湖里。**包括定义数据源、制定数据同步策略、移动数据、编制数据目录等。
    - 第二类工具，**关注如何对湖中的数据进行分析、挖掘、利用。**数据湖需要具备完善的数据管理能力、多样化的数据分析能力、全面的数据生命周期管理能力、安全的数据获取和数据发布能力。如果没有这些数据治理工具，元数据缺失，湖里的数据质量就没法保障，最终会由数据湖变质为数据沼泽。

数据仓库和数据湖的不同类比于仓库和湖泊：仓库存储着来自特定来源的货物；而湖泊的水来自河流、溪流和其他来源，并且是原始数据。

数据湖与数据仓库的对比

公众号：旧时光大数据

维度	数据湖	数据仓库
应用场景	可以探索性分析所有类型的数据，包括机器学习、数据发现、特征分析、预测等	通过历史的结构化数据进行数据分析
使用成本	起步成本低，后期成本较高	起步成本高，后期成本较低
数据质量	包含大量原始数据，使用前需要清洗和标准化处理	质量高，可作为事实依据
适用对象	数据科学家、数据开发人员为主	业务分析师为主

5、数据仓库特点

1）数据仓库是面向主题的

数据仓库中的数据是按照一定的主题域进行组织的，每一个主题对应一个宏观的分析领域。数据仓库排除对于决策无用的数据，提供特定主题的简明视图。

举个例子：

比如说一个公司会有很多的部门，不同的部门都会去数据仓库拿数据，做自己要做的报表，我们把这一个部门或是某一个业务，也就是独立从我们数据仓库中获取数据的单元，把它称作为主题，也可以理解为一个主题就是一个部门。这个部门作为一个主题会从数据仓库总去获取数据，用于完成需要的报表。

2）数据仓库是集成的

数据仓库中的数据不是一开始就是在里面的，而是从各个分散的数据库中抽取出来的。但是有一个问题，就是这些来自不同数据库的数据会有重复和不一样的地方，如字段的同名异议、异名同义、单位不统一，字长不统一等。所以在集成的过程中，还要对数据进行清洗、规划、去敏等操作。

一句话就是，数据仓库是对企业内不同业务部门数据完整集合，而且还是处理过的数据。

3）数据仓库的数据是稳定的

数据仓库中的数据主要是为了给企业做决策时分析使用，涉及的主要是对数据的查询，一般情况下不会对数据进行修改，如果数据仓库中的历史数据超过存储期限，则会直接删除。

因为数据仓库涉及的操作主要是查询，所以它的系统要比数据库简单很多，但是数据仓库涉及到查询的数据量一般都很大，所以在数据查询就有更高的要求。

一句话记忆，数仓里不存在数据的更新和删除（不是指数据到期的删除）操作。

4）数据仓库中的数据是随时间变化而变化的

数据仓库中的数据不可更新是针对应用来说的，也就是说，数据仓库的用户进行分析处理是不进行数据更新操作的。但并不是说，在从数据集成输入数据仓库开始到最后被删除的整个生存周期中，所有的数据仓库数据都是永远不变的。

数据仓库的数据是随着时间变化而变化的主要表现如下：

- 数据仓库随着时间变化不断增加新的数据内容。数据仓库系统必须不断捕捉OLTP数据库中变化的数据，追加到数据仓库当中去，也就是要不断的生成OLTP数据库的快照，经统一集成增加到数据仓库中去；但对于确实不在变化的数据库快照，如果捕捉到新的变化数据，则只生成一个新的数据库快照增加进去，而不会对原有的数据库快照进行修改。
- 数据库随着时间变化不断删去旧的数据内容 。数据仓库内的数据也有存储期限，一旦过了这一期限，过期数据就要被删除。

公众号：旧时光大数据

- 数据仓库中包含有大量的综合数据，这些综合数据中很多跟时间有关，如数据经常按照时间段进行综合，或隔一定的时间片进行抽样等等。这些数据要随着时间的变化不断地进行从新综合。因此数据仓库的数据特征都包含时间项，以标明数据的历史时期。

一句话理解，数仓里会完整的记录某个对象在一段时期内的变化情况。

数仓的基本原理

问过的一些公司：ebay

参考答案：

1、数据仓库

数据仓库（Data Warehouse），可简写为DW或DWH。它是为企业所有级别的决策制定过程，提供所有类型数据支持的战略集合。它是单个数据存储，出于分析性报告和决策支持目的而创建。为需要业务智能的企业，提供指导业务流程改进、监视时间、成本、质量以及控制。

数据仓库存在的意义在于对企业的所有数据进行汇总，为企业各个部门提供统一的， 规范的数据出口。

2、数仓分层

第一层：

ODS——原始数据层：存放原始数据

第二层：

DWD——数据明细层：对ODS层数据进行清洗、维度退化、脱敏等。

第三层：

DWS——数据汇总层： 对DWD层数据进行一个轻度的汇总。

第四层：

DM——数据集市层：为各种统计报表提供数据。

数仓架构

可回答：数据仓库架构及建设

问过的一些公司：美团x2，vivo，58x2，作业帮，有赞

参考答案：

架构是数据仓库建设的总体规划，从整体视角描述了解决方案的高层模型，描述了各个子系统的功能以及关系，描述了数据从源系统到决策系统的数据流程。**业务需求回答了要做什么，架构就是回答怎么做的问题。**

1、架构的价值

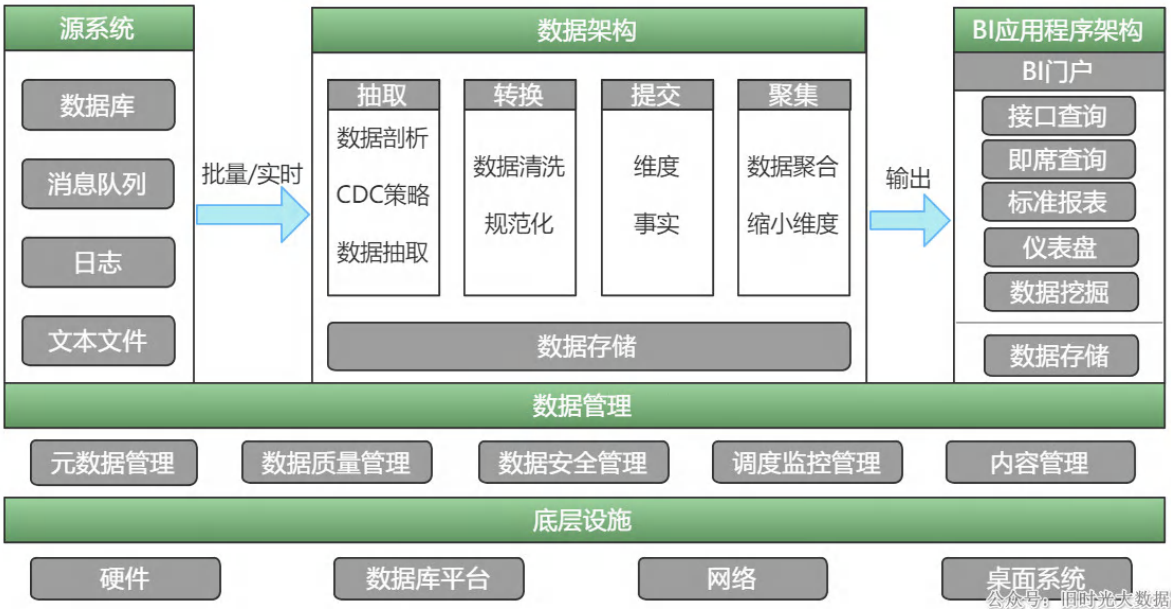
- 最大满足需求
- 便于团队沟通
- 便于进行规划
- 提高灵活性、生成率
- 便于维护
- 便于团队学习

公众号：旧时光大数据

2、数据仓库架构

数据仓库的核心功能从源系统抽取数据，通过清洗、转换、标准化，将数据加载到BI平台，进而满足业务用户的数据分析和决策支持。

数据仓库架构包含三个部分：**数据架构、应用程序架构、底层设施。**



1）底层设施

底层设施为架构提供了基础，底层设施包括硬件、数据库平台、网络和桌面系统。

（1）硬件

硬件主要指服务器硬件，主要有数据库服务器、ETL服务器、调度服务器、报表服务器、BI门户服务器、接口服务器。

（2）数据库平台

数据库平台分为二大类：联机事务处理OLTP（on-line transaction processing）、联机分析处理OLAP（On-Line Analytical Processing），主要有Oracel，MySQL。OLAP是为数据分析而设计的数据库管理系统。主要有Teradata, Greenplum，Hive，Kudu。

（3）桌面系统

数据仓库不同的应用对桌面系统也有不同的要求，开发工具主要有Window、Mac面系统，部署服务器主要有Unix桌面系统，系统BI应用程序主要有Window、Mac、移动设备桌面系统。

（4）网络

网络是底层设施的基础，特别是大数据时代对网络的要求越来越高。

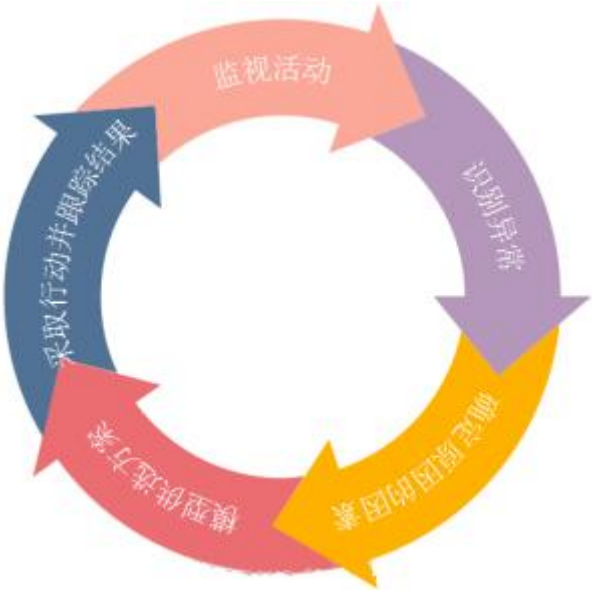
2）BI应用程序架构

数据仓库是数据处理的后台，业务用户并不关心后台怎么处理。BI应用是数据呈现的前台，是业务用户进行查询的入口。BI应用程序的体验也是衡量数据仓库是否成功的主要因素。

（1）BI分析周期

业务分析从监视活动开始识别某个问题或时机，进而采取行动，最终回到监视该活动产生的结果上来，达到数据驱动业务增长的目的。分析周期把这个过程分为五个不同的阶段。

公众号：旧时光大数据



（2）BI应用分类

接口查询

- 数据以接口的形式提供给上下游系统，供上下业务系统进行查询。主要有推和拉二种模式。

即席查询

- 业务用户根据自己的需求，自定义查询请求，后台自动组织SQL语句访问维度模型。

标准报表

- 根据业务用户的需求，进行定制报表。

仪表盘

- 它是向企业展示度量信息和关键业务指标现状的数据可视化工具。

数据挖掘

- 为数据挖掘工具提供标准基础数据。

运营查询

- 为了减少业务系统的大数据量查询压力，数据仓库为业务系统提供实时的查询。

（3）数据存储

- 为了提高查询性能，BI工具需要把数据存储在本地服务器上
- OLAP多维模型需要把数据存储成Cube格式
- 把数据存储成文件格式，放在其他服务器上

3）数据结构

数据架构主要描述数据从源系统抽取数据，然后经过清洗、规范化、提交形成标准模型，最终提交给业务用户，以及对数据的管理。

（1）源系统

数据仓库一般会面临多个、异构数据源的问题，主要分为结构化，半结构化以及非结构化数据。为了便于管理需要对源系统建立元数据信息。

（2）抽取

因为源系统的多样性，源抽取阶段一般选择使用工具。在抽取之前还要做以下工作：

公众号：旧时光大数据

数据剖析是对数据的技术性分析，对数据的内容、一致性和结构进行描述。对源系统的数据质量进行评估。

- 数据剖析
- 变化数据捕获策略

为了减少对源系统的影响，一般只抽取变化的数据，也需要识别物理删除的数据。CDC策略主要有：

添加审计列

- 在源系统追加日期字段，当数据发生变化的时候，系统会自动更新该值。如果由后台人员手工修改数据，可能就发生遗漏。

数据比较

- 比较源系统和数据仓库的数据，只抽取变化的数据。这种方法需要全量的数据，比较耗费资源。可以视数据量的大小而定。

读取日志

- 读取数据库操作日志信息，同步到数据仓库中。一般日志的有效期比较短，一旦发生要重跑的情况，可能以前的日志已经被清空了。

消息队列

- 把事务信息放到消息队列里，以流的形式同步到数据仓库。这种方式即可以减轻源系统的压力，又能做到实时同步。

（3）数据转换

数据从源系统抽取过来之后，就要进入数据转换阶段。 这一阶段是数据仓库开发核心阶段。主要有以下步骤：

清洗

数据清洗是制定转换规则，筛选数据并纠正数据的过程。清洗的目的是改进源系统的数据质量，但是不要在数据仓库做过多的清洗，源系统的数据质量应该在源头处理。清洗的主要内容包括：

- 制定数据转换规则
- 提交错误事实表

规范化

规范化就是整合各个源系统的数据，把数据统一命名，统一取值，建立企业标准版本数据。主要内容包括：

- 数据标准化
- 删除重复数据
- 数据共存策略

（4）提交

提交就要根据维度模型生成维度表和事实表。 提交主要内容包括：

- 选择合适的缓慢变化维类型
- 为维表生成代理键
- 管理不同粒度的层次维
- 管理专项维
- 生成维度桥接表
- 生成代理键管道
- 选择合适的事实表类型
- 处理延迟到达的事实

公众号：旧时光大数据

- 生成维度表
- 生成事实表

（5）聚集

聚集是指根据事务事实表进行更高粒度的聚合以及生成相对应的维度表。主要内容包括：

- 数据聚合
- 缩小维度表
- 生成OLAP多维数据集

（6）数据存储

数据存储是指在在数据的生命周期内对数据的管理，主要内容包括：

- 数据备份
- 历史数据归档
- ETL过程中数据分层存储

3、数仓架构设计

1）数据设计方法

数据仓库建立之前，就必须考虑其实现方法，通常有自顶向下、自底向上和两者结合进行的这样三种实现方案。

（1）自顶向下实现

自顶向下的实现需要在项目开始时完成更多计划和设计工作，这就需要涉及参与数据仓库实现的每个工作组、部门或业务线中的人员。要使用的数据源、安全性、数据结构、数据质量、数据标准和整个数据模型的有关决策一般需要在真正的实现开始之前就完成。

（2）自底向上实现

自底向上的实现包含数据仓库的规划和设计，无需等待安置好更大业务范围的数据仓库设计。这并不意味着不会开发更大业务范围的数据仓库设计；随着初始数据仓库实现的扩展，将逐渐增加对它的构建。现在，该方法得到了比自顶向下方法更广泛的接受，因为数据仓库的直接结果可以实现，并可以用作扩展更大业务范围实现的证明。

（3）两者结合的折中实现

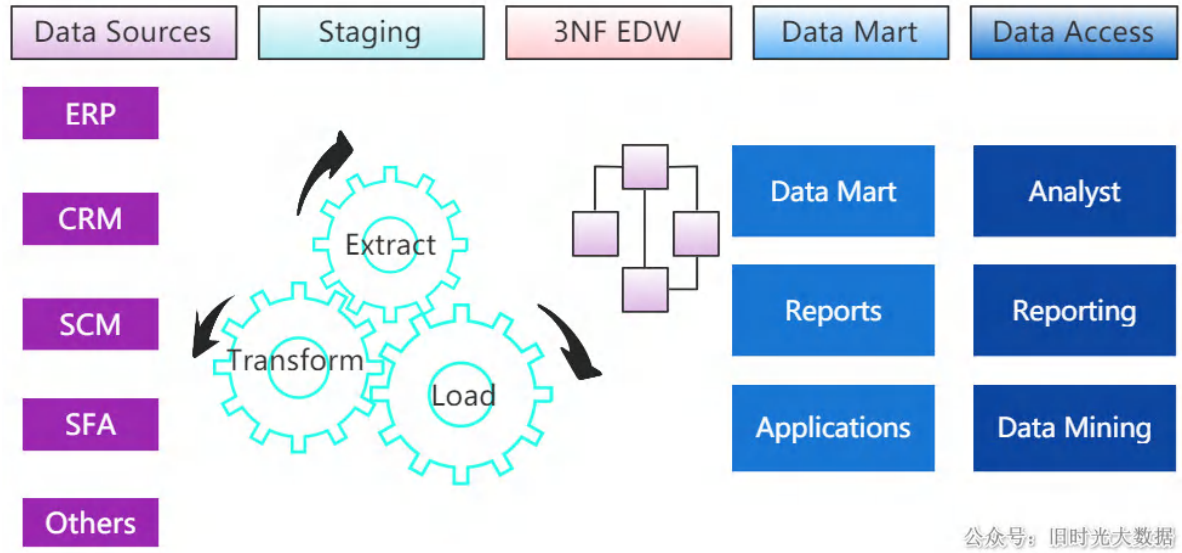
每种实现方法都有利弊。在许多情况下，最好的方法可能是某两种的组合。该方法的关键之一就是确定业务范围的架构需要用于支持集成的计划和设计的程度，因为数据仓库是用自底向上的方法进行构建。在使用自底向上或阶段性数据仓库项目模型来构建业务范围架构中的一系列数据集市时，您可以一个接一个地集成不同业务主题领域中的数据集市，从而形成设计良好的业务数据仓库。这样的方法可以极好地适用于业务。在这种方法中，可以把数据集市理解为整个数据仓库系统的逻辑子集，换句话说数据仓库就是一致化了的数据集市的集合。

2）数仓架构争论

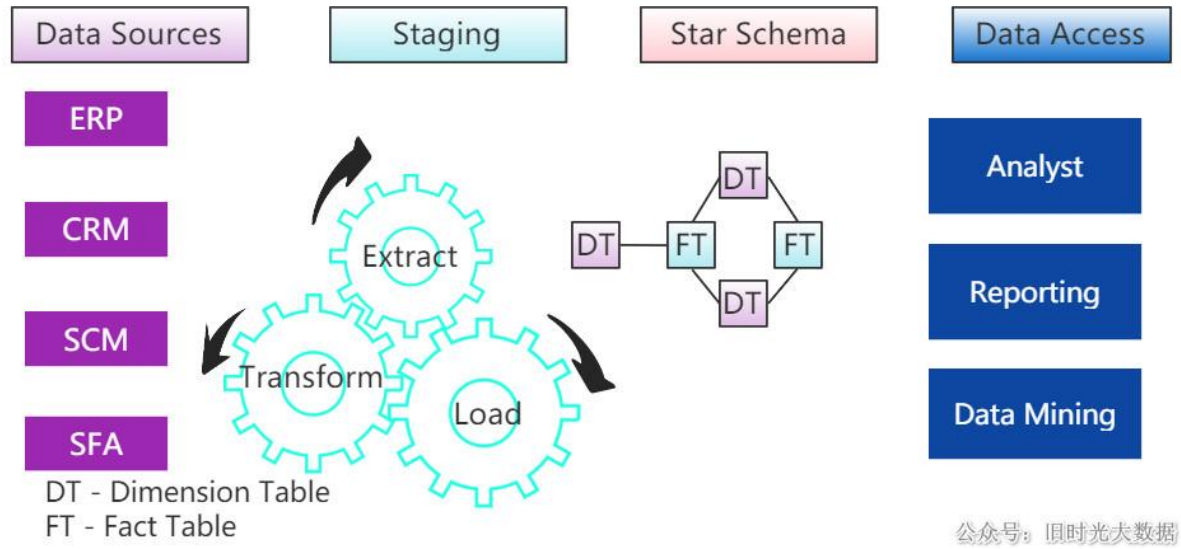
关于Inmon 和 Kimball的大辩论：Ralph Kimball 和 Bill Inmon 一直是商业智能领域中的革新者，开发并测试了新的技术和体系结构。在BI/DW领域中，围绕“哪一种数据仓库架构（Data Warehouse Architecture）最佳？”的争论一直没有休止，这个问题同时也是企业在建立DW时需要决策的关键问题：Bill Inmon的集线器架构/企业信息工厂架构（Hub and Spoke / CIF – Corporate Information Factory）与Ralph Kimball的数据集市/数据仓库总线架构（Data Mart Bus Architecture/Data Warehouse Bus Architecture）则是DW架构的争论焦点。

公众号：旧时光大数据

Bill Inmon 将数据仓库定义为“一个面向主题的、集成的、非易变的、随时间变化的用于支持管理的决策过程的数据集合”；他通过“面向主题”表示应该围绕主题来组织数据仓库中的数据，例如客户、销售、产品等等。每个主题区域仅仅包含该主题相关的信息。数据仓库应该一次增加一个主题，并且当需要容易地访问多个主题时，应该创建以数据仓库为来源的数据集市。换言之，某个特定数据集市中的所有数据都应该来自于面向主题的数据存储。 Inmon 的方法包含了更多上述工作而减少了对于信息的初始访问。但他认为这个集中式的体系结构持续下去将提供更强的一致性和灵活性，并且从长远来看将真正节省资源和工作。下图是他的设计方法图解：



Ralph Kimball 说“数据仓库仅仅是构成它的数据集市的联合”，他认为“可以通过一系列维数相同的数据集市递增地构建数据仓库”。每个数据集市将联合多个数据源来满足特定的业务需求。通过使用“一致的”维，能够共同看到不同数据集市中的信息。Kimball 的数据仓库结构也就是著 名的数据仓库总线(BUS)。设计方法如下图：



3）数仓架构选型

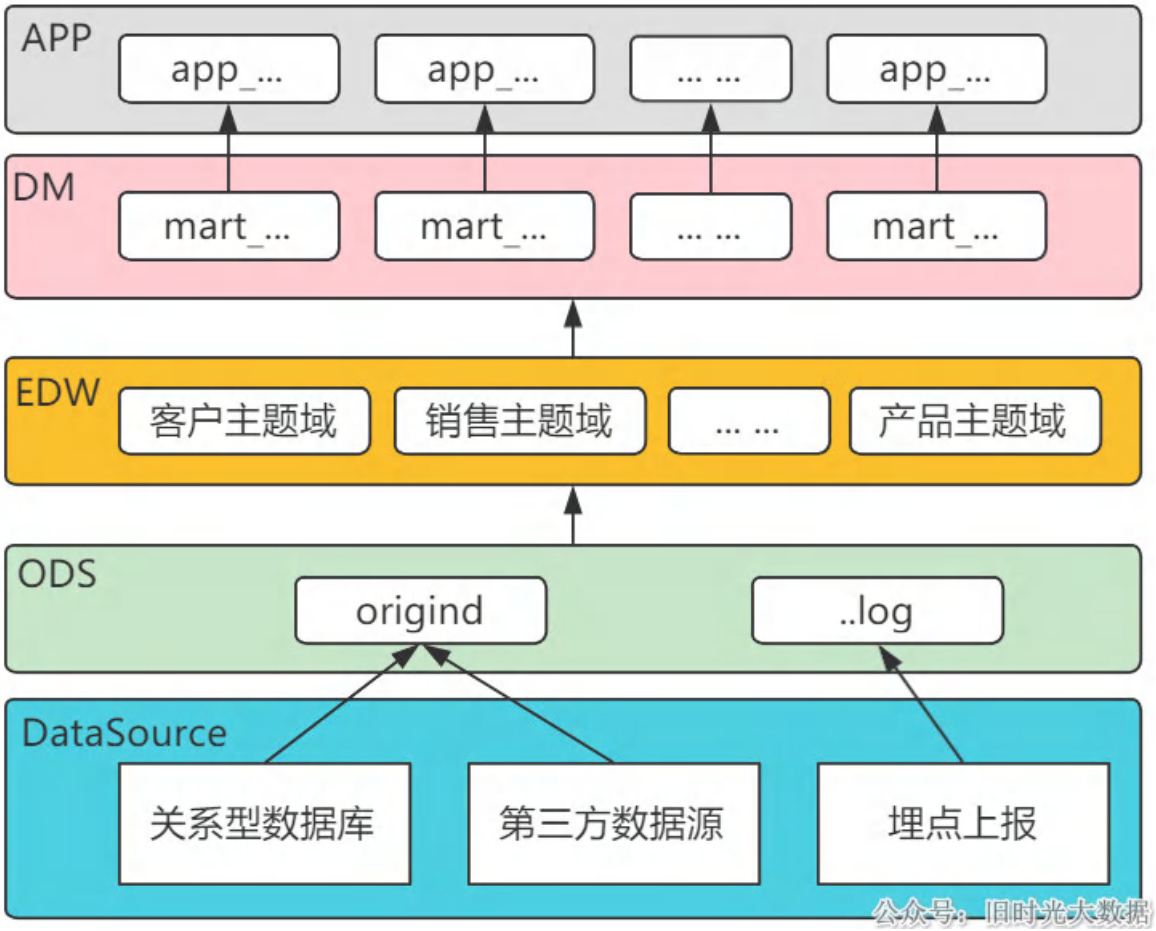
数据仓库架构的选取，与其所处的企业环境和业务的发展有着密切的关系：Inmon提倡的数据仓库建设方法，需要数据仓库建设人员自顶向下进行建设，数据仓库开发人员需要在数据仓库建设之前对企业各业务线进行深入的调研，有着非常全面的了解，然后根据企业各业务特点进行主题域划分。这种建设方式建设周期比较长，规划设计比较复杂，但是一旦建成，这个集中式的体系结构将提供更强的一致性和灵活性，并且从长远来看将真正节省资源和工作；Kimball提倡的数据仓库仅仅是构成它的数据集市的联合，各部门或业务可以根据自身的发展，建设符合自身主题的数据集市，并持续丰富完善这些数据集市。在应对企业级数据需求时，将这些数据集市的维度信息进行统一整理规范，然后通过一致的维度信息，将这些数据集市连接起来，使数据集市形成一个覆盖企业所有部门或业务的数据仓库，对外提供服务。

公众号：旧时光大数据

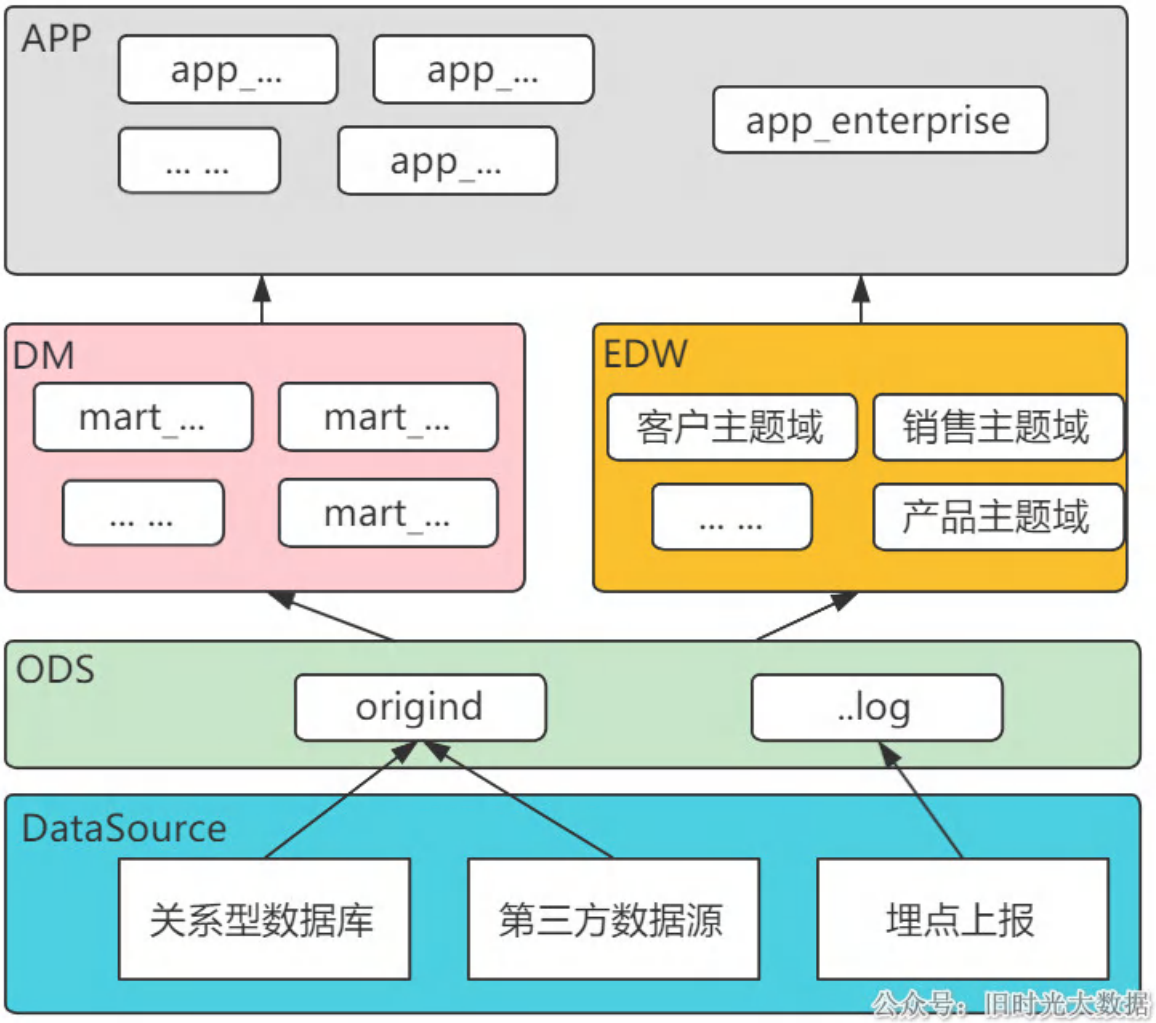
根据企业发展阶段和业务发展的速度建议：传统的、业务成熟的企业可以考虑采用Inmon方法建设数据仓库；业务复杂而且差异较大、发展速度又非常快的企业可以考虑Kimball方法建设数据仓库。

4）企业发展中的数据仓库建设变迁

企业或新部门，在初期发展过程中业务量少、组织形式相对简单。使得数仓建设人员可以站在全局的高度，俯视整个公司的业务流程，对其进行梳理归类，并抽取数据模型。以自上而下的方式建设数据仓库。所以在初期数据仓库建设的过程中基本采用了Inmon提倡的数据仓库建设方法，采用了DataSource-->ODS→EDW→DM-->APP的结构。即由ODS层完成各部门数据源的集成，在ODS的基础上建设了覆盖公司所有业务的包含众多主题的统一的数据仓库，然后由这个统一的数据仓库作为唯一的数据源，为各部门的数据集市提供数据支持。如下图：



但是一旦企业或部门发展速度非常快，业务量急剧增大，而且业务的组织形式趋于复杂，不同的业务之间可能存在巨大的差距。数据仓库的建设如果再继续沿用自伤而下的方式就会带来很多困难，例如在Inmon模式下EDW规划复杂、建设周期长，不能非常快速的响应各部门的需求，所以该方案逐步不能适应公司的发展。为了适应企业的发展，经过数仓开发人员的不断探索尝试，基本上倾向于采用混合模式建设数据仓库，即采用Inmon+Kimball的变种模式，结构如下：



从图上可以看出，与原有的架构最大的区别是：各部门数据集市的数据源并不是唯一的从EDW中获取，而是从各部门数据源所集成到的ODS层获取。但是有各部门数据集市也会涉及到跨部门的数据统计，所以这种公司级的数据应用还是从企业级数据仓库中获取。也就是各部门数据集市来支持各部门业务需求；企业级数据需求，从各部门数据集市或ODS层抽取公共模型进行建设（例如：公司级订单、用户等），并且在这里将各部门集市所依赖的公共维度进行统一，来支持公司级或跨部门的业务需求。

数据仓库建设中的数据建模这里就不多概括了。

数据仓库分层（层级划分），每层做什么？分层的好处？

可回答：1）数仓为什么分层；2）数仓分层的作用

问过的一些公司：字节 x 2，字节(2021.07)-(2021.08)-(2021.10)，阿里 x 2，爱奇艺，百度 x 2，百度(2021.08)，网易 x 3，网易(2021.09)x2，美团 x 4，美团(2021.09)x2，京东，京东(2021.09)，贝壳，keep，马蜂窝 x 2，转转，滴滴，小米，米哈游，有赞 x 2，猿辅导，58 x 2，作业帮社招，字节社招，腾讯社招 x 2，端点数据(2021.07)，百度(2021.09)，蔚来(2021.09)，恒生(2021.09)，快手(2021.09)，唯品会(2021.10)

参考答案：

首先，我要知道数据仓库分层架构的目标是什么？是为了实现维度建模，进而支撑决策分析目标。

数据分层从关系型在线交易系统到面向主题的数据仓库系统，从范式建模到维度建模的必经之路。

数据分层是一套让我们的数据体系更有序的行之有效的数据组织和管理方法。数据分层不是银弹，也没有绝对标准，当然也不能包治百病，不能解决所有的数据问题，但是，数据分层却可以给我们带来如下的好处：

公众号：旧时光大数据

隔离原始数据：不论是数据的异常还是数据敏感度，使真实数据与统计数据解耦开。

数据结构化更清晰：每一个数据分层都有它的作用域和职责，在使用表的时候能更方便地定位和理解。

数据血缘追踪：提供给外界使用的是一张业务表，但是这张业务表可能来源很多张表。如果有一张来源表出问题了，我们可以快速准确的定位到问题，并清楚每张表的作用范围。

增强数据复用能力：减少重复开发，通过数据分层规范化，开发一些通用的中间层数据，能够减少重复计算，提高单张业务表的使用率，提升系统的执行效率。

简化复杂的问题：把一个复杂的业务分成多个步骤实现，每一层只处理单一的步骤，比较简单和容易理解。而且便于维护数据的准确性，当数据出现问题之后，可以不用修复所有的数据，只需要从有问题的步骤开始修复。

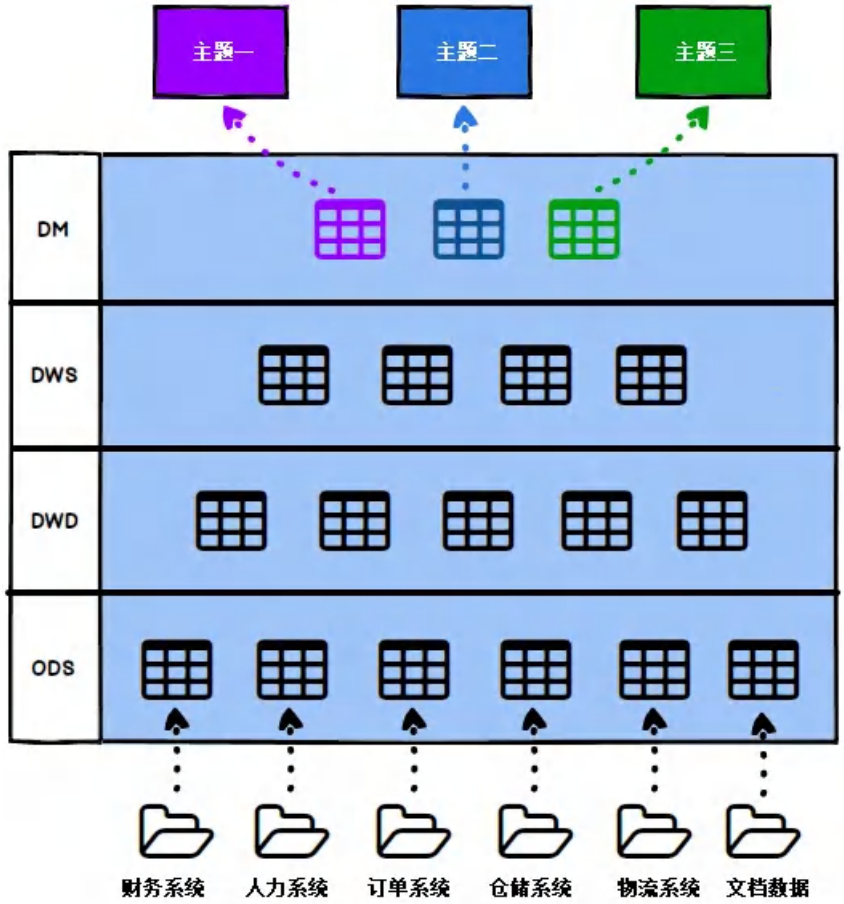
减少业务的影响：业务可能会经常变化，这样做就不必改一次业务就需要重新接入数据。

减少重复开发：规范数据分层，开发一些通用的中间层数据，能够减少极大的重复计算。

统一数据口径：通过数据分层，提供统一的数据出口，统一对外输出的数据口径。

分层的核心思想就是解耦，再解耦，把复杂的问题简单化。

数据仓库基础分层主要是分为四层，如下图所示



如上图所示，一个公司可能有多个业务系统，而数据仓库就是将所有的业务系统按照某种组织架构整合起来，形成一个仓储平台，也就是数仓。

1、四层分层

第一层：

ODS——**原始数据层**：存放原始数据

公众号：旧时光大数据

ODS层即操作数据存储，是最接近数据源中数据的一层，数据源中的数据，经过抽取、洗净、传输，也就说传说中的ETL之后，装入本层；一般来说ODS层的数据和源系统的数据是同构的，主要目的是简化后续数据加工处理的工作。从数据粒度上来说ODS层的数据粒度是最细的。ODS层的表通常包括两类，一个用于存储当前需要加载的数据，一个用于存储处理完后的历史数据。历史数据一般保存3-6个月后需要清除，以节省空间。但不同的项目要区别对待，如果源系统的数据量不大，可以保留更长的时间，甚至全量保存；数据在装入本层前需要做以下工作：去噪、去重、提脏、业务提取、单位统一、砍字段、业务判别。

第二层：

DWD——数据明细层：对ODS层数据进行清洗、维度退化、脱敏等。覆盖所有系统的、完整的、干净的、具有一致性的数据层。

该层一般保持和ODS层一样的数据粒度，并且提供一定的数据质量保证，在ODS的基础上对数据进行加工处理，提供更干净的数据。同时，为了提高数据明细层的易用性，该层会采用一些维度退化手法，当一个维度没有数据仓库需要的任何数据时，就可以退化维度，将维度退化至事实表中，减少事实表和维表的关联。例如：订单id,这种量级很大的维度，没必要用一张维度表来进行存储，而我们一般在进行数据分析时订单id又非常重要，所以我们将订单id冗余在事实表中，这种维度就是退化维度。

第三层：

DWS——数据服务层： 对DWD层数据进行一个轻度的汇总。

DWS层为公共汇总层，会进行轻度汇总，粒度比明细数据稍粗，会针对度量值进行汇总，目的是避免重复计算。该层数据表会相对比较少，大多都是宽表(一张表会涵盖比较多的业务内容，表中的字段较多)。按照主题划分，如订单、用户等，生成字段比较多的宽表，用于提供后续的业务查询，OLAP分析，数据分发等。

第四层：

DM——数据集市层：为各种统计报表提供数据。

存放的是轻度聚合的数据，也可以称为数据应用层，基于DWD、DWS上的基础数据，整合汇总成分析某一个主题域的报表数据。主要是提供给数据产品和数据分析使用的数据，通常根据业务需求，划分成流量、订单、用户等，生成字段比较多的宽表，用于提供后续的业务查询，OLAP分析，数据分发等。从数据粒度来说，这层的数据是汇总级的数据，也包括部分明细数据。从数据的时间跨度来说，通常是DW层的一部分，主要的目的是为了满足用户分析的需求，而从分析的角度来说，用户通常只需要分析近几年的即可。从数据的广度来说，仍然覆盖了所有业务数据。

2、三层分层

上述四层数仓，如果是问的三层数仓，就相当于是把DWD、DWS合并成DW层，往细的方面分，DW还包括DWM层（数据中间层），三层分层如下：

第一层：

ODS——原始数据层：存放原始数据

第二层：

DW——数据仓库层：数据清洗，初步汇总

本层将从 ODS 层中获得的数据按照主题建立各种数据模型，每一个主题对应一个宏观的分析领域，数据仓库层排除对决策无用的数据，提供特定主题的简明视图。在DW层会保存BI系统中所有的历史数据，例如保存10年的数据。

第三层：

DM——数据集市层

3、五层分层

公众号：旧时光大数据

五层分层如下：

第一层：

ODS——原始数据层：存放原始数据

第二层：

DWD——数据明细层：对ODS层数据进行清洗、维度退化、脱敏等。

第三层：

DWS——数据汇总层： 对DWD层数据进行一个轻度的汇总。

第四层：

ADS——数据应用层：为各种统计报表提供数据

该层是基于DW层的数据，整合汇总成主题域的服务数据，用于提供后续的业务查询等。

第五层：

DIM——维表层：基于维度建模理念思想，建立整个企业的一致性维度。

维表层主要包含两部分数据：

高基数维度数据：一般是用户资料表、商品资料表类似的资料表。数据量可能是千万级或者上亿级别。

低基数维度数据：一般是配置表，比如枚举值对应的中文含义，或者日期维表。数据量可能是个位数或者几千几万。

数据分层是根据什么?

问过的一些公司：美团

参考答案：

数据分层没有标准，数据分层要结合当下的技术，当前的数据量，业务的复杂度通盘考量。操作型数据存储（ODS）、数据仓库（DW）、数据集市（DM）其实就是数据分层的原始的指导建议，其中操作型数据存储是满足多个业务系统获取数据并制作操作型报表的需求，保留半年以内的明细准实时数据；数据仓库是为了管理决策、战略分析和计划提供单一整合点，保留长期的明细和汇总数据；数据集市是用于特定部门或公共分析需求并定制的，采用维度建模方式保留中期的明细和汇总数据。

数仓分层的原则与思路

可回答：你们数仓有几层，怎么划分的（结合划分原则，先考虑好情况）

问过的一些公司：远景智能(2021.08)，网易严选(2021.08)

参考答案：

数据仓库分层没有绝对的规范，适合的就是最好的，至于分几层，建议按照目前的业务和建设现状，进行合理解构和分层设计，一般刚开始做，建议3、4层。规划1-1.5年的架构，然后不断的建设、优化、再优化。不断逼近满足所有需求。

下面针对一些场景说下分层的想法：

场景一：时间紧任务重，急于看结果

这种场景，直接连各个业务数据库，抽取数据到大数据平台，根据需求组合join或者汇总count、sum就行，就不要分层了，有的公司项目就是将各个业务系统数据抽取到oracle，你看都没有大数据平台就做了。

场景二：公司业务简单，且相对比较固定，数据来源不多，结构也很清晰，需求也不多

直接使用通用的数仓架构就行，ODS起到解耦业务数据库+异构数据源的问题，DWD解决数据脏乱差的问题，DWS服用的指标计算，ADS直接面向前台业务需求。

场景三：公司业务复杂，业务变化较快

可以考虑多一层DWT层做汇总，多一层解耦，业务变化的时候，我们只改DWS层就好了，最多穿透到DWT层。业务变化的时候调整一下，工作量也不会太大，最重要的是能保证底层结构的稳定和数据分析的可持续性。

场景四：公司业务较为复杂，集团性公司，下辖多个部门业务部门事业线，业务部门间业务内容交叉不大

可以在数仓通用分层架构上，增加一层DM层，也就是数据集市层，各个数据集市层，单独供数，甚至有单独的计算资源，这样可以避免因为计算任务代码混在一起、数据权限拆分等问题带来的数据变更成本。

一个好的数仓模型分层，应该具备的要素

一个好的数仓模型分层，应该具备的要素是**数据模型可复用，完善且规范的**。

完善度：

- DWD：跨层引用率
- DWS/ADS/DM：汇总数据查询比例

复用度：

- DWD/DWS：模型引用系数

规范度：

- 有多少表没有主题域、业务过程归属
- 模型命名不规范
- 字段命名不规范

好的数仓设计标准：数据比较丰富完善、数据复用性强、规范性强。

从完善度上来讲，主要衡量DWD层和汇总层两块的完善度，DWD层完善度，主要是希望DWD等尽可能被汇总层引用，ODS层被除了DWD层外的尽可能少的引用，最好是没有。

从复用度上来讲，我们希望80%需求由20%的表来支持。直接点讲，就是大部分（80%以上）的需求，都用DWS的表来支持。

从规范度上来讲，主要从表名、字段名来看，一个规范的表名应该包括层级、主题域、分区规则，抽取类型等信息。字段规范应该是和词根一致，同字段同名等。

数据仓库分层没有绝对的规范，适合的就是最好的，数据仓库分层的核心逻辑是解耦，在有限时间、资源等条件下满足业务需求，同时又要兼顾业务的快速变化。所以我们作为数据架构师，需要兼顾业务的复杂变化，以及开发的复杂度和可维护性，在两者之间做一个平衡和取舍，选择合适的分层架构。

另外分层架构是需要不断的优化调整的，不能超前太多，也不能脱离业务。按照Inmon和Kimball吵了十几年的经验上看，建议架构设计时，按超越当前实际情况1~1.5年的设计是比较合适的。

数仓建模常用模型吗？区别、优缺点？

公众号：旧时光大数据

可回答：1）维度建模模型（有几种）；2）说下维度建模和关系建模；3）为什么会有星型和雪花型，各自的适合什么样的数据场景；4）介绍下数仓建模

问过的一些公司：蘑菇街，ebay，有赞，转转，携程(2021.09)，好未来(2021.08)，网易严选(2021.09)，快手(2021.09)，蔚来(2021.09)，字节(2021.10)x2，唯品会(2021.10)，陌陌(2021.10)，美团到家(2021.10)，美团(2021.08)

参考答案：

数据仓库建模的目标是通过建模的方法更好的组织、存储数据，以便在性能、成本、效率和数据质量之间找到最佳平衡点。

现在数据处理大致可以分为两大类：

操作型处理，叫联机事务处理 OLTP（On-Line Transaction Processing），也可以称面向交易的处理系统，它是针对具体业务在数据库联机的日常操作，通常对少数记录进行查询、修改。用户较为关心操作的响应时间、数据的安全性、完整性和并发支持的用户数等问题。传统的数据库系统作为数据管理的主要手段，主要用于操作型处理。

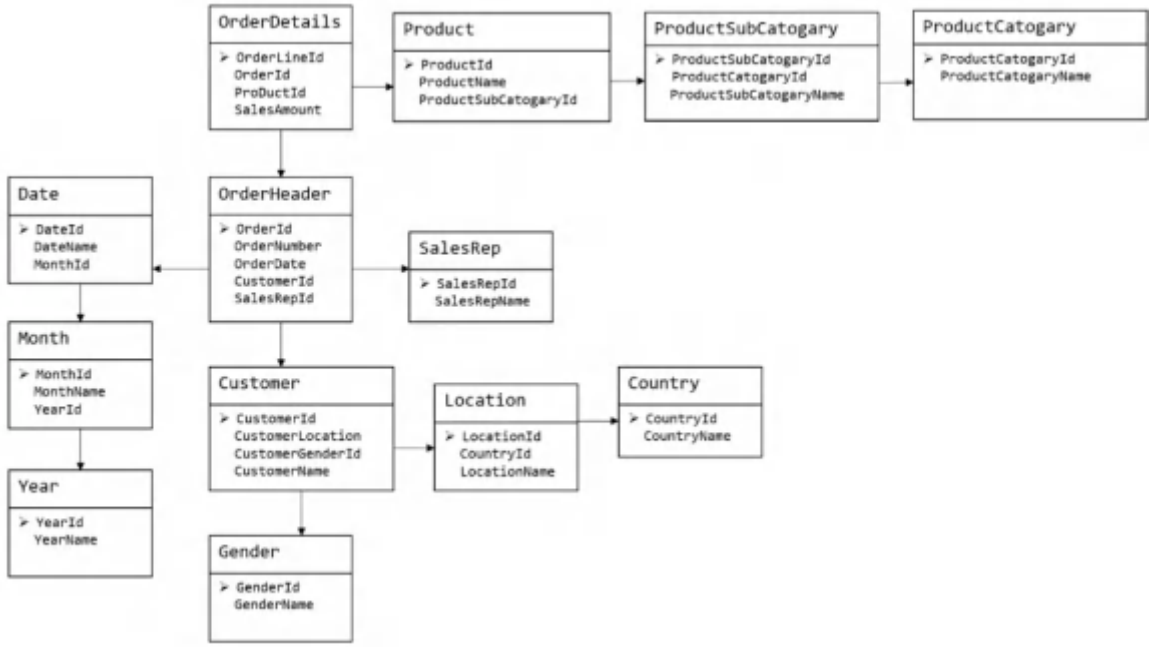
分析型处理，叫联机分析处理 OLAP（On-Line Analytical Processing），一般针对某些主题的历史数据进行分析，支持管理决策。

OLTP与OLAP的主要异同如下表：

操作型处理	分析型处理
细节的	综合的或提炼的
实体-关系（ER）模型	星型或雪花模型
存取瞬间数据	存储历史数据，不包含最近的数据
可更新的	只读，只追加
一次操作一个单元	一次操作一个集合
性能要求高，响应时间短	性能要求宽松
面向事务	面向分析
一次操作数据量小	一次操作数据量大
支持日常操作	支持决策需求
数据量小	数据量大
客户订单、库存水平和银行账户等	客户收益分析、市场细分等

1、关系（ER实体）建模

公众号：旧时光大数据



关系模型如图所示，严格遵循第三范式，从图中可以看出，**较为松散、零碎，物理表数量多，而数据冗余程度低**。由于数据分布于众多的表中，这些**数据可以更为灵活地被应用，功能性较强**。关系模型主要应用与OLTP系统中，为了保证数据的一致性以及避免冗余，所以大部分业务系统的表都是遵循第三范式的。

这是数据仓库之父Bill Inmon提出的建模方法，即实体关系（Entity Relationship，ER）模型。这是从全企业的高度设计一个3NF模型，用实体关系模型来描述企业业务，在范式理论上符合3NF。

关系模型主要应用于OLTP系统中，为了保证数据的一致性以及避免冗余，所以大部分业务系统的表都是遵循第三范式的。

特点：设计思路自上而下，适合上游基础数据存储，同一份数据只存储一份，没有数据冗余，方便解耦，易维护，缺点是开发周期一般比较长，维护成本高。

范式理论：

范式可以理解为设计一张数据表的表结构，符合的标准级别，也就是规范和要求。

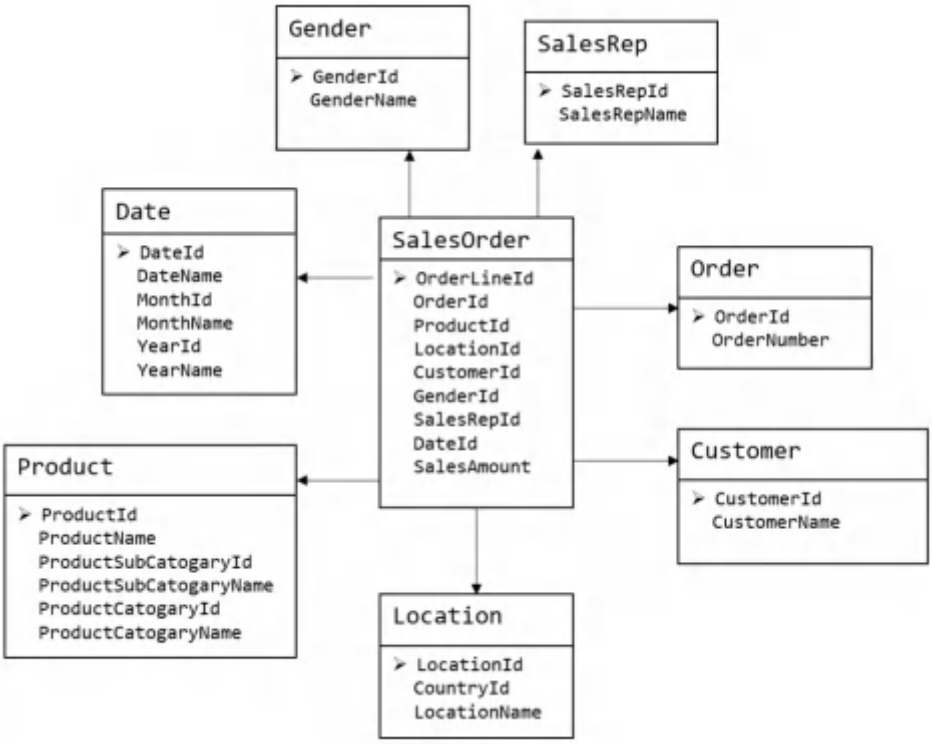
优点：关系型数据库设计时，遵照一定的规范要求，**目的在于降低数据的冗余性**。

缺点：范式的缺点是获取数据时，需要通过**Join拼接**出最后的数据。

分类：目前业界范式有**第一范式(1NF)、第二范式(2NF)、第三范式(3NF)**、巴斯-科德范式(BCNF)、第四范式(4NF)、第五范式(5NF)。

2、维度建模

公众号：旧时光大数据



维度模型如图所示，主要应用于OLAP系统中，通常以某一个事实表为中心进行表的组织，主要面向业务，特征是**可能存在数据的冗余，但是能方便的得到数据**。

关系模型虽然冗余少，但是在大规模数据，跨表分析统计查询过程中，会造成多表关联，这会大大降低执行效率。所以一般都会采用维度模型建模，把相关各种表整理成两种：事实表和维度表两种。

在维度建模的基础上又可分为三种模型：星型模型、雪花模型、星座模型。

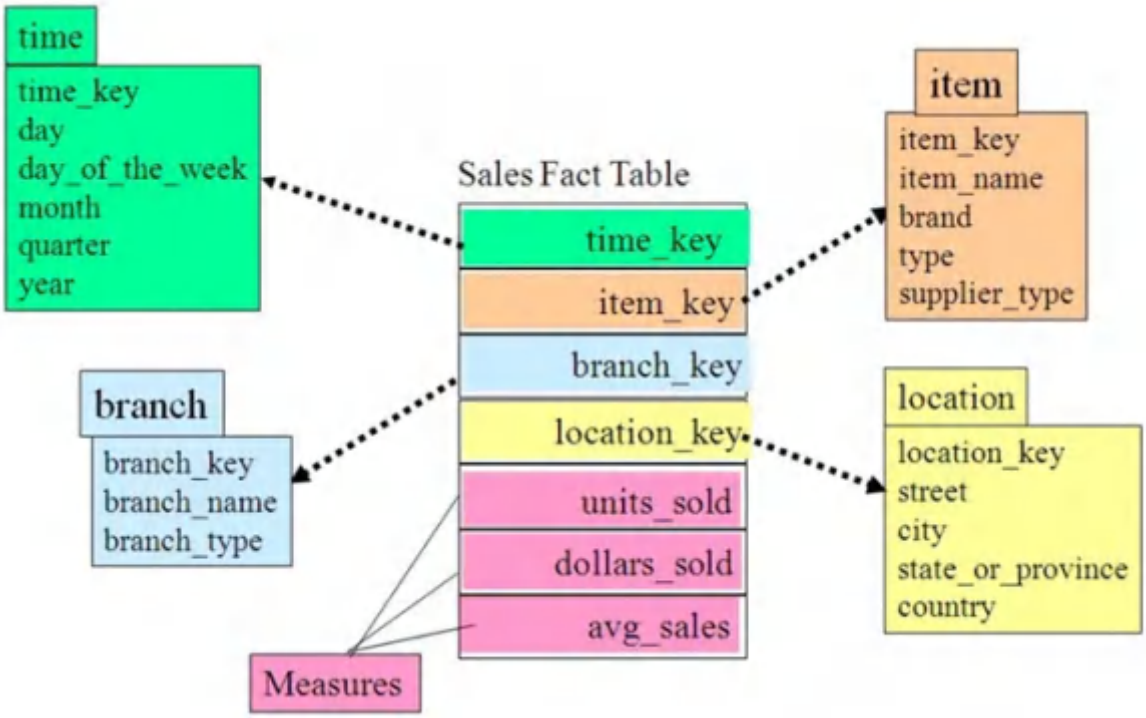
维度建模是从分析决策的需求出发构建模型，为分析需求服务，因此它重点关注用户如何更快速的完成需求分析，同事具有较好的大规模复杂查询的相应能力。其典型的代表是星型模型，以及在一些特殊场景下使用的雪花模型。

维度建模设计分为以下步骤：

- 选择需要进行分析决策的业务过程
- 定义粒度
- 识别维度
- 确认事实

1）星型模型

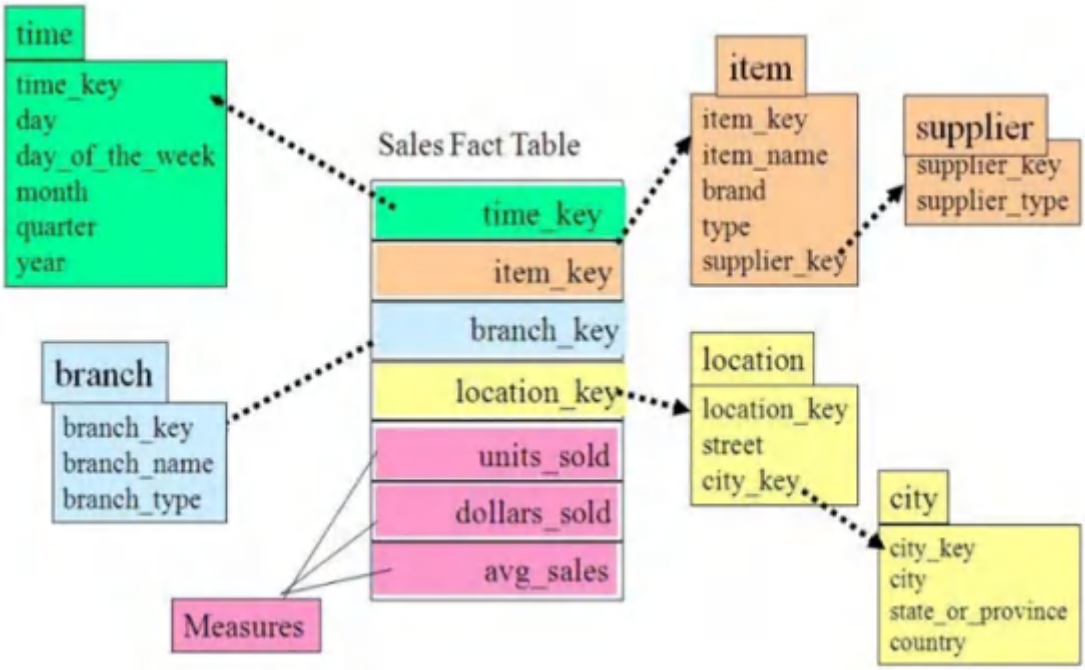
公众号：旧时光大数据



星型模式是维度模型中最简单的形式，也是数据仓库以及数据集市开发中使用最广泛的形式。星型模式由事实表和维度表组成，一个星型模式中可以有一个或多个事实表，每个事实表引用任意数量的维度表。

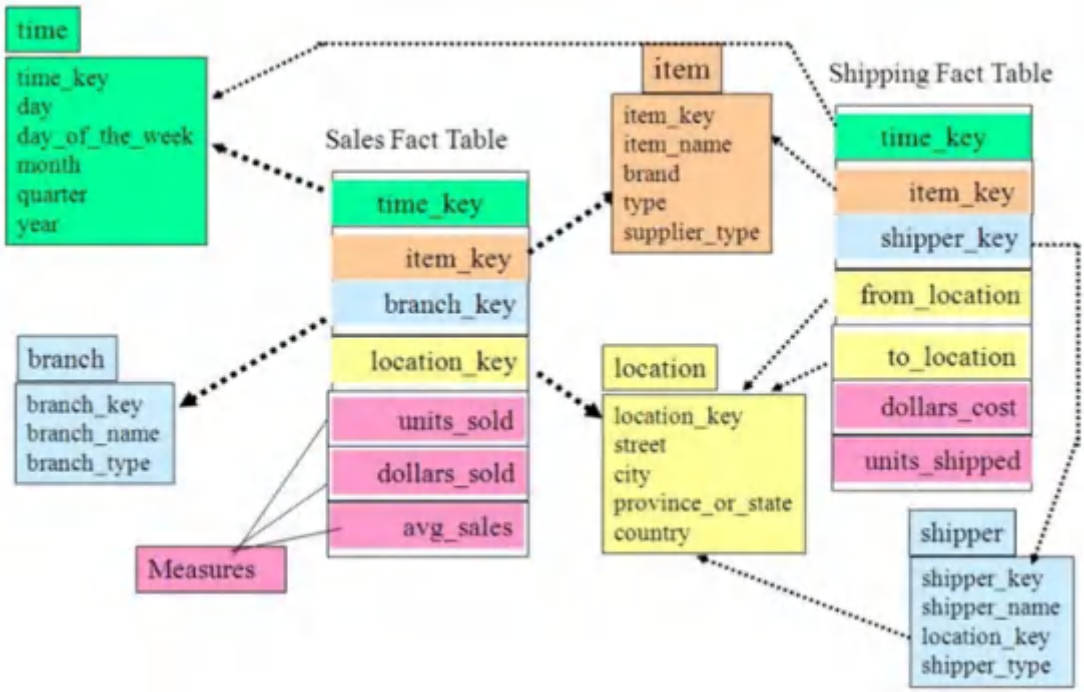
星型模型与雪花模型的区别主要在于维度的层级，标准的星型模型维度只有一层，而雪花模型可能会涉及多层。

2）雪花模型



雪花模式是一种多维模型中表的逻辑布局，与星型模式相同，雪花模式也是由事实表和维度表所组成。所谓的“雪花化”就是将星型模型中的维度表进行规范化处理。当所有的维度表完成规范化后，就形成了以事实表为中心的雪花型结构，即雪花模式。

3）星座模型



数据仓库由多个主题构成，包含多个事实表，而维表是公共的，可以共享（例如两张事实表共用一些维度表时，就叫做星型模型），这种模式可以看做星型模式的汇集，因而称作星系模式或者事实星座模式。

4）模型选择

在数据仓库建模时，会涉及到模式的选择，我们要根据不同模式的特点选择适合具体业务的模式。

星型还是雪花，取决于性能优先，还是灵活更优先。

在实际开发中，不会绝对选择一种，根据情况灵活组合，甚至并存（一层维度和多层维度都保存）。但是整体看来，更倾向于维度更少的星型模型。

3、建模方法总结

ER模型以及维度模型是当前主流的建模方法。

ER模型常用于OLTP数据库建模，应用到构建数仓时更偏重数据整合，**站在企业整体考虑，将各个系统的数据按相似性、一致性合并处理，为数据分析、决策服务，但并不便于直接用来支持分析。**

ER模型的特点如下：

- 需要全面梳理企业所有的业务和数据流；
- 实施周期长；
- 对建模人员要求高。

维度建模是面向分析场景而生，针对分析场景构建数仓模型；重点关注快速、灵活的解决分析需求，同时能够提供大规模数据的快速响应性能。**针对性强，主要应用于数据仓库构建和OLAP引擎低层数据模型。**

- 不需要完整的梳理企业业务流程和数据；
- 实施周期根据主题边界而定，容易快速实现 demo。

星型模型和雪花模型的区别？应用场景？优劣对比

问过的一些公司：字节跳动，快手，美团×4，美团(2021.09)，网易，携程 x 2，快手，多益，拼多多，米哈游，快手(2021.09)

公众号：旧时光大数据

问过的一些公司：

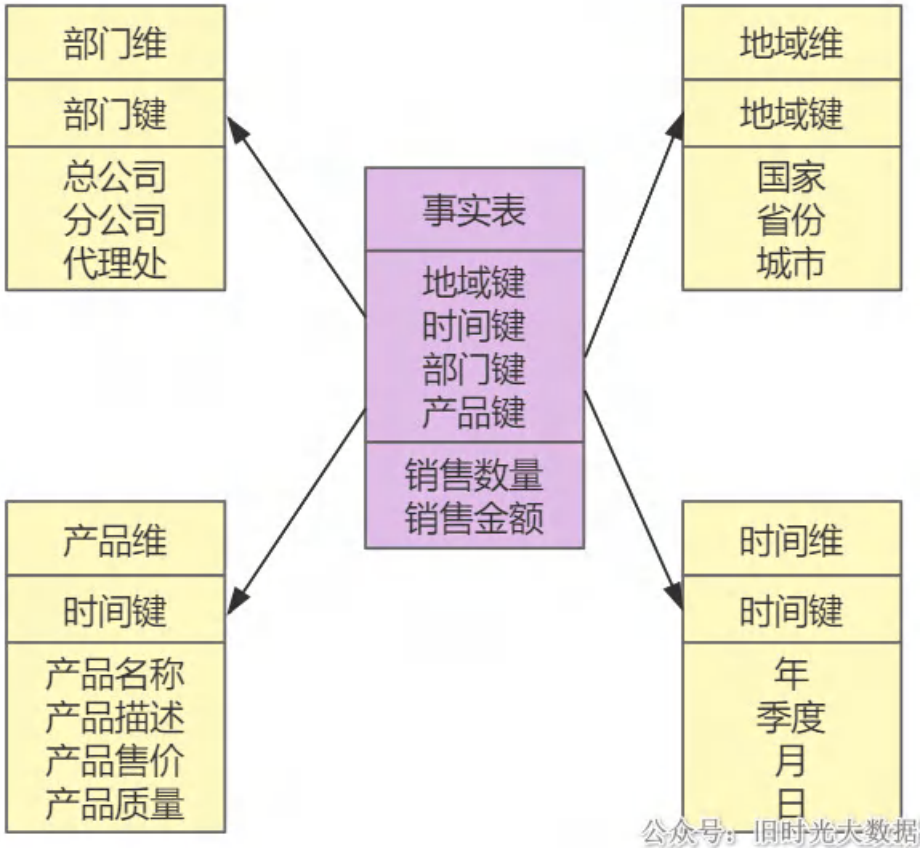
1、概述

1）星型模型

当所有维表都直接连接到事实表上时，整个图解就像星星一样，故将该模型称为星型模型。

星型架构是一种非正规化的结构，多维数据集的每一个维度都直接与事实表相连接，不存在渐变维度，所以数据有一定的冗余，因为有数据的冗余，很多的统计情况下，不需要和外表关联进行查询和数据分析，因此**效率相对较高。**如在地域维度表中，存在国家 A 省 B 的城市 C 以及国家 A 省 B 的城市 D 两条记录，那么国家 A 和省 B 的信息分别存储了两次，即存在冗余。

销售数据仓库中的星型模型



星型模型强调的是对维度进行预处理，将多个维度集合到一个事实表，形成一个宽表。这也是我们在使用hive时，经常会看到一些大宽表的原因，大宽表一般都是事实表，包含了维度关联的主键和一些度量信息，而维度表则是事实表里面维度的具体信息，使用时候一般通过join来组合数据，相对来说对OLAP的分析比较方便。

2）雪花模型

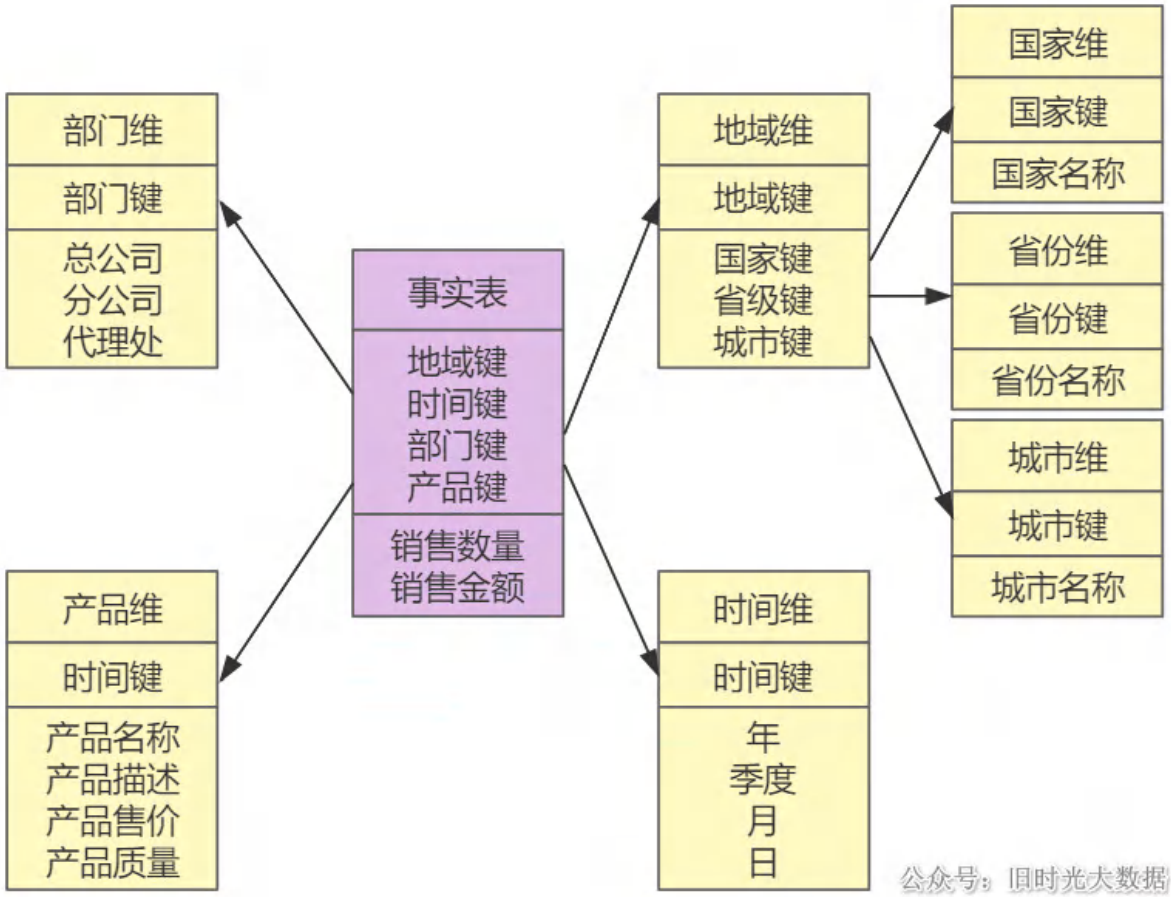
当有一个或多个维表没有直接连接到事实表上，而是通过其他维表连接到事实表上时，其图解就像多个雪花连接在一起，故称雪花模型。

雪花模型是对星型模型的扩展。它对星型模型的维表进一步层次化，原有的各维表可能被扩展为小的事实表，形成一些局部的 " 层次 " 区域，这些被分解的表都连接到主维度表而不是事实表。如下图，将地域维表又分解为国家，省份，城市等维表。

它的优点是：通过最大限度地减少数据存储量以及联合较小的维表来改善查询性能，较少数据冗余，但是在分析数据的时候，操作比较复杂，需要join的表比较多所以其性能并不一定比星型模型高，因此查询性能会相对较低。

销售数据仓库中的雪花型模型

公众号：旧时光大数据



星型模型因为数据的冗余所以很多统计查询不需要做外部的连接，因此一般情况下效率比雪花型模型要高。星型结构不用考虑很多正规化的因素，设计与实现都比较简单。雪花型模型由于去除了冗余，有些统计就需要通过表的联接才能产生，所以效率不一定有星型模型高。正规化也是一种比较复杂的过程，相应的数据库结构设计、数据的 ETL、以及后期的维护都要复杂一些。因此在冗余可以接受的前提下，实际运用中星型模型使用更多，也更有效率。

2、区别及优缺点

星型模型优点：星型模型因为数据的冗余，所以很多统计查询不需要做外部的连接，因此一般情况下效率比雪花模型要高。星型模型不用考虑很多正规化的因素，设计与实现都比较简单。

星型模型缺点：有一定数据冗余。

雪花模型的优点： 通过最大限度的减少数据量以及连接较小的维度表来实现改善查询的功能，雪花结构减少的数据的冗余。

雪花模型缺点：在雪花模型需要事实表和维度表之间的连接较多，因此查询性能会相对较低。

正规化也是一种比较复杂的过程，相应的数据库结构设计、数据的ETL、以及后期的维护都要复杂一些。因此在冗余可以接受的前提下，实际运用中星型模型使用更多，也更有效率。

公众号：旧时光大数据

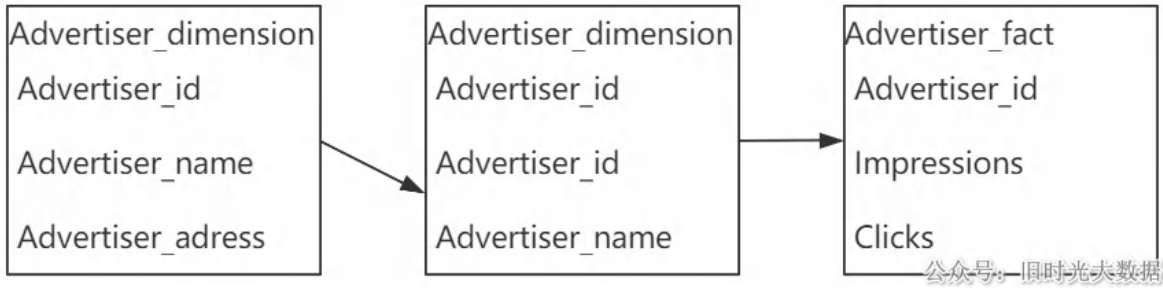
属性	星型模型	雪花模型
数据总量	多	少
可读性	容易	差
表个数	少	多
查询速度	快	慢
冗余度	高	低
对实时表的情况	增加宽度	字段比较少，冗余低
扩展性	差	好

3、使用选择

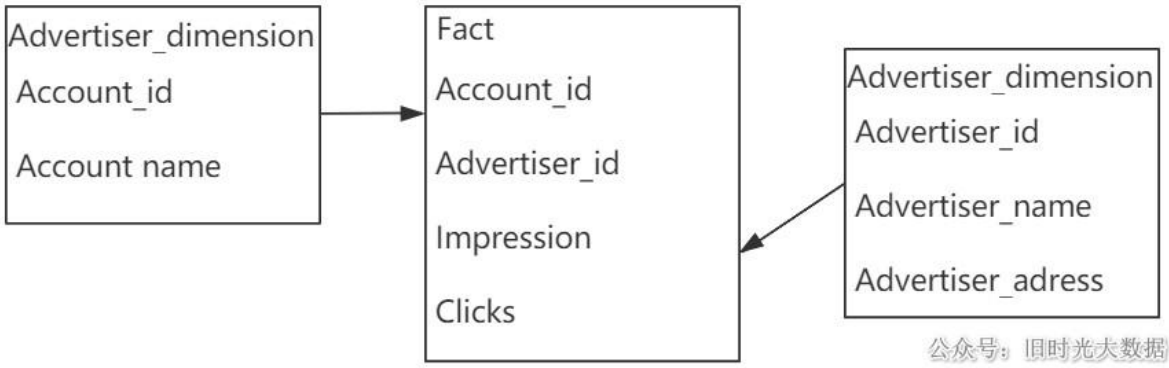
星形模型和雪花模型是数据仓库中常用到的两种方式，而它们之间的对比要从四个角度来进行讨论。

1）数据优化

雪花模型使用的是规范化数据，也就是说数据在数据库内部是组织好的，以便消除冗余，因此它能够有效地减少数据量。通过引用完整性，其业务层级和维度都将存储在数据模型之中。



相比较而言，星形模型实用的是反规范化数据。在星形模型中，维度直接指的是事实表，业务层级不会通过维度之间的参照完整性来部署。



2）业务模型

主键是一个单独的唯一键(数据属性)，为特殊数据所选择。在上面的例子中，Advertiser_ID就将是一个主键。外键(参考属性)仅仅是一个表中的字段，用来匹配其他维度表中的主键。在我们所引用的例子中，Advertiser_ID将是Account_dimension的一个外键。

在雪花模型中，数据模型的业务层级是由一个不同维度表主键-外键的关系来代表的。而在星形模型中，所有必要的维度表在事实表中都只拥有外键。

3）性能

第三个区别在于性能的不同。雪花模型在维度表、事实表之间的连接很多，因此性能方面会比较低。举个例子，如果你想要知道Advertiser 的详细信息，雪花模型就会请求许多信息，比如Advertiser Name、ID以及那些广告主和客户表的地址需要连接起来，然后再与事实表连接。

而星形模型的连接就少的多，在这个模型中，如果你需要上述信息，你只要将Advertiser的维度表和事实表连接即可。

4）ETL

雪花模型加载数据集市，因此ETL操作在设计上更加复杂，而且由于附属模型的限制，不能并行化。

星形模型加载维度表，不需要再维度之间添加附属模型，因此ETL就相对简单，而且可以实现高度的并行化。

5）小结

雪花模型使得维度分析更加容易，比如“针对特定的广告主，有哪些客户或者公司是在线的?”，星形模型用来做指标分析更适合，比如“给定的一个客户他们的收入是多少?”。

适用情况：星型模型更适用于做指标分析，而雪花模型更适用于做维度分析

数仓建模有哪些方式?

问过的一些公司：美团(2021.09)x2，唯品会(2021.10)

参考答案：

1、维度建模

维度建模按数据组织类型划分可分为星型模型、雪花模型、星座模型。

1）**星型模型**

星型模型主要是维表和事实表，以事实表为中心，所有维度直接关联在事实表上，呈星型分布。

2）**雪花模型**

雪花模型，在星型模型的基础上，维度表上又关联了其他维度表。这种模型维护成本高，性能方面也较差，所以一般不建议使用。尤其是基于hadoop体系构建数仓，减少join就是减少shuffle，性能差距会很大。

星型模型可以理解为，一个事实表关联多个维度表，雪花模型可以理解为一个事实表关联多个维度表，维度表再关联维度表。

3）**星座模型**

数据仓库由多个主题构成，包含多个事实表，而维表是公共的，可以共享（例如两张事实表共用一些维度表时，就叫做星型模型），这种模式可以看做星型模式的汇集，因而称作星系模式或者事实星座模式。

2、范式建模（ER建模）

从全企业的高度设计一个3NF模型，用实体加关系描述的数据模型描述企业业务架构，在范式理论上符合3NF。此建模方法，对建模人员的能力要求非常高。

特点：设计思路自上而下，适合上游基础数据存储，同一份数据只存储一份，没有数据冗余，方便解耦，易维护，缺点是开发周期一般比较长，维护成本高。

3、Data Vault模型

公众号：旧时光大数据

DataVault由Hub（关键核心业务实体）、Link（关系）、Satellite（实体属性） 三部分组成，是Dan Linstedt发起创建的一种模型方法论，它是在ER关系模型上的衍生，同时设计的出发点也是为了实现数据的整合，并非为数据决策分析直接使用。

4、Anchor模型

Anchor模型是对Data Vault模型做了进一步规范化处理，它是一个高度可扩展的模型，所有的扩展只是添加而不是修改，因此它将模型规范到6NF，基本变成了K-V结构模型。企业很少使用。

数仓建模的流程?

可回答：1）数仓如何建模；2）数仓建设的原理

问过的一些公司：字节，字节(2021.10)，好未来，快手(2021.09)，京东(2021.09)x2

参考答案：

数据仓库的发展大致经历了这样的三个过程：

简单报表阶段：这个阶段，系统的主要目标是解决一些日常的工作中业务人员需要的报表，以及生成一些简单的能够帮助领导进行决策所需要的汇总数据。大部分表现形式为数据库和前端报表工具。

数据集市阶段：这个阶段，主要是根据某个业务部门的需要，进行一定的数据的采集，整理，按照业务人员的需要，进行多维报表的展现，能够提供对特定业务指导的数据，并且能够提供特定的领导决策数据。

数据仓库阶段：这个阶段，主要是按照一定的数据模型，对整个企业的数据进行采集，整理，并且能够按照各个业务部门的需要，提供跨部门的，完全一致的业务报表数据，能够通过数据仓库生成对对业务具有指导性的数据，同时，为领导决策提供全面的数据支持。

通过数据仓库建设的发展阶段，可以看出，数据仓库的建设和数据集市的建设的重要区别就在于数据模型的支持。因此，数据模型的建设，对于我们数据仓库的建设，有着决定性的意义。

一般来说，**数据模型的建设主要能够帮助我们解决以下的一些问题**：

1）进行全面的业务梳理，改进业务流程

- 在业务模型建设的阶段，能够帮助我们的企业或者是管理机关对本单位的业务进行全面的梳理。
- 通过业务模型的建设，我们应该能够全面了解该单位的业务架构图和整个业务的运行情况，能够将业务按照特定的规律进行分门别类和程序化。
- 同时，帮助我们进一步的改进业务的流程，提高业务效率，指导我们的业务部门的生产。

2）建立全方位的数据视角，消灭信息孤岛和数据差异

- 通过数据仓库的模型建设，能够为企业提供一个整体的数据视角，不再是各个部门只是关注自己的数据。
- 而且通过模型的建设，勾勒出了部门之间内在的联系，帮助消灭各个部门之间的信息孤岛的问题。
- 更为重要的是，通过数据模型的建设，能够保证整个企业的数据的一致性，各个部门之间数据的差异将会得到有效解决。

3）解决业务的变动和数据仓库的灵活性

- 通过数据模型的建设，能够很好的分离出底层技术的实现和上层业务的展现。
- 当上层业务发生变化时，通过数据模型，底层的技术实现可以非常轻松的完成业务的变动，从而达到整个数据仓库系统的灵活性。

4）帮助数据仓库系统本身的建设

公众号：旧时光大数据

- 通过数据仓库的模型建设，开发人员和业务人员能够很容易的达成系统建设范围的界定，以及长期目标的规划，从而能够使整个项目组明确当前的任务，加快整个系统建设的速度。

1、建模流程

就是**业务模型->概念模型->逻辑模型->物理模型**的这样一个流程，下面我们详细解释一下各个模型阶段都要做什么。

1）业务建模：需求沟通

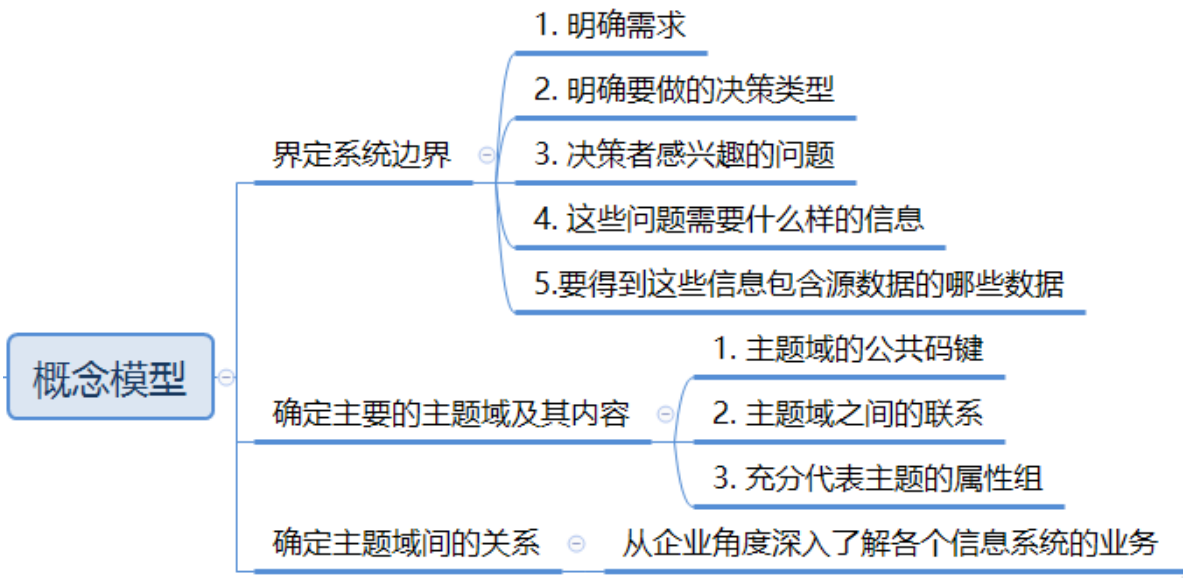
- 划分整个单位的业务，一般按照业务部门的划分，进行各个部分之间业务工作的界定，理清各业务部门之间的关系。
- 深入了解各个业务部门的内具体业务流程并将其程序化。
- 提出修改和改进业务部门工作流程的方法并程序化。
- 数据建模的范围界定，整个数据仓库项目的目标和阶段划分。

业务建模阶段其实是一次和业务人员梳理业务的过程，在这个过程中，不仅能帮助我们技术人员更好的理解业务，另一方面，也能够发现业务流程中的一些不合理的环节，加以改善和改进。

2）领域（概念）建模：画图想好怎么做

- 抽取关键业务概念，并将之抽象化。
- 将业务概念分组，按照业务主线聚合类似的分组概念。
- 细化分组概念，理清分组概念内的业务流程并抽象化。
- 理清分组概念之间的关联，形成完整的领域概念模型。

概念模型具体要求如下：



领域概念建模就是运用了实体建模法，从纷繁的业务表象背后通过实体建模法，抽象出实体，事件，说明等抽象的实体，从而找出业务表象后抽象实体间的相互的关联性，保证了我们数据仓库数据按照数据模型所能达到的一致性和关联性。

3）逻辑建模：表设计

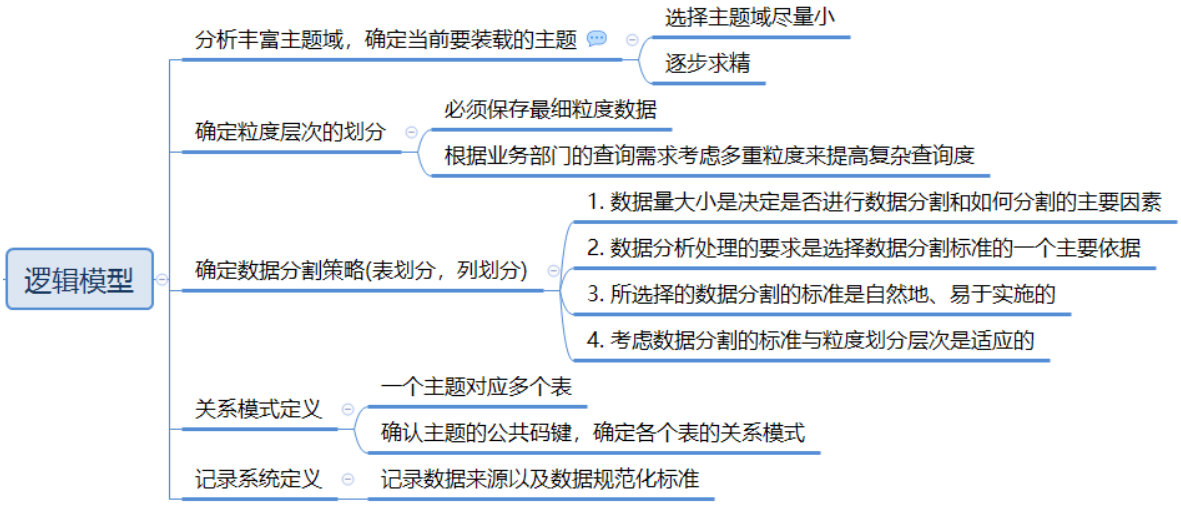
业务概念实体化，并考虑其具体的属性。

事件实体化，也就是所谓的事实，并考虑其属性内容。

说明实体化，也就是所谓的维度，并考虑其属性内容。

逻辑模型具体要求如下：

公众号：旧时光大数据

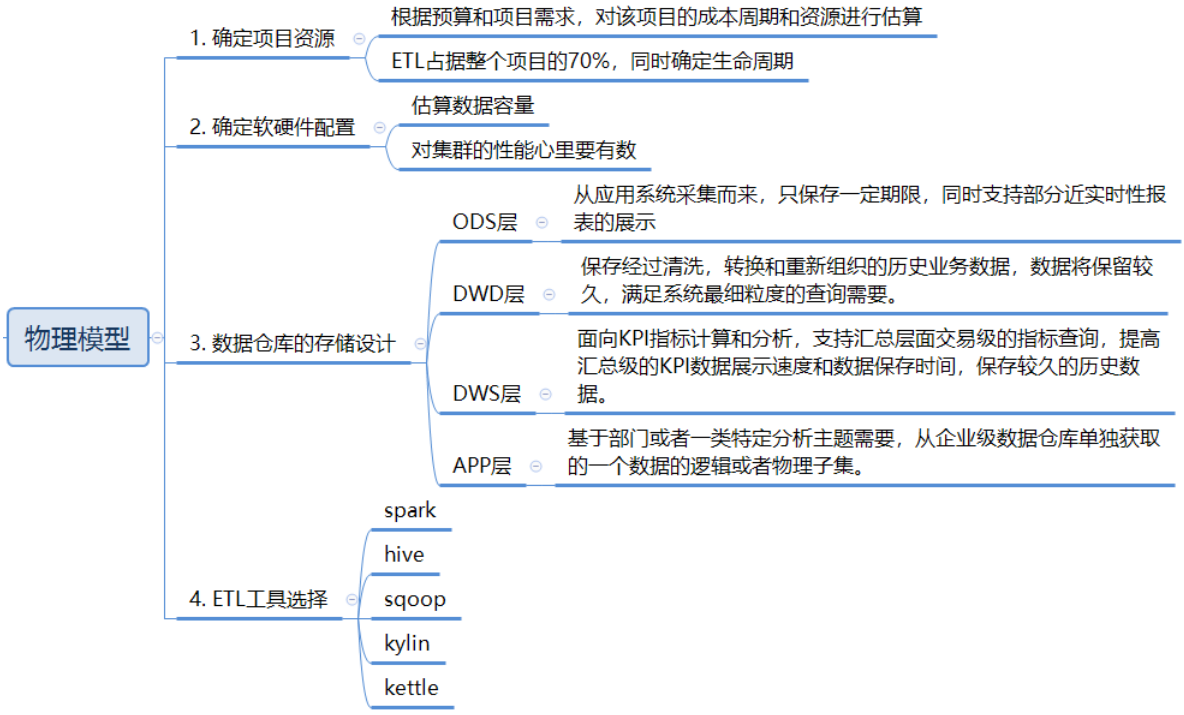


总体来说就是建表，前面已经画出了关系图，这里只要将表里头有哪些字段考虑出来就可以，如果是事实表就考虑事实字段和业务主键，如果是维度表就考虑维度属性，SCD策略等等。在这里需要确定数据粒度，如果多个指标都用到一个字段，则取粒度最小的指标。如果不确定指标的量度，则取毫秒级作为粒度。

4）物理建模：建表

- 针对特定物理化平台，做出相应的技术调整。
- 针对模型的性能考虑，对特定平台作出相应的调整。
- 针对管理的需要，结合特定的平台，做出相应的调整。
- 生成最后的执行脚本，并完善。

物理模型具体要求如下：



综合现实的大数据平台、采集工具、etl工具、数仓组件、性能要求、管理要求等多方面因素，设计出具体的项目代码，完成数仓的搭建。

总结来说，上面的模型设计流程大部分应用于DWD层，也就是事实维度层。通过建模，捋清逻辑，把业务落实到一张张表，并梳理表于表之间的关系。

2、建模过程

假设现在在构建一张订单表

公众号：旧时光大数据

从多个维度进行统计组合，形成多维度数据集，来从多个角度观察业务过程的好坏

1）选择业务过程

- 确认哪些业务处理流程是数据仓库应该覆盖的，是维度方法的基础。因此，建模的第一个步骤是描述需要建模的业务流程。例如，需要了解和分析一个零售店的销售情况，那么与该零售店销售相关的所有业务流程都是需要关注的。为了描述业务流程，可以简单地使用纯文本将相关内容记录下来，或者使用“业务流程建模标注”（BPMN）方法，也可以使用统一建模语言（UML）或其他类似的方法。
- 业务过程就是需要那种业务场景下产生的订单表(划分到那个业务线和数据域)
- 业务过程就是用户下单的订单记录表

2）选择数据域

（1）申明粒度

- 粒度就是确认一条记录代表的含义或者是细化到何种程度(一条记录代表一个订单还是多个订单，如拼团的时候团长的单)
- 在选择维度和事实前必须声明粒度，因为每个候选维度或事实必须与定义的粒度保持一致。在一个事实所对应的所有维度设计中强制实行粒度一致性是保证数据仓库应用性能和易用性的关键。
- 从给定的业务流程获取数据时，原始粒度是最低级别的粒度。建议从原始粒度数据开始设计，因为原始记录能够满足无法预期的用户查询。汇总后的数据粒度对优化查询性能很重要，但这样的粒度往往不能满足对细节数据的查询需求。
- 不同的事实可以有不同的粒度，但同一事实中不要混用多种不同的粒度。维度模型建立完成之后，还有可能因为获取了新的信息，而回到这步修改粒度级别。

（2）确认维度

- 维度的粒度必须和第二步所声明的粒度一致。
- 维度表是事实表的基础，也说明了事实表的数据是从哪里采集来的。
- 典型的维度都是名词，如日期、商店、库存等。维度表存储了某一维度的所有相关数据，例如，日期维度应该包括年、季度、月、周、日等数据。

（3）确认事实

- 这一步识别数字化的度量，构成事实表的记录。它是和系统的业务用户密切相关的，因为用户正是通过对事实表的访问获取数据仓库存储的数据。大部分事实表的度量都是数字类型的，可累加，可计算，如成本、数量、金额等。

3、模型设计的思路

业务需求驱动，数据驱动，构造数据仓库有两种方式：一是自上而下，一是自下而上。

1）自上而下

Bill Inmon先生推崇“自上而下”的方式，即一个企业建立唯一的数据中心，就像一个数据的仓库，其中数据是经过整合、经过清洗、去掉脏数据的、标准的，能够提供统一的视图。要建立这样的数据仓库，并不从它需要支持哪些应用入手，而是要从整个企业的环境入手，分析其中的概念，应该有什么样的数据，达成整体概念。

2）自下而上

Ralph Kimball先生推崇“自下而上”的方式，他认为建设数据仓库应该按照实际的应用需求，加载需要的数据，不需要的数据不要加载到数据仓库中。这种方式建设周期较短，客户能够很快看到结果。（针对客户的需求，需求要什么就做什么）

4、模型落地实现

- 按照命名规范创建表
- 开发生成维表和事实表的代码

公众号：旧时光大数据

- 进行代码逻辑测试，验证数据加工逻辑的正确性
- 代码发布，加入调度并配置相应的质量监控和报警机制

维度建模的步骤，如何确定这些维度的

可回答：维度建模怎么做的

问过的一些公司：招银网络，字节(2021.07)-(2021.08)，端点数据(2021.07)x2，美团(2021.09)

参考答案：

维度建模主要有4个步骤：选取业务过程、定义粒度、确定维度和确定事实。这4个步骤贯穿了维度建模的整个过程和环节。

1、选取业务

过程业务过程即企业和组织的业务活动，它们一般都有相应的源头业务系统支持。对于一个超市来说，其最基本的业务活动就是用户收银台付款；对于一个保险公司来说，最基本的业务活动是理赔和保单等。当然在实际操作中，业务活动有可能并不是那么简单直接，此时听取用户的意见通常是这一环节最为高效的方式。

需要注意的是，这里谈到的业务过程并不是指业务部门或者职能。模型设计中，应将注意力集中放在业务过程而不是业务部门，如果建立的维度模型是同部门捆绑在一起的，就无法避免出现数据不一致的情况（如业务编码、含义等）。因此，确保数据一致性的最佳办法是从企业和公司全局与整体角度，对于某一个业务过程建立单一的、一致的维度模型。

2、定义粒度

定义粒度意味着对事实表行实际代表的内容和含义给出明确的说明。粒度传递了事实表度量值相联系的细节所达到的程度的信息。其实质就是如何描述事实表的单个行。

典型的粒度定义包括：

- 超市顾客小票的每一个子项；
- 医院收费单的明细子项；
- 个人银行账户的每一次存款或者取款行为；
- 个人银行账户每个月的余额快照。

对于维度设计来说，在事实表粒度上达成一致非常重要，如果没有明确的粒度定义，则不能进入后面的环节。如果在后面的环节中发现粒度的定义不够或者是错误的，那么也必须返回这一环节重新定义粒度。

在定义粒度过程中，应该最大限度地选择业务过程中最为原子性的粒度，这样可以带来后续的最大灵活度，也可以满足业务用户的任何粒度的分析需求。

3、确认维度

定义了粒度之后，相关业务过程的细节也就确定了，对应的维度就很容易确定。正如前文所述，维度是对度量的上下文和环境的描述。通过维度，业务过程度量与事实就会变得丰富和丰满起来。对于订单来说，常见的维度会包含商品、日期、买家、卖家、门店等而每一个维度还可以包含大量的描述信息，比如商品维度表会包含商品名称、标签价、商品品牌、商品类目、商品上线时间等。

4、确认事实

确定事实通过业务过程分析可能要分析什么来确定。定义粒度之后，事实和度量一般也很容易确定，比如超市的订单活动，相关的度量显然是销售数量和销售金额。

公众号：旧时光大数据

在实际维度事实设计中，可能还会碰到度量拆分的问题，比如超市开展单个小票满10减10元的活动，如果小票金额超过10元，这10元的优惠额如何分配到每一个小票子项实际设计中，可以和业务方具体讨论并制订具体的拆分分配算法。

维度建模和范式建模区别

可回答：1）什么是维度建模？2）维度建模与范式建模相比有什么缺点？

问过的一些公司：阿里，美团x6，ebay，网易，好未来(2021.08)，虎牙(2021.08)

参考答案：

1、范式建模

Inmon提出的集线器的自上而下（EDW-DM）的数据仓库架构。操作型或事务型系统的数据源，**通过ETL抽取转换和加载到数据仓库的ODS层，然后通过ODS的数据建设原子数据的数据仓库EDW**，EDW不是多维格式的，不方便上层应用做数据分析，所以需要通过汇总建设成多维格式的数据集市层。

优势：易于维护，高度集成；

劣势：结构死板，部署周期较长。

范式建模应用在EDW层

一个符合第三范式的关系必须具有以下三个条件：

- 每个属性的值唯一，不具有多义性；
- 每个非主属性必须完全依赖于整个主键，而非主键的一部分；
- 每个非主属性不能依赖于其他关系中的属性，因为这样的话，这种属性应该归到其他关系中去。

但是由于EDW的数据是原子粒度的，数据量比较大，完全规范的3范式在数据的交互的时候效率比较低下，所以通常会根据实际情况在事实表上做一些冗余，减少过多的数据交互。

2、维度建模

Kimball提出的总线式的自下而上（DM-DW）的数据仓库架构。同样的，操作型或事务型系统的数据源，**通过ETL抽取转换和加载到数据仓库的ODS层，然后通过ODS的数据，利用维度建模方法建设一致维度的数据集市**。通过一致性维度可以将数据集市联系在一起，由所有的数据集市组成数据仓库。

优势：构建迅速，最快的看到投资回报率，敏捷灵活；

劣势：作为企业资源不太好维护，结构复杂，数据集市集成困难。

在维度建模的基础上又分为三种模型：星型模型、雪花模型、星座模型。

具体选择哪种建模模型，目前实际企业实际开发中，不会选择绝对一种，根据情况灵活组合，甚至并存（一层维度和多层维度都保存）。但是整体来看，更倾向于维度更少的星型模型。尤其是Hadoop体系，减少Join就是减少Shuffle，性能差距很大。

维度表和事实表的区别?

问过的一些公司：

参考答案：

简单的说**维度表**就是我们观察该事物的角度（维度）；**事实表**就是我们要关注的内容。

事实表：表格里存储了能体现实际数据或详细数值，一般由维度编码和事实数据组成。

公众号：旧时光大数据

表示对分析主题所属类型的描述。比如"昨天早上张三在京东花费200元购买了一个皮包"。那么以购买为主题进行分析，可从这段信息中提取三个维度：时间维度(昨天早上)，地点维度(京东), 商品维度(皮包)。通常来说维度表信息比较固定，且数据量小。

维度表：表格里存放了具有独立属性和层次结构的数据，一般由维度编码和对应的维度说明(标签)组成。

表示对分析主题的度量。比如上面那个例子中，200元就是事实信息。事实表包含了与各维度表相关联的外码，并通过JOIN方式与维度表关联。事实表的度量通常是数值类型，且记录数会不断增加，表规模迅速增长。

比如你要分析产品销售情况, 你可以选择按类别来进行分析,或按区域来分析. 这样的按..分析就构成一个维度。前面的示例就可以有两个维度：类型和区域。下面是两个常见的维度表结构：

产品维度表：Prod_id, Product_Name, Category, Color, Size, Price

时间维度表：TimeKey, Season, Year, Month, Date

而事实表是数据聚合后依据某个维度生成的结果表。它的结构示例如下：

销售事实表：Prod_id(引用产品维度表), TimeKey(引用时间维度表), SalesAmount(销售总量，以货币计), Unit(销售量)

事实数据和维度数据的识别必须依据具体的主题问题而定。**事实表用来存储事实的度量（measure）及指向各个维的外键值。维表用来保存该维的元数据，即维的描述信息，包括维的层次及成员类别等。**

什么是ER模型?

问过的一些公司：虎牙(2021.08)

参考答案：

在信息系统中，**将事务抽象为"实体"（Entity）、"属性"（Property）、"关系"（Relationship）来表示数据关联和事物描述**，这种对数据的抽象建模通常被称为ER实体关系模型。

实体：通常为参与到过程中的主体，客观存在的，比如商品、仓库、货位、汽车，此实体非数据库表的实体表。

属性：对主体的描述、修饰即为属性，比如商品的属性有商品名称、颜色、尺寸、重量、产地等。

关系：现实的物理事件是依附于实体的，比如商品入库事件，依附实体商品、货位，就会有"库存"的属性产生；用户购买商品，依附实体用户、商品，就会有"购买数量"、"金额"的属性产品。

实体之间建立关系时，存在对照关系：

- 1:1：即1对1的关系
- 1:n：即1对多的关系
- n:m：即多对多的关系

在日常建模中，"实体"用矩形表示，"关系"用菱形，"属性"用椭圆形。ER实体关系模型也称为E-R关系图。

OLAP、OLTP解释（区别）

问过的一些公司：美团x2，字节，ebay，阿里

参考答案：

公众号：旧时光大数据

操作型处理，叫**联机事务处理 OLTP**（On-Line Transaction Processing）

也可以称面向交易的处理系统，它是针对具体业务在数据库联机的日常操作，通常对少数记录进行查询、修改。用户较为关心操作的响应时间、数据的安全性、完整性和并发支持的用户数等问题。传统的数据库系统作为数据管理的主要手段，主要用于操作型处理。

分析型处理，叫**联机分析处理 OLAP**（On-Line Analytical Processing）

一般针对某些主题的历史数据进行分析，支持管理决策。

OLTP与OLAP的异同如下表所示：

操作型处理	分析型处理
细节的	综合的或提炼的
实体-关系（ER）模型	星型或雪花模型
存取瞬间数据	存储历史数据，不包含最近的数据
可更新的	只读，只追加
一次操作一个单元	一次操作一个集合
性能要求高，响应时间短	性能要求宽松
面向事务	面向分析
一次操作数据量小	一次操作数据量大
支持日常操作	支持决策需求
数据量小	数据量大
客户订单、库存水平和银行账户等	客户收益分析、市场细分等

三范式是什么，举些例子

问过的一些公司：字节，快手，转转

参考答案：

1、范式理论概述

目前业界范式有：第一范式(1NF)、第二范式(2NF)、第三范式(3NF)、巴斯-科德范式(BCNF)、第四范式(4NF)、第五范式(5NF)。

范式的标准定义是：符合某一种级别的关系模式的集合，表示一个关系内部各属性之间的联系的合理化程度。通俗地讲，**范式可以理解为一张数据表的表结构所符合的某种设计标准的级别。**

使用范式的根本目的是：**减少数据冗余，尽量让每个数据只出现一次，获取数据时通过 join 拼接出最后的数据。**

2、范式基本概念

案例所需表如下：

公众号：旧时光大数据

学号	姓名	系名	系主任	课名	分数
20170901176	王小强	计算机系	马小腾	C 语言	95
20170901176	王小强	计算机系	马小腾	计算机基础	99
20170901176	王小强	计算机系	马小腾	高等数学	80
20170901179	李阳	经管系	王小石	经济学	95
20170901179	李阳	经管系	王小石	管理学	92
20170901186	张小俊	数学系	钱小森	高等数学	89
20170901186	张小俊	数学系	钱小森	线性代数	96

1）函数依赖

若在一张表中，在属性（或属性组）X 的值确定的情况下，必定能确定属性 Y 的值，那么就可以说 Y 函数依赖于 X，写作 X → Y。

也就是说，在数据表中，如果符合函数依赖，那么不存在任意两条记录，它们在 X 属性（或属性组）上的值相同，而在 Y 属性上的值不同。这也就是“函数依赖”名字的由来，类似于函数关系 y = f(x)，在 x 的值确定的情况下，y 的值一定是确定的。

例如，对于表 3 中的数据，找不到任何一条记录，它们的学号相同而对应的姓名不同。所以我们可以说姓名函数依赖于学号，写作 学号 → 姓名。但是反过来，因为可能出现同名的学生，所以有可能不同的两条学生记录，它们在姓名上的值相同，但对应的学号不同， 所以我们不能说学号函数依赖于姓名。

表中其他的函数依赖关系还有如：

- 系名 → 系主任
- 学号 → 系主任
- (学号，课名) → 分数

以下函数依赖关系则不成立：

- 学号 → 课名
- 学号 → 分数
- 课名 → 系主任
- (学号，课名) → 姓名

2）完全函数依赖

在一张表中，若 X → Y，且对于 X 的任何一个真子集（假如属性组 X 包含超过一个属性的话），X' → Y 不成立，那么我们称 Y 对于 X 完全函数依赖，记做：

$$X \xrightarrow{F} Y$$

例如：

学号 $\xrightarrow{F}$ 姓名

(学号，课名) $\xrightarrow{F}$ 分数（注：因为同一个的学号对应的分数不确定，同一个课名 对应的分数也不确定）

3）部分函数依赖

假如 Y 函数依赖于 X，但同时 Y 并不完全函数依赖于 X ，那么我们就称 Y 部分函数依赖于 X，记做：

$$X \xrightarrow{P} Y$$

公众号：旧时光大数据

简单来说，(学号，课名) → 系名，学号 → 系名，那么(学号，课名) $\xrightarrow{P}$ 系名。

4）传递函数依赖

假如 Z 函数依赖于 Y，且 Y 函数依赖于 X，且 Y 不包含于 X，X 不函数依赖于 Y，那么我们就称 Z 传递函数依赖于 X，记做：

$$X \xrightarrow{T} Y$$

简单来说，系名 → 系主任，学号 → 系名，那么学号 $\xrightarrow{T}$ 系主任。

3、一范式

一范式（1NF）：**域应该是原子性的，即数据库表的每一列都是不可分割的原子数据项。**

域：域就是列的取值范围，比如性别的域就是（男，女）

不符合一范式的表格设计如下：

ID	商品	商家ID	用户ID
001	5 台电脑	XXX旗舰店	00001

很明显上表所示的表格设计是不符合第一范式的，商品列中的数据不是原子数据项，是可以进行分割的，因此对表格进行修改，让表格符合第一范式的要求，修改结果如下表所示：

ID	商品	数量	商家ID	用户ID
001	电脑	5	XXX旗舰店	00001

实际上，**1NF 是所有关系型数据库的最基本要求**，你在关系型数据库管理系统（RDBMS），例如 SQL Server，Oracle，MySQL 中创建数据表的时候，**如果数据表的设计不符合这个最基本的要求，那么操作一定是不能成功的。**也就是说，只要在 RDBMS 中已经存在的数据表，一定是符合 1NF 的。

4、二范式

二范式（2NF）：在 1NF 的基础上，实体的属性完全函数依赖于主关键字（混合主键），不能存在部分函数依赖于主关键字（混合主键）。

如果存在某些属性只依赖混合主键中的部分属性，那么不符合二范式。

不符合二范式的表格设计如下：

学生 ID	姓名	所属系	系主任	所修课程	分数
20170901176	王小强	计算机系	马小腾	000001	95
20170901176	王小强	计算机系	马小腾	000002	99

上述表格中是**混合主键（学生 ID + 所修课程）**，但是所属系和系主任这两个属性只依赖于混合主键中的学生 ID 这一个属性，因此，不符合第二范式。

如果有一天学生的所属系要调整，那么所属系和系主任这两列都需要修改，如果这个学生修了多门课程，那么表中的多行数据都要修改，这是非常麻烦的，不符合第二范式。

为了消除这种部分依赖，只有一个办法，就是**将大数据表拆分成两个或者更多个更小的数据表。**

符合二范式的表格设计如下：

公众号：旧时光大数据

学生 ID	所修课程	分数
20170901176	000001	95
20170901176	000002	99

学生 ID	所属系	系主任
20170901176	计算机系	马小腾
20170901176	计算机系	马小腾

学生 ID	姓名
20170901176	王小强

通过上述的修改，当一个学生的所属系需要调整时，不管学生修了多少门课程，都只需要改变上表中的一行数据即可。

5、三范式

3NF 在 2NF 的基础之上，消除了非主属性对于主键（复合主键）的传递依赖。

不符合三范式的表格设计如下：

订单ID	商品ID	商品颜色	商品尺寸	商家ID	用户ID
001	0001	深空灰	300 *270* 40	XXX旗舰店	00001

很明显，上表中，商品颜色依赖于商品 ID，商品 ID 依赖于订单 ID，那么非主属性商品颜色就传递依赖于订单 ID，因此不符合三范式，解决方案是**将大数据表拆分成两个或者更多个更小的数据表。**

符合三范式的表格设计如下：

订单ID	商品ID	商家ID	用户ID
001	0001	XXX旗舰店	00001

订单ID	商品ID	商家ID	用户ID
001	0001	XXX旗舰店	00001

以下为扩展内容：

6、BC范式（BGFN）

定义：关系模式R<U,F>中，若每一个决定因素都包含码，则R<U,F>属于BCFN。

理解：根据定义我们可以得到结论，一个满足BC范式的关系模式有：

- 所有非主属性对每一个码都是完全函数依赖；
- 所有主属性对每一个不包含它的码也是完全函数依赖；
- 没有任何属性完全函数依赖于非码的任何一组属性。

公众号：旧时光大数据

例如有关系模式C(Cno, Cname, Pcno)，Cno，Cname，Pcno依次表示课程号、课程名、先修课。可知关系C只有一个码Cno，且没有任何属性对Cno部分函数依赖或传递函数依赖，所以关系C属于第三范式，同时Cno是C中的唯一决定因素，所以C也属于BC范式。

7、第四范式（4NF）

定义： 限制关系模式的属性之间不允许有非平凡且非函数依赖的多值依赖。

理解： 显然一个关系模式是4NF，则必为BCNF。也就是说，当一个表中的非主属性互相独立时（3NF），这些非主属性不应该有多值，若有多值就违反了4NF。

8、第五范式（5NF）

第五范式，又称为完美范式， 越往下，冗余度越低。

第五范式有以下要求：

- 必须满足第四范式；
- 表必须可以分解为较小的表，除非那些表在逻辑上拥有与原始表相同的主键。

第五范式是在第四范式的基础上做的进一步规范化。第四范式处理的是相互独立的多值情况，而第五范式则处理相互依赖的多值情况。

9、反范式

一般来说，数据库只需要满足第三范式（3NF）即可。

反范式化设计数据库，是为了提高查询效率，采用空间换时间的实现思路。

应用场景：当**冗余的信息有价值**或者能**大幅度提高查询效率**的时候，我们才会采取反范式的优化。

一些情况下，比如存在频繁查询时，可以容忍适当的冗余设计，目的是减少多表关联查询，提高效率。

例如：订单表中冗余了商品信息和用户相关信息，避免查询订单时关联用户表和商品表去查询相关信息，提高效率。

优点：增加数据表中的冗余字段来提高数据库的读性能

缺点：

- 存储空间变大了
- 一个表中字段做了修改，另一个表中冗余字段也需要做同步修改，否则数据不一致
- 若采用存储过程来支持数据的更新、删除等额外操作，如果更新频繁会非常消耗系统资源
- 在数据量小的情况下，反范式不能体现性能的优势，可能还会让数据库的设计更加复杂

10、范式化设计与反范式设计的优缺点

1）范式化设计（时间换空间）

优点：范式化的表减少了数据冗余，数据表更新操作快、占用存储空间小。

缺点：

- 查询时需要对多个表进行关联，查询性能降低
- 索引优化会更难进行

2）反范式化设计（空间换时间）

反范式化的过程就是通过增加数据表中的冗余字段来提高数据库的读（查询）性能，但冗余数据会牺牲数据一致性。

优点：

- 可以减少表关联

公众号：旧时光大数据

- 可能更好的进行索引优化

缺点：

- 存在大量冗余数据
- 数据维护成本更高（删除异常，插入异常，更新异常）

维度设计过程，事实设计过程

问过的一些公司：美团，快手

参考答案：

1、维度建模

维度建模主要有4个步骤：选取业务过程、定义粒度、确定维度和确定事实。这4个步骤贯穿了维度建模的整个过程和环节。

1）选取业务

过程业务过程即企业和组织的业务活动，它们一般都有相应的源头业务系统支持。对于一个超市来说，其最基本的业务活动就是用户收银台付款；对于一个保险公司来说，最基本的业务活动是理赔和保单等。当然在实际操作中，业务活动有可能并不是那么简单直接，此时听取用户的意见通常是这一环节最为高效的方式。

需要注意的是，这里谈到的业务过程并不是指业务部门或者职能。模型设计中，应将注意力集中放在业务过程而不是业务部门，如果建立的维度模型是同部门捆绑在一起的，就无法避免出现数据不一致的情况（如业务编码、含义等）。因此，确保数据一致性的最佳办法是从企业和公司全局与整体角度，对于某一个业务过程建立单一的、一致的维度模型。

2）定义粒度

定义粒度意味着对事实表行实际代表的内容和含义给出明确的说明。粒度传递了事实表度量值相联系的细节所达到的程度的信息。其实质就是如何描述事实表的单个行。

典型的粒度定义包括：

- 超市顾客小票的每一个子项；
- 医院收费单的明细子项；
- 个人银行账户的每一次存款或者取款行为；
- 个人银行账户每个月的余额快照。

对于维度设计来说，在事实表粒度上达成一致非常重要，如果没有明确的粒度定义，则不能进入后面的环节。如果在后面的环节中发现粒度的定义不够或者是错误的，那么也必须返回这一环节重新定义粒度。

在定义粒度过程中，应该最大限度地选择业务过程中最为原子性的粒度，这样可以带来后续的最大灵活度，也可以满足业务用户的任何粒度的分析需求。

3）确认维度

定义了粒度之后，相关业务过程的细节也就确定了，对应的维度就很容易确定。正如前文所述，维度是对度量的上下文和环境的描述。通过维度，业务过程度量与事实就会变得丰富和丰满起来。对于订单来说，常见的维度会包含商品、日期、买家、卖家、门店等而每一个维度还可以包含大量的描述信息，比如商品维度表会包含商品名称、标签价、商品品牌、商品类目、商品上线时间等。

4）确认事实

公众号：旧时光大数据

确定事实通过业务过程分析可能要分析什么来确定。定义粒度之后，事实和度量一般也很容易确定，比如超市的订单活动，相关的度量显然是销售数量和销售金额。

在实际维度事实设计中，可能还会碰到度量拆分的问题，比如超市开展单个小票满10减10元的活动，如果小票金额超过10元，这10元的优惠额如何分配到每一个小票子项实际设计中，可以和业务方具体讨论并制订具体的拆分分配算法。

2、事实表设计

事实建模主要有5个步骤：选取业务过程、定义粒度、确定维度、确定事实和冗余维度。

1）选择业务过程以及确定事实表类型

比如淘宝的订单流转的业务过程有四个：创建订单，买家付款，卖家发货，买家确认收货。

明确了业务过程后，根据具体业务需求来选择与维度建模有关的业务过程。

比如买家付款这个业务过程，那么事实表应只包括买家付款这一个业务过程的单事务事实表。

总而言之就是选择了哪些业务过程，那么所建立的事实表应为包含了所有业务过程的累积快照事实表。

2）声明粒度

粒度声明非常重要，尽量选择最细级别的原子粒度，以确保事实表的应用具有最大的灵活性，比如一次购物车下单，一个父订单可能是购物车，一个子订单是每个商品的订单，那么订单事实表选择子订单粒度

3）确定维度

完成粒度声明意味着声明了主键，对应的维度组合就可以确定了。

应该选择能够清楚描述业务过程的维度信息。

例如订单事实表，粒度为子订单，相关的维度有卖家、买家、商品，收货人，时间等维度。

4）确定事实

应该选择与业务过程有关的所有事实，且事实的粒度要和声明的粒度一致，比如在淘宝订单付款事务事实表中，同粒度的事实有子订单分摊的支付金额、邮费、优惠金额等。

5）冗余维度

大数据的事实表设计中，冗余尽可能多的维度让下游方便使用，减少连表数量。

维度设计中有整合和拆分，有哪些方法，并详细说明

问过的一些公司：美团

参考答案：

1、垂直整合

在不同来源表中可能会包含这相同的数据集。只是存储的信息不同。如电商的会员相关的表包含了会员基础信息表、会员扩展表、会员等级信息表等，依照维度设计方法，尽量整合到会员维度模型中，丰富其维度属性。

2、水平整合

不同来源表包含不同的数据集，不同子集之间无交叉，也可以存在部分交叉。如上会员体系中还可能存在各种类型会员表，如淘宝生态中 淘宝会员、国际站会员、支付宝会员等水平类型的表。这种情况的整合过程中，首先需要考虑各个会员体系是否有交叉，如果存在交叉，则需要去重；如果不存在交叉，则需要考虑不同子集的自然键是否存在冲突，如果不冲突，则可以考虑将各子集的自然键作为整合后的表

公众号：旧时光大数据

的自然键；另一种方式是设置超自然键，将来源表各子集的自然键加工成个字段作为超自然键。

3、水平拆分

维度通常可以按照类别和类型细分。如某宝系商品表，根据业务线或行业等可以对商品进行细分，如淘宝的商品、天猫的商品、 飞猪旅行的商品、淘宝海外的商品、天猫国际的商品等。不同分类的商品，其维度属性可能相同，也可能不同。航旅商品想比较于普通商品都有都有商品价格、标题、类型、上架时间、类目等维度属性。但是航旅的商品除了有这些公共属性外，还有酒店、点、门票、旅行等自己独特的维度属性。

针对此问题一般有两种解决方案：

- 将维度的不同分类实例化为不同的维度，同时在主维度表中保存公共属性；
- 维护单一维度，包含所有的可能的属性。

具体选择哪种方案，在数据模型设计过程中需要考虑的因素有很多，基本不可能满足各个特性指标的最优化。在设计过程中需要重点考虑以下几个原则。

- 扩展性：当源系统、业务逻辑变化时，能通过较少的成本快速扩展模型，保持核心模型的相对稳定性。软件工程中的高内聚、低稠合的思想是重要的指导方针。
- 效能：在性能和成本方面取得平衡。通过牺牲一定的存储成本，达到性能和逻辑的优化。
- 易用性：模型可理解性高、访问复杂度低。用户能够方便地从模型中找到对应的数据表，并能够方便地查询和分析。

根据数据模型的设计思想我们可以如下考虑和选择：

- 第一个是维度不同分类的属性差异情况。当维度属性随类型变化较大时，将所有可能的属性建立在一个表中时不切合实际的，也没有必要这样做。此时建议采用方案一，定义一个主维度用于存放公共属性；同时定义多个子维度，其中除了包含公共属性外，还包含各自的特殊属性。比如在阿里巴巴数据仓库维度体系中，依据此方法，构建了商品维度、航旅商品维度等。公共属性 般比较稳定，通过核心的商品维度，保证了核心维度的稳定性；通过扩展子维度的方式，保证了模型的扩展性。
- 第二个是业务的关联程度。两个相关性较低的业务，稠合在起弊大于利，对模型的稳定性和易用性影响较大。比如在数据仓库维度体系中，对某宝商品和 *** 商品构建两个维度。业务各自发展在数据仓库层面，属于不同数据集市，一 般不会相互调用，业务分析人员 般只针对本数据集市进行统计分析。如果设计成一个维度，由于不同业务各自发展， 此维度需要变更，淘宝业务变更亦然，稳定性很差；在易用性方面，会给数据使用方造成困扰。

4、垂直拆分

维度是维度建模的基础和灵魂，维度属性的丰富程度直接决定了数据仓库的能力。在进行维度设计时，依据维度设计的原则，尽可能丰富维度属性，同时进行反规范化处理。对于具体实现时可能存在的问题如下：

- 一是在“水平拆分”中提到的，由于维度分类的不同而存在特殊的维度属性，可以通过水平拆分的方式解决此问题。
- 二是某些维度属性的来源表产生时间较早，而某些维度属性的来表产出时间较晚；或者某些维度属性的热度高、使用频繁，而某些维度属性的热度低、较少使用 或或者某些维度属性经常变化，而某些维度属性比较稳定。在“水平拆分”中提到的模型设计的三个原则同样适合解决此问题。

出于扩展性、产出时间、易用性等方面的考虑，设计主从维度。主维度表存放稳定、产出时间早、认读高的属性；从表存放变化较快、产出时间较晚、热度低的属性。例如在商品维表的设计中，可以设计商品主维度和商品的扩展维度。主商品表时间可以每天1点产生，而扩展维度每天3点才能产生。由于商品扩展维度有冗余的库存等变化较快的数据，对于主维度进行缓慢变化的处理较为重要。通过存储的冗余和计算成本的增加，实现了商品主模型的稳定和产出时间的提前，对于整个数据仓库的稳定和下游应用的产出都有较大意义。

公众号：旧时光大数据

事实表设计分几种，每一种都是如何在业务中使用

可回答：1）事实表分类；2）累积型快照事实表做法

问过的一些公司：美团，字节(2021.07)x2

参考答案：

事实表作为数据仓库维度建模的核心，紧紧围绕着业务过程来设计，通过获取描述业务过程的度量来表达业务过程，包含了引用的维度 和与业务过程有关的度量。

1、事实表概述

事实表有三种类型 : **事务事实表、周期快照事实表**和**累积快照事实表。**

1）事务事实表

也称原子事实表，描述业务过程，**跟踪控件或时间上某点的度量事件**，保存的是最原子的数据。

类似于mysql binlog日志，每一次相关的 change 都记录下来，生成一行新的数据。

2）周期快照事实表

以一个周期为时间间隔，来记录事实，一般周期可以是每天、每周、每月、每年等。

只看某个业务过程，比如订单收货，数据按订单收货时间来切分，周期可以为每天、每月等。

3）累积快照事实表

用来**描述过程开始和结束之间的关键步骤事件，覆盖过程的整个生命周期**，通常具有多个日期字段来记录关键时间点；当过程随着生命周期不断变化时，记录也会随着过程的变化而被修改。

要看整个生命周期的多个业务过程，比如：创建订单 → 买家付款 → 卖家发货 → 买家确认收货。粒度是一个订单一行数据，创建订单时间，付款时间，发货时间，收货时间，分别作为一个字段，便于计算不同业务过程的时间间隔。

2、事务表对比

	事务事实表	周期快照事实表	累积快照事实表
时期/时间	离散事务时间点	以有规律的、可预测的	用于时间跨度不确定的不断变化的工作流
日期维度	事务日期	快照日期	相关业务过程涉及的多个日期
粒度	每行代表实体的一个事务	每行代表某时间周期的一个实体	每行代表一个实体的生命周期
事实	事务事实	累积事实	相关业务过程事实和时间间隔事实
事实表加载	插入	插入	插入与更新
事实表更新	不更新	不更新	业务过程变更时更新

3、事实表设计原则

原则 1：尽可能包含所有与业务过程相关的事实

公众号：旧时光大数据

- 分析哪些事实与业务过程相关，是设计过程中非常重要的关注点；
- 在事实表中，尽量包含所有与业务过程相关的事实，即使存在冗余，由于事实通常是数字型，存储开销不会太大；

原则 2：只选择与业务过程相关的事实

- 如订单的下单这个业务过程，事实表中不应该存在支付金额这个表示支付业务过程的事实；

原则 3：分解不可加性事实为可加的组件

- 如订单的优惠率，应分解为订单原价金额与订单优惠金额两个事实存储在事实表中；

原则 4：在选择维度和事实之前必须先声明粒度

- 粒度用于确定事实表中一行所表示业务的细节层次，决定了维度模型的扩展性；
- 每个维度和事实必须与所定义的粒度保持一致；
- 设计事实表时，粒度定义越细越好，一般从最低级别的原子粒度开始；
    - 因为原子粒度提供了最大限度的灵活性，可以支持无法预期的各种细节层次的用户需求；

原则 5：在同一个事实表中不能有多种不同粒度的事实

- 疑问：怎么判断不同事实的粒度是否相同？
    - 粒度为票一级；（实际业务中，一个订单可以同时支付多张票）
    - 票支付金额和票折扣金额，两个事实的粒度为 “票级”，与定义的粒度一致；
    - 订单支付金额和订单票数，两个事实的粒度为 “订单级”，属于上一层订单级数据，与 “票级” 事实表的粒度不一致，且不能进行汇总；
    - 如果，以订单金额和订单票数这两个维度汇总总金额和总票数，会造成大量的重复计算；

原则 6：事实的单位要保持一致

- 如订单金额、订单优惠金额、订单运费这 3 个事实，应该采用统一的计量单位，统一为元或者分，以方便使用；

原则 7：对事实的 null 值要处理

- 原因：在数据库中，null 值对常用数字型字段的 SQL 过滤条件都不生效；如，大于、小于、等于、大于或等于、小于或等于；
- 处理：用 0 代替 null ；

原则 8：使用退化维度提高事实表的易用性

- 事实表中存储各种类型的常用维度信息，较少下游用户使用时关联多个表的操作；
- 通过退化维度，可以实现对事实表的过滤查询、控制聚合层次、排序数据、定义主从关系等；
- 易用性：对事实表，更较少关联操作、过滤查询、控制聚合层次、排序数据、定义主从关系等；

在 Kimball 的维度建模中，通常按照星形模型的方式设计，通过事实表的外键关联专门的维表，这种方式来获取维度，谨慎使用退化维表；这与大数据领域的事实表设计不一样。

- 思路：通过增加冗余存储，减少计算开销，提高使用效率

4、事实表设计方法

上上一题也有说过，这里大同小异，也可以看上上一题的

Kimball 的维度模型设计 4 步法：选择业务过程、声明粒度、确定维度、确定事实。

当前的互联网大数据环境，维度模型的设计，是基于 Kimball 的四步维度建模方法进行了更进一步的改进。

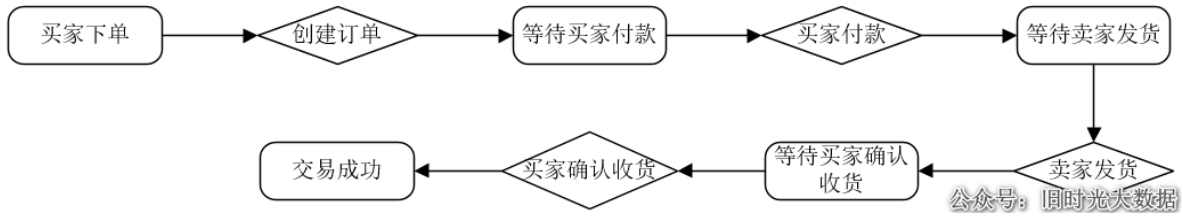
第一步：选择业务过程及确定事实表类型

公众号：旧时光大数据

思路：详细分析需求，对业务的整个生命周期进行分析，明确**关键的业务步骤**，从而选择与需求有关的业务过程。

以实例说明：如何选择业务过程？如何确定事实表类型？

例：淘宝的一个交易订单



1）分析业务的生命周期：如上图，**业务过程通常使用行为动词表示业务执行的活动**；

2）明确关键的业务步骤：该订单流转的业务过程有 4 个：创建订单 → 买家付款 → 卖家发货 → 买家确认收货；

3）根据业务需求，选择与维度建模有关的业务过程；

- 如是选择 “买家付款” 这个业务过程，还是选择 “创建订单” 和 “买家付款” 这两个业务过程，具体根据业务情况来定。

4）根据所选的业务过程确定事实表类型；

- 如选择 “买家付款” 这个业务过程，则事实表类型应为只包含买家付款这一个业务过程的 “单事务事实表”；
- 如选择了所有 4 个业务过程，并且需要分享各业务过程的时间间隔，则事实表类型应为包含了所有 4 个业务过程的 “累积快照事实表”。

第二步：声明粒度

粒度的作用：

- 粒度的声明，意味着**精确定义事实表的每一行所表示的业务含义**；
- 明确的粒度能够确保对实表中行的意思的理解不会产生混淆，保证所有的事实按照同样的细节层次记录。

粒度的选择：尽量选择最细级别的原子粒度，以确保事实表的应用具有最大的灵活性。

- 灵活性：支持无法预期的各种细节层次的用户需求；
- 对于订单级别，粒度可以定义为最细的订单级别。（如，父子订单，事实表的粒度可以定 “子订单级别”；）

第三步：确定维度

完成了粒度声明，就意味着确定了主键，对应的维度组合以及相关的维度字段也可以确定了。

选择维度的原则：应该选择能够描述清楚业务过程所处的环境的维度信息；

- 如淘宝订单 “付款事务事实表” 中，粒度为 “子订单”，相关的维度有买家、卖家、商品、收货人信息、业务类型、订单时间等。

第四步：确定事实

确定原则：**选择与业务过程有关的所有事实，且事实的粒度要与所声明的事实表的粒度一致**；

思路：可以通过回答 “过程的度量是什么” 来确定；

注意：将不可加性事实分解为可加的组件；（分解的原则：可以通过分解后的可加的属性值，计算得到不可加性事实）

公众号：旧时光大数据

单事务事实表、多事务事实表区别与作用

可回答：事务表有几种类别

问过的一些公司：拼多多，美团(2021.09)

参考答案：

事务事实表用于跟踪定义业务过程的个体行为

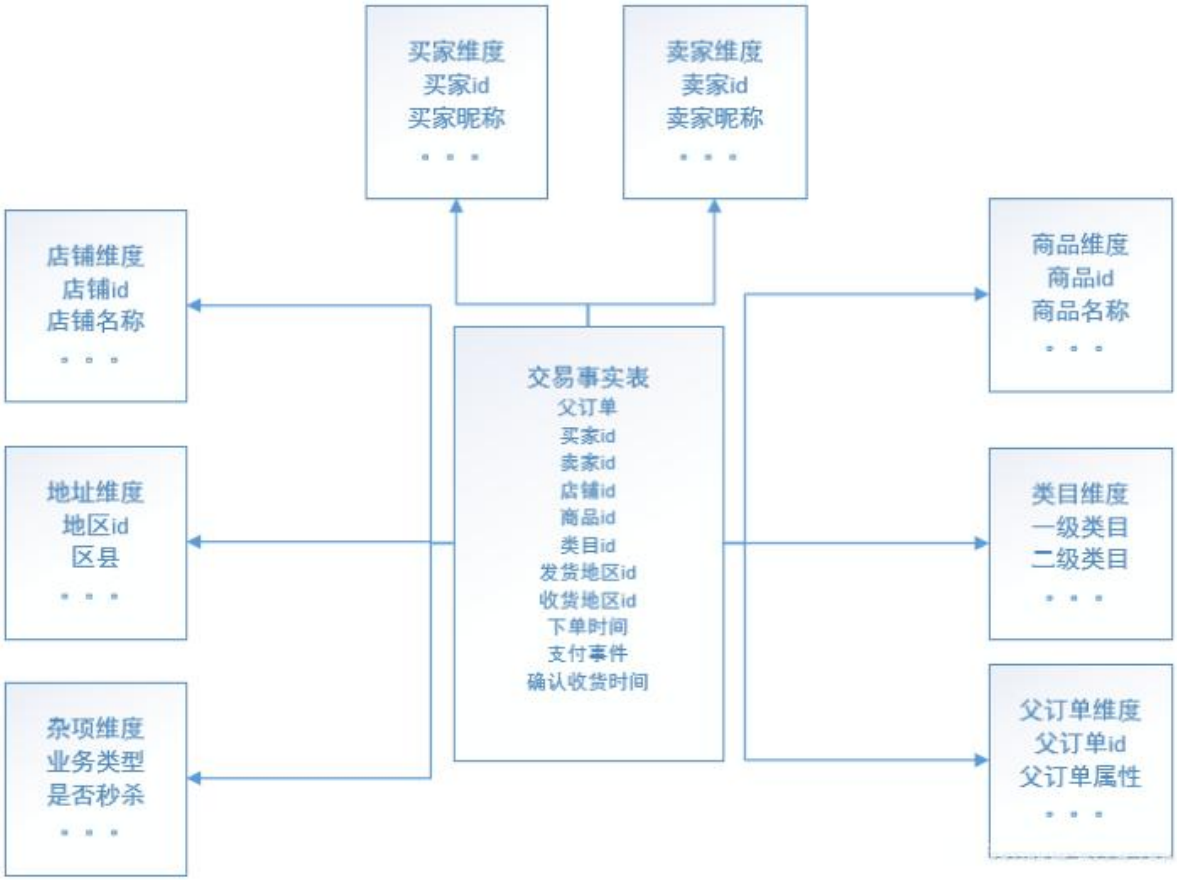
设计案例场景如下：为交易事务设计事实表

1）业务分析：交易事务包括下单、支付、发货、完结四个业务过程。

2）确定粒度：同一个订单中可以包含多个在商品，每个商品对应一个子订单。在上述四个业务过程中下单、支付、完结选择子订单作为粒度，而发货业务过程包含物流信息，以父订单为粒度。

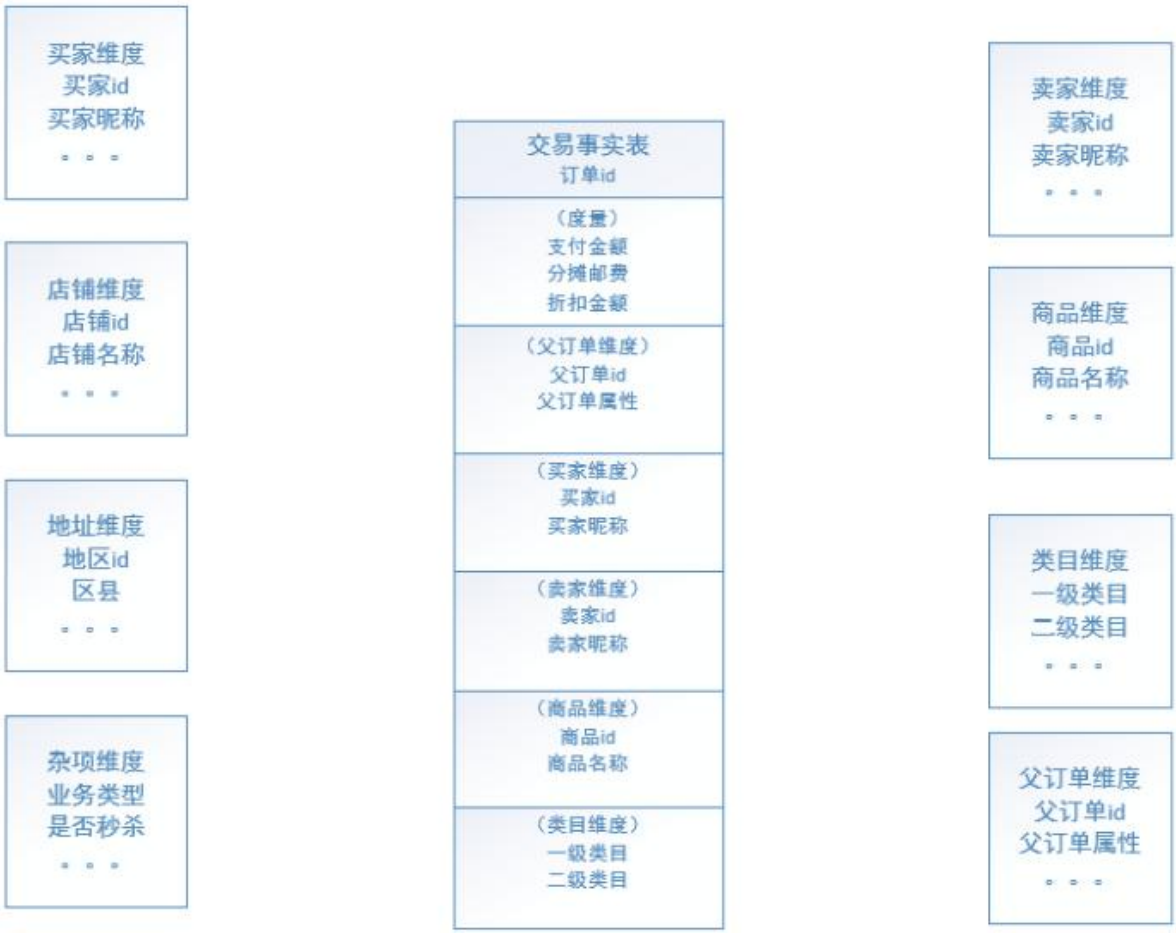
3）确定维度：卖家、买家、商品、商品类目、发货地区、收货地区、父订单一级杂项维度。

4）确定事实：每个业务过程有自己的事实，如下单过程的下单金额、下单数量；支付过程的支付金额、积分金额等。



5）冗余维度：为了提升效率，把常用的维度荣誉到事实表

公众号：旧时光大数据



1、单事务事实表

一个业务过程一张事实表，方便对每个业务做独立分析

2、多事务事实表

将不同业务过程放在同一个事实表中，又可以分为不同业务过程使用不同事实字段和不同业务过程使用相同事实字段两种。

1）不同业务过程使用不同事实字段，一般用于业务相似，粒度相同但是业务过程的度量差异大的场景。有两个典型的问题：

- 在数据中遇到不是当前业务过程的度量，采用零值处理
- 表中存在多个业务，如何标记？给每一个数据加一个属性标识是否当日业务

2）不同业务过程使用相同事实字段，用一个标签字段标识是那种业务（如商品的收藏/删除）。一般用于业务相似，粒度相同同时业务过程的度量差异不大的场景。但是有一个问题要注意，因为用同一个字段标识不同业务的度量，所以数据一个周期内会有多条记录。

3、单事务事实表与多事务事实表对比

	单事务事实表	多事务事实表
粒度	相互不相关	相同粒度
维度	相互不相关	一致
事实	只取当前业务事实	同时保留多个过程事实，非当前业务的事实用零值处理
冗余维度	多个业务过程需要冗余多次	多个业务过程只冗余一次
计算成本	较多，每个业务单独计算一次	多个业务融合在一张表，只需计算一次

另外，如果一个业务过程的事实量巨大，不宜使用多事务事实表，会造成大量零值。

说下一致性维度、一致性事实、总线矩阵

问过的一些公司：拼多多，好未来(2021.08)

参考答案：

在Kimball的维度建模的数据仓库中，关于多维体系结构（MD）有三个关键性概念：总线架构（Bus Architecture），一致性维度（Conformed Dimension）和一致性事实（Conformed Fact）。

1、总线架构

在多维体系结构（MD）（也就是总线架构）的数据仓库架构中，主导思想是分步建立数据仓库，由数据集市组合成企业的数据仓库。但是，在建立第一个数据集市前，架构师首先要做的就是设计出在整个企业内具有统一解释的标准化的维度和事实，即一致性维度和一致性事实。而开发团队必须严格的按照这个体系结构来进行数据集市的迭代开发。

一致性维度就好比企业范围内的一组总线，不同数据集市的事实的就好比插在这组总线上的元件。这也是称之为总线架构的原因。

实际设计过程中，我们通常把总线架构列表成矩阵的形式，其中列为一致性维度，行为不同的业务处理过程，即事实，在交叉点上打上标记表示该业务处理过程与该维度相关。这个矩阵也称为总线矩阵（Bus Matrix）。

总线架构和一致性维度、一致性事实共同组成了Kimball的多维体系结构的基础，也建立了一套可以逐步建立数据仓库的方法论。由于总线架构是多维体系结构的核心，所以我们有时就把多维体系结构直接称为总线架构。

2、价值链的意义

每家机构都有一个关键业务过程组成的潜在价值链，这个价值链确定机构主体活动的自然逻辑流程。数据仓库建设就是围绕着价值链建立一致化的维度和事实。

3、数据总矩阵

公众号：旧时光大数据

矩阵的每一行对应都对应机构中的一个业务过程，每一列都和一个业务维度相对应，用叉号填充显示的是和每一行相关的列。业务过程应该先从单个数据源系统开始，然后再进行多数据源的合并。

企业数据仓库总线矩阵是DW/BI系统的一个总体数据架构，提供了一种可用于分解企业数据仓库规划任务的合理方法，开发团队可以独立的，异步的完成矩阵的各个业务过程，迭代地去建立一个集成的企业数据仓库。

数据仓库总线矩阵						
维度 / 业务过程	日期	产品	存储	仓库	供应商	运输商
购买订单	X	X			X	
商品进货	X	X				X
商品库存	X	X		X		
商场进货	X	X	X			X
商场库存	X	X		X		
商场营销	X	X				

4、一致性维度

在多维体系结构中，没有物理上的数据仓库，由物理上的数据集市组合成逻辑上的数据仓库。而且数据集市的建立是可以逐步完成的，最终组合在一起，成为一个数据仓库。如果分步建立数据集市的过程出现了问题，数据集市就会变成孤立的集市，不能组合成数据仓库，而一致性维度的提出正式为了解决这个问题。

一致性维度的范围是总线架构中的维度，即可能会在多个数据集市中都存在的维度，这个范围的选取需要架构师来决定。一致性维度的内容和普通维度并没有本质上区别，都是经过数据清洗和整合后的结果。一致性维度建立的地点是多维体系结构的后台（Back Room），即数据准备区。

在多维体系结构的数据仓库项目组内需要有专门的维度设计师，他的职责就是建立维度和维护维度的一致性。在后台建立好的维度同步复制到各个数据集市。这样所有数据集市的这部分维度都是完全相同的。建立新的数据集市时，需要在后台进行一致性维度处理，根据情况来决定是否新增和修改一致性维度，然后同步复制到各个数据集市。这是不同数据集市维度保持一致的要点。

在同一个集市内，一致性维度的意思是两个维度如果有关系，要么就是完全一样的，要么就是一个维度在数学意义上是另一个维度的子集。

例如，如果建立月维度话，月维度的各种描述必须与日期维度中的完全一致，最常用的做法就是在日期维度上建立视图生成月维度。这样月维度就可以是日期维度的子集，在后续钻取等操作时可以保持一致。如果维度表中的数据量较大，出于效率的考虑，应该建立物化视图或者实际的物理表。这样，维度保持一致后，事实就可以保存在各个数据集市中。虽然在物理上是独立的，但在逻辑上由一致性维度使所有的数据集市是联系在一起，随时可以进行交叉探察等操作，也就组成了数据仓库。

5、一致性事实

在建立多个数据集市时，完成一致性维度的工作就已经完成了一致性的80% - 90%的工作量。余下的工作就是建立一致性事实。一致性事实和一致性维度有些不同，一致性维度是由专人维护在后台（Back Room），发生修改时同步复制到每个数据集市，而事实表一般不会在多个数据集市间复制。需要查询多个数据集市中的事实时，一般通过交叉探查（drill across）来实现。

公众号：旧时光大数据

为了能在多个数据集市间进行交叉探查，一致性事实主要需要保证两点：第一个是KPI的定义及计算方法要一致，第二个是事实的单位要一致性。如果业务要求或事实上就不能保持一致的话，建议不同单位的事实分开建立字段保存。

这样，一致性维度将多个数据集市结合在一起，一致性事实保证不同数据集市间的事实数据可以交叉探查，一个分布式的数据仓库就建成了。

6、小结

总线矩阵：业务过程和维度的交点。

一致性维度：同一集市的维度表，内容相同或包含。一致性维度要么是统一的，要么是维度表的一个子集。

一致性事实：不同集市的同一事实，需保证口径一致，单位统一。指每个度量在整个数据仓库中都是唯一的统计口径，为了避免歧义，一个度量只有唯一的业务术语。

从ODS层到DW层的ETL，做了哪些工作?

问过的一些公司：快手

参考答案：

这里，我们将数据模型分为三层：数据运营层(原始数据层)（ ODS ）、数据仓库层（DW）和数据应用层（APP）。

ODS层存放的是接入的原始数据，DW层是存放我们要重点设计的数据仓库中间层数据，APP是面向业务定制的应用数据。

1、数据运营层：ODS（Operational Data Store）

“面向主题的”数据运营层，也叫ODS层，是最接近数据源中数据的一层，数据源中的数据，经过抽取、洗净、传输，也就说传说中的 ETL 之后，装入本层。本层的数据，总体上大多是按照源头业务系统的分类方式而分类的。

一般来讲，为了考虑后续可能需要追溯数据问题，因此对于这一层就不建议做过多的数据清洗工作，原封不动地接入原始数据即可，至于数据的去噪、去重、异常值处理等过程可以放在后面的DWD层来做。

2、数据仓库层：DW（Data Warehouse）

数据仓库层是我们在做数据仓库时要核心设计的一层，在这里，**从 ODS 层中获得的数据按照主题建立各种数据模型。**DW层又细分为 DWD（Data Warehouse Detail）层、DWM（Data WareHouse Middle）层和DWS（Data WareHouse Servce）层。

1）数据明细层：DWD（Data Warehouse Detail）

该层一般保持和ODS层一样的数据粒度，并且提供一定的数据质量保证。同时，为了提高数据明细层的易用性，**该层会采用一些维度退化手法，将维度退化至事实表中，减少事实表和维表的关联。**

另外，在**该层也会做一部分的数据聚合，将相同主题的数据汇集到一张表中**，提高数据的可用性。

2）数据中间层：DWM（Data WareHouse Middle）

该层**会在DWD层的数据基础上，对数据做轻度的聚合操作，生成一系列的中间表**，提升公共指标的复用性，减少重复加工。

直观来讲，就是对通用的核心维度进行聚合操作，算出相应的统计指标。

3）数据服务层：DWS（Data WareHouse Servce）

公众号：旧时光大数据

又称数据集市或宽表。**按照业务划分，如流量、订单、用户等，生成字段比较多的宽表**，用于提供后续的业务查询，OLAP分析，数据分发等。

一般来讲，该层的数据表会相对比较少，一张表会涵盖比较多的业务内容，由于其字段较多，因此一般也会称该层的表为宽表。

在实际计算中，如果直接从DWD或者ODS计算出宽表的统计指标，会存在计算量太大并且维度太少的问题，因此**一般的做法是，在DWM层先计算出多个小的中间表，然后再拼接成一张DWS的宽表。**由于宽和窄的界限不易界定，也可以去掉DWM这一层，只留DWS层，将所有的数据在放在DWS亦可。

3、数据应用层：APP（Application）

在这里，主要是提供给数据产品和数据分析使用的数据，一般会存放在 ES、PostgreSql、Redis等系统中供线上系统使用，也可能会存在 Hive 或者 Druid 中供数据分析和数据挖掘使用。比如我们经常说的报表数据，一般就放在这里。

数据仓库与（传统）数据库的区别?

问过的一些公司：网易，字节，浩鲸云，美团，蔚来(2021.09)

参考答案：

数据库：传统的关系型数据库的主要应用，主要是基本的、日常的事务处理，例如银行交易。

数据仓库：数据仓库系统的主要应用主要是OLAP（On-Line Analytical Processing），支持复杂的分析操作，侧重决策支持，并且提供直观易懂的查询结果。

- 业务数据库中的数据结构是为了完成交易而设计的，不是为了而查询和分析的便利设计的。
- 业务数据库大多是读写优化的，即又要读（查看商品信息），也要写（产生订单，完成支付）。因此对于大量数据的读（查询指标，一般是复杂的只读类型查询）是支持不足的。

而怎么解决这个问题，此时我们就需要建立一个数据仓库了，公司也算开始进入信息化阶段了。数据仓库的作用在于：

- 数据结构为了分析和查询的便利；
- 只读优化的数据库，即不需要它写入速度多么快，只要做大量数据的复杂查询的速度足够快就行了。

那么在这里前一种业务数据库（读写都优化）的是业务性数据库，后一种是分析性数据库，即数据仓库。

数据库与数据仓库的区别实际讲的是OLTP与OLAP的区别：

- 操作型处理，叫联机事务处理OLTP（On-Line Transaction Processing），也可以称面向交易的处理系统，它是针对具体业务在数据库联机的日常操作，通常对少数记录进行查询、修改。用户较为关心操作的响应时间、数据的安全性、完整性和并发的支持用户数等问题。传统的数据库系统作为数据管理的主要手段，主要用于操作型处理。
- 分析型处理，叫联机分析处理OLAP（On-Line Analytical Processing）一般针对某些主题历史数据进行分析，支持管理决策。

OLTP与OLAP的异同如下表所示：

操作型处理	分析型处理
细节的	综合的或提炼的
实体-关系（ER）模型	星型或雪花模型
存取瞬间数据	存储历史数据，不包含最近的数据
可更新的	只读，只追加
一次操作一个单元	一次操作一个集合
性能要求高，响应时间短	性能要求宽松
面向事务	面向分析
一次操作数据量小	一次操作数据量大
支持日常操作	支持决策需求
数据量小	数据量大
客户订单、库存水平和银行账户等	客户收益分析、市场细分等

数据质量是怎么保证的，有哪些方法保证

可回答：如何保证数据质量

问过的一些公司：腾讯社招，好未来(2021.08)，京东(2021.09)

参考答案：

1）**从技术层面来说**，需要构建一套高效、健壮的ETL程序，以此保证数据清洗、转换后数据的正确性和一致性。

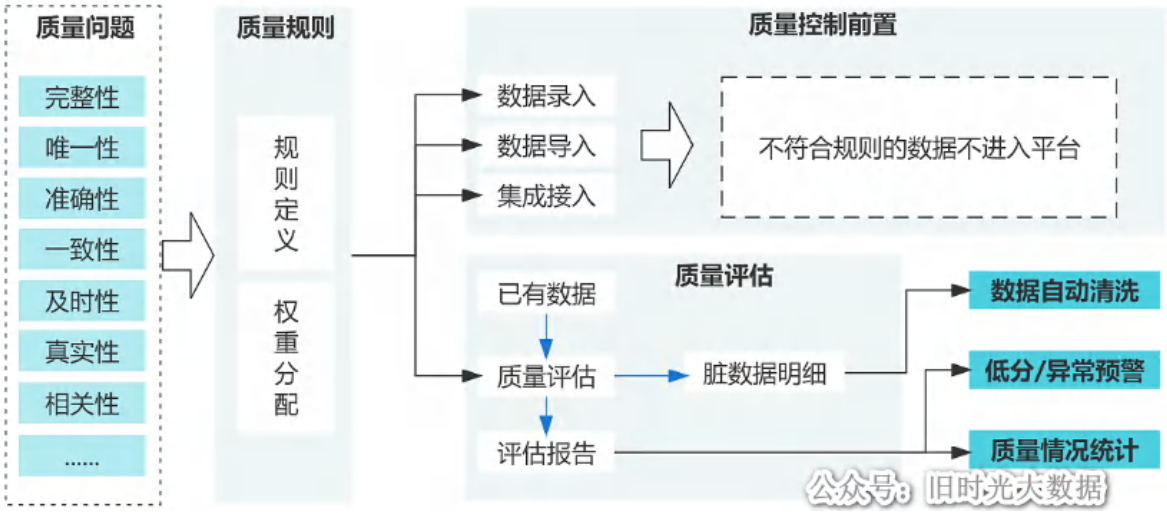
2）**从流程上来说**，整个ETL是多个任务，按步骤顺序执行的一个过程，后置任务依赖前置任务，定期执行，整个流程需要自动化，并且哪个环节出现了问题，给予预警，通知相关维护人员及时处理。

3）**从管理层面上来说**，数据仓库是构建在公司各个业务系统之上，它是一面镜子，很多时候它能反映出业务系统的问题，所以需要管理层的支持和约束，比如通过第一条说的事后自动检验机制反映出业务系统的维护错误，需要相应的业务系统维护人员及时处理。

数据治理流程如下：

如下图，基本流程如下：**发现数据质量问题 > 定义数据质量规则 > 质量控制 > 质量评估 > 质量优化。**

公众号：旧时光大数据



扩展内容：

这里主要是阿里对数仓的一些数据质量保证原则

1、数据质量保障原则

阿里对数据仓库主要从四个方面评估数据质量

1）完整性

确保数据不存在缺失

2）准确性

确保数据不存在异常或错误

3）一致性

体现在从业务仓库加工到数据仓库，再到各个消费节点，必须是同一种类型，长度也需要保持一致。

4）及时性

确保数据能及时产出，越来越多的应用希望数据更快产出。

2、数据质量方法概述

阿里的业务复杂，种类繁多的产品每天产生数以亿计的数据，每天的数据量在PB级以上，而数据消费端的应用又层出不穷，各类数据产品如雨后春笋般出现。为了满足这些数据应用，数据仓库的规模也不断膨胀，同时数据质量的保障也越来越复杂。

基于上述背景，提出一套数据质量建设方法

1）消费场景知晓

主要是通过数据资产和基于元数据的应用链路分析解决消费场景知晓的问题。

2）数据生成加工各个环节卡点效验

在各个数据生成环节，格局不同资产等级做出相应的处理。

3）风险点监控

在线数据

- 主要是针对在线系统日常运行产出的数据进行业务规则的效验，以保证数据质量，其主要使用实时业务检测平台BCP（Biz-Check-Platform）。

离线数据

公众号：旧时光大数据

- 主要是针对离线系统日常运行产出的数据进行数据质量监控和实效性监控，其中数据质量监控主要是使用DQC，实效性监控主要是使用摩萨德。

4）质量衡量

对质量的衡量既有事前的衡量，如DQC覆盖率；又有事后的衡量，主要用于跟进质量问题，确定质量问题原因、责任人、解决情况等，并用于数据质量的复盘，避免类似事件再次发生。根据质量问题对不同等级资产的影响程度，确定其是属于低影响的事件还是具有较大影响的故障。质量分则是综合事前和事后的衡量数据进行打分。

5）质量配套工具

针对数据质量的各个方面，都需要相关的工具进行保证，提高效率。

3、消费场景知晓

在数据快速增长，数据类产品和日常决策支持等系统层出不穷，数据仓库应接不暇，数据工程师很难确定几百PB的数据到底是否都是重要的，是否都要进行保障，是否有一些数据已经过期了，是否所有需要精确的进行保障。

基于上述疑问，阿里内部提出数据资产等级，解决消费场景知晓问题。

1）数据等级定义

毁灭性质

- 即数据一旦出错，将会引起重大资产损失，导致收益大损。记为A1（Asset）

全局性质

- 即数据直接或者间接用于集团级业务和效果的评估、重要平台的运维、对外数据产品的透露、影响用户在阿里系网站的行为。记为A2

局部性质

- 即数据直接或间接用于内部一般数据产品或者运营/产品报告，如果出现问题会给事业部或业务线造成影响或者工作效率降低。记为A3

一般性质

- 即数据主要用于小二的日常数据分析，出现问题不会带来太大的影响。记为A4

未知性质

- 不能明确说出数据的应用场景，则标注为未知。记为A5

2）数据资产等级落地

先给不同数据产品或应用划分数据资产等级，再依托元数据的上下游血缘，可以将整个加工消费链打上该数据资产等级。

总结：解决了消费场景知晓的问题，就知道了数据的重要等级，针对不同的等级，也将采取不同的保障措施。

4、数据加工过程卡点校验

1）在线系统卡点效验

主要是指在线系统 的数据生成过程中进行的卡点效验。

要向数据保障数据准确性，主要使用工具。

- 发布平台

公众号：旧时光大数据

    - 在业务进行重大变更时，订阅这个发布过程，然后给到离线开发人员。当然不会频繁通知离线开发人员，会根据数据资产等级通知相应人员
- 数据库表的变化感知
    - 数据库扩容对表的DDL变化，都需要主动通知离线开发人员。

有了开发工具，开发人员更重要

- 须知哪些是重要的核心数据资产
- 须知哪些只是内部分析数据使用

2）离线系统卡点效验

首先，代码提交时的卡点效验

- 由于开发人员素质不同，代码能力也有差异，所以需要工具代码扫描SQL SCAN，对每次提交代码进行扫描，将风险点提出来。

其次，是任务发布上线时卡点效验

- 发布上线前测试，代码评审和回归测试
    - 回归测试：指修改了旧代码后，重新进行测试以确认修改没引入新的错误。
    - 冒烟测试：指开发人员修复一个bug，测试人员专门针对这个问题测试。
- 发布上线后测试：Dry-Run测试或真实环境运行测试。
    - Dry-Run：不执行代码，仅运行执行计划，避免由于上线下环境不一致导致语法错误。

最后是节点变更或数据重刷前的变更通知

- 一般指使用通知中心将变更原因、变更逻辑、变更测试报告和变更时间等自动通知下游，下游对此次变更没有异议后，再按约定时间执行发布变更，将变更对下游影响降至最低。

5、风险点监控

主要是针对数据在日常运行过程中容易出现的风险进行监控并设置报警机制

1）在线数据风险点监控

采用实时业务检测平台BCP保障重要数据资产的质量。

BCP：制定效验规则用于验证数据的准确性

2）离线数据风险监控

数据准确性

- 使用DQC来监控数据

数据及时性

- 需要进行一系列的报警和优先级设置，使得重要的任务优先且正确产出。使用的工具有摩萨德。

6、质量衡量

评估各种数据仓库质量保障方案，有以下指标：

1）数据质量起夜率，及夜晚处理数据问题

2）数据质量事件

3）数据质量故障体系：指的是严重的数据质量事件

处理方案：

- 故障定义

公众号：旧时光大数据

- 故障等级
- 故障处理
- 故障REview

怎么衡量数仓的数据质量，有哪些指标

问过的一些公司：腾讯社招

参考答案：

1）**数据的准确性**

数据准确性（Accuracy）是**指数据采集值或者观测值和真实值之间的接近程度，也叫做误差值，误差越大，准确度越低。**

指数据中记录的信息和数据是否准确，数据记录的信息是否存在异常或错误。准确性关注的是数据记录中存在的错误，如字符型数据的乱码现象就存在着准确性的问题，还有就是异常的数值：异常大或者异常小的数值、不符合有效性要求的数值等。

2）**数据的精确性**

数据的精确性（Precision）是**指对同一对象的观测数据在重复测量时所得到不同数据间的接近程度。**精确性，也可以叫精准性。精确性与我们数据采集的精度有关系。精度高，要求数据采集的粒度越细，误差的容忍程度越低。

比如测量人的身高，我们可以精确到厘米，多次测量差异只会在厘米级别；测量北京到上海的距离，我们精确到公里，多次测量结果间的差异会
在公里级别:采用游标卡尺测量一个零件的厚度，可以精确到1/50毫米，多次测量的结果间的误差也只会在1/50毫米间。采用的测量方法和手段直接影响着数据的精确性。

3）**数据的真实性**

数据的真实性，也叫数据的正确性(Rightness)。**数据的正确性取决于数据采集过程的可控程度**，可控程度高，可追溯情况好，数据的真实性容易得到保障，而可控程度低或者无法追溯，数据造假后无法追溯，则真实性难以保证。为了提高数据的真实性，采用无人进行过程干涉的智能终端直接采集数据，能够更好地保证所采集数据的真实性，减少人为干预，减少数据造假，从而让数据更加正确地反应客观事物。

4）**数据的及时性**

数据的及时性（In-time）就是指**数据能否在需要的时候得到保证。**

比如月初会对上个月的经营和管理数据进行统计汇总，这些数据能否及时处理完成，财务能否在月度关账后及时核算。**数据的及时性是我们数据分析和挖掘及时性的保障。**如果公司的财务核算复杂，核算速度缓慢，上个月的数据在月中才能统计汇总完成，等需要调整财务策略的时候，已经到了月底了，一个月已经快过完了。特别是公司做大了之后，业务覆盖多个市场，多个国家，数据不能及时汇总，会影响到高层决策的及时程度，数据的及时性与企业数据处理的速度和效率有直接的关系，为了提高数据的及时性，越来越多的公司采用管理信息系统，并在管理信息系统中附加各种自动数据外理功能，能够在数据上传系统之后自动完成绝大部分报表，从而保证数据外理的效率。计算机自动外理中间层数据是提高企业数据处理效率的有效手段。

除了保证数据采集的及时性和数据外理的效率问题外，还需要从制度和流程上保证数据传输的及时性，数据报表完成了，要及时或者在要求的时间范围内发送到指定的部门，或者上传到指定的存储空间。

5）**数据的即时性**

指**数据采集时间节点和数据传输的时间节点**，一个数据在数据源头采集后立即存储，并立即加工呈现，就是即时数据，而经过一段时间之后再传输到信息系统中，则数据即时性就稍差。

公众号：旧时光大数据

比如微博的数据采集，当用户发布了微博，数据立即能够被抓取和加工，会生成即时微博数据报告，并随着时间推移，数据不断变化，我们可以称作是即时采集和处理的。一个生产设备的仪表即时反应着设备的温度、电压、电流、气压等数据，这些数据生成数据流，随时监控设备的运行状况，这个数据可以看作是即时数据。而当设备的即时运行数据存储下来，用来分析设备运行状况与设备寿命的关系，这些数据就成为历史数据。

6）**数据的完整性**

数据的完整性是**从数据采集到的程度来衡量的，是应采集和实际采集到数据之间的比例。**

比如一条信息采集12个数据点，如我们采集员工信息数据的时候，要求填写姓名，出生日期，性别，民族、籍贯，身高、血型、婚姻状况，最高学历，最高学历专业、最高学历毕业院校、最高学历毕业时间等12项信息，而某一员工仅仅填写了部分信息，如只填写了其中的5项，则该员工所填写数据的完整性只有一半。

一个公司数据的完整性体现着这个公司对数据的重视程度。要求采集数据而实际上并未完整采集，只采集了一部分，这就是不完整的，往往是公司对数据采集质量要求不到位导致的。公司要求每个人都填写完整的个人信息表，而有部分员工拒绝填写，公司2000员工，只有1200人填写了完整的个人信息表，则这个数据集就是不完整的。

另外，对干动态数据，还要从时间轴上去衡量数据采集的完整性。比如，我们要求每小时采集一次数据，每天会形成24个数据点，记录为24条数据，但是员工渎职，只记录了20次，那么这个数据集也是不完整的。

7）**数据的全面性**

数据的全面性和完整性不同，完整性衡量的是应采集和实际采集的差异。而**全面性指的是数据采集点的遗漏情况。**

比如说，我们要采集员工行为数据，我们只采集了员工上班打卡和下班打卡的数据，上班时间的员工行为数据并未采集，或者没有找到合适的方法来采集。那么，这个数据集就是不全面的。

比如描述一个产品的包装，仅仅描述了产品包装的正面和背面，没有记录产品包装的侧面，则就是不全面的。我们记录一个客户的交易数据，我们只采集了客户订单中的产品、订单中产品的价格和数量，而没有采集客户送货地址，采购时间，这个数据采集就是不全面的。

比如腾讯OO和微信的用户数据记录了客户交流沟通的数据；阿里和京东的用户数据记录了用户的购买交易数据；百度地图记录了用户出行的数据；大众点评和美团记录了客户餐饮娱乐的数据。对于全面描述一个人的生活的衣食住行各方面，这些公司的数据都是不全面的，而如果把他们的数据整合起来，则会形成更加全面的数据。所以说，数据的全面性是一个相对的概念。过度追求数据的全面性是不现实的。

8）**数据的关联性**

数据的关联性是**指各个数据集之间的关联关系。**

比如员工工资数据和工绩效考核数据是通过员工这个资源关联在一起来的，而且绩效数据直接关系到工资的多少。采购订单数据与生产订单数据之间通过物料的追溯机制进行关联，而生产订单又是由员工完成的，即通过员工作业数据与员工信息数据关联起来。

增量表、全量表和拉链表

问过的一些公司：携程(2021.09)

参考答案：

增量表：记录更新周期内新增的数据，即在原表中数据的基础上新增本周期内产生的新数据；

全量表：记录更新周期内的全量数据，无论数据是否有变化都需要记录；

拉链表：一种数据存储和处理的技术方式，可以记录数据的历史信息，记录数据从开始一直到当前所有变化的信息。

案例详解：

1、增量表

以页面访问数据表为例，假设该表从2020-06-01开始记录数据，按天更新，分区为dt。2020-06-01产生了三条访问数据，如下表：

页面名	页面id	report_time	dt
列表页	aaa	2020-06-01 08:15:37	2020-06-01
商详页	bbb	2020-06-01 09:15:37	2020-06-01
首页	ccc	2020-06-01 12:15:37	2020-06-01

2020-06-02首页和商详页又产生了2条访问数据，该两条即为2020-06-02新增的数据，表更新后，dt分区2020-06-02新增2条数据（标红），此时数据表如下：

页面名	页面id	report_time	dt
列表页	aaa	2020-06-01 08:15:37	2020-06-01
商详页	bbb	2020-06-01 09:15:37	2020-06-01
首页	ccc	2020-06-01 12:15:37	2020-06-01
商详页	bbb	2020-06-02 14:15:37	2020-06-02
首页	ccc	2020-06-02 15:15:37	2020-06-02

以此类推，2020-06-03又产生1条访问数据，表更新后，2020-06-03分区下新增1条数据（标黄），此时数据表如下：

页面名	页面id	report_time	dt
列表页	aaa	2020-06-01 08:15:37	2020-06-01
商详页	bbb	2020-06-01 09:15:37	2020-06-01
首页	ccc	2020-06-01 12:15:37	2020-06-01
商详页	bbb	2020-06-02 14:15:37	2020-06-02
首页	ccc	2020-06-02 15:15:37	2020-06-02
列表页	aaa	2020-06-03 10:15:37	2020-06-03

因此，**增量表每次更新是在原表数据的基础上记录本周期内新增的数据**，如上例，按天更新的流量表，每次更新只新增一天内产生的新数据。注意：每次新产生的数据是以最新分区增加到表中，原先的数据依然存在于表中，如今天是2020-06-03，新增1条数据到表中，dt=2020-06-03，但2020-06-01的数据依然在表中，可以按照dt=2020-06-01进行查询。

2、全量表

以用户表为例，假设该表从2020-06-01开始记录数据，按天更新，分区为dt。

2020-06-01有三个用户注册，数据表如下：

用户名	手机号	性别	creat_time	dt
xiaohong	111	女	2020-06-01 08:15:37	2020-06-01
xiaoming	222	男	2020-06-01 09:15:37	2020-06-01
xiaojun	333	男	2020-06-01 12:15:37	2020-06-01

公众号：旧时光大数据

2020-06-02有一名用户注册，即新增了一名用户（标红），表更新后2020-06-02分区内会记录全量的数据，包括2020-06-01的用户数据（标绿），此时数据表如下：

用户名	手机号	性别	creat_time	dt
xiaohong	111	女	2020-06-01 08:15:37	2020-06-01
xiaoming	222	男	2020-06-01 09:15:37	2020-06-01
xiaojun	333	男	2020-06-01 12:15:37	2020-06-01
xiaohong	111	女	2020-06-01 08:15:37	2020-06-02
xiaoming	222	男	2020-06-01 09:15:37	2020-06-02
xiaojun	333	男	2020-06-01 12:15:37	2020-06-02
Lisa	444	女	2020-06-02 15:15:37	2020-06-02

同理，2020-06-03又有2名用户注册，即新增了2名用户（标蓝），表更新后2020-06-03分区内会记录全量数据，即包含2020-06-02的用户数据（标黄），此时数据表如下：

用户名	手机号	性别	creat_time	dt
xiaohong	111	女	2020-06-01 08:15:37	2020-06-01
xiaoming	222	男	2020-06-01 09:15:37	2020-06-01
xiaojun	333	男	2020-06-01 12:15:37	2020-06-01
xiaohong	111	女	2020-06-01 08:15:37	2020-06-02
xiaoming	222	男	2020-06-01 09:15:37	2020-06-02
xiaojun	333	男	2020-06-01 12:15:37	2020-06-02
Lisa	444	女	2020-06-02 15:15:37	2020-06-02
xiaohong	111	女	2020-06-01 08:15:37	2020-06-03
xiaoming	222	男	2020-06-01 09:15:37	2020-06-03
xiaojun	333	男	2020-06-01 12:15:37	2020-06-03
Lisa	444	女	2020-06-02 15:15:37	2020-06-03
Ella	555	女	2020-06-03 14:15:37	2020-06-03
Tony	666	男	2020-06-03 18:15:37	2020-06-03

https://blog.csdn.net/Jun2613

因此，**全量表每次更新都会记录全量数据，包括原全量数据和本次新增数据，即每个分区内的数据都是截至分区时间的全量总数据。**注意：全量表中每个分区内都是截至分区时间的全量数据，原先分区的数据依然存在于表中，只是每次更新会在最新分区内再更新一遍全量数据。如上例，按照dt=2020-06-03查询出的数据是截至2020-06-03的所有注册用户数据，也可以按照dt=2020-06-02查询截至2020-06-02的所有注册用户数据。

3、拉链表

拉链表中有开始时间（start_time）和结束时间（end_time）两个字段，同时有dt和dp两个分区字段。

- **start_time**：数据的开始时间
- **end_time**：数据有效的截至日期
- dp：一般有ACTIVE（线上）和EXPIRED（过期）两个分区。ACTIVE表示数据当前在线上使用，所以其end_time为4712-12-31（系统能处理的最大时间，表示一个达不到的无限向后延伸的时间，意味着该数据在线上永久有效）；EXPIRED表示数据过期，已变更，为历史状态，其end_time是数据变更时具体的时间。对于部分拉链表dp中还有HISTORY分区，此是由于有些拉链表数据量巨大，造成ACTIVE分区使用困难，因此将一部分业务上不再变更的数据转移到HISTORY分区。
- 数据所在的时间分区，记录数据从ACTIVE转移到EXPIRED的日期，即数据发生变更的时间，大部分与end_time一致；当dp中有HISTORY分区，且数据转移到HISTORY分区时，其dt为数据转移到HISTORY的时间。

以账户数据表为例（表主键为账户id），假设2020-06-01数据表中有3个账户信息，如下表：

公众号：旧时光大数据

账户id	cerat_time(账户创建时间)	余额	start_time	end_time	dp	dt
111	2020-05-20 08:15:37	500	2020-06-01	4712-12-31	ACTIVE	4712-12-31
222	2020-05-22 09:15:37	100	2020-06-01	4712-12-31	ACTIVE	4712-12-31
333	2020-05-25 12:15:37	1500	2020-06-01	4712-12-31	ACTIVE	4712-12-31

2020-06-02账户id为111的用户支出100元，账户余额变为400，则原数据（标黄）的end_time即原数据的结束时间变为2020-06-02，dp变为EXPIRED即数据变为过期历史状态，dt为数据从ACTIVE变为EXPIRED的时间即2020-06-02；而新增的数据即账户id为111的用户余额变为400这条数据如下标红所示。数据表变化如下：

账户id	cerat_time(账户创建时间)	余额	start_time	end_time	dp	dt
111	2020-05-20 08:15:37	500	2020-06-01	2020-06-02	EXPIRED	2020-06-02
111	2020-05-21 08:15:37	400	2020-06-02	4712-12-31	ACTIVE	4712-12-31
222	2020-05-22 09:15:37	100	2020-06-01	4712-12-31	ACTIVE	4712-12-31
333	2020-05-25 12:15:37	1500	2020-06-01	4712-12-31	ACTIVE	4712-12-31

同理，2020-06-03账户id为222的用户支出50元，余额为50元，账户id为333的用户支出1000元，余额为500元，此时数据标变化如下：（标黄为变更前数据，标红为变更后数据）

账户id	cerat_time(账户创建时间)	余额	start_time	end_time	dp	dt
111	2020-05-20 08:15:37	500	2020-06-01	2020-06-02	EXPIRED	2020-06-02
111	2020-05-21 08:15:37	400	2020-06-02	4712-12-31	ACTIVE	4712-12-31
222	2020-05-22 09:15:37	100	2020-06-01	2020-06-03	EXPIRED	2020-06-03
222	2020-05-23 09:15:37	50	2020-06-03	4712-12-31	ACTIVE	4712-12-31
333	2020-05-25 12:15:37	1500	2020-06-01	2020-06-03	EXPIRED	2020-06-03
333	2020-05-26 12:15:37	500	2020-06-03	4712-12-31	ACTIVE	4712-12-31

因此，**拉链表可以记录一条数据从开始到当前的所有历史信息，便于查询历史数据。**如还原2020-06-02的历史快照，使用end_time> ‘2020-06-02’ and start_time<= ‘2020-06-02’ 查询，end_time过滤2020-06-02之前的旧数据，start_time过滤2020-06-02之后的新数据。

综合部分面试题

涉及框架之间的比对之类的，放在这部分

Saprk和Flink的区别

可回答：1）Saprk和Flink的区别；2） Saprk Streaming和Flink Streaming的区别；3）Spark、Flink对比

问过的一些公司：爱奇艺x2，阿里x2，阿里云，蘑菇街x2，360， 字节x3，嘉云，字节社招，字节(2021.08)-(2021.10)，快手，中信信用卡，美团社招，贝壳，触宝，竞技世界，趋势科技，网易，网易严选(2021.09)，百度(2021.09)，蔚来(2021.09)，58(2021.09)

参考答案：

我们从以下几个方面介绍两个框架的主要区别：

1、设计理念方面

Spark的技术理念是使用微批来模拟流的计算，基于Micro-batch，数据流以时间为单位被切分为一个个批次，通过分布式数据集RDD进行批量处理，是一种伪实时。

Flink是基于事件驱动的，是面向流的处理框架，Flink基于每个事件一行一行地流式处理，是真正的流式计算。另外它也可以基于流来模拟批进行计算实现批处理。

2、架构方面

公众号：旧时光大数据

Spark在运行时的主要角色包括：Master、Worker、Driver、Executor。

Flink 在运行时主要包含：Jobmanager、Taskmanager和Slot。

3、流处理方面

Spark基于微批量处理，把流数据看成是一个个小的批处理数据块分别处理，所以延迟性只能做到秒级。

Flink基于每个事件处理，每当有新的数据输入都会立刻处理，是真正的流式计算，支持毫秒级计算。

由于相同的原因，Spark只支持基于时间的窗口操作（处理时间或者事件时间），而Flink支持的窗口操作则非常灵活，不仅支持时间窗口，还支持基于数据本身的窗口（另外还支持基于time、count、session，以及data-driven的窗口操作），开发者可以自由定义想要的窗口操作。

4、任务调度方面

Spark Streaming 支持的时间机制有限，只支持处理时间。使用processing time模拟event time必然会有误差， 如果产生数据堆积的话，误差则更明显。

Flink支持三种时间机制：事件时间，注入时间，处理时间，同时支持 watermark 机制处理迟到的数据，说明Flink在处理乱序大实时数据的时候,更有优势。

5、容错机制方面

Spark Streaming的容错机制是基于RDD的容错机制，会将经常用的RDD或者对宽依赖加Checkpoint。利用Spark Streaming的direct方式与Kafka可以保证数据输入源的，处理过程，输出过程符合exactly once。

Flink 则使用两阶段提交协议来保证exactly once。

6、吞吐量与延迟方面

Spark是基于微批的，而且流水线优化做的很好，所以说他的吞入量是最大的，但是付出了延迟的代价，它的延迟是秒级;

Flink是基于事件的，消息逐条处理，而且他的容错机制很轻量级，所以他能在兼顾高吞吐量的同时又有很低的延迟，它的延迟能够达到毫秒级;

7、迭代计算方面

Spark对机器学习的支持很好，因为可以在内存中缓存中间计算结果来加速机器学习算法的运行。但是大部分机器学习算法其实是一个有环的数据流，在Spark中，却是用无环图来表示。而Flink支持在运行时间中的有环数据流，从而可以更有效的对机器学习算法进行运算。

8、时间机制方面

Spark Streaming 支持的时间机制有限，只支持处理时间。

Flink 支持了流处理程序在时间上的三个定义：处理时间、事件时间、注入时间。同时也支持 watermark 机制来处理滞后数据。

Flink和Spark Streaming处理数据的时候，分别怎么做？各自优势点？

问过的一些公司：安恒信息，招银网络

参考答案：

Spark的技术理念是使用微批来模拟流的计算，基于Micro-batch，数据流以时间为单位被切分为一个个批次，通过分布式数据集RDD进行批量处理，是一种伪实时。

公众号：旧时光大数据

Flink是基于事件驱动的，是面向流的处理框架，Flink基于每个事件一行一行地流式处理，是真正的流式计算。另外它也可以基于流来模拟批进行计算实现批处理。

Spark基于微批量处理，把流数据看成是一个个小的批处理数据块分别处理，所以延迟性只能做到秒级。

Flink基于每个事件处理，每当有新的数据输入都会立刻处理，是真正的流式计算，支持毫秒级计算。

由于相同的原因，Spark只支持基于时间的窗口操作（处理时间或者事件时间），而Flink支持的窗口操作则非常灵活，不仅支持时间窗口，还支持基于数据本身的窗口（另外还支持基于time、count、session，以及data-driven的窗口操作），开发者可以自由定义想要的窗口操作。

为什么你觉得Flink比Spark Streaming好?

问过的一些公司：字节

参考答案：

1、反压机制

Flink 在数据传输过程中使用了分布式阻塞队列，一个阻塞队列中，当队列满了以后发送者会被天然阻塞住，这种阻塞功能相当于给这个阻塞队列提供了反压的能力。

Spark Streaming 为了实现反压这个功能，在原来的架构基础上构造了一个“速率控制器”，这个“速率控制器”会根据几个属性，如任务的结束时间、处理时长、处理消息的条数等计算一个速率。在实现控制数据的接收速率中用到了一个经典的算法，即“PID 算法”。

2、延迟方面

Spark Streaming是秒级别的

Structured Streaming是毫秒级别的

Flink是亚秒级别的

3、状态存储方面

Spark的状态管理目前做的比较简单,只有两个对应的算子（UpdateStateByKey和mapWithState）。

Flink 提供文件、内存、RocksDB 三种状态存储，五种类型的状态，（ValueState，ListState，ReducingState，AggregatingState，FoldingState，MapState）。

4、灵活的窗口

Spark只能根据处理时间窗口批量处理。

Flink可以基于处理时间，数据时间，没有记录等的窗口。

5、实时方面

Flink是真正的实时计算，在状态数据和Checkpoint容错上做的比较好，能够做到exactly once。

Saprk Streaming相比Flink有什么优点

问过的一些公司：

参考答案：

公众号：旧时光大数据

Spark中分布式RDD缓存是一个非常强大的功能，在这一点上比Flink好用很多。比如在实时计算过程中需要一些离线大数据与之关联，Spark相比占有比较大的优势。

Saprk比Flink还有占优的地方就是Spark的executor死了之后不会导致整个job挂掉，而是会创建新的executor再重新执行失败的任务。而Flink的某个task manager死了会导致整个job就失败了，必须设置checkpoint来进行容错。在机器迁移过程中Spark要比Flink方便许多。

Flink和Spark对于批处理的区别?

问过的一些公司:

参考答案:

Flink是流处理，用流的思想做批处理。

Spark是微批处理，用批的思想做流处理。

Spark基于微批量处理，把流数据看成是一个个小的批处理数据块分别处理，所以延迟性只能做到秒级。而Flink基于每个事件处理，每当有新的数据输入都会立刻处理，是真正的流式计算，支持毫秒级计算。由于相同的原因，Spark只支持基于时间的窗口操作（处理时间或者事件时间），而Flink支持的窗口操作则非常灵活，不仅支持时间窗口，还支持基于数据本身的窗口(另外还支持基于time、count、session，以及data-driven的窗口操作)，开发者可以自由定义想要的窗口操作。

Flink+Kafka怎么保证精准一次性消费

问过的一些公司：字节(2021.10)

参考答案:

两阶段提交+checkpoint+barriy+offset回拨

Hive和MySQL不同?

问过的一些公司：美团，猿辅导，途牛

参考答案:

1）查询语言不同：hive是hql语言，mysql是sql语句；

2）数据存储位置不同：hive是把数据存储在hdfs上，而mysql数据是存储在自己的系统中；

3）数据格式：hive数据格式可以用户自定义，mysql有自己的系统定义格式；

4）数据更新：hive不支持数据更新，只可以读，不可以写，而sql支持数据更新；

5）索引：hive没有索引，因此查询数据的时候是通过mapreduce很暴力的把数据都查询一遍，也造成了hive查询数据速度很慢的原因，而mysql有索引；

6）延迟性：hive延迟性高，原因就是上边一点所说的，而mysql延迟性低；

7）数据规模：hive存储的数据量超级大，而mysql只是存储一些少量的业务数据；

8）底层执行原理：hive底层是用的mapreduce，而mysql是excutor执行器；

9）使用环境：mysql使用环境几乎没有限制，hive是基于hadoop的；

公众号：旧时光大数据

10）mysql的handle的数据量较小，而hive的能handle数据量较大；

11）可扩展性：mysql的可扩展性较低，而hive的扩展性较高；

12）mysql可以允许局部数据插入、更新、删除等，而hive不支持局部数据的操作。

Hive与HBase的区别?

可回答：Hive和HBase的存储区别

问过的一些公司：阿里x2，阿里云，美团x2，腾讯，触宝，网易，海康(2021.09)

参考答案：

Hive是基于Hadoop的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供简单的sql查询功能，可以将sql语句转换为MapReduce任务进行运行。

HBase是Hadoop的数据库，一个分布式、可扩展、大数据的存储。

1、两者的特点

Hive帮助熟悉SQL的人运行MapReduce任务。因为它是JDBC兼容的，同时，它也能够和现存的SQL工具整合在一起。运行Hive查询会花费很长时间，因为它会默认遍历表中所有的数据。虽然有这样的缺点，一次遍历的数据量可以通过Hive的分区机制来控制。分区允许在数据集上运行过滤查询，这些数据集存储在不同的文件夹内，查询的时候只遍历指定文件夹(分区)中的数据。这种机制可以用来，例如，只处理在某一个时间范围内的文件，只要这些文件名中包括了时间格式。

HBase通过存储key/value来工作。它支持四种主要的操作：增加或者更新行，查看一个范围内的cell，获取指定的行，删除指定的行、列或者是列的版本。版本信息用来获取历史数据(每一行的历史数据可以被删除，然后通过Hbase compactions就可以释放出空间)。虽然HBase包括表格，但是schema仅仅被表格和列簇所要求，列不需要schema。Hbase的表格包括增加/计数功能。

2、限制

Hive目前不支持更新操作。另外，由于hive在hadoop上运行批量操作，它需要花费很长的时间，通常是几分钟到几个小时才可以获取到查询的结果。Hive必须提供预先定义好的schema将文件和目录映射到列，并且Hive与ACID不兼容。

HBase查询是通过特定的语言来编写的，这种语言需要重新学习。类SQL的功能可以通过Apache Phonenix实现，但这是以必须提供schema为代价的。另外，Hbase也并不是兼容所有的ACID特性，虽然它支持某些特性。最后但不是最重要的–为了运行Hbase，Zookeeper是必须的，zookeeper是一个用来进行分布式协调的服务，这些服务包括配置服务，维护元信息和命名空间服务。

3、应用场景

Hive适合用来对一段时间内的数据进行分析查询，例如，用来计算趋势或者网站的日志。Hive不应该用来进行实时的查询。因为它需要很长时间才可以返回结果。

Hbase非常适合用来进行大数据的实时查询。Facebook用Hbase进行消息和实时的分析。它也可以用来统计Facebook的连接数。

4、小总

Hive和Hbase是两种基于Hadoop的不同技术。Hive是一种类SQL的引擎，并且运行MapReduce任务，Hbase是一种在Hadoop之上的NoSQL的Key/vale数据库。当然，这两种工具是可以同时使用的。就像用Google来搜索，用FaceBook进行社交一样，Hive可以用来进行统计查询，HBase可以用来进行实时查询，数据也可以从Hive写到Hbase，设置再从Hbase写回Hive。

公众号：旧时光大数据

Hive与HDFS的关系与区别?

问过的一些公司：阿里，vivo(2021.06)

参考答案：

HDFS：Hadoop Distributed File System，Hadoop分布式文件系统，主要用来解决海量数据的存储问题。

Hive 是基于 Hadoop 的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供完整的 sql 查询功能，可以将 sql 语句转换为 MapReduce 任务进行运行。可以通过类 SQL 语句快速实现简单的 MapReduce 统计，不必开发专门的 MapReduce 应用，十分适合数据仓库的统计分析。

Hive 是建立在 Hadoop 上的数据仓库基础构架。它提供了一系列的工具，可以用来进行数据提取转化加载，这是一种可以存储、查询和分析存储在 Hadoop 中的大规模数据的机制。Hive 定义了简单的类 SQL 查询语言，称为 HQL，它允许熟悉 SQL 的用户查询数据。同时，这个语言也允许熟悉 MapReduce 开发者的开发自定义的 mapper 和 reducer 来处理内建的 mapper 和 reducer 无法完成的复杂的分析工作。

Spark和Hive的区别

问过的一些公司：字节，快手，百度

参考答案：

1、Hive

Hive是基于Hadoop的数据仓库工具，同时又是查询引擎，Spark SQL只是取代的Hive的查询引擎这一部分，企业可以使用Hive+Spark SQL进行开发。

Hive的主要工作如下：

- 把HQL翻译长map-reduce的代码，并且有可能产生很多mapreduce的job
- 把生产的Mapreduce代码及相关资源打包成jar并发布到Hadoop的集群当中并进行运行

Hive默认情况下用derby存储元数据，所以在生产环境下一般会采用多用户的数据库进行元数据的存储，并可以读写分离和备份，一般使用主节点写，从节点读，一般使用MySQL。

2、Spark

Spark SQL处理一切存储介质和各种格式的数据(可以扩展sparksql来读取更多类型的数据)；

Spark SQL把数据仓库的计算速度推向了新的高度（Tungsten成熟之后会更厉害）；

Spark SQL推出的Dataframe可以让数据仓库直接使用机器学习，图计算等复杂算法；

Hive+Spark SQL+DataFrame使用：

- Hive：负责廉价的数据仓库存储
- Spark Sql：负责高速的计算
- DataFrame：负责复杂的数据挖掘

3、Hive on Spark与Spark Sql的区别

Hive on Spark大体与Spark SQL结构类似，只是SQL解析器不同，但是计算引擎都是Spark。

4、Hive on Mapreduce和Spark SQL使用场景

Hive on Mapreduce场景

- Hive的出现可以让那些精通SQL技能、但是不熟悉MapReduce 、编程能力较弱与不擅长Java语言的用户能够在HDFS大规模数据集上很方便地利用SQL语言查询、汇总、分析数据，毕竟精通SQL语言的人要比精通Java语言的多得多。

公众号：旧时光大数据

- **Hive适合处理离线非实时数据。**

Spark SQL场景

- Spark既可以运行本地local模式，也可以以Standalone、cluster等多种模式运行在Yarn、Mesos上，还可以运行在云端例如EC2。此外，Spark的数据来源非常广泛，可以处理来自HDFS、HBase、Hive、Cassandra、Tachyon上的各种类型的数据。
- **实时性要求或者速度要求较高的场所。**

5、Hive on Mapreduce和Spark SQL性能对比

Spark SQL和Hive on Spark时间差不多，但都比Hive on mapreduce快很多，官方数据认为Spark会被传统mapreduce快10-100倍。

Spark和Hive的联系

问过的一些公司：携程(2021.09)

参考答案：

通常来说，Spark和Hive本质上是没有关系的，两者可以互不依赖。但是在企业实际应用中，经常把二者结合起来使用。Spark和Hive结合和使用的方式，主要有以下三种：

1、Hive on Spark

在这种模式下，数据是以table的形式存储在hive中的，用户处理和分析数据，使用的是Hive语法规范的hql。 但**这些hql，在用户提交执行时，底层会经过解析编译以Spark作业的形式来运行。**（事实上，Hive早期只支持一种底层计算引擎，即MapReduce，后期在Spark因其快速高效占领大量市场后，Hive社区才主动拥抱Spark，通过改造自身代码，支持了Spark作为其底层计算引擎。目前Hive支持了三种底层计算引擎，即mr，Tez和Spark。用户可以通过set hive.execution.engine=mr/tez/spark来指定具体使用哪个底层计算引擎。

2、Spark on Hive

Spark本身只负责数据计算处理，并不负责数据存储。**其计算处理的数据源，可以以插件的形式支持很多种数据源，这其中也包括hive。当我们使用Spark来处理分析存储在Hive中的数据时，这种模式就称为为Spark on Hive。**这种模式下，用户可以使用Spark的java/scala/pyhon/r等api，也可以使用Spark语法规范的sql，甚至也可以使用hive语法规范的hql（之所以也能使用hql，是因为Spark 在推广面世之初，就主动拥抱了hive，通过改造自身代码提供了原生对hql包括hive udf的支持，这也是市场推广策略的一种吧）。

3、Spark+Spark Hive catalog

这是Spark和Hive结合的一种新形势，随着数据湖相关技术的进一步发展，这种模式现在在市场上受到了越来越多用户的青睐。其本质是，**数据以orc/parquet/delta lake等格式存储在分布式文件系统如hdfs或对象存储系统如s3中，然后通过使用Spark计算引擎提供的scala/java/python等api或Spark语法规范的sql来进行处理。**由于在处理分析时针对的对象是table，而table的底层对应的才是hdfs/s3上的文件/对象，所以我们需要维护这种table到文件/对象的映射关系，而Spark自身就提供了Spark Hive catalog来维护这种table到文件/对象的映射关系。注意这里的Spark Hive catalog，其本质是使用了Hive的metasore相关api来读写表到文件/对象的映射关系（以及一起其他的元数据信息）到metasore db如mysql，postgresql等数据库中。（由于Spark编译时可以把Hive metastore api等相关代码一并打包到Spark的二进制安装包中，所以使用这种模式，我们并不需要额外单独安装hive）。

Hive和传统数据库的区别

公众号：旧时光大数据

问过的一些公司：字节

参考答案：

Hive 和关系数据库存储文件的系统不同，Hive 使用的是 hadoop 的 HDFS（hadoop 的分布式文件系统），关系数据库则是服务器本地的文件系统；

Hive 使用的计算模型是 mapreduce，而关系数据库则是自己设计的计算模型；

关系数据库都是为实时查询的业务进行设计的，而 Hive 则是为海量数据做数据挖掘设计的，实时性很差；实时性的区别导致 Hive 的应用场景和关系数据库有很大的不同；

Hive 很容易扩展自己的存储能力和计算能力，这个是继承 hadoop 的，而关系数据库在这个方面要比数据库差很多。

Spark和Hive对比，谁更好，你觉得为什么

问过的一些公司：美团(2021.08)

参考答案：

Spark和Hive谁更好，要看实际使用场景。

Hive：**数据存储和清洗**，处理海量数据，比如一个月、一个季度、一年的数据量，依然可以处理，虽然很慢。

Spark SQL：**数据清洗和流式计算**，上述情况下Spark SQL不支持，无法处理，因为其基于内存，量级过大承受不住，并且性价比不如Hive高。

综合来看：

Hive的强项在于：1）大数据存储；2）通过sql方式进行MapReduce操作，降低大数据使用门槛。

Spark强项在于：1）基于内存操作，速度快；2）流式计算。

运用上大多两者集合，Hive负责数据仓库存储，Spark sql负责计算，并结合DataFrame进行复杂的数据挖掘（包括机器学习、图计算等复杂算法）。

MySQL和HBase的对比（区别）

可回答：HBase和MySQL这些数据库比他的优势在哪里？（说优点）

问过的一些公司：滴滴，京东，美团，陌陌(2021.08)

参考答案：

Mysql：关系型数据库，主要面向OLTP，支持事务，支持二级索引，支持sql，支持主从、Group Replication架构模型（此处以Innodb为例，不涉及别的存储引擎）。

HBase：底层使用HDFS（存储计算分离），支持海量数据读写（尤其是写），支持上亿行、上百万列的，面向列的分布式NoSql数据库。天然分布式（数据分片、故障自恢复），主从架构，不支持事务，不支持二级索引，不支持sql。

1、数据存储方式

1）MySQL

MySQL采用行存储，MySQL行存储的方式比较适合OLTP业务。

公众号：旧时光大数据

MySQL优点：

- 体积小、速度快、总体拥有成本低，开源；
- 支持多种操作系统；
- 是开源数据库，提供的接口支持多种语言连接操作；
- MySQL的核心程序采用完全的多线程编程。线程是轻量级的进程，它可以灵活地为用户提供服务，而不过多的系统资源。用多线程和C语言实现的mysql能很容易充分利用CPU；
- MySql有一个非常灵活而且安全的权限和口令系统。当客户与MySql服务器连接时，他们之间所有的口令传送被加密，而且MySql支持主机认证；
- 支持ODBC for Windows， 支持所有的ODBC 2.5函数和其他许多函数， 可以用Access连接MySql服务器， 使得应用被扩展；
- 支持大型的数据库， 可以方便地支持上千万条记录的数据库。作为一个开放源代码的数据库，可以针对不同的应用进行相应的修改；
- 拥有一个非常快速而且稳定的基于线程的内存分配系统，可以持续使用面不必担心其稳定性；
- MySQL同时提供高度多样性，能够提供很多不同的使用者介面，包括命令行客户端操作，网页浏览器，以及各式各样的程序语言介面，例如C+，Perl，Java，PHP，以及Python。你可以使用事先包装好的客户端，或者干脆自己写一个合适的应用程序。MySQL可用于Unix，Windows，以及OS/2等平台，因此它可以用在个人电脑或者是服务器上。

MySQL缺点：

- 不支持热备份；
- MySQL最大的缺点是其安全系统，主要是复杂而非标准，另外只有到调用mysqladmin来重读用户权限时才发生改变；
- 没有一种存储过程(Stored Procedure)语言，这是对习惯于企业级数据库的程序员的最大限制；
- MySQL的价格随平台和安装方式变化。Linux的MySQL如果由用户自己或系统管理员而不是第三方安装则是免费的，第三方案则必须付许可费。Unix或linux 自行安装 免费 、Unix或Linux 第三方安装 收费。

2）HBase

HBase是面向列的NoSql数据库，列存储的方式比较适合OLAP业务，而HBase采用了列族的方式平衡了OLTP和OLAP，支持水平扩展，如果数据量比较大、对性能要求没有那么高、并且对事务没有要求的话，HBase也是个不错的选择。

HBase优点：

- 列的可以动态增加，并且列为空就不存储数据，节省存储空间
- HBase自动切分数据，使得数据存储自动具有水平scalability
- HBase可以提供高并发读写操作的支持

HBase缺点：

- 不能支持条件查询，只支持按照Row key来查询
- 暂时不能支持Master server的故障切换，当Master宕机后，整个存储系统就会挂掉（HBase虽然本身不能故障切换，但是可以配合ZooKeeper来实现HMaster主备节点的failover）

2、适用场景

Mysql	Hbase
行存储	列式存储
适用于OLTP业务	平衡了OLTP、OLAP业务
单机、可扩展性差	水平扩展
支持事务	不支持事务
强一致性	强一致性，时间线一致性
支持二级索引	不支持二级索引
支持全文索引	不支持全文索引

3、关系型数据库

优点：

- 数据之间有关系，进行数据的增删改查的时候是非常方便的
- 关系型数据库是有事务操作的，保证数据的完整性和一致性

缺点：

- 因为数据和数据是有关系的，底层是运行了大量的算法，大量算法会降低系统的效率，会降低性能
- 面对海量数据的增删改查的时候会显的无能为力
- 海量数据对数据进行维护变得非常的无力

因此，关系型数据库适合处理一般量级的数据

4、非关系型数据库

为了处理海量数据，非关系数据库设计之初就是为了替代关系型数据库的关系

优点：

- 海量数据的增删改查是可以的
- 海量数据的维护和处理非常轻松

缺点：

- 数据和数据没有关系，他们之间就是单独存在的
- 非关系数据库没有关系，没有强大的事务关系，没有保证数据的完整性和安全性

Spark和Hadoop之间的区别

问过的一些公司：小米云平台x2，百度，京东，华为x2，阿里x3，阿里蚂蚁(2021.08)，腾讯x3，字节x4，搜狐，农行开发，招银网络，美团(2021.09)

参考答案：

1、Hadoop

Hadoop是一个由Apache基金会所开发的分布式系统基础架构。 用户可以在不了解分布式底层细节的情况下，开发分布式程序。充分利用集群的威力进行高速运算和存储。 Hadoop实现了一个分布式文件系统（Hadoop Distributed File System ， HDFS）。HDFS 有高容错性的特点，并且设计用来部署在低廉的（low-cost）硬件上；而且它提供高吞吐量（high throughput）来访问应用程序的数据，适合那些有着

公众号：旧时光大数据

超大数据集（large data set）的应用程序。HDFS放宽了（relax）POSIX的要求，可以以流的形式访问（streaming access）文件系统中的数据。

Hadoop的框架最核心的设计就是：HDFS 和 MapReduce。HDFS为海量的数据提供了存储，而MapReduce为海量的数据提供了计算。

2、Spark

Apache Spark 是专为大规模数据处理而设计的快速通用的计算引擎。Spark 是UC Berkeley AMP lab (加州大学伯克利分校的AMP实验室)所开源的类Hadoop MapReduce的通用并行框架，Spark 拥有Hadoop MapReduce所具有的优点；但不同于MapReduce的是——Job中间输出结果可以保存在内存中，从而不再需要读写HDFS，因此Spark能更好地适用于数据挖掘与机器学习等需要迭代的 MapReduce 的算法。

Spark 在某些工作负载方面表现得更加优越，换句话说，Spark 启用了内存分布数据集，除了能够提供交互式查询外，它还可以优化迭代工作负载。

Spark 是在 Scala 语言中实现的，它将 Scala 用作其应用程序框架。与 Hadoop 不同，Spark 和 Scala 能够紧密集成，其中的 Scala 可以像操作本地集合对象一样轻松地操作分布式数据集。 尽管创建 Spark 是为了支持分布式数据集上的迭代作业，但是实际上它是对 Hadoop 的补充，可以在 Hadoop 文件系统中并行运行。通过名为 Mesos 的第三方集群框架可以支持此行为。

3、数据的存储和处理区别

Hadoop实质上更多是一个分布式系统基础架构: 它将巨大的数据集分派到一个由普通计算机组成的集群中的多个节点进行存储，同时还会索引和跟踪这些数据，大幅度提升大数据处理和分析效率。Hadoop可以独立完成数据的存储和处理工作，因为其除了**提供HDFS分布式数据存储功能**，还**提供MapReduce数据处理功能**。

Spark 是一个专门用来对那些分布式存储的大数据进行处理的工具，**没有提供文件管理系统，自身不会进行数据的存储**。它必须和其他的分布式文件系统进行集成才能运作。可以选择Hadoop的HDFS,也可以选择其他平台。

4、处理速度区别

Hadoop是磁盘级计算，计算时需要在磁盘中读取数据；其采用的是MapReduce的逻辑，把数据进行切片计算用这种方式来处理大量的离线数据。

Spark，它会在内存中以接近“实时”的时间完成所有的数据分析。Spark的批处理速度比MapReduce快近10倍，内存中的数据分析速度则快近100倍。比如实时的市场活动，在线产品推荐等需要对流数据进行分析场景就要使用Spark。

5、灾难恢复

Hadoop将每次处理后的数据写入磁盘中，对应对系统错误具有天生优势。

Spark的数据对象存储在弹性分布式数据集(RDD:)中。这些数据对象既可放在内存，也可以放在磁盘，所以RDD也提供完整的灾难恢复功能。

Spark为什么比MapReduce运行快？原因有哪些?

问过的一些公司：字节(2021.10)，蘑菇街，阿里，顺丰，祖龙娱乐，海康，虎牙(2021.09)，腾讯云(2021.10)，端点数据(2021.07)

参考答案：

1、Spark是基于内存进行数据处理的，MapReduce是基于磁盘进行数据处理的

公众号：旧时光大数据

MapReduce的设计：中间结果保存在文件中，提高了可靠性，减少了内存占用，但是牺牲了性能，因为数据没经过一次处理，都要保存到磁盘，然后再读取进行后续处理。
Spark的设计：数据在内存中进行交换，要快一些，所以性能方面比MapReduce要好。

2、Spark中具有DAG有向无环图，DAG有向无环图在此过程中减少了shuffle以及落地磁盘的次数

Spark 计算比 MapReduce 快的根本原因在于 DAG 计算模型。一般而言，DAG 相比MapReduce 在大多数情况下可以减少 shuffle 次数。Spark 的 DAGScheduler 相当于一个改进版的 MapReduce，如果计算不涉及与其他节点进行数据交换，Spark 可以在内存中一次性完成这些操作，也就是中间结果无须落盘，减少了磁盘 IO 的操作。但是，如果计算过程中涉及数据交换，Spark 也是会把 shuffle 的数据写磁盘的！有一个误区，Spark 是基于内存的计算，所以快，这不是主要原因，要对数据做计算，必然得加载到内存，Hadoop 也是如此，只不过 Spark 支持将需要反复用到的数据给 Cache 到内存中，减少数据加载耗时，所以 Spark 跑机器学习算法比较在行（需要对数据进行反复迭代）。Spark 基于磁盘的计算也是比 Hadoop 快。刚刚提到了 Spark 的 DAGScheduler 是个改进版的 MapReduce，所以 Spark天生适合做批处理的任务。Hadoop 的 MapReduce 虽然不如 spark 性能好，但是 HDFS 仍然是业界的大数据存储标准。

3、Spark是粗粒度资源申请，也就是当提交spark application的时候，application会将所有的资源申请完毕，如果申请不到资源就等待，如果申请到资源才执行application，task在执行的时候就不需要自己去申请资源，task执行快，当最后一个task执行完之后task才会被释放。优点是执行速度快，缺点是不能使集群得到充分的利用

MapReduce是细粒度资源申请，当提交application的时候，task执行时，自己申请资源，自己释放资源，task执行完毕之后，资源立即会被释放，task执行的慢，application执行的相对比较慢。优点是集群资源得到充分利用，缺点是application执行的相对比较慢。

Spark是基于内存的，而MapReduce是基于磁盘的迭代。

Spark和MapReduce之间的区别？各自优缺点?

问过的一些公司：字节 x 4，蘑菇街 x 2，美团 x 5，阿里 x 3，阿里(2021.10)x2，滴滴，网易，网易严选(2021.09)，作业帮社招，小米，小米(2021.08)，多益，华为精英计划(2021.07)，触宝(2021.07)，重庆富民银行(2021.09)，蔚来(2021.09)，腾讯(2021.10)，腾讯云(2021.10)

参考答案：

1、性能方面

Spark在内存中处理数据，而MapReduce 是通过map和reduce操作在磁盘中处理数据。因此从这个角度上讲Spark的性能应该是超过MapReduce的。

然而，既然在内存中处理，Spark就需要很大的内存容量。就像一个标准的数据库系统操作一样，Spark每次将处理过程加载到内存之中，然后该操作作为缓存一直保持在内存中直到下一步操作。如果Spark与其它资源需求型服务一同运行在YARN上，又或者数据块太大以至于不能完全读入内存，此时Spark的性能就会有很大的降低。

与此相反，MapReduce会在一个工作完成的时候立即结束该进程，因此它可以很容易的和其它服务共同运行而不会产生明显的性能降低。

当涉及需要重复读取同样的数据进行迭代式计算的时候，Spark有着自身优势。 但是当涉及单次读取、类似ETL（抽取、转换、加载）操作的任务，比如数据转化、数据整合等时，MapReduce绝对是不二之选，因为它就是为此而生的。

小结：当数据大小适于读入内存，尤其是在专用集群上时，Spark表现更好；MapReduce适用于那些数据不能全部读入内存的情况，同时它还可以与其它服务同时运行。

2、使用难度方面

公众号：旧时光大数据

Spark 有着灵活方便的Java，Scala和 Python 的API，同时对已经熟悉 SQL 的技术员工来说， Spark 还适用 Spark SQL（也就是之前被人熟知的 Shark）。多亏了 Spark 提供的简单易用的构造模块，我们可以很容易的编写自定义函数。它甚至还囊括了可以即时反馈的交互式命令模式。

Hadoop MapReduce是用Java编写的，但由于其难于编程而备受诟病。尽管需要一定时间去学习语法，Pig还是在一定程度上简化了这个过程，Hive也为平台提供了SQL的兼容。一些Hadoop工具也可以无需编程直接运行MapReduce任务。Xplenty就是一个基于Hadoop的数据整合服务，而且也不需要进行任何编程和部署。

尽管Hive提供了命令行接口，但MapReduce并没有交互式模式。诸如Impala，Presto和Tez等项目都在尝试希望为Hadoop提供全交互式查询模式。

安装与维护方面，Spark并不绑定在Hadoop上，虽然在Hortonworks（HDP 2.2版）和Cloudera（CDH 5版）的产品中Spark和MapReduce都包含在其分布式系统中。（注：Cloudera，Hortonworks及MapR是Hadoop领域三大知名的初创公司，致力于打造更好的Hadoop企业版应用）。

小结：Spark更易于编程，同时也包含交互式模式；MapReduce不易编程但是现有的很多工具使其更易于使用。

3、成本方面

Spark 集群的内存至少要和需要处理的数据块一样大，因为只有数据块和内存大小合适才能发挥出其最优的性能。所以如果真的需要处理非常大的数据，Hadoop 绝对是合适之选，毕竟硬盘的费用要远远低于内存的费用。

考虑到 Spark 的性能标准，在执行相同的任务的时候，需要的硬件更少而运行速度却更快，因此应该是更合算的，尤其是在云端的时候，此时只需要即用即付。

在技术人员方面，即使 Hadoop 从 2005 年就开始普及，但是 MapReduce 方面的专家仍然存在着短缺。而对于从 2010 年才开始普及的 Spark ，这又意味着什么呢？ 或许投身 Spark 学习的人正在快速增加，但是相比于 Hadoop MapReduce 仍然存在着更大的技术人才的缺口。

进一步讲，现存了大量的 Hadoop 即服务的资料和基于 Hadoop 的服务（比如我们 Xplenty 的数据整合服务），这些都降低对技术人员能力和底层硬件知识的要求。相比之下，几乎没有现有可选的 Spark 服务，仅有的那些也是新产品。

小结：根据基准要求， Spark 更加合算， 尽管人工成本会很高。依靠着更多熟练的技术人员和 Hadoop 即服务的供给， Hadoop MapReduce 可能更便宜。

4、兼容性

Spark既可以单独运行，也可以在 Hadoop YARN 上，或者在预置 Mesos 上以及云端。它支持实现 Hadoop 输入范式的数据源，所以可以整合所有 Hadoop 支持的数据源和文件格式。 根据 Spark 官方教程， 它还可以通过 JDBC 和 ODBC 同 BI（商业智能） 工具一起运行。 Hive 和 Pig 也在逐步实现这样的功能。

小结：Spark和Hadoop MapReduce具有相同的数据类型和数据源的兼容性。

5、数据处理

除了平常的数据处理，Spark 可以做的远不止这点：**它还可以处理图和利用现有的机器学习库**。高性能也使得 Spark 在实时处理上的表现和批处理上的表现一样好。这也催生了一个更好的机遇，那就是用一个平台解决所有问题而不是只能根据任务选取不同的平台，毕竟所有的平台都需要学习和维护。

Hadoop MapReduce 在批处理上表现卓越。如果需要进行实时处理，可以利用另外的平台比如 Storm 或者 Impala，而图处理则可以用 Giraph。MapReduce 过去是用 Mahout 做机器学习的，但其负责人已经将其抛弃转而支持 Spark 和 h2o（机器学习引擎）。

小结：Spark是数据处理的瑞士军刀；Hadoop MapReduce 是批处理的突击刀。

公众号：旧时光大数据

6、处理速度

Hadoop是磁盘级计算，计算时需要在磁盘中读取数据；其采用的是MapReduce的逻辑，把数据进行切片计算用这种方式来处理大量的离线数据.

Spark会在内存中以接近“实时”的时间完成所有的数据分析。Spark的批处理速度比MapReduce快近10倍，内存中的数据分析速度则快近100倍。

比如实时的市场活动，在线产品推荐等需要对流数据进行分析场景就要使用Spark。

Spark相比MapReduce的优点

问过的一些公司：字节

参考答案：

1、Spark 把中间数据放到内存中，迭代运算效率高

MapReduce 中计算结果需要落地，保存到磁盘上，这样势必会影响整体速度，而 Spark 支持 DAG 图的分布式并行计算的编程框架，减少了迭代过程中数据的落地，提高了处理效率。

2、Spark 容错性高

Spark 引进了弹性分布式数据集 RDD (Resilient DistributedDataset) 的抽象，它是分布在一组节点中的只读对象集合，这些集合是弹性的，如果数据集一部分丢失，则可以根据“血统”（即允许基于数据衍生过程）对它们进行重建。另外在RDD 计算时可以通过 CheckPoint 来实现容错。

3、Spark 更加通用

mapreduce 只提供了Map和Reduce两种操作，Spark提供的数据集操作类型有很多，大致分为：Transformations和Actions两大类。Transformations包括Map、Filter、FlatMap、Sample、GroupByKey、ReduceByKey、Union、Join、Cogroup、MapValues、Sort等多种操作类型，同时还提供Count, Actions包括Collect、Reduce、Lookup和Save等操作。

是不是用了Spark就不需要Hadoop了?

问过的一些公司：360

参考答案：

不是，Spark和Hadoop各有其长处。

Hadoop和Spark两者都是大数据框架，但是各自应用场景是不同的。Hadoop是一个分布式数据存储架构，它将巨大的数据集分派到一个由普通计算机组成的集群中的多个节点进行存储，降低了硬件的成本。Spark是那么一个专门用来对那些分布式存储的大数据进行处理的工具，它要借助hdfs的数据存储。

Hadoop提供分布式数据存储功能HDFS，还提供了用于数据处理的MapReduce。MapReduce是可以不依靠Spark数据的处理的。当然Spark也可以不依靠HDFS进行运作，它可以依靠其它的分布式文件系统。但是两者完全可以结合在一起，Hadoop提供分布式集群和分布式文件系统，Spark可以依附在Hadoop的HDFS，代替MapReduce弥补MapReduce计算能力不足的问题。

Spark Streaming和Storm的区别

问过的一些公司：蘑菇街

公众号：旧时光大数据

参考答案：

1、处理模型以及延迟

虽然这两个框架都提供可扩展性(Scalability)和可容错性(Fault Tolerance),但是它们的处理模型从根本上说是不一样的。Storm处理的是每次传入的一个事件，而Spark Streaming是处理某个时间段窗口内的事件流。因此，Storm处理一个事件可以达到亚秒级的延迟，而Spark Streaming则有秒级的延迟。

2、容错和数据保证

在容错数据保证方面的权衡方面，Spark Streaming提供了更好的支持容错状态计算。在Storm中，当每条单独的记录通过系统时必须被跟踪，所以Storm能够至少保证每条记录将被处理一次，但是在从错误中恢复过来时候允许出现重复记录，这意味着可变状态可能不正确地被更新两次。而Spark Streaming只需要在批处理级别对记录进行跟踪处理，因此可以有效地保证每条记录将完全被处理一次，即便一个节点发生故障。虽然Storm的 Trident library库也提供了完全一次处理的功能。但是它依赖于事务更新状态，而这个过程是很慢的，并且通常必须由用户实现。

简而言之,如果你需要亚秒级的延迟，Storm是一个不错的选择，而且没有数据丢失。如果你需要有状态的计算，而且要完全保证每个事件只被处理一次，Spark Streaming则更好。Spark Streaming编程逻辑也可能更容易，因为它类似于批处理程序，特别是在你使用批次(尽管是很小的)时。

3、实现和编程API

Storm主要是由Clojure语言实现，Spark Streaming是由Scala实现。如果你想看看这两个框架是如何实现的或者你想自定义一些东西你就得记住这一点。Storm是由BackType和 Twitter开发，而Spark Streaming是在UC Berkeley开发的。

Storm提供了Java API，同时也支持其他语言的API。 Spark Streaming支持Scala和Java语言(其实也支持Python)。另外Spark Streaming的一个很棒的特性就是它是在Spark框架上运行的。这样你就可以想使用其他批处理代码一样来写Spark Streaming程序，或者是在Spark中交互查询。这就减少了单独编写流批量处理程序和历史数据处理程序。

4、生产支持

Storm已经出现好多年了，而且自从2011年开始就在Twitter内部生产环境中使用，还有其他一些公司。而Spark Streaming是一个新的项目，并且在2013年仅仅被Sharethrough使用(据作者了解)。

Storm是 Hortonworks Hadoop数据平台中流处理的解决方案，而Spark Streaming出现在 MapR的分布式平台和Cloudera的企业数据平台中。除此之外，Databricks是为Spark提供技术支持的公司，包括了Spark Streaming。

5、集群管理集成

尽管两个系统都运行在它们自己的集群上，Storm也能运行在Mesos，而Spark Streaming能运行在YARN和 Mesos上。

Flume和Kafka的区别?

问过的一些公司：华为

参考答案：

它们都是流式数据采集框架。

Flume一般用于日志采集，可以定制很多数据源，减少开发量，基本架构是一个Flume进程agent（source、拦截器、选择器、channel<Memory Channel、File Channel>、sink），其中传递的是原子性的event数据。

公众号：旧时光大数据

Kafka一般用于日志缓存，是一个可持久的分布式消息队列，自带存储，提供push和pull两种存储数据功能；包括producer、kafka Cluster（broker：topic、partition）、consumer，依赖于Zookeeper（brokerid、topic、partition元数据存在ZNode，partition选举leader依赖Zookeeper）。

1、Flume和Kafka的侧重点不同

Flume追求的是数据和数据源、数据流向的多样性，适合多个生产者的场景；Flume有自己内置的多种source和sink组件，具体操作方式是编写source、channel和sink的.conf配置文件，开启flume组件的时候用命令关联读取配置文件实现。

Kafka追求的是高吞吐，高负载，同一topic下可以有多个partition，由于是pull模式拉取数据，因此适合多个消费者的场景；kafka没有内置的producer和consumer组件，需要自己编写代码。

2、Flume和Kafka的定位有所不同

Flume是cloudera公司研发，适合多个生产者；适合**下游数据消费者不多**的情况（一个消费者开一个channel）；适合**数据安全性要求不高**的操作（数据没有备份、没有副本）；适合与**Hadoop生态圈对接**的操作（HDFS、Hbase等）；适合生产和收集数据。

Kafka是linkedin公司研发，适合多个消费者；适合**数据下游消费众多**的情况（kafka从磁盘读，并且只找Leader读）；适合**数据安全性要求较高**的操作，支持replication（多副本）；适合消费数据。

Flume是用于将数据发送到HDFS的专用工具。

Kafka可以支持多个应用程序的数据流，而Flume专门用于Hadoop和大数据分析。

Kafka可以处理和监视分布式系统中的数据，而Flume则从分布式系统中收集数据以将数据存储在集中式数据存储中。

为什么使用Flume+Kafka?

可回答：为什么Flume+Kafka是经典组合?

问过的一些公司：华为，阿里

这个我们就要从业务需求考虑，一般使用Flume+Kafka架构都是希望完成实时流式的日志处理，后面再连接上Flink/Storm/Spark Streaming等流式实时处理技术，从而完成日志实时解析的目标。

生产环境中，往往是读取日志进行分析，而这往往是多数据源的，如果Kafka构建多个生产者使用文件流的方式向主题写入数据再供消费者消费的话，无疑非常的不方便。

如果Flume直接对接实时计算框架，当数据采集速度大于数据处理速度，很容易发生数据堆积或者数据丢失，而kafka可以当做一个消息缓存队列，从广义上理解，把它当做一个数据库，可以存放一段时间的数据。

Kafka属于中间件，一个明显的优势就是使各层解耦，使得出错时不会干扰其他组件。

因此数据从数据源到Flume再到Kafka时，数据一方面可以同步到HDFS做离线计算，另一方面可以做实时计算，可实现数据多分发。

Flume和Kafka是怎么配置的

问过的一些公司：海康(2021.09)

参考答案：

1、为什么要集成Flume和Kafka

公众号：旧时光大数据

一般使用Flume + Kafka来完成实时流式的日志处理，后面再连接上Spark Streaming等流式实时处理技术，从而完成日志实时解析的目标。如果Flume直接对接实时计算框架，当数据采集速度大于数据处理速度，很容易发生数据堆积或者数据丢失，而Kafka可以当做一个消息缓存队列，当数据从数据源到Flume再到Kafka时，数据一方面可以同步到HDFS做离线计算，另一方面可以做实时计算，可实现数据多分发。

2、Flume对接Kafka配置

1）Kafka作为Source端

```
a1.sources = r1
a1.channels = c1
a1.sinks = k1

# 配置Kafka作为Source端
a1.sources.r1.type = org.apache.flume.source.kafka.KafkaSource
a1.sources.r1.batchSize = 5000
a1.sources.r1.batchDurationMillis = 2000
a1.sources.r1.kafka.bootstrap.servers = centos1:9092
a1.sources.r1.kafka.topics = mytopic
a1.sources.r1.kafka.consumer.group.id = group1
a1.sources.r1.channels=c1

a1.channels.c1.type=memory
a1.channels.c1.capacity=1000
a1.channels.c1.transactionCapacity=100

a1.sinks.k1.type=logger
a1.sinks.k1.channel=c1
```

2）Kafka作为Sink端

```
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# netcat 监听端口
a1.sources.r1.type = netcat
a1.sources.r1.bind =master1
a1.sources.r1.port = 10000
a1.sources.r1.channels = c1
# 一行的最大字节数
a1.sources.r1.max-line-length = 1024000

# channels具体配置
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

# 配置Kafka作为Sink端
a1.sinks.k1.channel = c1
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.topic = hello
a1.sinks.k1.brokerList = master1:9092,slave1:9092,slave2:9092
a1.sinks.k1.requiredAcks = 1
a1.sinks.k1.batchSize = 20
```

公众号：旧时光大数据

Spark Streaming与Kafka集成，如何保证Exactly Once语义

问过的一些公司：快手

参考答案：

在Kafka、Storm、Flink、Spark Streaming等分布式流处理系统中（Kafka本质上是流处理系统，不是单纯的“消息队列”），存在三种消息传递语义（message delivery semantics），分别是：

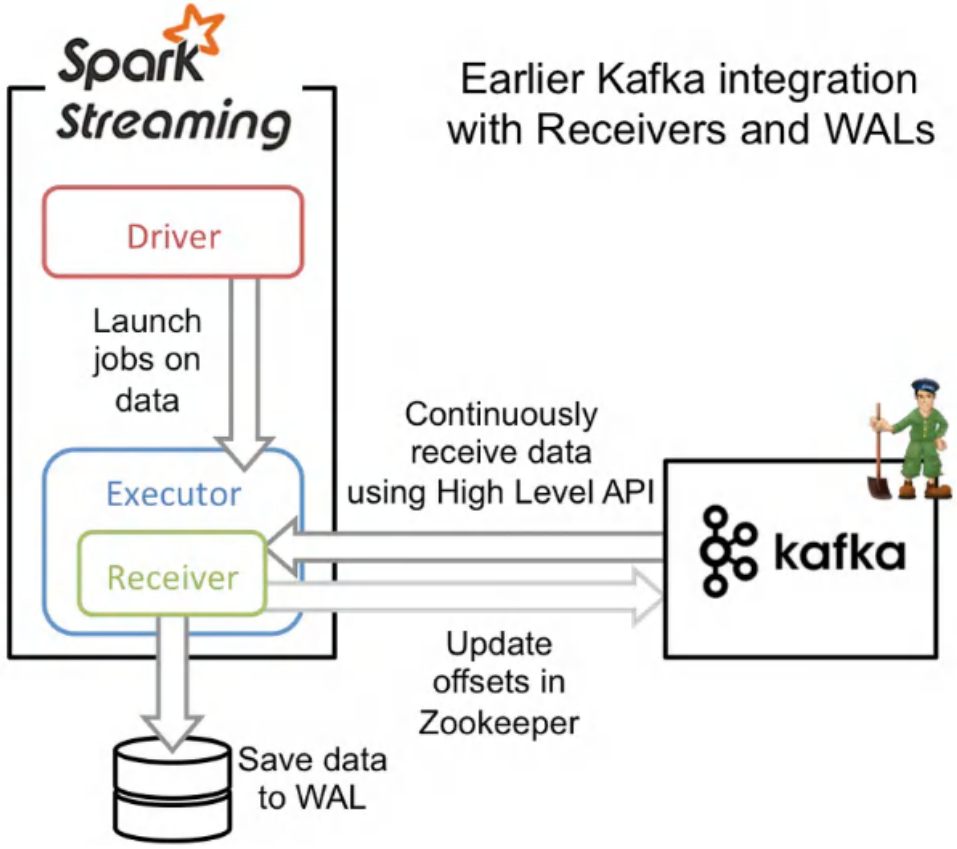
at least once：每条消息会被收到1次或多次。例如发送方S在超时时间内没有收到接收方R的通知（如ack），或者收到了R的报错，就会不断重发消息直至R传回ack。

at most once：每条消息会被收到0次或1次。也就是说S只负责向R发送消息，R也没有任何通知机制。无论R最终是否收到，S都不会重发。

exactly once：是上面两个的综合，保证S发送的每一条消息，R都会“不重不漏”地恰好收到1次。它是最强最精确的语义，也最难实现。

如上图所示，一个Spark Streaming程序由三步组成：**输入、处理逻辑、输出**。要达到exactly once的理想状态，需要三步协同进行，而不是只与处理逻辑有关。Kafka与Spark Streaming集成时有两种方法：旧的基于receiver的方法，新的基于direct stream的方法。

1、基于receiver的方法

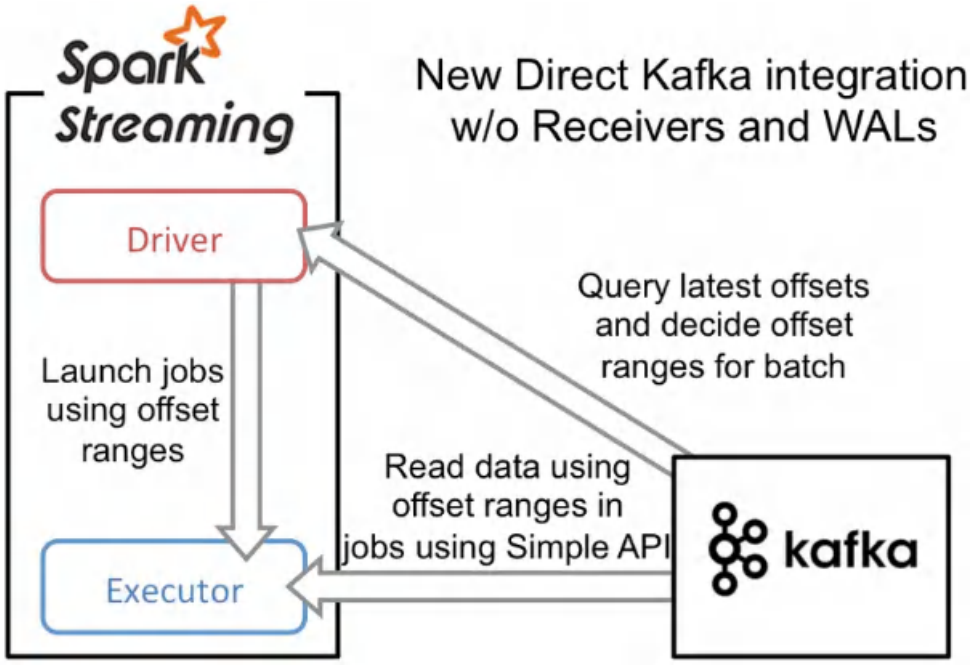


公众号：旧时光大数据

基于receiver的方法采用Kafka的高级消费者API，每个executor进程都不断拉取消息，并同时保存在executor内存与HDFS上的预写日志（write-ahead log/WAL）。当消息写入WAL后，自动更新ZooKeeper中的offset。

它可以保证at least once语义，但无法保证exactly once语义。虽然引入了WAL来确保消息不会丢失，但还有可能会出现消息已经写入WAL，但offset更新失败的情况，Kafka就会按上一次的offset重新发送消息。这种方式还会造成数据冗余（Kafka broker中一份，Spark executor中一份），使吞吐量和内存利用率降低。

2、基于direct stream的方法



基于direct stream的方法采用Kafka的简单消费者API，它的流程大大简化了。executor不再从Kafka中连续读取消息，也消除了receiver和WAL。还有一个改进就是Kafka分区与RDD分区是一一对应的，更可控。

driver进程只需要每次从Kafka获得批次消息的offset range，然后executor进程根据offset range去读取该批次对应的消息即可。由于offset在Kafka中能唯一确定一条消息，且在外部只能被Streaming程序本身感知到，因此消除了不一致性，达到了exactly once。

不过，由于它采用了简单消费者API，我们就需要自己来管理offset。否则一旦程序崩溃，整个流只能从earliest或者latest点恢复，这肯定是不稳妥的。

Kafka作为输入源可以保证exactly once，那么**处理逻辑**呢？答案是显然的，**Spark Streaming的处理逻辑天生具备exactly once语义。**

Spark RDD之所以被称为“弹性分布式数据集”，是因为它具有**不可变、可分区、可并行计算、容错**的特征。一个RDD只能由稳定的数据集生成，或者从其他RDD转换（transform）得来。如果在执行RDD lineage的过程中失败，那么只要源数据不发生变化，无论重新执行多少次lineage，都一定会得到同样的、确定的结果。

最后，我们**还需要保证输出过程也符合exactly once语义。**Spark Streaming的输出一般是靠foreachRDD()算子来实现，它默认是at least once的。如果输出过程中途出错，那么就会重复执行直到写入成功。为了让它符合exactly once，可以施加两种限制之一：**幂等性写入**（idempotent write）、**事务性写入**（transactional write）。

幂等性写入

幂等原来是数学里的概念，即f(f(x))=f(x)。幂等写入就是写入多次与写入一次的结果完全相同，可以自动将at least once转化为exactly once。这对于自带主键或主键组的业务比较合适（比如各类日志、MySQL binlog等），并且实现起来比较简单。

公众号：旧时光大数据

但是它要求处理逻辑是map-only的，也就是只能包含转换、过滤等操作，不能包含shuffle、聚合等操作。如果条件更严格，就只能采用事务性写入方法。

```
stream.foreachRDD { rdd =>
  rdd.foreachPartition { iter =>
    // make sure connection pool is set up on the executor before writing
    SetupJdbc(jdbcDriver, jdbcUrl, jdbcUser, jdbcPassword)

    iter.foreach { case (key, msg) =>
      DB.autoCommit { implicit session =>
        // the unique key for idempotency is just the text of the message itself, for example purposes
        sql"insert into idem_data(msg) values (${msg})".update.apply
      }
    }
  }
}
```

事务性写入

这里的事务与DBMS中的事务含义基本相同，就是对数据进行一系列访问与更新操作所组成的逻辑块。为了符合事务的**ACID特性**，必须引入一个唯一ID标识当前的处理逻辑，并且将计算结果与该ID一起落盘。ID可以由主题、分区、时间、offset等共同组成。

事务操作可以在foreachRDD()时进行。如果数据写入失败，或者offset写入与当前offset range不匹配，那么这一批次数据都将失败并且回滚。

```
// localTx is transactional, if metric update or offset update fails, neither will be committed
DB.localTx { implicit session =>
  // store metric data
  val metricRows = sql"""
update txn_data set metric = metric + ${metric}
  where topic = ${osr.topic}
""".update.apply()
  if (metricRows != 1) {
    throw new Exception("...")
  }

  // store offsets
  val offsetRows = sql"""
update txn_offsets set off = ${osr.untilOffset}
  where topic = ${osr.topic} and part = ${osr.partition} and off = ${osr.fromOffset}
""".update.apply()
  if (offsetRows != 1) {
    throw new Exception("...")
  }
}
```

Spark Shuffle和MapReduce Shuffle的区别？说下各自的特点和过程

可回答：Hive shuffle和Spark shuffle的区别（Hive任务本质是将SQL优化成MapReduce任务进行处理，因此Shuffle过程就是MapReduce任务的Shuffle过程，而Shuffle的过程主要是将数据从Map Task端拉往Reduce Task端。）

公众号：旧时光大数据

问过的一些公司：顺丰，阿里，阿里(2021.09)，头条，米哈游，京东，小米，字节(2021.07)-(2021.08)，欢聚(2021.09)

参考答案：

1、Shuffle简介

Shuffle的本意是洗牌、混洗的意思，把一组有规则的数据尽量打乱成无规则的数据。而在MapReduce中，Shuffle更像是洗牌的逆过程，指的是将map端的无规则输出按指定的规则“打乱”成具有一定规则的数据，以便reduce端接收处理。其在MapReduce中所处的工作阶段是map输出后到reduce接收前，具体可以分为map端和reduce端前后两个部分。

在shuffle之前，也就是在map阶段，MapReduce会对要处理的数据进行分片（split）操作，为每一个分片分配一个MapTask任务。接下来map会对每一个分片中的每一行数据进行处理得到键值对（key,value）此时得到的键值对又叫做“中间结果”。此后便进入reduce阶段，由此可以看出Shuffle阶段的作用是处理“中间结果”。

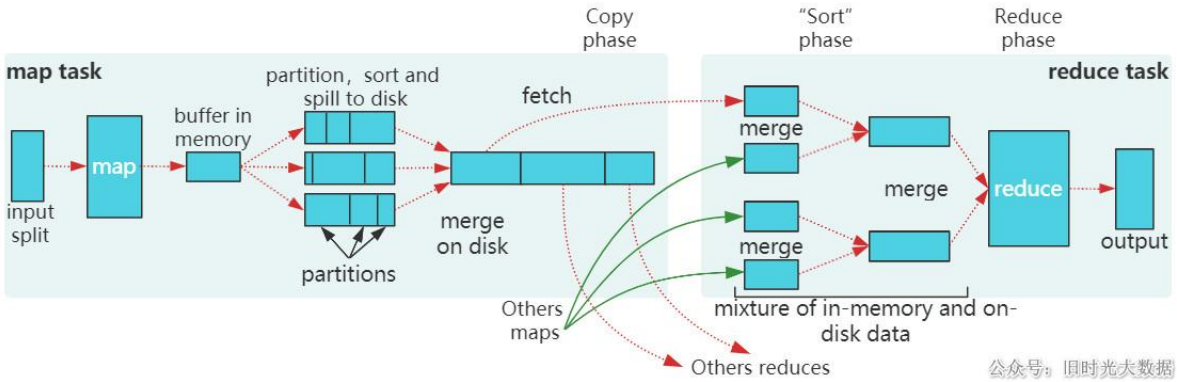
由于Shuffle涉及到了磁盘的读写和网络的传输，因此Shuffle性能的高低直接影响到了整个程序的运行效率。

2、MapReduce Shuffle

Hadoop的核心思想是MapReduce，但shuffle又是MapReduce的核心。shuffle的主要工作是从Map结束到Reduce开始之间的过程。shuffle阶段又可以分为Map端的shuffle和Reduce端的shuffle。

1）Map端的Shuffle

下图是MapReduce Shuffle的官方流程：



因为频繁的磁盘I/O操作会严重的降低效率，因此“中间结果”不会立马写入磁盘，而是优先存储到map节点的“环形内存缓冲区”，在写入的过程中进行分区（partition），也就是对于每个键值对来说，都增加了一个partition属性值，然后连同键值对一起序列化成字节数组写入到缓冲区（缓冲区采用的就是字节数组，默认大小为100M）。

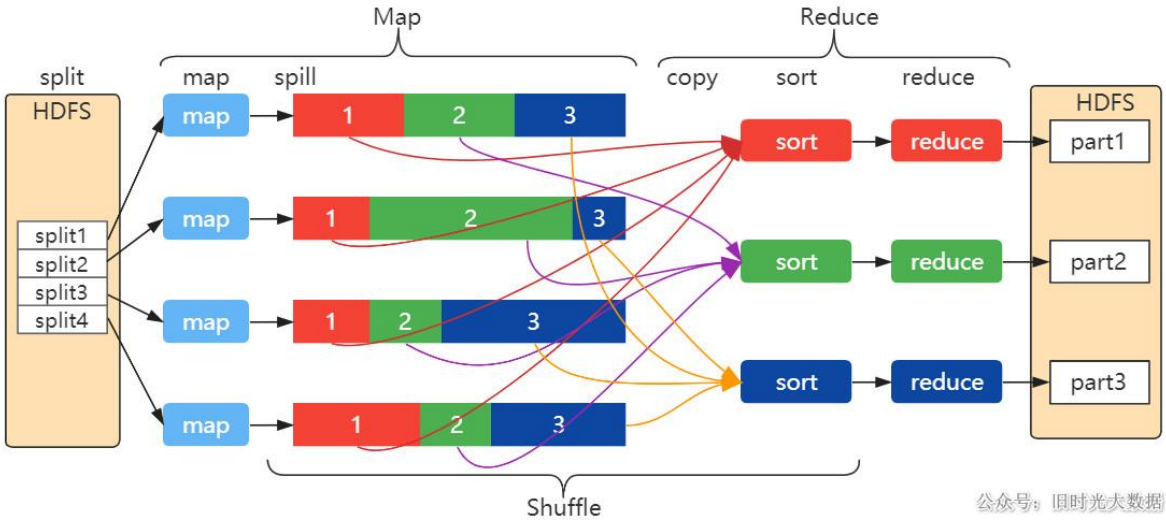
当写入的数据量达到预先设置的阈值后便会启动溢写出线程将缓冲区中的那部分数据溢出写（spill）到磁盘的临时文件中，并在写入前根据key进行排序（sort）和合并（combine，可选操作）。

溢出写过程按轮询方式将缓冲区中的内容写到mapreduce.cluster.local.dir属性指定的本地目录中。当整个map任务完成溢出写后，会对磁盘中这个map任务产生的所有临时文件（spill文件）进行归并（merge）操作生成最终的正式输出文件，此时的归并是将所有spill文件中的相同partition合并到一起，并对各个partition中的数据再进行一次排序（sort），生成key和对应的value-list，文件归并时，如果溢写文件数量超过参数min.num.spills.for.combine的值（默认为3）时，可以再次进行合并。

至此map端的工作已经全部结束，最终生成的文件也会存储在TaskTracker能够访问的位置。每个reduce task不间断的通过RPC从JobTracker那里获取map task是否完成的信息，如果得到的信息是map task已经完成，那么Shuffle的后半段开始启动。

2）Reduce端的shuffle

公众号：旧时光大数据



每个reduce task负责处理一个分区的文件，以下是reduce task的处理流程：

- reduce task从每个map task的结果文件中拉取对应分区的数据。因为数据在map阶段已经是分好区了，并且会有一个额外的索引文件记录每个分区的起始偏移量。所以reduce task取数的时候直接根据偏移量去拉取数据就ok。
- reduce task从每个map task拉取分区数据的时候会进行再次合并，排序，按照自定义的reducer的逻辑代码去处理。
- 最后就是Reduce过程了，在这个过程中产生了最终的输出结果，并将其写到HDFS上。

3）为什么要排序

key存在combine操作，排序之后相同的key放到一块显然方便做合并操作。

reduce task是按key去处理数据的。 如果没有排序那必须从所有数据中把当前相同key的所有value数据拿出来，然后进行reduce逻辑处理。显然每个key到这个逻辑都需要做一次全量数据扫描，影响性能，有了排序很方便的得到一个key对于的value集合。

reduce task按key去处理数据时，如果key按顺序排序，那么reduce task就按key顺序去读取，显然当读到的key是文件末尾的key那么就标志数据处理完毕。如果没有排序那还得有其他逻辑来记录哪些key处理完了，哪些key没有处理完。

虽有千万种理由需要这么做，但是很耗资源，并且像排序其实我们有些业务并不需要排序。

4）为什么要文件合并

因为内存放不下就会溢写文件，就会发生多次溢写，形成很多小文件，如果不合并，显然会小文件泛滥，集群需要资源开销去管理这些小文件数据。

任务去读取文件的数增多，打开的文件句柄数也会增多。

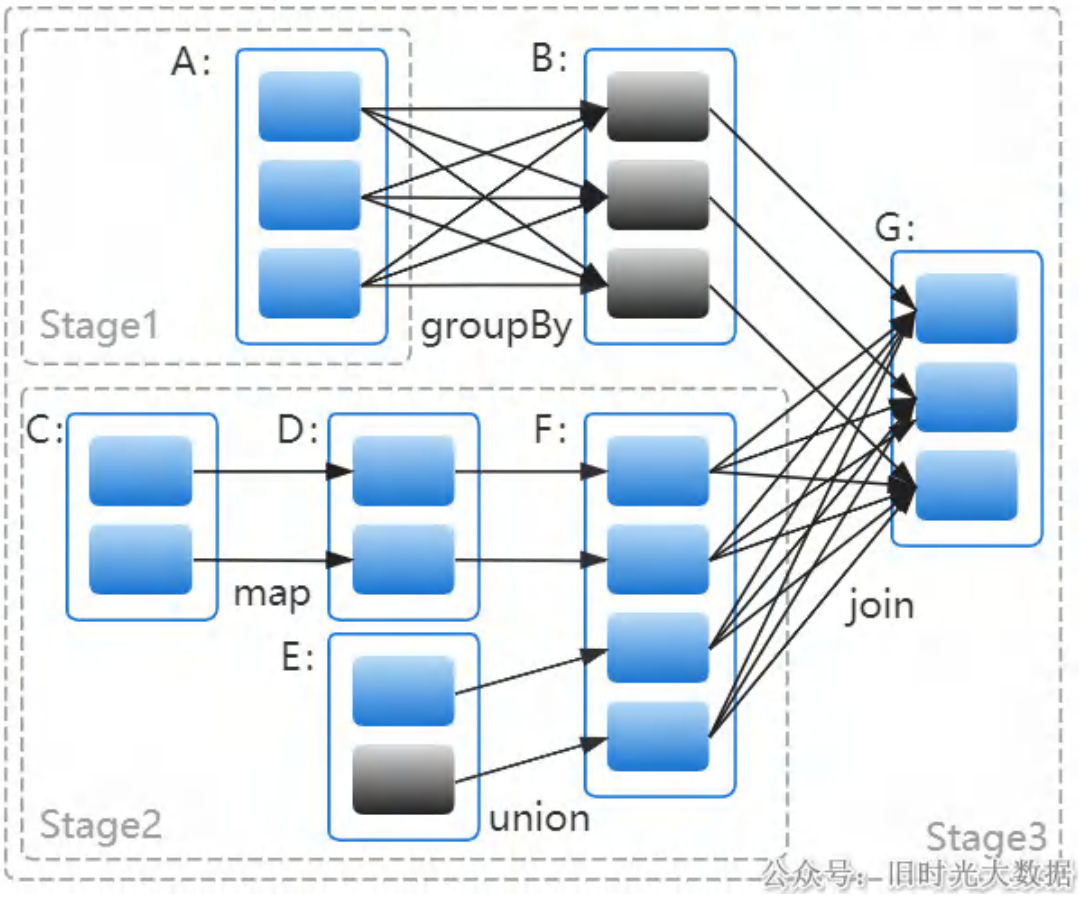
mapreduce是全局有序。单个文件有序，不代表全局有序，只有把小文件合并一起排序才会全局有序。

3、Spark Shuffle

Spark的Shuffle是在MapReduce Shuffle基础上进行的调优。其实就是对排序、合并逻辑做了一些优化。在Spark中Shuffle write相当于MapReduce 的map，Shuffle read相当于MapReduce 的reduce。

Spark丰富了任务类型，有些任务之间数据流转不需要通过Shuffle，但是有些任务之间还是需要通过Shuffle来传递数据，比如宽依赖的group by key以及各种by key算子。宽依赖之间会划分stage，而Stage之间就是Shuffle，如下图中的stage0，stage1和stage3之间就会产生Shuffle。

公众号：旧时光大数据



在Spark的中，负责shuffle过程的执行、计算和处理的组件主要就是ShuffleManager，也即shuffle管理器。ShuffleManager随着Spark的发展有两种实现的方式，分别为HashShuffleManager和SortShuffleManager，因此spark的Shuffle有Hash Shuffle和Sort Shuffle两种。

在Spark的版本的发展，ShuffleManager在不断迭代，变得越来越先进。

在Spark 1.2以前，默认的shuffle计算引擎是HashShuffleManager。该ShuffleManager而HashShuffleManager有着一个非常严重的弊端，就是会产生大量的中间磁盘文件，进而由大量的磁盘IO操作影响了性能。因此在Spark 1.2以后的版本中，默认的ShuffleManager改成了SortShuffleManager。

SortShuffleManager相较于HashShuffleManager来说，有了一定的改进。主要就在于，每个Task在进行shuffle操作时，虽然也会产生较多的临时磁盘文件，但是最后会将所有的临时文件合并(merge)成一个磁盘文件，因此每个Task就只有一个磁盘文件。在下一个stage的shuffle read task拉取自己的数据时，只要根据索引读取每个磁盘文件中的部分数据即可。

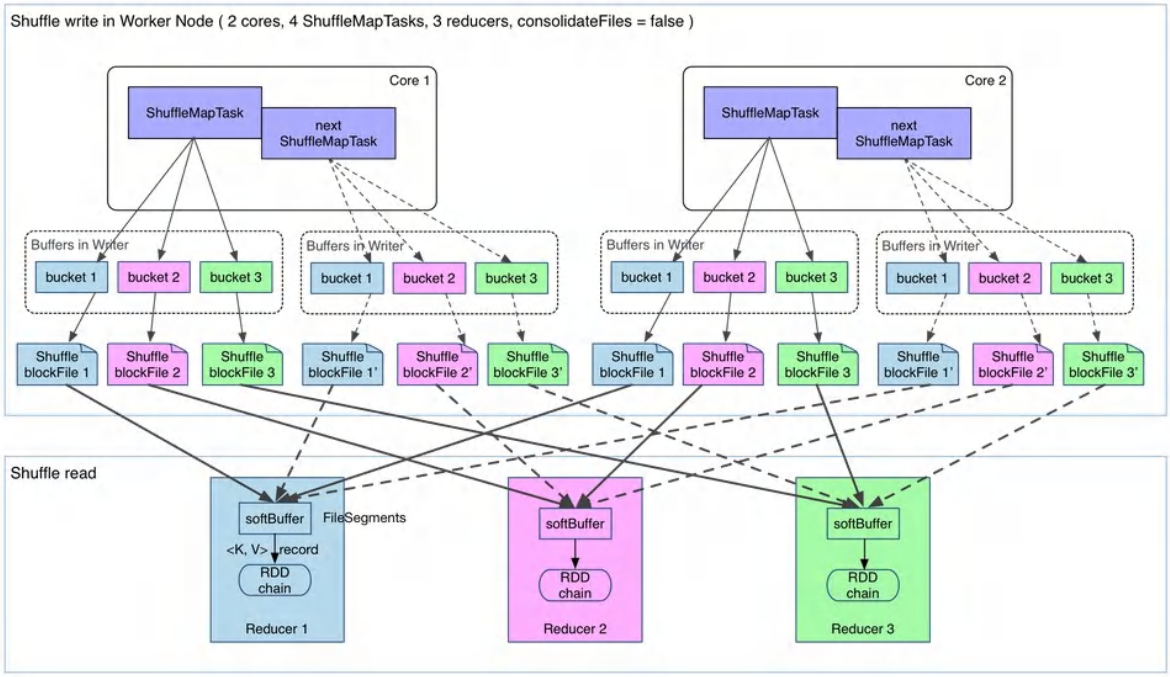
1）Hash Shuffle

HashShuffleManager的运行机制主要分成两种，一种是普通运行机制，另一种是合并的运行机制。合并机制主要是通过复用buffer来优化Shuffle过程中产生的小文件的数量。Hash shuffle是不具有排序的Shuffle。

普通机制的Hash Shuffle

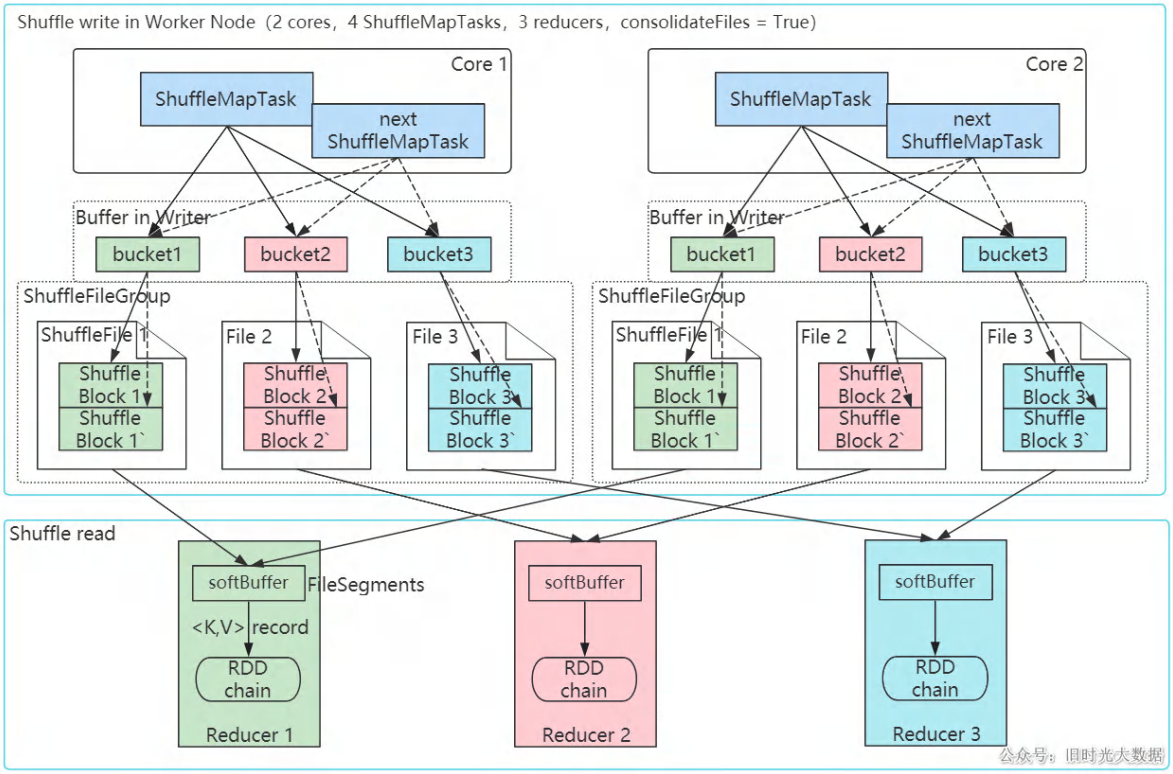
最开始使用的Hash Based Shuffle，每个Mapper会根据Reducer的数量创建对应的bucket，bucket的数量是M * R，M是map的数量，R是Reduce的数量。如下图所示：2个core 4个map task 3 个reduce task，会产生4*3=12个小文件。

公众号：旧时光大数据



优化后的Hash Shuffle

普通机制Hash Shuffle会产生大量的小文件(M * R），对文件系统的压力也很大，也不利于IO的吞吐量，后来做了优化（设置spark.shuffle.consolidateFiles=true开启，默认false），把在同一个core上的多个Mapper输出到同一个文件，这样文件数就变成core * R 个了。如下图所示：2个core 4个map task 3 个reduce task，会产生2*3=6个小文件。



Hash shuffle合并机制的问题：如果 Reducer 端的并行任务或者是数据分片过多的话则 Core * Reducer Task 依旧过大，也会产生很多小文件。进而引出了更优化的sort shuffle。在Spark 1.2以后的版本中，默认的ShuffleManager改成了SortShuffleManager。

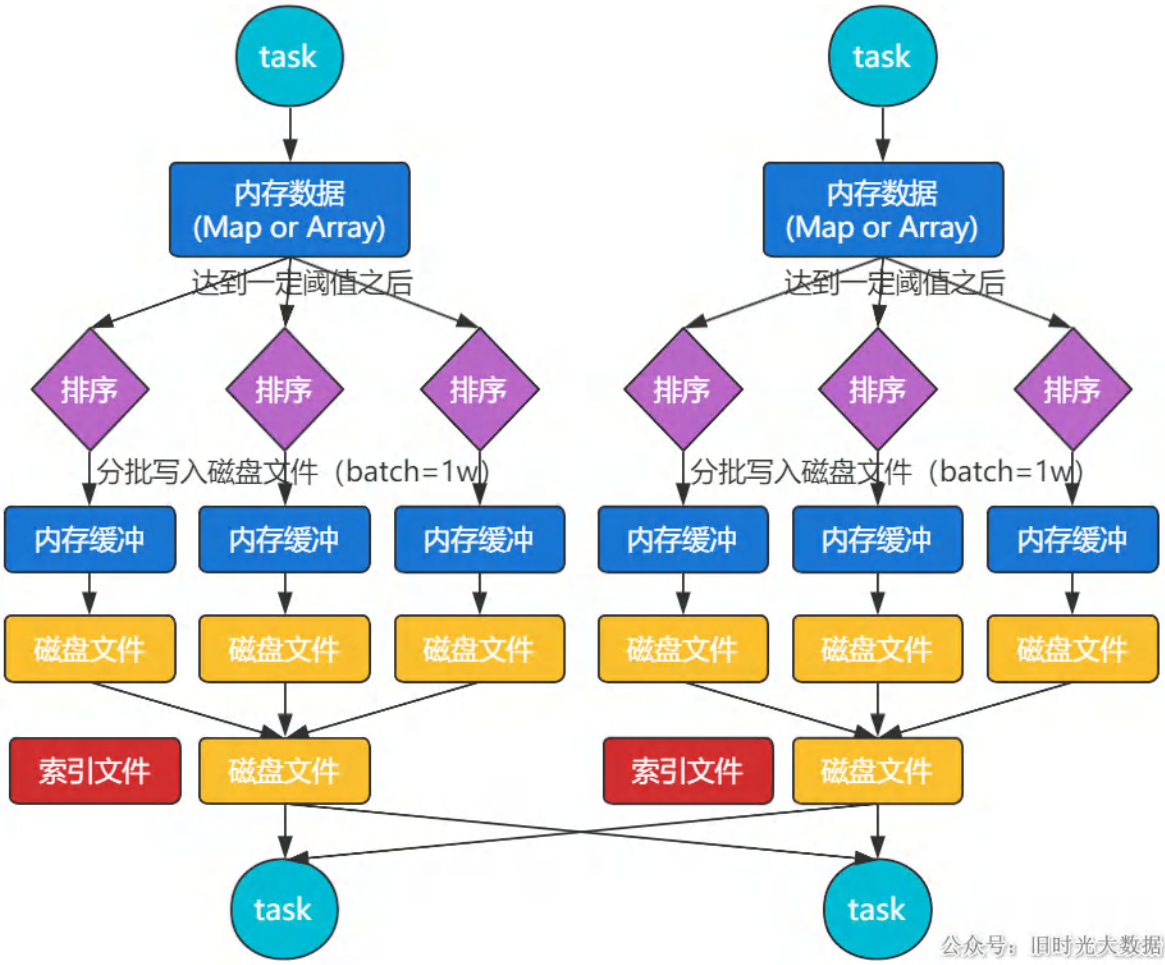
2）Sort Shuffle

SortShuffleManager的运行机制主要分成两种，一种是普通运行机制，另一种是bypass运行机制。当shuffle read task的数量小于等于spark.shuffle.sort.bypassMergeThreshold参数的值时(默认为200)，就会启用bypass机制。

公众号：旧时光大数据

普通机制的Sort Shuffle

这种机制和mapreduce差不多，在该模式下，数据会先写入一个内存数据结构中，此时根据不同的shuffle算子，可能选用不同的数据结构。如果是reduceByKey这种聚合类的shuffle算子，那么会选用Map数据结构，一边通过Map进行聚合，一边写入内存；如果是join这种普通的shuffle算子，那么会选用Array数据结构，直接写入内存。接着，每写一条数据进入内存数据结构之后，就会判断一下，是否达到了某个临界阈值。如果达到临界阈值的话，那么就会尝试将内存数据结构中的数据溢写到磁盘，然后清空内存数据结构。



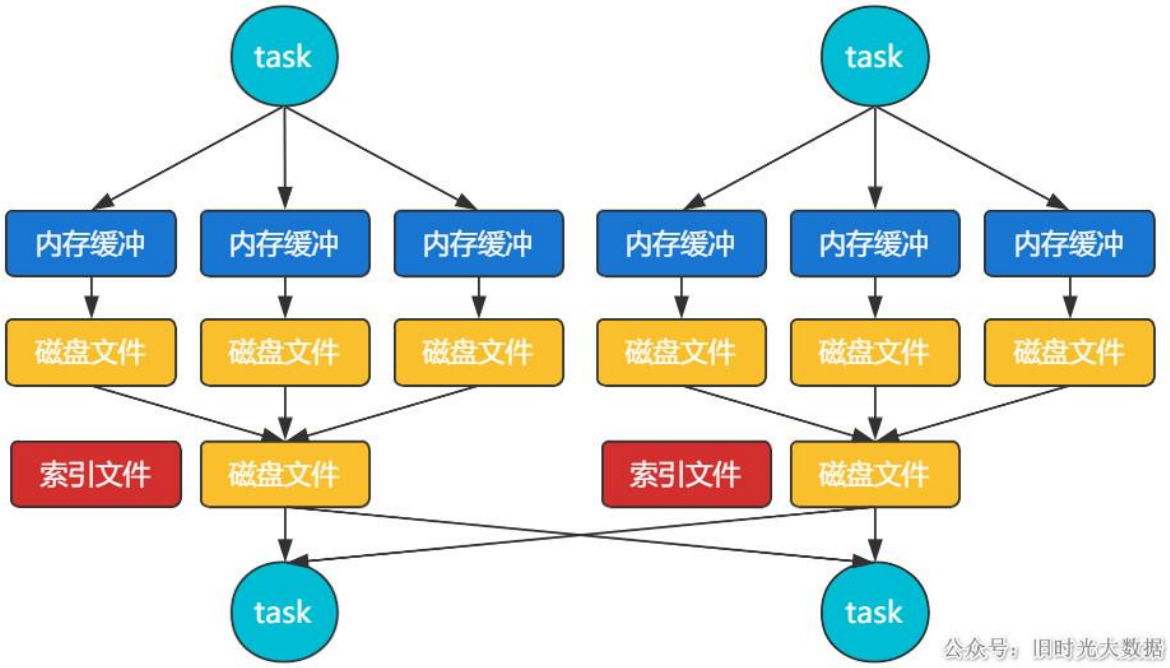
在溢写到磁盘文件之前，会先根据key对内存数据结构中已有的数据进行排序。排序过后，会分批将数据写入磁盘文件。默认的batch数量是10000条，也就是说，排序好的数据，会以每批1万条数据的形式分批写入磁盘文件。

一个task将所有数据写入内存数据结构的过程中，会发生多次磁盘溢写操作，也会产生多个临时文件。最后会将之前所有的临时磁盘文件都进行合并，由于一个task就只对应一个磁盘文件因此还会单独写一份索引文件，其中标识了下游各个task的数据在文件中的start offset与end offset。

SortShuffleManager由于有一个磁盘文件merge的过程，因此大大减少了文件数量，由于每个task最终只有一个磁盘文件所以文件个数等于上游shuffle write个数。

bypass机制的Sort Shuffle

公众号：旧时光大数据



bypass运行机制的触发条件如下：

- shuffle map task数量小于spark.shuffle.sort.bypassMergeThreshold参数的值，默认值200。
- 不是聚合类的shuffle算子(比如reduceByKey)。

此时task会为每个reduce端的task都创建一个临时磁盘文件，并将数据按key进行hash然后根据key的hash值，将key写入对应的磁盘文件之中。当然，写入磁盘文件时也是先写入内存缓冲，缓冲写满之后再溢写到磁盘文件的。最后，同样会将所有临时磁盘文件都合并成一个磁盘文件，并创建一个单独的索引文件。

该过程的磁盘写机制其实跟未经优化的HashShuffleManager是一模一样的，因为都要创建数量惊人的磁盘文件，只是在最后会做一个磁盘文件的合并而已。因此少量的最终磁盘文件，也让该机制相对未经优化的HashShuffleManager来说，shuffle read的性能会更好。

而该机制与普通SortShuffleManager运行机制的不同在于：

- 磁盘写机制不同；
- 不会进行排序。也就是说，启用该机制的最大好处在于，shuffle write过程中，不需要进行数据的排序操作，也就节省掉了这部分的性能开销。

3）Spark Shuffle总结

Shuffle 过程本质上都是将 Map 端获得的数据使用分区器进行划分，并将数据发送给对应的 Reducer 的过程。

Shuffle作为处理连接map端和reduce端的枢纽，其shuffle的性能高低直接影响了整个程序的性能和吞吐量。map端的shuffle一般为shuffle的Write阶段，reduce端的shuffle一般为shuffle的read阶段。Hadoop和spark的shuffle在实现上面存在很大的不同，spark的shuffle分为两种实现，分别为HashShuffle和SortShuffle。

HashShuffle又分为普通机制和合并机制，普通机制因为其会产生MR个数的巨量磁盘小文件而产生大量性能低下的Io操作，从而性能较低，因为其巨量的磁盘小文件还可能导致OOM，HashShuffle的合并机制通过重复利用buffer从而将磁盘小文件的数量降低到CoreR个，但是当Reducer 端的并行任务或者是数据分片过多的时候，依然会产生大量的磁盘小文件。

SortShuffle也分为普通机制和bypass机制，普通机制在内存数据结构(默认为5M)完成排序，会产生2M个磁盘小文件。而当shuffle map task数量小于spark.shuffle.sort.bypassMergeThreshold参数的值。或者算子不是聚合类的shuffle算子(比如reduceByKey)的时候会触发SortShuffle的bypass机制，SortShuffle的bypass机制不会进行排序，极大的提高了其性能。

公众号：旧时光大数据

在Spark 1.2以前，默认的shuffle计算引擎是HashShuffleManager，因为HashShuffleManager会产生大量的磁盘小文件而性能低下，在Spark 1.2以后的版本中，默认的ShuffleManager改成了SortShuffleManager。

SortShuffleManager相较于HashShuffleManager来说，有了一定的改进。主要就在于，每个Task在进行shuffle操作时，虽然也会产生较多的临时磁盘文件，但是最后会将所有的临时文件合并(merge)成一个磁盘文件，因此每个Task就只有一个磁盘文件。在下一个stage的shuffle read task拉取自己的数据时，只要根据索引读取每个磁盘文件中的部分数据即可。

4、Spark与MapReduce Shuffle的异同

从整体功能上看，两者并没有大的差别。 都是将 mapper（Spark 里是 ShuffleMapTask）的输出进行 partition，不同的 partition 送到不同的 reducer（Spark 里 reducer 可能是下一个 stage 里的 ShuffleMapTask，也可能是 ResultTask）。Reducer 以内存作缓冲区，边 shuffle 边 aggregate 数据，等到数据 aggregate 好以后进行 reduce（Spark 里可能是后续的一系列操作）。

从流程的上看，两者差别不小。 Hadoop MapReduce 是 sort-based，进入 combine和 reduce的 records 必须先 sort。这样的好处在于 combine/reduce可以处理大规模的数据，因为其输入数据可以通过外排得到（mapper 对每段数据先做排序，reducer 的 shuffle 对排好序的每段数据做归并）。以前 Spark 默认选择的是 hash-based，通常使用 HashMap 来对 shuffle 来的数据进行合并，不会对数据进行提前排序。如果用户需要经过排序的数据，那么需要自己调用类似 sortByKey的操作。在Spark 1.2之后，sort-based变为默认的Shuffle实现。

从流程实现角度来看，两者也有不少差别。 Hadoop MapReduce 将处理流程划分出明显的几个阶段：map, spill, merge, shuffle, sort, reduce等。每个阶段各司其职，可以按照过程式的编程思想来逐一实现每个阶段的功能。在 Spark 中，没有这样功能明确的阶段，只有不同的 stage 和一系列的 transformation，所以 spill, merge, aggregate 等操作需要蕴含在 transformation中。

MapReduce和Spark的Shuffle过程对比

	MapReduce	Spark
collect	在内存中构造了一块数据结构用于map输出的缓冲	没有在内存中构造一块数据结构用于map输出的缓冲，而是直接把输出写到磁盘文件
sort	map输出的数据有排序	map输出的数据没有排序
merge	对磁盘上的多个spill文件最后进行合并成一个输出文件	在map端没有merge过程，在输出时直接是对应一个reduce的数据写到一个文件中，这些文件同时存在并发写，最后不需要合并成一个
copy框架	jetty	netty或者直接socket流
对于本节点上的文件	仍然是通过网络框架拖取数据	不通过网络框架，对于在本节点上的map输出文件，采用本地读取的方式
copy过来的数据存放位置	先放在内存，内存放不下时写到磁盘	一种方式全部放在内存； 另一种方式先放在内存
merge sort	最后会对磁盘文件和内存中的数据进行合并排序	对于采用另一种方式时也会有合并排序的过程

公众号：旧时光大数据

若Spark要保存数据到HDFS上，要用什么算子?

问过的一些公司：小米

参考答案：

Action算子：saveAsTextFile

Hive SQL和Spark SQL的区别？各自优势和不足？为什么不用Spark SQL替代Hive SQL?

可回答：Hive和Spark SQL的区别?

问过的一些公司：蘑菇街x2，京东，有赞，网易(2021.09)

参考答案：

1、Hive

Hive是基于Hadoop的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供完整的sql查询功能，可以将sql语句转换为MapReduce任务进行运行。其优点是学习成本低，可以通过类SQL语句快速实现简单的MapReduce统计，不必开发专门的MapReduce应用，十分适合数据仓库的统计分析。

Hive是建立在 Hadoop 上的数据仓库基础构架。它提供了一系列的工具，可以用来进行数据提取转化加载(ETL)，这是一种可以存储、查询和分析存储在 Hadoop 中的大规模数据的机制。Hive 定义了简单的类 SQL 查询语言，称为 HQL，它允许熟悉 SQL 的用户查询数据。同时，这个语言也允许熟悉 MapReduce 开发者的开发自定义的 mapper 和 reducer 来处理内建的 mapper 和 reducer 无法完成的复杂的分析工作。

2、Spark SQL

Spark SQL主要用于结构化数据处理和对Spark数据执行类SQL的查询。通过Spark SQL，可以针对不同格式的数据执行ETL操作（如JSON，Parquet，数据库）然后完成特定的查询操作。一般来说，Spark每支持一种新的应用开发，都会引入一个新的Context及相应的RDD，对于SQL这一特性来说，引入的就是SQLContext和SchemaRDD。

注意：在Spark1.3之后，SchemaRDD已经更名为DataFrame，但它本质就类似一个RDD，因为可以将DataFrame无缝的转换成一个RDD。

3、比对

Hive是基于Hadoop的一个数据仓库工具，同时也是查询引擎，基于Hadoop做数据清洗（ETL）、报表、数据分析等。但是，对于实时插入、更新、删除数据，还要求强一致性和毫秒级响应，这个就不是Hive的长处，因为MapReduce处理数据是一个个阶段进行的，每个阶段都要落盘，不能多个map串联使用。

对于Spark，有RDD，可以把map过程串起来，中间过程存在内存中，再做一些优化，官方给出的答案是比Hadoop处理速度快乐10-100倍。

Hive适合处理离线非实时数据、数据仓库存储。

Spark SQL适合实时性要求或者速度要求较高的场所。

分布式存储系统和分布式计算框架区别?

问过的一些公司：美团大众点评

参考答案：

公众号：旧时光大数据

分布式存储系统：

将数据分散存储在多台独立的设备上。传统的网络存储系统采用集中的存储服务器存放所有数据，存储服务器成为系统性能的瓶颈，也是可靠性和安全性的焦点，不能满足大规模存储应用的需要。分布式网络存储系统采用可扩展的系统结构，利用多台存储服务器分担存储负荷，利用位置服务器定位存储信息，它不但提高了系统的可靠性、可用性和存取效率，还易于扩展。比如HDFS。

分布式计算框架：

分布式计算是利用网络把成千上万台计算机连接起来，组成一台虚拟的超级计算机，完成单台计算机无法完成的超大规模的问题求解。开放分布式计算架构是指以分布式计算技术为基础，用于解决大规模的问题开放式软件架构。开放分布式计算架构具有较好的可移植性和可裁剪性。比如MapReduce（Hadoop）、Spark、Flink。

分布式存储是指将数据（副本）分散存储在多台设备上，比如HDFS默认副本数量是3个，这保证了分布式存储的数据的稳定性。

分布式计算是指把需要进行大量计算的工程数据分区成小块，由多台计算机分别计算，再上传运算结果后，将结果统一合并得出数据结论。

ETL过程?

问过的一些公司：创略科技

参考答案：

ETL是英文Extract-Transform-Load 的缩写，用来描述将数据从来源端经过抽取（extract）、转换（transform）、加载（load）至目的端的过程。

ETL过程中的主要环节就是**数据抽取**、**数据转换和加工**、**数据装载**。

在ETL三个部分中，花费时间最长的是“T”(Transform，清洗、转换)的部分，一般情况下这部分工作量是整个ETL的2/3。

为了实现这些功能，各个ETL工具一般会进行一些功能上的扩充，例如工作流、调度引擎、规则引擎、脚本支持、统计信息等。

1、抽取作业

数据抽取是从数据源中抽取数据的过程。实际应用中，数据源较多采用的是关系数据库。

1）从数据库中抽取数据的方式：

全量抽取

全量抽取类似于数据迁移或数据复制，它将数据源中的表或视图的数据原封不动的从数据库中抽取出来，并转换成自己的ETL工具可以识别的格式。

全量抽取比较简单。

增量抽取

增量抽取只抽取自上次抽取以来数据库中要抽取的表中新增或修改的数据。

在ETL使用过程中，增量抽取较全量抽取应用更广，如何捕获变化的数据是增量抽取的关键。

对捕获方法一般有两点要求：

- 准确性，能够将业务系统中的变化数据按一定的频率准确地捕获到；
- 性能，不能对业务系统造成太大的压力，影响现有业务。

公众号：旧时光大数据

2）手工开发抽取作业时候的常用方法：

当数据源和DW为同一类数据库时

一般情况下，DBMS(SQLServer、Oracle)都会提供数据库链接功能，可以在数据源（业务系统）和DW内建立数据库链接（如DB2的联邦数据库NICKNAME），然后在DW内直接SELECT访问。

- 优点是实现使用简单，逻辑简单；
- 缺点是容易被滥用对源数据库造成较大的负载压力。

当数据源和ODS为不同类型数据库时

将源数据库的数据导出为文本文件，利用FTP协议进行传输导入ODS区域。

- 优点是实现简单，对源系统压力较小。
- 缺点是传输步骤增加了，处理需要的时间增加。

将部分数据库间能通过ODBC建立源数据库和目标数据库链接，此时也能直接使用SELECT获取数据。

- 优点是实现使用简单，逻辑简单；
- 缺点是容易被滥用对源数据库造成较大的负载压力，且建立时较为复杂。

3）更新数据的时间和数量的问题

实时抽取数据

这类抽取方式在数据仓库中很少见到，因为一般来说数据仓库对数据的实时性要求并不高。

实时抽取常见于BI中的CRM系统，比如在实时营销中，客户一旦进行了某类操作就实时触发对应的营销行为。

1.时间戳方式

要求源表中存在一个或多个字段(时间戳),其值随着新纪录的增加而不断增加，执行数据抽取时，程序定时循环检查通过时间戳对数据进行过滤，抽取结束后，程序记录时间戳信息。

- 优点是对源系统的侵入较小
- 缺点是抽取程序需要不断扫描源系统的表，对其 有一定压力

2.触发器方式

要求用户在源数据库中有创建触发器和临时表的权限，触发器捕获新增的数据到临时表中，执行抽取时，程序自动从临时表中读取数据。

- 优点是实时性极高
- 缺点是对源系统的侵入性较大，同时会对源数据库造成很大的压力（行级触发器），很可能影响源系统的正常业务

3.程序接口方式

改造源系统，在修改数据时通过程序接口同步发送数据至目标库，发送数据的动作可以跟业务修改数据动作脱耦，独立发送。

- 优点是对源系统的造成压力较小，实时性较强；
- 缺点是需要对源系统的侵入性较强，需要源系统做较大的改造。

批量抽取数据

为了保证数据抽取时数据的准确性、完整性和唯一性，同时降低抽取作业对源数据库造成的压力，抽取作业的加载必须避开源数据的生成时间。

这种方法一般用于实时性要求不高的数据。比如T+1或者每月1日进行抽取。

1.常用实现

公众号：旧时光大数据

- 日志检查
  - 需要源数据库生成数据完毕之后，在外部生成日志。
  - 抽取程序定时检查源系统的执行日志，发现完成标志后发起抽取作业。
  - 优点是可靠性高，对源数据库造成的压力较小。
  - 缺点是需要源数据库配合生成可供检查的外部日志。
- 约定时间抽取
  - 可以直接约定一个加载完毕同时对源数据库压力较小的时间（如每日凌晨2点），抽取程序建立定时任务，时间一到自动发起抽取作业。
    - 优点是对源数据库的侵入性和造成的压力较小；
    - 缺点是可靠性不高，可能会发生数据未生成完毕也直接进行抽取的情况。

2.根据下载时候对数据的筛选方式可以分为

- 全量下载
  - 适用：
    - 源数据量较小，如维表。
    - 数据变化较大，比如90%的数据都产生了变化的表。
    - 变化的数据不能预期，无法标示，如账户表。
  - 优缺点：
    - 优点在于下载较为简单且能容纳任何情况的数据变化；
    - 缺点是如果数据量较大，需要抽取相当长的时间，同时会占用大量的IO和网络资源。
- 增量下载：常用于数据只增不减的表，如交易明细表等。
  - 感知增量的方式如下：
  - 1）触发器
  - 在要抽取的表上建立需要的触发器，一般要建立插入、修改、删除三个触发器，每当源表中的数据发生变化，就被相应的触发器将变化的数据写入一个临时表，抽取线程从临时表中抽取数据，临时表中抽取过的数据被标记或删除。
    - 优点是数据抽取的性能较高，下载的数据较小，速度较快，占用资源少。
    - 缺点是要求业务表建立触发器，对业务系统有一定的影响，使用限制较大，有时候需要源系统进行改造支持。
  - 2）时间戳
  - 它是一种基于快照比较的变化数据捕获方式，在源表上增加一个时间戳字段，系统中更新修改表数据的时候，同时修改时间戳字段的值。当进行数据抽取时，通过比较系统时间与时间戳字段的值来决定抽取哪些数据。
    - 的数据库的时间戳支持自动更新，即表的其它字段的数据发生改变时，自动更新时间戳字段的值。
    - 有的数据库不支持时间戳的自动更新，这就要求业务系统在更新业务数据时，手工更新时间戳字段。
  - 优缺点：
    - 优点：同触发器方式一样，时间戳方式的性能也比较好，数据抽取相对清楚简单；
    - 缺点：对业务系统也有很大的倾入性（加入额外的时间戳字段），特别是对不支持时间戳的自动更新的数据库，还要求业务系统进行额外的更新时间戳操作。另外，无法捕获对时间戳以前数据的delete和update操作,在数据准确性上受到了一定的限制。
  - 3）全表比对
  - 典型的全表比对的方式是采用MD5校验码。
  - ETL工具事先为要抽取的表建立一个结构类似的MD5临时表，该临时表记录源表主键以及根据所有字段的数据计算出来的MD5校验码。

公众号：旧时光大数据

- 每次进行数据抽取时，对源表和MD5临时表进行MD5校验码的比对，从而决定源表中的数据是新增、修改还是删除，同时更新MD5校验码。
  - 优点是对源系统的倾入性较小（仅需要建立一个MD5临时表）
  - 缺点也是显而易见的，与触发器和时间戳方式中的主动通知不同，MD5方式是被动的进行全表数据的比对，性能较差。
  - 当表中没有主键或唯一列且含有重复记录时，MD5方式的准确性较差。
- 4）日志对比
  - 通过分析数据库自身的日志来判断变化的数据。

ETL处理的数据源除了关系数据库外，还可能是文件，例如txt文件、excel文件、xml文件等。

对文件数据的抽取一般是进行全量抽取，一次抽取前可保存文件的时间戳或计算文件的MD5校验码，下次抽取时进行比对，如果相同则可忽略本次抽取。

2、转换作业

这一步包含了数据的清洗和转换。

从数据源中抽取的数据不一定完全满足目的库的要求，例如数据格式的不一致、数据输入错误、数据不完整等等，因此有必要对抽取出的数据进行数据转换和加工。

数据的转换和加工可以在ETL引擎中进行，也可以在数据抽取过程中利用关系数据库的特性同时进行。

1）数据清洗

任务是过滤不符合条件或者错误的数据。

这一步常常出现在刚刚开始建立数据仓库或者源业务系统仍未成熟的时候，此时发现错误数据需要联系源业务系统进行更正，部分可预期的空值或者测试用数据可以过滤掉。

2）数据转换和加工

这一步是整个ETL流程中最为占用时间和资源的一步。数据转换包含了简单的数据不一致转换，数据粒度转换和耗时的数据关联整合或拆分动作。这里可能存在各种各样千奇百怪的需求。对于核心数据仓库来说，里面往往是对数据进行按照主题划分合并的动作。同时，也会添加一些为了提升执行效率而进行反范式化添加的冗余字段。数据的转换和加工可以在ETL引擎中进行，也可以在数据抽取过程中利用关系数据库的特性同时进行。根据实现方式的不同，可以区分为使用数据库存储过程转换和使用高级语言转换。

ETL引擎中的数据转换和加工

ETL引擎中一般以组件化的方式实现数据转换。

常用的数据转换组件有字段映射、数据过滤、数据清洗、数据替换、数据计算、数据验证、数据加解密、数据合并、数据拆分等。这些组件如同一条流水线上的一道道工序，它们是可插拔的，且可以任意组装，各组件之间通过数据总线共享数据。有些ETL工具还提供了脚本支持，使得用户可以以一种编程的方式定制数据的转换和加工行为。

在数据库中进行数据加工

关系数据库本身已经提供了强大的SQL、函数来支持数据的加工，如在SQL查询语句中添加where条件进行过滤，查询中重命名字段名与目的表进行映射，substr函数，case条件判断等等。

下面是一个SQL查询的例子。

```
1   select ID as USERID, substr(TITLE, 1, 20) as TITLE, case when REMARK is null then
    ' ' else REMARK end as CONTENT from TB_REMARK where ID > 100;
```

相比在ETL引擎中进行数据转换和加工，直接在SQL语句中进行转换和加工更加简单清晰，性能更高。对于SQL语句无法处理的可以交由ETL引擎处理。

公众号：旧时光大数据

使用数据库存储过程转换

使用SQL开发存储过程完成转换作业是很多银行常用的方法。

- 优点是开发简单、能支持绝大部分转换场景；
- 缺点在于占用资源多且受制于单一数据库性能，无法做到横向扩展。

因此，除了业务的理解能力外，对SQL海量数据处理的优化能力在此也非常重要。

比如：

- 利用数据库的分区性，选择良好的分区键。
- 建表时合理选择主键和索引，关联时候必须使用主键或索引进行关联。
- 关注数据库对SQL的流程优化逻辑，尽量选择拆分复杂SQL，引导数据库根据你选择流程进行数据处理
- 合理反范式化设计表，留出适当的冗余字段，减少关联动作。

具体的优化根据不同的数据库有着不同的处理方式，根据所选用的数据库不同而定。

使用高级语言转换

使用高级语言包含了常用的开发C/C++/JAVA等程序对抽取的数据进行预处理。

- 优点是运行效率较高，可以通过横向扩展服务器数量来提高系统的转换作业处理能力；
- 缺点是开发较为复杂，同时虽然能进行较为复杂的逻辑的开发，但是对于大数据量的关联的支持能力较弱，特别是有复数的服务器并行处理的时候。

3、加载作业

将转换和加工后的数据装载到目的库中通常是ETL过程的最后步骤。

转换作业生成的数据有可能直接插入目标数据库，一般来说，这种情况常见于使用数据库存储过程进行转换作业的方案。此时，ETL作业位于目标数据库上，加载作业只需要使用INSERT或者LOAD的方式导入目标表即可。此时转换作业和加载作业往往是在同一加工中完成的。

当使用高级语言开发时，ETL作业有着专门的ETL服务器，此时，转换作业生成的往往是文本文件，在转换作业完成后需要使用目标库特有的工具导入或者通过INSERT入目标库。同时，根据抽取作业的数据抽取方式的不同（全量、增量），对目标表进行替换或者插入动作。

装载数据的最佳方法取决于所执行操作的类型以及需要装入多少数据。当目的库是关系数据库时，一般来说有两种装载方式：

- 直接SQL语句进行insert、update、delete操作。
- 采用批量装载方法，如bcp、bulk、关系数据库特有的批量装载工具或api。

大多数情况下会使用第一种方法，因为它们进行了日志记录并且是可恢复的。

但是，批量装载操作易于使用，并且在装入大量数据时效率较高。使用哪种数据装载方法取决于业务系统的需要。

4、流程控制

抽取加载和转换作业需要一个集中的调度平台控制他们的运行，决定执行顺序，进行错误捕捉和处理。较为原始的ETL系统就是使用CRON做定时控制，定时调起相应的程序或者存储过程。但是这种方式过于原始，只能进行简单的调起动作，无法实现流程依赖行为，同时按步执行的流程控制能力也弱，错误处理能力几乎没有，只适合于极其简单的情况。

对于自行开发的较为完善的ETL系统，往往需要具有以下几个能力：

1）流程步骤控制能力

公众号：旧时光大数据

调度平台必须能够控制整个ETL流程（抽取加载和转换作业），进行集中化管理，不能有流程游离于系统外部。

2）系统的划分和前后流程的依赖

由于整个ETL系统里面可能跨越数十个业务系统，开发人员有数十拨人，必须支持按照业务系统对ETL流程进行划分管理的能力。

同时必须具有根据流程依赖进行调度的能力，使得适当的流程能在适当的时间调起。

3）合理的调度算法

同一时间调起过多流程可能造成对源数据库和ETL服务器还有目标数据库形成较大负载压力，故必须有较为合理的调度算法。

4）日志和警告系统

必须对每一步的流程记录日志，起始时间，完成时间，错误原因等，方便ETL流程开发人员检查错误。

对于发生错误的流程，能及时通知错误人员进行错误检查和修复。

5）较高可靠性

5、常用商业ETL工具

常用的ETL工具有Ascential公司的Datastage、Informatica公司的Powercenter、 NCR Teradata公司的ETL Automation等。

1）Datastage

是使用高级语言进行开发ETL服务器的代表。使用JAVA进行开发E/T/L的整个流程，同时支持平行添加服务器提升处理效率的方法。

2）Powercenter

与Datastage类似，但元数据更加开放，存放在关系数据库中，可以很容易被访问。再有Powercenter不能像Datastage运行多个实例，且不支持定制开发，参数控制更乱。

3）Automation

基于Teradata的TD数据库的ETL调度框架。其ETL流程是使用DSQL的存储过程进行开发，利用TD数据库的海量数据处理能力，也具有一定的平行扩展能力。

数据湖和数据仓库的区别

问过的一些公司：美团(2021.08)

参考答案：

1、数据湖

数据湖是以其自然格式存储的数据的系统或存储库，同行是对象blob或文件。数据湖通常是企业所有数据的单一存储，包括源系统数据的原始副本，以及用于报告、可视化、分析和机器学习等任务的转换数据。数据湖可以包括来自关系数据库（行和列）的结构化数据，半结构化数据（CSV，日志，XML，JSON），非结构数据（电子邮件、文档、PDF）和二进制数据（图像、音频、视频）。

2、数据仓库

公众号：旧时光大数据

是一个面向主题的、集成的、相对稳定的、反映历史变化的数据集合，用于支持管理决策。其主要功能是将组织透过资讯系统之联机事务处理（OLTP）经年累月所积累的大量资料，透过数据仓库理论所特有的资料存储架构，作一有系统的分析整理，以利各种分析方法如联机分析处理（OLAP）、数据挖掘（Data Mining）之进行，并进而支持如决策支持系统（DSS）、主管资讯系统（EIS）之创建，**帮助决策者能快速有效的自大量资料中，分析出有价值的资讯，以利决策拟定及快速回应外在环境变动，帮助构建商业智能（BI）。**

通俗一点，**数据仓库存在的意义在于对企业的所有数据进行汇总，为企业各个部门提供统一的，规范的数据出口。**

3、数据仓库VS数据湖

相较而言，数据湖是较新的技术，拥有不断演变的架构。数据湖存储任何形式（包括结构化和非结构化）和任何格式（包括文本、音频、视频和图像）的原始数据。根据定义，数据湖不会接受数据治理，但专家们都认为良好的数据管理对预防数据湖转变为数据沼泽不可或缺。数据湖在数据读取期间创建模式。与数据仓库相比，数据湖缺乏结构性，而且更灵活；它们还提供了更高的敏捷性。值得一提的是，数据湖非常适合使用机器学习和深度学习来执行各种任务，比如数据挖掘和数据分析，以及提取非结构化数据等。

	数据仓库	数据湖
类型	结构化数据，而且这些数据必须与数据仓库事先定义的模型吻合	所有类型数据，如结构化数据、半结构化、非结构化数据等，数据的类型依赖于数据源系统的原始数据格式
目的	处理结构化数据，将他们或者转换为多维数据，或者转换为报表，以满足后续的高级报表及数据分析需求	非常适合深度分析，包括高级数据分析、机器学习、深度学习等
特点	高性能、可重复性、持续使用	便于探索、创新、灵活性高

离线处理和实时处理的区别

问过的一些公司：字节(2021.08)

参考答案：

1、离线处理

- 数据量大且时间周期长；
- 在大量数据上进行复杂的批量运算；
- 数据在计算之前已经固定，不再会发生变化；
- 能够方便的查询批量计算的结果。

2、实时处理

- 数据实时到达；
- 数据到达次序独立，不受应用系统所控制；
- 数据规模大且无法预知容量；
- 原始数据一经处理，除非特意保存，否则不能被再次取出处理，或者再次提取数据代价昂贵。

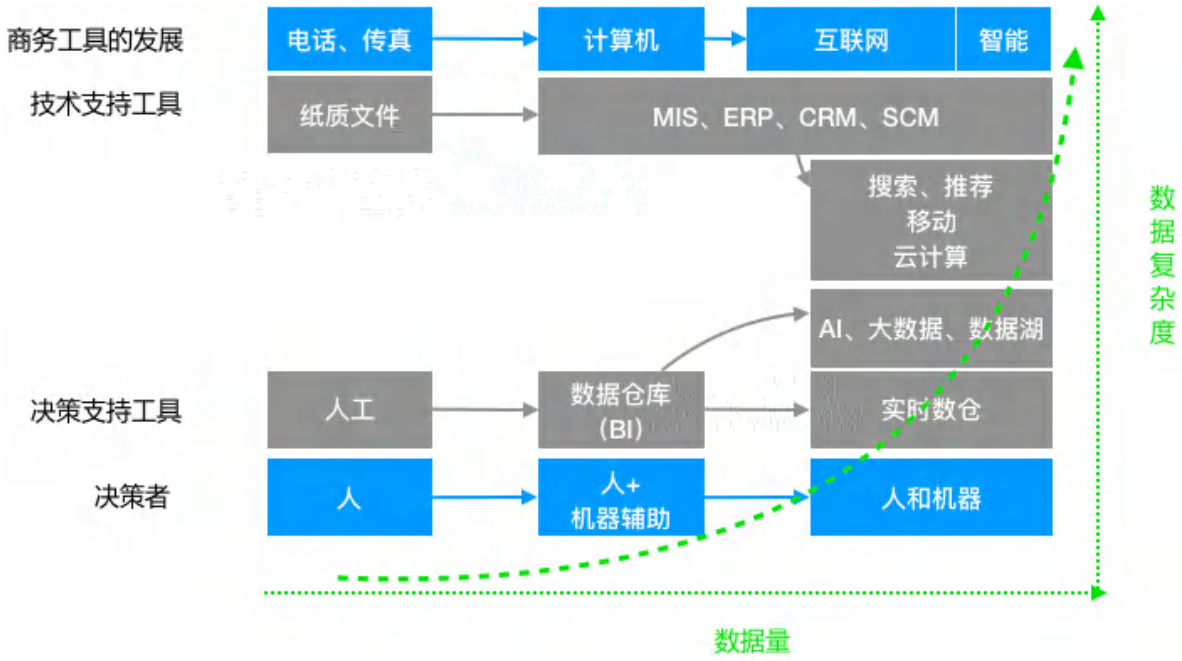
实时数仓和离线数仓的区别?

问过的一些公司：有赞

参考答案：

数据仓库的趋势：

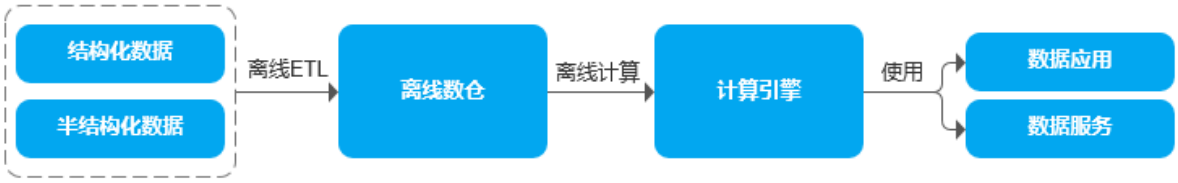
- 实时数据仓库以满足实时化&自动化决策需求
- 大数据&数据湖以支持大量&复杂数据类型



1、数据仓库架构的演变

从1990年 Inmon 提出数据仓库概念到今天，数仓架构经历了最初的传统数仓架构——离线数仓库——离线大数据架构、Lambda 架构、Kappa 架构以及 Flink 的火热带出的流批一体架构，数据架构技术不断演进，本质是在往流批一体的方向发展，让用户能以最自然、最小的成本完成实时计算。

1）传统数仓架构

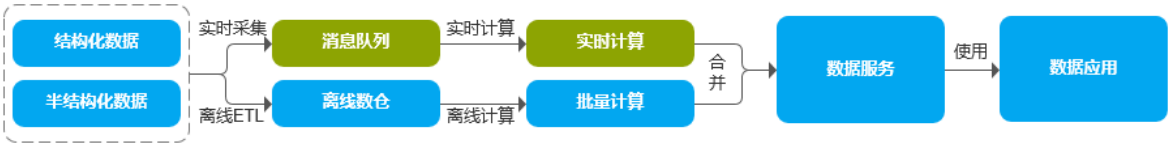


这是比较传统的一种方式，**结构或半结构化数据通过离线ETL定期加载到离线数仓，之后通过计算引擎取得结果，供前端使用。**这里的离线数仓+计算引擎，通常是使用大型商业数据库来承担，例如Oracle、DB2、Teradata等。

2）离线大数据架构

随着数据规模的不断增大，传统数仓方式难以承载海量数据。随着大数据技术的普及，采用大数据技术来承载存储与计算任务。当然，也可以使用传传统数据库集群或MPP架构数据库来完成。例如Hadoop+Hive/Spark、Oracle RAC、GreenPlum等。

3）Lambda架构（实时）



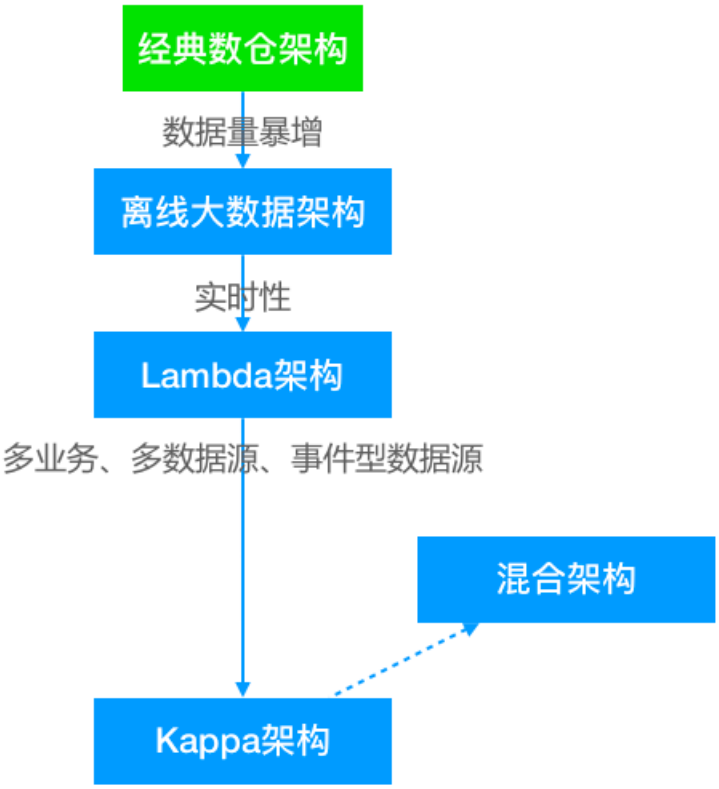
随着业务的发展，随着业务的发展，人们对数据实时性提出了更高的要求。此时，出现了Lambda架构，其将对实时性要求高的部分拆分出来，增加条实时计算链路。**从源头开始做流式改造，将数据发送到消息队列中，实时计算引擎消费队列数据，完成实时数据的增量计算**。与此同时，批量处理部分依然存在，实时与批量并行运行。最终由统一的数据服务层合并结果给于前端。一般是以批量处理结果为准，实时结果主要为快速响应。

4）Kappa架构（实时）

Lambda架构，一个比较严重的问题就是需要维护两套逻辑。一部分在批量引擎实现，一部分在流式引擎实现，维护成本很高。此外，对资源消耗也较大。而后面诞生的Kappa架构，正是为了解决上述问题。其在数据需要重新处理或数据变更时，可通过历史数据重新处理来完成。方式是通过上游重放完成(从数据源拉取数据重新计算)。Kappa架构最大的问题是流式重新处理历史的吞吐能力会低于批处理，但这个可以通过增加计算资源来弥补。

5）混合架构

上述架构各有其适应场景，有时需要综合使用上述架构组合满足实际需求。当然这也必将带来架构的复杂度。用户应根据自身需求，有所取舍。在一般大多数场景下，是可以使用单一架构解决问题。现在很多产品在流批一体、海量、实时性方面也有非常好的表现，可以考虑这种“全能手”解决问题。



2、三种大数据数据仓库架构

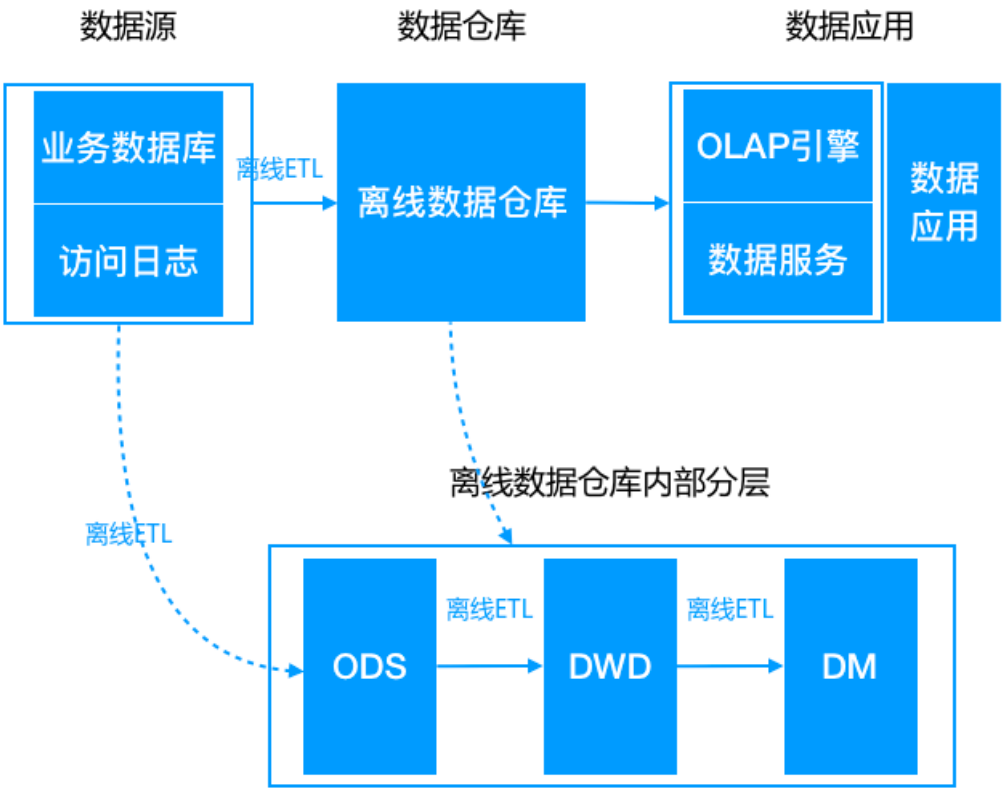
1）离线大数据架构

数据源通过离线的方式导入到离线数仓中。下游应用根据业务需求选择直接读取 DM 或加一层数据服务，比如 MySQL 或 Redis。数据仓库从模型层面分为三层：

- ODS，操作数据层，保存原始数据；
- DWD，数据仓库明细层，根据主题定义好事实与维度表，保存最细粒度的事实数据；
- DM，数据集市/轻度汇总层，在 DWD 层的基础之上根据不同的业务需求做轻度汇总；

典型的数仓存储是 HDFS/Hive，ETL 可以是 MapReduce 脚本或 HiveSQL。

公众号：旧时光大数据



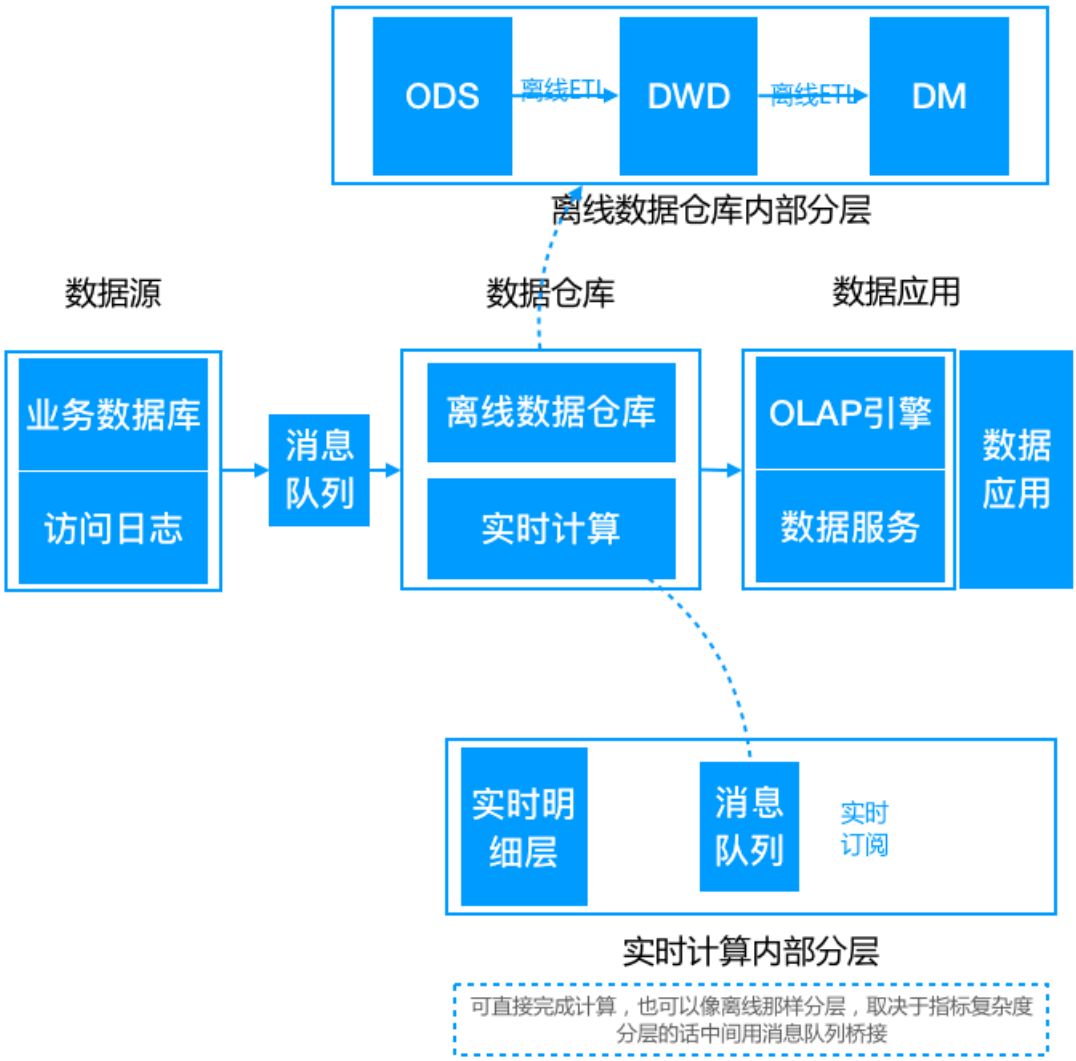
2）Lambda架构（实时）

Lambda 架构问题：

同样的需求需要开发两套一样的代码：这是 Lambda 架构最大的问题，两套代码不仅仅意味着开发困难（同样的需求，一个在批处理引擎上实现，一个在流处理引擎上实现，还要分别构造数据测试保证两者结果一致），后期维护更加困难，比如需求变更后需要分别更改两套代码，独立测试结果，且两个作业需要同步上线。

资源占用增多：同样的逻辑计算两次，整体资源占用会增多，多出实时计算这部分

公众号：旧时光大数据

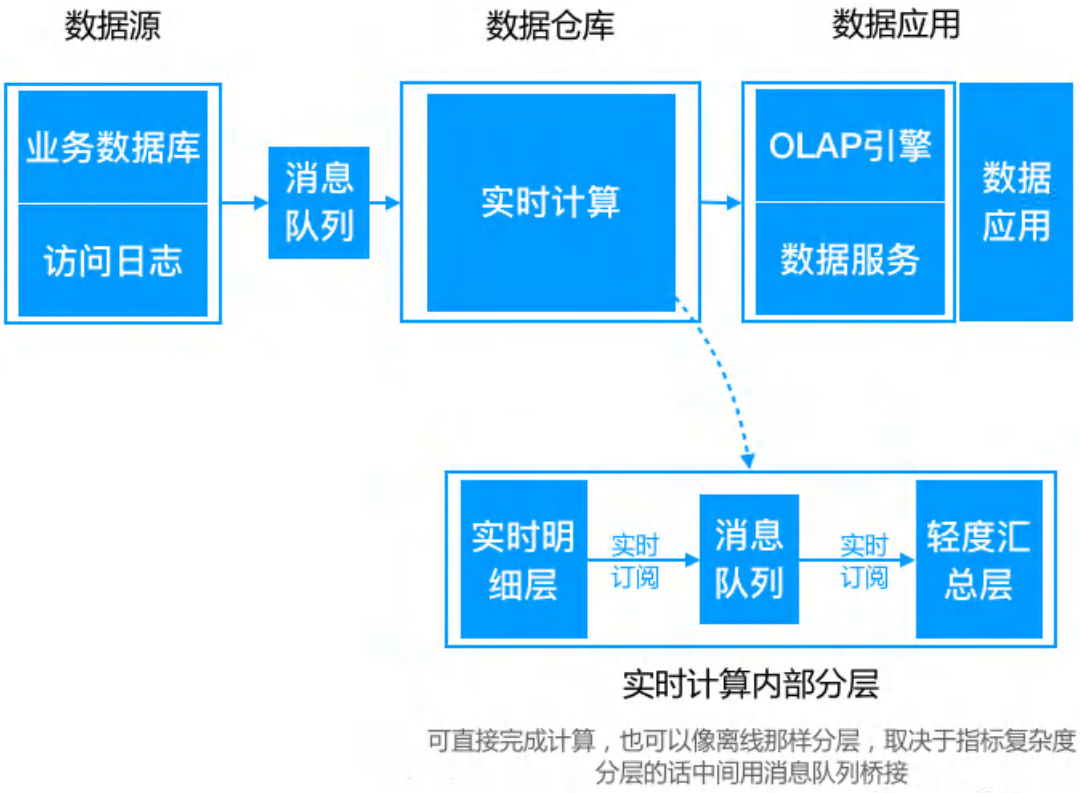


3）Kappa 架构（实时）

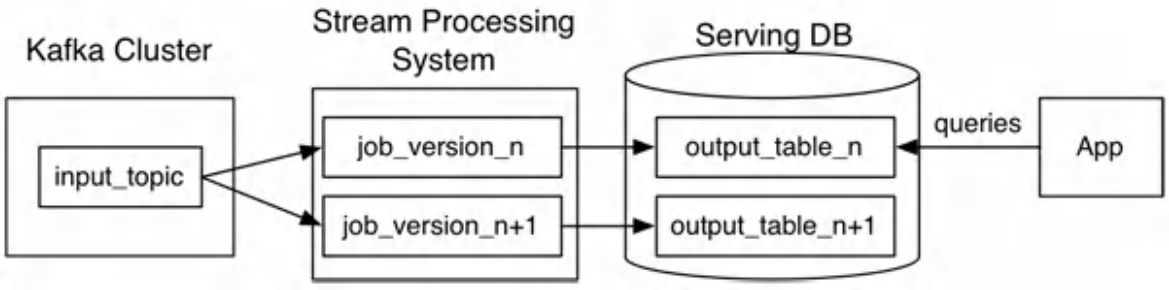
Lambda 架构虽然满足了实时的需求，但带来了更多的开发与运维工作，其架构背景是流处理引擎还不完善，流处理的结果只作为临时的、近似的值提供参考。后来随着 Flink 等流处理引擎的出现，流处理技术很成熟了，这时为了解决两套代码的问题，LickedIn 的 Jay Kreps 提出了 Kappa 架构。

- Kappa 架构可以认为是 Lambda 架构的简化版（只要移除 lambda 架构中的批处理部分即可）。
- 在 Kappa 架构中，需求修改或历史数据重新处理都通过上游重放完成。
- Kappa 架构最大的问题是流式重新处理历史的吞吐能力会低于批处理，但这个可以通过增加计算资源来弥补。

公众号：旧时光大数据



Kappa 架构的重新处理过程：

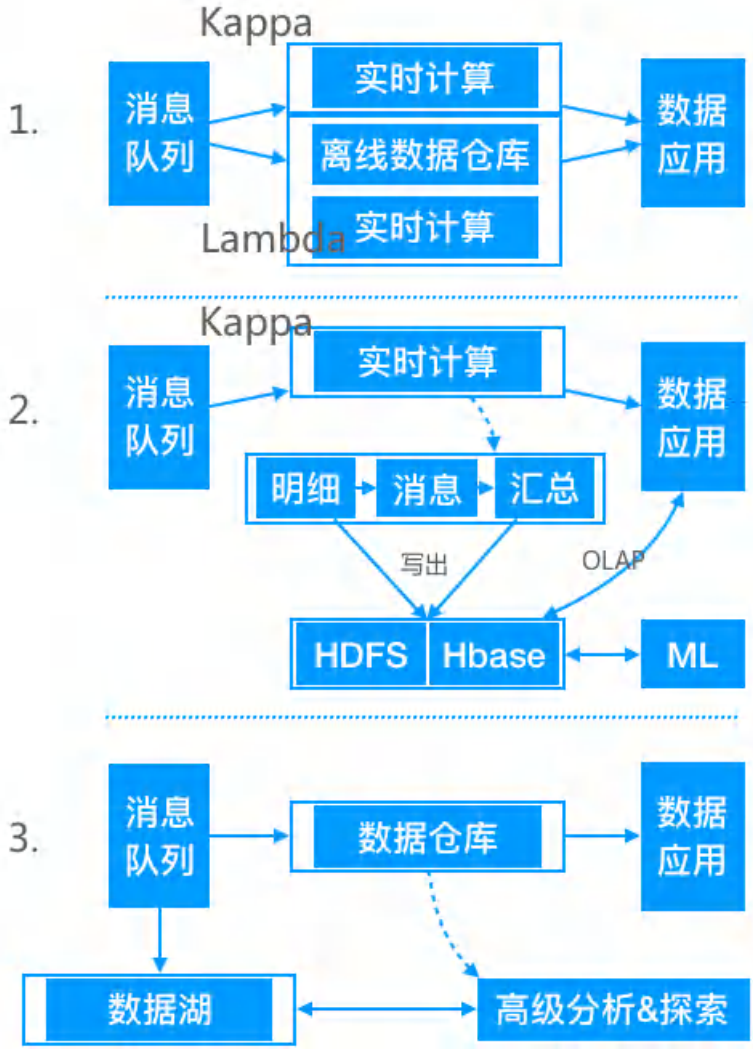


4）Lambda 架构与 Kappa 架构的对比

对比项	Lambda架构	Lappa架构
实时性	实时	实时
计算资源	批和流同时运行，资源开销大	只有流处理，仅针对新需求开发阶段运行两个作业，资源开销小
重新计算时吞吐	批式全量处理，吞吐较高	流式全量处理，吞吐较批处理低
开发、测试	每个需求都需要两套不同代码，开发、测试、上线难度较大	只需实现一套代码，开发、测试、上线难度相对较小
运维成本	维护两套系统（引擎），运维成本大	只需维护一套系统（引擎），运维成本小

在真实的场景中，很多时候并不是完全规范的 Lambda 架构或 Kappa 架构，可以是两者的混合，比如大部分实时指标使用 Kappa 架构完成计算，少量关键指标（比如金额相关）使用 Lambda 架构用批处理重新计算，增加一次校对过程。Kappa 架构并不是中间结果完全不落地，现在很多大数据系统都需要支持机器学习（离线训练），所以实时中间结果需要落地对应的存储引擎供机器学习使用，另外有时候还需要对明细数据查询，这种场景也需要把实时明细层写出到对应的引擎中。后面案例会涉及到。

公众号：旧时光大数据



Hadoop（HDFS）和MySQL的区别?

问过的一些公司：趋势科技

参考答案：

1、Hadoop

优点：

- Hadoop是天然分布式架构的，使用hdfs可以存储海量数据，通过分布式拓展可以获取无限的存储资源；使用map-reduce可以并行计算海量数据，通过分布式拓展同样可以获取无限的计算资源。
- hadoop支持存储和计算各种格式数据。数据来源可以使文本，mysql结构化数据，key/value键值对等

缺点：

- 入门门槛高，需要学习大量存储计算框架，而且还需要有一定分布式思维。
- 消耗资源比较多，一般来说都要很多节点同时工作，成本比较高。
- 处理数据量大，一般都是PB级别的，即使多个节点同时工作，依然处理一批数据长达小时级别（使用flink和spark可以达到实时计算）。
- 维护成本高，架构复杂，需要经验丰富开发人员作业。
- 使用map-reduce编写计算程序，需要编写map和reduce代码，不如mysql直接使用sql语句方便（hive,pig等框架支持sql转换为mapreduce程序）。

使用场景：

公众号：旧时光大数据

- 适合处理离线大数据，做一些报表，适合已经超过mysql处理能力范围之外的存储和计算。
- 由于计算存储可无限扩展，可以基于这个能力做一些数据分析挖掘工作，真正体现大数据框架的优势和价值。

2、MySQL

优点：

- mysql使用表存储数据,使用sql直接查询和一些聚合(sum,avg..)计算,使用操作简单；
- mysql上手入门门槛低,基本上就是安装,学会使用基本的DDl,DQL就算是入门了；
- mysql只是作为一个存储中间件,一般是单节点,主从节点的结构,架构相对简单,生态圈相对干净；
- mysql响应速度快，基本处理速度能达到毫秒级别。

缺点：

- 一般单节点运行,存储数据量小,一般单表达到几千万条数据量，性能就会受到影响；当然目前也可以采用分表分库（mycat,sharding jdbc）来解决。但是处理的数据依然有限。
- mysql因为处理数据集有限，很容易就达到检索索引瓶颈，因此当达到一定数据量时，为了满足毫秒级别响应，经常要做一些sql优化，借助redis缓存，分表分库来实现；对于一般开发人员而言是一个很大的挑战，需要很多优化方案，而且可能带来新的问题。
- mysql是关系型数据库，一般只方便用来处理结构固定的表结构数据，不支持或者不擅长别的格式数据的存储和处理 。
- mysql不太适合做一些报表统计类的业务，比如运营统计之类的，处理的话，耗时非常长，而且也是要做成离线库单独跑job，而且也只能统计小数据集的统计业务，毕竟单机处理能力有限。

使用场景：

- 因为mysql响应快，一般用于处理线上业务，比如订单级别的数据，一般数据量不会很大，完全可以通过mycat, sharding jdbc分表分库进行压力分担来解决。
- mysql作为一个关系型数据库，适合处理结构单一的表结构数据。
- 能处理小数据集的报表运营统计，但是挺耗费时间的，而且单机处理能力有限，一般做成离线数据库单独处理数据。

3、总结

mysql以处理小数据量，响应速度快为优势，服务于和用户交互场景。

hadoop以处理大数据量，服务于用户交互后场景的数据分析和挖掘场景。

说说Storm、Flink、Spark的区别，各自的优缺点，适用场景

问过的一些公司：百度，京东，阿里x2，华为，趋势科技

参考答案：

从流处理的角度将Flink与Spark和Storm这两个框架进行比较，会主要基于以下这几点展开：

功能性（Functionality）

是否能很好解决流处理功能上的痛点 , 比如event time和out of order data。

容错性（Fault Tolerance）

在failure之后能否恢复到故障之前的状态，并输出一致的结果；此外容错的代价也是越低越好，因为其直接影响性能。

吞吐量(throughputs)& 延时(latency)

性能相关的指标，高吞吐和低延迟某种意义上是不可兼得的，但好的流引擎应能兼顾高吞吐&低延时。

公众号：旧时光大数据

1、功能性（Functionality）

event time：指数据或者事件真正发生时间，比如用户点击网页时产生一条点击事件的数据，点击时间就是这条数据固有的event time。

processing time：指计算框架处理这条数据的时间。

spark DStream和storm 1.0以前版本往往都折中地使用processing time来近似地实现event time相关的业务。显然，使用processing time模拟event time必然会产生一些误差， 特别是在产生数据堆积的时候，误差则更明显，甚至导致计算结果不可用。

在使用event time时，自然而然需要解决由网络延迟等因素导致的迟到或者乱序数据的问题。为了解决这个问题， spark、storm及flink都引入了watermark和lateness的概念。

- watermark：是引擎处理事件的时间进度，代表一种状态，一般随着数据中的event time的增长而增长。比如 watermark(t)代表整个流的event time处理进度已经到达t， 时间是有序的，那么streaming不应该会再收到timestamp t' < t的数据，而只会接受到timestamp t' >= t的数据。 如果收到一条timestamp t' < t的数据， 那么就说明这条数据是迟到的。
- lateness：表示可以容忍迟到的程度，在lateness可容忍范围内的数据还会参与计算，超过的会被丢弃。

spark底层对static batch data和streaming data有共同的rdd抽象，完美兼容互操作。而flink中DataSet 和DataStream是完全独立的，不可以直接交互。

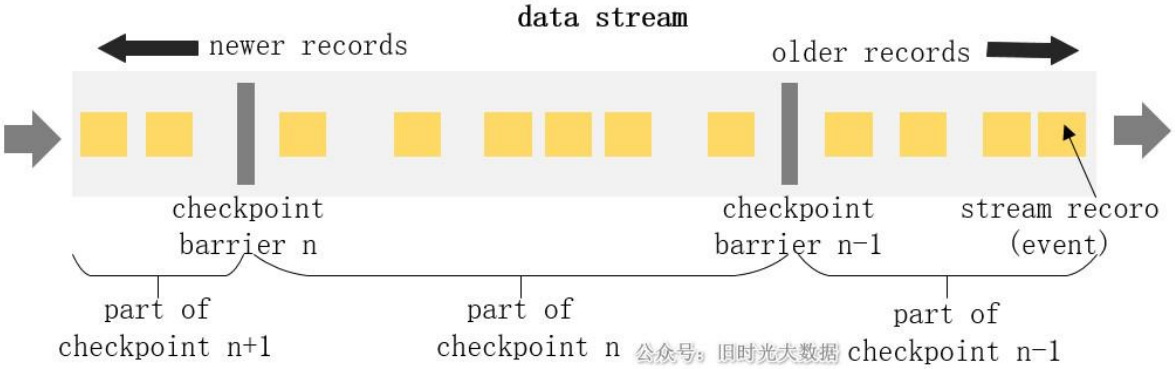
此外，flink还可以运行storm的topology，带来较强的移植性。另外一个有趣的功能是可以自由调整job latency and throughputs的取舍关系，比如需要high throughputs的程序可以牺牲latency来获得更大的throughputs。

2、容错性（Fault Tolerance）

spark依赖checkpoint机制来进行容错，只要batch执行到doCheckpoint操作前挂了，那么该batch就会被完整的重新计算。spark可以保证计算过程的exactly once（不包含sink的exactly once）。

storm的容错通过ack机制实现，每个bolt或spout处理完成一条data后会发送一条ack消息给acker bolt。当该条data被所有节点都处理过后，它会收到来自所有节点ack， 这样一条data处理就是成功的。storm可以保证数据不丢失，但是只能达到at least once语义。此外，因为需要每条data都做ack，所以容错的开销很大。storm trident是基于micro¬batched实现了exactly once语义。

flink使用Chandy-Chandy-Lamport Algorithm 来做Asynchronous Distributed Snapshots（异步分布式快照），其本质也是checkpoint。如下图，flink定时往流里插入一个barrier（隔栏），这些barriers把数据分割成若干个小的部分，当barrier流到某个operator时，operator立即会对barrier对应的一小部分数据做checkpoint并且把barrier传给下游（checkpoint操作是异步的，并不会打断数据的处理），直到所有的sink operator做完自己checkpoint后，一个完整的checkpoint才算完成。当出现failure时，flink会从最新完整的checkpoint点开始恢复。



flink的checkpoint机制非常轻量，barrier不会打断streaming的流动，而且做checkpoint操作也是异步的。其次，相比storm需要ack每条data，flink做的是small batch的checkpoint，容错的代价相对要低很多。最重要的是flink的checkpoint机制能保证exactly once。

公众号：旧时光大数据

3、吞吐量和延迟（Throughputs& Latency）

1）吞吐量（throughputs）

spark是mirco-batch级别的计算，各种优化做的也很好，它的throughputs是最大的。但是需要提一下，有状态计算（如updateStateByKey算子）需要通过额外的rdd来维护状态，导致开销较大，对吞吐量影响也较大。

storm的容错机制需要对每条data进行ack，因此容错开销对throughputs影响巨大，throughputs下降甚至可以达到70%。storm trident是基于micro-batch实现的，throughput中等。

flink的容错机制较为轻量，对throughputs影响较小，而且拥有图和调度上的一些优化机制，使得flink可以达到很高 throughputs。

storm在打开ack容错机制后，throughputs下降非常明显。而flink在开启checkpoint和关闭的情况下throughputs变化不大，说明flink的容错机制确实代价不高。对比官网的benchmark，我们也进行了throughputs的测试，实测结果是flink throughputs是storm的3.5倍，而且在解除了kafka集群和flink集群的带宽瓶颈后，flink自身又提高了1.6倍。

2）延迟（latency）

spark基于micro-batch实现，提高了throughputs，但是付出了latency的代价。一般spark的latency是秒级别的。

storm是native streaming实现，可以轻松的达到几十毫秒级别的latency，在几款框架中它的latency是最低的。storm trident是基于micro-batch实现的，latency较高。

flink也是native streaming实现，也可以达到百毫秒级别的latency。

从flink官网给出的和storm的latency对比benchmark得知。storm可以达到平均5毫秒以内的latency，而flink的平均latency也在30毫秒以内。两者的99%的data都在55毫秒latency内处理完成，表现都很优秀。

4、总结

综合对比spark、storm和flink的功能、容错和性能如下表

框架	Storm	Spark	Flink
Streaming模型	Native	Micro-batch	Native
确保性	At-least-once	Exactly-once	Exactly-once
容错性	Record-ACK	Checkpoint	Checkpoint
容错开销	高	中等	低
延迟	非常低	高	低
吞吐量	低	高	高

如果想要的是一个允许增量计算的高速事件处理系统，Storm会是最佳选择。

如果必须有状态的计算，恰好一次的递送，并且不介意高延迟的话，那么可以考虑Spark Streaming，特别如果你还计划图形操作、机器学习或者访问SQL的话，Apache Spark的stack允许你将一些library与数据流相结合(Spark SQL，Mllib，GraphX)，它们会提供便捷的一体化编程模型。尤其是数据流算法(例如：K均值流媒体)允许Spark实时决策的促进。

Flink支持增量迭代，具有对迭代自动优化的功能，在迭代式数据处理上，比Spark更突出，Flink基于每个事件一行一行地流式处理，真正的流式计算，流式计算跟Storm性能差不多，支持毫秒级计算，而Spark则只能支持秒级计算。

HDFS与HBase有什么关系?

问过的一些公司：百度

参考答案：

1、HDFS文件存储系统和HBase分布式数据库

HDFS是Hadoop分布式文件系统。

HBase的数据通常存储在HDFS上。HDFS为HBase提供了高可靠性的底层存储支持。

Hbase是Hadoop database，即Hadoop数据库。它是一个适合于非结构化数据存储的数据库，HBase基于列的而不是基于行的模式。

HBase是Google Bigtable的开源实现，类似Google Bigtable利用GFS作为其文件存储系统，HBase利用Hadoop HDFS作为其文件存储系统；Google运行MapReduce来处理Bigtable中的海量数据，HBase可以使用Hadoop MapReduce来处理HBase中的海量数据。

HDFS为HBase提供了高可靠性的底层存储支持，Hadoop MapReduce为HBase提供了高性能的计算能力，Zookeeper为HBase提供了稳定服务和failover机制。Pig和Hive还为HBase提供了高层语言支持，使得在HBase上进行数据统计处理变的非常简单。Sqoop则为HBase提供了方便的RDBMS（关系型数据库）数据导入功能，使得传统数据库数据向HBase中迁移变的非常方便。

2、HBase本身作为一个分布式数据库

HBase 本身其实可以完全不要考虑 HDFS 的，完全可以只把 HBase 当作是一个分布式高并发 k-v 存储系统，只不过它底层的文件系统是通过 HDFS 来支持的罢了。换做其他的分布式文件系统也是一样的，不影响 HBase 的本质。甚至如果不考虑文件系统的分布式或稳定性等特性的话，完全可以用简单的本地文件系统，甚至内存文件系统来代替。

HBase 在 HDFS 之上提供了：

- 高并发实时随机写，通过 LSM（内存+顺序写磁盘）的方式提供了 HDFS 所不拥有的实时随机写及修改功能
- 高并发实时点读及扫描了解一下 LSM 算法，在文件系统之上有数据库，在业务层面，HBase 完全可以独立于 HDFS 来理解

3、HBase可以满足大规模数据的实时处理需求

HDFS面向批量访问模式，不是随机访问模式Hadoop可以很好地解决大规模数据的离线批量处理问题，但是，受限于Hadoop MapReduce编程框架的高延迟数据处理机制，使得Hadoop无法满足大规模数据实时处理应用的需求。

- 传统的通用关系型数据库无法应对在数据规模剧增时导致的系统扩展性和性能问题(分库分表也不能很好解决)
- 传统关系数据库在数据结构变化时一般需要停机维护;空列浪费存储空间

因此，业界出现了一类面向半结构化数据存储和处理的高可扩展、低写入/查询延迟的系统，例如键值数据库、文档数据库和列族数据库(如BigTable和HBase等)。

HBase多是应用于互联网服务领域和传统行业的众多在线式数据分析处理系统中。

存储格式的选择，行式存储与列式存储的优劣

问过的一些公司：美团

公众号：旧时光大数据

参考答案：

1、优缺点分析

列存储优点：

- 单列数据保存在一起，不同列分开存储，导致存下同样一个表需要更多的Block文件，看起来是更复杂了，但是基于列和列分开存储，这种形式天生就适合分布式的存储，并能完全利用并发写入和并发读取的能力
- 同一列存放在一起，数据类型相同，则更好的进行压缩
- 同一列存放在一起，则排序更加方便，基于排序方便，where某一列会更加快

行存储优点：

- 更容易实现事务性、一致性控制。

应用场景：

- 关系型数据库基本都是行存储，Mysql、Oracle等，因其更强调一致性和事务性
- NoSQL多数为列存储，因为高效、吞吐量高，但事务、一致性较弱（也有做到事务性比较强的，但实现更加复杂，需要更多东西来配合）

总结：

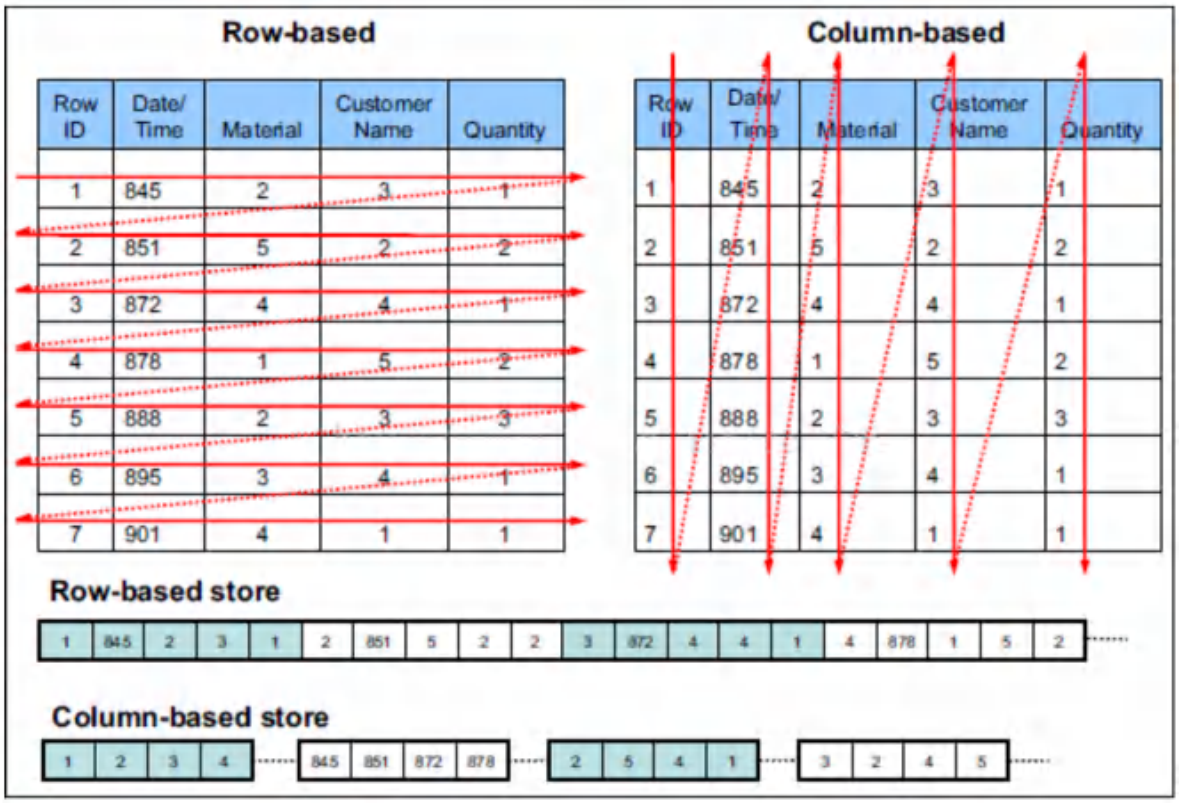
- 行：一致性、事务更加容易实现
- 列：吞吐量高、性能强，一致性、事务性较弱

2、列式存储和行式存储的比较

列式存储和行式存储是针对数据在存储介质中的排序形式而言的，假设存在一张table，那么：

- 行式存储：依次连续存储第**1、2、3...行**的数据到存储介质中；
- 列式存储：依次连续存储第**1、2、3...列**的数据到存储介质中。

下图所示为行式存储和列式存储的示意图，一张table包含5个字段（列）即rowid、date/time、customer name以及quantity，共7行，图中的红色箭头表示存储顺序。



3、行式存储和列式存储的优缺点和适用场景

公众号：旧时光大数据

存储形式的差异决定了适用场景的不同：

1）行式存储适合”**针对行”的查询**：

比如（mysql）select * from table_name limit 1，因为只会读取图1-1中的“row-based store 第1个绿色部分的数据”（只有指定的行“1 845 2 3 1”才会被读取），而select rowid from table_name则需要读取row-based store**所有**绿色部分的数据（虽然目的仅是要查询1个字段）；此外还适用于insert/update操作比较多的场景，因为只需要更改部分数据块即可。

2）列式存储适合“**针对列”的查询**：

比如select rowid from table_name，因为只会读取上图中的“column-based store 第1个绿色部分的数据”（查询时只有涉及到的字段会被读取），而select * from table_name limit 1则需要读取column-based stores所有绿色部分的数据（虽然目的就是要查询第1行的数据）；但是不适用于insert/update操作比较多的场景，比如当插入1个row时，由于列式存储导致同一个row的数据被分散在多个数据块中，因此需要去遍历所有数据块的数据。此外由于同一个字段连续存储（同一列的内容有很多值是重复的，可以压缩），因此更加便于编码压缩。

存储类别	适用于	不适用于
row-based	针对行的查询， insert/update操作	针对列的查询
column-based	针对列的查询， 编码压缩减小存储空间占用	针对行的查询， insert/update操作

综合来看，列式存储比较适合大数据量（压缩比高）、分析型操作（针对少数几列）；不适合频率较高的删除（全列检索）、更新（重新压缩）操作。

Hive、HBase、HDFS之间的关系

问过的一些公司：

参考答案：

1、Hive

Hive 是基于 Hadoop 的一个数据仓库工具，提供静态数据的动态查询。其使用类SQL语言，底层经过编译转为MapReduce程序，在Hadoop上运行，数据存储在HDFS上。

2、HDFS

HDFS是GFS的一种实现，他的完整名字是分布式文件系统，类似于FAT32，NTFS，是一种文件格式，是底层的。

Hive与Hbase的数据一般都存储在HDFS上。Hadoop HDFS为他们提供了高可靠性的底层存储支持。

3、HBase

Hbase是Hadoop database，即Hadoop数据库。它是一个适合于非结构化数据存储的数据库，HBase基于列的而不是基于行的模式。

HBase是Google Bigtable的开源实现，类似Google Bigtable利用GFS作为其文件存储系统，HBase利用Hadoop HDFS作为其文件存储系统；Google运行MapReduce来处理Bigtable中的海量数据，HBase可以使用Hadoop MapReduce来处理HBase中的海量数据。

公众号：旧时光大数据

Hadoop HDFS为HBase提供了高可靠性的底层存储支持，Hadoop MapReduce为HBase提供了高性能的计算能力，Zookeeper为HBase提供了稳定服务和failover机制。Pig和Hive还为HBase提供了高层语言支持，使得在HBase上进行数据统计处理变的非常简单。 Sqoop则为HBase提供了方便的RDBMS（关系型数据库）数据导入功能，使得传统数据库数据向HBase中迁移变的非常方便。

Hive中的数据在哪存放，MySQL的在哪存放?

问过的一些公司：端点数据(2021.07)

参考答案：

Hive是把数据存储在HDFS上

MySQL数据是存储在自己的系统中，文件默认存放位置MySQL安装目录下的data文件夹。

当数据量非常多，HDFS扛不住Flume采集的压力怎么办

问过的一些公司：端点数据(2021.07)

参考答案：

使用Kafka做一个削峰

如果上游数据时有突发流量，下游可能扛不住，或者下游没有足够多的机器来保证冗余，Kafka在中间可以起到一个缓冲的作用，把消息暂存在Kafka中，下游服务就可以按照自己的节奏进行慢慢处理。

Hadoop和gp（GreenPlum）区别

问过的一些公司：阿里(2021.09)

参考答案：

GreenPlum是**分布式数据库系统**。它是典型关系型数据库产品，是面向查询的关系型数据库。它的特点主要就是**查询速度快，数据装载速度快，批量DML处理快**。而且性能可以随着硬件的添加呈线性增加，拥有非常良好的可扩展性。因此，它主要适用于面向分析的应用。GreenPlum基于Apache MADLib具有高级机器学习功能，支持快速复杂查询分析，满足各种BI用户需求。

Hadoop是一种**分布式计算框架**，涉及分布式存储HDFS，分布式计算MapReduce，Yarn作业调度和集群资源管理框架。

可以把GreenPlum部署到Hadoop上，完成数据的分析处理。

MapReduce/Spark手撕WordCount

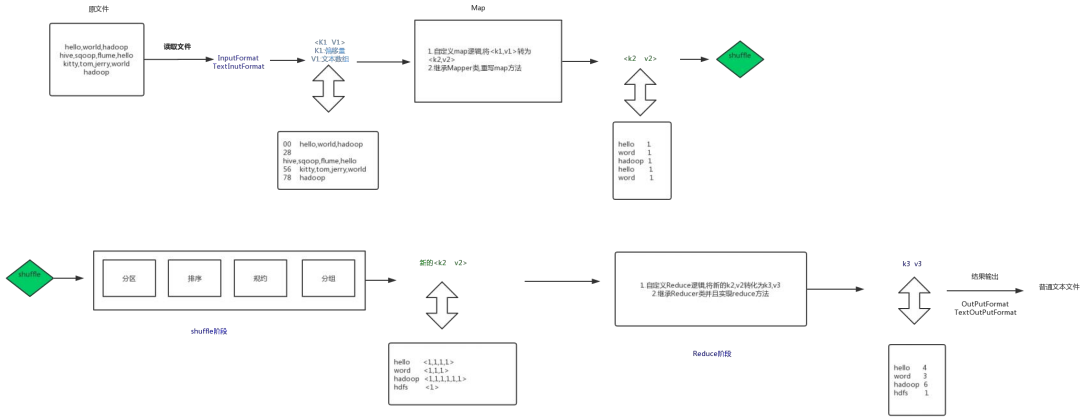
问过的一些公司：贝壳(2021.08)

参考答案：

1、MapReduce

公众号：旧时光大数据

WordCount流程



准备要处理的文件，并上传到hdfs分布式文件系统中

1）自定义mapper

```
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class WordCountMapper extends Mapper<LongWritable,Text,Text,LongWritable>
{

    //map方法就是将<k1,v1>转为<k2,v2>

    /**
     * @param key      k1 行偏移量
     * @param value    v1 每一行的文本数据
     * @param context   上下文对象,传递
     * @throws IOException
     * @throws InterruptedException
     */
    @Override
    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
        Text text = new Text();
        LongWritable longWritable = new LongWritable();
        //1.将一行的文本数据进行拆分
        String[] split = value.toString().split(",");
        //2.遍历数组,组装v2和k2
        for (String word : split) {
            text.set(word);
            longWritable.set(1);
            //3.将k2和v2 传入上下文中
            context.write(text,longWritable);

        }

    }
```

公众号：旧时光大数据

```
}
```

2）自定义reduce

```
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

/**
 * KEYIN  k2
 * VALUEIN v2
 * KEYOUT k3
 * VALUEOUT v3
 */

public class wordCountReducer extends
Reducer<Text,LongWritable,Text,LongWritable> {
    //将新的k2,v2转为k3和v3,将k3和v3写入上下文中
    /**
     * @param key  新k2
     * @param values  新v2
     * @param context  上下文对象
     * @throws IOException
     * @throws InterruptedException
     */
    @Override
    protected void reduce(Text key, Iterable<LongWritable> values, Context
context) throws IOException, InterruptedException {
        long count=0;
        //遍历集合,将集合中的数字相加,得到v3
        for (LongWritable value : values) {
            count+=value.get();
        }
        context.write(key,new LongWritable(count));
        //将k3和v3写入上下文中

    }
}
```

3）编写主代码

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

import java.net.URI;

public class JobMain extends Configured implements Tool {
```

公众号：旧时光大数据

```
16      //指定一个job任务
17      @Override
18      public int run(String[] strings) throws Exception {
19          //1.创建一个job任务对象
20          Job job = Job.getInstance(super.getConf(), "wordcount");
21          //如果打包运行出错,需要加入该配置
22          //job.setJarByClass(JobMain.class);
23          //2.配置job任务对象(8个步骤)
24          //第一步 指定文件的读取方式和读取路径
25          job.setInputFormatClass(TextInputFormat.class);
26          TextInputFormat.addInputPath(job,new
    Path("hdfs://node01:8020/wordcount"));
27          //TextInputFormat.addInputPath(job,new
    Path("file:///E:\\mapreduce\\input"));
28
29          //第二步 指定map阶段处理方式和数据类型
30          job.setMapperClass(WordCountMapper.class);
31          job.setMapOutputKeyClass(Text.class);
32          job.setMapOutputValueClass(LongWritable.class);
33
34          //3,4,5,6  采用默认方式
35          //第七步 指定 redce 的处理方式和数据类型
36          job.setReducerClass(wordCountReducer.class);
37          job.setOutputKeyClass(Text.class);
38          job.setOutputValueClass(LongWritable.class);
39
40          //第八步,设置输出类型
41          job.setOutputFormatClass(TextOutputFormat.class);
42
43          Path path =new Path("hdfs://node01:8020/wordcount_out");
44          //设置输出路径
45          TextOutputFormat.setOutputPath(job,path);
46          //TextOutputFormat.setOutputPath(job,new
    Path("file:///E:\\mapreduce\\output"));
47          FileSystem fileSystem = FileSystem.get(new URI("hdfs://node01:8020"),
    new Configuration());
48
49          boolean a = fileSystem.exists(path);
50          if(a){
51
52              fileSystem.delete(path,true);
53          }
54
55          //等待任务结束
56          boolean b = job.waitForCompletion(true);
57
58          return b?0:1;
59      }
60
61      public static void main(String[] args) throws Exception {
62          Configuration configuration = new Configuration();
63
64          //启动job任务
65          int run = ToolRunner.run(configuration, new JobMain(), args);
66          System.exit(run);
67
68      }
69  }
```

公众号：旧时光大数据

2、Spark

使用netcat工具向9999端口不断的发送数据，通过SparkStreaming读取端口数据并统计不同单词出现的次数

1）代码

```
object StreamWordCount {

  def main(args: Array[String]): Unit = {

    //1.初始化Spark配置信息
    val sparkConf = new SparkConf().setMaster("local[*]").setAppName("StreamWordCount")

    //2.初始化SparkStreamingContext
    val ssc = new StreamingContext(sparkConf, Seconds(3))

    //3.通过监控端口创建DStream，读进来的数据为一行行
    val lineStreams = ssc.socketTextStream("linux1", 9999)

    //将每一行数据做切分，形成一个个单词
    val wordStreams = lineStreams.flatMap(_.split(" "))

    //将单词映射成元组（word,1）
    val wordAndOneStreams = wordStreams.map((_, 1))

    //将相同的单词次数做统计
    val wordAndCountStreams = wordAndOneStreams.reduceByKey(_+_)

    //打印
    wordAndCountStreams.print()

    //启动SparkStreamingContext
    ssc.start()
    ssc.awaitTermination()
  }
}
```

2）启动程序并通过netcat发送数据：

```
nc -lk 9999
hello spark
```

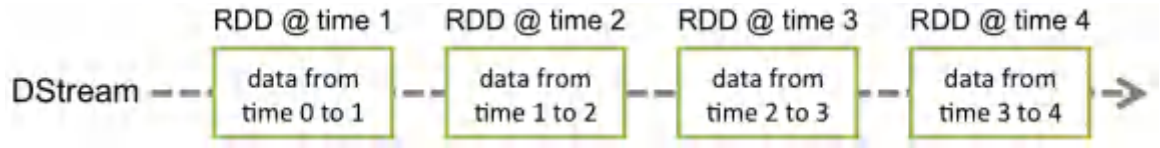
简洁版：

```
object WordCount {
    def main(args: Array[String]): Unit = {
        val lines = List("兰 亭 临 帖", "行 书 如 行 云 流 水")
        println(lines.flatMap(_.split(" ")).map((_,1)).groupBy(_._1).map(x => (x._1, x._2.size)).toList.sortBy(_._2))
    }
}
```

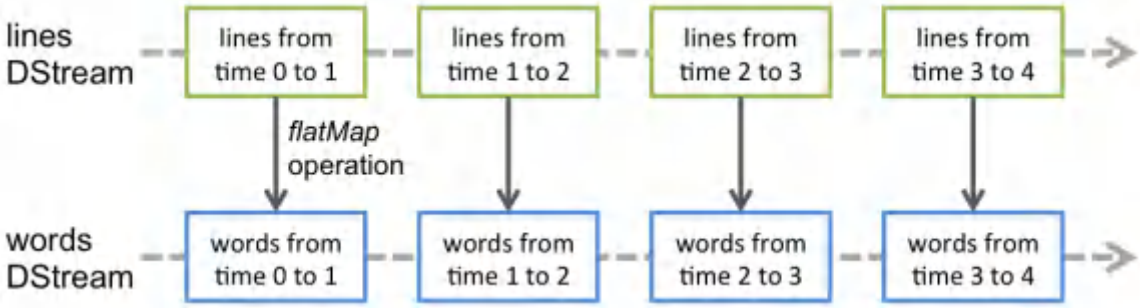
3）WordCount解析

Discretized Stream是Spark Streaming的基础抽象，代表持续性的数据流和经过各种Spark原语操作后的结果数据流。在内部实现上，DStream是一系列连续的RDD来表示。每个RDD含有一段时间间隔内的数据。

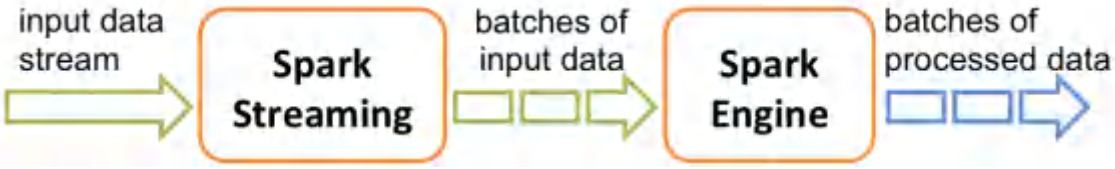
公众号：旧时光大数据



对数据的操作也是按照RDD为单位来进行的



计算过程由Spark Engine来完成



为什么要使用Scala开发Spark而不使用python

问过的一些公司：58同城(2021.07)

参考答案：

1、从性能上来看

由于**Java虚拟机的存在，Scala比Python快十倍**，而Python在数据分析和有效数据处理的性能方面则较慢。Python首先调用涉及大量代码处理的Spark库，并且性能自动降低。

同时，**当内核数量有限时，Scala更好。**如果他们的人数增加，那么Scala也会开始表现出怪异的行为，并且不被专业人员所喜欢。这里，问题来了，性能应该基于内核或数据处理来决定。显然，数据处理应视为性能的主要决定因素，毫无疑问，对于大数据Spark项目，Scala的性能要优于Python。

2、从并发上来看

基于大数据系统的复杂性，迫切**需要能够将各种数据库程序或服务集成在一起的编程语言。Scala在这里享有很高的偏爱，因为它提供了多个标准库和核心**，可帮助在大数据生态系统中快速集成数据库。

使用Scala，开发人员可以编写具有多个并发原语的更高效，可维护和可读的代码。同时，**Python不能很好地支持并发和多线程。**如果您将Python用于大型数据项目，则在该特定时间间隔内，Python进程中只有一个CPU处于活动状态。

3、从安全性来看

在为Spark项目开发代码时，开发人员需要不断对其进行重构。Scala是一种静态类型的语言，提供了一种捕获编译时错误的接口。与像Python这样的动态类型化语言相比，Scala中的重构代码轻松而轻松地体验。

4、从高级功能来看

Scala具有各种存在性类型，隐式和宏。

Scala在框架，库，隐式，宏等方面总是更加强大。对于NLP和机器学习，Python是最佳选择。

公众号：旧时光大数据

Scala和Java有什么区别

问过的一些公司：美团

参考答案：

1、什么是Scala

Scala是一种多范式的编程语言，其设计的初衷是要集成面向对象编程和函数式编程的各种特性。Scala运行于Java平台（Java虚拟机），并兼容现有的Java程序。

2、为什么要学Scala

优雅：这是框架设计师第一个要考虑的问题，框架的用户是应用开发程序员，API是否优雅直接影响用户体验。

速度快：Scala语言表达能力强，一行代码抵得上Java多行，开发速度快；Scala是静态编译的，所以和JRuby,Groovy比起来速度会快很多。

能融合到Hadoop生态圈：Hadoop现在是大数据事实标准，Spark并不是要取代Hadoop，而是要完善Hadoop生态。JVM语言大部分可能会想到Java，但Java做出来的API太丑，或者想实现一个优雅的API太费劲。

3、什么是Java

Java是一门面向对象编程语言，不仅吸收了C++语言的各种优点，还摒弃了C++里难以理解的多继承、指针等概念，因此Java语言具有功能强大和简单易用两个特征。Java语言作为静态面向对象编程语言的代表，极好地实现了面向对象理论，允许程序员以优雅的思维方式进行复杂的编程。

4、为什么要学Java

Java具有简单性、面向对象、分布式、健壮性、安全性、平台独立与可移植性、多线程、动态性等特点[2] 。Java可以编写桌面应用程序、Web应用程序、分布式系统和嵌入式系统应用程序等 。

5、Java和Scala的一些对比

公众号：旧时光大数据

	java	scala
对字符串的支持	Java需要采用“+”进行字符串的连接。	scala采用三个双引号“”“支持换行字符串
方法返回值	Java需要显示的采用return进行值的返回。	scala的return是可选的，方法调用会自动返回最后求值的表达式。如果scala使用了return则需要显示指定方法的返回值
类和方法修饰符的默认值	java默认是protected.	scala默认是public
默认导入的类	java默认导入java.lang包	scala默认导入java.lang包、scala包、scala.Predef类。
接口	java支持接口	scala不支持接口interface，采用trait（类似于Java中的抽象类）。
类成员和单例对象	java由类成员，单例对象需要自己实现。	scala语言机制上支持单例对象和伴生对象，伴生类。伴生类和伴生对象需要在一个类文件中使用，在使用伴生对象时，系统隐式地调用apply生成一个伴生实例的对象。

数据库面试题

数据库中的事务是什么，MySQL中是怎么实现的

可回答：1）什么是数据库事务；2）MySQL的事务原理；3）数据库事务是基于什么实现的?

问过的一些公司：字节，字节(2021.10)，阿里，百度，美团点评，米哈游，ebay多益

参考答案：

1、什么是事务

事务是应用程序中一系列严密的操作，所有操作必须成功完成，否则在每个操作中所作的所有更改都会被撤消。也就是事务具有原子性，一个事务中的一系列的操作要么全部成功，要么一个都不做。

事务的结束有两种，当事务中的所以步骤全部成功执行时，事务提交。如果其中一个步骤失败，将发生回滚操作，撤消撤消之前到事务开始时的所以操作。

2、MySQL中事务的实现原理

事务的目的是要实现可靠性以及并发处理。

可靠性：数据库要保证当insert或update操作时抛异常或者数据库crash的时候需要保障数据的操作前后的一致，想要做到这个，需要知道修改之前和修改之后的状态，所以就有了undo log和redo log。

并发处理：也就是说当多个并发请求过来，并且其中有一个请求是对数据修改操作的时候会有影响，为了避免读到脏数据，所以需要对事务之间的读写进行隔离，需要用到MySQL的事务隔离功能。

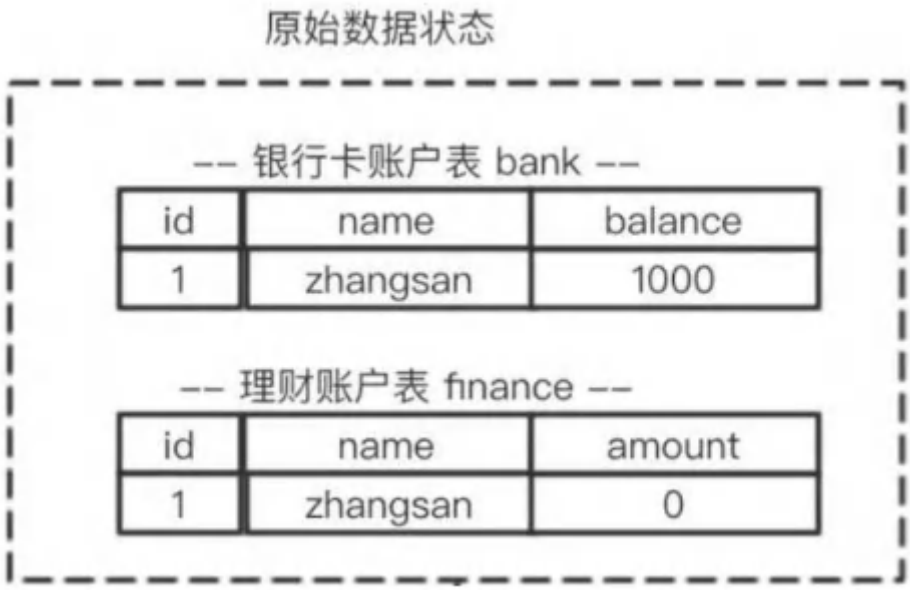
2.1 redo log 与 undo log介绍

1）redo log

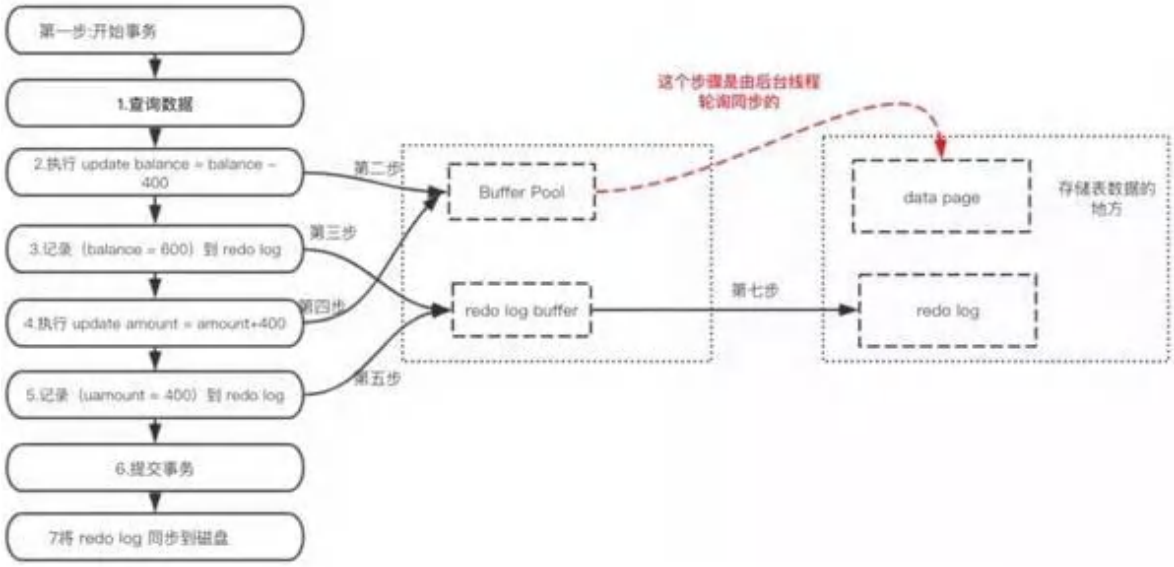
公众号：旧时光大数据

什么是redo log ?

redo log叫做重做日志，是用来实现事务的持久性。该日志文件由两部分组成：重做日志缓冲（redo log buffer）以及重做日志文件（redo log）,前者是在内存中，后者在磁盘中。当事务提交之后会把所有修改信息都会存到该日志中。假设有个表叫做tb1(id,username) 现在要插入数据（3，ceshi）




```
start transaction;
select balance from bank where name="zhangsan";
// 生成 重做日志 balance=600
update bank set balance = balance - 400;
// 生成 重做日志 amount=400
update finance set amount = amount + 400;
commit;
```



redo log 有什么作用?

mysql 为了提升性能不会把每次的修改都实时同步到磁盘，而是会先存到Boffer Pool(缓冲池)里头，把这个当作缓存来用。然后使用后台线程去做缓冲池和磁盘之间的同步。

那么问题来了，如果还没来的同步的时候宕机或断电了怎么办？还没来得及执行上面图中红色的操作。这样会导致丢部分已提交事务的修改信息！

所以引入了redo log来记录已成功提交事务的修改信息，并且会把redo log持久化到磁盘，系统重启之后在读取redo log恢复最新数据。

公众号：旧时光大数据

总结：

redo log是用来恢复数据的 用于保障已提交事务的持久化特性。

2）undo log

什么是 undo log ?

undo log 叫做回滚日志，用于记录数据被修改前的信息。他正好跟前面所说的重做日志所记录的相反，重做日志记录数据被修改后的信息。undo log主要记录的是数据的逻辑变化，为了在发生错误时回滚之前的操作，需要将之前的操作都记录下来，然后在发生错误时才可以回滚。

还用上面那两张表

每次写入数据或者修改数据之前都会把修改前的信息记录到 undo log。

undo log 有什么作用?

undo log 记录事务修改之前版本的数据信息，因此假如由于系统错误或者rollback操作而回滚的话可以根据undo log的信息来进行回滚到没被修改前的状态。

总结：

undo log是用来回滚数据的用于保障未提交事务的原子性

2.2 mysql锁技术以及MVCC基础

1）mysql锁技术

当有多个请求来读取表中的数据时可以不采取任何操作，但是多个请求里有读请求，又有修改请求时必须有一种措施来进行并发控制。不然很有可能会造成不一致。

读写锁

解决上述问题很简单，只需用两种锁的组合来对读写请求进行控制即可，这两种锁被称为：

- 共享锁(shared lock)，又叫做"读锁"，读锁是可以共享的，或者说多个读请求可以共享一把锁读数据，不会造成阻塞。
- 排他锁(exclusive lock)，又叫做"写锁"，写锁会排斥其他所有获取锁的请求，一直阻塞，直到写入完成释放锁。

公众号：旧时光大数据

	读锁	写锁
读锁	可并行	不可并行
写锁	不可并行	不可并行

读写锁兼容性

总结：

通过读写锁，可以做到读读可以并行，但是不能做到写读，写写并行。

事务的隔离性就是根据读写锁来实现的。

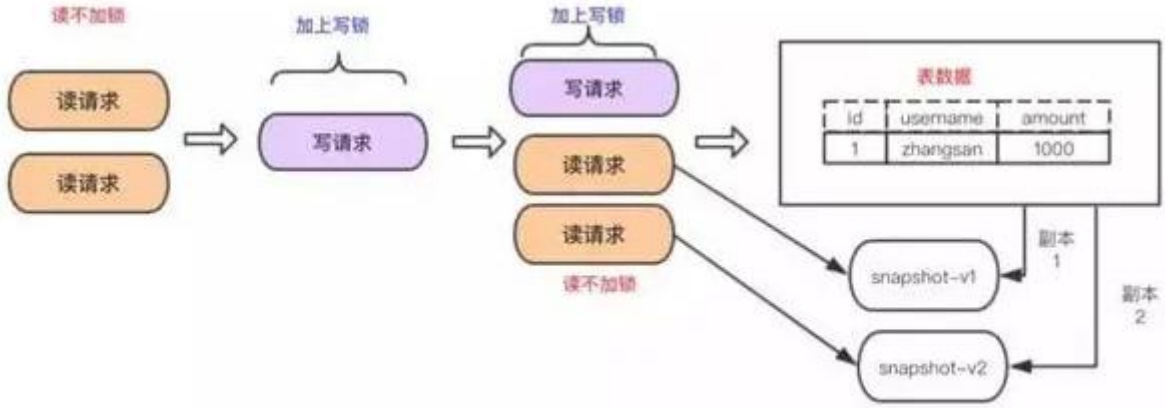
2）MVCC基础

MVCC (MultiVersion Concurrency Control) 叫做多版本并发控制。

InnoDB的 MVCC ，是通过在每行记录的后面保存两个隐藏的列来实现的。这两个列，一个保存了行的创建时间，一个保存了行的过期时间，当然存储的并不是实际的时间值，而是系统版本号

MVCC在mysql中的实现依赖的是undo log与read view

- undo log：undo log 中记录某行数据的多个版本的数据。
- read view：用来判断当前版本数据的可见性



提交读级别下的读写锁的使用

2.3 事务的实现

前面介绍的重做日志，回滚日志以及锁技术就是实现事务的基础。

- 事务的原子性是通过undolog来实现。
- 事务的持久性性是通过redolog来实现
- 事务的隔离性是通过(读写锁+MVCC)来实现
- 事务的一致性是通过原子性，持久性，隔离性共同实现

原子性，持久性，隔离性的目的也是为了保障数据的一致性！

总之，ACID只是个概念，事务最终目的是要保障数据的可靠性，一致性。

1）原子性的实现

什么是原子性：

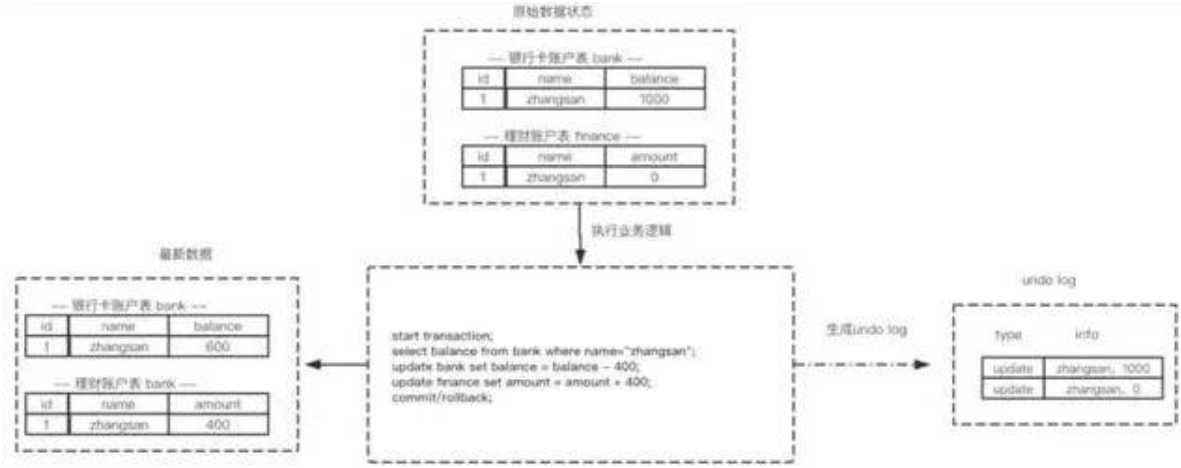
一个事务必须被视为不可分割的最小工作单位，一个事务中的所有操作要么全部成功提交，要么全部失败回滚，对于一个事务来说不可能只执行其中的部分操作，这就是事务的原子性。

公众号：旧时光大数据

那么数据库是怎么实现的呢？就是通过回滚操作。所谓回滚操作就是当发生错误异常或者显式的执行rollback语句时需要把数据还原到原先的模样，所以这时候就需要用到undo log来进行回滚，接下来看一下undo log在实现事务原子性时怎么发挥作用的

（1）undo log 的生成

假设有两个表 bank和finance，表中原始数据如图所示，当进行插入，删除以及更新操作时生成的undo log如下面图所示：



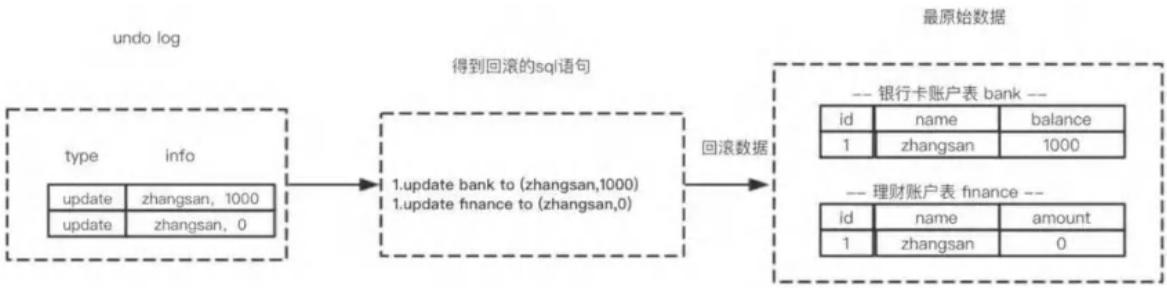
A.事务开始.
B.查询数据
C.进行update 操作 balance = balance – 400
D.记录(zhangsan,1000) 到undo log 回滚时需要把这条更新过来
E.进行update 操作 amount = amount + 400
F.记录(zhangsan,0) 到undo log 回滚时需要把这条更新过来
G.事务提交/回滚

根据上面流程可以得出如下结论：

- 每条数据变更(insert/update/delete)操作都伴随一条undo log的生成，并且回滚日志必须先于数据持久化到磁盘上；
- 所谓的回滚就是根据回滚日志做逆向操作，比如delete的逆向操作为insert，insert的逆向操作为delete，update的逆向为update等。

（2）根据undo log 进行回滚

为了做到同时成功或者失败，当系统发生错误或者执行rollback操作时需要根据undo log 进行回滚



回滚操作就是要还原到原来的状态，undo log记录了数据被修改前的信息以及新增和被删除的数据信息，根据undo log生成回滚语句，比如：

- 如果在回滚日志里有新增数据记录，则生成删除该条的语句
- 如果在回滚日志里有删除数据记录，则生成生成该条的语句
- 如果在回滚日志里有修改数据记录，则生成修改到原先数据的语句

2）持久性的实现

事务一旦提交，其所做的修改会永久保存到数据库中，此时即使系统崩溃修改的数据也不会丢失。

先了解一下MySQL的数据存储机制，MySQL的表数据是存放在磁盘上的，因此想要存取的时候都要经历磁盘IO,然而即使是使用SSD磁盘IO也是非常消耗性能的。为此，为了提升性能InnoDB提供了缓冲池(Buffer Pool)，Buffer Pool中包含了磁盘数据页的映射，可以当做缓存来使用：

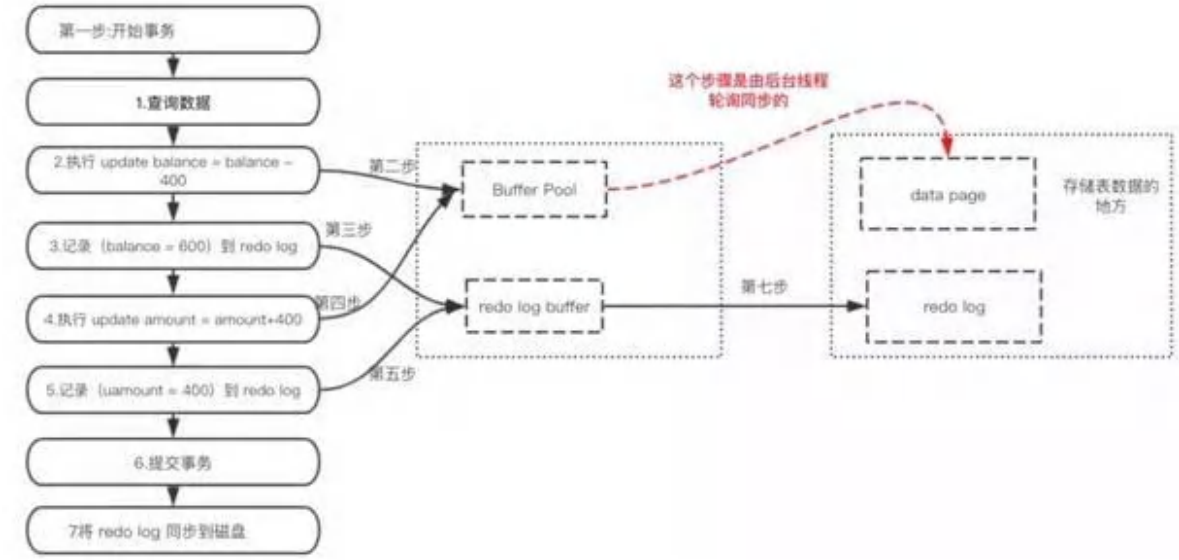
读数据：会首先从缓冲池中读取，如果缓冲池中没有，则从磁盘读取再放入缓冲池；

写数据：会首先写入缓冲池，缓冲池中的数据会定期同步到磁盘中；

上面这种缓冲池的措施虽然在性能方面带来了质的飞跃，但是它也带来了新的问题，当MySQL系统宕机，断电的时候可能会丢数据！！！

因为我们的数据已经提交了，但此时是在缓冲池里头，还没来得及在磁盘持久化，所以我们急需一种机制需要存一下已提交事务的数据，为恢复数据使用。

于是 redo log就派上用场了。下面看下redo log是什么时候产生的



既然redo log也需要存储，也涉及磁盘IO为啥还用它?

- redo log 的存储是顺序存储，而缓存同步是随机操作。
- 缓存同步是以数据页为单位的，每次传输的数据大小大于redo log。

3）隔离性的实现

隔离性是事务ACID特性里最复杂的一个。在SQL标准里定义了四种隔离级别，每一种级别都规定一个事务中的修改，哪些是事务之间可见的，哪些是不可见的。

级别越低的隔离级别可以执行越高的并发，但同时实现复杂度以及开销也越大。

MySQL隔离级别有以下四种（级别由低到高）：

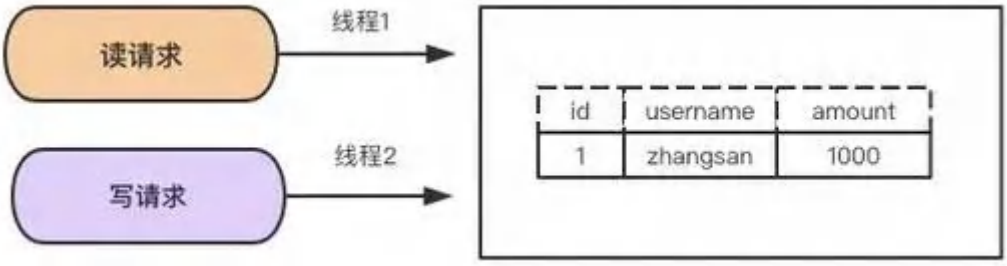
- READ UNCOMMITED(未提交读)
- READ COMMITED(提交读)
- REPEATABLE READ(可重复读)
- SERIALIZABLE (可重复读)

只要彻底理解了隔离级别以及他的实现原理就相当于理解了ACID里的隔离型。前面说过原子性，隔离性，持久性的目的都是为了要做到一致性，但隔离型跟其他两个有所区别，原子性和持久性是为了要实现数据的可性保障靠，比如要做到宕机后的恢复，以及错误后的回滚。

那么隔离性是要做到什么呢？ 隔离性是要管理多个并发读写请求的访问顺序。 这种顺序包括串行或者是并行

说明一点，写请求不仅仅是指insert操作，又包括update操作。

公众号：旧时光大数据



总之，从隔离性的实现可以看出这是一场数据的**可靠性与性能之间的权衡**：

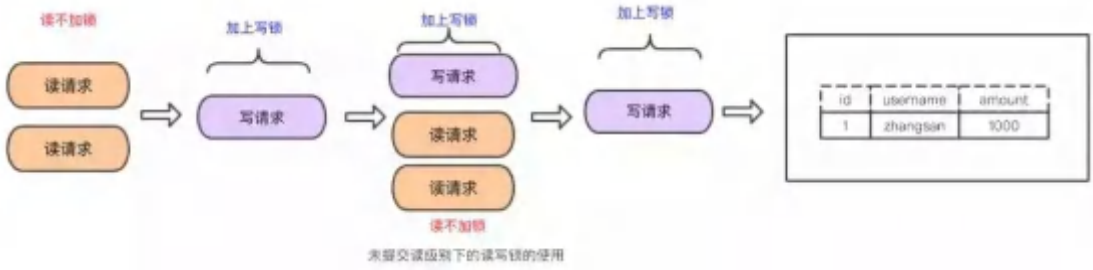
可靠性性高的，并发性能低(比如Serializable)。可靠性低的，并发性能高(比如 Read Uncommited)

（1）READ UNCOMMITTED

在READ UNCOMMITTED隔离级别下，事务中的修改即使还没提交，对其他事务是可见的。事务可以读取未提交的数据，造成脏读。

因为读不会加任何锁，所以写操作在读的过程中修改数据，所以会造成脏读。好处是可以提升并发处理性能，能做到读写并行。

换句话说，读的操作不能排斥写请求。



优点：读写并行，性能高

缺点：造成脏读

（2）READ COMMITTED

一个事务的修改在他提交之前的所有修改，对其他事务都是不可见的。其他事务能读到已提交的修改变化。在很多场景下这种逻辑是可以接受的。

InnoDB在 READ COMMITTED，使用排它锁,读取数据不加锁而是使用了MVCC机制。或者换句话说他采用了读写分离机制。

但是该级别会产生不可重读以及幻读问题。

什么是不可重读?

在一个事务内多次读取的结果不一样。

为什么会产生不可重复读?

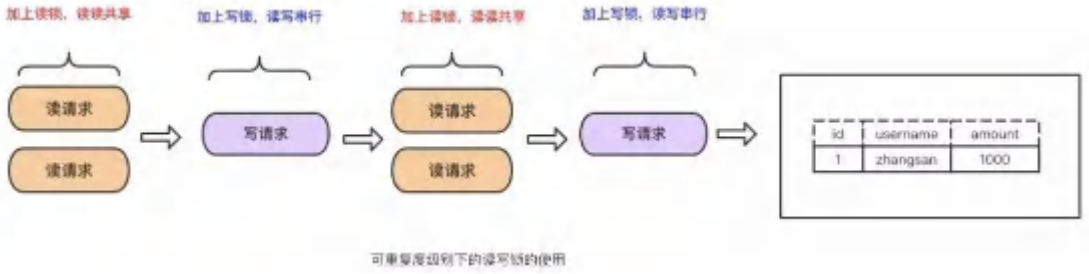
这跟 READ COMMITTED 级别下的MVCC机制有关系，在该隔离级别下每次 select的时候新生成一个版本号，所以每次select的时候读的不是一个副本而是不同的副本。

在每次select之间有其他事务更新了我们读取的数据并提交了，那就出现了不可重复读

（3）REPEATABLE READ(Mysql默认隔离级别)

在一个事务内的多次读取的结果是一样的。这种级别下可以避免，脏读，不可重复读等查询问题。mysql 有两种机制可以达到这种隔离级别的效果，分别是采用读写锁以及MVCC。

采用读写锁实现：



可重复度级别下的读写锁的使用

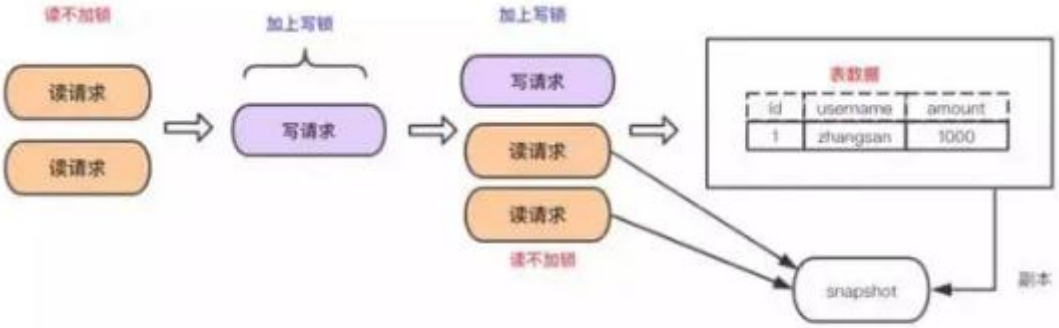
为什么能可重复读?

只要没释放读锁，在次读的时候还是可以读到第一次读的数据。

优点：实现起来简单

缺点：无法做到读写并行

采用MVCC实现：



可重复读级别下的读写锁的使用

为什么能可重复读?

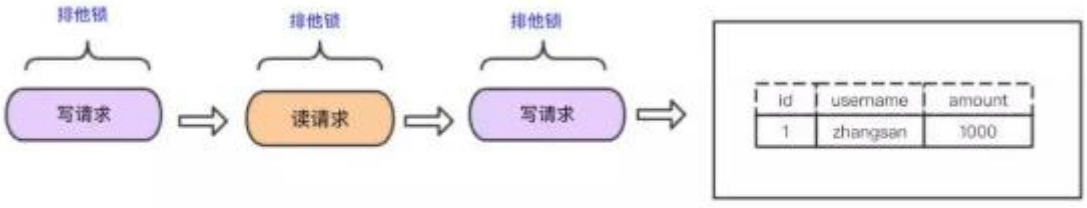
因为多次读取只生成一个版本，读到的自然是相同数据。

优点：读写并行

缺点：实现的复杂度高

（4）SERIALIZABLE

该隔离级别理解起来最简单，实现也最简单。在隔离级别下除了不会造成数据不一致问题，没其他优点。



序列化读级别下的读写锁的使用

表1-1：ANSI SQL隔离级别

隔离级别	脏读可能性	不可重复读可能性	幻读可能性	加锁读
READ UNCOMMITTED	Yes	Yes	Yes	No
READ COMMITTED	No	Yes	Yes	No
REPEATABLE READ	No	No	Yes	No
SERIALIZABLE	No	No	No	Yes

公众号：旧时光大数据

4）一致性的实现

数据库总是从一个一致性的状态转移到另一个一致性的状态。

下面举个例子，zhangsan 从银行卡转400到理财账户：

```
start transaction;
select balance from bank where name="zhangsan";
// 生成 重做日志 balance=600
update bank set balance = balance - 400;
// 生成 重做日志 amount=400
update finance set amount = amount + 400;
commit;
```

- 假如执行完 update bank set balance = balance - 400;之发生异常了，银行卡的钱也不能平白无故的减少，而是回滚到最初状态。
- 又或者事务提交之后，缓冲池还没同步到磁盘的时候宕机了，这也是不能接受的，应该在重启的时候恢复并持久化。
- 假如有并发事务请求的时候也应该做好事务之间的可见性问题，避免造成脏读，不可重复读，幻读等。在涉及并发的情况下往往在性能和一致性之间做平衡，做一定的取舍，所以隔离性也是对一致性的一种破坏。

总结

实现事务采取了哪些技术以及思想?

- 原子性：使用 undo log，从而达到回滚
- 持久性：使用 redo log，从而达到故障后恢复
- 隔离性：使用锁以及MVCC,运用的优化思想有读写分离，读读并行，读写并行
- 一致性：通过回滚，以及恢复，和在并发环境下的隔离做到一致性。

MySQL事务的特性?

可回答：1）数据库四大特性；2）事务的ACID分别是什么

问过的一些公司：字节，字节(2021.09)，美团，美团(2021.08)x2，多益，顺丰，华为精英计划(2021.07)，Shopee(2021.07)

参考答案：

分别是原子性、一致性、隔离性、持久性。

1、原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

2、一致性（Consistency）

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。举例来说，假设用户A和用户B两者的钱加起来一共是1000，那么不管A和B之间如何转账、转几次账，事务结束后两个用户的钱相加起来应该还得是1000，这就是事务的一致性。

3、隔离性（Isolation）

公众号：旧时光大数据

隔离性是当多个用户并发访问数据库时，比如同时操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。关于事务的隔离性数据库提供了多种隔离级别，稍后会介绍到。

4、持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务已经正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成。否则的话就会造成我们虽然看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。这是不允许的。

数据库事务的隔离级别？解决了什么问题？默认事务隔离级别？

可回答：1）为什么要有隔离级别？2）谈谈事务的四个隔离级别

问过的一些公司：陌陌，好未来，阿里，阿里蚂蚁(2021.08)，顺丰，51，头条，美团点评 x 4，美团(2021.08)x2，平安，ebay，小米，唯品会(2021.07)x2，字节(2021.07)-(2021.08)-(2021.09)，蔚来(2021.07)，Shopee(2021.08)-(2021.07)，好未来(2021.08)x2

参考答案：

1、事务的基本要素

原子性、一致性、隔离性、持久性

2、事务并发带来的四个问题

1）脏读（脏读脏读顾名思义是读出了问题，读了别人没有提交的数据）

假如小名明账户有100元，这里有两个事务A，B；在A事务查看账户时是100，然后这个时候B事务将100改成了200元，但是事务B还没提交，恰巧这时A又去看了自己账户发现钱变成了200元，如果这时候B回滚事务。那么读取的数据就是脏数据。这种现象总结下来就是A事务读取了B还没有提交的事务，称作脏读。

2）不可重复读（多次读取同一个数据发现结果不一致）

在一次事务中的多次查询同一数据发现查询结果不一致。

3）幻读（两次查询结果发现条数不一致）

在一个事务中可能有人提交了新的数据，导致查询结果数据量不一致。

4）数据丢失

两次更新操作可能会覆盖上一次数据的写入,导致数据丢失问题。

3、对于事务带来的问题的解决方案（四种隔离级别）

Read Uncommitted（读取未提交内容）

在一个事务中，可以读取到其他事务未提交的数据变化，这种读取其他会话还没提交的事务，叫做脏读现象，在生产环境中切勿使用。

Read Committed（读取提交内容）

在一个事务中，可以读取到其他事务已经提交的数据变化，这种读取也就叫做不可重复读，因为两次同样的查询可能会得到不一样的结果。

Repeatable Read（可重读）

公众号：旧时光大数据

MySQL默认隔离级别，在一个事务中，直到事务结束前，都可以反复读取到事务刚开始时看到的数据，并一直不会发生变化，避免了脏读、不可重复读现象，但是它还是无法解决幻读问题。

Serializable（可串行化）

这是最高的隔离级别，它强制事务串行执行，避免了前面说的幻读现象，简单来说，它会在读取的每一行数据上都加锁，所以可能会导致大量的超时和锁争用问题。

目前mysql默认使用可重复读隔离级别

脏读，幻读，不可重复读的定义

可回答：了解脏读和幻读吗

问过的一些公司：华为精英计划(2021.07)，Shopee(2021.08)

参考答案：

1）脏读（脏读脏读顾名思义是读出了问题，读了别人没有提交的数据）

假如小名明账户有100元，这里有两个事务A，B；在A事务查看账户时是100，然后这个时候B事务将100改成了200元，但是事务B还没提交，恰巧这时A又去看了自己账户发现钱变成了200元，如果这时候B回滚事务。那么读取的数据就是脏数据。这种现象总结下来就是A事务读取了B还没有提交的事务，称作脏读。

2）不可重复读（多次读取同一个数据发现结果不一致）

在一次事务中的多次查询同一数据发现查询结果不一致。

3）幻读（两次查询结果发现条数不一致）

在一个事务中可能有人提交了新的数据，导致查询结果数据量不一致。

4）数据丢失

两次更新操作可能会覆盖上一次数据的写入,导致数据丢失问题。

MySQL怎么实现可重复读?

问过的一些公司：阿里蚂蚁(2021.08)

参考答案：

MySQL默认的隔离级别是可重复读，即：事务A在读到一条数据之后，此时事务B对该数据进行了修改并提交，那么事务A再读该数据，读到的还是原来的内容。

MySQL可重复读是使用的的一种叫**MVCC**的控制方式实现的，即Mutil-Version Concurrency Control，多版本并发控制，类似于乐观锁的一种实现方式。

实现方式：

InnoDB为每行记录添加了一个事务ID，每当修改数据时，将当事务ID写入。当数据被修改时，版本号（事务ID）加1，在读取事务开始时，系统会给当前读事务一个版本号，**事务会读取版本号 <= 当前版本号的数据**。此时如果其他写事务修改了这条数据，那么这条数据的版本号就会加1，从而比当前读事务的版本号高，读事务自然而然的就读不到更新后的数据了。

公众号：旧时光大数据

数据库第三范式和第四范式区别?

可回答：1）数据库三大范式；2）数据库范式3NF、BCNF、4NF

问过的一些公司：阿里，美团x4，有赞，网易，趋势科技，腾讯，好未来，唯品会(2021.07)，携程(2021.09)，字节(2021.08)x3，兴业数金(2021.08)x2，陌陌(2021.10)，Shopee(2021.08)

参考答案：

第一范式：（字段不能重复且不能分解）

我们也叫1NF。这个范式主要还是让我们去看看表中不要存在可以被分割的列，同时表的列不能重复。当然，在实际操作过程中，我们如果录入相同的列，系统也是会报错的。

第二范式：（增加主键）

我们也叫2NF。这个范式的前提是必须要先满足第一范式的要求。当然，2NF的主要特点还是主键（从候选码挑选出来的字段，候选码是能决定唯一一行记录的属性组），所谓主键也是是能够决定一行数据的候选码。也就是说，主键可以是一列或者多列组成的，只要能够根据主键，马上能精确到特定的一行数据即可。

这里要注意的是，主键（我们有时候也会叫主属性）内存的值不能为空！

第三范式：（消除非主键的传递关系）

我们也叫3NF。这个范式的前提必须先满足第二范式的要求。第三范式主要是要看表中的非主键字段（列）与主键字段是否含有传递关系。什么叫是否有传递关系呢?

假设有一张表如下图：

	A	B	C	D	E	F
1	商品编号	商品名称	商品价格	商品类别编号	商品类别名称	商品类别描述
2						
3						
4						

这个表中的“商品类别名称”、“商品类别描述”其实是可以根据“商品类别编号”这个字段去检索到的，在这个表中具有字段传递关系。如果按照这个表去存储数据库的话，意味着要将“商品类别名称”、“商品类别描述”两个字段的数据重复很多次，使得表的空间产生严重冗余。因此，我们考虑将这个表拆分为两个表，如图所示。

	A	B	C	D
1	表1			
2	商品编号	商品名称	商品价格	商品类别编号
3				
4				
5				
6				
7				
8	表2			
9	商品类别编号	商品类别名称	商品类别描述	
10				
11				
12				
13				

这样建立数据表，就符合了数据库第三范式3NF规范。如果我们想要在表内单独增加一个商品类别也相当方便，假设我们系统想要显示出来我们的商品类别，那么就更方便了。

公众号：旧时光大数据

在实际开发中，我们的系统一般符合3NF就可以了，但是在实际工作生产过程中，为了优化我们的系统性能，有时候可能会牺牲数据空间换取工作性能，最终部分表的关系只能符合2NF。这种情况也是非常正常的。

BC范式：（消除主键内的传递关系）

这个范式也叫BCNF。这个范式的前提条件是要先满足第三范式的要求。在BC范式中，比起第三范式来说还多了一个主键内部传递关系的检查。我们举个例子，看图中的表：

	A	B	C	D
1	表1			
2	店铺	店长	商品名称	商品价格
3				
4		划线的三个字段是主键		
5				
6				

从这个表中，我们可以看出，商品价格是非主属性，店铺、店长、商品名称是主属性（主键），我们可以根据三个字段作为主键去确定找到某个商品的价格。

现在，作为老板的你，想要增加一家店铺，店铺也有商品，但是还没有招聘到店长，这个时候怎么办呢？如果按照以上3NF的要求设计的表，就会无法录入信息到表，因为主键是不能为空的。

所以，我们需要重新设计这个数据表，把它变成符合BCNF的表。即从主键中再次进行分解成其他的表。重新设计后，我们表如下：

	A	B	C	D
1	表1			
2	店铺	店长	商品名称	商品价格
3				
4				
5				
6				
7	表2			
8	店铺	店长		
9				
10				
11				
12				
13	表3			
14	店铺	商品名称		
15				
16				
17				
18				

这样设计就符合BCNF

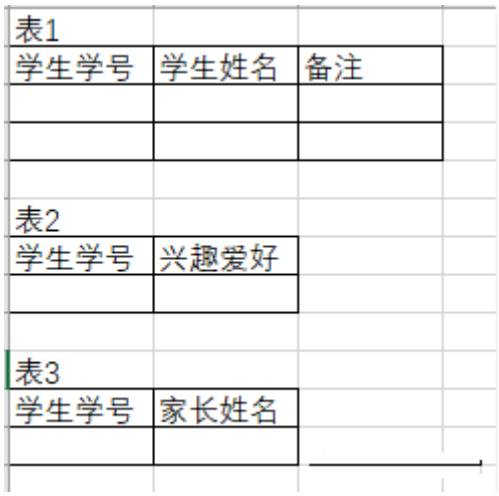
第四范式：（消除一个表内的多个多值）

我们也叫做4NF。这个范式的设计我们需要先满足BC要求的前提要求。在4NF中最为特别的就是在一个表内要消除掉多个多值情况。我们还是举个例子，如下表中存在多值的情况。

表1				
学生学号	学生姓名	兴趣爱好	家长姓名	备注

公众号：旧时光大数据

首先，上表的设计是符合BC范式的，但我们也能明显看到一个学生肯定会有多个兴趣爱好的情况，一个学生也会有多个家长。因此，我们也可以这个表调整成4NF规范。下图所示。



第五范式：（消除非候选码的表字段连接依赖）

这个范式我们也叫5NF。这个范式首先前提必须要满足4NF。第五范式是指关系模型R依赖均有R候选码所隐含，这是指在连接时，所连接的属性均为候选码。这个是几近于完美的范式，对字段关系要求极高，但是可能会消耗数据库很大性能。这里不做举例了。主要强调一下，5NF主要就是表连接时注意，只能是候选码才能连接才可以。

MySQL的存储引擎?

问过的一些公司：字节，百度(2021.08)

参考答案：

1、数据库存储引擎

数据库存储引擎是数据库底层软件组织，数据库管理系统（DBMS）使用数据引擎进行创建、查询、更新和删除数据。不同的存储引擎提供不同的存储机制、索引技巧、锁定水平等功能，使用不同的存储引擎，还可以获得特定的功能。现在许多不同的数据库管理系统都支持多种不同的数据引擎。

因为在关系数据库中数据的存储是以表的形式存储的，所以存储引擎也可以称为表类型(Table Type，即存储和操作此表的类型)。

如创建一个InnoDB类型的表：

```
CREATE TABLE `brand` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `brand_name` varchar(64) NOT NULL,
  `brand_logo` varchar(255) NOT NULL,
  `description` varchar(255) NOT NULL,
  `parent_id` int(11) NOT NULL,
  `layer` int(11) NOT NULL DEFAULT '0',
  `enabled` int(11) NOT NULL DEFAULT '0',
  `path` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `brand_name` (`brand_name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

2、MySQL存储引擎

公众号：旧时光大数据

MySQL中常用的四种存储引擎分别是：**MyISAM**、**InnoDB**、**MEMORY**、**ARCHIVE**。MySQL 5.5版本后默认的存储引擎为`InnoDB`。

MySQL给开发者提供了查询存储引擎的功能，执行以下sql即可查询到mysql中的存储引擎

```
SHOW ENGINES
```

3、InnoDB存储引擎

InnoDB是事务型数据库的首选引擎，**InnoDB是目前MySQL的默认事务型引擎**，是目前最重要、使用最广泛的存储引擎。**支持事务安全表（ACID），支持行锁定和外键。**

优点：支持事务和崩溃修复能力；引入了行级锁和外键约束。

缺点：占用的数据空间相对较大。

InnoDB主要特性有：

1）InnoDB给MySQL提供了具有提交、回滚和崩溃恢复能力的事物安全（ACID兼容）存储引擎。InnoDB锁定在行级并且也在SELECT语句中提供一个类似Oracle的非锁定读。这些功能增加了多用户部署和性能。在SQL查询中，可以自由地将InnoDB类型的表和其他MySQL的表类型混合起来，甚至在同一个查询中也可以混合

2）InnoDB是为处理巨大数据量的最大性能设计。它的CPU效率可能是任何其他基于磁盘的关系型数据库引擎锁不能匹敌的

3）InnoDB存储引擎完全与MySQL服务器整合，InnoDB存储引擎为在主内存中缓存数据和索引而维持它自己的缓冲池。InnoDB将它的表和索引在一个逻辑表空间中，表空间可以包含数个文件（或原始磁盘文件）。这与MyISAM表不同，比如在MyISAM表中每个表被存放在分离的文件中。InnoDB表可以是任何尺寸，即使在文件尺寸被限制为2GB的操作系统上

4）InnoDB支持外键完整性约束，存储表中的数据时，每张表的存储都按主键顺序存放，如果没有显示在表定义时指定主键，InnoDB会为每一行生成一个6字节的ROWID，并以此作为主键

5）InnoDB被用在众多需要高性能的大型数据库站点上

InnoDB不创建目录，使用InnoDB时，MySQL将在MySQL数据目录下创建一个名为ibdata1的10MB大小的自动扩展数据文件，以及两个名为ib_logfile0和ib_logfile1的5MB大小的日志文件。

适用场景：需要事务支持，并且有较高的并发读写频率。

4、MyISAM存储引擎

数据以紧密格式存储。对于只读数据，或者表比较小、可以容忍修复操作，可以使用MyISAM引擎。MyISAM会将表存储在两个文件中，数据文件`.MYD`和索引文件`.MYI`。

优点：访问速度快。

缺点：MyISAM不支持事务和行级锁，不支持崩溃后的安全恢复，也不支持外键。

MyISAM主要特性有：

1）大文件（达到63位文件长度）在支持大文件的文件系统和操作系统上被支持

2）当把删除和更新及插入操作混合使用的时候，动态尺寸的行产生更少碎片。这要通过合并相邻被删除的块，以及若下一个块被删除，就扩展到下一块自动完成

3）每个MyISAM表最大索引数是64，这可以通过重新编译来改变。每个索引最大的列数是16

4）最大的键长度是1000字节，这也可以通过编译来改变，对于键长度超过250字节的情况，一个超过1024字节的键将被用上

5）BLOB和TEXT列可以被索引，支持FULLTEXT类型的索引，而InnoDB不支持这种类型的索引

公众号：旧时光大数据

6）NULL被允许在索引的列中，这个值占每个键的0~1个字节

7）所有数字键值以高字节优先被存储以允许一个更高的索引压缩

8）每个MyISAM类型的表都有一个AUTO_INCREMENT的内部列，当INSERT和UPDATE操作的时候该列被更新，同时AUTO_INCREMENT列将被刷新。所以说，MyISAM类型表的AUTO_INCREMENT列更新比InnoDB类型的AUTO_INCREMENT更快

9）可以把数据文件和索引文件放在不同目录

10）每个字符列可以有不同的字符集

11）有VARCHAR的表可以固定或动态记录长度

12）VARCHAR和CHAR列可以多达64KB

存储格式：

1）静态表（默认）：字段都是非变长的（每个记录都是固定长度的）。存储非常迅速、容易缓存，出现故障容易恢复；占用空间通常比动态表多。

2）动态表：占用的空间相对较少，但是频繁的更新删除记录会产生碎片，需要定期执行optimize table或myisamchk -r命令来改善性能，而且出现故障的时候恢复比较困难。

3）压缩表：使用myisampack工具创建，占用非常小的磁盘空间。因为每个记录是被单独压缩的，所以只有非常小的访问开支。

静态表的数据在存储的时候会按照列的宽度定义补足空格，在返回数据给应用之前去掉这些空格。如果需要保存的内容后面本来就有空格，在返回结果的时候也会被去掉。（其实是数据类型char的行为，动态表中若有这个数据类型也同样会有这个问题）

使用MyISAM引擎创建数据库，将产生3个文件。文件的名字以表名字开始，扩展名之处文件类型：frm文件存储表定义、数据文件扩展名为.MYD（MYData）、索引文件扩展名为.MYI（MYIndex）。

适用场景：如果表主要是用于插入新记录和读出记录，那么选择MyISAM能实现处理高效率。

5、MERGE存储引擎

MERGE存储引擎是一组MyISAM表的组合，这些MyISAM表结构必须完全相同，尽管其使用不如其它引擎突出，但是在某些情况下非常有用。说白了，Merge表就是几个相同MyISAM表的聚合器；Merge表中并没有数据，对Merge类型的表可以进行查询、更新、删除操作，这些操作实际上是对内部的MyISAM表进行操作。

场景：对于服务器日志这种信息，一般常用的存储策略是将数据分成很多表，每个名称与特定的时间端相关。例如：可以用12个相同的表来存储服务器日志数据，每个表用对应各个月份的名字来命名。当有必要基于所有12个日志表的数据来生成报表，这意味着需要编写并更新多表查询，以反映这些表中的信息。与其编写这些可能出现错误的查询，不如将这些表合并起来使用一条查询，之后再删除Merge表，而不影响原来的数据，删除Merge表只是删除Merge表的定义，对内部的表没有任何影响。

6、MEMORY存储引擎

MEMORY引擎将数据全部放在内存中，访问速度较快，但是一旦系统奔溃的话，数据都会丢失。

MEMORY引擎默认使用哈希索引，将键的哈希值和指向数据行的指针保存在哈希索引中。

优点：访问速度较快。

缺点：

- 哈希索引数据不是按照索引值顺序存储，无法用于排序。
- 不支持部分索引匹配查找，因为哈希索引是使用索引列的全部内容来计算哈希值的。
- 只支持等值比较，不支持范围查询。

公众号：旧时光大数据

- 当出现哈希冲突时，存储引擎需要遍历链表中所有的行指针，逐行进行比较，直到找到符合条件的行。

MEMORY主要特性有：

1）MEMORY表的每个表可以有多达32个索引，每个索引16列，以及500字节的最大键长度

2）MEMORY存储引擎执行HASH和BTREE缩影

3）可以在一个MEMORY表中有非唯一键值

4）MEMORY表使用一个固定的记录长度格式

5）MEMORY不支持BLOB或TEXT列

6）MEMORY支持AUTO_INCREMENT列和对可包含NULL值的列的索引

7）MEMORY表在所由客户端之间共享（就像其他任何非TEMPORARY表）

8）MEMORY表内存被存储在内存中，内存是MEMORY表和服务器在查询处理时的空闲中，创建的内部表共享

9）当不再需要MEMORY表的内容时，要释放被MEMORY表使用的内存，应该执行DELETE FROM或TRUNCATE TABLE，或者删除整个表（使用DROP TABLE）

MEMORY存储引擎默认使用哈希（HASH）索引，其速度比使用B-+Tree型要快，但也可以使用B树型索引。由于这种存储引擎所存储的数据保存在内存中，所以其保存的数据具有不稳定性，比如如果mysqld进程发生异常、重启或计算机关机等等都会造成这些数据的消失，所以这种存储引擎中的表的生命周期很短，一般只使用一次。现在mongodb、redis等NOSQL数据库愈发流行，MEMORY存储引擎的使用场景越来越少。

场景：如果需要该数据库中一个用于查询的临时表。

7、ARCHIVE存储引擎

Archive是归档的意思，在归档之后很多的高级功能就不再支持了，仅仅支持最基本的插入和查询两种功能。在MySQL 5.5版以前，Archive是不支持索引，但是在MySQL 5.5以后的版本中就开始支持索引了。**Archive拥有很好的压缩机制**，它使用zlib压缩库，在记录被请求时会实时压缩，所以它经常被用来当做仓库使用。

场景：由于**高压缩**和**快速插入**的特点Archive非常适合作为日志表的存储引擎（适合存储大量独立的、作为历史记录的数据），但是前提是不经常对该表进行查询操作（这种引擎不支持索引，所以查询性能较差）。

8、CSV存储引擎

使用该引擎的MySQL数据库表会在MySQL安装目录data文件夹中的和该表所在数据库名相同的目录中生成一个.CSV文件（所以，它可以将CSV类型的文件当做表进行处理），这种文件是一种普通文本文件，每个数据行占用一个文本行。该种类型的存储引擎不支持索引，即使用该种类型的表没有主键列；另外也不允许表中的字段为null。csv的编码转换需要格外注意。

场景：这种引擎支持从数据库中拷入/拷出CSV文件。如果从电子表格软件输出一个CSV文件，将其存放在MySQL服务器的数据目录中，服务器就能够马上读取相关的CSV文件。同样，如果写数据库到一个CSV表，外部程序也可以立刻读取它。在实现某种类型的日志记录时，CSV表作为一种数据交换格式，特别有用。

9、BLACKHOLE存储引擎（黑洞引擎）

该存储引擎支持事务，而且支持mvcc的行级锁，写入这种引擎表中的任何数据都会消失，主要用于做日志记录或同步归档的中继存储，这个存储引擎除非有特别目的，否则不适合使用。

公众号：旧时光大数据

场景：如果配置一主多从的话，多个从服务器会在主服务器上分别开启自己相对应的线程，执行binlogdump命令而且多个此类进程并不是共享的。为了避免因多个从服务器同时请求同样的事件而导致主机资源耗尽，可以单独建立一个伪的从服务器或者叫分发服务器。

10、PERFORMANCE_SCHEMA存储引擎

该引擎主要用于收集数据库服务器性能参数。这种引擎提供以下功能：提供进程等待的详细信息，包括锁、互斥变量、文件信息；保存历史的事件汇总信息，为提供MySQL服务器性能做出详细的判断；对于新增和删除监控事件点都非常容易，并可以随意改变mysql服务器的监控周期，例如（CYCLE、MICROSECOND）。 MySQL用户是不能创建存储引擎为PERFORMANCE_SCHEMA的表。

场景： DBA能够较明细得了解性能降低可能是由于哪些瓶颈。

11、Federated存储引擎

该存储引擎可以不同的Mysql服务器联合起来，逻辑上组成一个完整的数据库。这种存储引擎非常适合数据库分布式应用。

Federated存储引擎可以使你在本地数据库中访问远程数据库中的数据，针对federated存储引擎表的查询会被发送到远程数据库的表上执行，本地是不存储任何数据的。

缺点：

1）对本地虚拟表的结构修改，并不会修改远程表的结构

2）truncate 命令，会清除远程表数据

3）drop命令只会删除虚拟表，并不会删除远程表

4）不支持 alter table 命令

5）select count(*), select * from limit M, N 等语句执行效率非常低，数据量较大时存在很严重的问题，但是按主键或索引列查询，则很快，如以下查询就非常慢（假设 id 为主索引）

```
1    select id from db.tablea where id >100 limit 10 ;
```

而以下查询就很快：

```
1    select id from db.tablea where id >100 and id<150;
```

6）如果虚拟虚拟表中字段未建立索引，而实体表中为此字段建立了索引，此种情况下，性能也相当差。但是当给虚拟表建立索引后，性能恢复正常。

7）类似 where name like "str%" limit 1 的查询，即使在 name 列上创建了索引，也会导致查询过慢，是因为federated引擎会将所有满足条件的记录读取到本地，再进行 limit 处理。

场景： dblink。

12、存储引擎的选择

公众号：旧时光大数据

功 能	MYISAM	Memory	InnoDB	Archive
存储限制	256TB	RAM	64TB	None
支持事物	No	No	Yes	No
支持全文索引	Yes	No	No	No
支持数索引	Yes	Yes	Yes	No
支持哈希索引	No	Yes	No	No
支持数据缓存	No	N/A	Yes	No
支持外键	No	No	Yes	No

数据库有哪些锁?

问过的一些公司：滴滴，ebay，蔚来(2021.07)

参考答案：

1、锁

锁是网络数据库中的一个非常重要的概念，当多个用户同时对数据库并发操作时，会带来数据不一致的问题，所以，锁主要用于多用户环境下保证数据库完整性和一致性。

数据库锁出现的目的：处理并发问题

并发控制的主要采用的技术手段：乐观锁、悲观锁和时间戳。

2、锁分类

从数据库系统角度分为三种：排他锁、共享锁、更新锁。

从程序员角度分为两种：一种是悲观锁，一种乐观锁。

3、悲观锁（Pessimistic Lock）

顾名思义，很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人拿这个数据就会block（阻塞），直到它拿锁。

悲观锁（Pessimistic Lock）：正如其名，具有强烈的独占和排他特性。它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

传统的关系数据库里用到了很多这种锁机制，比如行锁、表锁、读锁、写锁等，都是在操作之前先上锁。

悲观锁按使用性质划分

公众号：旧时光大数据

1）共享锁（Share Lock）

S锁，也叫读锁，用于所有的只读数据操作。共享锁是非独占的，允许多个并发事务读取其锁定的资源。

性质

- 多个事务可封锁同一个共享页；
- 任何事务都不能修改该页；
- 通常是该页被读取完毕，S锁立即被释放。

在SQL Server中，默认情况下，数据被读取后，立即释放共享锁。

例如，执行查询语句“SELECT * FROM my_table”时，首先锁定第一页，读取之后，释放对第一页的锁定，然后锁定第二页。这样，就允许在读操作过程中，修改未被锁定的第一页。

例如，语句“SELECT * FROM my_table HOLDLOCK”就要求在整个查询过程中，保持对表的锁定，直到查询完成才释放锁定。

2）排他锁（Exclusive Lock）

X锁，也叫写锁，表示对数据进行写操作。如果一个事务对对象加了排他锁，其他事务就不能再给它加任何锁了。（某个顾客把试衣间从里面反锁了，其他顾客想要使用这个试衣间，就只有等待锁从里面打开了。）

性质

- 仅允许一个事务封锁此页；
- 其他任何事务必须等到X锁被释放才能对该页进行访问；
- X锁一直到事务结束才能被释放。

产生排他锁的SQL语句如下：select * from ad_plan for update;

3）更新锁

U锁，在修改操作的初始化阶段用来锁定可能要被修改的资源，这样可以避免使用共享锁造成的死锁现象。

性质

- 用来预定要对此页施加X锁，它允许其他事务读，但不允许再施加U锁或X锁；
- 当被读取的页要被更新时，则升级为X锁；
- U锁一直到事务结束时才能被释放。

悲观锁按作用范围划分为：行锁、表锁。

1）行锁

锁的作用范围是行级别。

2）表锁

锁的作用范围是整张表。

数据库能够确定那些行需要锁的情况下使用行锁，如果不知道会影响哪些行的时候就会使用表锁。

举个例子，一个用户表user，有主键id和用户生日birthday。

当你使用update … where id=?这样的语句时，数据库明确知道会影响哪一行，它就会使用行锁；

当你使用update … where birthday=?这样的的语句时，因为事先不知道会影响哪些行就可能会使用表锁。

4、乐观锁（Optimistic Lock）

公众号：旧时光大数据

顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以，不会上锁。但是在更新的时候会判断一下在此期间别人有没有更新这个数据，可以使用版本号等机制。

乐观锁（ Optimistic Locking ）： 相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。

悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。而乐观锁机制在一定程度上解决了这个问题。

乐观锁，大多是基于数据版本（ Version ）记录机制实现。

数据版本：为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个 “version” 字段来实现。读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write_condition机制的其实都是提供的乐观锁。

乐观锁的实现方式

1）版本号（version）

就是给数据增加一个版本标识，在数据库上就是表中增加一个version字段，每次更新把这个字段加1，读取数据的时候把version读出来，更新的时候比较version，如果还是开始读取的version就可以更新了，如果现在的version比老的version大，说明有其他事务更新了该数据，并增加了版本号，这时候得到一个无法更新的通知，用户自行根据这个通知来决定怎么处理，比如重新开始一遍。这里的关键是判断version和更新两个动作需要作为一个原子单元执行，否则在你判断可以更新以后正式更新之前有别的事务修改了version，这个时候你再去更新就可能会覆盖前一个事务做的更新，造成第二类丢失更新，所以你可以使用update … where … and version=”old version”这样的语句，根据返回结果是0还是非0来得到通知，如果是0说明更新没有成功，因为version被改了，如果返回非0说明更新成功。

2）时间戳（使用数据库服务器的时间戳）

和版本号基本一样，只是通过时间戳来判断而已，注意时间戳要使用数据库服务器的时间戳不能是业务系统的时间。

3）待更新字段

和版本号方式相似，只是不增加额外字段，直接使用有效数据字段做版本控制信息，因为有时候我们可能无法改变旧系统的数据库表结构。假设有个待更新字段叫count,先去读取这个count,更新的时候去比较数据库中count的值是不是我期望的值（即开始读的值），如果是就把我修改的count的值更新到该字段，否则更新失败。java的基本类型的原子类型对象如AtomicInteger就是这种思想。

4）所有字段

和待更新字段类似，只是使用所有字段做版本控制信息，只有所有字段都没变化才会执行更新。

乐观锁几种方式的区别

新系统设计可以使用version方式和timestamp方式，需要增加字段，应用范围是整条数据，不论那个字段修改都会更新version,也就是说两个事务更新同一条记录的两个不相关字段也是互斥的，不能同步进行。旧系统不能修改数据库表结构的时候使用数据字段作为版本控制信息，不需要新增字段，待更新字段方式只要其他事务修改的字段和当前事务修改的字段没有重叠就可以同步进行，并发性更高。

并发控制会造成两种锁

- 活锁
- 死锁

并发控制会造成活锁和死锁，就像操作系统那样，会因为互相等待而导致。

公众号：旧时光大数据

1）活锁

定义：指的是T1封锁了数据R，T2同时也请求封锁数据R，T3也请求封锁数据R，当T1释放了锁之后，T3会锁住R，T4也请求封锁R，则T2就会一直等待下去。

解决方法：采用“先来先服务”策略可以避免。

2）死锁

定义：就是我等你，你又等我，双方就会一直等待下去。比如：T1封锁了数据R1，正请求对R2封锁，而T2封住了R2，正请求封锁R1，这样就会导致死锁，死锁这种没有完全解决的方法，只能尽量预防。

预防方法：

- 一次封锁法，指的是一次性把所需要的数据全部封锁住，但是这样会扩大了封锁的范围，降低系统的并发度；
- 顺序封锁法，指的是事先对数据对象指定一个封锁顺序，要对数据进行封锁，只能按照规定的顺序来封锁，但是这个一般不大可能的。

系统判定死锁的方法：

- 超时法：如果某个事物的等待时间超过指定时限，则判定为出现死锁；
- 等待图法：如果事务等待图中出现了回路，则判断出现了死锁。

对于解决死锁的方法，只能是撤销一个处理死锁代价最小的事务，释放此事务持有的所有锁，同时对撤销的事务所执行的数据修改操作必须加以恢复。

说下悲观锁、乐观锁

问过的一些公司：字节(2021.07)

参考答案：

乐观锁：对于数据冲突保持一种乐观态度，操作数据时不会对操作的数据进行加锁，只有到数据提交的时候才通过一种机制来验证数据是否存在冲突。

悲观锁：对于数据冲突保持一种悲观态度，在修改数据之前把数据锁住，然后再对数据进行读写，在它释放锁之前任何人都不能对其数据进行操作，直到前面一个人把锁释放后下一个人数据加锁才可对数据进行加锁，然后才可以对数据进行操作，一般数据库本身锁的机制都是基于悲观锁的机制实现的。

分布式数据库是什么?

问过的一些公司：滴滴

参考答案：

1、分布式数据库是什么?

分布式数据库系统：一个粗略的定义是“分布式数据库由一组数据组成，这些数据物理上分布在计算机网络的不同节点上（亦称场地）上，逻辑上是属于同一个系统。”

这里强调两点：

- 分布性：数据库中的数据不是存储在同一场地，更确切的说，不存储在同一计算机的存储设备上，这就可以和集中式数据库相互区别。
- 逻辑整体性：这些数据逻辑上是互相联系的，是一个整体（逻辑上如同集中数据库）。

公众号：旧时光大数据

精确的分布式数据库定义：分布式数据库是由一组数据组成的，这组数据分布在计算机网络中的不同的计算机上，网络中的每个节点具有独立处理的能力（称为场地自治），可以执行局部应用。同时，每个节点也能通过网络通信子系统执行全局应用。与之前的定义相比，更注重场地自治性以及自治场地之间的协作性。

2、分布式数据库系统的特点

分布式数据库系统是在集中式数据库系统技术的基础上发展起来的。它具有自己的特性和特征。集中式数据库的许多概念和技术，如数据独立性、数据共享性和减少冗余度、并发控制、完整性、安全性和恢复等。

1）数据独立性

在集中式数据库系统中，数据独立性包括两个方面：数据的逻辑独立性和数据的物理独立性。其含义是用户程序与数据的全局逻辑结构及数据的存储结构无关。在分布式数据库系统中除了数据的逻辑独立性和物理独立性外，还有数据分布独立性亦称为分布透明性。分布透明性是指用户不必关心数据的逻辑分片，不必关心数据物理位置分布的细节，也不必关心重复副本（冗余数据问题）一致性问题，同时也不必关心局部场地上数据库支持哪种数据模型。

2）集中于自治相结合的控制结构

数据库是多个用户共享的资源，在集中式数据库系统中，为了保证数据库的安全性和完整性，对共享数据库的控制是集中的，并有DBA负责监督和维护系统的正常运行。

在分布式数据库系统中，数据的共享有两个层次：

- 局部共享：即在局部数据库中存储局部场地各用户的共享数据，这些数据是本场地用户常用的。
- 全局共享：即在分布式数据库系统的各个场地也存储其它场地的用户共享的数据，支持系统的全局应用。

因此，相应的控制机构也具有两个层次：集中和自治。

3）适当增加数据冗余度

在集中式数据库系统中，尽量减少冗余度是系统目标之一。其原因是，冗余数据不仅浪费空间，而且容易造成各数据副本之间的不一致性，为了保证数据的一致性，系统要付出一定的维护代价，减少冗余度的目标是用数据共享来达到的。

而在分布式数据系统中却希望存储必要的荣誉数据，在不同的场地存储同一数据的多个副本，其原因是：

- 提高系统的可靠性、可用性：当某一场地出现故障时，系统可以对另一场地上的相同副本进行操作，不会因为一处故障而造成整个系统的瘫痪。
- 提高系统性能：系统可以选择用户最近的数据副本进行操作，减少通信代价，改善整个系统的性能。冗余副本之间数据不一致的问题是分布式数据库系统必须着力解决的问题。

4）全局一致性、可串行性和可恢复性

分布式数据库系统中个局部数据库应满足集中式数据库的一致性、并发事务的可串行性和可恢复性。除此以外还应保证数据库的全局一致性、全局并发食物的可串行性和系统的全局可恢复性。

死锁产生的条件是什么？如何预防死锁？

可回答：死锁是什么，如何破坏死锁

问过的一些公司：多益，陌陌(2021.10)

参考答案：

公众号：旧时光大数据

1、死锁（Deadlock）

所谓死锁：是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

由于资源占用是互斥的，当某个进程提出申请资源后，使得有关进程在无外力协助下，永远分配不到必需的资源而无法继续运行，这就产生了一种特殊现象死锁。

一种情形，此时执行程序中两个或多个线程发生永久堵塞（等待），每个线程都在等待被其他线程占用并堵塞了的资源。例如，如果线程A锁住了记录1并等待记录2，而线程B锁住了记录2并等待记录1，这样两个线程就发生了死锁现象。

计算机系统中,如果系统的资源分配策略不当，更常见的可能是程序员写的程序有错误等，则会导致进程因竞争资源不当而产生死锁的现象。

锁有多种实现方式，比如意向锁，共享 - 排他锁，锁表，树形协议，时间戳协议等等。锁还有多种粒度，比如可以在表上加锁，也可以在记录上加锁。

产生死锁的原因主要是：

- 系统资源不足。
- 进程运行推进的顺序不合适。
- 资源分配不当等。

如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则就会因争夺有限的资源而陷入死锁。其次，进程运行推进顺序与速度不同，也可能产生死锁。

产生死锁的四个必要条件：

- 互斥条件：一个资源每次只能被一个进程使用。
- 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
- 不剥夺条件:进程已获得的资源，在未使用完之前，不能强行剥夺。
- 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

2、死锁的避免与预防

死锁避免的基本思想：系统对进程发出每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，如果分配后系统可能发生死锁,则不予分配，否则予以分配。这是一种保证系统不进入死锁状态的动态策略。

理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。只要打破四个必要条件之一就能有效预防死锁的发生：

打破互斥条件：改造独占性资源为虚拟资源，大部分资源已无法改造。

打破不可抢占条件：当一进程占有一独占性资源后又申请一独占性资源而无法满足，则退出原占有的资源。

打破占有且申请条件：采用资源预先分配策略，即进程运行前申请全部资源，满足则运行，不然就等待，这样就不会占有且申请。

打破循环等待条件：实现资源有序分配策略，对所有设备实现分类编号，所有进程只能采用按序号递增的形式申请资源。

3、死锁避免和死锁预防的区别

公众号：旧时光大数据

死锁预防是设法至少破坏产生死锁的四个必要条件之一，严格的防止死锁的出现；而死锁避免则不那么严格的限制产生死锁的必要条件的存在，因为即使死锁的必要条件存在，也不一定发生死锁。死锁避免是在系统运行过程中注意避免死锁的最终发生。

介绍下数据库的join（内连接、外连接、全连接），内连接和外连接（左、右连接）的区别

可回答：sql的left join和inner join的区别?

问过的一些公司：字节，字节(2021.08)，猿辅导，美团x3，网易云音乐

参考答案：

sql中的连接查询有inner join(内连接）、left join(左连接)、right join（右连接）、full join（全连接）四种方式，它们之间其实并没有太大区别，仅仅是查询出来的结果有所不同。

例如有以下两张表：

"Persons" 表：

Id_P	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

"Orders" 表：

Id_O	OrderNo	Id_P
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	65

Orders表通过外键Id_P和Persons表进行关联。

1、inner join

在两张表进行连接查询时，只保留两张表中完全匹配的结果集。

sql如下：

```
ELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
INNER JOIN Orders
ON Persons.Id_P=Orders.Id_P
ORDER BY Persons.LastName
```

查询结果集：

公众号：旧时光大数据

LastName	FirstName	OrderNo
Adams	John	22456
Adams	John	24562
Carter	Thomas	77895
Carter	Thomas	44678

此种连接方式Orders表中Id_P字段在Persons表中找不到匹配的，则不会列出来。

2、left join

在两张表进行连接查询时，会返回左表所有的行，即使在右表中没有匹配的记录。

sql如下：

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
LEFT JOIN Orders
ON Persons.Id_P=Orders.Id_P
ORDER BY Persons.LastName
```

查询结果如下：

LastName	FirstName	OrderNo
Adams	John	22456
Adams	John	24562
Carter	Thomas	77895
Carter	Thomas	44678
Bush	George	

可以看到，左表（Persons表）中LastName为Bush的行的Id_P字段在右表（Orders表）中没有匹配，但查询结果仍然保留该行。

3、right join

在两张表进行连接查询时，会返回右表所有的行，即使在左表中没有匹配的记录。

sql如下：

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
RIGHT JOIN Orders
ON Persons.Id_P=Orders.Id_P
ORDER BY Persons.LastName
```

查询结果如下：

LastName	FirstName	OrderNo
Adams	John	22456
Adams	John	24562
Carter	Thomas	77895
Carter	Thomas	44678
		34764

Orders表中最后一条记录Id_P字段值为65，在左表中没有记录与之匹配，但依然保留。

公众号：旧时光大数据

4、full join

在两张表进行连接查询时，返回左表和右表中所有没有匹配的行。

sql如下：

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
FULL JOIN Orders
ON Persons.Id_P=Orders.Id_P
ORDER BY Persons.LastName
```

查询结果如下：

LastName	FirstName	OrderNo
Adams	John	22456
Adams	John	24562
Carter	Thomas	77895
Carter	Thomas	44678
Bush	George	
		34764

查询结果是left join和right join的并集。

MySQL的join过程

问过的一些公司：百度

参考答案：

使用join的通用结构如下：

```
SELECT <row_list>
  FROM <left_table>
    <inner|left|right> JOIN <right_table>
      ON <join condition>
        WHERE <where_condition>
```

执行顺序如下(SQL语句里第一个被执行的总是FROM子句)：

- FROM：对左右两张表执行**笛卡尔积**，产生第一张表vt1。行数为n*m（n为左表的行数，m为右表的行数）；
- ON：根据ON的条件逐行筛选vt1，将结果插入vt2中；
- JOIN：**添加外部行**，如果指定了LEFT JOIN(LEFT OUTER JOIN)，则先遍历一遍左表的每一行，其中不在vt2的行会被插入到vt2，该行的剩余字段将被填充为NULL，形成vt3；如果指定了RIGHT JOIN也是同理。但如果指定的是INNER JOIN，则不会添加外部行，上述插入过程被忽略，vt2=vt3（所以**INNER JOIN的过滤条件放在ON或WHERE里 执行结果是没有区别的**）；
- WHERE：对vt3进行条件过滤，满足条件的行被输出到vt4；
- SELECT：取出vt4的指定字段到vt5。

join过程大体是**笛卡尔积->on条件过滤->添加外部行->where条件->select**

公众号：旧时光大数据

MySQL有哪些存储引擎?

可回答：了解数据库的什么引擎

问过的一些公司：美团(2021.08)

参考答案：MyISAM、InnoDB、Memory、Archive

功 能	MYISAM	Memory	InnoDB	Archive
存储限制	256TB	RAM	64TB	None
支持事物	No	No	Yes	No
支持全文索引	Yes	No	No	No
支持数索引	Yes	Yes	Yes	No
支持哈希索引	No	Yes	No	No
支持数据缓存	No	N/A	Yes	No
支持外键	No	No	Yes	No

数据库中存储引擎MyISAM与InnoDB的区别

可回答：MySQL有什么存储引擎，有什么区别

问过的一些公司：字节，字节(2021.08)-(2021.09)，好未来，百度，美团点评，美团(2021.08)x4，美团买菜(2021.09)x2，Shopee(2021.07)，招银网络(2021.09)

参考答案：

区别：

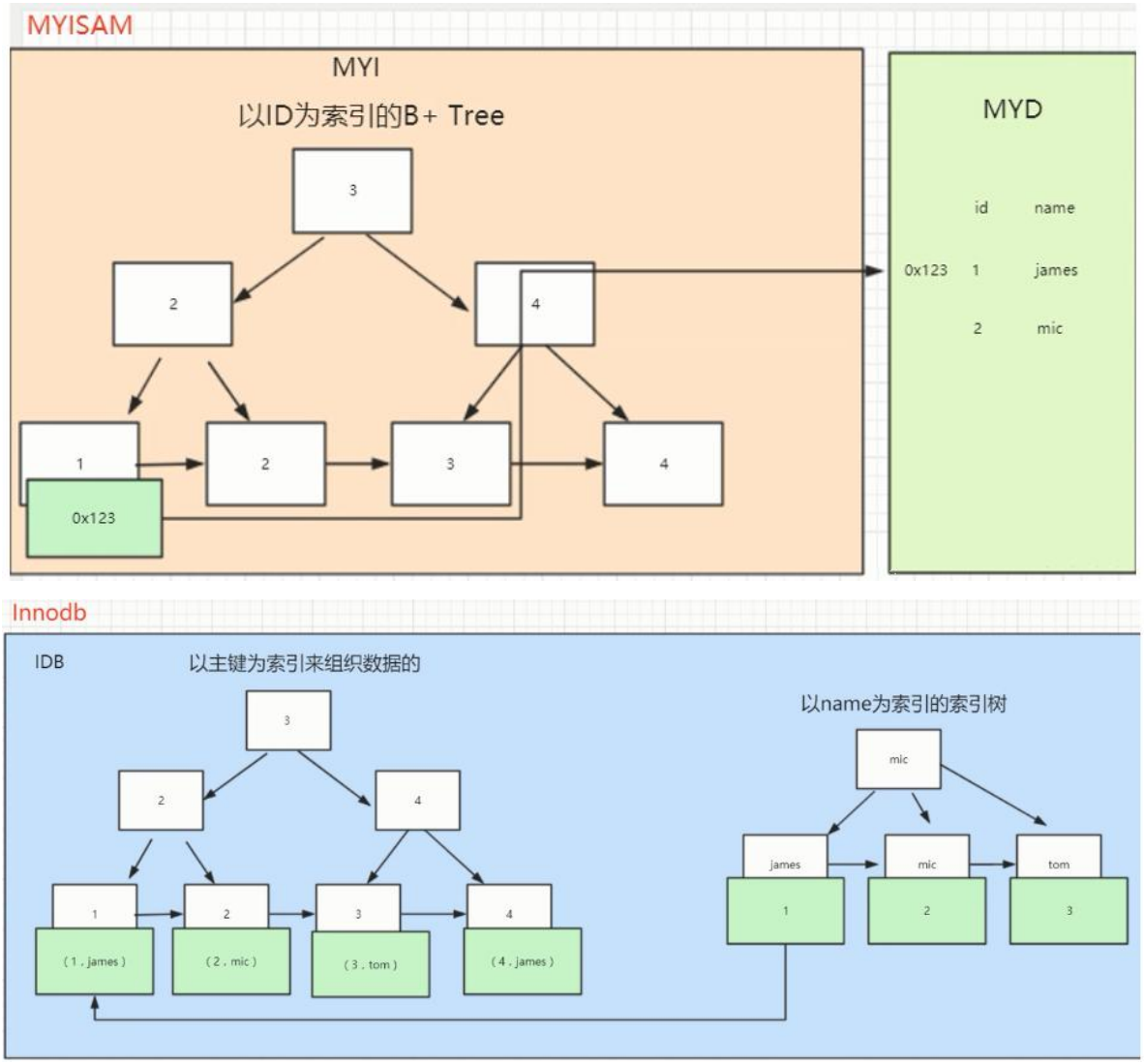
1、InnoDB支持事务，MyISAM不支持，对于InnoDB每一条SQL语言都默认封装成事务，自动提交，这样会影响速度，所以最好把多条SQL语言放在begin和commit之间，组成一个事务；

2、InnoDB支持外键，而MyISAM不支持。对一个包含外键的InnoDB表转为MYISAM会失败；

3、InnoDB是聚集索引，使用B+Tree作为索引结构，数据文件是和（主键）索引绑在一起的（表数据文件本身就是按B+Tree组织的一个索引结构），必须要有主键，通过主键索引效率很高。但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，因为主键太大，其他索引也都会很大。

MyISAM是非聚集索引，也是使用B+Tree作为索引结构，索引和数据文件是分离的，索引保存的是数据文件的指针。主键索引和辅助索引是独立的。

也就是说：InnoDB的B+树主键索引的叶子节点就是数据文件，辅助索引的叶子节点是主键的值；而MyISAM的B+树主键索引和辅助索引的叶子节点都是数据文件的地址指针。



4、InnoDB不保存表的具体行数，执行select count(*) from table时需要全表扫描。而MyISAM用一个变量保存了整个表的行数，执行上述语句时只需要读出该变量即可，速度很快（注意不能加有任何WHERE条件）；

那么为什么InnoDB没有了这个变量呢?

因为InnoDB的事务特性，在同一时刻表中的行数对于不同的事务而言是不一样的，因此count统计会计算对于当前事务而言可以统计到的行数，而不是将总行数储存起来方便快速查询。InnoDB会尝试遍历一个尽可能小的索引除非优化器提示使用别的索引。如果二级索引不存在，InnoDB还会尝试去遍历其他聚簇索引。

如果索引并没有完全处于InnoDB维护的缓冲区（Buffer Pool）中，count操作会比较费时。可以建立一个记录总行数的表并让你的程序在INSERT/DELETE时更新对应的数据。和上面提到的问题一样，如果此时存在多个事务的话这种方案也不太好用。如果得到大致的行数值已经足够满足需求可以尝试SHOW TABLE STATUS。

5、Innodb不支持全文索引，而MyISAM支持全文索引，在涉及全文索引领域的查询效率上MyISAM速度更快高；

注意：5.7以后的InnoDB支持全文索引了

6、MyISAM表格可以被压缩后进行查询操作

7、InnoDB支持表、行(默认)级锁，而MyISAM支持表级锁

InnoDB的行锁是实现在索引上的，而不是锁在物理行记录上。潜台词是，如果访问没有命中索引，也无法使用行锁，将要退化为表锁。

公众号：旧时光大数据

8、InnoDB表必须有唯一索引（如主键）（用户没有指定的话会自己找/生产一个隐藏列Row_id来充当默认主键），而Myisam可以没有

9、Innodb存储文件有frm、ibd，而Myisam是frm、MYD、MYI

- Innodb：frm是表定义文件，ibd是数据文件
- Myisam：frm是表定义文件，myd是数据文件，myi是索引文件

如何选择：

- 是否要支持事务，如果要请选择innodb，如果不需要可以考虑MyISAM；
- 如果表中绝大多数都只是读查询，可以考虑MyISAM，如果既有读也有写，请使用InnoDB。
- 系统奔溃后，MyISAM恢复起来更困难，能否接受；
- MySQL5.5版本开始Innodb已经成为Mysql的默认引擎（之前是MyISAM），说明其优势是有目共睹的，如果你不知道用什么，那就用InnoDB，至少不会差。

MyIsam适用于什么场景?

问过的一些公司：美团(2021.08)

参考答案：

1）做很多count 的计算；

2）插入不频繁，查询非常频繁；

3）没有事务。

InnoDB和MyIsam针对读写场景?

问过的一些公司：美团(2021.08)

参考答案：

如果表中绝大多数都只是读查询，可以考虑MyISAM，如果既有读也有写，请使用InnoDB。

MySQL Innodb实现了哪个隔离级别?

问过的一些公司：唯品会(2021.07)

参考答案：

在InnoDB存储引擎中，Repeatable Read 是默认的事务隔离级别。

同时该引擎的实现基于多版本的并发控制协议——MVCC (Multi-Version Concurrency Control)，解决了幻读问题，脏读和不可重复读也是不存在的。MVCC最大的好处就在于读不加锁，读写不冲突，这样极大的增加了系统的并发性能。

InnoDB数据引擎的特点

问过的一些公司：字节(2021.08)

参考答案：

公众号：旧时光大数据

1）**支持事务**（事务是指逻辑上的一组操作，组成这组操作的各个单元，要么全成功，要么全失败）

2）**行级锁定**（更新时一般是锁定当前行）：通过索引实现，全表扫描仍然会是锁定整个表，注意间隙锁的影响

3）**读写阻塞与事务隔离级别相关**

4）**具有非常高效的缓存特性**，能缓存索引，也能缓存数据

5）整个**表和主键以Cluster方式存储**，组成一颗平衡树

6）所有**Secondary Index都会保存主键信息**

7）**支持分区，表空间。**类似于Oracle数据库

8）**支持外键约束，不支持全文索引**，5.5之前支持，后面不再支持

9）和MyISAM相比，**InnoDB对于硬件资源要求比较高**

InnoDB用什么索引

问过的一些公司：字节(2021.08)，贝壳找房(2021.11)

参考答案：

聚集索引，使用B+Tree作为索引结构。

InnoDB的主键索引与行记录是存储在一起的，故叫做聚集索引（Clustered Index）：没有单独区域存储行记录。

必须要有主键，通过主键索引效率很高。但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，因为主键太大，其他索引也都会很大。

Hash索引缺点

问过的一些公司：字节(2021.08)

参考答案：

缺点：

1）不能避免读取行

哈希索引只包含哈希值和行指针，而不存储字段值，所以不能使用索引中的值来避免读取行。不过，访问内存中的行的速度很快，所以大部分情况下这一点对性能的影响并不明显。

2）无法用于排序

哈希索引数据并不是按照索引值顺序存储的，所以也就无法用于排序。

3）无法使用部分索引列匹配查找

哈希索引也不支持部分索引列匹配查找，因为哈希索引始终是使用索引列的全部内容来计算哈希值的。例如，在数据列（A,B）上建立哈希索引，如果查询只有数据列A，则无法使用该索引。

4）只支持等值查找

哈希索引只支持等值比较查询，包括=、IN()、<=>（注意<>和<=>是不同的操作）。也不支持任何范围查询，例如WHERE price>100。

公众号：旧时光大数据

5）存在Hash冲突

访问哈希索引的数据非常快，除非有很多哈希冲突（不同的索引列值却有相同的哈希值）。当出现哈希冲突的时候，存储引擎必须遍历链表中所有的行指针，逐行进行比较，直到找到所有符合条件的行。

同时，当哈希冲突很多的时候，一些索引维护操作的代价也会很高。例如，如果在某个选择性很低（哈希冲突很多）的列上建立哈希索引，那么当从表中删除一行时，存储引擎需要遍历对应哈希值的链表中的每一行，找到并删除对应行的引用，冲突越多，代价越大。

优点：

因为索引自身只需存储对应的哈希值，所以索引的结构十分紧凑，这也让**哈希索引查找的速度非常快**

数据库索引的类型，各有什么优缺点?

问过的一些公司：字节x2，百度，阿里，美团点评，陌陌，妙盈科技，美团x6，美团(2021.09)，猿辅导，顺丰，有赞x2，蔚来(2021.07)，Shopee(2021.07)，兴业数金(2021.09)

参考答案：

索引的几种类型分别是普通索引、唯一索引、聚集索引、主键索引、全文索引几种。

使用索引的优点：

- 提高数据的搜索速度
- 加快表与表之间的连接速度
- 在信息检索过程中，若使用分组及排序子句进行时，通过建立索引能有效的减少检索过程中所需的分组及排序时间，提高检索效率。

使用索引的缺点:

- 在我们建立数据库的时候，需要花费的时间去建立和维护索引，而且随着数据量的增加，需要维护它的时间也会增加。
- 在创建索引的时候会占用存储空间。
- 在我们需要修改表中的数据时，索引还需要进行动态的维护，所以对数据库的维护带来了一定的麻烦。

唯一索引：数据列不允许重复，允许为NULL值，一个表允许多个列创建唯一索引。

主键索引：数据列不允许重复，不允许为NULL值，一个表只能有一个主键。

聚集索引：我们在表中添加数据的顺序，与我们创建的索引键值相同，而且一个表中只能有一个聚集索引。

普通索引：基本的索引类型，没有唯一性的限制，允许为NULL值。它的结构主要以B+树和哈希索引为主，主要是对数据表中的数据进行精确查找。

全文索引：是目搜索引擎使用的一种关键技术，它的作用是搜索数据表中的字段是不是包含我们搜索的关键字，就像搜索引擎中的模糊查询。

MySQL的索引有哪些？索引如何优化?

可回答：1）MySQL索引建过吗？用过哪些索引？什么类型的？2）索引用法，优化方法？3）索引的作用，索引的实现？4）MySQL的索引有哪些？谈谈索引？为什么不用哈希表？5）数据库索引有什么作用?

公众号：旧时光大数据

问过的一些公司：头条，百度，美团点评，360，小米，vivo，拼多多，顺丰，阿里，映客直播，蚂蚁金服，字节(2021.07)-(2021.08)，阿里蚂蚁(2021.08)，友塔游戏(2021.08)，蔚来(2021.09)x2，兴业数金(2021.08)，网易有道(2021.09)，网易(2021.09)，欢聚(2021.09)，茄子科技(2021.09)，Shopee(2021.07)

参考答案：

1、索引是什么

索引，在MySQL中也叫“键（key）”，是存储引擎用于快速找到记录的一种数据结构。如果把数据库的一张表比作一本书，那索引则是这本书的目录，通过目录，我们能快速找到我们想要的主题所对应的页码。**索引的作用即类似于书的目录，帮助我们快速定位到相关数据行的位置。**

好的索引能使查询的性能提高几个数量级，而差的索引在大数据量的表中甚至会使性能急剧下降。“最优”的索引有时比一个“好的”索引性能要好两个数量级。

2、MySQL的索引

索引有很多种类型，可以为不同的场景提供更好的性能。在MySQL中，索引是在存储引擎层而非服务器层实现的，而不同的存储引擎的索引的工作方式并不一样，且不是所有的存储引擎都支持所有类型的索引。同时，值得一提的是，不同的存储引擎对同一类型的索引，其底层的实现一般是不同的。

MySQL支持以下几种类型的索引：

- B-Tree索引
- 哈希索引
- 空间数据索引（R-Tree）
- 全文索引
- 其他索引类别

3、B-Tree索引

B-Tree索引是最常见的索引类型，它使用B-Tree数据结构来存储数据，大多数MySQL引擎都支持这种索引。（Archive引擎是一个例外：5.1之前Archive不支持任何索引，直到5.1才开始支持单个自增列AUTO_INCREMENT的索引。）

在MySQL中，“B-Tree”只是一个术语的统称，因为不同的存储引擎可能使用的是其他存储结构来实现这种索引，但仅仅只是命名为“B-Tree”。例如，**NDB集群存储引擎**内部实际上使用了**T-Tree结构**存储这种索引；**InnoDB**则使用的是**B+Tree结构**存储这种索引。只是它们都将其命名为“B-Tree”。

1）B-Tree索引在不同引擎中的差异

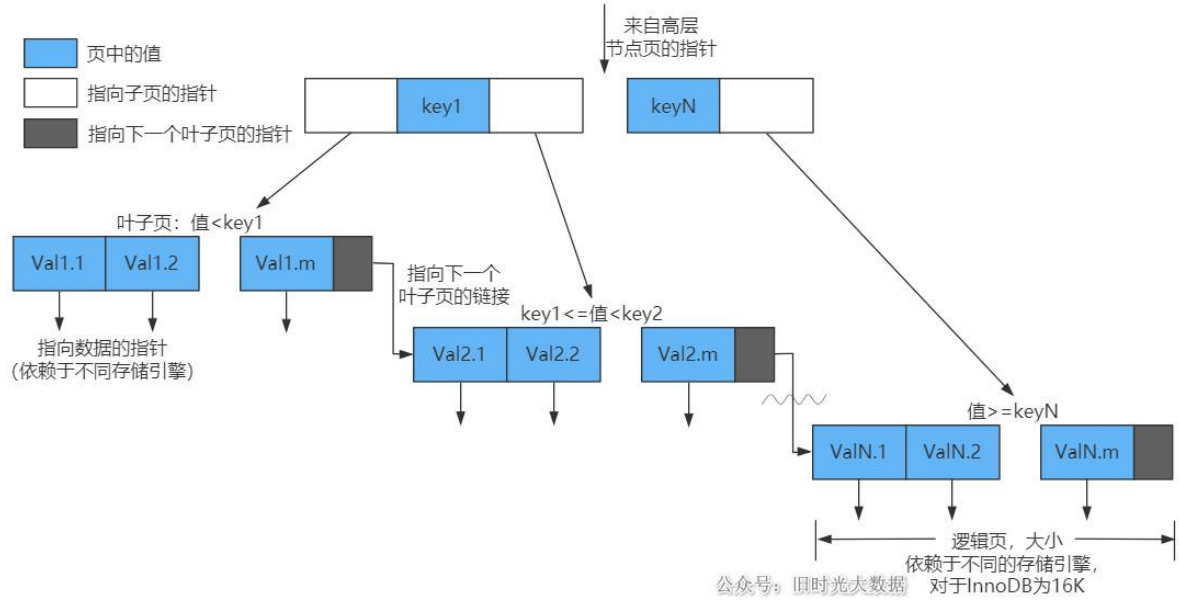
不同的存储引擎使用B-Tree索引的方式也不同，性能也各有不同，各有优劣。下面拿MyISAM 和InnoDB进行对比。

	InnoDB	MyISAM
存储方式	前缀压缩技术	按照原数据格式
引用方式	通过数据的物理位置引用被索引的行	根据主键引用被索引的行

2）InnoDB的B-Tree技术实现是B+Tree

InnoDB的B-Tree索引从技术上来说实际上是B+Tree实现的，这种实现使得所有的值都是按照顺序存储的（所以很适合查找范围数据），并且每一个叶子页到根的距离相同。MyISAM使用的结构有所不同，但基本思想类似。

公众号：旧时光大数据



B-Tree索引能够加快访问数据的速度，靠的就是上面这种数据结构。它使得存储引擎不再需要进行全表扫描来获取所需数据，取而代之的是从索引的根节点开始搜索。

根节点中存放了指向子节点的指针，存储引擎根据这些指针向下层查找。通过比较节点页的值和要查找的值可以找到合适的指针进入下层子节点，这些指针实际上定义了子节点页中值的上限和下限。最终存储引擎要么能找到对应的值，要么该记录不存在。叶子节点页有相应的指针，但叶子节点的指针不是指向其他的节点页，而是指向被索引的数据（不同引擎的“指针”类型不同）。

树的深度和表的大小直接相关，表的数据量越大，树的层数越多。

3）创建一个多列索引

```
CREATE TABLE People (
      last_name    varchar(50)        not null,
      first_name   varchar(50)        not null,
      dob          date             not null,
      gender       enum('m','f')    not null,
      key(last_name,first_name,dob)
);
```

4）B-Tree索引支持的查询类型

MySQL的B-Tree索引适用于全键值、键值范围或键前缀查找，其中键前缀查找只适用于根据最左前缀的查找。前面所述的索引可细分为如下几种类型。

（1）全值匹配

全值匹配指的是和索引中的所有列进行匹配。

例如上面的People表的索引（last_name,first_name,dob）可以用于查找last_name='Zeng',first_name='Chuang',dob='1996-01-01'的人。这就是使用了索引中的所有列进行匹配，即全值匹配。

（2）匹配最左前缀

可以只使用索引的第一个列进行匹配。

例如可以用于查找last_name='Zeng'的人，即用于查找姓为Zeng的人，这里只使用了索引的最左列进行匹配，即匹配最左前缀。

（3）匹配列前缀

可以只匹配某一列的值的开头部分。

公众号：旧时光大数据

例如可以用于查找last_name LIKE ‘Z%’的人，即用于查找所有以Z开头的姓的人，这里只使用了索引最左列的前缀进行匹配，即匹配列前缀。

（4）匹配范围值

可以只适用索引的第一列查找符合某个范围内的数据。

例如可以用于查找last_name BETWEEN ‘Qiu’ AND ‘Zeng’的人，即用于查找姓在Qiu和Zeng之间的人，这里只使用了索引最左列的前缀进行范围匹配，即匹配范围值。

（5）精确匹配某一列并范围匹配另外一列

可以使第一列全匹配，第二列范围匹配。

例如可以用于查找last_name=’Zeng’ AND first_name LIKE ’C%’的人，即用于查找姓是Zeng，名字以C开头的人，这里使用了索引的最左列精确匹配，第二列进行范围匹配，即精确匹配某一列并范围匹配另外一列。

（6）只访问索引的查询

查询只需访问索引，而无须访问数据行。

例如select last_name, first_name where last_name=’Zeng’; 这里只查询索引所包含的last_name和first_name列，则无须读取数据行。

5）B-Tree索引的限制

（1）只能按照索引的最左列开始查找。

例如People表中的索引无法用于查找first_name为’Chuang’的人，也无法查找某个特定生日的人，因为这两个列都不是最左数据列。

（2）只能按照索引最左列的最左前缀进行匹配。

例如People表中的索引无法查找last_name LIKE ‘%eng’的人，虽然last_name就是此索引的最左列，但MySQL索引无法查找以‘eng’结尾的last_name的记录。

（3）只能按照索引定义的顺序从左到右进行匹配，不能跳过索引中的列。

例如People表中的索引无法用于查找last_name=’Zeng’ AND bod=’1996-01-01’的人，因为MySQL无法跳过索引中的某一列而使用索引中最左列和排在末尾的列进行组合。如果不指定索引中中间的列，则MySQL只能使用索引的最左列，即第一列。

（4）如果查询中有某个列的范围查询，则其右边所有列都无法使用索引优化查找。

例如有这样一个查询：where last_name=’Zeng’ AND first_name LIKE ’C%’ AND dob=’1996-01-01’; 这个查询只能使用索引的前两列，因为这里LIKE是一个范围条件，则first_name后面的索引列都将失效。（优化点：尽量不要在索引列中使用LIKE等范围条件，改用多个等于条件来替代，保证后面的索引列能生效。）

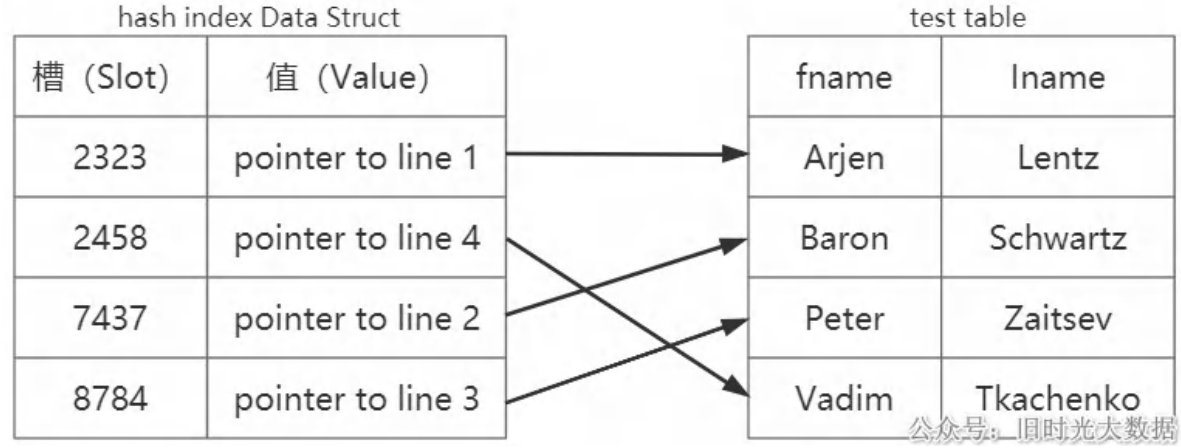
4、哈希索引

1）哈希索引是什么

哈希索引（hash index）基于哈希表实现，只有精确匹配索引所有列的查询才有效。

对于每一行数据，存储引擎都会对所有的索引列计算一个哈希码（hash code），不同键值的行计算出来的哈希码也不一样。哈希索引将所有的哈希码存储在索引中，并在哈希表中保存指向每个数据行的指针。

公众号：旧时光大数据

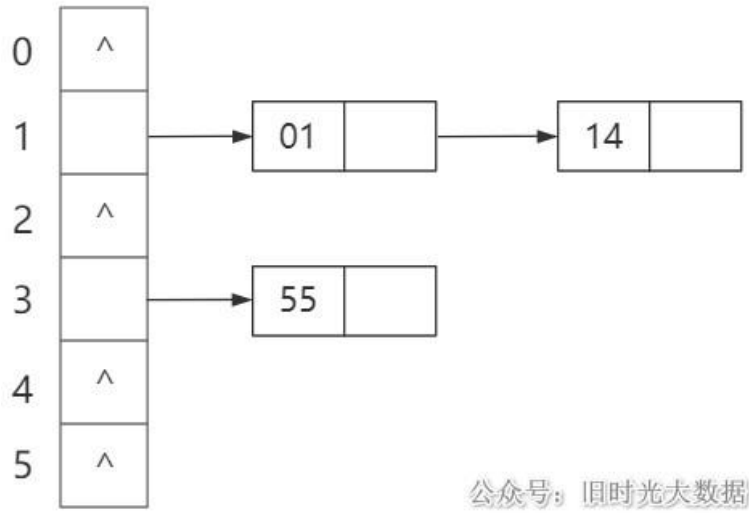


2）不同存储引擎对哈希索引的支持

	Memory引擎	NDB集群引擎	InnoDB引擎
是否支持哈希索引	显示支持，是Memory引擎表的默认索引类型（也支持B-Tree索引）	支持唯一哈希索引，所起作用特殊	自适应哈希索引（adaptive hash index）

（1） Memory引擎的哈希索引

Memory引擎不仅支持唯一哈希索引，还支持非唯一哈希索引。非唯一哈希索引指的是：如果多个列的哈希值相同，索引会以链表的方式存放多个指向不同记录的指针到同一个哈希条目中。



使用Memory引擎在建表时创建哈希索引

```
CREATE TABLE testhash (
    fname VARCHAR(50) NOT NULL,
    Lname VARCHAR(50) NOT NULL,
    KEY USING HASH(fname)
)ENGINE=MEMORY;
```

（2）NDB集群引擎的哈希索引

（3）InnoDB引擎的哈希索引

InnoDB引擎有一个特殊的功能叫“自适应哈希索引（adaptive hash index）”：当InnoDB注意到某些索引值被使用得非常频繁时，它会在内存中基于B-Tree索引之上再创建一个哈希索引，让B-Tree索引也具有哈希索引的一些优点，如快速的哈希查找。

公众号：旧时光大数据

“自适应哈希索引”是一个完全自动的、内部的行为，用户无法控制或配置，不过如果有必要，完全可以关闭该功能。

3）哈希索引的优势、限制及适用场景

（1）优势：哈希索引查找的速度非常快。

原因：索引自身只需存储对应的哈希值，使得索引的结构十分紧凑。

（2）哈希索引的限制

- 不能使用索引中的值来避免读取行。因为哈希索引只包含哈希值和行指针，而不存储字段值。
- 不能用于排序。因为哈希索引数据不是按照索引值顺序存储的。
- 不支持部分索引列匹配查找。因为哈希索引必须使用索引列的全部内容来计算哈希值。
- 只支持等值比较查询（包括=、IN()、<=>），不支持任何范围查询（如where price > 100)。
- 当出现哈希冲突时（不同的索引列值却有相同的哈希值），访问速度会变慢。因为存储引擎必须遍历链表中所有的行指针，逐行进行比较，知道找到所有符合条件的行。
- 若哈希冲突很多，一些索引维护操作的代价也会很高。如：在某个哈希冲突很多的列上建立哈希索引，当从表中删除一行时，存储引擎需要遍历对应哈希值的链表中的每一行，找到并删除对应行的引用。*冲突越多，代价越大*。

（3）哈希索引的适用场景

因为上面的这些限制，使得哈希索引适用的场景比较有限。而一旦适用哈希索引，则它带来的性能提升将非常显著。如，在数据仓库应用中有一种经典的“星型”schema，需要许多关联才能建立查找表，哈希索引就非常适合查找表的需求。

4）创建自定义哈希索引

（1）思路

在B-Tree基础上创建一个伪哈希索引。这和真正的哈希索引不是一回事，因为还是使用B-Tree进行查找，但是它使用哈希值而不是键本身进行索引查找。只需要在查询的WHERE字句中手动指定使用哈希函数。

（2）哈希索引查找实例

一张表中存储了大量的URL，并需要根据URL进行搜索查找。如果使用B-Tree来存储URL，存储的内容会很大。正常情况下会有如下查询：

```
1   SELECT id FROM url WHERE url="http://www.mysql.com";
```

若删除原来URL列上的索引，而新增一个被索引的url_crc列，使用CRC32做哈希，就可以使用下面的方式查询：

```
1   SELECT id FROM url WHERE url="http://www.mysql.com" AND
    url_crc=CRC32("http://www.mysql.com");
```

这样做性能会非常高，因为MySQL优化器会使用这个选择性很高而体积很小的基于url_crc列的索引来完成查找。及时有多个记录相同的索引值，查找仍然很快，因为MySQL优化器会先筛选出匹配的索引行记录，然后根据具体的url值进行比对，返回完全符合条件的行。

（3）使用触发器维护哈希值

新增一列url_crc列之后需要维护这个哈希值。可以手动维护，也可以使用触发器实现。

首先创建如下表：

公众号：旧时光大数据

```
CREATE TABLE pseudohash (
  id  int  unsigned  NOT  NULL  auto_increment,
  url  varchar(255)  NOT NULL,
  url_crc  int unsigned  NOT NULL DEFAULT 0,
  PRIMARY KEY(id)
);
```

然后创建触发器：

```
DELIMITER //
CREATE TRIGGER pseudohash_crc_ins BEFORE INSERT ON pseudohash FOR EACH ROW BEGIN
SET NEW.url_crc=crc32(NEW.url);
END;
//
CREATE TRIGGER pseudohash_crc_upd BEFORE UPDATE ON pseudohash FOR EACH ROW BEGIN
SET NEW.url_crc=crc32(NEW.url);
END;
//
DELIMITER ;


```

最后验证触发器如何维护哈希索引：

```
INSERT INTO pseudohash(url) VALUES('http://www.mysql.com');
SELECT * FROM pseudohash;

UPDATE pseudohash SET url='http://www.mysql.com' WHERE id=1;
SELECT * FROM pseudohash;


```

（4）规避使用SHA1()和MD5()

SHA1()和MD5()计算出来的哈希值是非常长的字符串，会浪费大量空间，比较时也会更慢。SAH1()和MD5()是强加密函数，设计目标是最大限度消除冲突，但这里并不需要这样高的要求。

简单哈希函数的冲突在一个可以接受的范围，同时又能够提供更好的性能。

（5）处理哈希冲突

1.定义哈希函数

如果表的数据量非常大，CRC32()会出现大量的哈希冲突，则可以考虑自己实现一个简单的64位哈希函数。这个自定义函数要返回整数，而不是字符串。一个简单的办法可以使用MD5()返回值的一部分来作为自定义哈希函数。这可能比自己写一个哈希算法的性能要查，但这样实现最简单：

```
SELECT CONV(RIGHT(MD5('http://www.mysql.com'),16),16,10) AS HASH64;


```

注：CONV(N,from_base,to_base)

N是要转换的数据，from_base是原进制，to_base是目标进制。

2.WHERE字句中包含常量值

当使用哈希索引进行查询的时候，必须在WHERE字句中包含常量值：

公众号：旧时光大数据

```
SELECT id FROM url WHERE url="http://www.mysql.com" AND
url_crc=CRC32("http://www.mysql.com");
```

因为所谓的"生日悖论"，出现哈希冲突的概率的增长速度可能比想象的要快得多。CRC32()返回的是32位的整数，当索引有93000条记录时出现冲突的概率是1%。

如果不想查询具体值，例如只是统计记录数（不精确的），则可以不带入列值，直接使用CRC32()的哈希值查询即可。

还可以使用如FNV64()函数作为哈希函数，这是移植自Percona Server的函数，可以以插件的方式在任何MySQL版本中使用，哈希值为64位，速度快，且冲突比CRC32()要少很多。

5、空间数据索引（R-Tree）

MyISAM表支持空间索引，可以用作地理数据存储。和B-Tree索引不同，这类索引无须前缀索引。空间索引会从所有维度来索引数据。查询时，可以有效地使用任意维度来组合查询。必须使用MySQL的GIS相关函数如MBRCONTAINS()等来维护数据。MySQL的GIS支持并不完善，所以大部分人都不会使用这个特性。

开源关系数据库系统中对GIS的解决方案做得比较好的是PostgreSQL和PostGIS。

6、全文索引

全文索引是一种特殊类型的索引，它查找的是文本中的关键词，而不是直接比较索引中的值。全文搜索和其他几类索引的匹配方式完全不一样。它有许多需要注意的细节，如停用词、词干和复数、布尔搜索等。全文索引更类似于搜索引擎做的事情，而不是简单的WHERE条件匹配。

7、其他索引类型

TokuDB使用形树索引（fractal tree index），这是一类较新开发的数据结构，既有B-Tree的很多优点，也避免了B-Tree的一些缺点。

还有InnoDB的聚簇索引、覆盖索引等。

ScaleDB使用Patricia tries。

其他存储引擎技术如InfiniDB和Infobright则使用了一些特殊的数据结构来优化某些特殊的查询。

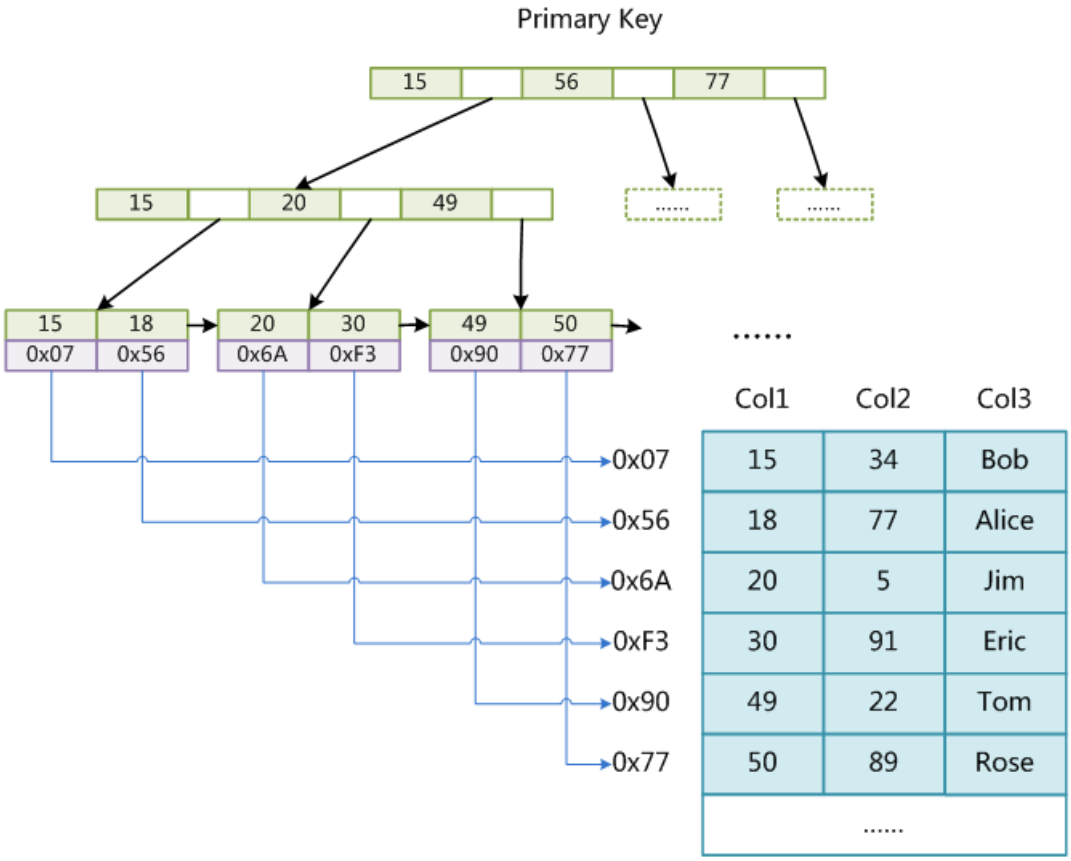
8、MySQL索引实现

在MySQL中，索引属于存储引擎级别的概念，不同存储引擎对索引的实现方式是不同的，接下来主要说下MyISAM和InnoDB两个存储引擎的索引实现方式。

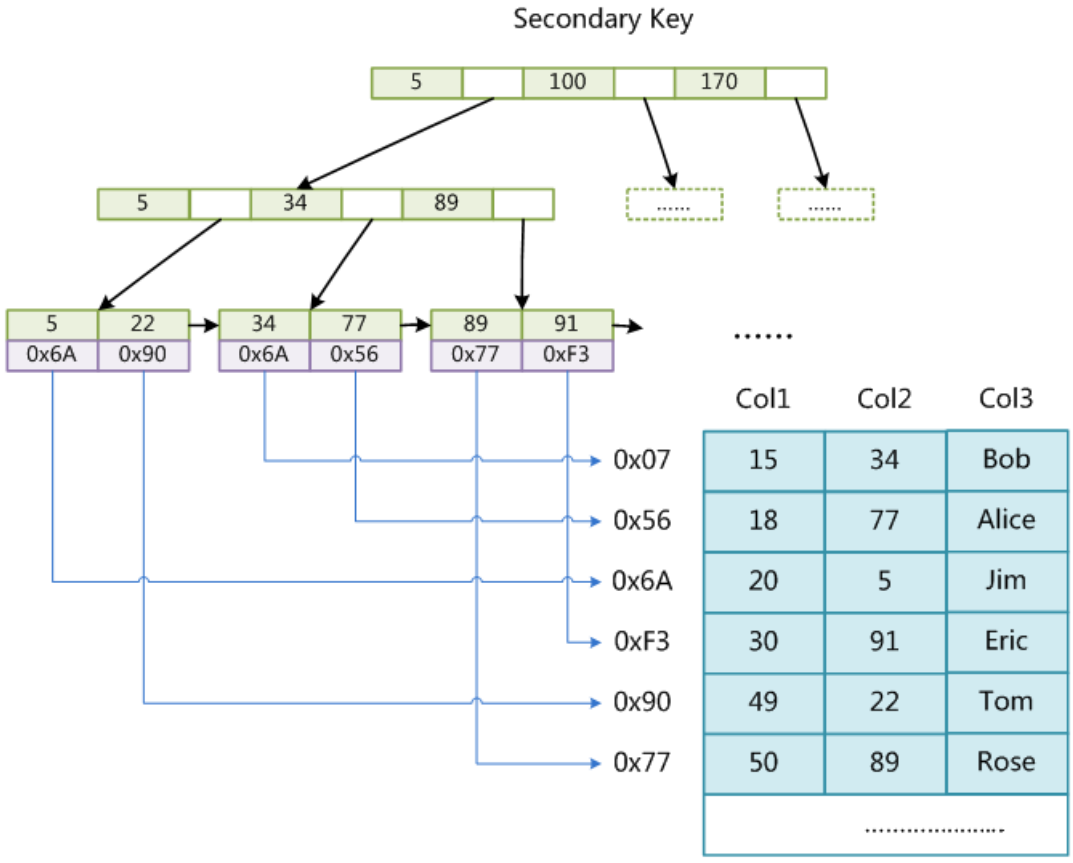
1）MyISAM索引实现

MyISAM引擎使用B+Tree作为索引结构，叶节点的data域存放的是数据记录的地址。下图是MyISAM索引的原理图：

公众号：旧时光大数据



这里设表一共有三列，假设我们以Col1为主键，则上图是一个MyISAM表的主索引（Primary key）示意。可以看出MyISAM的索引文件仅仅保存数据记录的地址。在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复。如果我们在Col2上建立一个辅助索引，则此索引的结构如下图所示：



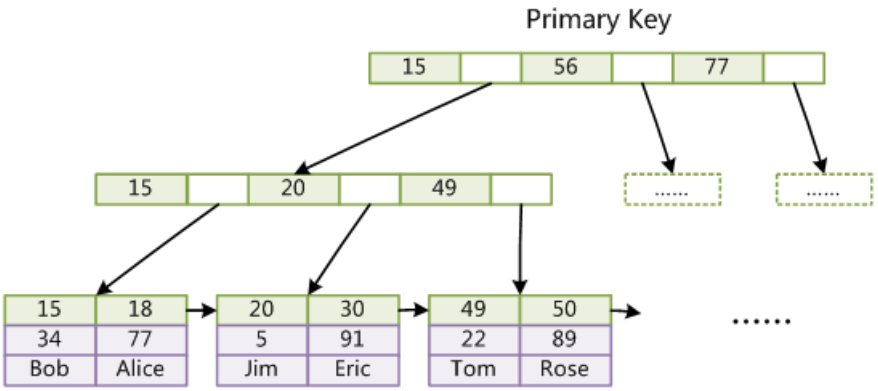
同样也是一颗B+Tree，data域保存数据记录的地址。因此，MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值为地址，读取相应数据记录。

MyISAM的索引方式也叫做“非聚集”的，之所以这么称呼是为了与InnoDB的聚集索引区分。

2）InnoDB索引实现

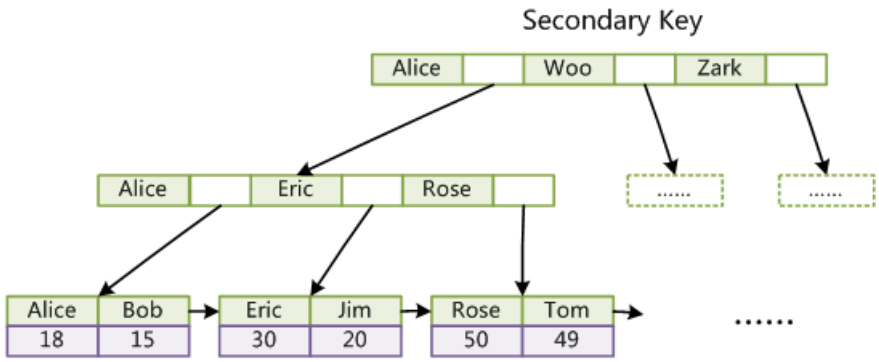
虽然InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同。

第一个重大区别是InnoDB的数据文件本身就是索引文件。从上文知道，MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。



上图是InnoDB主索引（同时也是数据文件）的示意图，可以看到叶节点包含了完整的数据记录。这种索引叫做聚集索引。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。

第二个与MyISAM索引的不同是InnoDB的辅助索引data域存储相应记录主键的值而不是地址。换句话说，InnoDB的所有辅助索引都引用主键作为data域。例如，下图为定义在Col3上的一个辅助索引：



这里以英文字符的ASCII码作为比较准则。聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。

了解不同存储引擎的索引实现方式对于正确使用和优化索引都非常有帮助，例如知道了InnoDB的索引实现后，就很容易明白为什么不建议使用过长的字段作为主键，因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。再例如，用非单调的字段作为主键在InnoDB中不是个好主意，因为InnoDB数据文件本身是一颗B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个很好的选择。

9、索引优化

- 最左前缀匹配原则
- 主键外检一定要建索引
- 对 where,on,group by,order by 中出现的列使用索引
- 尽量选择区分度高的列作为索引,区分度的公式是count(distinct col)/count(*)，表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些状态、性别字段可能在大数据

公众号：旧时光大数据

面前区分度就是0

- 对较小的数据列使用索引,这样会使索引文件更小,同时内存中也可以装载更多的索引键
- 索引列不能参与计算，保持列“干净”，比如from_unixtime(create_time) = ’2014-05-29’就不能使用到索引，原因很简单，b+树中存的都是数据表中的字段值，但进行检索时，需要把所有元素都应用函数才能比较，显然成本太大。所以语句应该写成create_time = unix_timestamp(’2014-05-29’);
- 为较长的字符串使用前缀索引
- 尽量的扩展索引，不要新建索引。比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可
- 不要过多创建索引,权衡索引个数与DML之间关系，DML也就是插入、删除数据操作。这里需要权衡一个问题，建立索引的目的是为了提高查询效率的，但建立的索引过多，会影响插入、删除数据的速度，因为我们修改的表数据，索引也需要进行调整重建
- 对于like查询，”%”不要放在前面。

  ```
  SELECT * FROM houdunwang WHERE uname LIKE'后盾%' -- 走索引
  SELECT * FROM houdunwang WHERE uname LIKE "%后盾%" -- 不走索引
  ```

- 查询where条件数据类型不匹配也无法使用索引

  字符串与数字比较不使用索引;

  ```
  CREATE TABLE a ( a char(10));
  EXPLAIN SELECT * FROM a WHERE a ="1" – 走索引
  ```

  EXPLAIN SELECT * FROM `a` WHERE `a` =1 – 不走索引

  正则表达式不使用索引,这应该很好理解,所以为什么在SQL中很难看到regexp关键字的原因

有哪些数据结构可以作为索引呢?

可回答：MySQL索引的数据结构

问过的一些公司：美团(2021.08)，友塔游戏(2021.08)，京东(2021.09)

参考答案：

索引的数据结构主要有**B+树**和**哈希表**。

B树与B+树的区别?

问过的一些公司：Shopee(2021.07)，字节(2021.08)，蔚来(2021.09)，美团(2021.09)

参考答案：

B树非叶子结点和叶子结点都存储数据，因此查询数据时，时间复杂度最好为 O(1)，最坏为 O(log n)。而B+树只在叶子结点存储数据，非叶子结点存储关键字，且不同非叶子结点的关键字可能重复，因此查询数据时，时间复杂度固定为O(log n)。

B+树叶子结点之间用链表相互连接，因而只需扫描叶子结点的链表就可以完成一次遍历操作，B树只能通过中序遍历。

为什么使用B+树作为索引结构?

可回答：1）InnoDB为什么使用B+树作为存储引擎；2）用B+树不用B树原因

问过的一些公司：阿里蚂蚁(2021.08)，美团(2021.08)，字节(2021.08)，蔚来(2021.09)，阿里(2021.09)，网易有道(2021.09)

公众号：旧时光大数据

参考答案：

1、B+树能显著减少IO次数，提高效率

B+树的节点只存储索引key值，具体信息的地址存在于叶子节点的地址中。这就使以页为单位的索引中可以存放更多的节点。减少更多的I/O支出。

2、B+树的查询效率更加稳定

B+树的查询效率更加稳定，任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

3、B+树更加适合在区间查询的情况

由于B+树的数据都存储在叶子结点中，叶子结点均为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，而在数据库中基于范围的查询是非常频繁的，所以通常B+树用于数据库索引。

不使用B+树，可以用那个数据类型实现一个索引结构

问过的一些公司：阿里蚂蚁(2021.08)

参考答案：

可使用哈希表，也就是哈希索引。

哈希索引是基于哈希表实现的，对于每一行数据，存储引擎会对索引列进行哈希计算得到哈希码，并且哈希算法要尽量保证不同的列值计算出的哈希码值是不同的，将哈希码的值作为哈希表的key值，将指向数据行的指针作为哈希表的value值。这样查找一个数据的时间复杂度就是O(1)，一般多用于精确查找。

介绍下MySQL的联合索引

问过的一些公司：美团，猿辅导

参考答案：

比较简单的是单列索引（b+tree）。遇到多条件查询时，不可避免会使用到多列索引。联合索引又叫复合索引。

联合索引，相对于一般索引只有一个字段，联合索引可以为多个字段创建一个索引。它的原理也很简单，比如，我们在（a,b,c）字段上创建一个联合索引，则索引记录会首先按照A字段排序，然后再按照B字段排序然后再是C字段，因此，联合索引的特点就是：

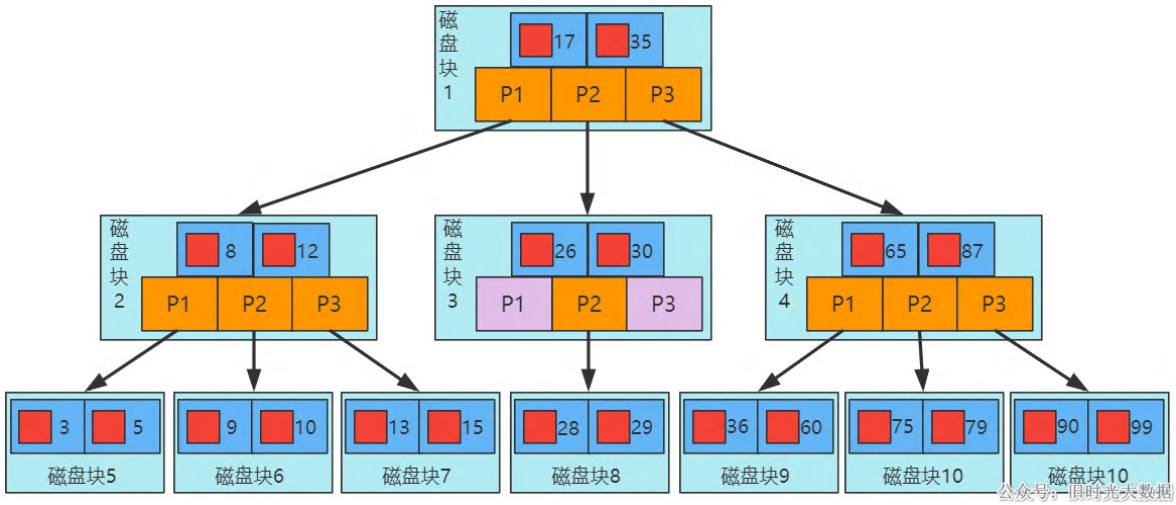
- 第一个字段一定是有序的
- 当第一个字段值相等的时候，第二个字段又是有序的，比如下表中当A=2时所有B的值是有序排列的，依次类推，当同一个B值得所有C字段是有序排列的

```
| A | B | C |
| 1 | 2 | 3 |
| 1 | 4 | 2 |
| 1 | 1 | 4 |
| 2 | 3 | 5 |
| 2 | 4 | 4 |
| 2 | 4 | 6 |
| 2 | 5 | 5 |
```

其实联合索引的查找就跟查字典是一样的，先根据第一个字母查，然后再根据第二个字母查，或者只根据第一个字母查，但是不能跳过第一个字母从第二个字母开始查。这就是所谓的最左前缀原理。

b+tree结构如下：

每一个磁盘块在mysql中是一个页，页大小是固定的，mysql innodb的默认的页大小是16k，每个索引会分配在页上的数量是由字段的大小决定。当字段值的长度越长，每一页上的数量就会越少，因此在一定数据量的情况下，索引的深度会越深，影响索引的查找效率。



对于复合索引（多列b+tree，使用多列值组合而成的b+tree索引）。遵循最左侧原则，从左到右的使用索引中的字段，一个查询可以只使用索引中的一部份，但只能是最左侧部分。例如索引是key index (a,b,c). 可以支持a a,b a,b,c 3种组合进行查找，但不支持 b,c进行查找。当使用最左侧字段时，索引就十分有效。

创建表test如下：

```
create table test(
    a int,
    b int,
    c int,
    KEY a(a,b,c)
);
```

比如(a,b,c)的时候，b+数是按照从左到右的顺序来建立搜索树的，比如当(a=? and b=? and c=?)这样的数据来检索的时候，b+树会优先比较a列来确定下一步的所搜方向，如果a列相同再依次比较b列和c列，最后得到检索的数据；但当(b=? and c=?)这样的没有a列的数据来的时候，b+树就不知道下一步该查哪个节点，因为建立搜索树的时候a列就是第一个比较因子，必须要先根据a列来搜索才能知道下一步去哪里查询。比如当(a=? and c=?)这样的数据来检索时，b+树可以用a列来指定搜索方向，但下一个字段b列的缺失，所以只能把a列的数据找到，然后再匹配c列的数据了， 这个是非常重要的性质，即索引的最左匹配特性。以下通过例子分析索引的使用情况，以便于更好的理解联合索引的查询方式和使用范围。

公众号：旧时光大数据

1、多列索引在and查询中应用

```
select * from test where a=? and b=? and c=?;   /*查询效率最高，索引全覆盖。*/
select * from test where a=? and b=?;   /*索引覆盖a和b。*/
select * from test where b=? and a=?;   /*经过mysql的查询分析器的优化，索引覆盖a和b。*/
select * from test where a=?;   /*索引覆盖a。*/
select * from test where b=? and c=?;   /*没有a列，不走索引，索引失效。*/
select * from test where c=?;   /*没有a列，不走索引，索引失效。*/


```

2、多列索引在范围查询中应用

```
/*索引覆盖a和b，因b列是范围查询，因此c列不能走索引。*/
select * from test where a=? and b between ? and ? and c=?;
/*a列走索引，因a列是范围查询，因此b列是无法使用索引。*/
select * from test where a between ? and ? and b=?;
/*a列走索引，因a列是范围查询，b列是范围查询也不能使用索引。*/
select * from test where a between ? and ? and b between ? and ? and c=?;


```

3、多列索引在排序中应用

```
/*a、b、c三列全覆盖索引，查询效率最高。*/
select * from test where a=? and b=? order by c;
/*a、b列使用索引查找，因b列是范围查询，因此c列不能使用索引，会出现file sort。*/
select * from test where a=? and b between ? and ? order by c;


```

总结：

联合索引的使用在写where条件的顺序无关，mysql查询分析会进行优化而使用索引。但是减轻查询分析器的压力，最好和索引的从左到右的顺序一致。使用等值查询，多列同时查询，索引会一直传递并生效。因此等值查询效率最好。索引查找遵循最左侧原则。但是遇到范围查询列之后的列索引失效。排序也能使用索引，合理使用索引排序，避免出现file sort。

联合索使用原则

问过的一些公司：兴业数金(2021.09)

参考答案：

最左原则

数据库有必要建索引吗?

问过的一些公司：美团点评

参考答案：

为什么要创建索引呢？这是因为，创建索引可以大大提高系统的性能。

- 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。
- 可以大大加快 数据的检索速度，这也是创建索引的最主要的原因。

公众号：旧时光大数据

- 可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。
- 在使用分组和排序 子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。
- 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？这种想法固然有其合理性，然而也有其片面性。虽然，索引有许多优点，但是，为表中的每一个列都增加索引，是非常不明智的。这是因为，**增加索引也有许多不利的一个方面。**

- 创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。
- 索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。
- 当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

索引是建立在数据库表中的某些列的上面。因此，在创建索引的时候，应该仔细考虑在哪些列上可以创建索引，在哪些列上不能创建索引。一般来说，**应该在这些列上创建索引**，例如：

- 在经常需要搜索的列上，可以加快搜索的速度；
- 在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；
- 在经常用在连接的列上，这 些列主要是一些外键，可以加快连接的速度；
- 在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；
- 在经常需要排序的列上创 建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；
- 在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

同样，对于有些列不应该创建索引。一般来说，**不应该创建索引的的这些列具有下列特点：**

- 对于那些在查询中很少使用或者参考的列不应该创建索引。这是因 为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。
- 对于那 些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。
- 对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。
- 当修改性能远远大于检索性能时，不应该创建索引。这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

创建索引的方法和索引的特征

1）创建索引的方法

创建索引有多种方法，这些方法包括直接创建索引的方法和间接创建索引的方法。直接创建索引，例如使用CREATE INDEX语句或者使用创建索引向导，间接创建索引，例如在表中定义主键约束或者唯一性键约束时，同时也创建了索引。虽然，这两种方法都可以创建索引，但是，它们创建索引的具体内容是有区别的。

直接创建索引：

- 使用CREATE INDEX语句或者使用创建索引向导来创建索引，这是最基本的索引创建方式，并且这种方法最具有柔性，可以定制创建出符合自己需要的索引。在使用这种方式 创建索引时，可以使用许多选项，例如指定数据页的充满度、进行排序、整理统计信息等，这样可以优化索引。使用这种方法，可以指定索引的类型、唯一性和复合 性，也就是说，既可以创建聚簇索引，也可以创建非聚簇索引，既可以在一个列上创建索引，也可以在两个或者两个以上的列上创建索引。

间接创建索引：

公众号：旧时光大数据

- 通过定义主键约束或者唯一性键约束，也可以间接创建索引。主键约束是一种保持数据完整性的逻辑，它限制表中的记录有相同的主键记录。在创建主键约束时，系统自动创建了一个唯一性的聚簇索引。虽然，在逻辑上，主键约束是一种重要的结构，但是，在物理结构上，与主键约束相对应的结构是唯一性的聚簇索引。换句话说，在物理实现上，不存在主键约束，而只存在唯一性的聚簇索引。同样，在创建唯一性键约束时，也同时创建了索引，这种索引则是唯一性的非聚簇索引。因此，当使用约束创建索引时，索引的类型和特征基本上都已经确定了，由用户定制的余地比较小。

当在表上定义主键或者唯一性键约束时，如果表 中已经有了使用CREATE INDEX语句创建的标准索引时，那么主键约束或者唯一性键约束创建的索引覆盖以前创建的标准索引。也就是说，**主键约束或者唯一性键约束创建的索引的优先 级高于使用CREATE INDEX语句创建的索引。**

2）索引的特征

索引有两个特征，即唯一性索引和复合索引。

唯一性索引：

- 保证在索引列中的全部数据是唯一的，不会包含冗余数据。如果表中已经有一个主键约束或者唯一性键约束，那么当创建表或者修改表时，SQL Server自动创建一个唯一性索引。然而，如果必须保证唯一性，那么应该创建主键约束或者唯一性键约束，而不是创建一个唯一性索引。当创建唯一性索引 时，应该认真考虑这些规则：当在表中创建主键约束或者唯一性键约束时，SQL Server自动创建一个唯一性索引；如果表中已经包含有数据，那么当创建索引时，SQL Server检查表中已有数据的冗余性；每当使用插入语句插入数据或者使用修改语句修改数据时，SQL Server检查数据的冗余性：如果有冗余值，那么SQL Server取消该语句的执行，并且返回一个错误消息；确保表中的每一行数据都有一个唯一值，这样可以确保每一个实体都可以唯一确认；只能在可以保证实体 完整性的列上创建唯一性索引，例如，不能在人事表中的姓名列上创建唯一性索引，因为人们可以有相同的姓名。

复合索引：

- 就是一个索引创建在两个列或者多个列上。在搜索时，当两个或者多个列作为一个关键值时，最好在这些列上创建复合索引。当创建复合索引时，应该考虑这些规则：最多可以把16个列合并成一个单独的复合索引，构成复合索引的列的总长度不能超过900字节，也就是说复合列的长度不能太长；在复合索引中，所 有的列必须来自同一个表中，不能跨表建立复合列；在复合索引中，列的排列顺序是非常重要的，因此要认真排列列的顺序，原则上，应该首先定义最唯一的列，例 如在（COL1，COL2）上的索引与在（COL2，COL1）上的索引是不相同的，因为两个索引的列的顺序不同；为了使查询优化器使用复合索引，查询语 句中的WHERE子句必须参考复合索引中第一个列；当表中有多个关键列时，复合索引是非常有用的；使用复合索引可以提高查询性能，减少在一个表中所创建的 索引数量。

MySQL缺点?

问过的一些公司：美团点评

参考答案：

不支持热备份；

MySQL最大的缺点是其安全系统，主要是复杂而非标准，另外只有到调用mysqladmin来重读用户权限时才发生改变；

没有一种存储过程(Stored Procedure)语言，这是对习惯于企业级数据库的程序员的最大限制；

MySQL的价格随平台和安装方式变化。Linux的MySQL如果由用户自己或系统管理员而不是第三方安装则是免费的，第三方案则必须付许可费。Unix或linux 自行安装 免费 、Unix或Linux 第三方安装收费；

公众号：旧时光大数据

什么是脏读？怎么解决？

问过的一些公司：头条，唯品会(2021.07)

参考答案：

脏读就是指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。

解决脏读问题：

开启MySQL的READ COMMITTED隔离等级。

为什么要有三大范式，建数据库时一定要遵循吗？

问过的一些公司：唯品会(2021.07)

参考答案：

有范式，自然就有反范式，不满足范式的模型，就是**反范式模型**。

范式的优点：范式避免了数据冗余，减少数据库的空间，减轻维护数据完整性的麻烦。

范式的缺点：范式往往需要进行多表连接，当数据量很大时，效率比较低。

范式与反范式的比较：

1）查询记录时，**范式模式往往要进行多表连接，而反范式只需在同一张表中查询**，当数据量很大的时候，显然反范式的效率会更好。

2）**反范式有很多重复的数据**，会占用更多的内存，查询时可能会较多地使用Group By或Distinct等耗时耗性能的关键字。

3）当要**修改更新数据**时（例如修改Accounting部门的领导为Russell），范式更灵活，而反范式要修改全部的数据，且易出错。

关于范式与反范式的比较还有很多很多，二者各有各的好处，在实际开发中应该根据需要合理地混用不同的模式，最大程度地结合发挥各自的优点。

数据库一般对哪些列建立索引？索引的数据结构？

问过的一些公司：字节(2021.07)

参考答案：

数据库一般会对**外键列、主键列、频繁更新、频繁修改索引的列**建立索引。

索引的数据结构主要有B+树和哈希表，对应的索引分别为B+树索引和哈希索引。InnoDB引擎的索引类型有B+树索引和哈希索引，默认的索引类型为B+树索引。

MySQL中索引的建立需要考虑哪些问题

问过的一些公司：58同城(2021.08)

参考答案：

1）索引不在多，而在于合适

公众号：旧时光大数据

2）在记录较多条的表中，对经常用作where条件的字段设置索引（表的主键，外键会自动创建索引）

3）假如多个条件经常同时出现，可以考虑多个字段建立联合索引，否则，建立多个单独索引

4）频繁进行写入操作的表，要注意不能建立太多索引

5）如果有多个字段，需要做联合索引，在做联合索引的时候，要把识别率最高的字段放在最左边，因为MySQL索引查询会遵循最左前缀匹配的原则，即最左优先，在检索数据时从联合索引的最左边开始匹配

6）索引字段，尽量避免NULL，应该指定列为NOT

使用NULL，除非你就想存储NULL。在MySQL中含有空值的列很难进行查询优化。因为它们使得索引、索引的统计信息以及比较运算更加复杂。可以用0、一个特殊的值或一个空串代替空值，这是设计表的时候要考虑的问题。

关系型数据库与非关系型数据库区别

问过的一些公司：字节(2021.07)

参考答案：

1、关系型数据库

常用的几个：MySQL、Oracle

关系型数据库最典型的**数据结构是表**，由**二维表**及其**之间的联系**所组成的一个数据组织。

优点：

1）易于维护：都是使用表结构，格式一致；

2）使用方便：SQL语言通用，可用于复杂查询；

3）复杂操作：支持SQL，可用于一个表以及多个表之间非常复杂的查询。

缺点：

1）读写性能比较差，尤其是海量数据的高效率读写；

2）固定的表结构，灵活度稍欠；

3）高并发读写需求，传统关系型数据库来说，硬盘I/O是一个很大的瓶颈。

2、非关系型数据库

常用的几个：Redis、HBase、MongoDB

非关系型数据库严格上不是一**加粗样式**种数据库，应该**是一种数据结构化存储方法的集合，可以是文档或者键值对等。**

优点：

1）格式灵活：存储数据的格式可以是key,value形式、文档形式、图片形式等等，文档形式、图片形式等等，使用灵活，应用场景广泛，而关系型数据库则只支持基础类型。

2）速度快：nosql可以使用硬盘或者随机存储器作为载体，而关系型数据库只能使用硬盘；

3）高扩展性；

4）成本低：nosql数据库部署简单，基本都是开源软件。

缺点：

公众号：旧时光大数据

1）不提供sql支持，学习和使用成本较高；

2）无事务处理；

3）数据结构相对复杂，复杂查询方面稍欠。

3、小结

首先一般**非关系型数据库是基于CAP模型**，而传统的**关系型数据库是基于ACID模型**的

1）数据存储结构

首先关系型数据库一般都有固定的表结构，并且需要通过DDL语句来修改表结构，不是很容易进行扩展，而非关系型数据库的存储机制就有很多了，比如基于文档的，K-V键值对的，还有基于图的等，对于数据的格式十分灵活没有固定的表结构，方便扩展，因此如果业务的数据结构并不是固定的或者经常变动比较大的，那么非关系型数据库是个好的选择。

2）可扩展性

传统的关系型数据库给人一种横向扩展难，不好对数据进行分片等，而一些非关系型数据库则原生就支持数据的水平扩展(比如mongodb的sharding机制)，并且这可能也是很多NoSQL的一大卖点，其实象Mysql这种关系型数据库的水平扩展也并不是难，即使NoSQL水平扩展容易但对于向跨分片进行joins这种场景都没有什么太好的解决办法，不管是关系型还是非关系型数据库，解决水平扩展或者跨分片Joins这种场景，在应用层和数据库层中间加一层中间件来做数据处理也许是个好的办法。

3）数据一致性

非关系型数据库一般强调的是数据最终一致性，而不没有像ACID一样强调数据的强一致性，从非关系型数据库中读到的有可能还是处于一个中间态的数据，因此如果你的业务对于数据的一致性要求很高，那么非关系型数据库并不一个很好的选择，非关系型数据库可能更多的偏向于OLAP场景，而关系型数据库更多偏向于OLTP场景。

MySQL与Redis区别

问过的一些公司：字节(2021.07)

回答技巧：一般答第一点就可以了

参考答案：

1、从类型上看

MySQL是关系型数据库，**主要用于存放持久化数据**，将数据存储在硬盘中，**读取速度较慢。**

Redis是NOSQL，即非关系型数据库，也是缓存数据库，即**将数据存储在缓存中，缓存的读取速度快，能够大大的提高运行效率**，但是保存时间有限。

缓存就是数据交换的缓冲区（cache），当浏览器执行请求时，首先会对在缓存中进行查找，如果存在，就获取；否则就访问数据库。缓存的好处就是读取速度快。

2、从运行机制上来看

MySQL作为持久化存储的关系型数据库，相对薄弱的地方在于每次请求访问数据库时，都存在着I/O操作，如果反复频繁的访问数据库。1）会在反复链接数据库上花费大量时间，从而导致运行效率过慢；2）反复的访问数据库也会导致数据库的负载过高，那么此时缓存的概念就衍生了出来。

Redis数据库就是一款缓存数据库，用于存储使用频繁的数据，这样减少访问数据库的次数，提高运行效率。

3、从数据存放位置上看

公众号：旧时光大数据

数据存放位置MySQL：数据放在磁盘

Redis：数据放在内存

列式数据库和行式数据库优劣比对

问过的一些公司：阿里，多益(2021.09)，百度(2021.09)

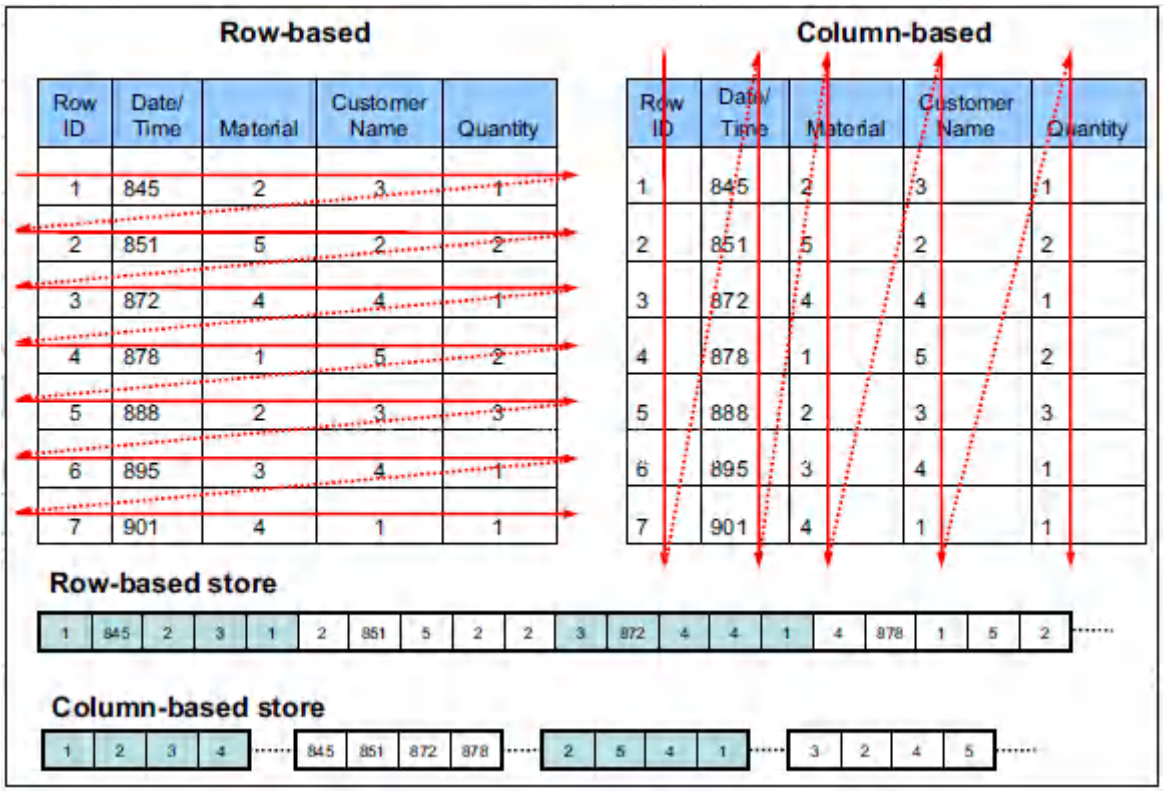
参考答案：

1、行式存储和列式存储

传统的关系型数据库，如 Oracle、DB2、MySQL、SQL SERVER 等采用行式存储法(Row-based)，在基于行式存储的数据库中， 数据是按照行数据为基础逻辑存储单元进行存储的， 一行中的数据在存储介质中以连续存储形式存在。

列式存储(Column-based)是相对于行式存储来说的，新兴的 Hbase、HP Vertica、EMC Greenplum 等分布式数据库均采用列式存储。在基于列式存储的数据库中， 数据是按照列为基础逻辑存储单元进行存储的，一列中的数据在存储介质中以连续存储形式存在。

将表放入存储系统中有两种方法，而我们绝大部分是采用行存储的。行存储法是将各行放入连续的物理位置，这很像传统的记录和文件系统。列存储法是将数据按照列存储到数据库中，与行存储类似，下图是两种存储方法的图形化解释。



应用行式存储的数据库系统称为行式数据库，同理应用列式存储的数据库系统称为列式数据库。随着列式数据库的发展，传统的行式数据库加入了列式存储的支持，形成具有两种存储方式的数据库系统。

2、区别

1）传统行式数据库

- 数据是按行存储的
- 没有索引的查询使用大量I/O
- 建立索引和物化视图需要花费大量时间和资源
- 面对查询的需求，数据库必须被大量膨胀才能满足性能要求

公众号：旧时光大数据

2）列式数据库

- 数据按列存储，每一列单独存放
- 数据即是索引
- 只访问查询设计的列，大量降低系统I/O
- 每一列由一个线索来处理，查询的并发处理
- 数据类型一致，数据特征相似

为啥列存储可以大幅降低系统的I/O呢?

列式存储的主要优点之一就是可以大幅降低系统的I/O，尤其是在海量数据查询时，I/O向来是系统的主要瓶颈之一。通过下面这张图，相信大家能够彻底明白这一点。

3、应用场景

在比较了行式数据库与列式数据库之后，我们更关心的是如何根据业务场景需要选择对应的数据库系统。

行式更适合OLTP，比如传统的基于增删改查操作的应用。列式更适合OLAP，非常适合于在数据仓库领域发挥作用，比如数据分析、海量存储和商业智能；涉及不经常更新的数据。

由于设计上的不同，列式数据库在并行查询处理和压缩上更有优势。而且数据是以列为单元存储，完全不用考虑数据建模或者说建模更简单了。要查询计算哪些列上的数据，直接读取列就行。

最后我们需要务实的指出，没有万能的数据库，列式数据库也并非万能，只不过给DBA提供了更多的选择，DBA需根据自己的应用场景自行选择。

除了UTF-8还有什么编码格式

问过的一些公司：阿里蚂蚁(2021.08)

参考答案：

ASCII，Unicode，GB2312

布隆过滤器的基本原理是什么？局限性是什么？使用什么方法可以增加删除的功能?

问过的一些公司：阿里蚂蚁(2021.08)

参考答案：

公众号：旧时光大数据

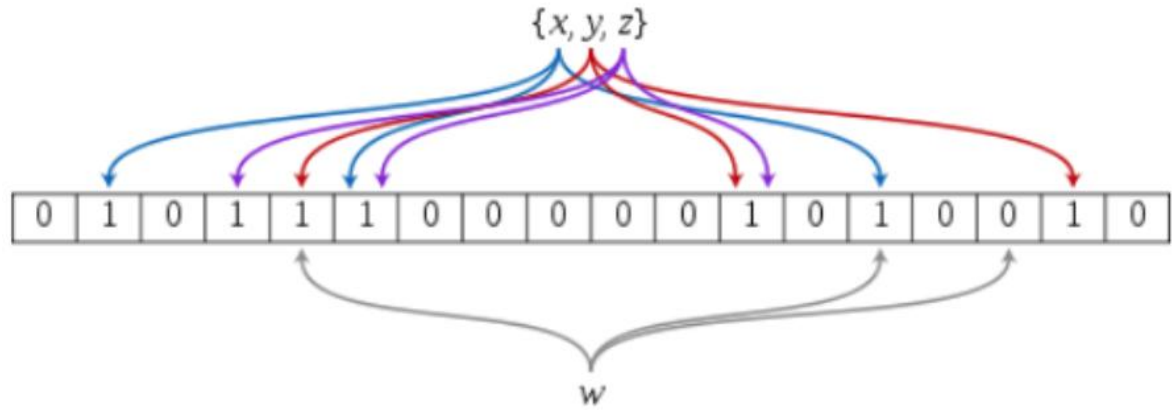
1、布隆过滤器的概念及基本原理

1）概念

布隆过滤器（Bloom Filter)是一种紧凑型的、比较巧妙的**概率型数据结构**，特点是**高效地插入和查询**，可以用来告诉你**某样东西一定不存在或者可能存在**，它是用多个哈希函数，将一个数据映射到位图结构中。**此种方式不仅可以提升查询效率，也可以节省大量的内存空间**，但是布隆过滤器也存在一定的缺陷：**数据只能插入不能删除。**

2）基本原理

当一个元素被加入集合时，通过K个散列函数将这个元素映射成一个位数组中的K个点，把它们置为1。检索时，我们只要看看这些点是不是都是1就（大约）知道集合中有没有它了：如果这些点有任何一个0，则被检元素一定不在；如果都是1，则被检元素很可能在。这就是布隆过滤器的基本思想。



2、局限性

1）有误判率，即存在假阳性（False Position），即不能准确判断元素是否在集合中（补救方法：再建立一个白名单，存储可能会误判的数据）

2）不能获取元素本身

3）一般情况下不能从布隆过滤器中删除元素

4）如果采用计数方式删除，可能会存在计数回绕问题

3、布隆过滤器（Bloom Filter）增加删除的功能

布隆过滤器本身不支持元素删除，因为删除一个元素时可能会影响到其他元素。

案例如下：

“niuke"利用三个不同的哈希函数返回的哈希值为1、4、7，“csdnnn"利用三个不同的哈希函数返回的哈希值为3、4。8，假设删除“niuke"这个元素，则1、4、7对应的下标置0，但是“csdnnn"这个元素与“niuke”对应的下标4重合，这样就会将“csdnnn”这个元素也删除，所以布隆过滤器不支持删除。

一种支持删除的方法：将布隆过滤器中的每个比特位扩展成一个小的计数器，插入元素时给k个计数器(k个哈希函数计算出的哈希地址)加一，删除元素时，给k个计数器减一，通过多占用几倍存储空间的代价来增加删除操作。

但是这中删除方法也存在一定缺陷：

- 无法确认元素是否真正在布隆过滤器中
- 存在计数回绕

你在哪些场景下使用了布隆过滤器?

公众号：旧时光大数据

问过的一些公司：阿里蚂蚁(2021.08)

参考答案：

1）**网页爬虫对URL的去重**，避免爬去相同的URL地址。

2）**垃圾邮件过滤**，从数十亿个垃圾邮件列表中判断某邮箱是否是杀垃圾邮箱。

3）**解决数据库缓存击穿**，黑客攻击服务器时，会构建大量不存在于缓存中的key向服务器发起请求，在数据量足够大的时候，频繁的数据库查询会导致挂机。

4）**秒杀系统**，查看用户是否重复购买。

SQL慢查询的解决方案（优化）？

问过的一些公司：美团(2021.08)

参考答案：

从这几方面考虑：索引+sql语句+数据库结构优化+优化器优化+架构优化。

1、索引

- 尽量覆盖索引，5.6支持索引下推
- 组合索引符合最左匹配原则
- 避免索引失效
- 再写多读少的场景下，可以选择普通索引而不要唯一索引。更新时，普通索引可以使用change buffer进行优化，减少磁盘IO,将更新操作记录到change bufer，等查询来了将数据读到内存再进行修改.
- 索引建立原则（一般建在where和order by，基数要大，区分度要高，不要过度索引，外键建索引）

2、SQL语句

1）分页查询优化

该方案适用于主键自增的表，可以把Limit查询转换成某个位置的查询。

```
1   select * from tb_sku where id>20000 limit 10;
2
```

2）优化insert语句

- 多条插入语句写成一条
- 在事务中插数据
- 数据有序插入（主键索引）

3、数据库结构优化

1）将字段多的表分解成多个表有些字段使用频率高，有些低，数据量大时，会由于使用频率低的存在而变慢，可以考虑分开。

2）对于经常联合查询的表，可以考虑建立中间表。

4、优化器优化

优化器使用MRR

原理：MRR 【Multi-Range Read】将ID或键值读到buffer排序，通过把「随机磁盘读」，转化为「顺序磁盘读」，减少磁盘IO，从而提高了索引查询的性能。

公众号：旧时光大数据

对于Myisam，在去磁盘获取完整数据之前，会先按照rowid排好序，再去顺序的读取磁盘。

对于Innodb，则会按照聚簇索引键值排好序，再顺序的读取聚簇索引。

磁盘预读：请求一页的数据时，可以把后面几页的数据也一起返回，放到数据缓冲池中，这样如果下次刚好需要下一页的数据，就不再需要到磁盘读取（局部性原理）。

索引本身就是为了减少磁盘 IO，加快查询，而 MRR，则是把索引减少磁盘 IO 的作用，进一步放大

5、架构优化

读/写分离（主库写，从库读）

聚簇索引、非聚簇索引说一下

可回答：聚簇索引和非聚簇索引的区别

问过的一些公司：友塔游戏(2021.08)，兴业数金(2021.08)，网易有道(2021.09)，贝壳找房(2021.11)，字节(2021.10)

参考答案：

1、聚簇索引

聚簇索引就是**按照每张表的主键构造一颗B+树，同时叶子节点中存放的就是整张表的行记录数据，也将聚集索引的叶子节点称为数据页。**这个特性决定了索引组织表中数据也是索引的一部分，每张表只能拥有一个聚簇索引。

优点：

- 数据访问更快，因为聚簇索引将索引和数据保存在同一个B+树中，因此从聚簇索引中获取数据比非聚簇索引更快。
- 聚簇索引对于主键的排序查找和范围查找速度非常快。

缺点：

- 插入速度严重依赖于插入顺序，按照主键的**顺序插入**是最快的方式，否则将会出现页分裂，严重影响性能。因此，对于InnoDB表，我们一般都会定义一个**自增的ID列为主键。**
- **更新主键的代价很高**，因为将会导致被更新的行移动。因此，对于InnoDB表，我们一般定义主键为不可更新。
- 二级索引访问需要两次索引查找，第一次找到主键值，第二次根据主键值找到行数据。

2、非聚簇索引（辅助索引）

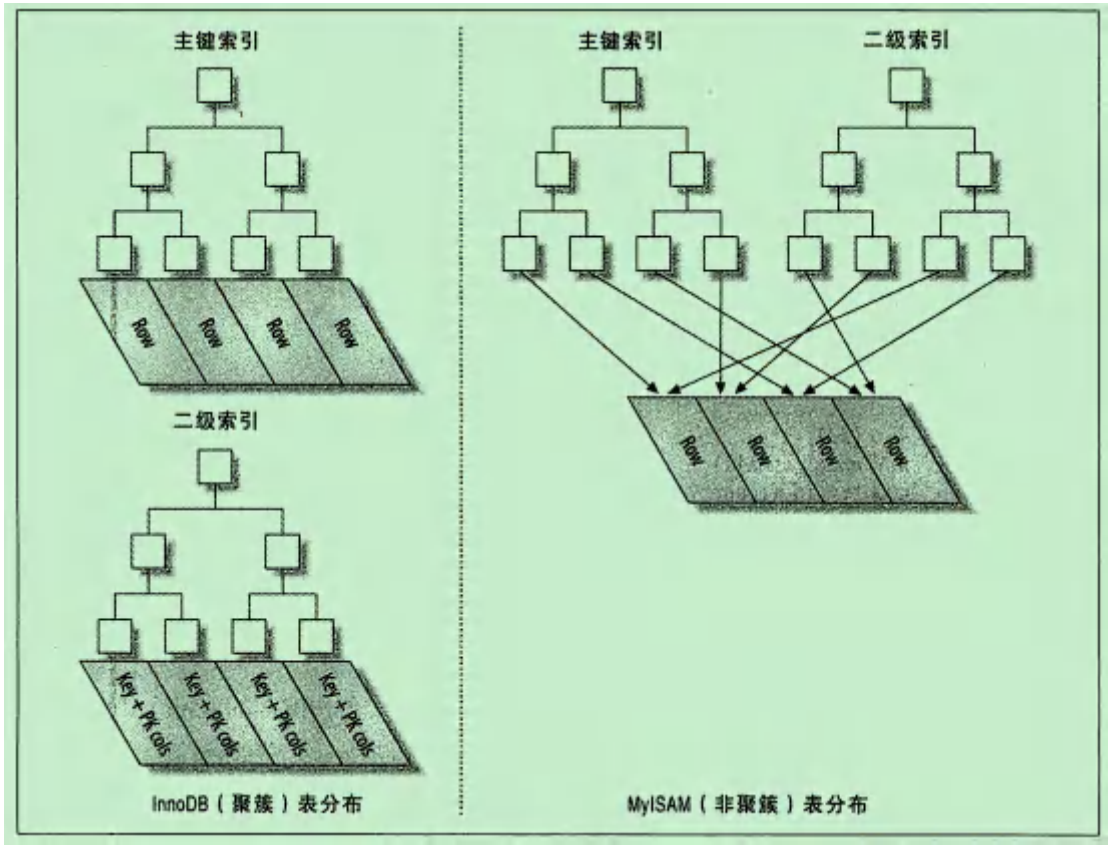
在**聚簇索引之上创建的索引称之为辅助索引，辅助索引访问数据总是需要二次查找。**辅助索引叶子节点存储的不再是行的物理位置，而是主键值。通过辅助索引首先找到的是主键值，再通过主键值找到数据行的数据页，再通过数据页中的Page Directory找到数据行。

3、聚簇索引与非聚簇索引的区别

聚簇索引是将索引和整条记录存放在一起，找到索引就找到了记录。

非聚簇索引只存储索引字段和记录所在的位置，通过索引找到记录所在的位置，然后再根据记录所在位置去获取记录。

一般来讲一堆数据记录最多只能有一个聚簇索引，但可以有很多非聚簇索引。



4、聚簇索引与非聚簇索引的优缺点对比

聚簇索引的查找记录要比非聚簇索引快，因为聚簇索引查找到索引就查找到了数据位置，而非聚簇索引查找到索引之后，根据记录的数据地址，再去查找数据。

一个数据表只能有一个聚簇索引，但可以有多个非聚簇索引。

聚簇索引和非聚簇索引都可以加快查询速度，但同时也都对写入速度会有影响；聚簇索引对写入的速度影响更大一些。

5、聚簇索引与非聚簇索引的使用场景

InnoDB的主键使用的都是聚簇索引，而MyASM**无论是主键索引还是二级索引，使用的都是非聚簇索引。**

哈希索引和B+相比的优势和劣势?

问过的一些公司：友塔游戏(2021.08)

参考答案：

优势：

等值查询，哈希索引明显有绝对优势

因为只需要经过一次算法即可找到相应的键值；当然了，这个前提是，键值都是唯一的。如果键值不是唯一的，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应的数据。

劣势：

1）哈希索引对于范围查询显得无能为力，B+对于范围查找不需要做全表扫描

公众号：旧时光大数据

如果是范围查询检索，这时候哈希索引就毫无用武之地了，因为原先是有序的键值，经过哈希算法后，有可能变成不连续的了，就没办法再利用索引完成范围查询检索。B+ Tree索引底层是多路查询平衡树，节点是天然有序的（左节点小于服节点，右节点大于父节点），所以对于范围查找的时候不需要做全表扫描。

2） 哈希索引无法进行排序，而B+ Tree索引底层是多路查询平衡树，节点是天然有序的

3） 哈希索引遇到大量哈希值相等的情况后性能并不一定就会比B+ Tree索引高。

对于选择性比较低的索引键，如果创建哈希索引，那么将会存在大量记录指针信息存于同一个 哈希值相关联。这样要定位某一条记录时就会非常麻烦，会浪费多次表数据的访问，而造成整体性能低下。

MVCC知道吗?

问过的一些公司：陌陌(2021.10)，蔚来(2021.09)

参考答案：

MVCC(`Multiversion concurrency control`) 就是**同一份数据保留多版本的一种方式，进而实现并发控制**。在查询的时候，通过 `read view` 和版本链找到对应版本的数据。

作用：提升并发性能。对于高并发场景，MVCC比行级锁开销更小。

公众号：旧时光大数据