Flink 面试八股文

1.1 版

本文档来自公众号：**五分钟学大数据**

微信扫码关注

目录

1. 简单介绍一下 Flink

Flink 是一个面向流处理和批处理的分布式数据计算引擎，能够基于同一个 Flink 运行，可以提供流处理和批处理两种类型的功能。 在 Flink 的世界观中，一切都是由流组成的，离线数据是有界的流；实时数据是一个没有界限的流：这就是所谓的有界流和无界流。

2. Flink 的运行必须依赖 Hadoop 组件吗

Flink 可以完全独立于 Hadoop，在不依赖 Hadoop 组件下运行。但是做为大数据的基础设施，Hadoop 体系是任何大数据框架都绕不过去的。Flink 可以集成众多 Hadooop 组件，例如 Yarn、Hbase、HDFS 等等。例如，Flink 可以和 Yarn 集成做资源调度，也可以读写 HDFS，或者利用 HDFS 做检查点。

3. Flink 集群运行时角色

Flink 运行时由两种类型的进程组成：一个 JobManager 和一个或者多个 TaskManager。

Client 不是运行时和程序执行的一部分，而是用于准备数据流并将其发送给 JobManager。之后，客户端可以断开连接（分离模式），或保持连接来接收进程报告（附加模式）。客户端可以作为触发执行 Java/Scala 程序的一部分运行，也可以在命令行进程 `./bin/flink run ...` 中运行。

可以通过多种方式启动 JobManager 和 TaskManager：直接在机器上作为 standalone 集群启动、在容器中启动、或者通过 YARN 等资源框架管理并启动。TaskManager 连接到 JobManagers，宣布自己可用，并被分配工作。

JobManager：

JobManager 具有许多与协调 Flink 应用程序的分布式执行有关的职责：它决定何时调度下一个 task（或一组 task）、对完成的 task 或执行失败做出反应、协调 checkpoint、并且协调从失败中恢复等等。这个进程由三个不同的组件组成：

- ResourceManager

ResourceManager 负责 Flink 集群中的资源提供、回收、分配，管理 task slots。

- Dispatcher

Dispatcher 提供了一个 REST 接口，用来提交 Flink 应用程序执行，并为每个提交的作业启动一个新的 JobMaster。它还运行 Flink WebUI 用来提供作业执行信息。

- JobMaster

JobMaster 负责管理单个 JobGraph 的执行。Flink 集群中可以同时运行多个作业，每个作业都有自己的 JobMaster。

TaskManagers：

TaskManager（也称为 worker）执行作业流的 task，并且缓存和交换数据流。必须始终至少有一个 TaskManager。在 TaskManager 中资源调度的最小单位是 task slot。TaskManager 中 task slot 的数量表示并发处理 task 的数量。请注意一个 task slot 中可以执行多个算子。

4. Flink 相比 Spark Streaming 有什么区别

1. 架构模型

Spark Streaming 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。

2. 任务调度

Spark Streaming 连续不断的生成微小的数据批次，构建有向无环图 DAG，Spark Streaming 会依次创建 DStreamGraph、JobGenerator、JobScheduler。

Flink 根据用户提交的代码生成 StreamGraph，经过优化生成 JobGraph，然后提交给 JobManager 进行处理，JobManager 会根据 JobGraph 生成 ExecutionGraph，ExecutionGraph 是 Flink 调度最核心的数据结构，JobManager 根据 ExecutionGraph 对 Job 进行调度。

3. 时间机制

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。

4. 容错机制

对于 Spark Streaming 任务，我们可以设置 checkpoint，然后假如发生故障并重启，我们可以从上次 checkpoint 之处恢复，但是这个行为只能使得数据不丢失，可能会重复处理，不能做到恰一次处理语义。

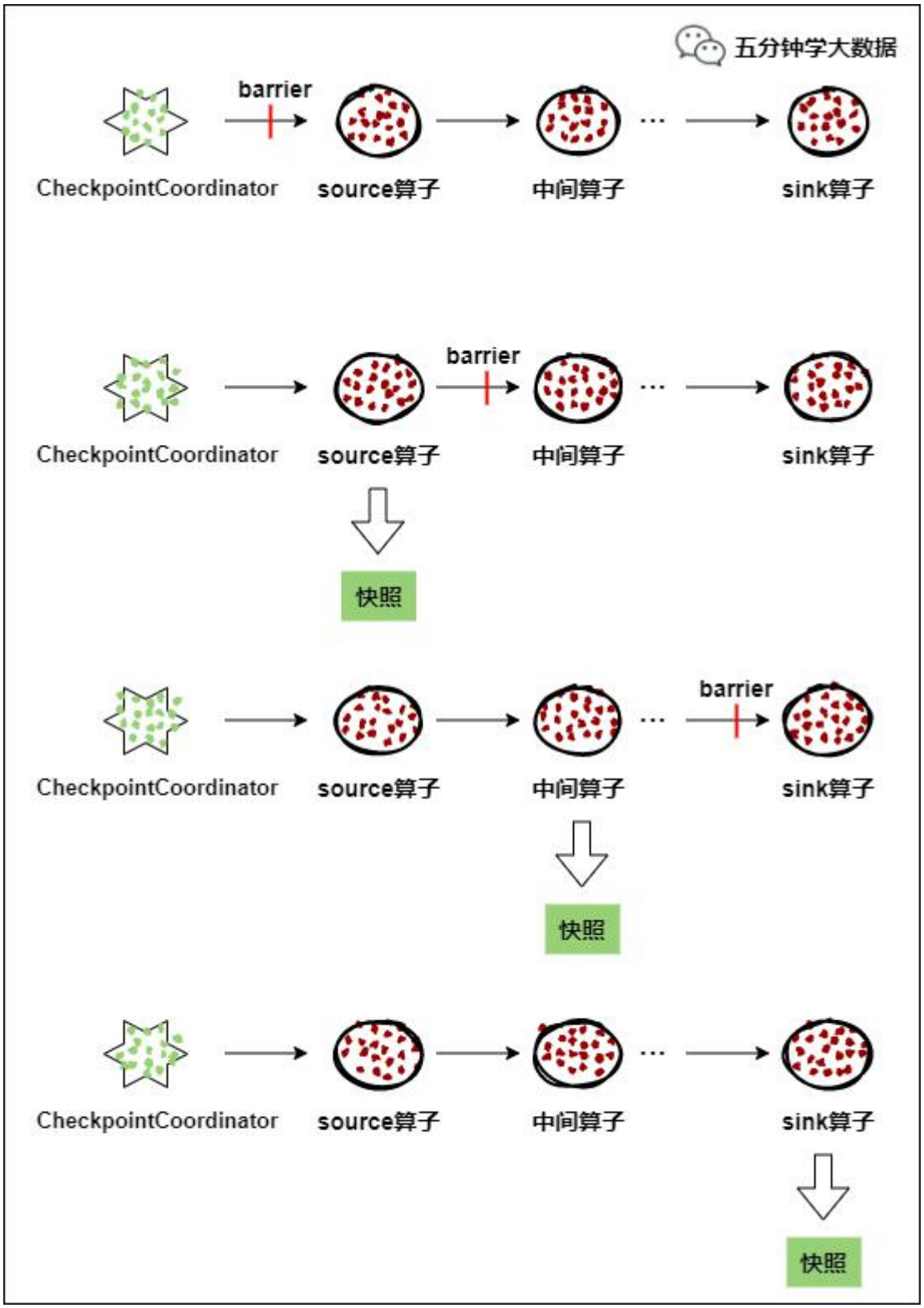
Flink 则使用两阶段提交协议来解决这个问题。

5. 介绍下 Flink 的容错机制（checkpoint）

Checkpoint 机制是 Flink 可靠性的基石，可以保证 Flink 集群在某个算子因为某些原因(如 异常退出)出现故障时，能够将整个应用流图的状态恢复到故障之前的某一状态，保证应用流图状态的一致性。Flink 的 Checkpoint 机制原理来自“Chandy-Lamport algorithm”算法。

每个需要 Checkpoint 的应用在启动时，Flink 的 JobManager 为其创建一个 CheckpointCoordinator(检查点协调器)，CheckpointCoordinator 全权负责本应用的快照制作。

CheckpointCoordinator(检查点协调器)，CheckpointCoordinator 全权负责本应用的快照制作。



1. CheckpointCoordinator(检查点协调器) 周期性的向该流应用的所有 source 算子发送 barrier(屏障)。

2. 当某个 source 算子收到一个 barrier 时，便暂停数据处理过程，然后将自己的当前状态制作成快照，并保存到指定的持久化存储中，最后向 CheckpointCoordinator 报告自己快照制作情况，同时向自身所有下游算子广播该 barrier，恢复数据处理
3. 下游算子收到 barrier 之后，会暂停自己的数据处理过程，然后将自身的相关状态制作成快照，并保存到指定的持久化存储中，最后向 CheckpointCoordinator 报告自身快照情况，同时向自身所有下游算子广播该 barrier，恢复数据处理。
4. 每个算子按照步骤 3 不断制作快照并向下游广播，直到最后 barrier 传递到 sink 算子，快照制作完成。
5. 当 CheckpointCoordinator 收到所有算子的报告之后，认为该周期的快照制作成功；否则，如果在规定的时间内没有收到所有算子的报告，则认为本周期快照制作失败。

文章推荐：

Flink 可靠性的基石-checkpoint 机制详细解析

6. Flink checkpoint 与 Spark Streaming 的有什么区别或优势吗

spark streaming 的 checkpoint 仅仅是针对 driver 的故障恢复做了数据和元数据的 checkpoint。而 flink 的 checkpoint 机制 要复杂了很多，它采用的是轻量级的分布式快照，实现了每个算子的快照，及流动中的数据的快照。

7. Flink 是如何保证 Exactly-once 语义的

Flink 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：

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

预提交（preCommit）将内存中缓存的数据写入文件并关闭

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

丢弃（abort）丢弃临时文件

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。

两阶段提交协议详解：八张图搞懂 Flink 的 Exactly-once

8. 如果下级存储不支持事务，Flink 怎么保证 exactly-once

端到端的 exactly-once 对 sink 要求比较高，具体实现主要有幂等写入和事务性写入两种方式。

幂等写入的场景依赖于业务逻辑，更常见的是用事务性写入。而事务性写入又有预写日志（WAL）和两阶段提交（2PC）两种方式。

如果外部系统不支持事务，那么可以用预写日志的方式，把结果数据先当成状态保存，然后在收到 checkpoint 完成的通知时，一次性写入 sink 系统。

9. Flink 常用的算子有哪些

分两部分：

1. 数据读取，这是 Flink 流计算应用的起点，常用算子有：

- 从内存读：fromElements
- 从文件读：readTextFile
- Socket 接入 ：socketTextStream
- 自定义读取：createInput

2. 处理数据的算子，常用的算子包括：Map（单输入单输出）、FlatMap（单输入、多输出）、Filter（过滤）、KeyBy（分组）、Reduce（聚合）、Window（窗口）、Connect（连接）、Split（分割）等。

推荐阅读：一文学完 Flink 流计算常用算子（Flink 算子大全）

10. Flink 任务延时高，如何入手

在 Flink 的后台任务管理中，我们可以看到 Flink 的哪个算子和 task 出现了反压。最主要的手段是资源调优和算子调优。资源调优即是对作业中的 Operator

的并发数（parallelism）、CPU（core）、堆内存（heap_memory）等参数进行调优。作业参数调优包括：并行度的设置，State 的设置，checkpoint 的设置。

11. Flink 是如何处理反压的

Flink 内部是基于 producer-consumer 模型来进行消息传递的，Flink 的反压设计也是基于这个模型。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。下游消费者消费变慢，上游就会受到阻塞。

12. 如何排查生产环境中的反压问题

1. 反压出现的场景

反压经常出现在促销、热门活动等场景。短时间内流量陡增造成数据的堆积或者消费速度变慢。

它们有一个共同的特点：数据的消费速度小于数据的生产速度。

2. 反压监控方法

通过 Flink Web UI 发现反压问题。

Flink 的 TaskManager 会每隔 50 ms 触发一次反压状态监测，共监测 100 次，并将计算结果反馈给 JobManager，最后由 JobManager 进行计算反压的比例，然后进行展示。

这个比例展示逻辑如下：

OK: 0 <= Ratio <= 0.10，表示状态良好正；

LOW: 0.10 < Ratio <= 0.5，表示有待观察；

HIGH: 0.5 < Ratio <= 1，表示要处理了（增加并行度/subTask/检查是否有数据倾斜/增加内存）。

0.01，代表 100 次中有一次阻塞在内部调用。

3. flink 反压的实现方式

Flink 任务的组成由基本的“流”和“算子”构成，“流”中的数据在“算子”间进行计算和转换时，会被放入分布式的阻塞队列中。当消费者的阻塞队列满时，则会降低生产者的数据生产速度

4. 反压问题定位和处理

Flink 会因为数据堆积和处理速度变慢导致 checkpoint 超时，而 checkpoint 是 Flink 保证数据一致性的关键所在，最终会导致数据的不一致发生。
数据倾斜：可以在 Flink 的后台管理页面看到每个 Task 处理数据的大小。当数据倾斜出现时，通常是简单地使用类似 KeyBy 等分组聚合函数导致的，需要用户将热点 Key 进行预处理，降低或者消除热点 Key 的影。
GC：不合理的设置 TaskManager 的垃圾回收参数会导致严重的 GC 问题，我们可以通过 `-XX:+PrintGCDetails` 参数查看 GC 的日志。
代码本身：开发者错误地使用 Flink 算子，没有深入了解算子的实现机制导致性能问题。我们可以通过查看运行机器节点的 CPU 和内存情况定位问题。

13. Flink 中的状态存储

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。选择的状态存储策略不同，会影响状态持久化如何和 checkpoint 交互。Flink 提供了三种状态存储方式：MemoryStateBackend、FsStateBackend、RocksDBStateBackend。

14. Operator Chains（算子链）这个概念你了解吗

为了更高效地分布式执行，Flink 会尽可能地将 operator 的 subtask 链接（chain）在一起形成 task。每个 task 在一个线程中执行。将 operators 链接成 task 是非常有效的优化：它能减少线程之间的切换，减少消息的序列化/反序列化，减少数据在缓冲区的交换，减少了延迟的同时提高整体的吞吐量。这就是我们所说的算子链。

15. Flink 的内存管理是如何做的

Flink 并不是将大量对象存在堆上，而是将对象都序列化到一个预分配的内存块上。此外，Flink 大量的使用了堆外内存。如果需要处理的数据超出了内存限制，则会将部分数据存储到硬盘上。Flink 为了直接操作二进制数据实现了自己的序列化框架。

16. 如何处理生产环境中的数据倾斜问题

1. flink 数据倾斜的表现：

任务节点频繁出现反压，增加并行度也不能解决问题；

部分节点出现 OOM 异常，是因为大量的数据集中在某个节点上，导致该节点内存被爆，任务失败重启。

2. 数据倾斜产生的原因：

业务上有严重的数据热点，比如滴滴打车的订单数据中北京、上海等几个城市的订单量远远超过其他地区；

技术上大量使用了 KeyBy、GroupBy 等操作，错误的使用了分组 Key，人为产生数据热点。

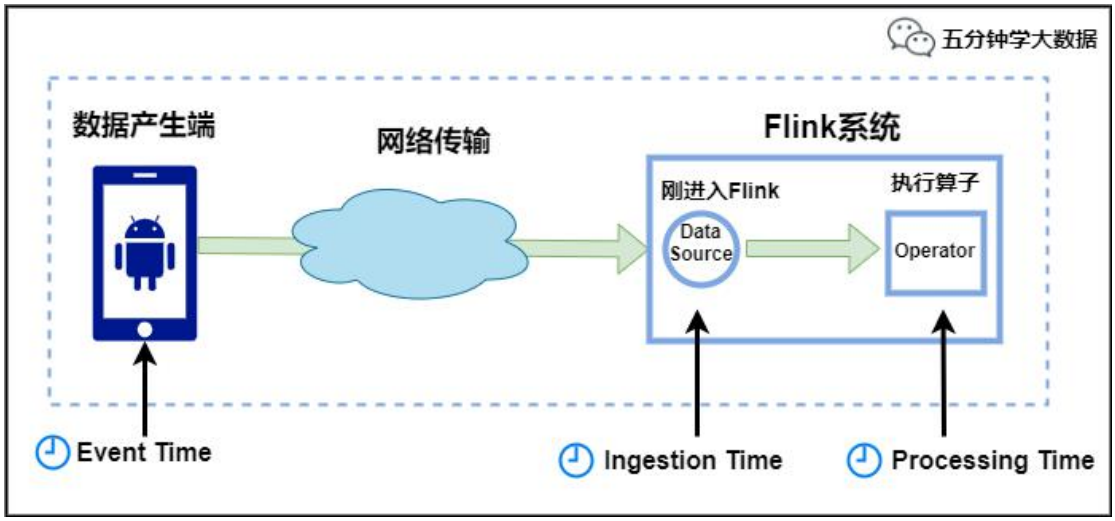
3. 解决问题的思路：

业务上要尽量避免热点 key 的设计，例如我们可以把北京、上海等热点城市分成不同的区域，并进行单独处理；

技术上出现热点时，要调整方案打散原来的 key，避免直接聚合；此外 Flink 还提供了大量的功能可以避免数据倾斜。

17. Flink 中的 Time 有哪几种

Flink 中的时间有三种类型，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。

例如，一条日志进入 Flink 的时间为 2021-01-22 10:00:00.123，到达 Window 的系统时间为 2021-01-22 10:00:01.234，日志的内容如下：

```
2021-01-06 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。

18. Flink 对于迟到数据是怎么处理的

Flink 中 WaterMark 和 Window 机制解决了流式数据的乱序问题，对于因为延迟而顺序有误的数据，可以根据 eventTime 进行业务处理，对于延迟的数据 Flink 也有自己的解决办法，主要的办法是给定一个允许延迟的时间，在该时间范围内仍可以接受处理延迟数据

设置允许延迟的时间是通过 allowedLateness(lateness: Time)设置

保存延迟数据则是通过 sideOutputLateData(outputTag: OutputTag[T])保存

获取延迟数据是通过 DataStream.getSideOutput(tag: OutputTag[X])获取

文章推荐：

Flink 中极其重要的 Time 与 Window 详细解析

19. Flink 中 window 出现数据倾斜怎么解决

window 产生数据倾斜指的是数据在不同的窗口内堆积的数据量相差过多。本质上产生这种情况的原因是数据源头发送的数据量速度不同导致的。出现这种情况一般通过两种方式来解决：

- 在数据进入窗口前做预聚合
- 重新设计窗口聚合的 key

20. Flink CEP 编程中当状态没有到达的时候会将数据保存在哪里

在流式处理中，CEP 当然是要支持 EventTime 的，那么相对应的也要支持数据的迟到现象，也就是 watermark 的处理逻辑。CEP 对未匹配成功的事件序列的处理，和迟到数据是类似的。在 Flink CEP 的处理逻辑中，状态没有满足的和迟到的数据，都会存储在一个 Map 数据结构中，也就是说，如果我们限定判断事件序列的时长为 5 分钟，那么内存中就会存储 5 分钟的数据，这在我看来，也是对内存的极大损伤之一。
推荐阅读：一文学会 Flink CEP

21. Flink 设置并行度的方式

们在实际生产环境中可以从四个不同层面设置并行度：

1. 操作算子层面(Operator Level)

```
.map(new RollingAdditionMapper()).setParallelism(10) //将操作算子设置并行度
```

2. 执行环境层面(Execution Environment Level)

```
$FLINK_HOME/bin/flink 的-p 参数修改并行度
```

3. 客户端层面(Client Level)

```
env.setParallelism(10)
```

4. 系统层面(System Level)

全局配置在 flink-conf.yaml 文件中，parallelism.default，默认是 1：可以设置默认值大一点
需要注意的优先级：**算子层面>环境层面>客户端层面>系统层面**。

22. Flink 中 Task 如何做到数据交换

在一个 Flink Job 中，数据需要在不同的 task 中进行交换，整个数据交换是有 TaskManager 负责的，TaskManager 的网络组件首先从缓冲 buffer 中收集 records，然后再发送。Records 并不是一个一个被发送的，是积累一个批次再发送，batch 技术可以更加高效的利用网络资源。

23. Flink 的内存管理是如何做的

Flink 并不是将大量对象存在堆上，而是将对象都序列化到一个预分配的内存块上。此外，Flink 大量的使用了堆外内存。如果需要处理的数据超出了内存限制，则会将部分数据存储到硬盘上。Flink 为了直接操作二进制数据实现了自己的序列化框架。

24. 介绍下 Flink 的序列化

Flink 摒弃了 Java 原生的序列化方法，以独特的方式处理数据类型和序列化，包含自己的类型描述符，泛型类型提取和类型序列化框架。
TypeInformation 是所有类型描述符的基类。它揭示了该类型的一些基本属性，并且可以生成序列化器。
TypeInformation 支持以下几种类型：

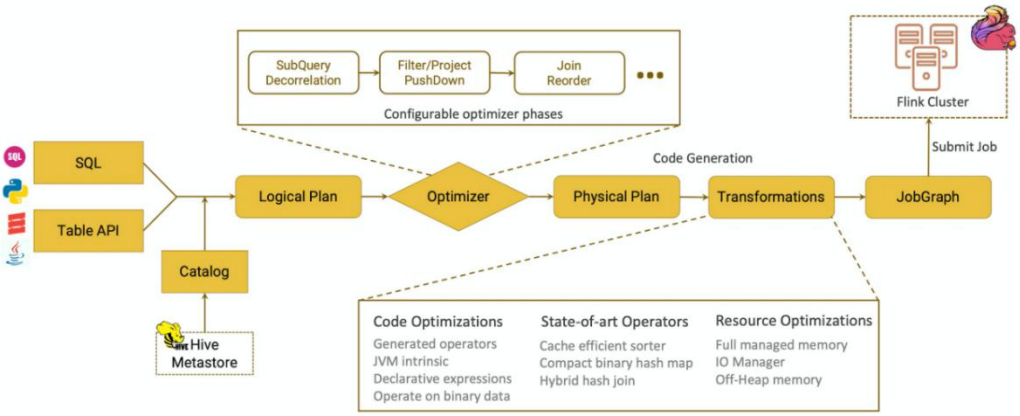
- BasicTypeInfo: 任意 Java 基本类型或 String 类型
- BasicArrayTypeInfo: 任意 Java 基本类型数组或 String 数组
- WritableTypeInfo: 任意 Hadoop Writable 接口的实现类
- TupleTypeInfo: 任意的 Flink Tuple 类型(支持 Tuple1 to Tuple25)。Flink tuples 是固定长度固定类型的 Java Tuple 实现
- CaseClassTypeInfo: 任意的 Scala CaseClass(包括 Scala tuples)
- PojoTypeInfo: 任意的 POJO (Java or Scala)，例如，Java 对象的所有成员变量，要么是 public 修饰符定义，要么有 getter/setter 方法
- GenericTypeInfo: 任意无法匹配之前几种类型的类

25. Flink 海量数据高效去重

1. 基于状态后端。
2. 基于 HyperLogLog：不是精准的去重。
3. 基于布隆过滤器（BloomFilter）；快速判断一个 key 是否存在于某容器，不存在就直接返回。
4. 基于 BitMap；用一个 bit 位来标记某个元素对应的 Value，而 Key 即是该元素。由于采用了 Bit 为单位来存储数据，因此可以大大节省存储空间。

5. 基于外部数据库；选择使用 Redis 或者 HBase 存储数据，我们只需要设计好存储的 Key 即可，不需要关心 Flink 任务重启造成的状态丢失问题。

26. Flink SQL 的是如何实现的



构建抽象语法树的事情交给了 Calcite 去做。SQL query 会经过 Calcite 解析器转变成 SQL 节点树，通过验证后构建成 Calcite 的抽象语法树（也就是图中的 Logical Plan）。另一边，Table API 上的调用会构建成 Table API 的抽象语法树，并通过 Calcite 提供的 RelBuilder 转变成 Calcite 的抽象语法树。然后依次被转换成逻辑执行计划和物理执行计划。

在提交任务后会分发到各个 TaskManager 中运行，在运行时会使用 Janino 编译器编译代码后运行。