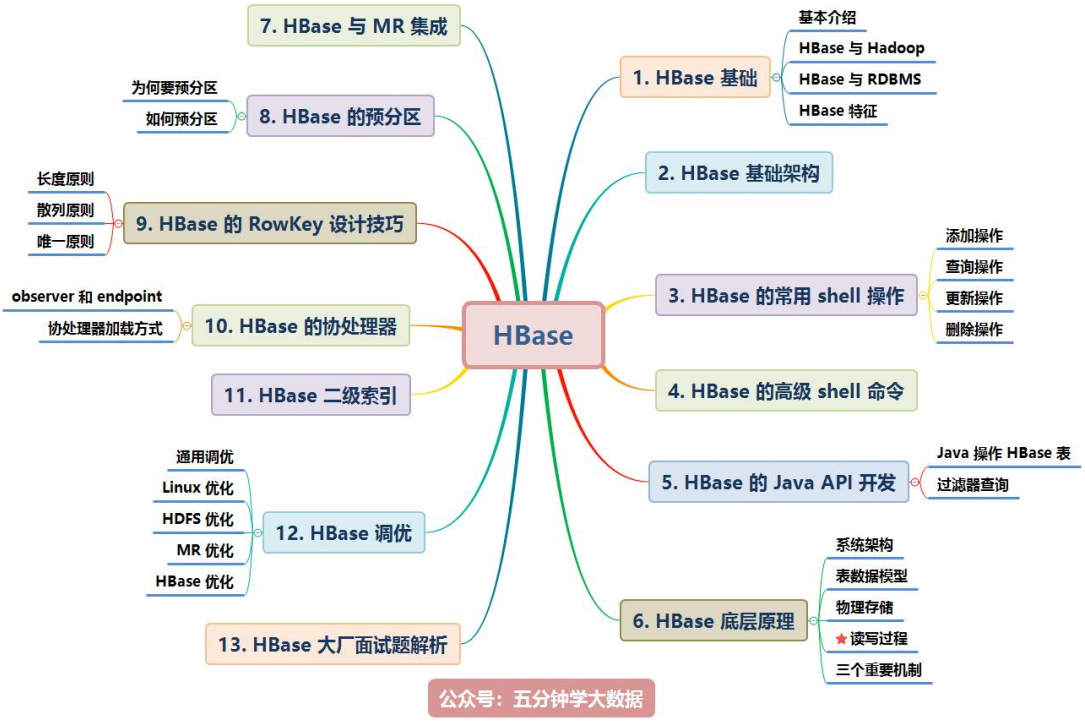
HBase 知识体系吐血总结

本文档来自公众号：**五分钟学大数据**

微信扫码关注

目录

HBase 涉及的知识点如下图所示，本文将逐一讲解：



本文档参考了关于 HBase 的官网及其他众多资料整理而成，为了整洁的排版及舒适的阅读，对于模糊不清晰的图片及黑白图片进行重新绘制成了高清彩图。

一、HBase 基础

1. HBase 基本介绍

简介

HBase 是 BigTable 的开源 Java 版本。是建立在 HDFS 之上，提供高可靠性、高性能、列存储、可伸缩、实时读写 NoSql 的数据库系统。

它介于 NoSql 和 RDBMS 之间，仅能通过主键(row key)和主键的 range 来检索数据，仅支持单行事务(可通过 hive 支持来实现多表 join 等复杂操作)。

主要用来存储结构化和半结构化的松散数据。

Hbase 查询数据功能很简单，不支持 join 等复杂操作，不支持复杂的事务（行级的事务） Hbase 中支持的数据类型：byte[] 与 hadoop 一样，Hbase 目标主要依靠横向扩展，通过不断增加廉价的商用服务器，来增加计算和存储能力。

HBase 中的表一般有这样的特点：

- 大：一个表可以有上十亿行，上百万列
- 面向列:面向列(族)的存储和权限控制，列(族)独立检索。
- 稀疏:对于为空(null)的列，并不占用存储空间，因此，表可以设计的非常稀疏。

HBase 的发展历程

HBase 的原型是 Google 的 BigTable 论文，受到了该论文思想的启发，目前作为 Hadoop 的子项目来开发维护，用于支持结构化的数据存储。

官方网站：http://hbase.apache.org

- 2006 年 Google 发表 BigTable 白皮书
- 2006 年开始开发 HBase
- 2008 HBase 成为了 Hadoop 的子项目
- 2010 年 HBase 成为 Apache 顶级项目

2. HBase 与 Hadoop 的关系

HDFS

- 为分布式存储提供文件系统
- 针对存储大尺寸的文件进行优化，不需要对 HDFS 上的文件进行随机读写
- 直接使用文件

- 数据模型不灵活
- 使用文件系统和处理框架
- 优化一次写入，多次读取的方式

HBase

- 提供表状的面向列的数据存储
- 针对表状数据的随机读写进行优化
- 使用 key-value 操作数据
- 提供灵活的数据模型
- 使用表状存储，支持 MapReduce，依赖 HDFS
- 优化了多次读，以及多次写

3. RDBMS 与 HBase 的对比

关系型数据库

结构：

- 数据库以表的形式存在
- 支持 FAT、NTFS、EXT、文件系统
- 使用 Commit log 存储日志
- 参考系统是坐标系统
- 使用主键（PK）
- 支持分区
- 使用行、列、单元格

功能：

- 支持向上扩展
- 使用 SQL 查询
- 面向行，即每一行都是一个连续单元
- 数据总量依赖于服务器配置
- 具有 ACID 支持
- 适合结构化数据
- 传统关系型数据库一般都是中心化的
- 支持事务
- 支持 Join

HBase

结构：

- 数据库以 region 的形式存在
- 支持 HDFS 文件系统

- 使用 WAL（Write-Ahead Logs）存储日志
- 参考系统是 Zookeeper
- 使用行键（row key）
- 支持分片
- 使用行、列、列族和单元格

功能：

- 支持向外扩展
- 使用 API 和 MapReduce 来访问 HBase 表数据
- 面向列，即每一列都是一个连续的单元
- 数据总量不依赖具体某台机器，而取决于机器数量
- HBase 不支持 ACID（Atomicity、Consistency、Isolation、Durability）
- 适合结构化数据和非结构化数据
- 一般都是分布式的
- HBase 不支持事务
- 不支持 Join

4. HBase 特征简要

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。

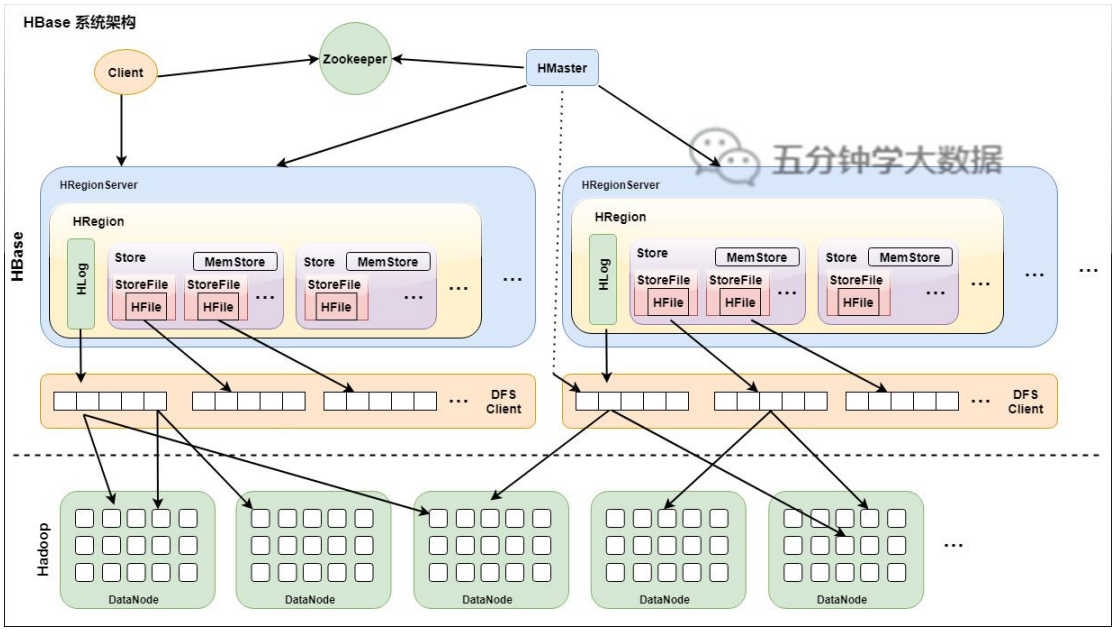
4. 高并发

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。

二、HBase 基础架构



- HMaster

功能：

1. 监控 RegionServer
2. 处理 RegionServer 故障转移
3. 处理元数据的变更
4. 处理 region 的分配或移除
5. 在空闲时间进行数据的负载均衡
6. 通过 Zookeeper 发布自己的位置给客户端

- RegionServer

功能：

1. 负责存储 HBase 的实际数据
2. 处理分配给它的 Region
3. 刷新缓存到 HDFS
4. 维护 HLog
5. 执行压缩
6. 负责处理 Region 分片

组件：

1. Write-Ahead logs

HBase 的修改记录，当对 HBase 读写数据的时候，数据不是直接写进磁盘，它会在内存中保留一段时间（时间以及数据量阈值可以设定）。但把数据保存在内存中可能有更高的概率引起数据丢失，为了解决这个问题，数据会先写在一个叫做 Write-Ahead logfile 的文件中，然后再写入内存中。所以在系统出现故障的时候，数据可以通过这个日志文件重建。

2. HFile

这是在磁盘上保存原始数据的实际的物理文件，是实际的存储文件。

3. Store

HFile 存储在 Store 中，一个 Store 对应 HBase 表中的一个列族。

4. MemStore

顾名思义，就是内存存储，位于内存中，用来保存当前的数据操作，所以当数据保存在 WAL 中之后，RegsionServer 会在内存中存储键值对。

5. Region

Hbase 表的分片，HBase 表会根据 RowKey 值被切分成不同的 region 存储在 RegionServer 中，在一个 RegionServer 中可以有多个不同的 region。

三、HBase 常用 shell 操作

1）添加操作

1. 进入 HBase 客户端命令操作界面

```
$ bin/hbase shell
```

2. 查看帮助命令

```
hbase(main):001:0> help
```

3. 查看当前数据库中有哪些表

```
hbase(main):002:0> list
```

4. 创建一张表

创建 user 表，包含 info、data 两个列族

```
hbase(main):010:0> create 'user', 'info', 'data'
```

或者

```
hbase(main):010:0> create 'user', {NAME => 'info', VERSIONS => '3'},{NAME => 'data'}
```

5. 添加数据操作

向 user 表中插入信息，row key 为 rk0001，列族 info 中添加 name 列标示符，值为 zhangsan

```
hbase(main):011:0> put 'user', 'rk0001', 'info:name', 'zhangsan'
```

向 user 表中插入信息，row key 为 rk0001，列族 info 中添加 gender 列标示符，值为 female

```
hbase(main):012:0> put 'user', 'rk0001', 'info:gender', 'female'
```

向 user 表中插入信息，row key 为 rk0001，列族 info 中添加 age 列标示符，值为 20

```
hbase(main):013:0> put 'user', 'rk0001', 'info:age', 20
```

向 user 表中插入信息，row key 为 rk0001，列族 data 中添加 pic 列标示符，值为 picture

```
hbase(main):014:0> put 'user', 'rk0001', 'data:pic', 'picture'
```

2）查询操作

1. 通过 rowkey 进行查询

获取 user 表中 row key 为 rk0001 的所有信息

```
hbase(main):015:0> get 'user', 'rk0001'
```

2. 查看 rowkey 下面的某个列族的信息

获取 user 表中 row key 为 rk0001，info 列族的所有信息

```
hbase(main):016:0> get 'user', 'rk0001', 'info'
```

3. 查看 rowkey 指定列族指定字段的值

获取 user 表中 row key 为 rk0001，info 列族的 name、age 列标示符的信息

```
hbase(main):017:0> get 'user', 'rk0001', 'info:name', 'info:age'
```

4. 查看 rowkey 指定多个列族的信息

获取 user 表中 row key 为 rk0001，info、data 列族的信息

```
hbase(main):018:0> get 'user', 'rk0001', 'info', 'data'
```

或者这样写

```
hbase(main):019:0> get 'user', 'rk0001', {COLUMN => ['info', 'data']}
```

或者这样写

```
hbase(main):020:0> get 'user', 'rk0001', {COLUMN => ['info:name', 'data:pic']}
```

5. 指定 rowkey 与列值查询

获取 user 表中 row key 为 rk0001，cell 的值为 zhangsan 的信息

```
hbase(main):030:0> get 'user', 'rk0001', {FILTER => "ValueFilter(=, 'binary:zhangsa
n')"}
```

6. 指定 rowkey 与列值模糊查询

获取 user 表中 row key 为 rk0001，列标示符中含有 a 的信息

```
hbase(main):031:0> get 'user', 'rk0001', {FILTER => "(QualifierFilter(=,'substring:
a'))"}
```

继续插入一批数据

```
hbase(main):032:0> put 'user', 'rk0002', 'info:name', 'fanbingbing'
hbase(main):033:0> put 'user', 'rk0002', 'info:gender', 'female'
hbase(main):034:0> put 'user', 'rk0002', 'info:nationality', '中国'
hbase(main):035:0> get 'user', 'rk0002', {FILTER => "ValueFilter(=, 'binary:中国')"}
```

7. 查询所有数据

查询 user 表中的所有信息

```
scan 'user'
```

8. 列族查询

查询 user 表中列族为 info 的信息

```
scan 'user', {COLUMNS => 'info'}
scan 'user', {COLUMNS => 'info', RAW => true, VERSIONS => 5}
scan 'user', {COLUMNS => 'info', RAW => true, VERSIONS => 3}
```

9. 多列族查询

查询 user 表中列族为 info 和 data 的信息

```
scan 'user', {COLUMNS => ['info', 'data']}
scan 'user', {COLUMNS => ['info:name', 'data:pic']}
```

10. 指定列族与某个列名查询

查询 user 表中列族为 info、列标示符为 name 的信息

```
scan 'user', {COLUMNS => 'info:name'}
```

11. 指定列族与列名以及限定版本查询

查询 user 表中列族为 info、列标示符为 name 的信息,并且版本最新的 5 个

```
scan 'user', {COLUMNS => 'info:name', VERSIONS => 5}
```

12. 指定多个列族与按照数据值模糊查询

查询 user 表中列族为 info 和 data 且列标示符中含有 a 字符的信息

```
scan 'user', {COLUMNS => ['info', 'data'], FILTER => "(QualifierFilter(=,'substring
a'))"}
```

13. rowkey 的范围值查询

查询 user 表中列族为 info，rk 范围是(rk0001, rk0003)的数据

```
scan 'user', {COLUMNS => 'info', STARTROW => 'rk0001', ENDROW => 'rk0003'}
```

14. 指定 rowkey 模糊查询

查询 user 表中 row key 以 rk 字符开头的

```
scan 'user',{FILTER=>"PrefixFilter('rk')"}
```

15. 指定数据范围值查询

查询 user 表中指定范围的数据

```
scan 'user', {TIMERANGE => [1392368783980, 1392380169184]}
```

16. 统计一张表有多少行数据

```
count 'user'
```

3）更新操作

1. 更新数据值

更新操作同插入操作一模一样，只不过有数据就更新，没数据就添加。

2. 更新版本号

将 user 表的 f1 列族版本号改为 5

```
hbase(main):050:0> alter 'user', NAME => 'info', VERSIONS => 5
```

4）删除操作

1. 指定 rowkey 以及列名进行删除

删除 user 表 row key 为 rk0001，列标示符为 info:name 的数据

```
hbase(main):045:0> delete 'user', 'rk0001', 'info:name'
```

2. 指定 rowkey，列名以及字段值进行删除

删除 user 表 row key 为 rk0001，列标示符为 info:name，timestamp 为 1392383705316 的数据

```
delete 'user', 'rk0001', 'info:name', 1392383705316
```

3. 删除一个列族

删除一个列族

```
alter 'user', NAME => 'info', METHOD => 'delete'
```

或者

```
alter 'user', NAME => 'info', METHOD => 'delete'
```

4. 清空表数据

```
hbase(main):017:0> truncate 'user'
```

5. 删除表

首先需要先让该表为 disable 状态，使用命令：

```
hbase(main):049:0> disable 'user
```

然后才能 drop 这个表，使用命令：

```
 hbase(main):050:0> drop 'user'
```

注意：如果直接 drop 表，会报错：Drop the named table. Table must first be disabled

四、HBase 的高级 shell 管理命令

1. status

例如：显示服务器状态

```
hbase(main):058:0> status 'node01'
```

2. whoami

显示 HBase 当前用户，例如：

```
hbase> whoami
```

3. list

显示当前所有的表

```
hbase> list
```

4. count

统计指定表的记录数，例如：

```
hbase> count 'user'
```

5. describe

展示表结构信息

```
hbase> describe 'user'
```

6. exists

检查表是否存在，适用于表量特别多的情况

```
hbase> exists 'user'
```

7. is_enabled、is_disabled

检查表是否启用或禁用

```
hbase> is_enabled 'user'
```

8. alter

该命令可以改变表和列族的模式，例如：
为当前表增加列族：

```
hbase> alter 'user', NAME => 'CF2', VERSIONS => 2
```

为当前表删除列族：

```
hbase(main):002:0>  alter 'user', 'delete' => 'CF2'
```

9. disable/enable

禁用一张表/启用一张表

10. drop

删除一张表，记得在删除表之前必须先禁用

11. truncate

清空表

五、HBase 的 Java API 开发

1. 开发 javaAPI 操作 HBase 表数据

1. 创建表 myuser

```
@Test
    public void createTable() throws IOException {
        //创建配置文件对象，并指定zookeeper的连接地址
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.property.clientPort", "2181");
        configuration.set("hbase.zookeeper.quorum", "node01,node02,node03");
        //集群配置↓
        //configuration.set("hbase.zookeeper.quorum", "101.236.39.141,101.236.46.11
4,101.236.46.113");
        configuration.set("hbase.master", "node01:60000");

        Connection connection = ConnectionFactory.createConnection(configuration);
        Admin admin = connection.getAdmin();
        //通过HTableDescriptor来实现我们表的参数设置，包括表名，列族等等
        HTableDescriptor hTableDescriptor = new HTableDescriptor(TableName.valueOf(
"myuser"));
        //添加列族
        hTableDescriptor.addFamily(new HColumnDescriptor("f1"));
        //添加列族
        hTableDescriptor.addFamily(new HColumnDescriptor("f2"));
        //创建表
        boolean myuser = admin.tableExists(TableName.valueOf("myuser"));
        if(!myuser){
            admin.createTable(hTableDescriptor);
        }
        //关闭客户端连接
        admin.close();
    }
```

2. 向表中添加数据

```
@Test
    public  void  addDatas() throws IOException {
        //获取连接
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum", "node01:2181,node02:2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        //获取表
        Table myuser = connection.getTable(TableName.valueOf("myuser"));
        //创建put对象，并指定rowkey
        Put put = new Put("0001".getBytes());
        put.addColumn("f1".getBytes(),"id".getBytes(), Bytes.toBytes(1));
        put.addColumn("f1".getBytes(),"name".getBytes(), Bytes.toBytes("张三"));
```

```
        put.addColumn("f1".getBytes(),"age".getBytes(), Bytes.toBytes(18));

        put.addColumn("f2".getBytes(),"address".getBytes(), Bytes.toBytes("地球人"));
        put.addColumn("f2".getBytes(),"phone".getBytes(), Bytes.toBytes("15874102589"));
        //插入数据
        myuser.put(put);
        //关闭表
        myuser.close();
    }
```

3. 查询数据

初始化一批数据到 HBase 当中用于查询

```
@Test
    public void insertBatchData() throws IOException {

        //获取连接
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum", "node01:2181,node02:2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        //获取表
        Table myuser = connection.getTable(TableName.valueOf("myuser"));
        //创建 put 对象，并指定 rowkey
        Put put = new Put("0002".getBytes());
        put.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(1));
        put.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("曹操"));
        put.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(30));
        put.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));
        put.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("沛国谯县"));
        put.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("16888888888"));
        put.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("helloworld"));

        Put put2 = new Put("0003".getBytes());
        put2.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(2));
        put2.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("刘备"));
        put2.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(32));
        put2.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));
        put2.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("幽州涿郡涿县"));
        put2.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("17888888888"));
```

```
        put2.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("talk is chea
p , show me the code"));


        Put put3 = new Put("0004".getBytes());
        put3.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(3));
        put3.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("孙权"));
        put3.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(35));
        put3.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));
        put3.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("下邳"));
        put3.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("1288888888
8"));
        put3.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("what are you
 弄啥嘞！"));

        Put put4 = new Put("0005".getBytes());
        put4.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(4));
        put4.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("诸葛亮"));
        put4.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(28));
        put4.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));
        put4.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("四川隆中
"));
        put4.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("1488888888
8"));
        put4.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("出师表你背了嘛
"));

        Put put5 = new Put("0005".getBytes());
        put5.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(5));
        put5.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("司马懿"));
        put5.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(27));
        put5.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));
        put5.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("哪里人有
待考究"));
        put5.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("1588888888
8"));
        put5.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("跟诸葛亮死掐
"));


        Put put6 = new Put("0006".getBytes());
        put6.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(5));
        put6.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("xiaobubu—吕
布"));
```

```
        put6.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(28));
        put6.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));
        put6.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("内蒙人"));
        put6.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("15788888888"));
        put6.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("貂蝉去哪了"));

        List<Put> listPut = new ArrayList<Put>();
        listPut.add(put);
        listPut.add(put2);
        listPut.add(put3);
        listPut.add(put4);
        listPut.add(put5);
        listPut.add(put6);

        myuser.put(listPut);
        myuser.close();
    }
```

按照 rowkey 进行查询获取所有列的所有值

查询主键 rowkey 为 0003 的人：

```
 @Test
    public  void searchData() throws IOException {
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        Table myuser = connection.getTable(TableName.valueOf("myuser"));

        Get get = new Get(Bytes.toBytes("0003"));
        Result result = myuser.get(get);
        Cell[] cells = result.rawCells();
        //获取所有的列名称以及列的值
        for (Cell cell : cells) {
            //注意，如果列属性是 int 类型，那么这里就不会显示
            System.out.println(Bytes.toString(cell.getQualifierArray(),cell.getQualifierOffset(),cell.getQualifierLength()));
            System.out.println(Bytes.toString(cell.getValueArray(),cell.getValueOffset(),cell.getValueLength()));
        }

        myuser.close();
}
```

按照 rowkey 查询指定列族下面的指定列的值：

```
@Test
    public void searchData2() throws IOException {
        //获取连接
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:
2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        Table myuser = connection.getTable(TableName.valueOf("myuser"));
        //通过 rowKey 进行查询
        Get get = new Get("0003".getBytes());
get.addColumn("f1".getBytes(),"id".getBytes());

        Result result = myuser.get(get);
        System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".getByt
es())));
        System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "age".getBy
tes())));
        System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "name".g
etBytes())));
        myuser.close();
}
```

通过 startRowKey 和 endRowKey 进行扫描：

```
 @Test
    public  void scanRowKey() throws IOException {
        //获取连接
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:
2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        Table myuser = connection.getTable(TableName.valueOf("myuser"));
        Scan scan = new Scan();
        scan.setStartRow("0004".getBytes());
        scan.setStopRow("0006".getBytes());
        ResultScanner resultScanner = myuser.getScanner(scan);
        for (Result result : resultScanner) {
            //获取 rowkey
            System.out.println(Bytes.toString(result.getRow()));
            //遍历获取得到所有的列族以及所有的列的名称
            KeyValue[] raw = result.raw();
            for (KeyValue keyValue : raw) {
                //获取所属列族
                System.out.println(Bytes.toString(keyValue.getFamilyArray(),keyValu
e.getFamilyOffset(),keyValue.getFamilyLength()));
```

```
                System.out.println(Bytes.toString(keyValue.getQualifierArray(),keyV
alue.getQualifierOffset(),keyValue.getQualifierLength()));
            }
            //指定列族以及列打印列当中的数据出来
            System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".ge
tBytes())));
            System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "age".g
etBytes())));
            System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "nam
e".getBytes())));
        }
        myuser.close();
    }
```

通过 scan 进行全表扫描：

```
@Test
    public void scanAllData() throws IOException {
        //获取连接
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:
2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        Table myuser = connection.getTable(TableName.valueOf("myuser"));

        Scan scan = new Scan();
        ResultScanner resultScanner = myuser.getScanner(scan);
        for (Result result : resultScanner) {
            //获取rowkey
            System.out.println(Bytes.toString(result.getRow()));

            //指定列族以及列打印列当中的数据出来
            System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".ge
tBytes())));
            System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "age".g
etBytes())));
            System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "nam
e".getBytes())));
        }
        myuser.close();
    }
```

2. 过滤器查询

过滤器的类型很多，但是可以分为两大类——**比较过滤器，专用过滤器**。

过滤器的作用是在服务端判断数据是否满足条件，然后只将满足条件的数据返回给客户端；

hbase 过滤器的比较运算符：

```
LESS  <
LESS_OR_EQUAL <=
EQUAL =
NOT_EQUAL <>
GREATER_OR_EQUAL >=
GREATER >
NO_OP 排除所有
```

Hbase 过滤器的比较器（指定比较机制）：

```
BinaryComparator  按字节索引顺序比较指定字节数组，采用 Bytes.compareTo(byte[])
BinaryPrefixComparator 跟前面相同，只是比较左端的数据是否相同
NullComparator 判断给定的是否为空
BitComparator 按位比较
RegexStringComparator 提供一个正则的比较器，仅支持 EQUAL 和非 EQUAL
SubstringComparator 判断提供的子串是否出现在 value 中。
```

1） 比较过滤器

1. rowKey 过滤器 RowFilter

通过 RowFilter 过滤比 rowKey 0003 小的所有值出来

```
@Test
    public  void rowKeyFilter() throws IOException {
        //获取连接
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:
2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        Table myuser = connection.getTable(TableName.valueOf("myuser"));

        Scan scan = new Scan();
        RowFilter rowFilter = new RowFilter(CompareFilter.CompareOp.LESS_OR_EQUAL,
new BinaryComparator(Bytes.toBytes("0003")));

        scan.setFilter(rowFilter);
        ResultScanner resultScanner = myuser.getScanner(scan);
        for (Result result : resultScanner) {
            //获取rowkey
```

```
            System.out.println(Bytes.toString(result.getRow()));

            //指定列族以及列打印列当中的数据出来
            System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".ge
tBytes())));
            System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "age".g
etBytes())));
            System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "nam
e".getBytes())));
        }
        myuser.close();
    }
```

2. 列族过滤器 FamilyFilter

查询比 f2 列族小的所有的列族内的数据

```
@Test
    public  void familyFilter() throws IOException {
        //获取连接
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:
2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        Table myuser = connection.getTable(TableName.valueOf("myuser"));
        Scan scan = new Scan();
        FamilyFilter familyFilter = new FamilyFilter(CompareFilter.CompareOp.LESS,
new SubstringComparator("f2"));
        scan.setFilter(familyFilter);
        ResultScanner resultScanner = myuser.getScanner(scan);
        for (Result result : resultScanner) {
            //获取rowkey
            System.out.println(Bytes.toString(result.getRow()));
            //指定列族以及列打印列当中的数据出来
            System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".ge
tBytes())));
            System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "age".g
etBytes())));
            System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "nam
e".getBytes())));
        }
        myuser.close();
    }
```

3. 列过滤器 QualifierFilter

只查询 name 列的值

```
@Test
public  void qualifierFilter() throws IOException {
        //获取连接
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:
2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        Table myuser = connection.getTable(TableName.valueOf("myuser"));
        Scan scan = new Scan();
        QualifierFilter qualifierFilter = new QualifierFilter(CompareFilter.Compare
Op.EQUAL, new SubstringComparator("name"));
        scan.setFilter(qualifierFilter);
        ResultScanner resultScanner = myuser.getScanner(scan);
        for (Result result : resultScanner) {
            //获取rowkey
            System.out.println(Bytes.toString(result.getRow()));
            //指定列族以及列打印列当中的数据出来
        //    System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".
getBytes())));
            System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "nam
e".getBytes())));
        }
        myuser.close();
}
```

4. 列值过滤器 ValueFilter

查询所有列当中包含 8 的数据

```
@Test
public  void valueFilter() throws IOException {
        //获取连接
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:
2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        Table myuser = connection.getTable(TableName.valueOf("myuser"));
        Scan scan = new Scan();
        ValueFilter valueFilter = new ValueFilter(CompareFilter.CompareOp.EQUAL, ne
w SubstringComparator("8"));

        scan.setFilter(valueFilter);
        ResultScanner resultScanner = myuser.getScanner(scan);
```

```
        for (Result result : resultScanner) {
            //获取rowkey
            System.out.println(Bytes.toString(result.getRow()));
            //指定列族以及列打印列当中的数据出来
            //    System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "
id".getBytes())));
            System.out.println(Bytes.toString(result.getValue("f2".getBytes(), "pho
ne".getBytes())));
        }
        myuser.close();
}
```

2）专用过滤器

1. 单列值过滤器 SingleColumnValueFilter

SingleColumnValueFilter 会返回满足条件的整列值的所有字段

```
@Test
public void singleColumnFilter() throws IOException {
        //获取连接
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:
2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        Table myuser = connection.getTable(TableName.valueOf("myuser"));
        Scan scan = new Scan();
        SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilt
er("f1".getBytes(), "name".getBytes(), CompareFilter.CompareOp.EQUAL, "刘备
".getBytes());
        scan.setFilter(singleColumnValueFilter);
        ResultScanner resultScanner = myuser.getScanner(scan);
        for (Result result : resultScanner) {
            //获取rowkey
            System.out.println(Bytes.toString(result.getRow()));
            //指定列族以及列打印列当中的数据出来
            System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".ge
tBytes())));
            System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "nam
e".getBytes())));
            System.out.println(Bytes.toString(result.getValue("f2".getBytes(), "pho
ne".getBytes())));
        }
```

```
        myuser.close();
}
```

2. 列值排除过滤器 SingleColumnValueExcludeFilter

与 SingleColumnValueFilter 相反，会排除掉指定的列，其他的列全部返回

3. rowkey 前缀过滤器 PrefixFilter

查询以 00 开头的所有前缀的 rowkey

```
@Test
public void preFilter() throws IOException {

        //获取连接
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:
2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        Table myuser = connection.getTable(TableName.valueOf("myuser"));
        Scan scan = new Scan();
        PrefixFilter prefixFilter = new PrefixFilter("00".getBytes());
        scan.setFilter(prefixFilter);
        ResultScanner resultScanner = myuser.getScanner(scan);
        for (Result result : resultScanner) {
            //获取rowkey
            System.out.println(Bytes.toString(result.getRow()));
            //指定列族以及列打印列当中的数据出来
            System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".ge
tBytes())));
            System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "nam
e".getBytes())));
            System.out.println(Bytes.toString(result.getValue("f2".getBytes(), "pho
ne".getBytes())));
        }
        myuser.close();
}
```

4. 分页过滤器 PageFilter

分页过滤器 PageFilter

```
@Test
public void pageFilter2() throws IOException {
        //获取连接
```

```
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum", "node01:2181,node02:2181,node03:
2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        Table myuser = connection.getTable(TableName.valueOf("myuser"));
        int pageNum = 3;
        int pageSize = 2;
        Scan scan = new Scan();
        if (pageNum == 1) {
            PageFilter filter = new PageFilter(pageSize);
            scan.setStartRow(Bytes.toBytes(""));
            scan.setFilter(filter);
            scan.setMaxResultSize(pageSize);
            ResultScanner scanner = myuser.getScanner(scan);
            for (Result result : scanner) {
                //获取rowkey
                System.out.println(Bytes.toString(result.getRow()));
                //指定列族以及列打印列当中的数据出来
//              System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".
getBytes())));
                System.out.println(Bytes.toString(result.getValue("f1".getBytes(),
"name".getBytes())));
                //System.out.println(Bytes.toString(result.getValue("f2".getBytes(),
 "phone".getBytes())));
            }

        }else{
            String startRowKey ="";
            PageFilter filter = new PageFilter((pageNum - 1) * pageSize + 1  );
            scan.setStartRow(startRowKey.getBytes());
            scan.setMaxResultSize((pageNum - 1) * pageSize + 1);
            scan.setFilter(filter);
            ResultScanner scanner = myuser.getScanner(scan);
            for (Result result : scanner) {
                byte[] row = result.getRow();
                startRowKey =  new String(row);
            }
            Scan scan2 = new Scan();
            scan2.setStartRow(startRowKey.getBytes());
            scan2.setMaxResultSize(Long.valueOf(pageSize));
            PageFilter filter2 = new PageFilter(pageSize);
            scan2.setFilter(filter2);

            ResultScanner scanner1 = myuser.getScanner(scan2);
```

```
            for (Result result : scanner1) {
                byte[] row = result.getRow();
                System.out.println(new String(row));
            }
        }
        myuser.close();
}
```

3）多过滤器综合查询 FilterList

需求：使用 SingleColumnValueFilter 查询 f1 列族，name 为刘备的数据，并且同时满足 rowkey 的前缀以 00 开头的数据（PrefixFilter）

```
@Test
public void manyFilter() throws IOException {
        //获取连接
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum", "node01:2181,node02:2181,node03:
2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        Table myuser = connection.getTable(TableName.valueOf("myuser"));
        Scan scan = new Scan();
        FilterList filterList = new FilterList();

        SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilt
er("f1".getBytes(), "name".getBytes(), CompareFilter.CompareOp.EQUAL, "刘备
".getBytes());
        PrefixFilter prefixFilter = new PrefixFilter("00".getBytes());
        filterList.addFilter(singleColumnValueFilter);
        filterList.addFilter(prefixFilter);
        scan.setFilter(filterList);
        ResultScanner scanner = myuser.getScanner(scan);
        for (Result result : scanner) {
            //获取rowkey
            System.out.println(Bytes.toString(result.getRow()));
            //指定列族以及列打印列当中的数据出来
//            System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".
getBytes())));
            System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "nam
e".getBytes())));
            //System.out.println(Bytes.toString(result.getValue("f2".getBytes(), "p
hone".getBytes())));
        }
```

```
        myuser.close();
}
```

3. 根据 rowkey 删除数据

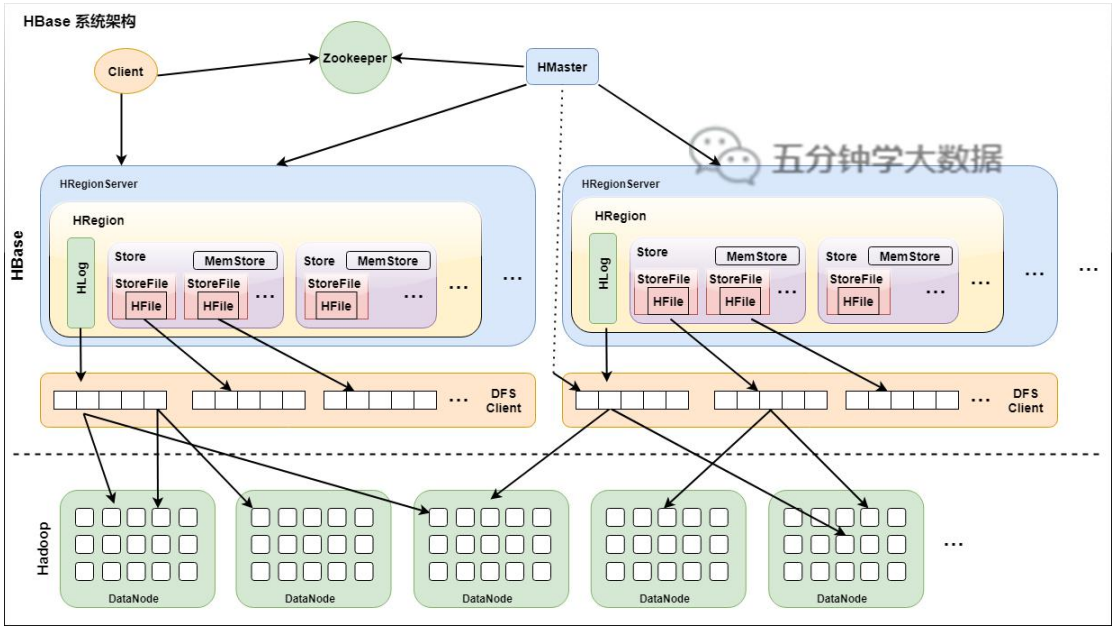
```
@Test
public  void  deleteByRowKey() throws IOException {
        //获取连接
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:
2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        Table myuser = connection.getTable(TableName.valueOf("myuser"));
        Delete delete = new Delete("0001".getBytes());
        myuser.delete(delete);
        myuser.close();
}
```

4. 删除表操作

```
@Test
public void  deleteTable() throws IOException {
        //获取连接
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum","node01:2181,node02:2181,node03:
2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        Admin admin = connection.getAdmin();
        admin.disableTable(TableName.valueOf("myuser"));
        admin.deleteTable(TableName.valueOf("myuser"));
        admin.close();
}
```

六、HBase 底层原理

1. 系统架构



根据这幅图，解释下 HBase 中各个组件

1) Client

1. 包含访问 hbase 的接口，Client 维护着一些 cache 来加快对 hbase 的访问，比如 regione 的位置信息.

2) Zookeeper

HBase 可以使用内置的 Zookeeper，也可以使用外置的，在实际生产环境，为了保持统一性，一般使用外置 Zookeeper。

Zookeeper 在 HBase 中的作用：

1. 保证任何时候，集群中只有一个 master
2. 存贮所有 Region 的寻址入口
3. 实时监控 Region Server 的状态，将 Region server 的上线和下线信息实时通知给 Master

3) HMaster

1. 为 Region server 分配 region
2. 负责 region server 的负载均衡

3. 发现失效的 region server 并重新分配其上的 region
4. HDFS 上的垃圾文件回收
5. 处理 schema 更新请求

4）HRegion Server

1. HRegion server 维护 HMaster 分配给它的 region，处理对这些 region 的 IO 请求
2. HRegion server 负责切分在运行过程中变得过大的 region 从图中可以看到，Client 访问 HBase 上数据的过程并不需要 HMaster 参与（寻址访问 Zookeeper 和 HRegion server，数据读写访问 HRegione server）

HMaster 仅仅维护者 table 和 HRegion 的元数据信息，负载很低。

2. HBase 的表数据模型

RowKey(行键)	Column Family1(列簇)		Column Family2(列簇)		timestamp(时间戳)
	name(列)	age(列)	salary(列)	role(列)	
rk001	zhangsan	26	12000	CTO	t1
rk002	lisi	18			t2
rk003	wangwu	35	32000		t3
rk004	zhaoliu	31		CEO	t4
rk005	sun	31		staff	t5
rk006		43			t6
rk007	yang				t7

HBase 的表结构

1）行键 Row Key

与 nosql 数据库一样,row key 是用来检索记录的主键。访问 hbase table 中的行，只有三种方式：

1. 通过单个 row key 访问
2. 通过 row key 的 range
3. 全表扫描

Row Key 行键可以是任意字符串(**最大长度是 64KB**，实际应用中长度一般为 10-100bytes)，在 hbase 内部，row key 保存为字节数组。

Hbase 会对表中的数据按照 rowkey 排序(字典顺序)

存储时，数据按照 Row key 的字典序(byte order)排序存储。设计 key 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)。

注意： 字典序对 int 排序的结果是

1,10,100,11,12,13,14,15,16,17,18,19,2,20,21 ... 。**要保持整形的自然序，行键必须用 0 作左填充。**

行的一次读写是原子操作 (不论一次读写多少列)。这个设计决策能够使用户很容易的理解程序在对同一个行进行并发更新操作时的行为。

2) 列族 Column Family

HBase 表中的每个列，都归属于某个列族。列族是表的 schema 的一部分(而列不是)，**必须在使用表之前定义**。

列名都以列族作为前缀。例如 courses:history ， courses:math 都属于 courses 这个列族。

访问控制、磁盘和内存的使用统计都是在列族层面进行的。 列族越多，在取一行数据时所要参与 IO、搜寻的文件就越多，所以，如果没有必要，不要设置太多的列族。

3) 列 Column

列族下面的具体列，属于某一个 ColumnFamily，类似于在 mysql 当中创建的具体的列。

4) 时间戳 Timestamp

HBase 中通过 row 和 columns 确定的为一个存贮单元称为 cell。每个 cell 都保存着同一份数据的多个版本。版本通过时间戳来索引。时间戳的类型是 64 位整型。**时间戳可以由 hbase(在数据写入时自动)赋值**，此时时间戳是精确到毫秒的当前系统时间。时间戳也可以由客户显式赋值。如果应用程序要避免数据版本冲突，就必须自己生成具有唯一性的时间戳。**每个 cell 中，不同版本的数据按照时间倒序排序**，即最新的数据排在最前面。

为了避免数据存在过多版本造成的的管理 (包括存贮和索引)负担，hbase 提供了两种数据版本回收方式：

1. 保存数据的最后 n 个版本
2. 保存最近一段时间内的版本（设置数据的生命周期 TTL）。

用户可以针对每个列族进行设置。

5) 单元 Cell

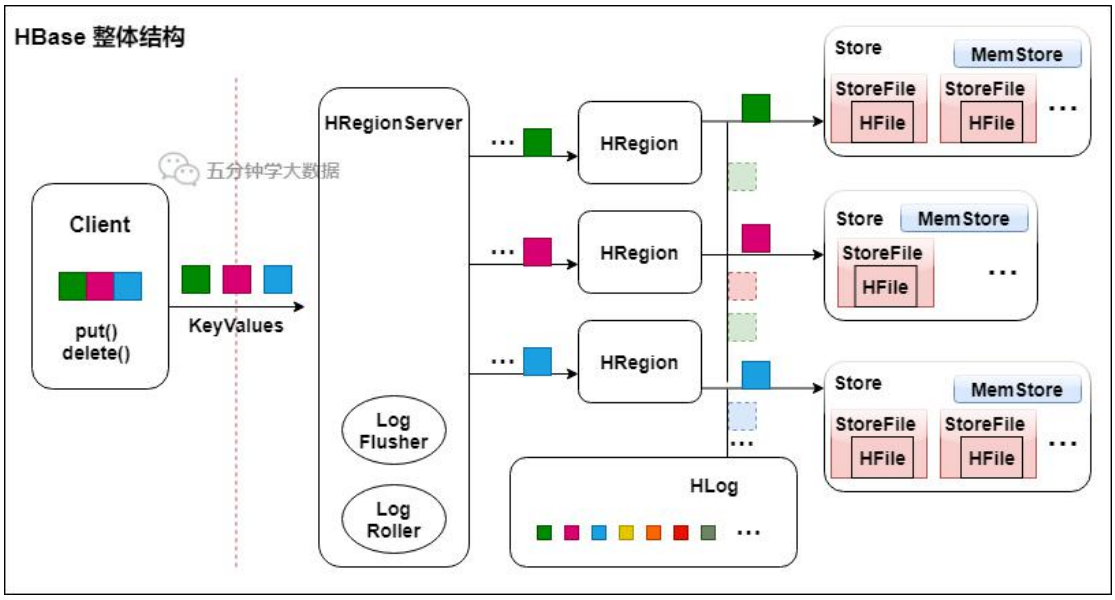
由{row key, column(=<family> + <label>), version} 唯一确定的单元。 cell 中的数据是没有类型的，全部是字节码形式存贮。

6) 版本号 VersionNum

数据的版本号，每条数据可以有多个版本号，默认值为系统时间戳，类型为 Long。

3. 物理存储

1) 整体结构



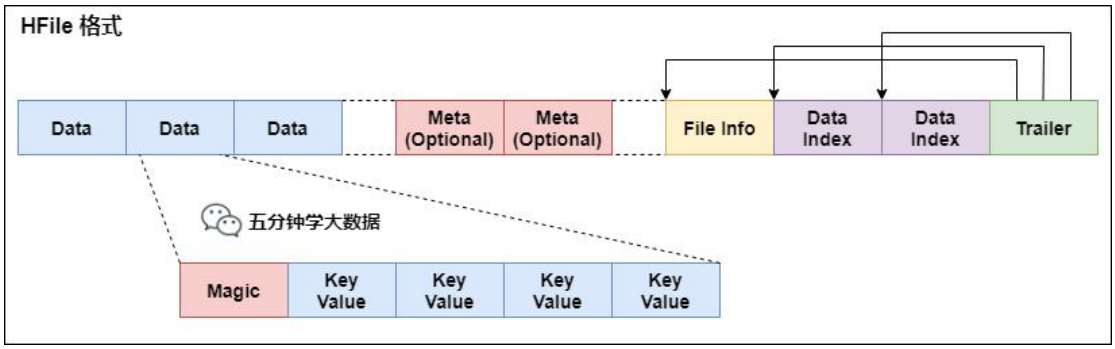
HBase 整体结构

1. Table 中的所有行都按照 Row Key 的字典序排列。
2. Table 在行的方向上分割为多个 HRegion。
3. HRegion 按大小分割的(默认 10G)，每个表一开始只有一 个 HRegion，随着数据不断插入表，HRegion 不断增大，当增大到一个阀值的时候，HRegion 就会等分会两个新的 HRegion。当 Table 中的行不断增多，就会有越来越多的 HRegion。
4. HRegion 是 HBase 中分布式存储和负载均衡的最小单元。最小单元就表示不同的 HRegion 可以分布在不同的 HRegion Server 上。但一个 HRegion 是不会拆分到多个 Server 上的。
5. HRegion 虽然是负载均衡的最小单元，但并不是物理存储的最小单元。 事实上，HRegion 由一个或者多个 Store 组成，每个 Store 保存一个 Column Family。 每个 Strore 又由一个 MemStore 和 0 至多个 StoreFile 组成。如上图。

2) StoreFile 和 HFile 结构

StoreFile 以 HFile 格式保存在 HDFS 上。

HFile 的格式为：



HFile 格式

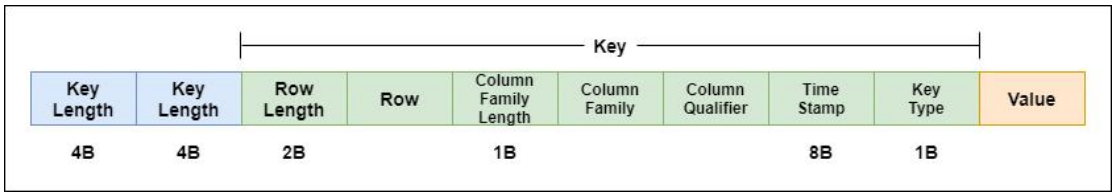
首先 HFile 文件是不定长的，长度固定的只有其中的两块：Trailer 和 FileInfo。正如图中所示的，Trailer 中有指针指向其他数 据块的起始点。

File Info 中记录了文件的一些 Meta 信息，例如：AVG_KEY_LEN, AVG_VALUE_LEN, LAST_KEY, COMPARATOR, MAX_SEQ_ID_KEY 等。

Data Index 和 Meta Index 块记录了每个 Data 块和 Meta 块的起始点。

Data Block 是 HBase I/O 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，大号的 Block 有利于顺序 Scan，小号 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。

HFile 里面的每个 KeyValue 对就是一个简单的 byte 数组。但是这个 byte 数组里面包含了很多项，并且有固定的结构。我们来看看里面的具体结构：



HFile 具体结构

开始是两个固定长度的数值，分别表示 Key 的长度和 Value 的长度。紧接着是 Key，开始是固定长度的数值，表示 RowKey 的长度，紧接着是 RowKey，然后是固定长度的数值，表示 Family 的长度，然后是 Family，接着是 Qualifier，然后是两个固定长度的数值，表示 Time Stamp 和 Key Type（Put/Delete）。Value 部分没有这么复杂的结构，就是纯粹的二进制数据了。

HFile 分为六个部分：

1. Data Block 段 – 保存表中的数据，这部分可以被压缩.
2. Meta Block 段 (可选的) – 保存用户自定义的 kv 对，可以被压缩。

3. File Info 段 - Hfile 的元信息，不被压缩，用户也可以在这一部分添加自己的元信息。
4. Data Block Index 段 - Data Block 的索引。每条索引的 key 是被索引的 block 的第一条记录的 key。
5. Meta Block Index 段 (可选的) - Meta Block 的索引。
6. 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。目前 HFile 的压缩支持两种方式：Gzip，Lzo。

3) Memstore 与 StoreFile

一个 HRegion 由多个 Store 组成，每个 Store 包含一个列族的所有数据 Store 包括位于内存的 Memstore 和位于硬盘的 StoreFile。

写操作先写入 Memstore，当 Memstore 中的数据量达到某个阈值，HRegionServer 启动 FlashCache 进程写入 StoreFile,每次写入形成单独一个 StoreFile

当 StoreFile 大小超过一定阈值后，会把当前的 HRegion 分割成两个，并由 HMaster 分配给相应的 HRegion 服务器，实现负载均衡

客户端检索数据时，先在 memstore 找，找不到再找 storefile。

4) HLog(WAL log)

WAL 意为 Write ahead log，类似 mysql 中的 binlog,用来 做灾难恢复时用，Hlog 记录数据的所有变更,一旦数据修改，就可以从 log 中进行恢复。

每个 Region Server 维护一个 Hlog,而不是每个 Region 一个。这样不同 region(来自不同 table)的日志会混在一起，这样做的目的是不断追加单个文件相对于同

时写多个文件而言，可以减少磁盘寻址次数，**因此可以提高对 table 的写性能**。带来的麻烦是，如果一台 region server 下线，为了**恢复其上的 region，需要将 region server 上的 log 进行拆分**，然后分发到其它 region server 上进行恢复。**HLog 文件就是一个普通的 Hadoop Sequence File：**

1. HLog Sequence File 的 Key 是 HLogKey 对象，HLogKey 中记录了写入数据的归属信息，除了 table 和 region 名字外，同时还包括 sequence number 和 timestamp，timestamp 是”写入时间”，sequence number 的起始值为 0，或者是最近一次存入文件系统中 sequence number。
2. HLog Sequece File 的 Value 是 HBase 的 KeyValue 对象，即对应 HFile 中的 KeyValue，可参见上文描述。

4. 读写过程

1）读请求过程：

HRegionServer 保存着 meta 表以及表数据，要访问表数据，首先 Client 先去访问 zookeeper，从 zookeeper 里面获取 meta 表所在的位置信息，即找到这个 meta 表在哪个 HRegionServer 上保存着。

接着 Client 通过刚才获取到的 HRegionServer 的 IP 来访问 Meta 表所在的 HRegionServer，从而读取到 Meta，进而获取到 Meta 表中存放的元数据。

Client 通过元数据中存储的信息，访问对应的 HRegionServer，然后扫描所在 HRegionServer 的 Memstore 和 Storefile 来查询数据。

最后 HRegionServer 把查询到的数据响应给 Client。

查看 meta 表信息

```
hbase(main):011:0> scan 'hbase:meta'
```

2）写请求过程：

Client 也是先访问 zookeeper，找到 Meta 表，并获取 Meta 表元数据。

确定当前将要写入的数据所对应的 HRegion 和 HRegionServer 服务器。

Client 向该 HRegionServer 服务器发起写入数据请求，然后 HRegionServer 收到请求并响应。

Client 先把数据写入到 HLog，以防止数据丢失。

然后将数据写入到 Memstore。

如果 HLog 和 Memstore 均写入成功，则这条数据写入成功

如果 Memstore 达到阈值，会把 Memstore 中的数据 flush 到 Storefile 中。

当 Storefile 越来越多，会触发 Compact 合并操作，把过多的 Storefile 合并成一个大的 Storefile。

当 Storefile 越来越大，Region 也会越来越大，达到阈值后，会触发 Split 操作，将 Region 一分为二。

细节描述：

HBase 使用 MemStore 和 StoreFile 存储对表的更新。 数据在更新时首先写入 Log(WAL log)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。 当系统出现意外时，可能导致内存(MemStore)中的数据丢失，此时使用 Log(WAL log)来恢复 checkpoint 之后的数据。

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(minor_compact, 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 split，等分为两个 StoreFile。

由于对表的更新是不断追加的,compact 时,需要访问 Store 中全部的 StoreFile 和 MemStore，将他们按 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，合并的过程还是比较快。

5. HRegion 管理

1) HRegion 分配

任何时刻，**一个 HRegion 只能分配给一个 HRegion Server**。HMaster 记录了当前有哪些可用的 HRegion Server。以及当前哪些 HRegion 分配给了哪些 HRegion Server，哪些 HRegion 还没有分配。当需要分配的新的 HRegion，并且有一个 HRegion Server 上有可用空间时，HMaster 就给这个 HRegion Server 发送一个

装载请求，把 HRegion 分配给这个 HRegion Server。HRegion Server 得到请求后，就开始对此 HRegion 提供服务。

2）HRegion Server 上线

HMaster 使用 zookeeper 来跟踪 HRegion Server 状态。当某个 HRegion Server 启动时，会首先在 zookeeper 上的 server 目录下建立代表自己的 znode。由于 HMaster 订阅了 server 目录上的变更消息，当 server 目录下的文件出现新增或删除操作时，HMaster 可以得到来自 zookeeper 的实时通知。因此一旦 HRegion Server 上线，HMaster 能马上得到消息。

3）HRegion Server 下线

当 HRegion Server 下线时，它和 zookeeper 的会话断开，zookeeper 而自动释放代表这台 server 的文件上的独占锁。HMaster 就可以确定：

1. HRegion Server 和 zookeeper 之间的网络断开了。
2. HRegion Server 挂了。

无论哪种情况，HRegion Server 都无法继续为它的 HRegion 提供服务了，此时 HMaster 会删除 server 目录下代表这台 HRegion Server 的 znode 数据，并将这台 HRegion Server 的 HRegion 分配给其它还活着的节点。

6. HMaster 工作机制

1）master 上线

master 启动进行以下步骤:

1. 从 zookeeper 上获取唯一一个代表 active master 的锁，用来阻止其它 HMaster 成为 master。
2. 扫描 zookeeper 上的 server 父节点，获得当前可用的 HRegion Server 列表。
3. 和每个 HRegion Server 通信，获得当前已分配的 HRegion 和 HRegion Server 的对应关系。

4. 扫描.META.region 的集合，计算得到当前还未分配的 HRegion，将他们放入待分配 HRegion 列表。

2）master 下线

由于 HMaster 只维护表和 region 的元数据，而不参与表数据 IO 的过程，HMaster 下线仅导致所有元数据的修改被冻结(无法创建删除表，无法修改表的 schema，无法进行 HRegion 的负载均衡，无法处理 HRegion 上下线，无法进行 HRegion 的合并,唯一例外的是 HRegion 的 split 可以正常进行,因为只有 HRegion Server 参与)，表的数据读写还可以正常进行。因此 HMaster 下线短时间内对整个 HBase 集群没有影响。

从上线过程可以看到，HMaster 保存的信息全是可以冗余信息（都可以从系统其它地方收集到或者计算出来）

因此，一般 HBase 集群中总是有一个 HMaster 在提供服务，还有一个以上的‘HMaster’在等待时机抢占它的位置。

7. HBase 三个重要机制

1）flush 机制

1.（hbase.regionserver.global.memstore.size）默认;堆大小的 40% regionServer 的全局 memstore 的大小,超过该大小会触发 flush 到磁盘的操作,默认是堆大小的 40%,而且 regionserver 级别的 flush 会阻塞客户端读写

2.（hbase.hregion.memstore.flush.size）默认：128M 单个 region 里 memstore 的缓存大小，超过那么整个 HRegion 就会 flush,

3.（hbase.regionserver.optionalcacheflushinterval）默认：1h 内存中的文件在自动刷新之前能够存活的最长时间

4.（hbase.regionserver.global.memstore.size.lower.limit）默认：堆大小 * 0.4 * 0.95 有时候集群的“写负载”非常高，写入量一直超过 flush 的量，这时，我们就希望 memstore 不要超过一定的安全设置。在这种情况下，写操作就要被阻塞一直到 memstore 恢复到一个“可管理”的大小，这个大小就是默认值是堆大小 * 0.4 * 0.95，也就是当 regionserver 级别的 flush 操作发送后,会

阻塞客户端写,一直阻塞到整个 regionserver 级别的 memstore 的大小为 堆大小 * 0.4 *0.95 为止

5.（hbase.hregion.preclose.flush.size）默认为：5M 当一个 region 中的 memstore 的大小大于这个值的时候，我们又触发了 region 的 close 时，会先运行“pre-flush”操作，清理这个需要关闭的 memstore，然后 将这个 region 下线。当一个 region 下线了,我们无法再进行任何写操作。如果一个 memstore 很大的时候，flush 操作会消耗很多时间。"pre-flush" 操作意味着在 region 下线之前，会先把 memstore 清空。这样在最终执行 close 操作的时候，flush 操作会很快。

6.（hbase.hstore.compactionThreshold）默认：超过 3 个 一个 store 里面允许存的 hfile 的个数，超过这个个数会被写到新的一个 hfile 里面 也即是每个 region 的每个列族对应的 memstore 在 flush 为 hfile 的时候，默认情况下当超过 3 个 hfile 的时候就会对这些文件进行合并重写为一个新文件,设置个数越大可以减少触发合并的时间，但是每次合并的时间就会越长

2） compact 机制

把小的 storeFile 文件合并成大的 HFile 文件。 清理过期的数据，包括删除的数据 将数据的版本号保存为 1 个。

3） split 机制

当 HRegion 达到阈值,会把过大的 HRegion 一分为二。默认一个 HFile 达到 10Gb 的时候就会进行切分。

七、HBase 与 MapReduce 的集成

HBase 当中的数据最终都是存储在 HDFS 上面的,HBase 天生的支持 MR 的操作,我们可以通过 MR 直接处理 HBase 当中的数据，并且 MR 可以将处理后的结果直接存储到 HBase 当中去。

需求：读取 HBase 当中一张表的数据，然后将数据写入到 HBase 当中的另外一张表当中去。

注意：我们可以使用 TableMapper 与 TableReducer 来实现从 HBase 当中读取与写入数据。

这里我们将 myuser 这张表当中 f1 列族的 name 和 age 字段写入到 myuser2 这张表的 f1 列族当中去。

需求一：读取 myuser 这张表当中的数据写入到 HBase 的另外一张表当中去：

第一步：创建 myuser2 这张表

注意：列族的名字要与 myuser 表的列族名字相同

```
hbase(main):010:0> create 'myuser2','f1'
```

第二步：开发 MR 的程序

```
public class HBaseMR extends Configured implements Tool{


    public static class HBaseMapper extends  TableMapper<Text,Put>{
        /**
         *
         * @param key  我们的主键rowkey
         * @param value  我们一行数据所有列的值都封装在value里面了
         * @param context
         * @throws IOException
         * @throws InterruptedException
         */
        @Override
        protected void map(ImmutableBytesWritable key, Result value, Context contex
t) throws IOException, InterruptedException {
            byte[] bytes = key.get();
            String rowKey = Bytes.toString(bytes);
            Put put = new Put(key.get());
            Cell[] cells = value.rawCells();
            for (Cell cell : cells) {
                if("f1".equals(Bytes.toString(CellUtil.cloneFamily(cell)))){
                    if("name".equals(Bytes.toString(CellUtil.cloneQualifier(cell))))
{
                        put.add(cell);
                    }
                    if("age".equals(Bytes.toString(CellUtil.cloneQualifier(cell))))
{
                        put.add(cell);
                    }
                }
            }
          if(!put.isEmpty()){
                context.write(new Text(rowKey),put);
            }
```

```
        }
    }
    public static class HBaseReducer extends TableReducer<Text,Put,ImmutableBytesWr
itable>{
        @Override
        protected void reduce(Text key, Iterable<Put> values, Context context) thro
ws IOException, InterruptedException {
            for (Put value : values) {
                context.write(null,value);
            }
        }
    }
    @Override
    public int run(String[] args) throws Exception {
        Job job = Job.getInstance(super.getConf(), "hbaseMr");
        job.setJarByClass(this.getClass());
        Scan scan = new Scan();
        scan.setCaching(500);
        scan.setCacheBlocks(false);
        //使用TableMapReduceUtil 工具类来初始化我们的mapper
        TableMapReduceUtil.initTableMapperJob(TableName.valueOf("myuser"),scan,HBas
eMapper.class,Text.class,Put.class,job);
        //使用TableMapReduceUtil 工具类来初始化我们的reducer
        TableMapReduceUtil.initTableReducerJob("myuser2",HBaseReducer.class,job);

        job.setNumReduceTasks(1);

        boolean b = job.waitForCompletion(true);
        return b?0:1;
    }
    public static void main(String[] args) throws Exception {
        //创建HBaseConfiguration配置
        Configuration configuration = HBaseConfiguration.create();
        int run = ToolRunner.run(configuration, new HBaseMR(), args);
        System.exit(run);
    }
}
```

第三步：打包运行

将我们打好的 jar 包放到服务器上执行：

```
yarn jar hbaseStudy-1.0-SNAPSHOT.jar  cn.yuan_more.hbasemr.HBaseMR
```

需求二：读取 HDFS 文件，写入到 HBase 表当中去

第一步：准备数据文件

准备数据文件，并将数据文件上传到 HDFS 上面去。

第二步：开发 MR 程序

```
public class Hdfs2Hbase extends Configured implements Tool{
    @Override
    public int run(String[] args) throws Exception {
        Job job = Job.getInstance(super.getConf(), "hdfs2Hbase");
        job.setJarByClass(Hdfs2Hbase.class);
        job.setInputFormatClass(TextInputFormat.class);
        TextInputFormat.addInputPath(job,new Path("hdfs://node01:8020/hbase/input")
;
        job.setMapperClass(HdfsMapper.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(NullWritable.class);

        TableMapReduceUtil.initTableReducerJob("myuser2",HBaseReducer.class,job);
        job.setNumReduceTasks(1);
        boolean b = job.waitForCompletion(true);

        return b?0:1;
    }


    public static void main(String[] args) throws Exception {
        Configuration configuration = HBaseConfiguration.create();
        int run = ToolRunner.run(configuration, new Hdfs2Hbase(), args);
        System.exit(run);
    }


    public static class HdfsMapper extends Mapper<LongWritable,Text,Text,NullWritab
le>{
        @Override
        protected void map(LongWritable key, Text value, Context context) throws IO
Exception, InterruptedException {
            context.write(value,NullWritable.get());
        }
    }

    public static class HBaseReducer extends TableReducer<Text,NullWritable,Immutab
leBytesWritable>{

        @Override
        protected void reduce(Text key, Iterable<NullWritable> values, Context cont
```

```
ext) throws IOException, InterruptedException {
            String[] split = key.toString().split("\t");
            Put put = new Put(Bytes.toBytes(split[0]));
            put.addColumn("f1".getBytes(),"name".getBytes(),split[1].getBytes());
            put.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(Integer.pa
rseInt(split[2])));
            context.write(new ImmutableBytesWritable(Bytes.toBytes(split[0])),put);
        }
    }
}
```

需求四：通过 bulkload 的方式批量加载数据到 HBase 当中去

加载数据到 HBase 当中去的方式多种多样，我们可以使用 HBase 的 javaAPI 或者使用 sqoop 将我们的数据写入或者导入到 HBase 当中去，但是这些方式不是慢就是在导入的过程的占用 Region 资料导致效率低下，**我们也可以通过 MR 的程序，将我们的数据直接转换成 HBase 的最终存储格式 HFile，然后直接 load 数据到 HBase 当中去即可**。

HBase 中每张 Table 在根目录（/HBase）下用一个文件夹存储，Table 名为文件夹名，在 Table 文件夹下每个 Region 同样用一个文件夹存储，每个 Region 文件夹下的每个列族也用文件夹存储，而每个列族下存储的就是一些 HFile 文件，HFile 就是 HBase 数据在 HFDS 下存储格式，所以 HBase 存储文件最终在 hdfs 上面的表现形式就是 HFile，如果我们可以直接将数据转换为 HFile 的格式，那么我们的 HBase 就可以直接读取加载 HFile 格式的文件，就可以直接读取了。

优点：

1. 导入过程不占用 Region 资源
2. 能快速导入海量的数据
3. 节省内存

第一步：定义 mapper 类

```
public class LoadMapper  extends Mapper<LongWritable,Text,ImmutableBytesWritable,Pu
t>{
    @Override
    protected void map(LongWritable key, Text value, Mapper.Context context) throws
 IOException, InterruptedException {
        String[] split = value.toString().split("\t");
        Put put = new Put(Bytes.toBytes(split[0]));
        put.addColumn("f1".getBytes(),"name".getBytes(),split[1].getBytes());
        put.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(Integer.parseI
nt(split[2])));
```

```
        context.write(new ImmutableBytesWritable(Bytes.toBytes(split[0])),put);
    }
}
```

第二步：开发 main 程序入口类

```
public class HBaseLoad  extends Configured implements Tool {
    @Override
    public int run(String[] args) throws Exception {
        final String INPUT_PATH= "hdfs://node01:8020/hbase/input";
        final String OUTPUT_PATH= "hdfs://node01:8020/hbase/output_hfile";
        Configuration conf = HBaseConfiguration.create();
        Connection connection = ConnectionFactory.createConnection(conf);
        Table table = connection.getTable(TableName.valueOf("myuser2"));
        Job job= Job.getInstance(conf);
        job.setJarByClass(HBaseLoad.class);
        job.setMapperClass(LoadMapper.class);
        job.setMapOutputKeyClass(ImmutableBytesWritable.class);
        job.setMapOutputValueClass(Put.class);
        job.setOutputFormatClass(HFileOutputFormat2.class);
        HFileOutputFormat2.configureIncrementalLoad(job,table,connection.getRegionL
ocator(TableName.valueOf("myuser2")));
        FileInputFormat.addInputPath(job,new Path(INPUT_PATH));
        FileOutputFormat.setOutputPath(job,new Path(OUTPUT_PATH));
        boolean b = job.waitForCompletion(true);
        return b?0:1;
    }

    public static void main(String[] args) throws Exception {
        Configuration configuration = HBaseConfiguration.create();
        int run = ToolRunner.run(configuration, new HBaseLoad(), args);
        System.exit(run);
    }
}
```

第三步：将代码打成 jar 包然后运行

```
yarn jar original-hbaseStudy-1.0-SNAPSHOT.jar  cn.yuan_more.hbasemr.HBaseLoad
```

第四步：开发代码，加载数据

将输出路径下面的 HFile 文件，加载到 hbase 表当中去

```
public class LoadData {
    public static void main(String[] args) throws Exception {
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.property.clientPort", "2181");
        configuration.set("hbase.zookeeper.quorum", "node01,node02,node03");
```

```
        Connection connection =  ConnectionFactory.createConnection(configuration);
        Admin admin = connection.getAdmin();
        Table table = connection.getTable(TableName.valueOf("myuser2"));
        LoadIncrementalHFiles load = new LoadIncrementalHFiles(configuration);
        load.doBulkLoad(new Path("hdfs://node01:8020/hbase/output_hfile"), admin,ta
ble,connection.getRegionLocator(TableName.valueOf("myuser2")));
    }
}
```

或者我们也可以通过命令行来进行加载数据。

先将 hbase 的 jar 包添加到 hadoop 的 classpath 路径下

```
export HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase mapredcp`
```

然后执行以下命令，将 hbase 的 HFile 直接导入到表 myuser2 当中来

```
yarn jar /servers/hbase/lib/hbase-server-1.2.0.jar completebulkload /hbase/output_h
file myuser2
```

八、HBase 的预分区

1. 为何要预分区？

- 增加数据读写效率
- 负载均衡，防止数据倾斜
- 方便集群容灾调度 region
- 优化 Map 数量

2. 如何预分区？

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。

3. 如何设定预分区？

1）手动指定预分区

```
hbase(main):001:0> create 'staff','info','partition1',SPLITS => ['1000','2000','300
0','4000']
```

完成后如图：

Table Regions

Name	Region Server	Start Key	End Key	Requests
table1,,1495642218397.ab492f425a1ca6422073cb3ce19f7d7e.	z01:60020		1000	0
table1,1000,1495642218397.f6b5c80ed2214379f2b1b91b8515a935.	z03:60020	1000	2000	0
table1,2000,1495642218398.06cedace5adf2a4bd41e7e09cff8f9c3.	z01:60020	2000	3000	0
table1,3000,1495642218398.243f41c7f91b5fc6b71034a6ebe34f70.	z02:60020	3000	4000	0
table1,4000,1495642218398.fb95de71f9dfc7900986bee39ed1fd7a.	z02:60020	4000		0

2）使用 16 进制算法生成预分区

```
hbase(main):003:0> create 'staff2','info','partition2',{NUMREGIONS => 15, SPLITALGO
 => 'HexStringSplit'}
```

完成后如图：

Table Regions

Name	Region Server	Start Key	End Key	Requests
table3,,1495642786733.5368b42666fde7fb0b642daf3e1b3b2f.	z02:60020		11111111	0
table3,11111111,1495642786734.c45f2b8f700c7e6e6b35403da2c84249.	z01:60020	11111111	22222222	0
table3,22222222,1495642786734.3edd9c3cbf2efe64f147871764f380fb.	z03:60020	22222222	33333333	0
table3,33333333,1495642786734.91dab7e69f030638662e6c42de2d20b4.	z01:60020	33333333	44444444	0
table3,44444444,1495642786734.d6ce0f595f199d3d2a285d5fe5ec34bd.	z01:60020	44444444	55555555	0
table3,55555555,1495642786734.574da98eb76c0ef5a948859457bcdff0.	z02:60020	55555555	66666666	0
table3,66666666,1495642786734.b1d92cbe1bf5c06795984f456b2cd692.	z01:60020	66666666	77777777	0
table3,77777777,1495642786734.b3717a3950653ba8905862be766de600.	z02:60020	77777777	88888888	0
table3,88888888,1495642786734.90f42a9bec7a4909b9b1f4eb7783d6cc.	z03:60020	88888888	99999999	0
table3,99999999,1495642786734.7912adf2cba00a4dd28a3c479321b596.	z03:60020	99999999	aaaaaaaa	0
table3,aaaaaaaa,1495642786734.5af37085a70baea33ebededcd28219e1a.	z01:60020	aaaaaaaa	bbbbbbbb	0
table3,bbbbbbbb,1495642786734.09284191e02a938512bfe8d3e588d3ee.	z02:60020	bbbbbbbb	cccccccc	0
table3,cccccccc,1495642786734.c0f8b0d20fe5d0d7861e3c49db6cbe16.	z03:60020	cccccccc	dddddddd	0
table3,dddddddd,1495642786734.93600a2dc08821fd01f2ca00965bbaf8.	z03:60020	dddddddd	eeeeeeee	0
table3,eeeeeeee,1495642786734.4f4e56a23fd51df42b33f3dc5ba880a1.	z02:60020	eeeeeeee		0

3）分区规则创建于文件中

创建 splits.txt 文件内容如下：

vim splits.txt

```
aaaa
bbbb
cccc
dddd
```

然后执行：

然后执行：

```
hbase(main):004:0> create 'staff3','partition2',SPLITS_FILE => '/export/servers/spl
its.txt'
```

完成后如图：

Table Regions

Name	Region Server	Start Key	End Key	Requests
table2,,1495642727718.5ebcf21bfb7fe838288af5046d79d79d.	z03:60020		aaaa	0
table2,aaaa,1495642727718.7aa4c6aa1c84beb78971dc214ca853d4.	z03:60020	aaaa	bbbb	0
table2,bbbb,1495642727718.de32a1e5e21f4f73932eabd7572d9cdf.	z02:60020	bbbb	cccc	0
table2,cccc,1495642727719.ec6ea106d8b05a2575dc1e3f20391d23.	z02:60020	cccc	dddd	0
table2,dddd,1495642727719.8109c3415b405595ecc77020b4c22558.	z01:60020	dddd		0

4）使用 JavaAPI 创建预分区

代码如下：

```
@Test
    public void hbaseSplit() throws IOException {
        //获取连接
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum", "node01:2181,node02:2181,node03:
2181");
        Connection connection = ConnectionFactory.createConnection(configuration);
        Admin admin = connection.getAdmin();
        //自定义算法，产生一系列Hash 散列值存储在二维数组中
        byte[][] splitKeys = {{1,2,3,4,5},{'a','b','c','d','e'}};


        //通过 HTableDescriptor 来实现我们表的参数设置，包括表名，列族等等
        HTableDescriptor hTableDescriptor = new HTableDescriptor(TableName.valueOf(
"stuff4"));
        //添加列族
        hTableDescriptor.addFamily(new HColumnDescriptor("f1"));
        //添加列族
        hTableDescriptor.addFamily(new HColumnDescriptor("f2"));
        admin.createTable(hTableDescriptor,splitKeys);
        admin.close();
    }
```

九、HBase 的 rowKey 设计技巧

HBase 是三维有序存储的，通过 rowkey（行键），column key（column family 和 qualifier）和 TimeStamp（时间戳）这个三个维度可以对 HBase 中的数据进行快速定位。

HBase 中 rowkey 可以唯一标识一行记录，在 HBase 查询的时候，有以下几种方式：

1. 通过 get 方式，指定 rowkey 获取唯一一条记录；
2. 通过 scan 方式，设置 startRow 和 stopRow 参数进行范围匹配；
3. 全表扫描，即直接扫描整张表中所有行记录。

1. rowkey 长度原则

rowkey 是一个二进制码流，可以是任意字符串，最大长度 64kb，实际应用中一般为 10-100bytes，以 byte[]形式保存，一般设计成定长。

建议越短越好，不要超过 16 个字节，原因如下：

- 数据的持久化文件 HFile 中是按照 KeyValue 存储的，如果 rowkey 过长，比如超过 100 字节，1000w 行数据，光 rowkey 就要占用 100*1000w=10 亿个字节，将近 1G 数据，这样会极大影响 HFile 的存储效率；
- MemStore 将缓存部分数据到内存，如果 rowkey 字段过长，内存的有效利用率就会降低，系统不能缓存更多的数据，这样会降低检索效率。

2. rowkey 散列原则

如果 rowkey 按照时间戳的方式递增，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，由程序随机生成，低位放时间字段，这样将提高数据均衡分布在每个 RegionServer，以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，所有的数据都会集中在一个 RegionServer 上，这样在数据检索的时候负载会集中在个别的 RegionServer 上，造成热点问题，会降低查询效率。

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。

4. 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。然而糟糕的 rowkey 设计是热点的源头。

热点发生在大量的 client 直接访问集群的一个或极少数个节点（访问可能是读，写或者其他操作）。大量访问会使热点 region 所在的单个机器超出自身承受能力，引起性能下降甚至 region 不可用，这也会影响同一个 RegionServer 上的其他 region，由于主机无法服务其他 region 的请求。

设计良好的数据访问模式以使集群被充分，均衡的利用。为了避免写热点，设计 rowkey 使得不同行在同一个 region，但是在更多数据情况下，数据应该被写入集群的多个 region，而不是一个。

下面是一些常见的避免热点的方法以及它们的优缺点：

1）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。分配的前缀种类数量应该和你想使用数据分散到不同的 region 的数量一致。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。

2）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。

3）反转

第三种防止热点的方法时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，这样的就避免了以手机号那样比较固定开头导致热点问题。

3）时间戳反转

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][reverse_timestamp] ， [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。

其他一些建议：

- 尽量减少行键和列族的大小在 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，这个时候它们将会占用大量的存储空间。
- 列族尽可能越短越好，最好是一个字符。
- 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。

十、HBase 的协处理器

```
http://hbase.apache.org/book.html#cp
```

1. 起源

Hbase 作为列族数据库最经常被人诟病的特性包括：无法轻易建立“二级索引”，难以执行求和、计数、排序等操作。

比如，在旧版本的(<0.92)Hbase 中，统计数据表的总行数，需要使用 Counter 方法，执行一次 MapReduce Job 才能得到。

虽然 HBase 在数据存储层中集成了 MapReduce，能够有效用于数据表的分布式计算。然而在很多情况下，做一些简单的相加或者聚合计算的时候， 如果直接将计算过程放置在 server 端，能够减少通讯开销，从而获得很好的性能提升。于是， HBase 在 0.92 之后引入了协处理器(coprocessors)，实现一些激动人心的新特性：**能够轻易建立二次索引、复杂过滤器(谓词下推)以及访问控制**等。

2. 协处理器有两种： observer 和 endpoint

1）observer 协处理器

Observer 类似于传统数据库中的触发器，当发生某些事件的时候这类协处理器会被 Server 端调用。

Observer Coprocessor 就是一些散布在 HBase Server 端代码中的 hook 钩子，在固定的事件发生时被调用。

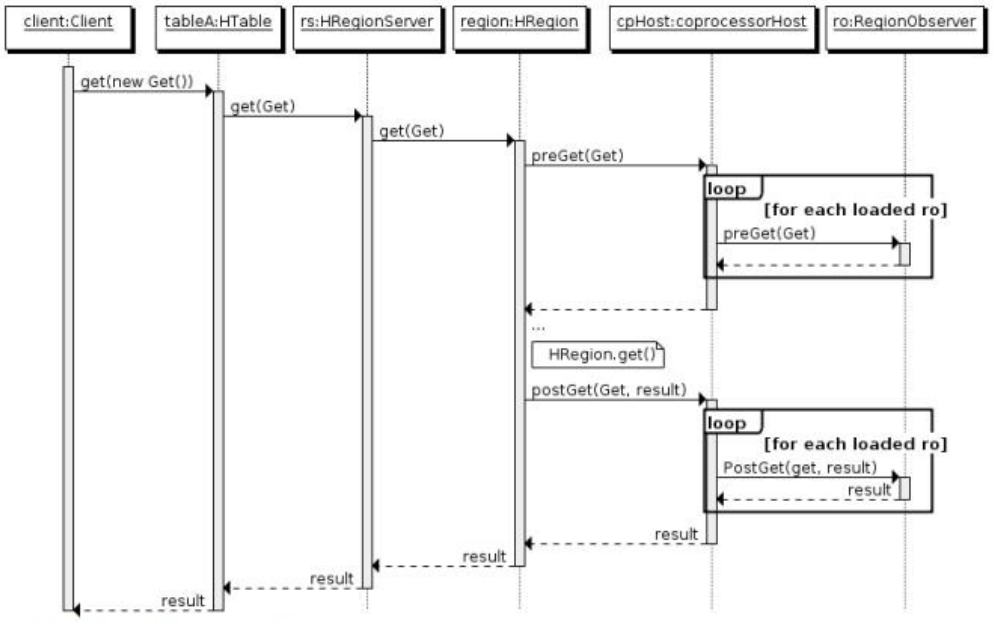
比如： put 操作之前有钩子函数 prePut，该函数在 put 操作执行前会被 Region Server 调用；在 put 操作之后则有 postPut 钩子函数。

以 HBase0.92 版本为例，它提供了三种观察者接口：

- RegionObserver：提供客户端的数据操纵事件钩子： Get、 Put、 Delete、 Scan 等。
- WALObserver：提供 WAL 相关操作钩子。
- MasterObserver：提供 DDL-类型的操作钩子。如创建、删除、修改数据表等。

到 0.96 版本又新增一个 RegionServerObserver

下图是以 RegionObserver 为例子讲解 Observer 这种协处理器的原理：



1、客户端发出 put 请求
2、该请求被分派给合适的 RegionServer 和 region
3、coprocessorHost 拦截该请求，然后在该表上登记的每个 RegionObserver 上调用 prePut()
4、如果没有被 prePut()拦截，该请求继续送到 region，然后进行处理
5、region 产生的结果再次被 CoprocessorHost 拦截，调用 postPut()
6、假如没有 postPut()拦截该响应，最终结果被返回给客户端

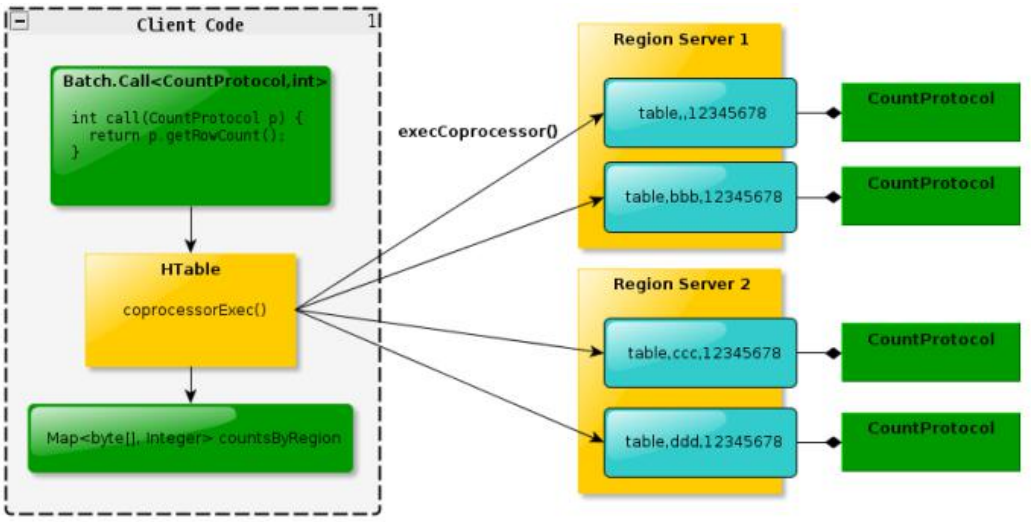
2） endpoint 协处理器

Endpoint 协处理器类似传统数据库中的存储过程，客户端可以调用这些 Endpoint 协处理器执行一段 Server 端代码，并将 Server 端代码的结果返回给客户端进一步处理，最常见的用法就是进行聚集操作。

如果没有协处理器，当用户需要找出一张表中的最大数据，即 max 聚合操作，就必须进行全表扫描，在客户端代码内遍历扫描结果，并执行求最大值的操作。这样的方法无法利用底层集群的并发能力，而将所有计算都集中到 Client 端统一执行，势必效率低下。

利用 Coprocessor，用户可以将求最大值的代码部署到 HBase Server 端，HBase 将利用底层 cluster 的多个节点并发执行求最大值的操作。即在每个 Region 范围内 执行求最大值的代码，将每个 Region 的最大值在 Region Server 端计算出，仅仅将该 max 值返回给客户端。在客户端进一步将多个 Region 的最大值进一步处理而找到其中的最大值。这样整体的执行效率就会提高很多。

下图是 EndPoint 的工作原理：



3. 协处理器加载方式

协处理器的加载方式有两种，我们称之为静态加载方式（ Static Load）和动态加载方式（ Dynamic Load）。

静态加载的协处理器称之为 System Coprocessor

动态加载的协处理器称之为 Table Coprocessor。

1）静态加载

通过修改 hbase-site.xml 这个文件来实现， 启动全局 aggregation，能过操纵所有的表上的数据。只需要添加如下代码：

```
<property>
<name>hbase.coprocessor.user.region.classes</name>
<value>org.apache.hadoop.hbase.coprocessor.AggregateImplementation</value>
</property>
```

2）动态加载

启用表 aggregation，只对特定的表生效。通过 HBase Shell 来实现。

disable 指定表

```
hbase> disable 'mytable'
```

添加 aggregation

```
hbase> alter 'mytable', METHOD => 'table_att','coprocessor'=>
'|org.apache.Hadoop.hbase.coprocessor.AggregateImplementation||'
```

重启指定表

```
hbase> enable 'mytable'
```

协处理器卸载

```
disable 'mytable'
alter 'mytable', METHOD => 'table_att_unset',NAME=>'coprocessor$1'
enable 'test'
```

十一、HBase 当中的二级索引的简要介绍

由于 HBase 的查询比较弱，如果需要实现类似于 `select name,salary,count(1),max(salary) from user group by name,salary order by salary` 等这样的复杂性的统计需求，基本上不可能，或者说比较困难，所以我们在使用 HBase 的时候，一般都会借助二级索引的方案来进行实现。

HBase 的一级索引就是 rowkey，我们只能通过 rowkey 进行检索。如果我们相对 hbase 里面列族的列列进行一些组合查询，就需要采用 HBase 的二级索引方案来进行多条件的查询。

1. MapReduce 方案 2. ITHBASE（Indexed-Transanctional HBase）方案 3. IHBASE（Index HBase）方案 4. Hbase Coprocessor(协处理器)方案 5. Solr+hbase 方案 6. CCIndex（complementalclustering index）方案

常见的二级索引我们一般可以借助各种其他的方式来实现，例如 Phoenix 或者 solr 或者 ES 等。

十二、HBase 调优

1. 通用优化

1. NameNode 的元数据备份使用 SSD。
2. 定时备份 NameNode 上的元数据，每小时或者每天备份，如果数据极其重要，可以 5~10 分钟备份一次。备份可以通过定时任务复制元数据目录即可。
3. 为 NameNode 指定多个元数据目录，使用 dfs.name.dir 或者 dfs.namenode.name.dir 指定。一个指定本地磁盘，一个指定网络磁盘。这样可以提供元数据的冗余和健壮性，以免发生故障。

4. 设置 dfs.namenode.name.dir.restore 为 true，允许尝试恢复之前失败的 dfs.namenode.name.dir 目录，在创建 checkpoint 时做此尝试，如果设置了多个磁盘，建议允许。
5. NameNode 节点必须配置为 RAID1（镜像盘）结构。
6. 保持 NameNode 日志目录有足够的空间，这些日志有助于帮助你发现问题。
7. 因为 Hadoop 是 IO 密集型框架，所以尽量提升存储的速度和吞吐量（类似位宽）。

2. Linux 优化

1. 开启文件系统的预读缓存可以提高读取速度

```
$ sudo blockdev --setra 32768 /dev/sda
```

提示：ra 是 readahead 的缩写

2. 关闭进程睡眠池

```
$ sudo sysctl -w vm.swappiness=0
```

3. 调整 ulimit 上限，默认值为比较小的数字

```
$ ulimit -n 查看允许最大进程数
$ ulimit -u 查看允许打开最大文件数
```

修改：

```
$ sudo vi /etc/security/limits.conf 修改打开文件数限制
```

末尾添加：

```
*               soft    nofile          1024000
*               hard    nofile          1024000
Hive            -       nofile          1024000
hive            -       nproc           1024000
```

```
$ sudo vi /etc/security/limits.d/20-nproc.conf 修改用户打开进程数限制
```

修改为：

```
#*          soft    nproc     4096
#root       soft    nproc     unlimited
*           soft    nproc     40960
root        soft    nproc     unlimited
```

4. 开启集群的时间同步 NTP。

5. 更新系统补丁（提示：更新补丁前，请先测试新版本补丁对集群节点的兼容性）

3. HDFS 优化（hdfs-site.xml）

1. 保证 RPC 调用会有较多的线程数

属性：dfs.namenode.handler.count
解释：该属性是 NameNode 服务默认线程数，的默认值是 10，根据机器的可用内存可以调整为 50~100
属性：dfs.datanode.handler.count
解释：该属性默认值为 10，是 DataNode 的处理线程数，如果 HDFS 客户端程序读写请求比较多，可以调高到 15~20，设置的值越大，内存消耗越多，不要调整的过高，一般业务中，5~10 即可。

2. 副本数的调整

属性：dfs.replication
解释：如果数据量巨大，且不是非常之重要，可以调整为 2~3，如果数据非常之重要，可以调整为 3~5。

3. 文件块大小的调整

属性：dfs.blocksize
解释：块大小定义，该属性应该根据存储的大量的单个文件大小来设置，如果大量的单个文件都小于 100M，建议设置成 64M 块大小，对于大于 100M 或者达到 GB 的这种情况，建议设置成 256M，一般设置范围波动在 64M~256M 之间。

4. MapReduce 优化（mapred-site.xml）

1. Job 任务服务线程数调整

mapreduce.jobtracker.handler.count
该属性是 Job 任务线程数，默认值是 10，根据机器的可用内存可以调整为 50~100

2. Http 服务器工作线程数

属性：mapreduce.tasktracker.http.threads
解释：定义 HTTP 服务器工作线程数，默认值为 40，对于大集群可以调整到 80~100

3. 文件排序合并优化

属性：mapreduce.task.io.sort.factor
解释：文件排序时同时合并的数据流的数量，这也定义了同时打开文件的个数，默认值为 10，如果调高该参数，可以明显减少磁盘 IO，即减少文件读取的次数。

4. 设置任务并发

属性：mapreduce.map.speculative
解释：该属性可以设置任务是否可以并发执行，如果任务多而小，该属性设置为 true 可以明显加快任务执行效率，但是对于延迟非常高的任务，建议改为 false，这就类似于迅雷下载。

5. MR 输出数据的压缩

属性：mapreduce.map.output.compress、
mapreduce.output.fileoutputformat.compress
解释：对于大集群而言，建议设置 Map-Reduce 的输出为压缩的数据，而对于小集群，则不需要。

6. 优化 Mapper 和 Reducer 的个数

属性：
mapreduce.tasktracker.map.tasks.maximum
mapreduce.tasktracker.reduce.tasks.maximum
解释：以上两个属性分别为一个单独的 Job 任务可以同时运行的 Map 和 Reduce 的数量。
设置上面两个参数时，需要考虑 CPU 核数、磁盘和内存容量。假设一个 8 核的 CPU，业务内容非常消耗 CPU，那么可以设置 map 数量为 4，如果该业务不是特别消耗 CPU 类型的，那么可以设置 map 数量为 40，reduce 数量为 20。这些参数的值修改完成之后，一定要观察是否有较长等待的任务，如果有的话，可以减少数量以加快任务执行，如果设置一个很大的值，会引起大量的上下文切换，以及内

存与磁盘之间的数据交换，这里没有标准的配置数值，需要根据业务和硬件配置以及经验来做出选择。

在同一时刻，不要同时运行太多的 MapReduce，这样会消耗过多的内存，任务会执行的非常缓慢，我们需要根据 CPU 核数，内存容量设置一个 MR 任务并发的最大值，使固定数据量的任务完全加载到内存中，避免频繁的内存和磁盘数据交换，从而降低磁盘 IO，提高性能。

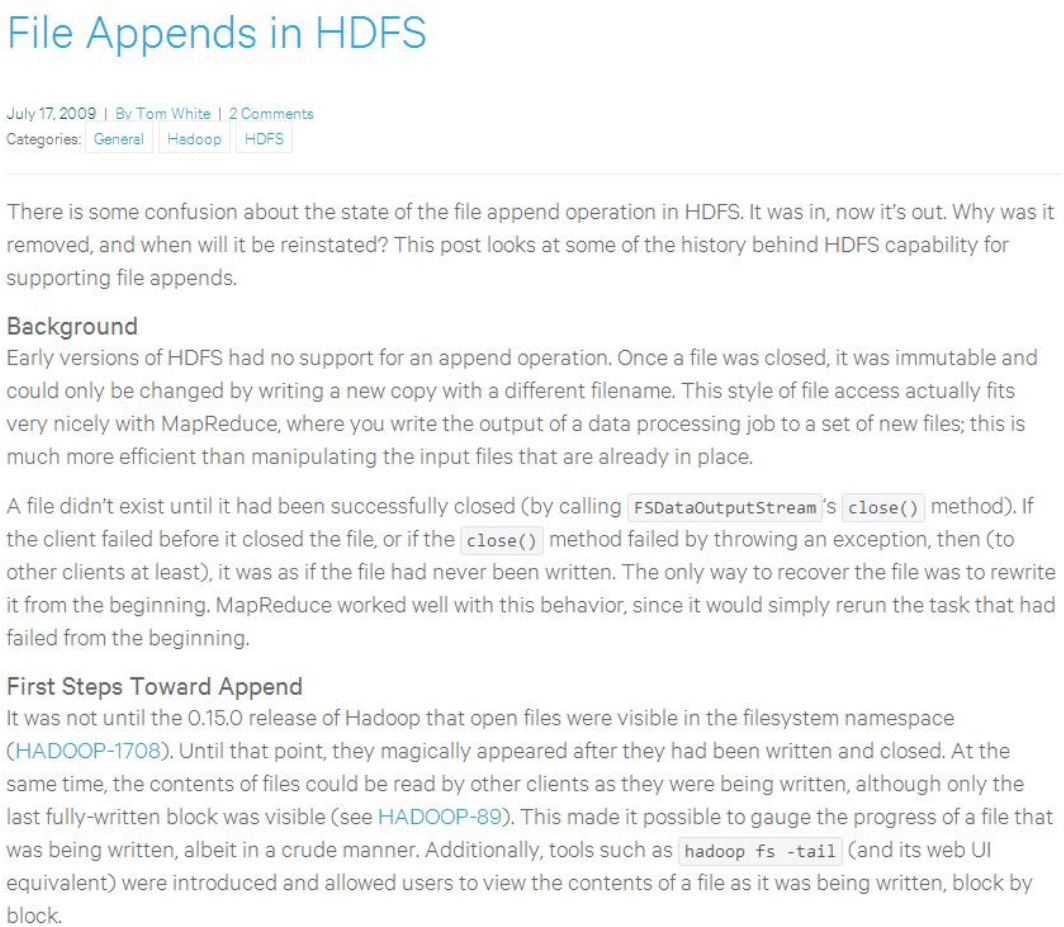
大概估算公式：

```
map = 2 + 2/3cpu_core
reduce = 2 + 1/3cpu_core
```

5. HBase 优化

1. 在 HDFS 的文件中追加内容

HDFS 不是不允许追加内容么？没错，请看背景故事：



File Appends in HDFS

July 17, 2009 | By Tom White | 2 Comments
Categories: General | Hadoop | HDFS

There is some confusion about the state of the file append operation in HDFS. It was in, now it's out. Why was it removed, and when will it be reinstated? This post looks at some of the history behind HDFS capability for supporting file appends.

Background

Early versions of HDFS had no support for an append operation. Once a file was closed, it was immutable and could only be changed by writing a new copy with a different filename. This style of file access actually fits very nicely with MapReduce, where you write the output of a data processing job to a set of new files; this is much more efficient than manipulating the input files that are already in place.

A file didn't exist until it had been successfully closed (by calling `FSDataOutputStream`'s `close()` method). If the client failed before it closed the file, or if the `close()` method failed by throwing an exception, then (to other clients at least), it was as if the file had never been written. The only way to recover the file was to rewrite it from the beginning. MapReduce worked well with this behavior, since it would simply rerun the task that had failed from the beginning.

First Steps Toward Append

It was not until the 0.15.0 release of Hadoop that open files were visible in the filesystem namespace (HADOOP-1708). Until that point, they magically appeared after they had been written and closed. At the same time, the contents of files could be read by other clients as they were being written, although only the last fully-written block was visible (see HADOOP-89). This made it possible to gauge the progress of a file that was being written, albeit in a crude manner. Additionally, tools such as `hadoop fs -tail` (and its web UI equivalent) were introduced and allowed users to view the contents of a file as it was being written, block by block.

属性：`dfs.support.append`

文件：hdfs-site.xml、hbase-site.xml

解释：开启 HDFS 追加同步，可以优秀的配合 HBase 的数据同步和持久化。默认值为 true。

2. 优化 DataNode 允许的最大文件打开数

属性：dfs.datanode.max.transfer.threads
文件：hdfs-site.xml
解释：HBase 一般都会同一时间操作大量的文件，根据集群的数量和规模以及数据动作，设置为 4096 或者更高。默认值：4096

3. **优化延迟高的数据操作的等待时间

属性：dfs.image.transfer.timeout
文件：hdfs-site.xml
解释：如果对于某一次数据操作来讲，延迟非常高，socket 需要等待更长的时间，建议把该值设置为更大的值（默认 60000 毫秒），以确保 socket 不会被 timeout 掉。

4. 优化数据的写入效率

属性：
mapreduce.map.output.compress
mapreduce.map.output.compress.codec
文件：mapred-site.xml
解释：开启这两个数据可以大大提高文件的写入效率，减少写入时间。第一个属性值修改为 true，第二个属性值修改为：
org.apache.hadoop.io.compress.GzipCodec

5. 优化 DataNode 存储

属性：dfs.datanode.failed.volumes.tolerated
文件：hdfs-site.xml
解释:默认为0,意思是当DataNode中有一个磁盘出现故障,则会认为该DataNode shutdown 了。如果修改为 1，则一个磁盘出现故障时，数据会被复制到其他正常的 DataNode 上，当前的 DataNode 继续工作。

6. 设置 RPC 监听数量

属性：hbase.regionserver.handler.count
文件：hbase-site.xml
解释：默认值为 30，用于指定 RPC 监听的数量，可以根据客户端的请求数进行调整，读写请求较多时，增加此值。

7. 优化 HStore 文件大小

属性：hbase.hregion.max.filesize
文件：hbase-site.xml
解释：默认值 10737418240（10GB），如果需要运行 HBase 的 MR 任务，可以减小此值，因为一个 region 对应一个 map 任务，如果单个 region 过大，会导致 map 任务执行时间过长。该值的意思就是，如果 HFile 的大小达到这个数值，则这个 region 会被切分为两个 Hfile。

8. 优化 hbase 客户端缓存

属性：hbase.client.write.buffer
文件：hbase-site.xml
解释：用于指定 HBase 客户端缓存，增大该值可以减少 RPC 调用次数，但是会消耗更多内存，反之则反之。一般我们需要设定一定的缓存大小，以达到减少 RPC 次数的目的。

9. 指定 scan.next 扫描 HBase 所获取的行数

属性：hbase.client.scanner.caching
文件：hbase-site.xml
解释：用于指定 scan.next 方法获取的默认行数，值越大，消耗内存越大。

6. 内存优化

HBase 操作过程中需要大量的内存开销，毕竟 Table 是可以缓存在内存中的，一般会分配整个可用内存的 70%给 HBase 的 Java 堆。但是不建议分配非常大的堆内存，因为 GC 过程持续太久会导致 RegionServer 处于长期不可用状态，一般 16~48G 内存就可以了，如果因为框架占用内存过高导致系统内存不足，框架一样会被系统服务拖死。

7. JVM 优化

涉及文件：hbase-env.sh

1. 并行 GC

参数：•-XX:+UseParallelGC•
解释：开启并行 GC

2. 同时处理垃圾回收的线程数

参数：-XX:ParallelGCThreads=cpu_core - 1
解释：该属性设置了同时处理垃圾回收的线程数。

3. 禁用手动 GC

参数：-XX:DisableExplicitGC
解释：防止开发人员手动调用 GC

8. Zookeeper 优化

1. 优化 Zookeeper 会话超时时间

参数：zookeeper.session.timeout
文件：hbase-site.xml
解释：In hbase-site.xml, set zookeeper.session.timeout to 30 seconds or less to bound failure detection (20-30 seconds is a good start)。
该值会直接关系到 master 发现服务器宕机的最大周期，默认值为 30 秒，如果该值过小，会在 HBase 在写入大量数据发生而 GC 时，导致 RegionServer 短暂的不可用，从而没有向 ZK 发送心跳包，最终导致认为从节点 shutdown。一般 20 台左右的集群需要配置 5 台 zookeeper。

十三、HBase 大厂面试题解析

1. Hbase 是怎么写数据的？

Client 写入 -> 存入 MemStore，一直到 MemStore 满 -> Flush 成一个 StoreFile，直至增长到一定阈值 -> 触发 Compact 合并操作 -> 多个 StoreFile 合并成一个 StoreFile，同时进行版本合并和数据删除 -> 当 StoreFiles Compact 后，逐步形成越来越大的 StoreFile -> 单个 StoreFile 大小超过一定阈值（默认 10G），触发 Split 操作，把当前 Region Split 成 2 个 Region，Region 会下线，新 Split 出的 2 个孩子 Region 会被 HMaster 分配到相应的 HRegionServer 上，使得原先 1 个 Region 的压力得以分流到 2 个 Region 上

由此过程可知，HBase 只是增加数据，没有更新和删除操作，用户的更新和删除都是逻辑层面的，在物理层面，更新只是追加操作，删除只是标记操作。

用户写操作只需要进入到内存即可立即返回，从而保证 I/O 高性能。

2. HDFS 和 HBase 各自使用场景

首先一点需要明白：Hbase 是基于 HDFS 来存储的。

HDFS：

1. 一次性写入，多次读取。
2. 保证数据的一致性。
3. 主要是可以部署在许多廉价机器中，通过多副本提高可靠性，提供了容错和恢复机制。

HBase：

1. 瞬间写入量很大，数据库不好支撑或需要很高成本支撑的场景。
2. 数据需要长久保存，且量会持久增长到比较大的场景。
3. HBase 不适用与有 join，多级索引，表关系复杂的数据模型。
4. 大数据量（100s TB 级数据）且有快速随机访问的需求。如：淘宝的交易历史记录。数据量巨大无容置疑，面向普通用户的请求必然要即时响应。
5. 业务场景简单，不需要关系数据库中很多特性（例如交叉列、交叉表，事务，连接等等）。

3. Hbase 的存储结构

Hbase 中的每张表都通过行键(rowkey)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，由 HRegionServer

管理，管理哪些 HRegion 由 Hmaster 分配。 HRegion 存取一个子表时，会创建一个 HRegion 对象，然后对表的每个列族（Column Family）创建一个 store 实例， 每个 store 都会有 0 个或多个 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，一个 HRegion 还拥有一个 MemStore 实例。

4. 热点现象（数据倾斜）怎么产生的，以及解决方法有哪些

热点现象：

某个小的时段内，对 HBase 的读写请求集中到极少数的 Region 上，导致这些 region 所在的 RegionServer 处理请求量骤增，负载量明显偏大，而其他的 RgionServer 明显空闲。

热点现象出现的原因：

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。然而糟糕的 rowkey 设计是热点的源头。

热点发生在大量的 client 直接访问集群的一个或极少数个节点（访问可能是读，写或者其他操作）。大量访问会使热点 region 所在的单个机器超出自身承受能力，引起性能下降甚至 region 不可用，这也会影响同一个 RegionServer 上的其他 region，由于主机无法服务其他 region 的请求。

热点现象解决办法：

为了避免写热点，设计 rowkey 使得不同行在同一个 region，但是在更多数据情况下，数据应该被写入集群的多个 region，而不是一个。常见的方法有以下这些：

1. **加盐**：在 rowkey 的前面增加随机数，使得它和之前的 rowkey 的开头不同。分配的前缀种类数量应该和你想使用数据分散到不同的 region 的数量一致。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。
2. **哈希**：哈希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
3. **反转**：第三种防止热点的方法时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，这样的就避免了以手机号那样比较固定开头导致热点问题

4. **时间戳反转**：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][reverse_timestamp],[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。
    - 比如需要保存一个用户的操作记录，按照操作时间倒序排序，在设计 rowkey 的时候，可以这样设计[userId 反转] [Long.Max_Value - timestamp]，在查询用户的所有操作记录数据的时候，直接指定反转后的 userId，startRow 是[userId 反转][000000000000],stopRow 是[userId 反转][Long.Max_Value - timestamp]
    - 如果需要查询某段时间的操作记录，startRow 是[user 反转][Long.Max_Value - 起始时间]，stopRow 是[userId 反转][Long.Max_Value - 结束时间]
5. **HBase 建表预分区**：创建 HBase 表时，就预先根据可能的 RowKey 划分出多个 region 而不是默认的一个，从而可以将后续的读写操作负载均衡到不同的 region 上，避免热点现象。

5. HBase 的 rowkey 设计原则

长度原则：100 字节以内，8 的倍数最好，可能的情况下越短越好。因为 HFile 是按照 keyvalue 存储的，过长的 rowkey 会影响存储效率；其次，过长的 rowkey 在 memstore 中较大，影响缓冲效果，降低检索效率。最后，操作系统大多为 64 位，8 的倍数，充分利用操作系统的最佳性能。

散列原则：高位散列，低位时间字段。避免热点问题。

唯一原则：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。

6. HBase 的列簇设计

原则：在合理范围内能尽量少的减少列簇就尽量减少列簇，因为列簇是共享 region 的，每个列簇数据相差太大导致查询效率低下。

最优：将所有相关性很强的 key-value 都放在同一个列簇下，这样既能做到查询效率最高，也能保持尽可能少的访问不同的磁盘文件。以用户信息为例，可以将必须的基本信息存放在一个列族，而一些附加的额外信息可以放在另一列族。

7. HBase 中 compact 用途是什么，什么时候触发，分为哪两种，有什么区别

在 hbase 中每当有 memstore 数据 flush 到磁盘之后，就形成一个 storefile，当 storeFile 的数量达到一定程度后，就需要将 storefile 文件来进行 compaction 操作。

Compact 的作用：

1. 合并文件
2. 清除过期，多余版本的数据
3. 提高读写数据的效率 4 HBase 中实现了两种 compaction 的方式：minor and major. 这两种 compaction 方式的 区别是：
4. Minor 操作只用来做部分文件的合并操作以及包括 minVersion=0 并且设置 ttl 的过 期版本清理，不做任何删除数据、多版本数据的清理工作。
5. Major 操作是对 Region 下的 HStore 下的所有 StoreFile 执行合并操作，最终的结果 是整理合并出一个文件。