# 第三章 HBase高级

**学习目标**

* 能够掌握HBase数据读写流程
* 能够掌握HBase BulkLoad数据批量装载
* 了解HBase的LSM以及相关数据结构（二叉树/B-树/B+树）
* 了解布隆过滤器工作原理
* 了解HBase的常用调优方式

第三章 HBase高级 1

1. 重要工作机制 2

1.1 读数据流程 2

1.2 数据存储流程 3

1.3 Region管理 5

1.4 Master工作机制 6

1.5 StoreFile合并 7

1.6 In-memory合并 10

2. HBase批量装载——Bulk load 11

2.1 简介 11

2.2 Bulk load MapReduce程序开发 12

2.3 银行转账记录海量冷数据存储案例 12

3. HBase的协处理器（Coprocessor） 19

3.1 起源 20

3.2 协处理器有两种： observer 和 endpoint 20

3.3 协处理器加载方式 22

4. HBase事务 23

5. HBase数据结构 24

5.1 常见树结构 24

5.2 LSM树数据结构 27

5.3 布隆过滤器 29

5.4 StoreFiles（HFile）结构 31

6. HBase调优 34

6.1 通用优化 34

6.2 Linux优化 36

6.3 HDFS优化（hdfs-site.xml） 37

6.4 HBase优化 38

6.5 指定scan.next扫描HBase所获取的行数 41

7. 内存优化 41

7.1 JVM优化 41

7.2 Zookeeper优化 42

## 重要工作机制

### 读数据流程

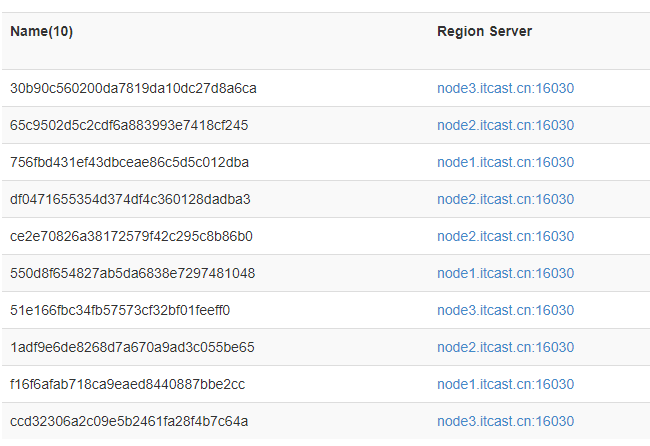
* 从zookeeper找到meta表的region的位置，然后读取meta表中的数据。而meta中又存储了用户表的region信息

|  |
| --- |
| ZK：/hbase/meta-region-server，该节点保存了meta表的region server数据 |

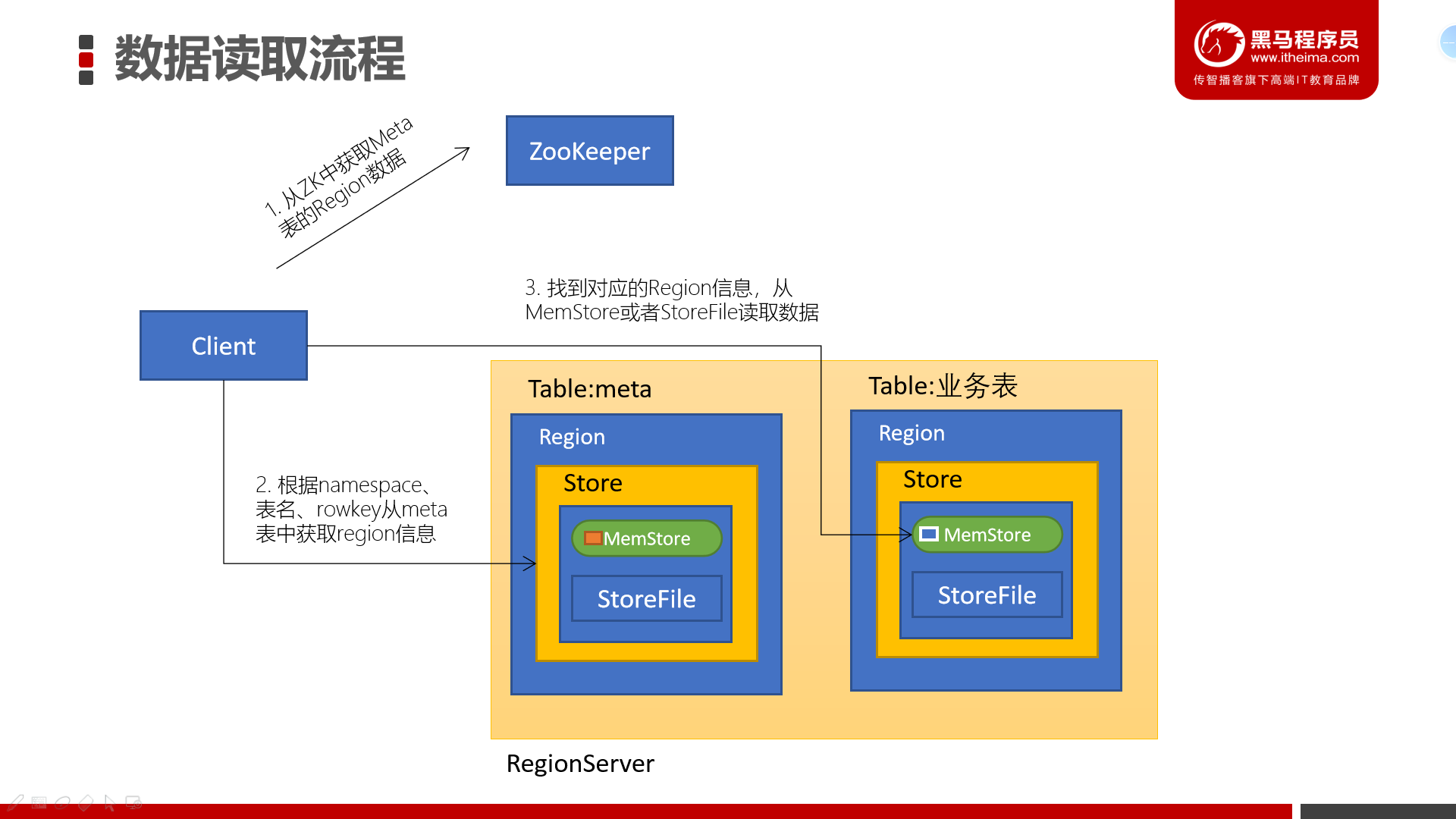
* 根据namespace、表名和rowkey根据meta表中的数据找到对应的region信息

|  |
| --- |
| hbase(main):014:0> **scan "hbase:meta", { FILTER => "PrefixFilter('ORDER\_DTL')"}**  ORDER\_DTL,,1599542264340.30b90c560200da7819da10dc27d8a6ca. column=info:state, timestamp=1599542721810, value=OPEN  ORDER\_DTL,,1599542264340.30b90c560200da7819da10dc27d8a6ca. column=info:regioninfo, timestamp=1599542721810, value={ENCODED => 30b90c560200da7819da10dc27d8a6ca, NAME => 'ORDER\_DTL,,1599542264340.30b90c560200da7819da10dc27d8a6ca.', STARTKEY => '', ENDKEY => '\x01'}  ORDER\_DTL,,1599542264340.30b90c560200da7819da10dc27d8a6ca. column=info:server, timestamp=1599542721810, value=node3.itcast.cn:16020 |

* 找到对应的regionserver，查找对应的region



* 从MemStore找数据，如果没有，再到StoreFile上读



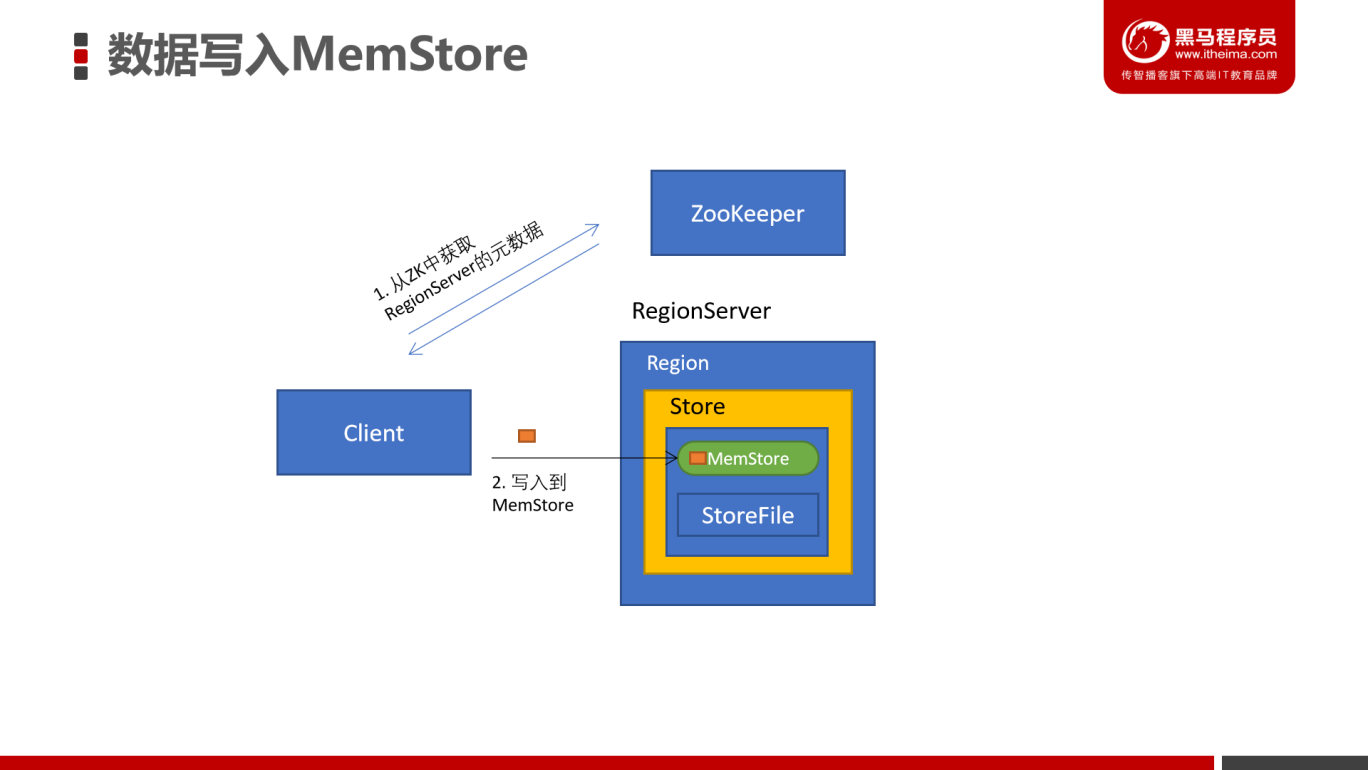
### 数据存储流程

**写内存，两阶段合并**

HBase的数据存储过程是分为几个阶段的。写入的过程与HBase的LSM结构对应。

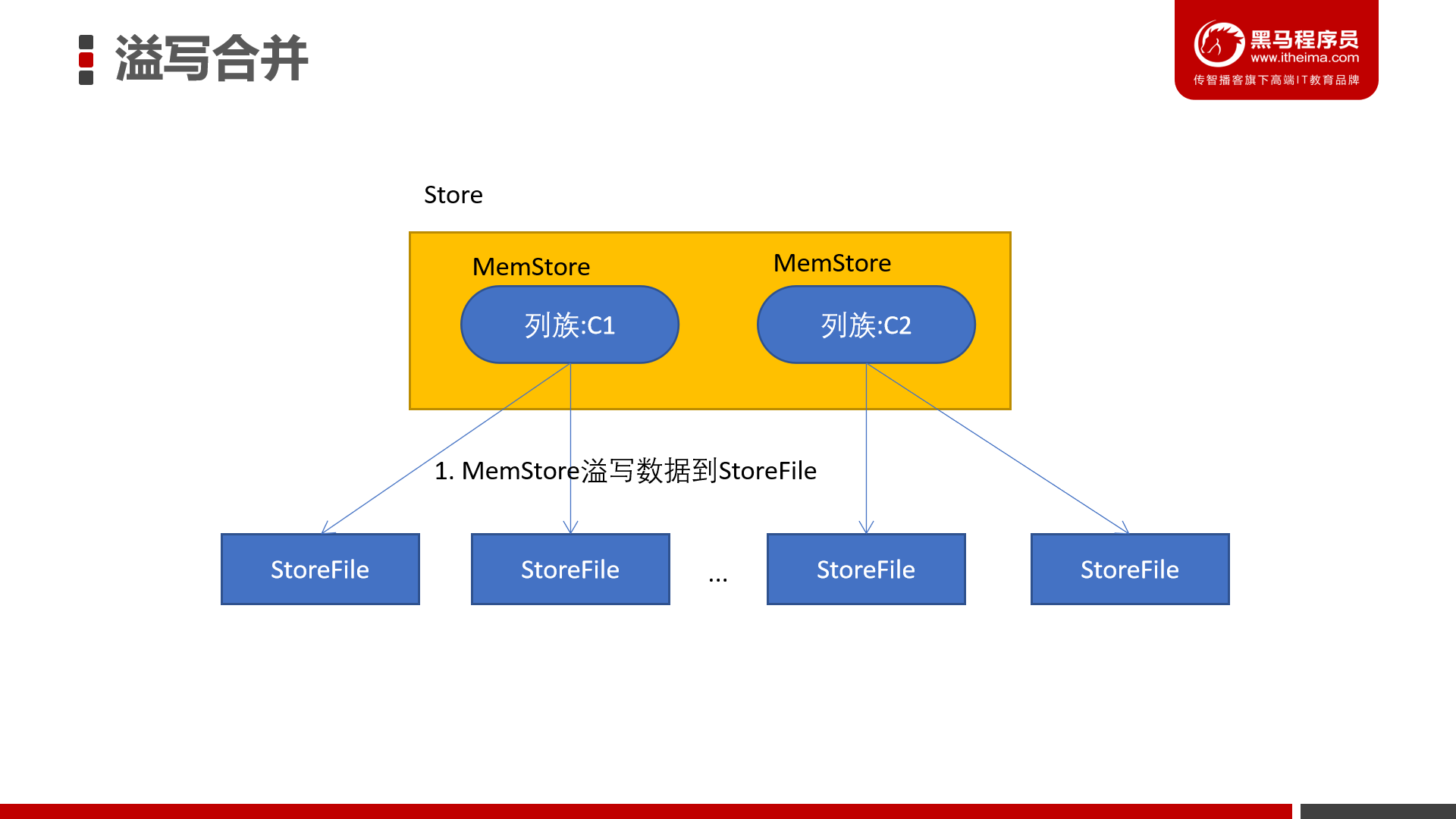
1. 为了提高HBase的写入速度，数据都是先写入到MemStore（内存）结构中
2. MemStore写到一定程度（默认128M），由后台程序将MemStore的内容flush刷写到HDFS中的StoreFile
3. 数据量较大时，会产生很多的StoreFile。这样对高效读取不利，HBase会将这些小的StoreFile合并，一般3-10个文件合并成一个更大的StoreFile

#### 写入MemStore



* Client访问zookeeper，从ZK中找到meta表的region位置
* 读取meta表中的数据，根据namespace、表名、rowkey获取对应的Region信息
* 通过刚刚获取的地址访问对应的RegionServer，拿到对应的表存储的RegionServer
* 去表所在的RegionServer进行数据的添加
* 查找对应的region，在region中寻找列族，先向MemStore中写入数据

#### MemStore溢写合并



##### 说明

* 当MemStore写入的值变多，触发溢写操作（flush），进行文件的溢写，成为一个StoreFile
* 当溢写的文件过多时，会触发文件的合并（Compact）操作，合并有两种方式（major，minor）

##### 触发条件

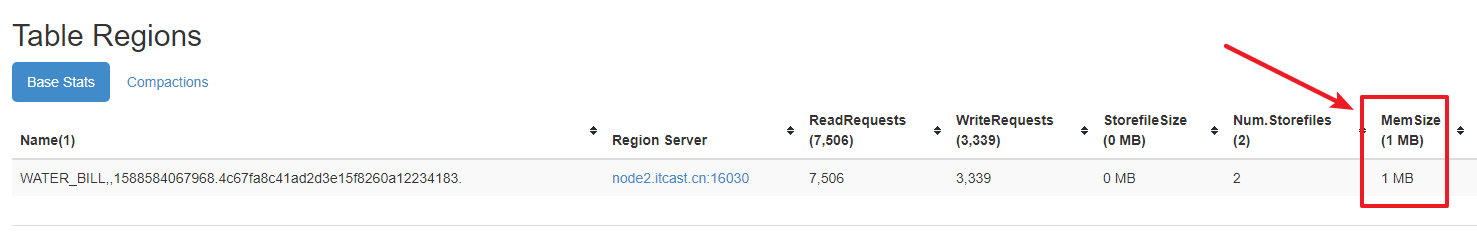
* 一旦MemStore达到128M时，则触发Flush溢出（Region级别）

|  |
| --- |
| <property>  <name>hbase.hregion.memstore.flush.size</name>  <value>134217728</value>  <source>hbase-default.xml</source>  </property> |

* MemStore的存活时间超过1小时（默认），触发Flush溢写（RegionServer级别）

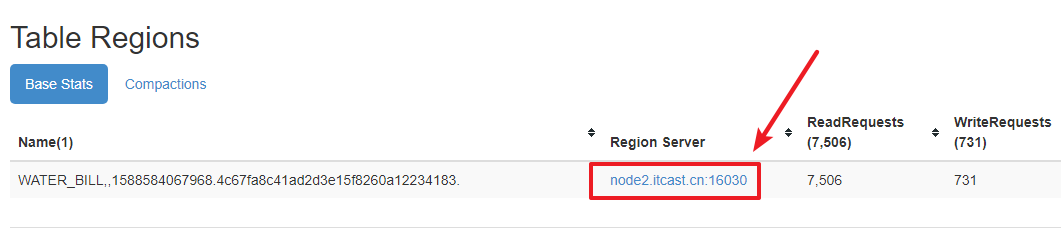
|  |
| --- |
| <property>  <name>hbase.regionserver.optionalcacheflushinterval</name>  <value>3600000</value>  <source>hbase-default.xml</source>  </property> |

#### 模拟数据查看MemStore使用情况

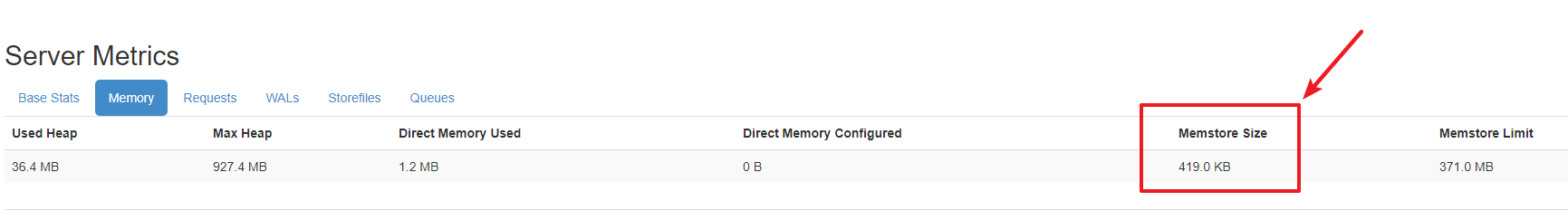


注意：此处小数是无法显示的，只显示整数位的MB。

* 在资料/测试程序中有一个GenWaterBill代码文件，将它导入到之前创建的Java操作HBase中，然后运行。
* 打开HBase WebUI > Table Details > 「WATER\_BILL」
* 打开Region所在的Region Server



点击Memory查看内存占用情况



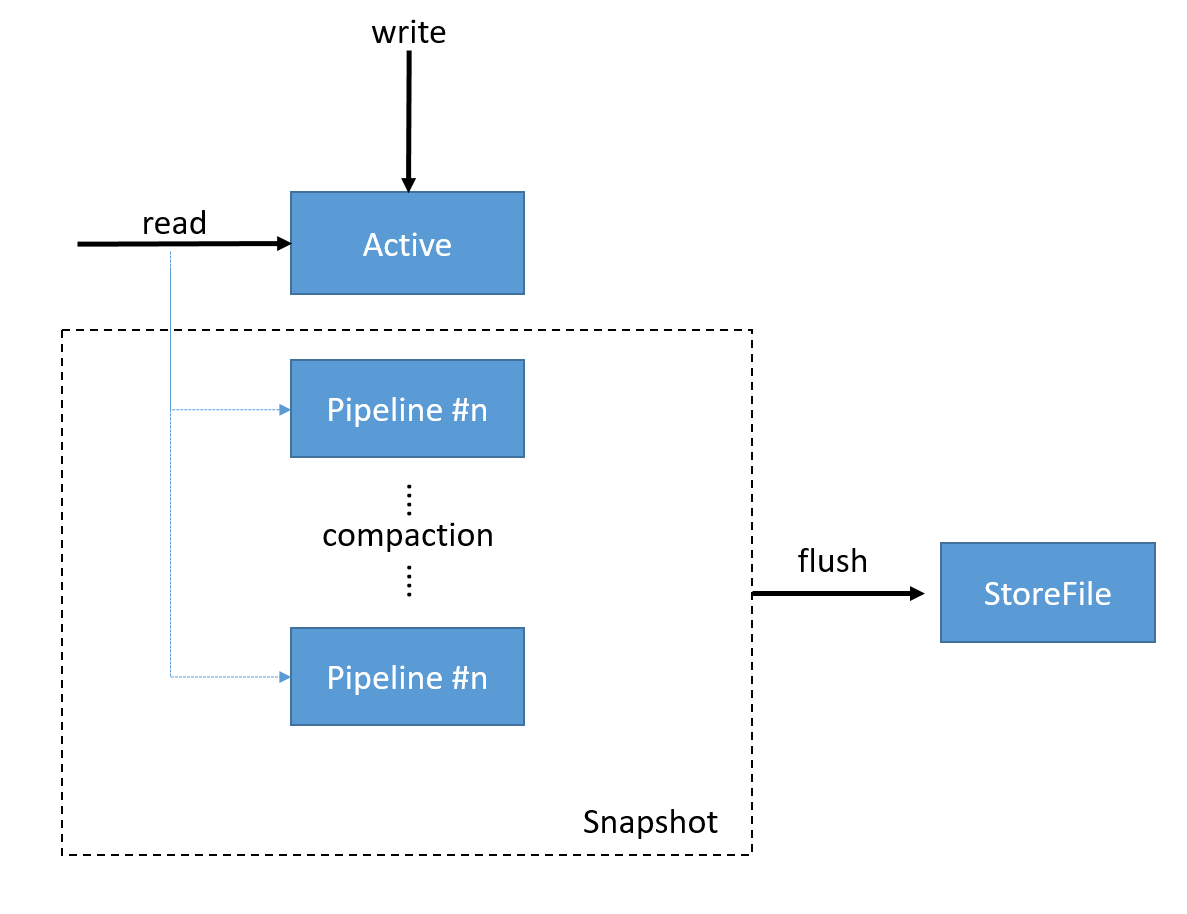
#### In-memory合并

##### In-memory compaction介绍

In-memory合并是**HBase 2.0**之后添加的。它与默认的MemStore的区别：实现了在内存中进行compaction（合并）。

在CompactingMemStore中，数据是以段（Segment）为单位存储数据的。MemStore包含了多个segment。

* 当数据写入时，首先写入到的是Active segment中（也就是当前可以写入的segment段）
* 在2.0之前，如果MemStore中的数据量达到指定的阈值时，就会将数据flush到磁盘中的一个StoreFile
* 2.0的In-memory compaction，active segment满了后，将数据移动到pipeline中。这个过程跟以前不一样，以前是flush到磁盘，而这次是将Active segment的数据，移到称为pipeline的内存当中。一个pipeline中可以有多个segment。而In-memory compaction会将pipeline的多个segment合并为更大的、更紧凑的segment，这就是compaction
* HBase会尽量延长CompactingMemStore的生命周期，以达到减少总的IO开销。当需要把CompactingMemStore flush到磁盘时，pipeline中所有的segment会被移动到一个snapshot中，然后进行合并后写入到HFile



##### compaction策略

但Active segment flush到pipeline中后，后台会触发一个任务来合并pipeline中的数据。合并任务会扫描pipeline中所有的segment，将segment的索引合并为一个索引。有三种合并策略：

* basic
  + Basic compaction策略不清理多余的数据版本，无需对cell的内存进行考核
  + basic适用于所有大量写模式
* eager
  + eager compaction会过滤重复的数据，清理多余的版本，这会带来额外的开销
  + eager模式主要针对数据大量过期淘汰的场景，例如：购物车、消息队列等
* adaptive
  + adaptive compaction根据数据的重复情况来决定是否使用eager策略
  + 该策略会找出cell个数最多的一个，然后计算一个比例，如果比例超出阈值，则使用eager策略，否则使用basic策略

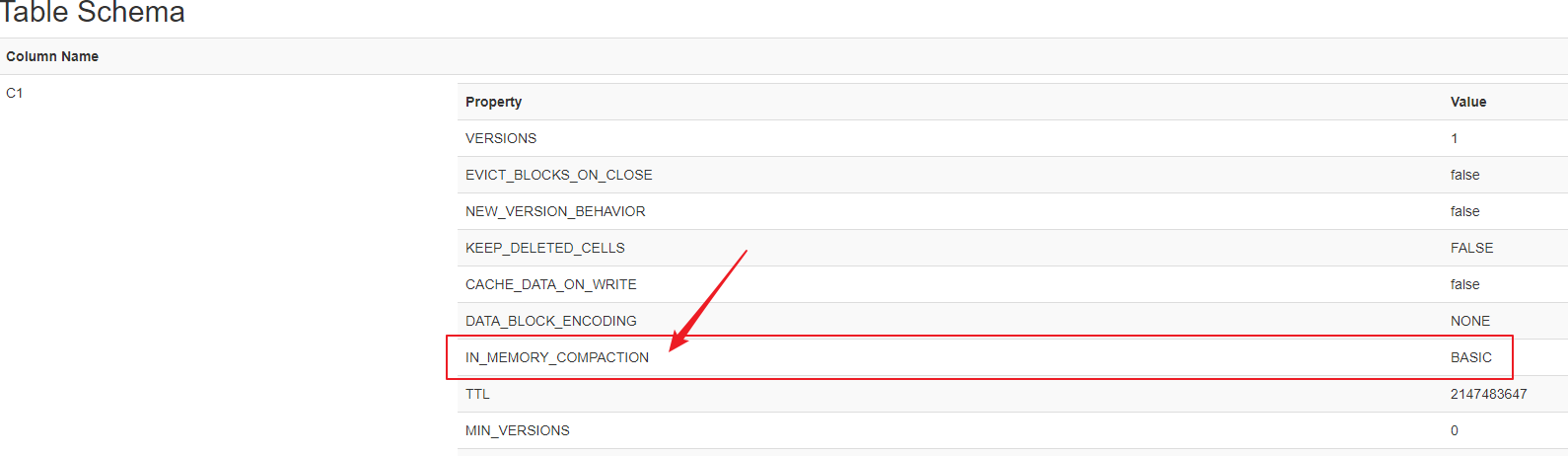
##### 配置

1. 可以通过hbase-site.xml来配置默认In Memory Compaction方式

|  |
| --- |
| <property>  <name>hbase.hregion.compacting.memstore.type</name>  <value><none|basic|eager|adaptive></value>  </property> |

1. 在创建表的时候指定

|  |
| --- |
| create "test\_memory\_compaction", {NAME => 'C1', IN\_MEMORY\_COMPACTION => "BASIC"} |



#### StoreFile合并

* 当MemStore超过阀值的时候，就要flush到HDFS上生成一个StoreFile。因此随着不断写入，HFile的数量将会越来越多，根据前面所述，StoreFile数量过多会降低读性能
* 为了避免对读性能的影响，需要对这些StoreFile进行compact操作，把多个HFile合并成一个HFile
* compact操作需要对HBase的数据进行多次的重新读写，因此**这个过程会产生大量的IO**。可以看到compact操作的本质就是以IO操作换取后续的读性能的提高

##### minor compaction

###### 说明

* Minor Compaction操作只用来做部分文件的合并操作，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
* 小范围合并，默认是3-10个文件进行合并，不会删除其他版本的数据
* Minor Compaction则只会选择数个StoreFile文件compact为一个StoreFile
* Minor Compaction的过程一般较快，而且IO相对较低

###### 触发条件

* 在打开Region或者MemStore时会自动检测是否需要进行Compact（包括Minor、Major）
* minFilesToCompact由hbase.hstore.compaction.min控制，默认值为3
* 即Store下面的StoreFile数量减去正在compaction的数量 >=3时，需要做compaction

<http://node1.itcast.cn:16010/conf>

|  |
| --- |
| <property>  <name>hbase.hstore.compaction.min</name>  <value>3</value>  <final>false</final>  <source>hbase-default.xml</source>  </property> |

##### major compaction

###### 说明

* Major Compaction操作是对Region下的Store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
* ，一般手动触发，会删除其他版本的数据（不同时间戳的）

###### 触发条件

* 如果无需进行Minor compaction，HBase会继续判断是否需要执行Major Compaction
* 如果所有的StoreFile中，最老（时间戳最小）的那个StoreFile的时间间隔大于Major Compaction的时间间隔（hbase.hregion.majorcompaction——默认7天）

|  |
| --- |
| <property>  <name>hbase.hregion.majorcompaction</name>  <value>604800000</value>  <source>hbase-default.xml</source>  </property> |

604800000毫秒 = 604800秒 = 168小时 = 7天

### Region管理

#### region分配

* 任何时刻，一个region只能分配给一个region server
* Master记录了当前有哪些可用的region server，以及当前哪些region分配给了哪些region server，哪些region还没有分配。当需要分配的新的region，并且有一个region server上有可用空间时，master就给这个region server发送一个装载请求，把region分配给这个region server。region server得到请求后，就开始对此region提供服务。

#### region server上线

* Master使用ZooKeeper来跟踪region server状态
* 当某个region server启动时
  + 首先在zookeeper上的server目录下建立代表自己的znode
  + 由于Master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，master可以得到来自zookeeper的实时通知
  + 一旦region server上线，master能马上得到消息。

#### region server下线

* 当region 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
* Master就可以确定
  + region server和zookeeper之间的网络断开了
  + region server挂了
* 无论哪种情况，region server都无法继续为它的region提供服务了，此时master会删除server目录下代表这台region server的znode数据，并将这台region server的region分配给其它还活着的节点

#### Region分裂

* 当region中的数据逐渐变大之后，达到某一个阈值，会进行裂变
  + 一个region等分为两个region，并分配到不同的RegionServer
  + 原本的Region会下线，新Split出来的两个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。

|  |
| --- |
| <-- Region最大文件大小为10G -->  <property>  <name>hbase.hregion.max.filesize</name>  <value>10737418240</value>  <final>false</final>  <source>hbase-default.xml</source>  </property> |

* HBase只是增加数据，所有的更新和删除操作，都是在Compact阶段做的
* 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高性能读写

##### 自动分区

之前，我们在建表的时候，没有涉及过任何关于Region的设置，由HBase来自动进行分区。也就是Region达到一定大小就会自动进行分区。最小的分裂大小和table的某个region server的region 个数有关，当store file的大小大于如下公式得出的值的时候就会split，公式如下:

|  |
| --- |
| Min (R^2 \* “hbase.hregion.memstore.flush.size”, “hbase.hregion.max.filesize”) R为同一个table中在同一个region server中region的个数。 |

* 如果初始时R=1,那么Min(128MB,10GB)=128MB,也就是说在第一个flush的时候就会触发分裂操作
* 当R=2的时候Min(22128MB,10GB)=512MB ,当某个store file大小达到512MB的时候，就会触发分裂
* 如此类推，当R=9的时候，store file 达到10GB的时候就会分裂，也就是说当R>=9的时候，store file 达到10GB的时候就会分裂
* split 点都位于region中row key的中间点

##### 手动分区

在创建表的时候，就可以指定表分为多少个Region。默认一开始的时候系统会只向一个RegionServer写数据，系统不指定startRow和endRow，可以在运行的时候提前Split，提高并发写入。后续，我们会讲解如何进行预分区。

### Master工作机制

#### 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master成为master
2. 一般hbase集群中总是有一个master在提供服务，还有一个以上的‘master’在等待时机抢占它的位置。
3. 扫描zookeeper上的server父节点，获得当前可用的region server列表
4. 和每个region server通信，获得当前已分配的region和region server的对应关系
5. 扫描.META.region的集合，计算得到当前还未分配的region，将他们放入待分配region列表

#### Master下线

* 由于master只维护表和region的元数据，而不参与表数据IO的过程，master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region的负载均衡
  + 无法处理region 上下线
  + 无法进行region的合并
  + 唯一例外的是region的split可以正常进行，因为只有region server参与
  + 表的数据读写还可以正常进行
* 因此master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，master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）

## HBase批量装载——Bulk load

### 简介

很多时候，我们需要将外部的数据导入到HBase集群中，例如：将一些历史的数据导入到HBase做备份。我们之前已经学习了HBase的Java API，通过put方式可以将数据写入到HBase中，我们也学习过通过MapReduce编写代码将HDFS中的数据导入到HBase。但这些方式都是基于HBase的原生API方式进行操作的。这些方式有一个共同点，就是需要与HBase连接，然后进行操作。HBase服务器要维护、管理这些连接，以及接受来自客户端的操作，会给HBase的存储、计算、网络资源造成较大消耗。此时，在需要将海量数据写入到HBase时，通过Bulk load（大容量加载）的方式，会变得更高效。可以这么说，进行大量数据操作，Bulk load是必不可少的。

我们知道，HBase的数据最终是需要持久化到HDFS。HDFS是一个文件系统，那么数据可定是以一定的格式存储到里面的。例如：Hive我们可以以ORC、Parquet等方式存储。而HBase也有自己的数据格式，那就是HFile。Bulk Load就是直接将数据写入到StoreFile（HFile）中，从而绕开与HBase的交互，HFile生成后，直接一次性建立与HBase的关联即可。使用BulkLoad，绕过了Write to WAL，Write to MemStore及Flush to disk的过程

更多可以参考官方对Bulk load的描述：<https://hbase.apache.org/book.html#arch.bulk.load>

### Bulk load MapReduce程序开发

Bulk load的流程主要分为两步：

1. 通过MapReduce准备好数据文件（Store Files）
2. 加载数据文件到HBase

### 银行转账记录海量冷数据存储案例

银行每天都产生大量的转账记录，超过一定时期的数据，需要定期进行备份存储。本案例，在MySQL中有大量转账记录数据，需要将这些数据保存到HBase中。因为数据量非常庞大，所以采用的是Bulk Load方式来加载数据。

* 项目组为了方便数据备份，每天都会将对应的转账记录导出为CSV文本文件，并上传到HDFS。我们需要做的就将HDFS上的文件导入到HBase中。
* 因为我们只需要将数据读取出来，然后生成对应的Store File文件。所以，我们编写的MapReduce程序，只有Mapper，而没有Reducer。

#### 项目准备工作

##### HBase中创建银行转账记录表

|  |
| --- |
| create\_namespace "ITCAST\_BANK"  # disable "TRANSFER\_RECORD"  # drop "TRANSFER\_RECORD"  create "ITCAST\_BANK:TRANSFER\_RECORD", { NAME => "C1", COMPRESSION => "GZ"}, { NUMREGIONS => 6, SPLITALGO => "HexStringSplit"} |

##### 创建项目

|  |  |
| --- | --- |
| groupid | cn.itcast |
| artifactid | bankrecord\_bulkload |

##### 导入POM依赖

|  |
| --- |
| <**repositories**><!-- 代码库 -->  <**repository**>  <**id**>aliyun</**id**>  <**url**>http://maven.aliyun.com/nexus/content/groups/public/</**url**>  <**releases**>  <**enabled**>true</**enabled**>  </**releases**>  <**snapshots**>  <**enabled**>false</**enabled**>  <**updatePolicy**>never</**updatePolicy**>  </**snapshots**>  </**repository**> </**repositories**>  <**dependencies**>   <**dependency**>  <**groupId**>org.apache.hbase</**groupId**>  <**artifactId**>hbase-client</**artifactId**>  <**version**>2.1.0</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.hbase</**groupId**>  <**artifactId**>hbase-mapreduce</**artifactId**>  <**version**>2.1.0</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.hadoop</**groupId**>  <**artifactId**>hadoop-mapreduce-client-jobclient</**artifactId**>  <**version**>2.7.5</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.hadoop</**groupId**>  <**artifactId**>hadoop-common</**artifactId**>  <**version**>2.7.5</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.hadoop</**groupId**>  <**artifactId**>hadoop-mapreduce-client-core</**artifactId**>  <**version**>2.7.5</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.hadoop</**groupId**>  <**artifactId**>hadoop-auth</**artifactId**>  <**version**>2.7.5</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.hadoop</**groupId**>  <**artifactId**>hadoop-hdfs</**artifactId**>  <**version**>2.7.5</**version**>  </**dependency**>  <**dependency**>  <**groupId**>commons-io</**groupId**>  <**artifactId**>commons-io</**artifactId**>  <**version**>2.6</**version**>  </**dependency**> </**dependencies**>  <**build**>  <**plugins**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-compiler-plugin</**artifactId**>  <**version**>3.1</**version**>  <**configuration**>  <**target**>1.8</**target**>  <**source**>1.8</**source**>  </**configuration**>  </**plugin**>  </**plugins**> </**build**> |

##### 创建包结构

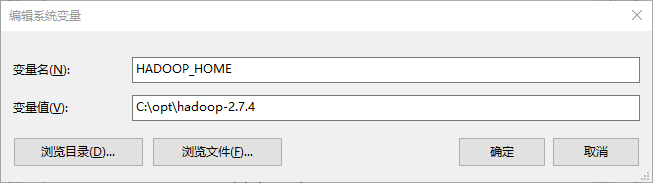
|  |  |
| --- | --- |
| 包 | 说明 |
| cn.itcast.bank\_record.bulkload.mr | MapReduce相关代码 |
| cn.itcast.bank\_record.entity | 实体类 |

##### 导入配置文件

将 core-site.xml、hbase-site.xml、log4j.properties三个配置文件拷贝到resources目录中。

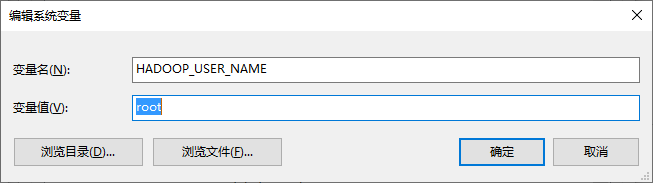
##### 确保Windows环境变量配置正确

1. HADOOP\_HOME



在资料包里面，有一个hadoop\_windows客户端文件夹，该文件夹中有一个压缩包，从压缩包中很多windows版本的客户端，找一个2.7.4版本，解压到指定目录即可。

1. HADOOP\_USER\_NAME



#### 编写实体类

实现步骤：

1. 创建实体类
2. 添加一个parse方法，用来将逗号分隔的字段，解析为实体类

|  |
| --- |
| **public class** TransferRecord {  **private** String **id**;  **private** String **code**;  **private** String **rec\_account**;  **private** String **rec\_bank\_name**;  **private** String **rec\_name**;  **private** String **pay\_account**;  **private** String **pay\_name**;  **private** String **pay\_comments**;  **private** String **pay\_channel**;  **private** String **pay\_way**;  **private** String **status**;  **private** String **timestamp**;  **private** Double **money**;   **public** TransferRecord() {  **this**.**id** = UUID.*randomUUID*().toString();  **this**.**code** = RandomStringUtils.*randomAlphanumeric*(12);  }   **public** String getId() {  **return id**;  }   **public void** setId(String id) {  **this**.**id** = id;  }   **public** String getCode() {  **return code**;  }   **public void** setCode(String code) {  **this**.**code** = code;  }   **public** String getRec\_account() {  **return rec\_account**;  }   **public void** setRec\_account(String rec\_account) {  **this**.**rec\_account** = rec\_account;  }   **public** String getRec\_bank\_name() {  **return rec\_bank\_name**;  }   **public void** setRec\_bank\_name(String rec\_bank\_name) {  **this**.**rec\_bank\_name** = rec\_bank\_name;  }   **public** String getRec\_name() {  **return rec\_name**;  }   **public void** setRec\_name(String rec\_name) {  **this**.**rec\_name** = rec\_name;  }   **public** String getPay\_account() {  **return pay\_account**;  }   **public void** setPay\_account(String pay\_account) {  **this**.**pay\_account** = pay\_account;  }   **public** String getPay\_name() {  **return pay\_name**;  }   **public void** setPay\_name(String pay\_name) {  **this**.**pay\_name** = pay\_name;  }   **public** String getPay\_comments() {  **return pay\_comments**;  }   **public void** setPay\_comments(String pay\_comments) {  **this**.**pay\_comments** = pay\_comments;  }   **public** String getPay\_channel() {  **return pay\_channel**;  }   **public void** setPay\_channel(String pay\_channel) {  **this**.**pay\_channel** = pay\_channel;  }   **public** String getPay\_way() {  **return pay\_way**;  }   **public void** setPay\_way(String pay\_way) {  **this**.**pay\_way** = pay\_way;  }   **public** String getStatus() {  **return status**;  }   **public void** setStatus(String status) {  **this**.**status** = status;  }   **public** String getTimestamp() {  **return timestamp**;  }   **public void** setTimestamp(String timestamp) {  **this**.**timestamp** = timestamp;  }   **public** Double getMoney() {  **return money**;  }   **public void** setMoney(Double money) {  **this**.**money** = money;  }   @Override  **public** String toString() {  **return id** + **","** + **code** + **","** + **rec\_account** + **","** + **rec\_bank\_name** + **","** + **rec\_name** + **","** + **pay\_account** + **","** + **pay\_name** + **","** + **pay\_comments** + **","** + **pay\_channel** + **","** + **pay\_way** + **","** + **status** + **","** + **timestamp** + **","** + **money**;  }   **public static** TransferRecord parse(String csvRow) {  String[] fieldArray = csvRow.split(**","**);  TransferRecord tr = **new** TransferRecord();  tr.setId(fieldArray[0]);  tr.setCode(fieldArray[1]);  tr.setRec\_account(fieldArray[2]);  tr.setRec\_bank\_name(fieldArray[3]);  tr.setRec\_name(fieldArray[4]);  tr.setPay\_account(fieldArray[5]);  tr.setPay\_name(fieldArray[6]);  tr.setPay\_comments(fieldArray[7]);  tr.setPay\_channel(fieldArray[8]);  tr.setPay\_way(fieldArray[9]);  tr.setStatus(fieldArray[10]);  tr.setTimestamp(fieldArray[11]);  tr.setMoney(Double.*parseDouble*(fieldArray[12]));   **return** tr;  } } |

#### 构建读取数据的Mapper

实现步骤：

1. 创建一个BankRecordMapper的类继承Mapper类，Mapper的泛型为
   1. 输入key：LongWritable
   2. 输入value：Text
   3. 输出key：ImmutableBytesWritable
   4. 输出value：MapReduceExtendedCell
2. 将Mapper获取到Text文本行，转换为TransferRecord实体类
3. 从实体类中获取ID，并转换为rowkey
4. 使用KeyValue类构建单元格，每个需要写入到表中的字段都需要构建出来单元格
5. 使用context.write将输出输出
   1. 构建输出key：new ImmutableBytesWrite(rowkey)
   2. 构建输出的value：new MapReduceExtendedCell(keyvalue对象)

|  |
| --- |
| **package** cn.itcast.bank\_record.mr;  **import** cn.itcast.bank\_record.tool.TransferRecord; **import** org.apache.hadoop.conf.Configuration; **import** org.apache.hadoop.fs.Path; **import** org.apache.hadoop.hbase.HBaseConfiguration; **import** org.apache.hadoop.hbase.KeyValue; **import** org.apache.hadoop.hbase.TableName; **import** org.apache.hadoop.hbase.client.\*; **import** org.apache.hadoop.hbase.io.ImmutableBytesWritable; **import** org.apache.hadoop.hbase.mapreduce.HFileOutputFormat2; **import** org.apache.hadoop.hbase.util.Bytes; **import** org.apache.hadoop.hbase.util.MapReduceExtendedCell; **import** org.apache.hadoop.io.LongWritable; **import** org.apache.hadoop.io.Text; **import** org.apache.hadoop.mapreduce.Job; **import** org.apache.hadoop.mapreduce.Mapper; **import** org.apache.hadoop.mapreduce.lib.input.FileInputFormat; **import** org.apache.hadoop.mapreduce.lib.input.TextInputFormat; **import** org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  **import** java.io.IOException;  **public class** BankRecordBulkLoad {  // 实现Mapper  // <KEYIN, VALUEIN> - 文件偏移量，文本  // <KEYOUT, VALUEOUT> - HBase ROWKEY，列族:列 - 值  // ImmutableBytesWrite - ROWKEY对应MapReduce中的类型  // MapReduceExtendedCell - MapReduce对应的Cell单元格数据  **static class** BankRecordMapper **extends** Mapper<LongWritable, Text, ImmutableBytesWritable, MapReduceExtendedCell> {  **protected final** String **CF\_NAME** = **"C1"**;   @Override  **protected void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {  // 1. 解析文本数据  String rowText = value.toString();  TransferRecord tr = TransferRecord.*parse*(rowText);  // 2. 构建ROWKEY  String rowkey = tr.getId();  // 3. 构建列值KeyValue对  KeyValue keyValueId = **new** KeyValue(Bytes.*toBytes*(rowkey), Bytes.*toBytes*(**CF\_NAME**), Bytes.*toBytes*(**"id"**), Bytes.*toBytes*(tr.getId()));  KeyValue keyValueCode = **new** KeyValue(Bytes.*toBytes*(rowkey), Bytes.*toBytes*(**CF\_NAME**), Bytes.*toBytes*(**"code"**), Bytes.*toBytes*(tr.getCode()));  KeyValue keyValueRec\_account = **new** KeyValue(Bytes.*toBytes*(rowkey), Bytes.*toBytes*(**CF\_NAME**), Bytes.*toBytes*(**"rec\_account"**), Bytes.*toBytes*(tr.getRec\_account()));  KeyValue keyValueRec\_bank\_name = **new** KeyValue(Bytes.*toBytes*(rowkey), Bytes.*toBytes*(**CF\_NAME**), Bytes.*toBytes*(**"rec\_bank\_name"**), Bytes.*toBytes*(tr.getRec\_bank\_name()));  KeyValue keyValueRec\_name = **new** KeyValue(Bytes.*toBytes*(rowkey), Bytes.*toBytes*(**CF\_NAME**), Bytes.*toBytes*(**"rec\_name"**), Bytes.*toBytes*(tr.getRec\_name()));  KeyValue keyValuePay\_account = **new** KeyValue(Bytes.*toBytes*(rowkey), Bytes.*toBytes*(**CF\_NAME**), Bytes.*toBytes*(**"pay\_account"**), Bytes.*toBytes*(tr.getPay\_account()));  KeyValue keyValuePay\_name = **new** KeyValue(Bytes.*toBytes*(rowkey), Bytes.*toBytes*(**CF\_NAME**), Bytes.*toBytes*(**"pay\_name"**), Bytes.*toBytes*(tr.getPay\_name()));  KeyValue keyValuePay\_comments = **new** KeyValue(Bytes.*toBytes*(rowkey), Bytes.*toBytes*(**CF\_NAME**), Bytes.*toBytes*(**"pay\_comments"**), Bytes.*toBytes*(tr.getPay\_comments()));  KeyValue keyValuePay\_channel = **new** KeyValue(Bytes.*toBytes*(rowkey), Bytes.*toBytes*(**CF\_NAME**), Bytes.*toBytes*(**"pay\_channel"**), Bytes.*toBytes*(tr.getPay\_channel()));  KeyValue keyValuePay\_way = **new** KeyValue(Bytes.*toBytes*(rowkey), Bytes.*toBytes*(**CF\_NAME**), Bytes.*toBytes*(**"pay\_way"**), Bytes.*toBytes*(tr.getPay\_way()));  KeyValue keyValueStatus = **new** KeyValue(Bytes.*toBytes*(rowkey), Bytes.*toBytes*(**CF\_NAME**), Bytes.*toBytes*(**"status"**), Bytes.*toBytes*(tr.getStatus()));  KeyValue keyValueTimestamp = **new** KeyValue(Bytes.*toBytes*(rowkey), Bytes.*toBytes*(**CF\_NAME**), Bytes.*toBytes*(**"timestamp"**), Bytes.*toBytes*(tr.getTimestamp()));  KeyValue keyValueMoney = **new** KeyValue(Bytes.*toBytes*(rowkey), Bytes.*toBytes*(**CF\_NAME**), Bytes.*toBytes*(**"money"**), Bytes.*toBytes*(tr.getMoney()));   context.write(**new** ImmutableBytesWritable(Bytes.*toBytes*(rowkey)), **new** MapReduceExtendedCell(keyValueId));  context.write(**new** ImmutableBytesWritable(Bytes.*toBytes*(rowkey)), **new** MapReduceExtendedCell(keyValueCode));  context.write(**new** ImmutableBytesWritable(Bytes.*toBytes*(rowkey)), **new** MapReduceExtendedCell(keyValueRec\_account));  context.write(**new** ImmutableBytesWritable(Bytes.*toBytes*(rowkey)), **new** MapReduceExtendedCell(keyValueRec\_bank\_name));  context.write(**new** ImmutableBytesWritable(Bytes.*toBytes*(rowkey)), **new** MapReduceExtendedCell(keyValueRec\_name));  context.write(**new** ImmutableBytesWritable(Bytes.*toBytes*(rowkey)), **new** MapReduceExtendedCell(keyValuePay\_account));  context.write(**new** ImmutableBytesWritable(Bytes.*toBytes*(rowkey)), **new** MapReduceExtendedCell(keyValuePay\_name));  context.write(**new** ImmutableBytesWritable(Bytes.*toBytes*(rowkey)), **new** MapReduceExtendedCell(keyValuePay\_comments));  context.write(**new** ImmutableBytesWritable(Bytes.*toBytes*(rowkey)), **new** MapReduceExtendedCell(keyValuePay\_channel));  context.write(**new** ImmutableBytesWritable(Bytes.*toBytes*(rowkey)), **new** MapReduceExtendedCell(keyValuePay\_way));  context.write(**new** ImmutableBytesWritable(Bytes.*toBytes*(rowkey)), **new** MapReduceExtendedCell(keyValueStatus));  context.write(**new** ImmutableBytesWritable(Bytes.*toBytes*(rowkey)), **new** MapReduceExtendedCell(keyValueTimestamp));  context.write(**new** ImmutableBytesWritable(Bytes.*toBytes*(rowkey)), **new** MapReduceExtendedCell(keyValueMoney));  }  } } |

#### 编写驱动类

实现步骤：

1. 使用HBaseConfiguration.create()加载配置文件
2. 创建HBase连接
3. 获取HTable
4. 构建MapReduce JOB
   1. 使用Job.getInstance构建一个Job对象
   2. 调用setJarByClass设置要执行JAR包的class
   3. 调用setInputFormatClass为TextInputFormat.class
   4. 设置MapperClass
   5. 设置输出键Output Key Class
   6. 设置输出值Output Value Class
   7. 设置输入输出到HDFS的路径
      1. FileInputFormat.setInputPaths
      2. FileOutputFormat.setOutputPath
   8. 使用connection.getRegionLocator获取HBase Region的分布情况
   9. 使用HFileOutputFormat2.configureIncrementalLoad配置HFile输出
5. 调用job.waitForCompletion执行MapReduce程序

|  |
| --- |
| **public static void** main(String[] args) **throws** IOException, ClassNotFoundException, InterruptedException {  // 1. 构建HBase配置  // 1.1 配置文件  Configuration conf = HBaseConfiguration.*create*();  conf.addResource(BankRecordBulkLoad.**class**.getClassLoader().getResource(**"hbase-site.xml"**));  // 1.2 获取HBase连接  Connection connection = ConnectionFactory.*createConnection*(conf);  // 1.3 获取HTable  String tableName = **"ITCAST\_BANK:TRANSFER\_RECORD"**;  Table htable = connection.getTable(TableName.*valueOf*(tableName));   // 2. 构建MapReduce JOB  // 2.1 构建JOB  Job job = Job.*getInstance*(conf, **"Transfer\_BulkLoad\_Job"**);  // 2.2 设置JOB主类、Mapper、输入  job.setJarByClass(BankRecordBulkLoad.**class**);  job.setInputFormatClass(TextInputFormat.**class**);  job.setMapperClass(BankRecordMapper.**class**);  job.setOutputKeyClass(ImmutableBytesWritable.**class**);  job.setOutputValueClass(MapReduceExtendedCell.**class**);   // 2.3 设置输入、输出  FileInputFormat.*setInputPaths*(job, **new** Path(**"hdfs://node1.itcast.cn:8020/bank/input"**));  FileOutputFormat.*setOutputPath*(job, **new** Path(**"hdfs://node1.itcast.cn:8020/bank/output"**));   // 3. 配置HBase StoreFile输出  // 获取HBase Region的分布情况  RegionLocator regionLocator = connection.getRegionLocator(TableName.*valueOf*(tableName));  // 配置HFile输出  HFileOutputFormat2.*configureIncrementalLoad*(job, htable, regionLocator);   // 4. 启动执行MapReduce JOB  **if**(job.waitForCompletion(**true**)) {  System.*exit*(0);  }  **else** {  System.*exit*(1);  }  } |

#### 上传数据到文件到HDFS

将资料中的数据集 bank\_record.csv 上传到HDFS的 /bank/input 目录。该文件中包含50W条的转账记录数据。

|  |
| --- |
| hadoop fs -mkdir -p /bank/input  hadoop fs -put bank\_record.csv /bank/input |

然后，执行MapReduce程序。

#### 加载数据文件到HBase

|  |
| --- |
| hbase org.apache.hadoop.hbase.tool.LoadIncrementalHFiles /bank/output ITCAST\_BANK:TRANSFER\_RECORD |

## HBase的协处理器（Coprocessor）

<http://hbase.apache.org/book.html#cp>

### 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

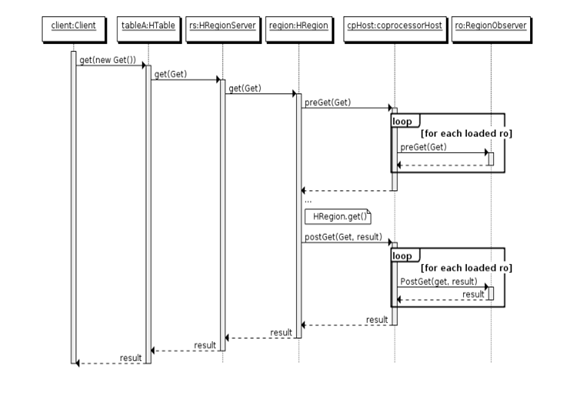
* 于是， HBase 在 0.92 之后引入了协处理器(coprocessors)，实现一些激动人心的新特性：能够轻易建立二次索引、复杂过滤器(谓词下推)以及访问控制等。

### 协处理器有两种： observer 和 endpoint

#### 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 Hbase2.0.0 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：

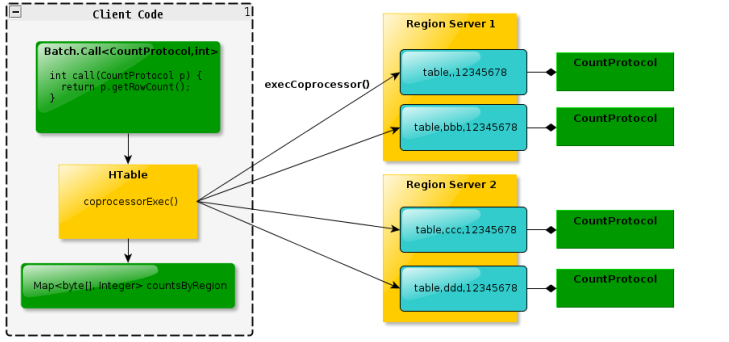


1. 客户端发起get请求
2. 该请求被分派给合适的RegionServer和Region
3. coprocessorHost拦截该请求，然后在该表上登记的每个RegionObserer上调用preGet()
4. 如果没有被preGet拦截，该请求继续送到Region，然后进行处理
5. Region产生的结果再次被coprocessorHost拦截，调用posGet()处理
6. 加入没有postGet()拦截该响应，最终结果被返回给客户端

#### 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：



#### 总结

* Observer 允许集群在正常的客户端操作过程中可以有不同的行为表现
* Endpoint 允许扩展集群的能力，对客户端应用开放新的运算命令
* observer 类似于 RDBMS 中的触发器，主要在服务端工作
* endpoint 类似于 RDBMS 中的存储过程，主要在 client 端工作
* observer 可以实现权限管理、优先级设置、监控、 ddl 控制、 二级索引等功能
* endpoint 可以实现 min、 max、 avg、 sum、 distinct、 group by 等功能

### 协处理器加载方式

协处理器的加载方式有两种：

* 静态加载方式（ Static Load）
* 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。

#### 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> |

为所有 table 加载了一个 cp class，可以用” ,”分割加载多个 class

#### 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||' |

* 重启启用表

|  |
| --- |
| hbase> enable 'mytable' |

#### 协处理器卸载

只需三步：

|  |
| --- |
| disable ‘test’  alter ‘test’, METHOD => ‘table\_att\_unset’, NAME => ‘coprocessor$1’  enable ‘test’ |

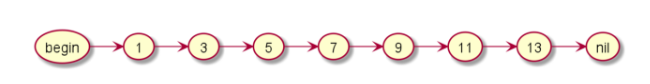
## HBase事务

HBase 支持特定场景下的 ACID，即当对同一行进行 Put 操作时保证完全的 ACID。可以简单理解为针对一行的操作，是有事务性保障的。HBase也没有混合读写事务。也就是说，我们无法将读操作、写操作放入到一个事务中。

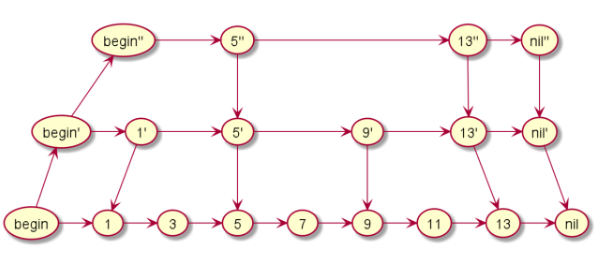
## HBase数据结构

在讲解HBase的LSM合并树之前，我们需要来了解一些常用的数据结构知识。

### 跳表



上图是一个有序链表，我们要检索一个数据就挨个遍历。如果想要再提升查询效率，可以变种为以下结构：



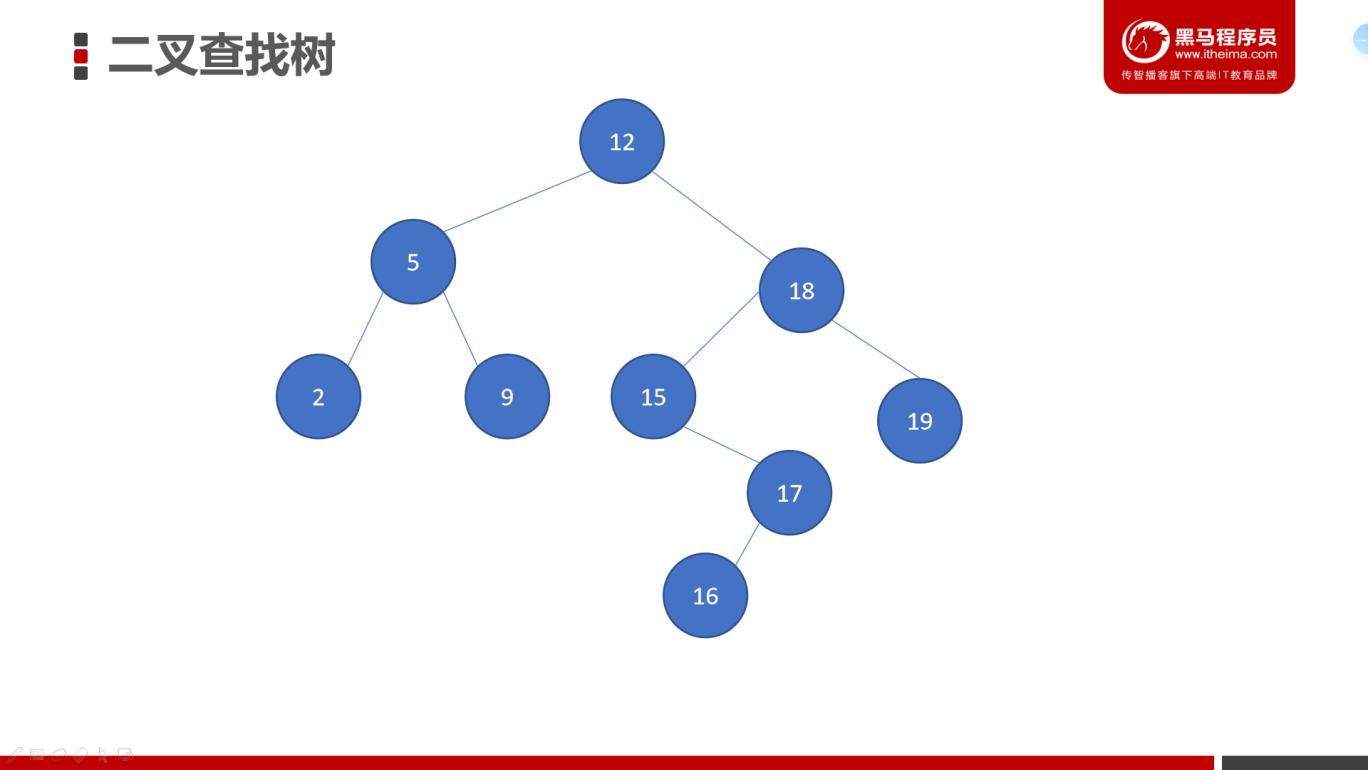
现在，我们要查询11，可以跳着来查询，从而加快查询速度。

### 常见树结构（扩展了解）

#### 二叉搜索树（Binary Search Tree）

##### 什么是二叉搜索树？

二叉搜索树也叫二叉查找树。它是一种比较特殊的二叉树。



##### 树的高度、深度、层数

* 深度

节点的深度是根节点到这个节点所经历的边的个数，深度是从上往下数的

* 高度

节点的高度是该节点到叶子节点的最长路径（边数），高度是从下往上数的

* 层数

根节点为第一层，往下依次递增

上图：

* 节点12的深度为0，高度为4，在第1层
* 节点15的深度为2，高度为2，在第3层

##### 二叉搜索树的特点

树中的每个节点，它的左子树中所有关键字值小于该节点关键字值，右子树中所有关键字值大于该节点关键字值

##### 二叉搜索树的查询方式

* 首先和根节点进行比较，如果等于根节点，则返回
* 如果小于根节点，则在根节点的左子树进行查找
* 如果大于根节点，则在根节点的右子树进行查找

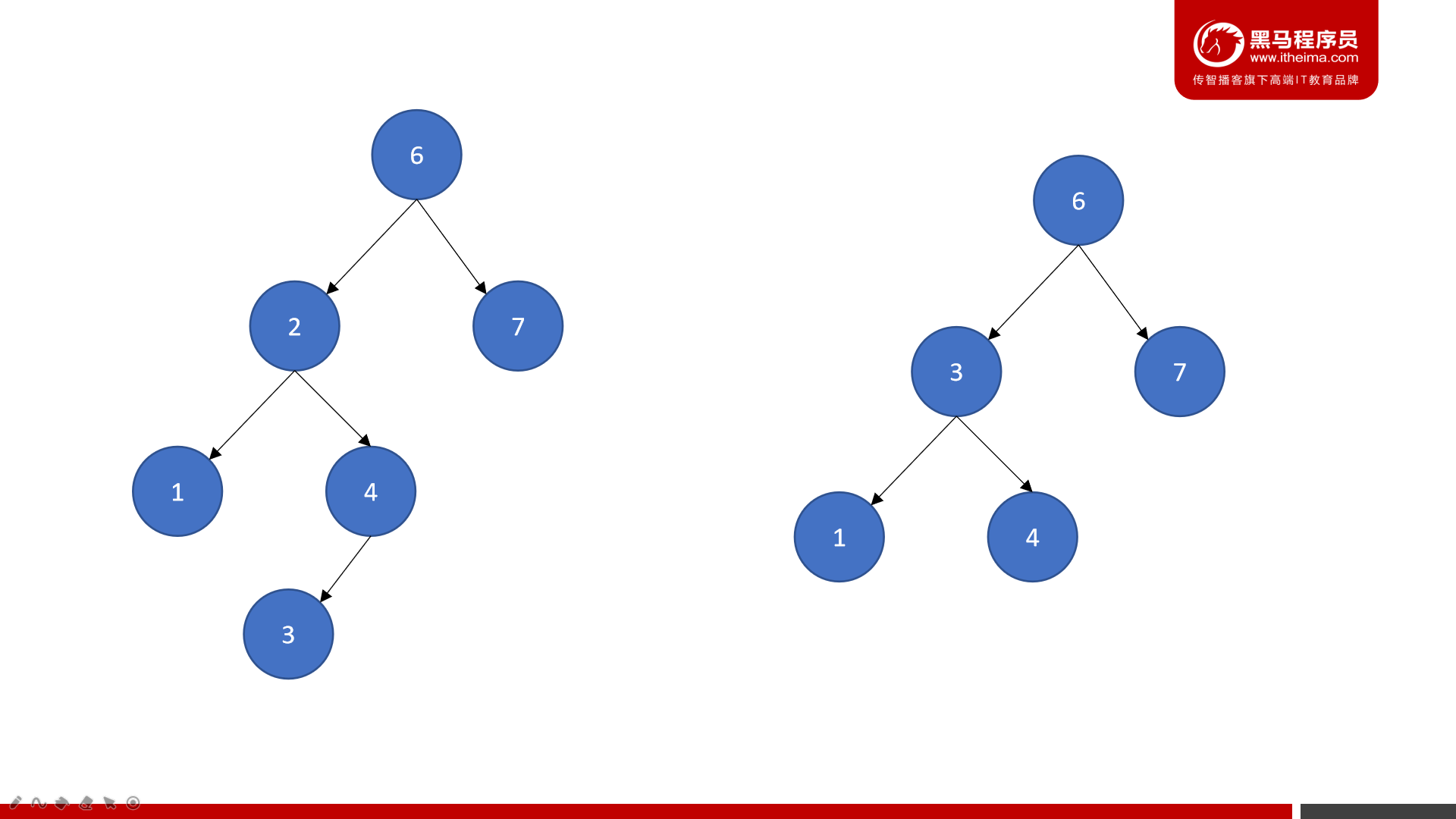
##### 二叉搜索树的缺点

因为二叉搜索树是一种二叉树，每个节点只能有两个子节点，但有较多节点时，整棵树的高度会比较大，树的高度越大，搜索的性能开销也就越大

#### 平衡二叉树（Balance Binary Tree）

##### 简介

* 平衡二叉树也称为AVL数
* 它是一颗空数，或者它的任意节点左右两个子树的高度差绝对值**不超过1**
* 平衡二叉树很好地解决了二叉查找树退化成链表的问题



上图：

1. 两棵树都是二叉查找树
2. 左边的不是平衡二叉树

节点6的子节点：节点2的高度为：2，节点7的高度为：0，| 2 – 0 | = 2 > 1）

1. 右边的是平衡二叉树

节点6的子节点：节点3的高度为：1，节点7的高度为：0，| 1 – 0 | = 1 = 1 ）

##### 平衡二叉树的特点

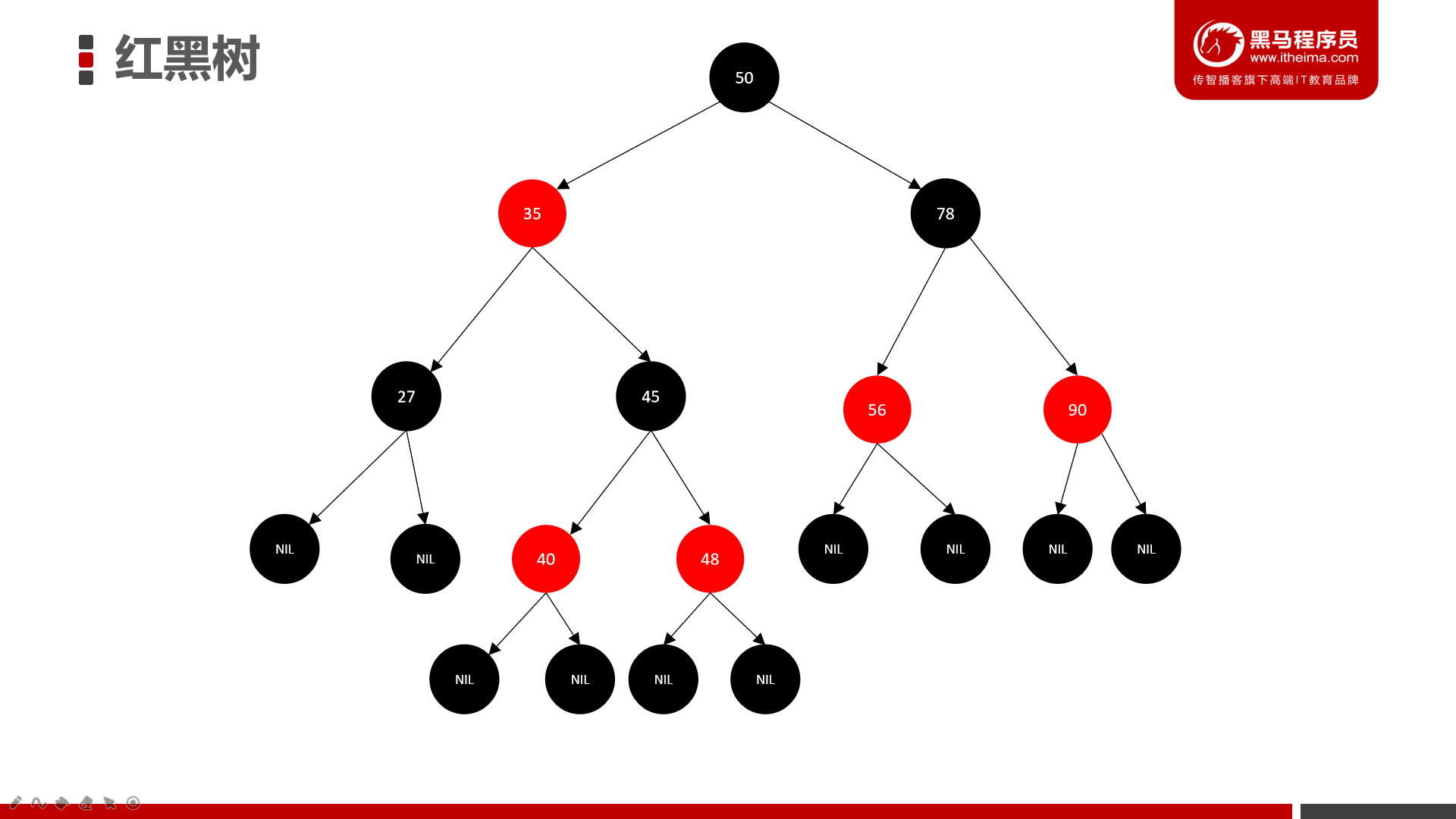
AVL树是高度平衡的（严格平衡），频繁的插入和删除，会引起频繁的rebalance，导致效率下降，它比较使用与插入/删除较少，查找较多的场景

#### 红黑树

##### 简介

红黑树是一种含有红黑节点并能自平衡的二叉搜索树，它满足以下性质：

* 每个节点要么是黑色，要么是红色
* 根节点是黑色
* 每个叶子节点（NIL）是黑色
* 每个红色结点的两个子结点一定都是黑色
* 任意一结点到每个叶子结点的路径都包含数量相同的黑结点



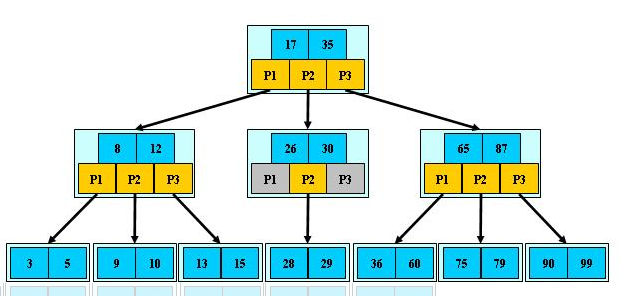
##### 红黑树的特点

和AVL树不一样，红黑树是一种弱平衡的二叉树，它的插入/删除效率更高，所以对于插入、删除较多的情况下，就用红黑树，而且查找效率也不低。例如：Java中的TreeMap就是基于红黑树实现的。

#### B树

##### 什么是B树

* B树是一种**平衡多路搜索树**
* 与二叉搜索树不同的是，B树的节点可以有多个子节点，不限于最多两个节点
* 它的子节点可以是几个或者是几千个



##### B树的特点

* 所有节点关键字是按递增次序排列，并遵循左小右大原则
* B-树有个最大的特点是有多个查找路径，而不像二叉搜索树，只有两路查找路径。
* 所有的叶子节点在同一层
* **逐层查找，找到节点后返回**

##### B-树的查找方式

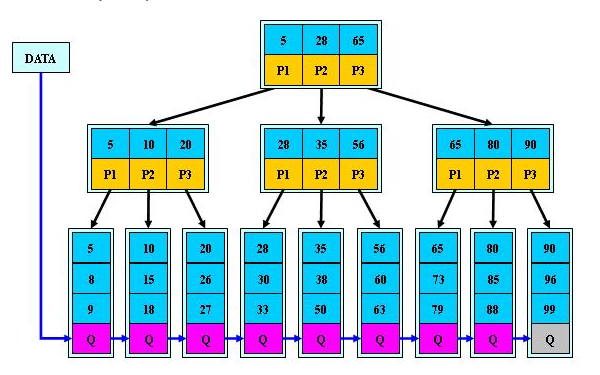
* 从根节点的关键字开始比较，例如：上图为13，判断大于还是小于
* 继续往下查找，因为节点可能会有多个节点，所以需要判断属于哪个区间
* 不断往下查找，直到找到为止或者没有找到返回Null

#### B+树结构

##### B+树简介

B+树是B树的升级版。B+树常用在**文件系统和数据库**中，B+树通过对每个节点存储数据的个数进行扩展，**可以让连续的数据进行快速访问，有效减少查询时间**，减少IO操作。它能够保持数据稳定有序，其插入与修改拥有较稳定的对数时间复杂度

例如：Linux的Ext3文件系统、Oracle、MySQL、SQLServer都会使用到B+树。



* B+ 树是一种树数据结构，是一个n叉树
* 每个节点通常有多个孩子
* 一颗B+树包含根节点、内部节点和叶子节点
* **只有叶子节点包含数据（所有数据都是在叶子节点中出现）**

##### B+树的特点

* 所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的

如果执行的是：select \* from user order by id，要全表扫描数据，那么B树就比较费劲了，但B+树就容易了，只要遍历最后的链表就可以了。

* 只会在叶子节点上搜索到数据
* 非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储
* 数据库的B+树高度大概在 2-4 层，也就是说查询到某个数据最多需要2到4次IO，相当于0.02到0.04s

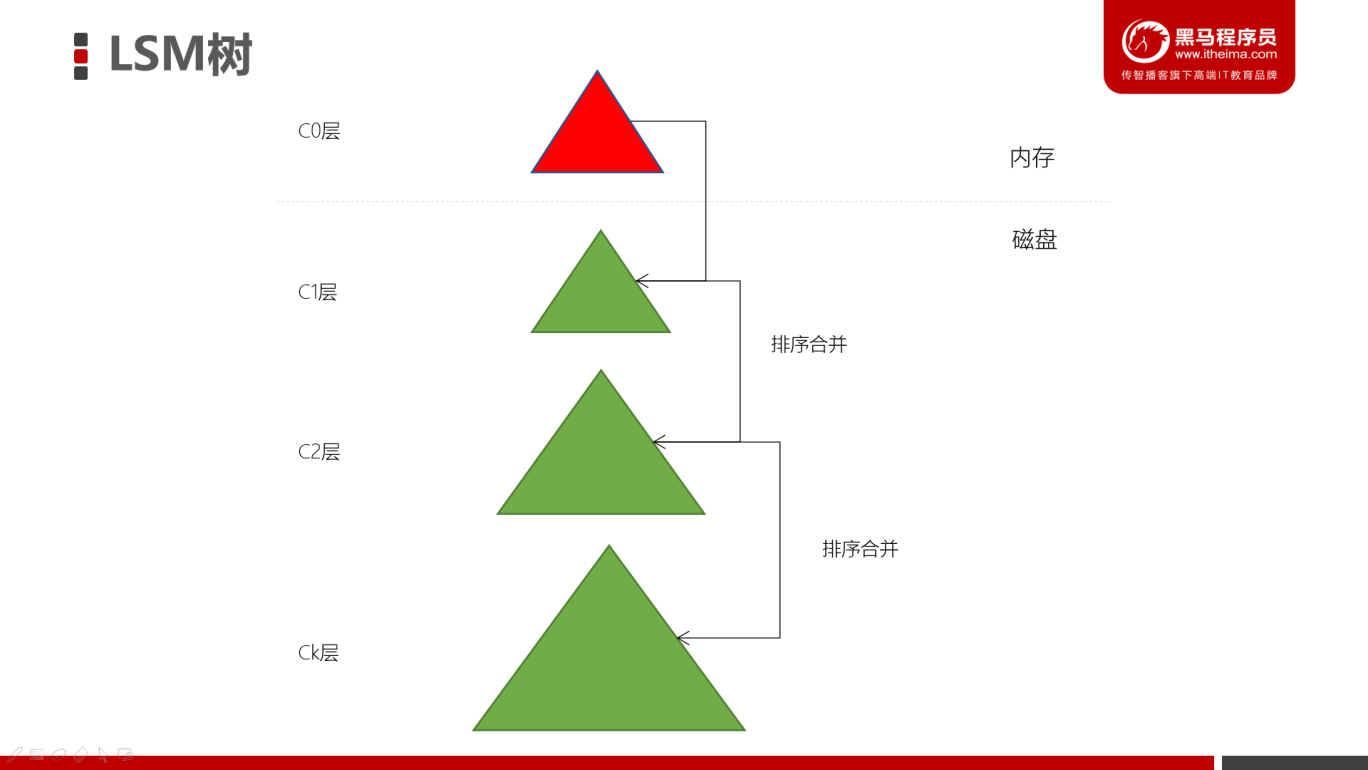
### LSM树数据结构

#### 简介

传统关系型数据库，一般都选择使用B+树作为索引结构，而在大数据场景下，HBase、Kudu这些存储引擎选择的是LSM树。LSM树，即日志结构合并树(Log-Structured Merge-Tree)。

* LSM树主要目标是快速建立索引
* B树是建立索引的通用技术，但如果并发写入压力较大时，B树需要大量的磁盘**随机IO**，而严重影响索引创建的速度，在一些写入操作非常频繁的应用场景中，就不太适合了
* LSM树通过磁盘的**顺序写**，来实现最好的写性能

#### LSM树设计思想



* LSM 的主要思想是划分不同等级的结构，换句话来理解，就是LSM中不止一个数据结构，而是存在多种结构
* 一个结构在内存、其他结构在磁盘（HBase存储结构中，有内存——MemStore、也有磁盘——StoreFile）
* 内存的结构可以是B树、红黑树、跳表等结构（HBase中是跳表），磁盘中的树就是一颗B+树
* C0层保存了最近写入的数据，数据都是有序的，而且可以随机更新、随机查询
* C1到CK层的数据都是存在磁盘中，每一层中key都是有序存储的

#### LSM的数据写入操作

* 首先将数据写入到WAL（Write Ahead log），写日志是顺序写，效率相对较高（PUT、DELETE都是顺序写）
* 数据项写入到内存中的C0结构中
* 只有内存中的C0结构超过一定阈值的时，将内存中的C0、和C1进行合并。这个过程就是Compaction（合并）
* 合并后的新的C1顺序写磁盘，替换之前的C1
* 但C1层达到一定的大小，会继续和下层合并，合并后旧的文件都可以删除，只保留最新的
* 整个写入的过程只用到了内存结构，Compaction由后台异步完成，不阻塞写入

#### LSM的数据查询操作

* 先在内存中查C0层
* 如果C0层中不存在数据，则查询C1层
* 不断逐层查询，最早的数据在CK层
* C0层因为是在内存中的结构中查询，所以效率较高。因为数据都是分布在不同的层结构中，所以一次查询，可能需要多次跨层次结构查询，所以读取的速度会慢一些。
* 根据以上，LSM树结构的程序适合于写密集、少量查询的场景

### 布隆过滤器

#### 简介

客户端：这个key存在吗？

服务器：不存在/不知道

本质上，布隆过滤器是一种数据结构，是一种比较巧妙的概率型数据结构。它的特点是高效地插入和查询。但我们要检查一个key是否在某个结构中存在时，通过使用布隆过滤器，我们可以快速了解到「这个key一定不存在或者可能存在」。相比于以前学习过的List、Set、Map这些数据结构，它更加高效、占用的空间也越少，但是它返回的结果是概率性的，是不确切的。

#### 应用场景

##### 缓存穿透

* 为了提高访问效率，我们会将一些数据放在Redis缓存中。当进行数据查询时，可以先从缓存中获取数据，无需读取数据库。这样可以有效地提升性能。
* 在数据查询时，首先要判断缓存中是否有数据，如果有数据，就直接从缓存中获取数据。
* 但如果没有数据，就需要从数据库中获取数据，然后放入缓存。如果大量访问都无法命中缓存，会造成数据库要扛较大压力，从而导致数据库崩溃。而使用布隆过滤器，当访问不存在的缓存时，可以迅速返回避免缓存或者DB crash。

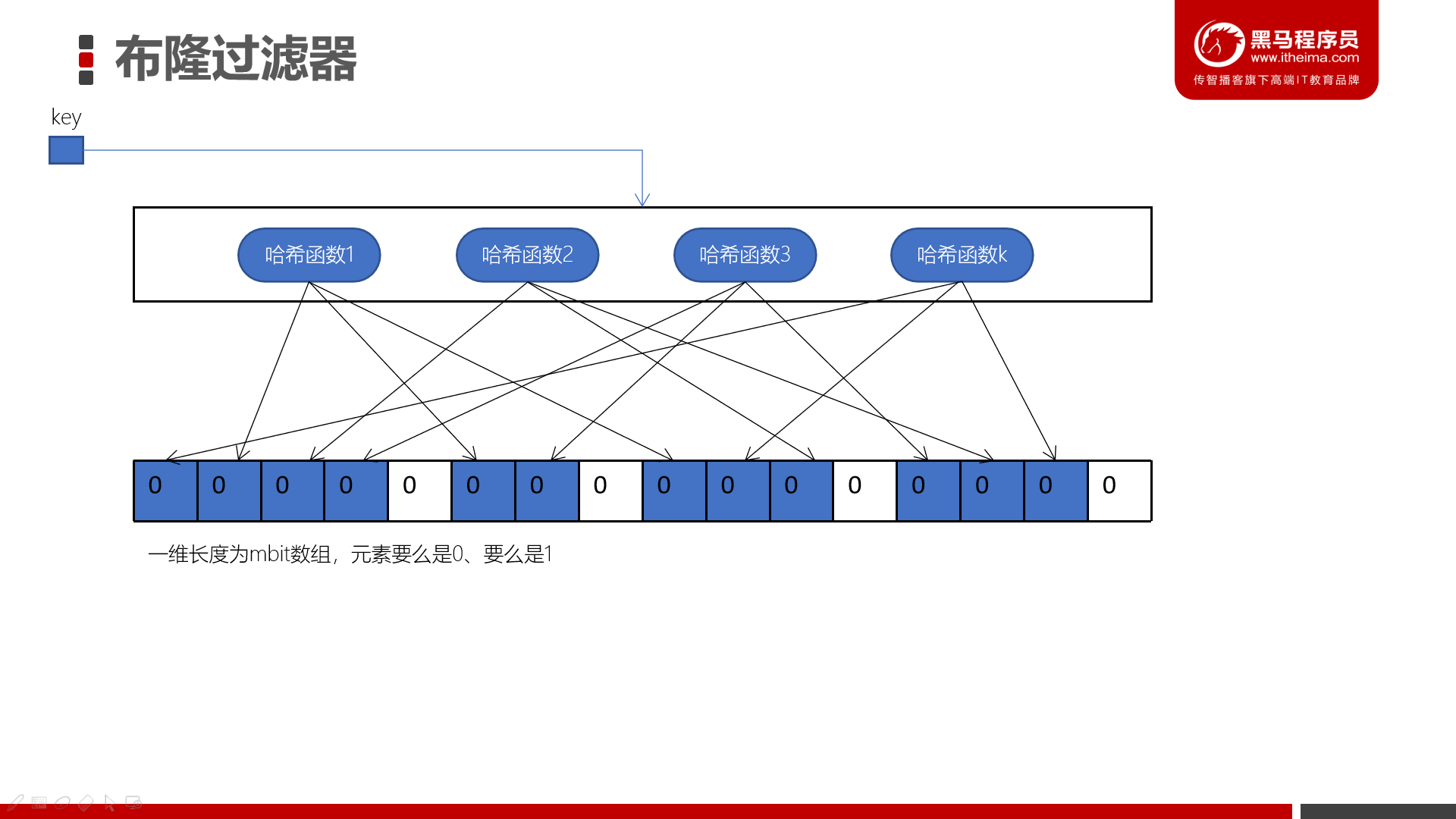
##### 判断某个数据是否在海量数据中存在

HBase中存储着非常海量数据，要判断某个ROWKEYS、或者某个列是否存在，使用布隆过滤器，可以快速获取某个数据是否存在。但有一定的误判率。但如果某个key不存在，一定是准确的。

#### HashMap的问题

* 要判断某个元素是否存在其实用HashMap效率是非常高的。HashMap通过把值映射为HashMap的Key，这种方式可以实现O(1)常数级时间复杂度。
* 但是，如果存储的数据量非常大的时候（例如：上亿的数据），HashMap将会耗费非常大的内存大小。而且也根本无法一次性将海量的数据读进内存。

#### 理解布隆过滤器



* 布隆过滤器是一个bit数组或者称为一个bit二进制向量
* 这个数组中的元素存的要么是0、要么是1
* k个hash函数都是彼此独立的，并将每个hash函数计算后的结果对数组的长度m取模，并将对一个的bit设置为1（蓝色单元格）
* 我们将每个key都按照这种方式设置单元格，就是「**布隆过滤器**」

#### 根据布隆过滤器查询元素

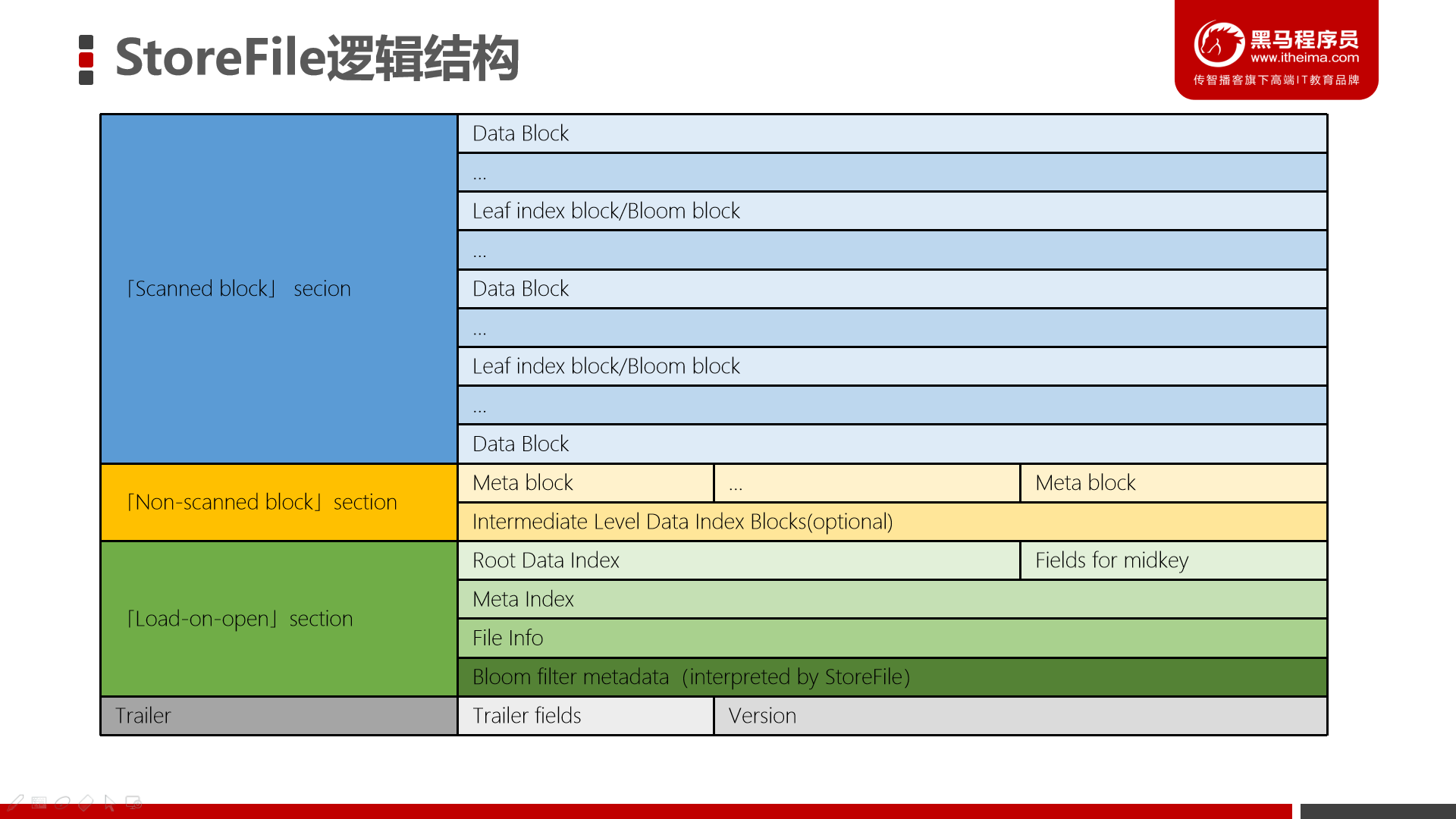
* 假设输入一个key，我们使用之前的k个hash函数求哈希，得到k个值
* 判断这k个值是否都为蓝色，如果有一个不是黑，那么这个key一定不存在
* 如果都有蓝色，那么key是可能存在（布隆过滤器会存在误判）
* 因为如果输入对象很多，而集合比较小的情况，会导致集合中大多位置都会被描黑，那么检查某个key时候为蓝色时，刚好某个位置正好被设置为蓝色了，此时，会错误认为该key在集合中

### StoreFiles（HFile）结构

StoreFile是HBase存储数据的文件格式。

#### HFile的逻辑结构

##### HFile逻辑结构图



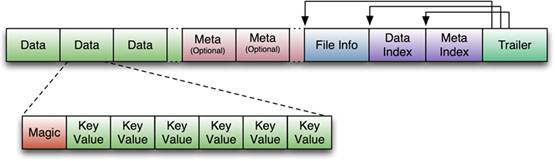
##### 逻辑结构说明

4大部分

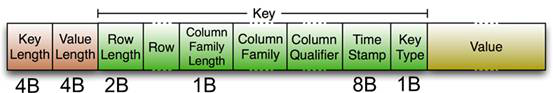
* Scanned block section
  + 扫描StoreFile时，所有的Data Block（数据块）都将会被读取
  + Leaf Index（LSM + C1树索引）、Bloom block（布隆过滤器）都会被读取
* Non-scanned block section
  + 扫描StoreFile时，不会被读取
  + 包含MetaBlock和Intermediate Level Data Index Blocks
* Opening-time data section
  + 在RegionServer启动时，需要将数据加载到内存中，包括数据块索引、元数据索引、布隆过滤器、文件信息。
* Trailer
  + 记录了HFile的基本信息
  + 各个部分的偏移值和寻址信息

#### StoreFile物理结构

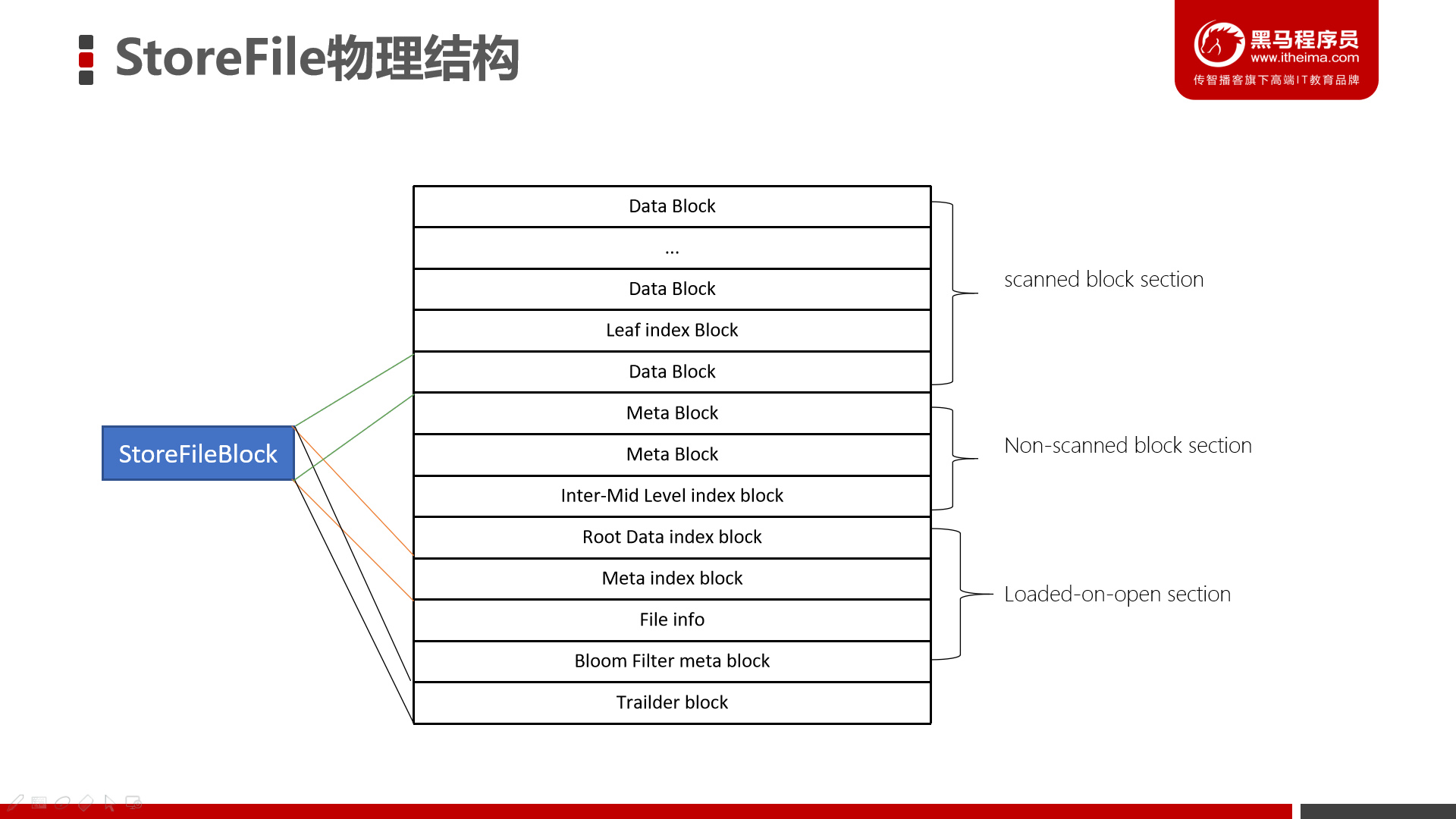
StoreFile是以Hfile的形式存储在HDFS上的。Hfile的格式为下图：

[](http://www.searchtb.com/wp-content/uploads/2011/01/image0080.jpg)

* 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：

[](http://www.searchtb.com/wp-content/uploads/2011/01/image0090.jpg)

1. 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
2. 紧接着是Key，开始是固定长度的数值，表示RowKey的长度
3. 紧接着是 RowKey，然后是固定长度的数值，表示Family的长度
4. 然后是Family，接着是Qualifier
5. 然后是两个固定长度的数值，表示Time Stamp和Key Type（Put/Delete）——每一种操作都会生成一个Key-Value。Value部分没有这么复杂的结构，就是纯粹的二进制数据了。



* Data Block段

保存表中的数据，这部分可以被压缩

* Meta Block段 (可选的)

保存用户自定义的kv对，可以被压缩。

* 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。

* 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。

* Meta Block Index段 (可选的)

Meta Block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

* 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。

## HBase调优

### 通用优化

#### NameNode的元数据备份使用SSD



#### 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。

#### 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。
* 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。

#### NameNode节点配置为RAID1（镜像盘）结构



#### 补充：什么是Raid0、Raid0+1、Raid1、Raid5



**Standalone**

最普遍的单磁盘储存方式。

**Cluster**

集群储存是通过将数据分布到集群中各节点的存储方式,提供单一的使用接口与界面,使用户可以方便地对所有数据进行统一使用与管理。

**Hot swap**

用户可以再不关闭系统,不切断电源的情况下取出和更换硬盘,提高系统的恢复能力、拓展性和灵活性。

**Raid0**

Raid0是所有raid中存储性能最强的阵列形式。其工作原理就是在多个磁盘上分散存取连续的数据,这样,当需要存取数据是多个磁盘可以并排执行,每个磁盘执行属于它自己的那部分数据请求,显著提高磁盘整体存取性能。但是不具备容错能力,适用于低成本、低可靠性的台式系统。

**Raid1**

又称镜像盘,把一个磁盘的数据镜像到另一个磁盘上,采用镜像容错来提高可靠性,具有raid中最高的数据冗余能力。存数据时会将数据同时写入镜像盘内,读取数据则只从工作盘读出。发生故障时,系统将从镜像盘读取数据,然后再恢复工作盘正确数据。这种阵列方式可靠性极高,但是其容量会减去一半。广泛用于数据要求极严的应用场合,如商业金融、档案管理等领域。只允许一颗硬盘出故障。

**Raid0+1**

将Raid0和Raid1技术结合在一起,兼顾两者的优势。在数据得到保障的同时,还能提供较强的存储性能。不过至少要求4个或以上的硬盘，但也只允许一个磁盘出错。是一种三高技术。

**Raid5**

Raid5可以看成是Raid0+1的低成本方案。采用循环偶校验独立存取的阵列方式。将数据和相对应的奇偶校验信息分布存储到组成RAID5的各个磁盘上。当其中一个磁盘数据发生损坏后,利用剩下的磁盘和相应的奇偶校验信息 重新恢复/生成丢失的数据而不影响数据的可用性。至少需要3个或以上的硬盘。适用于大数据量的操作。成本稍高、储存性强、可靠性强的阵列方式。

RAID还有其他方式，请自行查阅。

#### 保持NameNode日志目录有足够的空间，有助于帮助发现问题

#### Hadoop是IO密集型框架，所以尽量提升存储的速度和吞吐量

### Linux优化

#### 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

（尖叫提示：ra是readahead的缩写）

#### 最大限度使用物理内存

$ sudo sysctl -w vm.swappiness=0

* swappiness，Linux内核参数，控制换出运行时内存的相对权重
* swappiness参数值可设置范围在0到100之间，低参数值会让内核尽量少用交换，更高参数值会使内核更多的去使用交换空间
* 默认值为60（当剩余物理内存低于40%（40=100-60）时，开始使用交换空间）
* 对于大多数操作系统，设置为100可能会影响整体性能，而设置为更低值（甚至为0）则可能减少响应延迟

#### 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 |

#### 开启集群的时间同步NTP

#### 更新系统补丁

更新补丁前，请先测试新版本补丁对集群节点的兼容性

### HDFS优化（hdfs-site.xml）

#### 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。

#### 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。

#### 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。

### HBase优化

#### 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

#### 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。

#### 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

#### 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。

#### 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。

#### 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。

#### 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。

#### 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。

### 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。

#### JVM优化

涉及文件：hbase-env.sh

#### 并行GC

参数：-XX:+UseParallelGC

解释：开启并行GC

#### 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。

#### 禁用手动GC

参数：-XX:DisableExplicitGC

解释：防止开发人员手动调用GC

### Zookeeper优化

#### 优化Zookeeper会话超时时间

参数：zookeeper.session.timeout

文件：hbase-site.xml

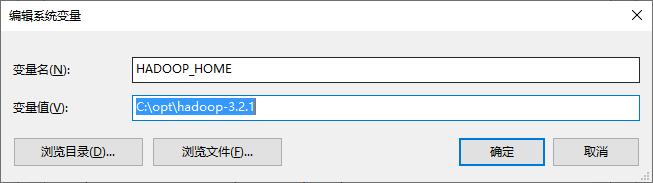
解释：In hbase-site.xml, set zookeeper.session.timeout to 30 seconds or less to bound failure detection (20-30 seconds is a good start).该值会直接关系到master发现服务器宕机的最大周期，默认值为30秒，如果该值过小，会在HBase在写入大量数据发生而GC时，导致RegionServer短暂的不可用，从而没有向ZK发送心跳包，最终导致认为从节点shutdown。一般20台左右的集群需要配置5台zookeeper。

## 常见问题处理

### Could not locate Hadoop executable: xxxx\bin\winutils.exe

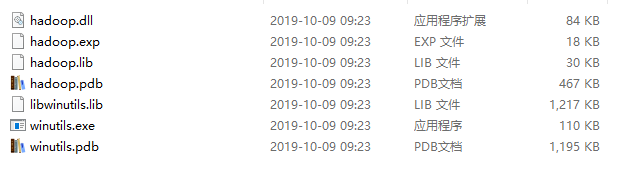
1. 配置以下环境变量

HADOOP\_HOME=<your local hadoop-ver folder>



在PATH环境变量中添加：%HADOOP\_HOME%\bin

1. 将资料中的winutils-master\hadoop-3.2.1\bin中的所有内容复制到HADOOP\_HOME\bin目录中



1. 重启IDEA，重新运行MapReduce程序

参考：<https://github.com/cdarlint/winutils>