**Hbase**

## 一、基础知识

**1 HBase出现背景**

（1）自 1970 年以来，关系数据库用于数据存储和维护有关问题的解决方案。大数据的出现后， 好多公司实现处理大数据并从中受益，并开始选择像 Hadoop 的解决方案。Hadoop 使用分 布式文件系统，用于存储大数据，并使用 MapReduce 来处理。Hadoop 擅长于存储各种格式 的庞大的数据，任意的格式甚至非结构化的处理。

（2）Hadoop 的限制

Hadoop 只能执行批量处理，并且只以顺序方式访问数据。这意味着必须搜索整个数据集， 即使是最简单的搜索工作。 当处理结果在另一个庞大的数据集，也是按顺序处理一个巨大的数据集。在这一点上，一个 新的解决方案，需要访问数据中的任何点（随机访问）单元。

（3）Hadoop 随机存取数据库

应用程序，如 HBase，Cassandra，CouchDB，Dynamo 和 MongoDB 都是一些存储大量数据和 以随机方式访问数据的数据库。

（4）总结：

1）海量数据量存储成为瓶颈，单台机器无法负载大量数据

2）单台机器 IO 读写请求成为海量数据存储时候高并发大规模请求的瓶颈

3）随着数据规模越来越大，大量业务场景开始考虑数据存储横向水平扩展，使得存储服 务可以增加/删除，而目前的关系型数据库更专注于一台机器

**2 HBase简介**

HBase 是 BigTable 的开源（源码使用 Java 编写）版本。是 Apache Hadoop 的数据库，是建 立在 HDFS 之上，被设计用来提供高可靠性、高性能、列存储、可伸缩、多版本的 NoSQL 的分布式数据存储系统，实现**对大型数据的实时、随机的读写访问**。

HBase 依赖于 HDFS 做底层的数据存储，BigTable依赖 Google GFS 做数据存储

HBase 依赖于 MapReduce 做数据计算，BigTabl依赖 Google MapReduce 做数据计算

HBase 依赖于 ZooKeeper 做服务协调，BigTable 依赖 Google Chubby 做服务协调

**NoSQL = NO SQL**

**NoSQL = Not Only SQL：会有一些把 NoSQL 数据的原生查询语句封装成 SQL，比如 HBase 就有 Phoenix 工具**

**存在亿万条记录的数据库，只有千万或者百万条记录使用RDBMS更加合适确保你的应用不需要使用RDBMS的高级特性（第二索引，事务机制，高级查询语言等）足够的硬件配置，即节点数，HDFS在少于5个节点时并不会表现得很好，HBase也存在相同情况。**

**3 HBase这个NoSQL 数据库的要点**

① 它介于 NoSQL 和 RDBMS 之间，仅能通过主键(rowkey)和主键的 range 来检索数据

② HBase 查询数据功能很简单，**不支持 join** 等复杂操作

③ **不支持复杂**的事务，只支持行级事务(可通过 hive 支持来实现多表 join 等复杂操作)。

④ HBase 中支持的数据类型：byte[]（底层所有数据的存储都是字节数组）

⑤ 主要用来存储结构化和半结构化的松散数据。

**4 结构化、半结构化和非结构化**

（1）结构化：数据结构字段含义确定，清晰，典型的如数据库中的表结构

（2）半结构化：具有一定结构，但语义不够确定，典型的如 HTML 网页，有些字段是确定的(title)， 有些不确定(table)

（3）非结构化：杂乱无章的数据，很难按照一个概念去进行抽取，无规律性.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。

**5 HBase 中的表特点**

1）、大：一个表可以有上十亿行，上百万列

2）、面向列：面向列(族)的存储和权限控制，列(簇)独立检索。

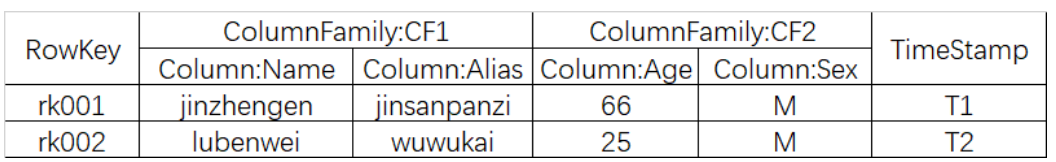
3）、稀疏：对于为空(null)的列，并不占用存储空间，因此，表可以设计的非常稀疏。

4）、无模式：每行都有一个可排序的主键和任意多的列，列可以根据需要动态的增加，同一 张表中不同的行可以有截然不同的列



## 二、核心知识

**1 HBase表结构逻辑视图**



**（1）Rowkey的概念**

Rowkey的概念和mysql中的主键是完全一样的，Hbase使用Rowkey来唯一的区分某一行的数据。

由于Hbase只支持3中查询方式：

1）、基于Rowkey的单行查询

2）、基于Rowkey的范围扫描

3）、全表扫描

因此，Rowkey对Hbase的性能影响非常大，Rowkey的设计就显得尤为的重要。设计的时候要兼顾基于Rowkey的单行查询也要键入Rowkey的范围扫描。具体Rowkey要如何设计后续会整理相关的文章做进一步的描述。这里大家只要有一个概念就是Rowkey的设计极为重要。

rowkey 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)，最好是 16。在 HBase 内部，rowkey 保存为字节数组。HBase 会对表中的数据按照 rowkey 排序 (字典顺序)

**（2）Column的概念**

列，可理解成MySQL列。

**（3）ColumnFamily的概念**

列族, HBase引入的概念。

Hbase通过列族划分数据的存储，列族下面可以包含任意多的列，实现灵活的数据存取。就像是家族的概念，我们知道一个家族是由于很多个的家庭组成的。列族也类似，列族是由一个一个的列组成（任意多）

Hbase表的创建的时候就必须指定列族。就像关系型数据库创建的时候必须指定具体的列是一样的。

Hbase的列族不是越多越好，官方推荐的是列族最好小于或者等于3。我们使用的场景一般是1个列族。

**（4）TimeStamp的概念**

TimeStamp对Hbase来说至关重要，因为它是实现Hbase多版本的关键。在Hbase中使用不同的timestame来标识相同rowkey行对应的不通版本的数据。**HBase 中通过 rowkey 和 columns 确定的为一个存储单元称为 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：

@保存数据的最后 n 个版本

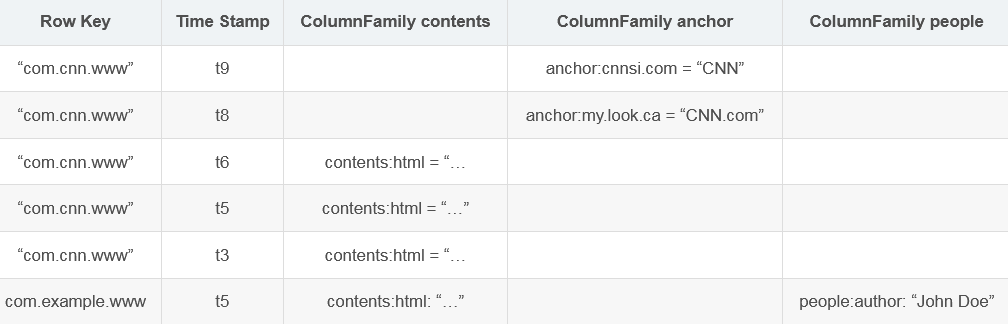
@保存最近一段时间内的版本（设置数据的生命周期 TTL）。

用户可以针对每个列簇进行设置。

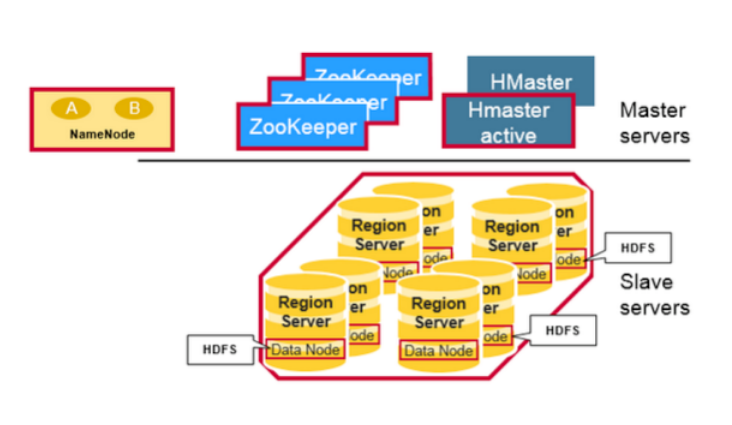
**（5）单元格（Cell）**

由{rowkey, column( = + ), 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。





**2 HBase体系架构**



**（1）Client**

包含访问HBase的接口并维护cache来加快HBase的访问

**（2）Zookeeper**

@保证任何时候， 集群中只有一个master.体现了服务高可用， 一旦挂了一个，将会启动一个。

@存保证集群中有且只有一个HMaster为Active

@存储hbase:meta，即所有Region的位置信息

@存储HBase中表格的元数据信息

@监控RegionServer状态，将RS的上下线情况汇报给HMaster

@ZooKeeper集群本身使用一致性协议(PAXOS协议)保证每个节点状态的一致性

**（3）HMaster,表的元数据信息都存在Zookeeper， HMaster相当于一个领导**

为RegionServer分配Region

新来的一个Region，我要将他放在哪个Region

负责RegionServer的负载均衡

当创建一个表，生成一个Region， 持续向表中插入数据，达到了阈值， 将分裂成2个Region, 数据再增加，将会分裂成3、4、…100个。不能让这100个Region在这一个HRegionServer, 而其他RegionServer中没有Region, 为了数据均衡， 将部分Region迁移到其他的HRegionServer.相当于工作的分配，一个员工的工作过多，将分配她的任务给其他人。

发现失效的RegionServer并重新分配其上的Region

管理用户对table的增删操作

1. **HRegionServer**

响应client的读写请求，进行I/O操作（直接绕过HMaster）

与HDFS交互，管理table数据

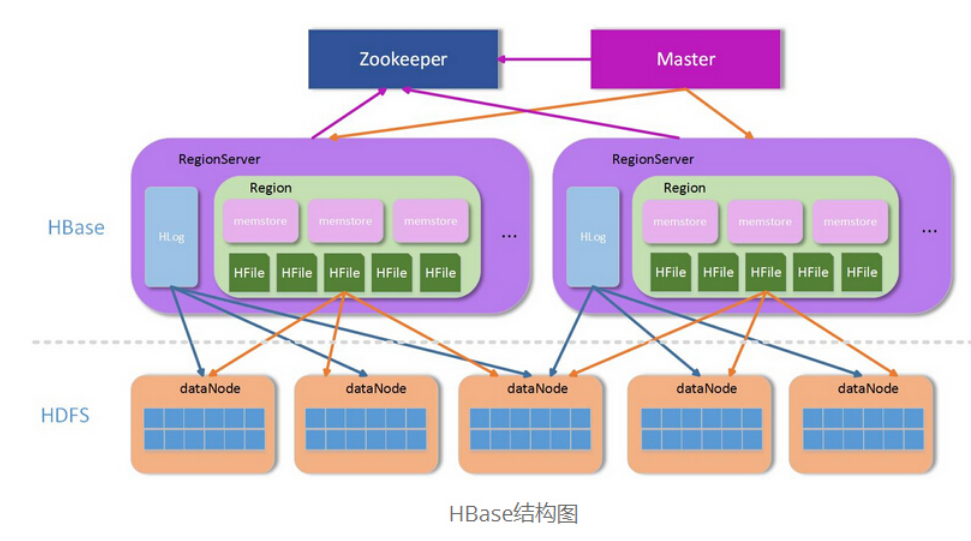
当Region的大小到达阀值时切分Region

维护Region,处理对这些Region的IO请求

负责切分在运行过程中变的过大的Region

当RegionServer中的Region过多，HMaster将会做一个负载均衡。





**3 HBase的数据模型**

**（1）Region**

当创建一个表，生成一个Region， 持续向表中插入数据，达到了阈值， 将分裂成2个Region, 数据再增加，将会分裂成3、4、…100个。

当表中的行不断增多，就会有越来越多的Region，在HMaster的调度写，使一个表被保存在多个RegionServer上。

**（2）Memstore与storefile**

一个Region由多个store组成，一个store对应一个CF(列族)

store包括位于内存中的memstore和位于磁盘中的storefile.写操作：先写入memstore,但memstore中的数据达到阈值， HRegionServer会启动flashcache进程系写入storefile, 每次写入形成一个单独的storefile。

当storefile文件增长到一定阈值后， 系统会进行合并（minor,majar compcation）, 在合并中进行版本合并和删除工作（majar）,形成更大的storefile.

当一个Region所有的storefile的大小和超过一个阈值后， 会把当前的Region分割成两个，并通过HMaster的调度， 分配到相应的HRegionServer中， 实现负载均衡。

客户端检索数据，现在memstore找， 找不到载找storefile.

1. **storefile 以HFile格式保存在HDFS上。**

**（4）Hlog也是存在HDFS上。**

**4 自动分区**

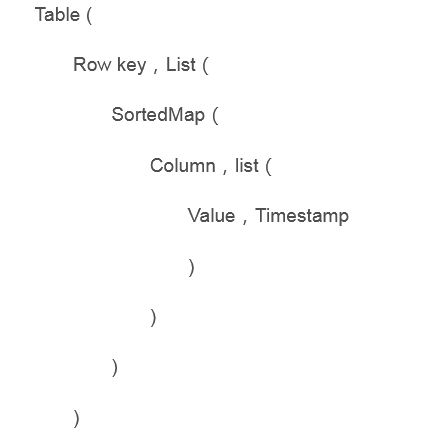
（1）HBase中扩展和负载均衡的基本单元为region，region的本质上是以行键排序的连续存储区间。若region太大，系统会把它们动态拆分，相反，就会把多个region合并以减少存储文件数量。

（2）每个region只能有一台region服务器（region server）加载，而一台region server可以同时加载多个region（一台机子一个region server）

（3）每台服务器能加载的region数量和每个region的最大存储大小取决于单台服务器的有效处理能力。每台服务器的最佳加载数量为10-1000，但每个region的最佳大小为1GB-2GBregion的拆分和服务相当于其他系统提供的自动分区。当一个服务器出现故障后，该服务器上的region可快速恢复，并获得细粒度的负载均衡，因为当服务于某个region的服务器当前负载过大、发生错误或者被停止使用导致不可用时，系统会将改region移到其他服务器上。region的拆分可以瞬间完成，因为拆分后的region读取的仍为原存储文件，直到合并把存储文件异步地写成独立文件

（4）注：HBase系统没有提供查询数据的特定域语言，数据存储不是以声明的方式完成的，是通过客户端API以纯粹的命令完成的

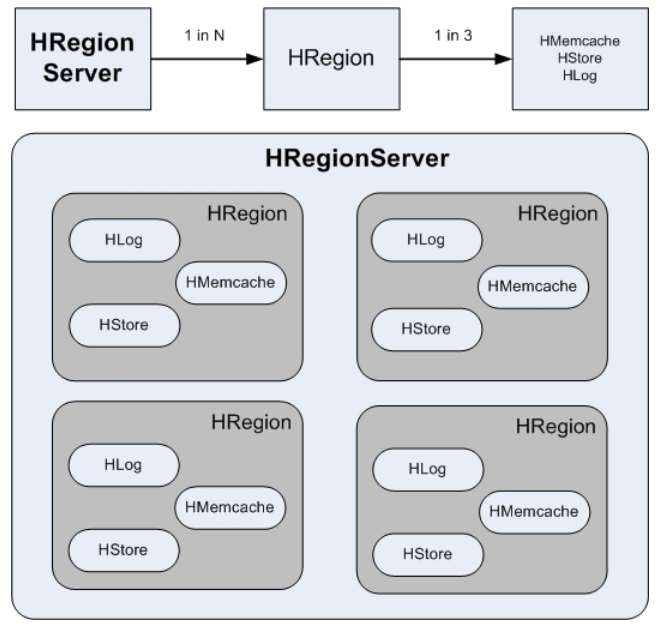
**5 存储结构**



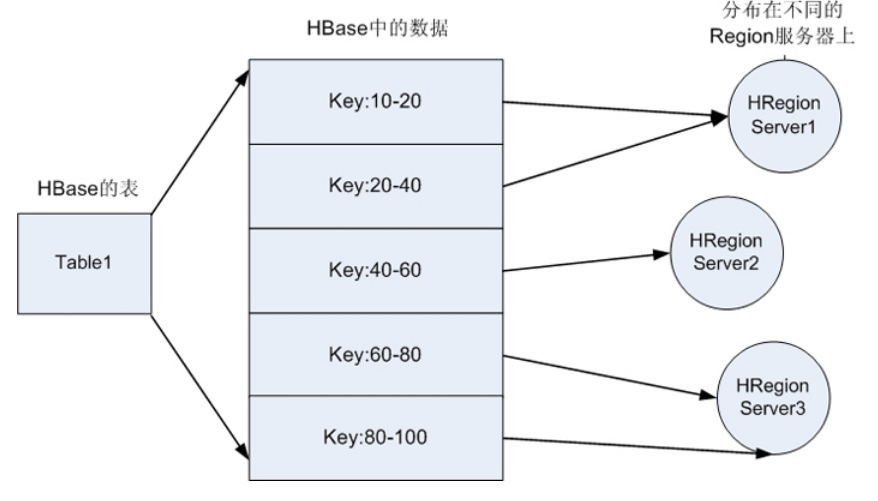
（1）在HBase中创建的一张表可以分布在多个Hregion，也就说一张表可以被拆分成多块，每一块称我们呼为一个Hregion。每个Hregion会保 存一个表里面某段连续的数据，用户创建的那个大表中的每个Hregion块是由Hregion服务器提供维护，访问Hregion块是要通过 Hregion服务器，而一个Hregion块对应一个Hregion服务器，一张完整的表可以保存在多个Hregion 上。HRegion Server 与Region的对应关系是一对多的关系。每一个HRegion在物理上会被分为三个部分：Hmemcache(缓存)、Hlog(日志)、HStore(持久层)。

上述这些关系在我脑海中的样子，如图所示：

（2）HRegionServer、HRegion、Hmemcache、Hlog、HStore之间的关系，如图所示：



（3）HBase表中的数据与HRegionServer的分布关系，如图所示：



（4）HBase读数据

HBase读取数据优先读取HMemcache中的内容，如果未取到再去读取Hstore中的数据，提高数据读取的性能。

**1、客户端通过 ZooKeeper 以及-ROOT-表和.META.表找到目标数据所在的 RegionServer(就是 数据所在的 Region 的主机地址)**

**2、联系 RegionServer 查询目标数据**

**3、RegionServer 定位到目标数据所在的 Region，发出查询请求**

**4、Region 先在 Memstore 中查找，命中则返回**

**5、如果在 Memstore 中找不到，则在 Storefile 中扫描 为了能快速的判断要查询的数据在不在这个 StoreFile 中，应用了 BloomFilter**

（5）HBase写数据

HBase写入数据会写到HMemcache和Hlog中，HMemcache建立缓存，Hlog同步Hmemcache和Hstore的事务日志，发起Flush Cache时，数据持久化到Hstore中，并清空HMemecache。

**1、Client 先根据 RowKey 找到对应的 Region 所在的 RegionServer**

**2、Client 向 RegionServer 提交写请求**

**3、RegionServer 找到目标 Region**

**4、Region 检查数据是否与 Schema 一致**

**5、如果客户端没有指定版本，则获取当前系统时间作为数据版本**

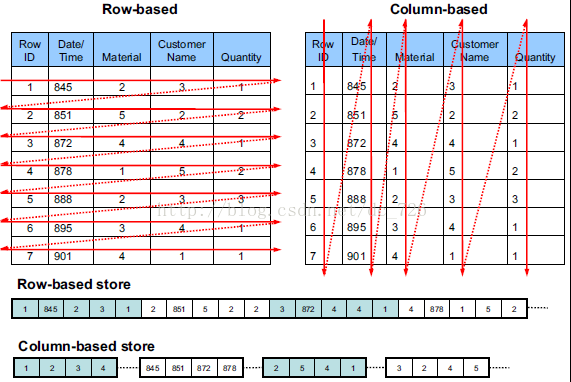
**6、将更新写入 WAL Log**

**7、将更新写入 Memstore**

**8、判断 Memstore 的是否需要 flush 为 StoreFile 文件。**

（6）客户端访问这些数据的时候通过Hmaster ，每个 Hregion 服务器都会和Hmaster 服务器保持一个长连接，Hmaster 是HBase分布式系统中的管理者，他的主要任务就是要告诉每个Hregion 服务器它要维护哪些Hregion。用户的这些都数据可以保存在Hadoop 分布式文件系统上。 如果主服务器Hmaster死机，那么整个系统都会无效。下面我会考虑如何解决Hmaster的SPFO的问题，这个问题有点类似Hadoop的SPFO 问题一样只有一个NameNode维护全局的DataNode，HDFS一旦死机全部挂了，也有人说采用Heartbeat来解决这个问题，但我总想找出 其他的解决方案，多点时间，总有办法的。

**假如系统中有一个User表，如果按照传统的RDBMS的话，User表中的列是固定的，比如schema 定义了name,age,sex等属性，User的属性是不能动态增加的。但是如果采用列存储系统，比如Hbase，那么我们可以定义User表，然后定义info 列族，User的数据可以分为：info:name = zhangsan,info:age=30,info:sex=male等，如果后来你又想增加另外的属性，这样很方便只需要info:newProperty就可以了。**



**6 HBase优化技巧**

**（1） 修改 linux 系统参数**

Linux系统最大可打开文件数一般默认的参数值是1024,如果你不进行修改并发量上来的时候会出现“Too Many Open Files”的错误，导致整个HBase不可运行，你可以用ulimit -n 命令进行修改，或者修改/etc/security/limits.conf 和/proc/sys/fs/file-max 的参数，具体如何修改可以去Google 关键字 “linux limits.conf ”

**（2） JVM 配置**

修改 hbase-env.sh 文件中的配置参数，根据你的机器硬件和当前操作系统的JVM(32/64位)配置适当的参数

HBASE\_HEAPSIZE 4000 HBase使用的 JVM 堆的大小

HBASE\_OPTS "‐server ‐XX:+UseConcMarkSweepGC"JVM GC 选项

HBASE\_MANAGES\_ZKfalse 是否使用Zookeeper进行分布式管理

**（3）HBase持久化**

重启操作系统后HBase中数据全无，你可以不做任何修改的情况下，创建一张表，写一条数据进行，然后将机器重启，重启后你再进入HBase的shell中使用 list 命令查看当前所存在的表，一个都没有了。是不是很杯具？没有关系你可以在hbase/conf/hbase-default.xml中设置hbase.rootdir的值，来设置文件的保存位置指定一个文件夹 ，例如：<value>file:///you/hbase-data/path</value>，你建立的HBase中的表和数据就直接写到了你的磁盘上，如图所示：同样你也可以指定你的分布式文件系统HDFS的路径例如: hdfs://NAMENODE\_SERVER:PORT/HBASE\_ROOTDIR，这样就写到了你的分布式文件系统上了。

**（4）配置HBase运行参数**

其次就需要对hbase/conf/hbase-default.xml 文件进行配置，以下是我认为比较重要的配置参数

hbase.client.write.buffer

描述：这个参数可以设置写入数据缓冲区的大小，当客户端和服务器端传输数据，服务器为了提高系统运行性能开辟一个写的缓冲区来处理它， 这个参数设置如果设置的大了，将会对系统的内存有一定的要求，直接影响系统的性能。

hbase.master.meta.thread.rescanfrequency

描述：多长时间 HMaster对系统表 root 和 meta 扫描一次，这个参数可以设置的长一些，降低系统的能耗。

hbase.regionserver.handler.count

描述：由于HBase/Hadoop的Server是采用Multiplexed, non-blocking I/O方式而设计的，所以它可以透过一个Thread来完成处理，但是由于处理Client端所呼叫的方法是Blocking I/O，所以它的设计会将Client所传递过来的物件先放置在Queue，并在启动Server时就先产生一堆Handler(Thread)，该Handler会透过Polling的方式来取得该物件并执行对应的方法，默认为25，根据实际场景可以设置大一些。

hbase.regionserver.thread.splitcompactcheckfrequency

描述：这个参数是表示多久去RegionServer服务器运行一次split/compaction的时间间隔，当然split之前会先进行一个compact操作.这个compact操作可能是minor compact也可能是major compact.compact后,会从所有的Store下的所有StoreFile文件最大的那个取midkey.这个midkey可能并不处于全部数据的mid中.一个row-key的下面的数据可能会跨不同的HRegion。

hbase.hregion.max.filesize

描述：HRegion中的HStoreFile最大值，任何表中的列族一旦超过这个大小将会被切分，而HStroeFile的默认大小是256M。

hfile.block.cache.size

描述：指定 HFile/StoreFile 缓存在JVM堆中分配的百分比，默认值是0.2，意思就是20%，而如果你设置成0，就表示对该选项屏蔽。

hbase.zookeeper.property.maxClientCnxns

描述： 这项配置的选项就是从zookeeper中来的，表示ZooKeeper客户端同时访问的并发连接数，ZooKeeper对于HBase来说就是一个入口这个参数的值可以适当放大些。

hbase.regionserver.global.memstore.upperLimit

描述：在Region Server中所有memstores占用堆的大小参数配置，默认值是0.4，表示40%，如果设置为0，就是对选项进行屏蔽。

hbase.hregion.memstore.flush.size

描述：Memstore中缓存的内容超过配置的范围后将会写到磁盘上，例如：删除操作是先写入MemStore里做个标记，指示那个value, column 或 family等下是要删除的，HBase会定期对存储文件做一个major compaction，在那时HBase会把MemStore刷入一个新的HFile存储文件中。如果在一定时间范围内没有做major compaction，而Memstore中超出的范围就写入磁盘上了。

**（5）HBase中log4j的日志**

HBase中日志输出等级默认状态下是把debug、 info 级别的日志打开的，可以根据自己的需要调整log级别，HBase的log4j日志配置文件在 hbase\conf\log4j.properties 目录下。

## 三、HBase的基本操作

**1.启动HBase Shell**

转到HBase解压包的“bin”目录下，使用命令“./hbase shell”或“sh hbase shell”命令即可启动HBase Shell。启动之后的信息如下：

HBase Shell; enter 'help<RETURN>' for list of supported commands.

Type "exit<RETURN>" to leave the HBase Shell

Version 1.0.1, r66a93c09df3b12ff7b86c39bc8475c60e15af82d, Fri Apr 17 22:14:06 PDT 2015

hbase(main):001:0>

为了验证HBase是否安装成功，我们可以输入“list”查看HBase中所有的表。该命令的执行情况如下：

hbase(main):001:0> list

TABLE

0 row(s) in 0.2390 seconds

=> []

由于是首次登录，因此HBase中包含了0个表。

**2.创建表**

HBase使用表作为顶级结构来存储数据，写数据到HBase，也就是写数据到表。下面创建一个有一个列族(“cf”)的表“mytable”：

hbase(main):002:0> create 'mytable', 'cf'

0 row(s) in 0.3250 seconds

=> Hbase::Table - mytable

hbase(main):003:0> list

TABLE

mytable

1 row(s) in 0.0090 seconds

=> ["mytable"]

使用“list”命令，我们可以看到，表创建成功。

**3.写数据**

表创建好之后，就需要写入一些数据。例如，我们想往表里写入“hello hbase”，那么我们就在“mytable”表的“first”行中的“cf:info”列对应的数据单元中插入“hello hbase”。命令执行如下：

hbase(main):004:0> put 'mytable', 'first', 'cf:info', 'hello hbase'

0 row(s) in 0.0970 seconds

我们还可以继续多增加几个值，如下：

hbase(main):005:0> put 'mytable', 'second', 'cf:name', 'zhou'

0 row(s) in 0.0180 seconds

hbase(main):006:0> put 'mytable', 'third', 'cf:nation', 'China'

0 row(s) in 0.0080 seconds

现在表里面有3行和3个数据单元。在使用列的时候，大家并没有提前定义这些列，也没有定义往每个列中存储的数据类型。因此，HBase是一种无模式的数据库。

**4.读数据**

HBase有两种方式读取数据：get和scan。get方式可以读取一行的数据信息，scan方式可以读取表中所有行的数据信息。

get命令执行如下：

hbase(main):008:0> get 'mytable', 'first'

COLUMN CELL

cf:info timestamp=1435548279711, value=hello hbase

1 row(s) in 0.0770 seconds

该命令输出了该行所有的数据单元。

scan命令执行如下：

hbase(main):009:0> scan 'mytable'

ROW COLUMN+CELL

first column=cf:info, timestamp=1435548279711, value=hello hbase

second column=cf:name, timestamp=1435548751549, value=zhou

third column=cf:nation, timestamp=1435548760826, value=China

3 row(s) in 0.0320 seconds

该命令返回了所有数据。HBase返回行的顺序，是按照行的名字排序的。HBase称之为行键(rowkey)。

**5.删除表**

要想删除一张表，需要先将其“disable”掉，然后再“drop”掉。命令执行如下：

hbase(main):010:0> disable 'mytable'

0 row(s) in 1.2380 seconds

hbase(main):011:0> drop 'mytable'

0 row(s) in 0.1770 seconds

hbase(main):012:0> list

TABLE

0 row(s) in 0.0070 seconds

=> []

## 三、Java连接HBase

hbase-0.20.6.jar

hadoop-core-0.20.1.jar

commons-logging-1.1.1.jar

zookeeper-3.3.0.jar log4j-1.2.91.jar

**1. 加载配置**

Configuration conf = new Configuration();

// conf.addResource("hbase-site-cluster.xml");//指定文件加载

conf = HBaseConfiguration.create(conf);

HBaseAdmin admin = new HBaseAdmin(conf);//HBaseAdmin负责跟表相关的操作如create,drop等

HTable table = new HTable(conf, Bytes.toBytes("blog"));//HTabel负责跟记录相关的操作如增删改查等

**2. 创建表**

if (admin.tableExists("blog")) {// 如果存在要创建的表，先删除，再创建

admin.disableTable("blog");

admin.deleteTable("blog");

System.out.println("blog" + " is exist,detele....");

}

HTableDescriptor desc = new HTableDescriptor("blog");

desc.addFamily(new HColumnDescriptor("article"));//增加article列族

desc.addFamily(new HColumnDescriptor("author"));//增加author列族

admin.createTable(desc);

**3. 增加记录(更新记录相同操作)**

Put put = new Put(Bytes.toBytes("1"));///增加一条行键是“1”的数据

put.add(Bytes.toBytes("article"), Bytes.toBytes("title"), Bytes.toBytes("Head First HBase"));//在“article”列族中往单元“title”存入“Head First HBase”

put.add(Bytes.toBytes("article"), Bytes.toBytes("content"), Bytes.toBytes("HBase is the Hadoop database. Use it when you need random, realtime read/write access to your Big Data."));//在“article”列族往单元“//在“article”列族中往单元“content”存入“HBase is the...”

put.add(Bytes.toBytes("article"), Bytes.toBytes("tags"), Bytes.toBytes("Hadoop,HBase,NoSQL"));

put.add(Bytes.toBytes("author"), Bytes.toBytes("name"), Bytes.toBytes("hujinjun"));

put.add(Bytes.toBytes("author"), Bytes.toBytes("nickname"), Bytes.toBytes("一叶渡江"));

table.put(put);

**4. 根据RowKey查询**

Result result = table.get(Bytes.toBytes("RowKey"));

for (KeyValue kv : result.list()) {

System.out.println("family:" + Bytes.toString(kv.getFamily()));

System.out.println("qualifier:" + Bytes.toString(kv.getQualifier()));

System.out.println("value:" + Bytes.toString(kv.getValue()));

System.out.println("Timestamp:" + kv.getTimestamp());

}

你还可以在Get的实例中设定限制条件来获取自己想要的数据量。

为了返回指定的列，可以执行命令addColumn()。

Get get = new Get(Bytes.toBytes("RowKey"));

get.addColumn(

Bytes.toBytes("article"),

Bytes.toBytes("author")

);

Result result =table.get.(get);

对于列族可执行命令addFamily()。检索特定值，如下所示

Get get = new Get(Bytes.toBytes("RowKey"));

get.addFamily(Bytes.toBytes("article"));

byte[] b = r.getValue(

Bytes.toBytes("article"),

Bytes.toBytes("title")

);

String title = Bytes.toString(b);//"Head First HBase"

**5. 遍历查询与迭代**

Scan scan = new Scan();

ResultScanner rs = null;

try

{

rs = table.getScanner(scan);

for (Result r : rs) {

for (KeyValue kv : r.list()) {

System.out.println("family:" + Bytes.toString(kv.getFamily()));

System.out.println("qualifier:" + Bytes.toString(kv.getQualifier()));

System.out.println("value:" + Bytes.toString(kv.getValue()));

}

}

} finally

{

rs.close();

}

**6. 删除记录**

删除指定的列

Delete delete = new Delete(Bytes.toBytes("行键"));

delete.deleteColumns(

Bytes.toBytes("列族"),

Bytes.toBytes("列名"));

table.delete(delete);

deleteColumns()方法从行中删除一个单元，这个和deleteColumn()方法不同（尾部少了s）。deleteColumn()方法删除单元的内容。

删除一条记录

Delete deleteAll = new Delete(Bytes.toBytes("行键"));

table.delete(deleteAll);

删除一张表

admin.disableTable("表名");

admin.delete("表名");

### 创建一张表的实例开销很大使用连接池比直接创建表句柄好 ###

推荐的创建HConnection的方法

//Configuration conf = HBaseConfiguration.create();

//conf.set("hbase.zookeeper.quorum", QUORUM);

//conf.set("hbase.zookeeper.property.clientPort", CLIENTPORT);

Configuration conf = new Configuration();

conf.addResource("hbase-site-cluster.xml");//指定文件加载

HConnection connection = HConnectionManager.createConnection(conf);

HTableInterface table = connection.getTable("tablename");

try {

// Use the table as needed, for a single operation and a single thread

} finally {

table.close();

connection.close();

}

**7 MapReduce从HDFS读取数据存储到HBase中**

**（1）现有HDFS中有一个student.txt文件，格式如下**

95002,刘晨,女,19,IS

95017,王风娟,女,18,IS

95018,王一,女,19,IS

95013,冯伟,男,21,CS

95014,王小丽,女,19,CS

95019,邢小丽,女,19,IS

95020,赵钱,男,21,IS

95003,王敏,女,22,MA

95004,张立,男,19,IS

95012,孙花,女,20,CS

95010,孔小涛,男,19,CS

95005,刘刚,男,18,MA

95006,孙庆,男,23,CS

95007,易思玲,女,19,MA

95008,李娜,女,18,CS

95021,周二,男,17,MA

95022,郑明,男,20,MA

95001,李勇,男,20,CS

95011,包小柏,男,18,MA

95009,梦圆圆,女,18,MA

95015,王君,男,18,MA

**（2）将HDFS上的这个文件里面的数据写入到HBase数据块中**

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.client.Put;

import org.apache.hadoop.hbase.mapreduce.TableMapReduceUtil;

import org.apache.hadoop.hbase.mapreduce.TableReducer;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.NullWritable;

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.output.FileOutputFormat;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

public class ReadHDFSDataToHbaseMR extends Configured implements Tool{

public static void main(String[] args) throws Exception {

int run = ToolRunner.run(new ReadHDFSDataToHbaseMR(), args);

System.exit(run);

}

@Override

public int run(String[] arg0) throws Exception {

Configuration conf = HBaseConfiguration.create();

conf.set("fs.defaultFS", "hdfs://myha01/");

conf.set("hbase.zookeeper.quorum", "hadoop1:2181,hadoop2:2181,hadoop3:2181");

System.setProperty("HADOOP\_USER\_NAME", "hadoop");

FileSystem fs = FileSystem.get(conf);

// conf.addResource("config/core-site.xml");

// conf.addResource("config/hdfs-site.xml");

Job job = Job.getInstance(conf);

job.setJarByClass(ReadHDFSDataToHbaseMR.class);

job.setMapperClass(HDFSToHbaseMapper.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(NullWritable.class);

TableMapReduceUtil.initTableReducerJob("student", HDFSToHbaseReducer.class, job,null,null,null,null,false);

job.setOutputKeyClass(NullWritable.class);

job.setOutputValueClass(Put.class);

Path inputPath = new Path("/student/input/");

Path outputPath = new Path("/student/output/");

if(fs.exists(outputPath)) {

fs.delete(outputPath,true);

}

FileInputFormat.addInputPath(job, inputPath);

FileOutputFormat.setOutputPath(job, outputPath);

boolean isDone = job.waitForCompletion(true);

return isDone ? 0 : 1;

}

public static class HDFSToHbaseMapper extends Mapper<LongWritable, Text, Text, NullWritable>{

@Override

protected void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException {

context.write(value, NullWritable.get());

}

}

/\*\*

\* 95015,王君,男,18,MA

\* \*/

public static class HDFSToHbaseReducer extends TableReducer<Text, NullWritable, NullWritable>{

@Override

protected void reduce(Text key, Iterable<NullWritable> values,Context context)

throws IOException, InterruptedException {

String[] split = key.toString().split(",");

Put put = new Put(split[0].getBytes());

put.addColumn("info".getBytes(), "name".getBytes(), split[1].getBytes());

put.addColumn("info".getBytes(), "sex".getBytes(), split[2].getBytes());

put.addColumn("info".getBytes(), "age".getBytes(), split[3].getBytes());

put.addColumn("info".getBytes(), "department".getBytes(), split[4].getBytes());

context.write(NullWritable.get(), put);

}

}

}

**（3）MapReduce从HBase读取数据计算平均年龄并存储到HDFS中**

import java.io.IOException;

import java.util.List;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.hbase.Cell;

import org.apache.hadoop.hbase.CellUtil;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.client.Result;

import org.apache.hadoop.hbase.client.Scan;

import org.apache.hadoop.hbase.io.ImmutableBytesWritable;

import org.apache.hadoop.hbase.mapreduce.TableMapReduceUtil;

import org.apache.hadoop.hbase.mapreduce.TableMapper;

import org.apache.hadoop.hbase.util.Bytes;

import org.apache.hadoop.io.DoubleWritable;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

public class ReadHbaseDataToHDFS extends Configured implements Tool{

public static void main(String[] args) throws Exception {

int run = ToolRunner.run(new ReadHbaseDataToHDFS(), args);

System.exit(run);

}

@Override

public int run(String[] arg0) throws Exception {

Configuration conf = HBaseConfiguration.create();

conf.set("fs.defaultFS", "hdfs://myha01/");

conf.set("hbase.zookeeper.quorum", "hadoop1:2181,hadoop2:2181,hadoop3:2181");

System.setProperty("HADOOP\_USER\_NAME", "hadoop");

FileSystem fs = FileSystem.get(conf);

// conf.addResource("config/core-site.xml");

// conf.addResource("config/hdfs-site.xml");

Job job = Job.getInstance(conf);

job.setJarByClass(ReadHbaseDataToHDFS.class);

// 取对业务有用的数据 info,age

Scan scan = new Scan();

scan.addColumn("info".getBytes(), "age".getBytes());

TableMapReduceUtil.initTableMapperJob(

"student".getBytes(), // 指定表名

scan, // 指定扫描数据的条件

HbaseToHDFSMapper.class, // 指定mapper class

Text.class, // outputKeyClass mapper阶段的输出的key的类型

IntWritable.class, // outputValueClass mapper阶段的输出的value的类型

job, // job对象

false

);

job.setReducerClass(HbaseToHDFSReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(DoubleWritable.class);

Path outputPath = new Path("/student/avg/");

if(fs.exists(outputPath)) {

fs.delete(outputPath,true);

}

FileOutputFormat.setOutputPath(job, outputPath);

boolean isDone = job.waitForCompletion(true);

return isDone ? 0 : 1;

}

public static class HbaseToHDFSMapper extends TableMapper<Text, IntWritable>{

Text outKey = new Text("age");

IntWritable outValue = new IntWritable();

// key是hbase中的行键

// value是hbase中的所行键的所有数据

@Override

protected void map(ImmutableBytesWritable key, Result value,Context context)

throws IOException, InterruptedException {

boolean isContainsColumn = value.containsColumn("info".getBytes(), "age".getBytes());

if(isContainsColumn) {

List<Cell> listCells = value.getColumnCells("info".getBytes(), "age".getBytes());

System.out.println("listCells:\t"+listCells);

Cell cell = listCells.get(0);

System.out.println("cells:\t"+cell);

byte[] cloneValue = CellUtil.cloneValue(cell);

String ageValue = Bytes.toString(cloneValue);

outValue.set(Integer.parseInt(ageValue));

context.write(outKey,outValue);

}

}

}

public static class HbaseToHDFSReducer extends Reducer<Text, IntWritable, Text, DoubleWritable>{

DoubleWritable outValue = new DoubleWritable();

@Override

protected void reduce(Text key, Iterable<IntWritable> values,Context context)

throws IOException, InterruptedException {

int count = 0;

int sum = 0;

for(IntWritable value : values) {

count++;

sum += value.get();

}

double avgAge = sum \* 1.0 / count;

outValue.set(avgAge);

context.write(key, outValue);

}

}

}

## 四、过滤器

**1 基础API中的查询操作在面对大量数据的时候是非常苍白的，这里Hbase提供了高级的查询方法：Filter。**

Filter可以根据簇、列、版本等更多的条件来对数据进行过滤，基于Hbase本身提供的三维有序（主键有序、列有序、版本有序），这些Filter可以高效的完成查询过滤的任务。

带有Filter条件的RPC查询请求会把Filter分发到各个RegionServer，是一个服务器端（Server-side）的过滤器，这样也可以降低网络传输的压力。

要完成一个过滤的操作，至少需要两个参数。一个是抽象的操作符，Hbase提供了枚举类型的变量来表示这些抽象的操作符：LESS/LESS\_OR\_EQUAL/EQUAL/NOT\_EUQAL等；另外一个就是具体的比较器（Comparator），代表具体的比较逻辑，如果可以提高字节级的比较、字符串级的比较等。有了这两个参数，我们就可以清晰的定义筛选的条件，过滤数据。

**2 抽象操作符（比较运算符）**

LESS <

LESS\_OR\_EQUAL <=

EQUAL =

NOT\_EQUAL <>

GREATER\_OR\_EQUAL >=

GREATER >

NO\_OP 排除所有

**3 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

**4 行键过滤器 RowFilter**

public class HbaseFilterTest {

private static final String ZK\_CONNECT\_KEY = "hbase.zookeeper.quorum";

private static final String ZK\_CONNECT\_VALUE = "hadoop1:2181,hadoop2:2181,hadoop3:2181";

private static Connection conn = null;

private static Admin admin = null;

public static void main(String[] args) throws Exception {

Configuration conf = HBaseConfiguration.create();

conf.set(ZK\_CONNECT\_KEY, ZK\_CONNECT\_VALUE);

conn = ConnectionFactory.createConnection(conf);

admin = conn.getAdmin();

Table table = conn.getTable(TableName.valueOf("student"));

Scan scan = new Scan();

Filter rowFilter = new RowFilter(CompareOp.GREATER, new BinaryComparator("95007".getBytes()));

scan.setFilter(rowFilter);

ResultScanner resultScanner = table.getScanner(scan);

for(Result result : resultScanner) {

List<Cell> cells = result.listCells();

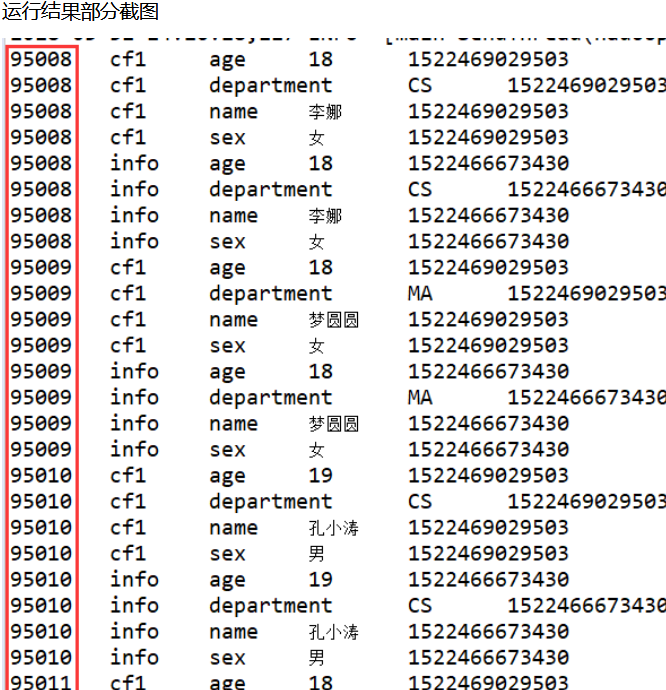
for(Cell cell : cells) {

System.out.println(cell);

}

}

}



**5 列簇过滤器 FamilyFilter**

Scan scan = new Scan();

Filter familyFilter = new FamilyFilter(CompareOp.EQUAL, new BinaryComparator("info".getBytes()));

scan.setFilter(familyFilter);

ResultScanner resultScanner = table.getScanner(scan);

**6 列过滤器 QualifierFilter**

Scan scan = new Scan();

18

19 Filter qualifierFilter = new QualifierFilter(CompareOp.EQUAL, new BinaryComparator("name".getBytes()));

20 scan.setFilter(qualifierFilter);

21 ResultScanner resultScanner = table.getScanner(scan);

**7 值过滤器 ValueFilter**

Filter valueFilter = new ValueFilter(CompareOp.EQUAL, new SubstringComparator("男"));

scan.setFilter(valueFilter);

**8 时间戳过滤器 TimestampsFilter**

List<Long> list = new ArrayList<>();

list.add(1522469029503l);

TimestampsFilter timestampsFilter = new TimestampsFilter(list);

scan.setFilter(timestampsFilter);

**9 单列值过滤器 SingleColumnValueFilter ----会返回满足条件的整行**

SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter(

"info".getBytes(), //列簇

"name".getBytes(), //列

CompareOp.EQUAL,

new SubstringComparator("刘晨"));

//如果不设置为 true，则那些不包含指定 column 的行也会返回

singleColumnValueFilter.setFilterIfMissing(true);

scan.setFilter(singleColumnValueFilter);

**10 单列值排除器 SingleColumnValueExcludeFilter**

SingleColumnValueExcludeFilter singleColumnValueExcludeFilter = new SingleColumnValueExcludeFilter(

"info".getBytes(),

"name".getBytes(),

CompareOp.EQUAL,

new SubstringComparator("刘晨"));

singleColumnValueExcludeFilter.setFilterIfMissing(true);

scan.setFilter(singleColumnValueExcludeFilter);

**11 前缀过滤器 PrefixFilter----针对行键**

PrefixFilter prefixFilter = new PrefixFilter("9501".getBytes());

scan.setFilter(prefixFilter);

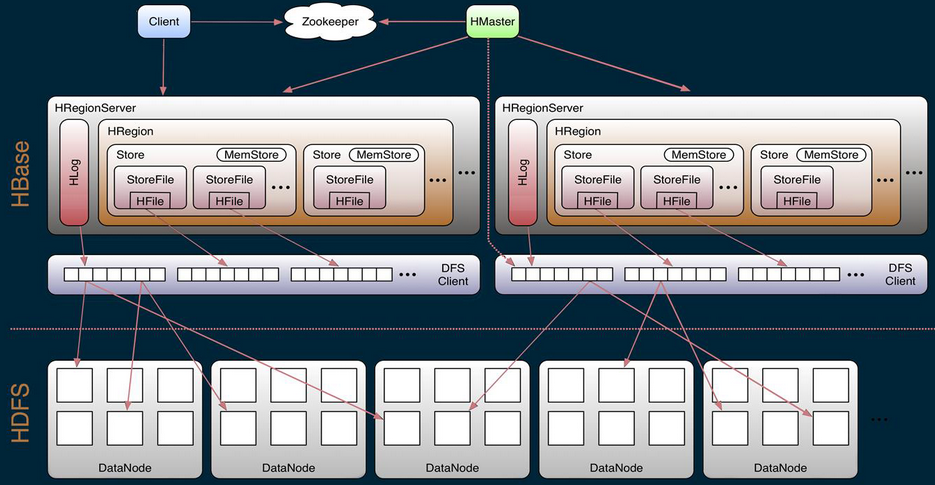
**12 列前缀过滤器 ColumnPrefixFilter**

ColumnPrefixFilter columnPrefixFilter = new ColumnPrefixFilter("name".getBytes());

scan.setFilter(columnPrefixFilter);

## 五、HBase 原理

**1 系统架构**



**（1）Client**

1）、HBase 有两张特殊表：

.META.：记录了用户所有表拆分出来的的 Region 映射信息，.META.可以有多个 Regoin

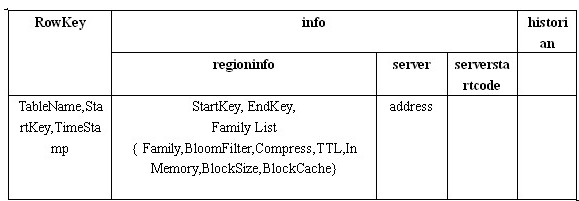
-ROOT-：记录了.META.表的 Region 信息，-ROOT-只有一个 Region，无论如何不会分裂

1. 、Client 访问用户数据前需要首先访问 ZooKeeper，找到-ROOT-表的 Region 所在的位置，然 后访问-ROOT-表，接着访问.META.表，最后才能找到用户数据的位置去访问，中间需要多次 网络操作，不过 client 端会做 cache 缓存。

**HBase -ROOT-和.META.表结构(region定位原理)**

在HBase中，大部分的操作都是在RegionServer完成的，Client端想要插入，删除，查询数据都需要先找到相应的RegionServer。什么叫相应的RegionServer？就是管理你要操作的那个Region的RegionServer。Client本身并不知道哪个RegionServer管理哪个Region，那么它是如何找到相应的RegionServer的？本文就是在研究源码的基础上揭秘这个过程。

在前面的文章“HBase存储架构”中我们已经讨论了HBase基本的存储架构。在此基础上我们引入两个特殊的概念：-ROOT-和.META.。这是什么？它们是HBase的两张内置表，从存储结构和操作方法的角度来说，它们和其他HBase的表没有任何区别，你可以认为这就是两张普通的表，对于普通表的操作对它们都适用。它们与众不同的地方是HBase用它们来存贮一个重要的系统信息——Region的分布情况以及每个Region的详细信息。好了，既然我们前面说到-ROOT-和.META.可以被看作是两张普通的表，那么它们和其他表一样就应该有自己的表结构。没错，它们有自己的表结构，并且这两张表的表结构是相同的，在分析源码之后我将这个表结构大致的画了出来：



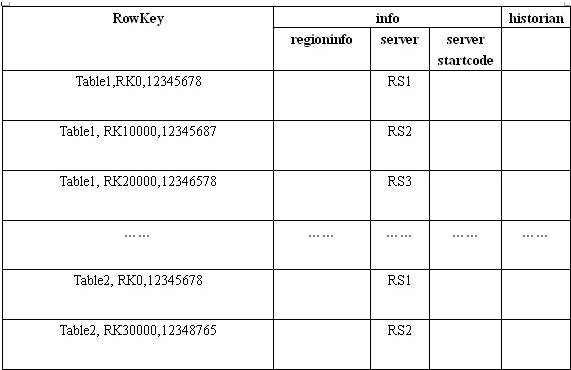
我们来仔细分析一下这个结构，每条Row记录了一个Region的信息。

首先是RowKey，RowKey由三部分组成：TableName, StartKey 和 TimeStamp。RowKey存储的内容我们又称之为Region的Name。哦，还记得吗？我们在前面的文章中提到的，用来存放Region的文件夹的名字是RegionName的Hash值，因为RegionName可能包含某些非法字符。现在你应该知道为什么RegionName会包含非法字符了吧，因为StartKey是被允许包含任何值的。将组成RowKey的三个部分用逗号连接就构成了整个RowKey，这里TimeStamp使用十进制的数字字符串来表示的。这里有一个RowKey的例子：

Table1,RK10000,12345678

然后是表中最主要的Family：info，info里面包含三个Column：regioninfo, server, serverstartcode。其中regioninfo就是Region的详细信息，包括StartKey, EndKey 以及每个Family的信息等等。server存储的就是管理这个Region的RegionServer的地址。

所以当Region被拆分、合并或者重新分配的时候，都需要来修改这张表的内容。到目前为止我们已经学习了必须的背景知识，下面我们要正式开始介绍Client端寻找RegionServer的整个过程。我打算用一个假想的例子来学习这个过程，因此我先构建了假想的-ROOT-表和.META.表。我们先来看.META.表，假设HBase中只有两张用户表：Table1和Table2，Table1非常大，被划分成了很多Region，因此在.META.表中有很多条Row用来记录这些Region。而Table2很小，只是被划分成了两个Region，因此在.META.中只有两条Row用来记录。这个表的内容看上去是这个样子的：



现在假设我们要从Table2里面插寻一条RowKey是RK10000的数据。那么我们应该遵循以下步骤：

1. 从.META.表里面查询哪个Region包含这条数据。

2. 获取管理这个Region的RegionServer地址。

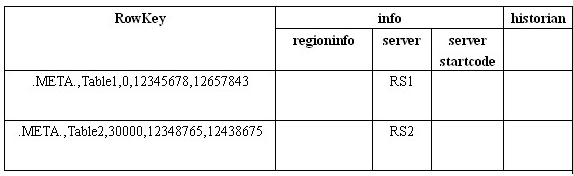
3. 连接这个RegionServer, 查到这条数据。

好，我们先来第一步。问题是.META.也是一张普通的表，我们需要先知道哪个RegionServer管理了.META.表，怎么办？有一个方法，我们把管理.META.表的RegionServer的地址放到ZooKeeper上面不久行了，这样大家都知道了谁在管理.META.。

貌似问题解决了，但对于这个例子我们遇到了一个新问题。因为Table1实在太大了，它的Region实在太多了，.META.为了存储这些Region信息，花费了大量的空间，自己也需要划分成多个Region。这就意味着可能有多个RegionServer在管理.META.。怎么办？在ZooKeeper里面存储所有管理.META.的RegionServer地址让Client自己去遍历？HBase并不是这么做的。

**HBase的做法是用另外一个表来记录.META.的Region信息，就和.META.记录用户表的Region信息一模一样。这个表就是-ROOT-表。这也解释了为什么-ROOT-和.META.拥有相同的表结构，因为他们的原理是一模一样的。**

假设.META.表被分成了两个Region，那么-ROOT-的内容看上去大概是这个样子的：



这么一来Client端就需要先去访问-ROOT-表。所以需要知道管理-ROOT-表的RegionServer的地址。这个地址被存在ZooKeeper中。默认的路径是：

/hbase/root-region-server

等等，如果-ROOT-表太大了，要被分成多个Region怎么办？嘿嘿，HBase认为-ROOT-表不会大到那个程度，因此-ROOT-只会有一个Region，这个Region的信息也是被存在HBase内部的。 现在让我们从头来过，我们要查询Table2中RowKey是RK10000的数据。整个路由过程的主要代码在org.apache.hadoop.hbase.client.HConnectionManager.TableServers中：

private HRegionLocation locateRegion(final byte[] tableName,

final byte[] row, boolean useCache) throws IOException {

if (tableName == null || tableName.length == 0) {

throw new IllegalArgumentException("table name cannot be null or zero length");

}

if (Bytes.equals(tableName, ROOT\_TABLE\_NAME)) {

synchronized (rootRegionLock) {

// This block guards against two threads trying to find the root

// region at the same time. One will go do the find while the

// second waits. The second thread will not do find.

if (!useCache || rootRegionLocation == null) {

this.rootRegionLocation = locateRootRegion();

}

return this.rootRegionLocation;

}

} else if (Bytes.equals(tableName, META\_TABLE\_NAME)) {

return locateRegionInMeta(ROOT\_TABLE\_NAME, tableName, row, useCache, metaRegionLock);

} else {

// Region not in the cache – have to go to the meta RS

return locateRegionInMeta(META\_TABLE\_NAME, tableName, row, useCache, userRegionLock);

}

}

这是一个递归调用的过程：

获取Table2，RowKey为RK10000的RegionServer => 获取.META.，RowKey为Table2,RK10000, 99999999999999的RegionServer => 获取-ROOT-，RowKey为.META.,Table2,RK10000,99999999999999,99999999999999的RegionServer => 获取-ROOT-的RegionServer => 从ZooKeeper得到-ROOT-的RegionServer => 从-ROOT-表中查到RowKey最接近（小于） .META.,Table2,RK10000,99999999999999,99999999999999的一条Row，并得到.META.的RegionServer => 从.META.表中查到RowKey最接近（小于）Table2,RK10000, 99999999999999的一条Row，并得到Table2的RegionServer => 从Table2中查到RK10000的Row

到此为止Client完成了路由RegionServer的整个过程，在整个过程中使用了添加“99999999999999”后缀并查找最接近（小于）RowKey的方法。对于这个方法大家可以仔细揣摩一下，并不是很难理解。

**（2）ZooKeeper**

1）、ZooKeeper 为 HBase 提供 Failover 机制，选举 Master，避免单点 Master 单点故障问题

2）、存储所有 Region 的寻址入口：-ROOT-表在哪台服务器上。-ROOT-这张表的位置信息

3）、实时监控 RegionServer 的状态，将 RegionServer 的上线和下线信息实时通知给 Master

4）、存储 HBase 的 Schema，包括有哪些 Table，每个 Table 有哪些 Column Family

**（3）Master**

1）、为 RegionServer 分配 Region

2）、负责 RegionServer 的负载均衡

3）、发现失效的 RegionServer 并重新分配其上的 Region

4）、HDFS 上的垃圾文件（HBase）回收

5）、处理 Schema 更新请求（表的创建，删除，修改，列簇的增加等等）

**（4）RegionServer**

1）、RegionServer 维护 Master 分配给它的 Region，处理对这些 Region 的 IO 请求

2）、RegionServer 负责 Split 在运行过程中变得过大的 Region，负责 Compact 操作

可以看到，client 访问 HBase 上数据的过程并不需要 master 参与（寻址访问 zookeeper 和 RegioneServer，数据读写访问 RegioneServer），Master 仅仅维护者 Table 和 Region 的元数据信息，负载很低。

.META. 存的是所有的 Region 的位置信息，那么 RegioneServer 当中 Region 在进行分裂之后 的新产生的 Region，是由 Master 来决定发到哪个 RegioneServer，这就意味着，只有 Master 知道 new Region 的位置信息，所以，由 Master 来管理.META.这个表当中的数据的 CRUD

所以结合以上两点表明，在没有 Region 分裂的情况，Master 宕机一段时间是可以忍受的。

**（5）HRegion**

table在行的方向上分隔为多个Region。Region是HBase中分布式存储和负载均衡的最小单元，即不同的region可以分别在不同的Region Server上，但同一个Region是不会拆分到多个server上。

Region按大小分隔，每个表一般是只有一个region。随着数据不断插入表，region不断增大，当region的某个列族达到一个阈值时就会分成两个新的region。

每个region由以下信息标识：< 表名,startRowkey,创建时间>

由目录表(-ROOT-和.META.)记录该region的endRowkey

**（6）Store**

每一个region由一个或多个store组成，至少是一个store，hbase会把一起访问的数据放在一个store里面，即为每个 ColumnFamily建一个store，如果有几个ColumnFamily，也就有几个Store。一个Store由一个memStore和0或者 多个StoreFile组成。 HBase以store的大小来判断是否需要切分region

**（7）MemStore**

memStore 是放在内存里的。保存修改的数据即keyValues。当memStore的大小达到一个阀值（默认**128MB**）时，memStore会被flush到文 件，即生成一个快照。目前hbase 会有一个线程来负责memStore的flush操作。

一个 Hregion 由多个 Store 组成，每个 Store 包含一个列族的所有数据。

Store 包括位于内存的一个 memstore 和位于硬盘的多个 storefile 组成。

写操作先写入 memstore，当 memstore 中的数据量达到某个阈值，HRegionServer 启动 flushcache 进程写入 storefile，每次写入形成单独一个 Hfile。

当总 storefile 大小超过一定阈值后，会把当前的 region 分割成两个，并由 HMaster 分配给相 应的 region 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。

**（8）StoreFile**

memStore内存中的数据写到文件后就是StoreFile，StoreFile底层是以HFile的格式保存。

StoreFile 以 HFile 格式保存在 HDFS 上，请看下图 HFile 的数据组织格式。

一个 Hregion 由多个 Store 组成，每个 Store 包含一个列族的所有数据。

Store 包括位于内存的一个 memstore 和位于硬盘的多个 storefile 组成。

写操作先写入 memstore，当 memstore 中的数据量达到某个阈值，HRegionServer 启动 flushcache 进程写入 storefile，每次写入形成单独一个 Hfile。

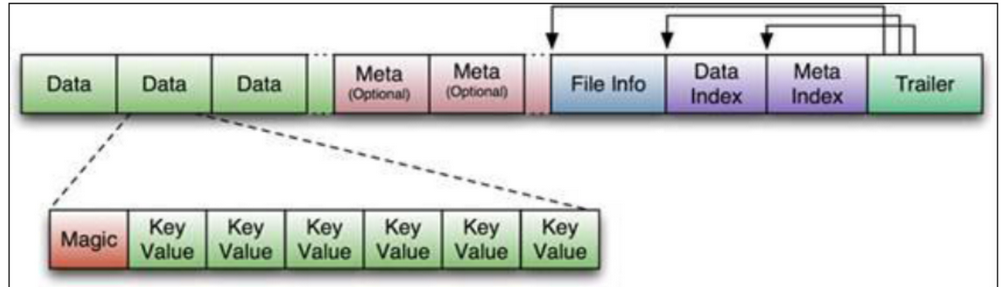
当总 storefile 大小超过一定阈值后，会把当前的 region 分割成两个，并由 HMaster 分配给相 应的 region 服务器，实现负载均衡。

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

**（9）HFile**

HBase中KeyValue数据的存储格式，HFile是Hadoop的 二进制格式文件，实际上StoreFile**就是对Hfile做了轻量级包装，即StoreFile底层就是HFile**

**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。**

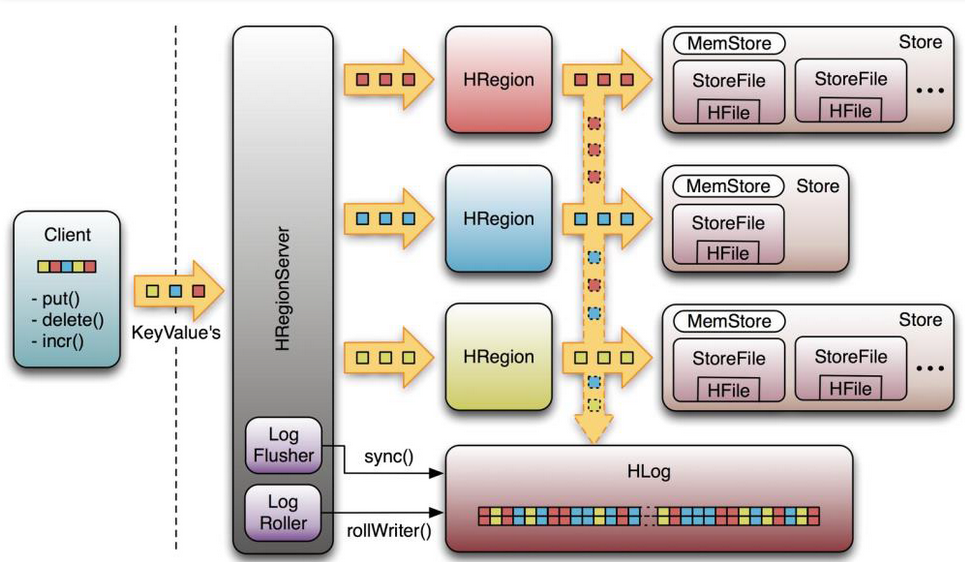


**（10）HLog**

HLog(WAL log)：WAL意为write ahead log，用来做灾难恢复使用，HLog记录数据的所有变更，一旦region server 宕机，就可以从log中进行恢复。

HLog文件就是一个普通的Hadoop Sequence File， Sequence File的value是key时HLogKey对象，其中记录了写入数据的归属信息，除了table和region名字外，还同时包括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的sequence number。 Sequence File的value是HBase的KeyValue对象，即对应HFile中的KeyValue。

**2 物理存储**



1）、Table 中的所有行都按照 RowKsey 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。 当表中的行不断增多，就会有越来越多的 HRegion。

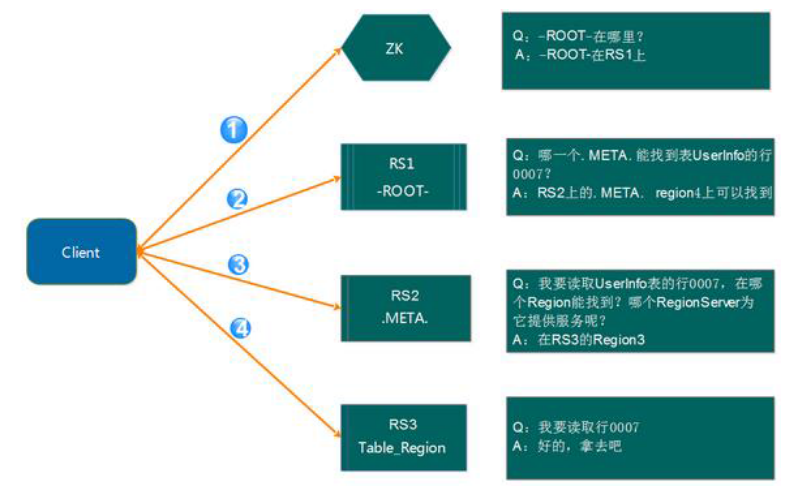
4）、HRegion 是 Hbase 中分布式存储和负载均衡的最小单元。最小单元就表示不同的 HRegion 可以分布在不同的 HRegionserver 上。但一个 HRegion 是不会拆分到多个 server 上的。

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

**3 Region 寻址机制**

既然读写都在 RegionServer 上发生，我们前面有讲到，每个 RegionSever 为一定数量的 Region 服务，那么 Client 要对某一行数据做读写的时候如何能知道具体要去访问哪个 RegionServer 呢？那就是接下来我们要讨论的问题

1. **老的 Region 寻址方式**



第 1 步：Client 请求 ZooKeeper 获得-ROOT-所在的 RegionServer 地址

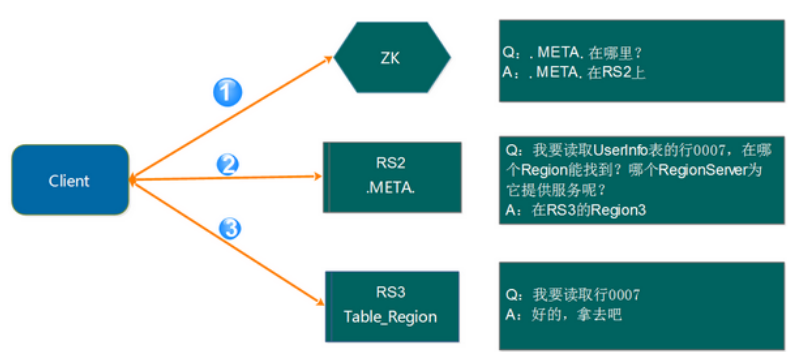
第 2 步：Client 请求-ROOT-所在的 RS 地址，获取.META.表的地址，Client 会将-ROOT-的相关 信息 cache 下来，以便下一次快速访问

第 3 步：Client 请求.META.表的 RegionServer 地址，获取访问数据所在 RegionServer 的地址， Client 会将.META.的相关信息 cache 下来，以便下一次快速访问

第 4 步：Client 请求访问数据所在 RegionServer 的地址，获取对应的数据

从上面的路径我们可以看出，用户需要 3 次请求才能直到用户 Table 真正的位置，这在一定 程序带来了性能的下降。在 0.96 之前使用 3 层设计的主要原因是考虑到元数据可能需要很 大。但是真正集群运行，元数据的大小其实很容易计算出来。在 BigTable 的论文中，每行 METADATA 数据存储大小为 1KB 左右，如果按照一个 Region 为 128M 的计算，3 层设计可以支持的 Region 个数为 2^34 个，采用 2 层设计可以支持 2^17（131072）。那么 2 层设计的情 况下一个集群可以存储 4P 的数据。这仅仅是一个 Region 只有 128M 的情况下。如果是 10G 呢? 因此，通过计算，其实 2 层设计就可以满足集群的需求。因此在 0.96 版本以后就去掉 了-ROOT-表了。

1. **新的 Region 寻址方式**



第 1 步：Client 请求 ZooKeeper 获取.META.所在的 RegionServer 的地址。

第 2 步：Client 请求.META.所在的 RegionServer 获取访问数据所在的 RegionServer 地址，Client 会将.META.的相关信息 cache 下来，以便下一次快速访问。

第 3 步：Client 请求数据所在的 RegionServer，获取所需要的数据。

总结去掉-ROOT-的原因有如下 2 点：

　　其一：提高性能

　　其二：2 层结构已经足以满足集群的需求

**4 RegionServer 工作机制**

**（1）Region 分配**

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

**（2）RegionServer 上线**

Master 使用 zookeeper 来跟踪 RegionServer 状态。当某个 RegionServer 启动时，会首先在 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。因此一旦 RegionServer 上线，Master 能马上得到消息。

**（3）RegionServer 下线**

当 RegionServer 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：

1、RegionServer 和 ZooKeeper 之间的网络断开了。

2、RegionServer 挂了。

无论种情况，RegionServer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 RegionServer 的 znode 数据，并将这台 RegionServer 的 Region 分配给其它还 活着的同志。

**5 Master 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 Master 成为 Master。

2）扫描 ZooKeeper 上的 server 父节点，获得当前可用的 RegionServer 列表。

3）和每个 RegionServer 通信，获得当前已分配的 Region 和 RegionServer 的对应关系。

4）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。

**（2）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 RegionServer 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）。因此，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。