## Hbase数据模型



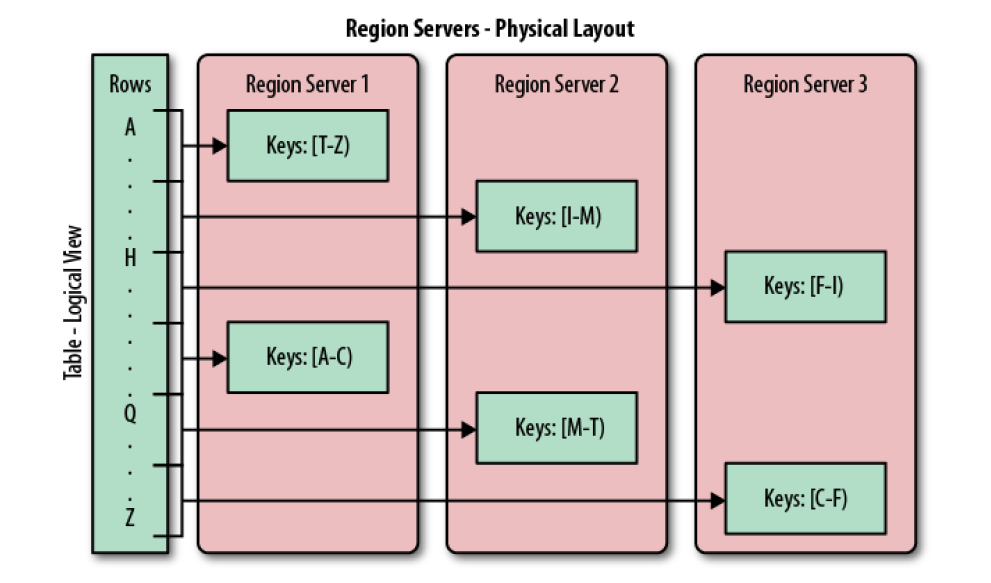
Hbase的一个数据模型实例

## 表和Region

当表包含的行数目非常大时，需要分布存储到每台机器上。因此需要根据行键的值对表中的行进行分区，每个行的区间构成一个分区，成为“Region”，它是负载均衡和数据分发的基本单位，这个region会被分发到不同的region服务器上。

初始时，每个表只包含一个Region，随着数据的不断插入，Region会不断增大，当一个Region包含的行数量达到一个阈值，就会被自动分裂成两个新的Region。

每个Region的默认大小事100-200M.Master服务器会把不同的region分发到不同的region服务器上，同一个region不会被发送到不同的region服务器上，每个region服务器负责管理一个region集合。



region的行分区加载到不同的服务器中

## 存储

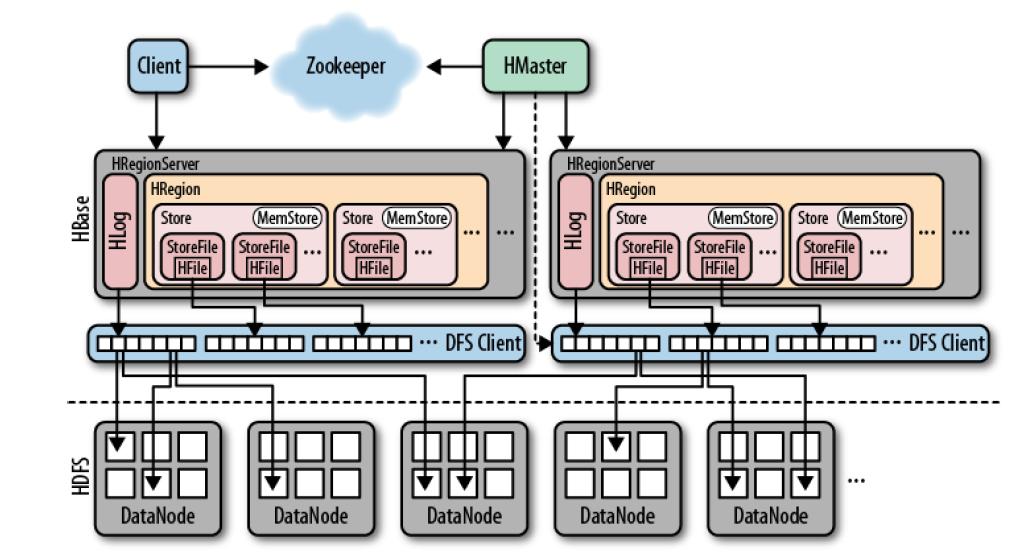


图1 hbase的架构

从上图中可以看出Hbase主要处理两种数据文件格式：预写日志(Write-Ahead Log,WAL)和实际数据文件。主要由HRegionServer管理。

**Region定位**

当尝试访问一行时，客户端首先联系zk集群，检索**.META.**文件所在的RegionServer，图2。（**.META.是一个元数据表，存放着所有关于Region的元数据**。以前的版本是先检索-ROOT-文件，现在已经不存在此文件）。通过.META文件可以找到该Region所在的RegionServer，图3。以上的查找过程会进行缓存，最后，通过查询.META.服务器来获取客户端查询的行键数据所在的region的服务器名（地址）。

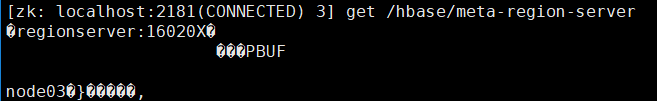


图2 zookeeper中保存了.meta表的地址(node03)

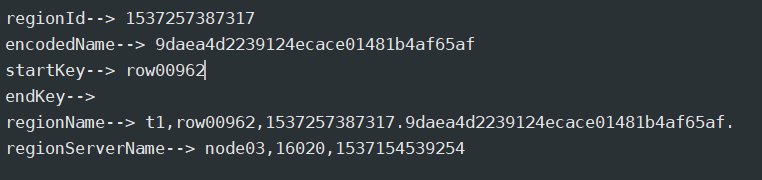


图3 META表中保存的Region的部分元数据信息

一旦知道了数据的位置，即region的位置，HBase会缓存这次的查询信息，同时直接联系管理实际数据的HRegionServer。之后再次查找时就不用查找Meta表了。

**基本工作原理**

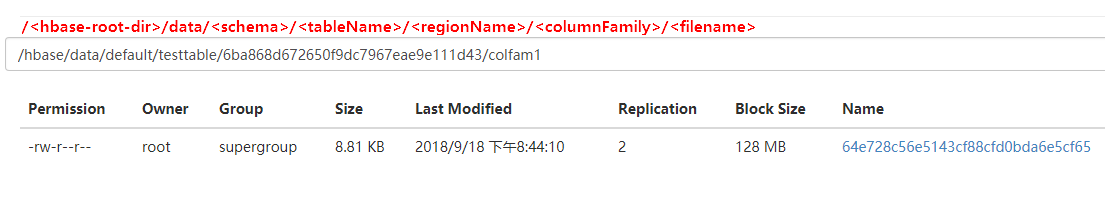
HRegionServer负责打开HRegion，并创建对应的Region实例，当HRegion被打开后，它为每个表的HcolumnFamily创建一个Store实例，每个Store实例包含一个或多个StoreFile实例，它是实际数据存储文件Hfile的轻量级封装。每个store还有其对应的MemStore（缓存），一个HrgionServer共享一个Hlog实例。

每个Region服务器内部管理了一系列的Region对象和一个Hlog文件，其中Hlog时磁盘上的记录文件，记录着所有的更新操作，每个Rgion对象由多个Store组成，每个Store代表一个列族，每个Store又包含一个MemStore和多个StoreFile，MemStore是内存中的缓存，保存最近更新的数据；StoreFile是磁盘上的文件。StoreFile的底层实现方式是Hdfs的Hfile。

Region服务器的工作过程主要包括：读写过程、缓存刷新、StoreFile的合并。

**路径**

Hbase的表在Hdfs中的文件路径



每个表都有一个单独的目录结构

## WAL(Write-Ahead Log)

Region服务器会将数据保存的到内存中，直到积攒足够多的数据再将其刷写到硬盘上，这样可以避免创建很多小文件，存储在内存中的数据是不稳定的。HBase中的解决办法是预写日志（WAL，Hlog是它的一种实现方式），每次编辑操作都回写入日志，只有写入日志成功才会通知客户端操作成功。

WAL中存储了对数据的所有更改。如果服务器崩溃了，可以有效的回放日志。如果将记录写入WAL失败了，整个操作也被认为是失败的。

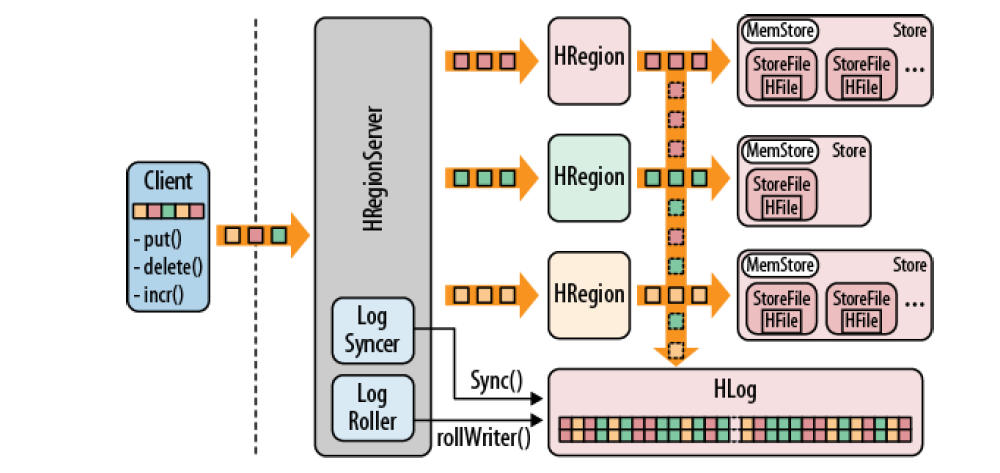


图4 所有的修改都先保存到WAL，再传递给memstore

图4展示了WAL如何与HBase的架构整合，每个region服务器共享一个Hlog，所以对于每一次修改它就像一个日志中心。

处理过程如下，客户端先启动一个操作修改数据，每个修改封装到一个keyValue对象中，发送给匹配region的regionServer。kv对象实例发送到RegionServer后，匹配相应的Hregion，数据写入Hlog(WAL的具体实现)，写入成功之后才会写入Store中的缓存MemStore中。当MemStore达到设定的大小之后，数据会异步写入StoreFile中，每次刷新都会产生一个新的StoreFile文件。

在系统启动的时候，每个RegionServer都会检查自己的Hlog，确认最近一次执行缓存刷新操作之后是否进行了新的写入操作，如果没有，那么所有缓存中的数据已经刷入磁盘文件中，否则需要把这些更新重新写入MemStore中，然后再次刷新缓存，最后删除旧得Hlog文件。

当某个RegionServer发生故障时，zk会通知master，Master首先处理故障RegionServer上的Hlog文件，根据日志记录对其进行Region拆分，分别放到相应的Region对象目录下。然后再把失效的Region重新分配到可用的RegionServer中，并把与该Region对象相关的Hlog日志记录也发送过去，然后新的RegionServer会重新执行一边Hlog的各种操作，把日志中的数据写入MemStore中，然后刷写到磁盘，完成数据恢复 。

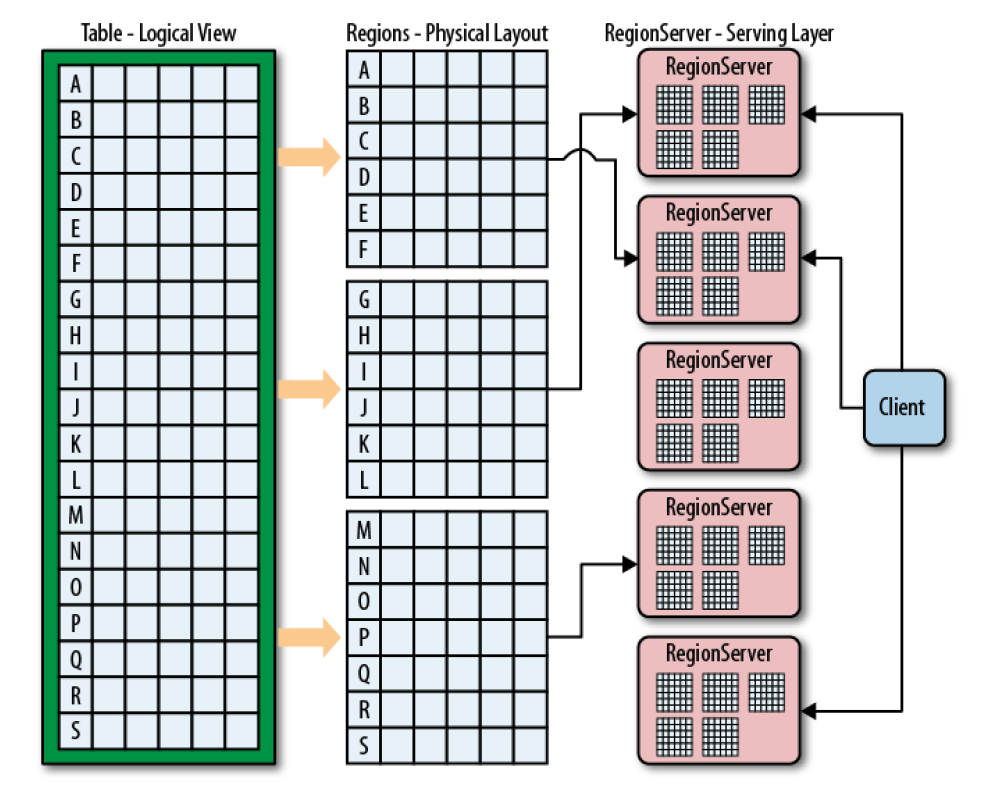
**关闭前预刷写**

当regionServer被要求关闭时，首先检查MemStore，任何配置大于hbase.hregion.preclose.flush.size的Memstore会刷写到磁盘。关闭regionServer会强制所有的MemStore刷鞋到磁盘，而不关心是否达到了配置的最大值。一旦所有的MemStore都刷写到磁盘了，region会关闭，且转移到其他region服务器时不会重新做WAL。

## Store的工作原理

Store对应了表中的一个列族，每个Store中包含了一个MemStore和若干个StoreFile。MemStore是内存缓冲区，用户写入数据时，首先把数据写入MemStore，存满时会刷新到磁盘一个StoreFile文件中。StoreFile文件数量会不断增加，当达到预先设定的数量时会触发文件的合并操作，多个StoreFile文件会被合并成一个大的StoreFile文件。当单个StoreFile文件超过一定阈值时，就会触发文件的分裂操作。同时，当前的一个父Region会被分裂成两个子Region，父Region会下线，新分裂的两个子Region会被Master分到相应的Region服务器上。

## 客户端API的管理功能



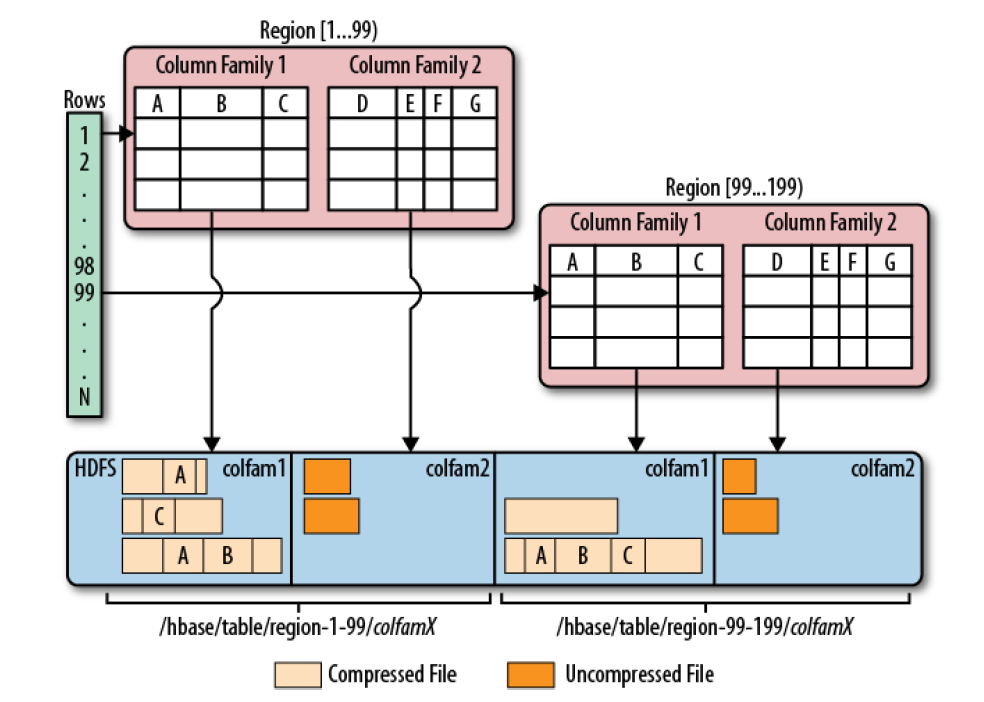
region中的行的逻辑与物理视图

### 表属性-HTableDescriptor

**memstore刷写大小**

默认是64M，通过HTableDescriptor设置

### 列族-HColumnDescriptor



列族映射到独立的存储文件

列族不能被重命名，通常是新建一个列族，然后使用API从旧列族中复制数据到新列族。

用户只能通过构造函数重命名列族。

**最大版本数**

每个列族可以设定每个值能保留的最大版本数量，在断言删除过程中，Hbase会移除最大版本数的数据。

**块大小**

HBase中所有的存储文件被划分成若干小存储块，这些小存储块在get或scan操作时会加载到内存中，这里指的是storeFile（StoreFile是对Hfile的轻量级封装），因此是针对列族而言的。

对于getBlockSize()的方法描述是这样的：The storefile/hfile blocksize for this column family.列族的块，或者说是Hfile的块不同于Hdfs中的块，HDFS的块用于拆分大文件以及提供分布式存储，默认是128M(Hadoop 2.x之后)，HBase中的Hfile大小默认是64k，主要用于高效先加载和缓存数据，并不依赖于HDFS的块大小，只用于HDFS的内部。

**生存期TTL（time to live）**

TTL设置了基于时间戳的一个临界值，内部的管理自动检查TTL值是否达到上限，超过了TTL的数据会被删除，TTL是在列族中设置，但是作用在所有的数据上。

## 过滤器

get()和scan()可以添加限制条件来减少查询得到的数据量，这些限制可以是列族、列、时间戳以及版本号。但是缺少一些细粒度的筛选功能，比如基于正则对行键或值筛选。

Get和Scan两个类都支持过滤器，因为他们的基本API不能对行键、列名和列值进行过滤，但是可以通过过滤器达到这个目的。所有的过滤器都在服务端生效。



过滤器在客户端创建，通过RPC传送到服务器端，然后在服务器端执行过滤操作.

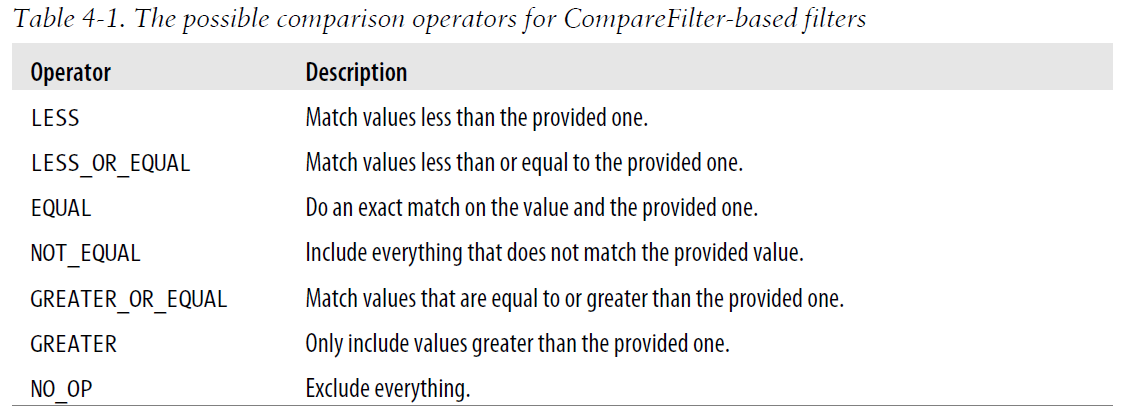
**被过滤掉的信息是不会传回客户端的**，过滤器不能用来指定用户需要哪些信息，而是在读取数据过程中不返回用户不想要的信息。

### 比较过滤器（Comparison Filter）

CompareFilter(CompareOp valueCompareOp,

WritableByteArrayComparable valueComparator)

CompareOp指定比较规则（比较运算符），而WritableByteArrayComparable指定一个比较器，提供多种方法来比较不同的键值。比较过滤器是保留符合条件的值，而不是过滤掉。



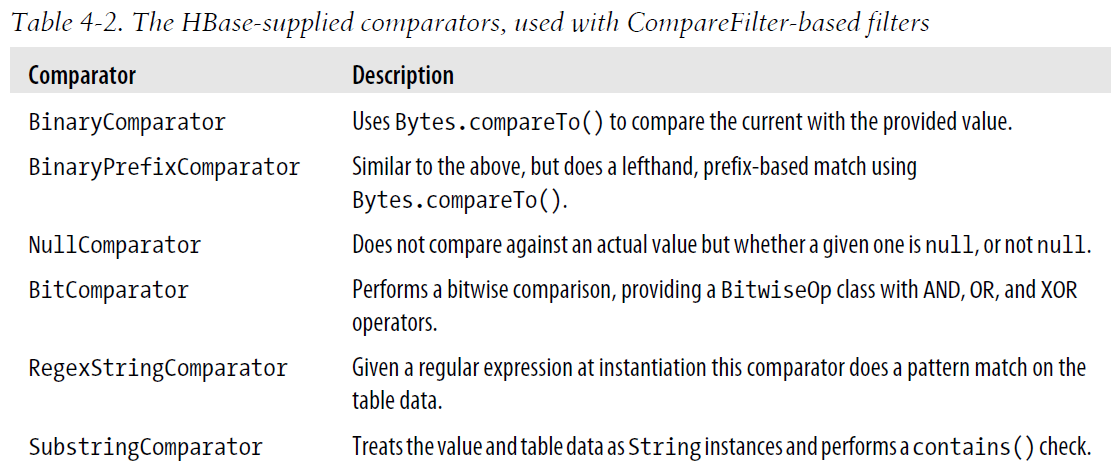


表1 主要的比较过滤器

|  |  |
| --- | --- |
| 行比较器（RowFilter） | 基于行键来过滤数据 |
| 列族过滤器（FamilyFilter） | 过滤特定的列族 |
| 列名过滤器（QualifierFilter） | 过滤特定的列 |
| 值过滤器（ValueFilter） | 筛选特定值的单元格 |
| 参考列过滤器（DenpendentColumnFilter） | 允许用户指定一个参考列，控制其他列的过滤，参考列过滤器使用的是参考列的时间戳，并在过滤时包括所有与引用时间戳相同的列。它相同与一个ValueFilter和一个时间戳过滤器的组合 |

### 专用过滤器

表2 主要的专用过滤器

|  |  |
| --- | --- |
| 单列值过滤器（SingleColumnValueFilter） | 用一列的值决定是否一行数据被过滤 |
| 单列排除过滤器（SingleColumnValue ExcludeFilter） |  |
| 前缀过滤器(PrefixFilter) | 所有与前缀匹配的行都会被返回 |
| 分页过滤器(PageFilter) | 用户可以使用这个过滤器对结果按**行**分页 |
| 行键过滤器(KeyOnlyFilter) | 只返回keyValue中的key，要设置  KeyValue.convertToKeyOnly(boolean)只返回键不返回值。 |
| 首次行键过滤器(FirstKeyOnlyFilter) |  |
| 包含结束的过滤器(InclusiveStopFilter) | 扫描操作中开始行被包含到结果中，但终止行被排除在外。 |
| 时间戳过滤器(TimestampFilter) | 需要装在一个时间戳序列。  TimeStampFilter(List<Long> timeList)  设置时间范围通过scan.setTimeRange() |
| 列计数过滤器(ColumnCountGetFilter) | 限制每行最多取回多少列 |
| 列分页过滤器(ColumnPaginationFilter)  列前缀过滤器(ColumnPrefixFilter) |  |
| 随机行过滤器(RandomRowFilter) | 随即过滤一些行 |

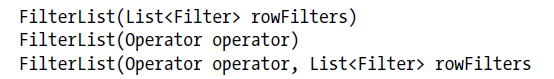
### 附加过滤器

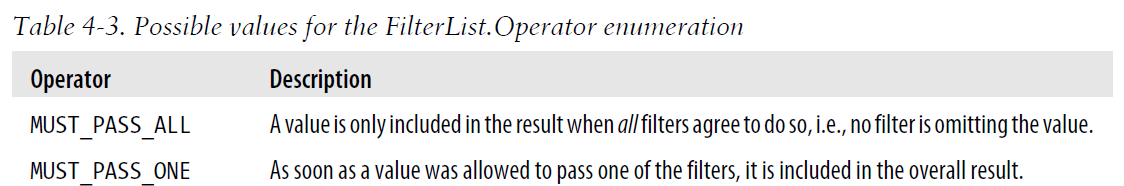
|  |  |
| --- | --- |
| 跳转过滤器(SkipFilter) | 可以对过滤器进行包装，当过滤器发现某一行的某一列数据需要过滤时，那么这一行数据就会整行被过滤掉。所以使用SkipFilter包装原始filter后，数据量一般会变少。 |
| 全匹配过滤器(WhileMatchFilter) | 当一条数据被过滤掉时，它就会直接放弃这次扫描操作。它使用其封装的过滤器来检查keyValue，并确认是否有一行数据因为行键或是列被跳过而过滤。 |

### 自定义过滤器

### FilterList

多个过滤器共同限制返回到客户端的结果。





## 计数器

命令格式：



**单计数器：**

一次只操作一个计数器，用户需要自己设定列，方法有HTable提供。

**多计数器：**

可以对多个列进行计数，使用Increment的addColumn(col,qualifier,amount)方法，通过HTable的increment(Increment incr)方法进行添加。

## 协处理器

协处理器允许用户在region服务器上运行自己的代码，允许用户执行region级别的操作。

协处理器的应用场景：如RDBMS中常见的sum()、avg()等聚合函数，以及SQL也可以在服务端进行，服务端只需在本地扫描并完成统计数据，然后把数值直接返回给客户端。

协处理器提供了一些类，主要分为observer和endpoint。

* Observer：回调函数(也称钩子函数)**在一定特定事件发生**时被执行。

RegionObserver：用于处理数据修改事件，它们与表的region紧密相连

MasterObserver：可以被用作管理或DDL类型的操作，是集群级事件

WALObserver：提供控制WAL的钩子函数。

* endpoint：除了事件处理之外还需要将用户自定义操作添加到服务端，用户代码可以被部署到管理数据的服务端，例如在服务端进行一些计算工作
* Observer协处理器的配置

需要实现Observer接口，完成后打成jar文件，加载方式通常有三种

方式一

shell动态加载，这是表级别的操作，对table的schmema进行修改，不用修改配置文件，不用重启hbase

alter 'table',METHOD='table\_att', 'coprocessor'=>'hdfs路径(jar包) | 主类 | 优先级 | 参数(可选) '

例子：

加载： alter 'callLog', METHOD => 'table\_att',

coprocessor'=>'hdfs://node00:8020/user/callLog/calleeObserver.jar|coprocessor.CalleeCoprocessor|1001'

取消：

alter 'callLog', METHOD => 'table\_att\_unset', NAME => 'coprocessor$1'

方式2

修改配置文件hbase-site.xml,这是系统级别的，影响所有表，需要重启hbase

<property>

<name>hbase.coprocessor.region/master/wal.classes</name>

<value>customF.RegionObserverExample</value>

</property>

把jar包放入hbase/lib，并且要分发到各个region

方式3 通过Api代码实现，即通过API方式加载写处理器