HBase Architecture

phylips@bmy

2011-10-1

1. 序

本文翻译自：http://ofps.oreilly.com/titles/9781449396107/architecture.html，最初发布在 http://duanple.blog.163.com/。

作为开源类 BigTable 实现。HBase 目前已经应用在很多互联网公司中。
项目主页：http://hbase.apache.org/

无论对于高级用户还是普通使用者来说，完整地理解所选择的系统在底层是如何工作的都是非常有用的。本章我们会解释下 HBase 的各个组成部分以及它们相互之间是如何协作的。

2. Seek vs. Transfer

在研究架构本身之前，我们还是先看一下传统 RDBMS 与它的替代者之间的根本上的不同点。特别地，我们将快速地浏览下关系型存储引擎中使用的 B 树及 B+树，以及作为 Bigtable 的存储架构基础的 Log-Structured Merge Tree。

注：需要注意的是 RDBMSs 并不是只能采用 B 树类型的结构，而且也不是所有的 NoSQL 解决方案都使用了与之不同的结构。通常我们都能看到各式各样的混搭型的技术方案，它们都具有一个相同的目标：使用那些对手头上的问题来说最佳的策略。下面我们会解释下为什么 Bigtable 使用了类 LSM-tree 的方式来实现这个目标。

2.1. B+树

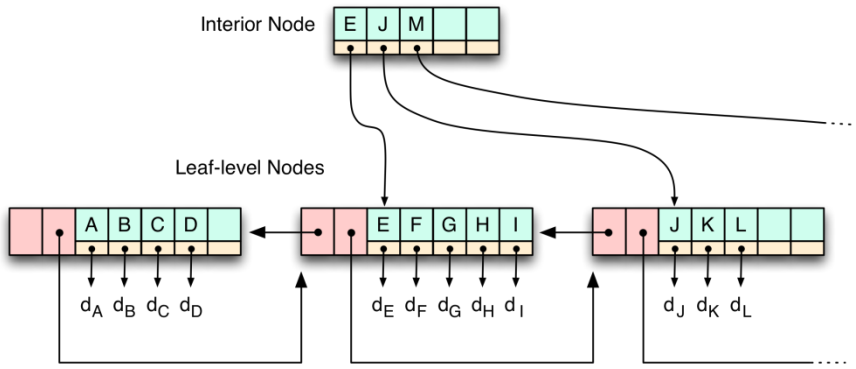
B+树有一些特性可以让用户根据 key 来对记录进行高效地插入，查找和删除。它可以利用每个 segment(也称为一个 page)的下界和上界以及 key 的数目来建立一个动态，多级索引结构。通过使用 segments，达到了比二叉树更高的扇出{!很明显二叉树一个节点只有 2 个出度，而 B+树是个多叉树，一个节点就是一个 segment，因此出度大小就由 segment 本身存储空间决定，出度增加后，就使得树高度变低，减少了所需 seek 操作的数目}，这就大大降低了查找某个特定的 key 所需的 IO 操作数。

此外，它也允许用户高效地进行 range 扫描操作。因为叶子节点相互之间根据 key 的顺序组成了一个链表，这就避免了昂贵的树遍历操作。这也是关系数据库系统使用 B+树进行索引的原因之一。

在一个 B+树索引中，可以得到 page 级别的 locality(这里的 page 概念等价于其他一些系统中 block 的概念)：比如，一个 leaf pages 结构如下。为了插入一个新的索引条目，比如是 key1.5，它会 使用一个新的 key1.5 → rowid 条目来更新 leaf page。

在 page 大小未超过它本身的容量之前，都比较简单。如果 page 大小超出限制，那么就需要将该 page 分割成两个新的 page。参见图 8.1

Figure 8.1. An example B+ tree with one full page



这里有个问题，新的 pages 相互之间不一定是相邻的。所以，现在如果你想查询从 key1 到 key3 之间的内容，就有可能需要读取两个相距甚远的 leaf pages。这也是为什么大部分的基于 B+-树的系统中都提供了 OPTIMIZE TABLE 命令的原因—该命令会顺序地对 table 进行重写，以删除碎片，减少文件尺寸，从而使得这种基于 range 的查询在磁盘上也是顺序进行的。

2.2. Log-Structured Merge-Trees

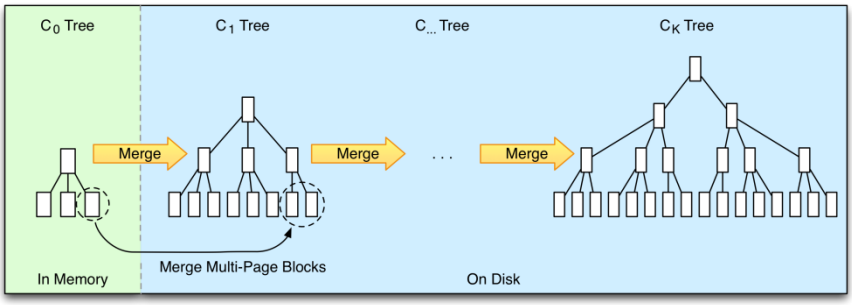
另一方面，LSM-tree，选择的是一种与之不同的策略。进入系统的数据首先会被存储到日志文件中，以完全顺序地方式。一旦日志中记录下了该变更，它就会去更新一个内存中的存储结构，该结构持有最近的那些更新以便于快速的查找。

当系统已经积累了足够的更新，以及内存中的存储结构填满的时候，它会将 key → record 对组成的有序链表 flush 到磁盘，创建出一个新的存储文件。此时，log 文件中对应的更新就可以丢弃了，因为所有的更新操作已经被持久化了。

存储文件的组织方式类似于 B 树，但是专门为顺序性的磁盘访问进行了优化。所有的 nodes 都被完全填充，存储为单 page 或者多 page 的 blocks。存储文件的更新是以一种 rolling merge 的方式进行的，比如，只有当某个 block 填满时系统才会将对应的内存数据和现有的多 page blocks 进行合并。

图 8.2 展示了一个多 page 的 block 如何从 in-memory tree 合并为一个存储磁盘上的树结构。最后，这些树结构会被用来 merge 成更大的树结构。

Figure 8.2. Multi-page blocks are iteratively merged across LSM trees



随着时间的推进将会有更多的 flush 操作发生，会产生很多存储文件，一个后台进程负责将这些文件聚合成更大的文件，这样磁盘 seek 操作就限制在一定数目的存储文件上。存储在磁盘上的树结构也可以被分割成多个存储文件。因为所有的存储数据都是按照 key 排序的，因此在现有节点中插入新的 keys 时不需要重新进行排序。

查找通过 merging 的方式完成，首先会搜索内存存储结构，接下来是磁盘存储文件。通过这种方式，从客户端的角度看到的就是一个关于所有已存储数据的一致性视图，而不管数据当前是否驻留在内存中。删除是一种特殊的更新操作，它会存储一个删除标记，该标记会在查找期间用来跳过那些已删除的 keys。当数据通过 merging 被重新写回时，删除标记和被该标记所遮蔽的 key 都会被丢弃掉。

用于管理数据的后台进程有一个额外的特性，它可以支持断言式的删除。也就是说删除操作可以通过在那些想丢弃的记录上设定一个 TTL(time-to-live)值来触发。比如，设定 TTL 值为 20 天，那么 20 天后记录就变成无效的了。Merge 进程会检查该断言，当断言为 true 时，它就会在写回的 blocks 中丢弃该记录。

B 数和 LSM-tree 本质上的不同点，实际上在于它们使用现代硬件的方式，尤其是磁盘。

Seek vs. Sort and Merge in Numbers

对于大规模场景，计算瓶颈在磁盘传输上。CPU RAM 和磁盘空间每 18-24 个月就会翻番，但是 seek 开销每年大概才提高 5%。

如前面所讨论的，有两种不同的数据库范式，一种是 Seek，另一种是 Transfer。RDBMS 通常都是 Seek 型的，主要是由用于存储数据的 B 树或者是 B+树结构引起的，在磁盘 seek 的速率级别上实现各种操作，通常每个访问需要 log(N)个 seek 操作。

另一方面，LSM-tree 则属于 Transfer 型。在磁盘传输速率的级别上进行文件的排序和 merges 以及 log(对应于更新操作)操作。根据如下的各项参数：

- 10 MB/second transfer bandwidth
- 10 milliseconds disk seek time
- 100 bytes per entry (10 billion entries)
- 10 KB per page (1 billion pages)

在更新 100,000,000 条记录的 1%时，将会花费：

- 1,000 days with random B-tree updates
- 100 days with batched B-tree updates
- 1 day with sort and merge

很明显，在大规模情况下，seek 明显比 transfer 低效。

比较 B+树和 LSM-tree 主要是为了理解它们各自的优缺点。如果没有太多的更新操作，B+树可以工作地很好，因为它们会进行比较繁重的优化来保证较低的访问时间。越快越多地将数据添加到随机的位置上，页面就会越快地变得碎片化。最终，数据传入的速度可能会超过优化进程重写现存文件的速度。更新和删除都是以磁盘 seek 的速率级别进行的，这就使得用户受限于最差的那个磁盘性能指标。

LSM-tree 工作在磁盘传输速率的级别上，同时可以更好地扩展到更大的数据规模上。同时也能保证一个比较一致的插入速率，因为它会使用日志文件+一个内存存储结构把随机写操作转化为顺序写。读操作与写操作是独立的，这样这两种操作之间就不会产生竞争。

存储的数据通常都具有优化过的存放格式。对于访问一个 key 所需的磁盘 seek 操作数也有一个可预测的一致的上界。同时读取该 key 后面的那些记录也不会再引入额外的 seek 操作。通常情况下，一个基于 LSM-tree 的系统的开销都是透明的：如果有 5 个存储文件，那么访问操作最多需要 5 次磁盘 seek。然而你没有办法判断一个 RDBMS 的查询需要多少次磁盘 seek，即使是在有索引的情况下。

3. Storage

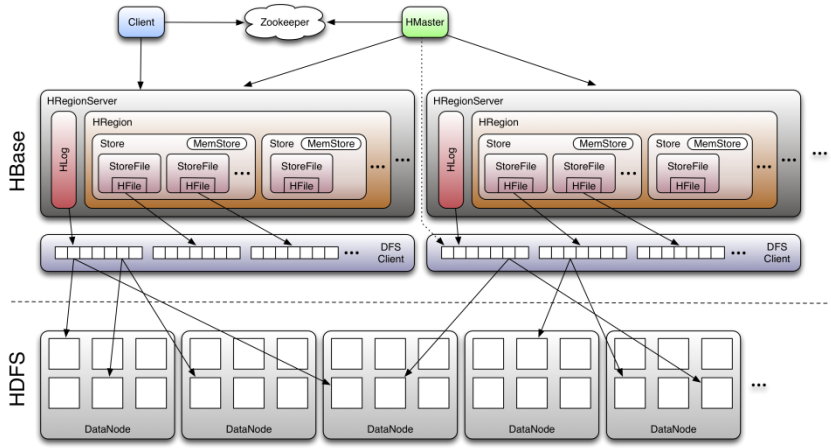
HBase一个比较不为人知的方面是数据在底层是如何存储的。大部分的用户可能从来都不需要关注它。但是当你需要按照自己的方式对各种高级配置项进行设置时可能就得不得不去了解它。Chapter 11, *Performance Tuning*列出了一些例子。Appendix A, *HBase Configuration Properties*有一个更全的参考列表。

需要了解这些方面的另一个原因是，如果因为各种原因，灾难发生了，然后你需要恢复一个 HBase 安装版本。这时候，知道所有的数据都存放在哪，如何在 HDFS 级别上访问它们，就变得很重要了。你就可以利用这些知识来访问那些通常情况下不可访问的数据。当然，这种事情最好不发生，但是谁能保证它不会发生呢？

3.1. 概览

作为理解HBase的文件存储层的各组成部分的第一步，我们先来画张结构图。Figure 8.3, “HBase handles files in the file system, which stores them transparently in HDFS”展示了HBase和HDFS是如何协作来存储数据的。

Figure 8.3. HBase handles files in the file system, which stores them transparently in HDFS



上图表明，HBase处理的两种基本文件类型：一个用于write-ahead log，另一个用于实际的数据存储。文件主要是由HRegionServer处理。在某些情况下，HMaster也会执行一些底层的文件操作(与 0.90.x相比，这在 0.92.0 中有些差别)。你可能也注意到了，当存储在HDFS中时，文件实际上会被划分为很多小blocks。这也是在你配置系统来让它可以更好地处理更大或更小的文件时，所需要了解的地方。更细节的内容，我们会在 the section called “HFile Format”里描述。

通常的工作流程是，一个新的客户端为找到某个特定的行 key 首先需要联系 Zookeeper Qurom。它会从 ZooKeeper 检索持有-ROOT- region 的服务器名。通过这个信息，它询问拥有-ROOT- region 的 region server，得到持有对应行 key 的.META. 表 region 的服务器名。这两个操作的结果都会被缓存下来，因此只需要查找一次。最后，它就可以查询.META.服务器然后检索到包含给定行 key 的 region 所在的服务器。

一旦它知道了给定的行所处的位置，比如，在哪个region里，它也会缓存该信息同时直接联系持有该region的HRegionServer。现在，客户端就有了去哪里获取行的完整信息而不需要再去查询.META.服务器。更多细节可以参考 the section called “Region Lookups”。

注：在启动 HBase 时，HMaster 负责把 regions 分配给每个 HRegionServer。包括-ROOT-和.META.表。更多细节参考 the section called “The Region Life Cycle”

HRegionServer打开region然后创建对应的HRegion对象。当HRegion被打开后，它就会为表中预先定义的每个HColumnFamily创建一个Store实例。每个Store实例又可能有多个StoreFile实例，StoreFile是对被称为HFile的实际存储文件的一个简单封装。一个Store实例还会有一个Memstore，以及一个由HRegionServer共享的HLog实例(见 the section called “Write-Ahead Log”)。

3.2. Write Path

客户端向HRegionServer产生一个HTable.put(Put)请求。HRegionServer将该请求交给匹配的HRegion实例。现在需要确定数据是否需要通过HLog类写入write-ahead log(the WAL)。该决定基于客户端使用 方法 `Put.setWriteToWAL(boolean)` 所设置的 flag。WAL 是一个标准的 Hadoop SequenceFile，里面存储了 HLogKey 实例。这些 keys 包含一个序列号和实际的数据，用来 replay 那些在服务器 crash 之后尚未持久化的数据。

一旦数据写入(or not)了 WAL，它也会被放入 Memstore。与此同时，还会检查 Memstore 是否满了，如果满了需要产生一个 flush 请求。该请求由 HRegionServer 的单独的线程进行处理，该线程会把数据写入到位于 HDFS 上的新 HFile 里。同时它也会保存最后写入的序列号，这样系统就知道目前为止持久化到哪了。

3.3. Files

HBase在HDFS上有一个可配置的根目录，默认设置为"/hbase"。 the section called "Co-Existing Clusters"说明了在共享HDFS集群时如何换用另一个根目录。可以使用 **hadoop dfs -lsr**命令来查看HBase存储的各种文件。在此之前，我们先创建并填写一个具有几个regions的table：

```
hbase(main):001:0> create 'testtable', 'colfam1', \
  { SPLITS => ['row-300', 'row-500', 'row-700' , 'row-900'] }
0 row(s) in 0.1910 seconds
hbase(main):002:0> for i in '0'..'9' do for j in '0'..'9' do
\
  for k in '0'..'9' do put 'testtable', "row-#{i}#{j}#{k}",
\
  "colfam1:#{j}#{k}", "#{j}#{k}" end end end
0 row(s) in 1.0710 seconds
0 row(s) in 0.0280 seconds
0 row(s) in 0.0260 seconds
...
hbase(main):003:0> flush 'testtable'
```

```
0 row(s) in 0.3310 seconds

hbase(main):004:0> for i in '0'..'9' do for j in '0'..'9' do
\

  for k in '0'..'9' do put 'testtable', "row-#{i}#{j}#{k}",
\

  "colfam1:#{j}#{k}", "#{j}#{k}" end end end

0 row(s) in 1.0710 seconds

0 row(s) in 0.0280 seconds

0 row(s) in 0.0260 seconds

...
```

Flush 命令会将内存数据写入存储文件，否则我们必须等着它直到超过配置的 flush 大小才会将数据插入到存储文件中。最后一轮的 put 命令循环是为了再次填充 write-ahead log。

下面是上述操作完成之后，HBase 根目录下的内容：

```
$ $HADOOP_HOME/bin/hadoop dfs -lsr /hbase

      ...

      0 /hbase/.logs

      0 /hbase/.logs/foo.internal,60020,1309812147645

      0 /hbase/.logs/foo.internal,60020,1309812147645/ \

foo.internal%2C60020%2C1309812147645.1309812151180

      0 /hbase/.oldlogs

     38 /hbase/hbase.id

      3 /hbase/hbase.version

      0 /hbase/testtable

    487 /hbase/testtable/.tableinfo
```

```
      0 /hbase/testtable/.tmp

      0 /hbase/testtable/1d562c9c4d3b8810b3dbeb21f5746855

      0
/hbase/testtable/1d562c9c4d3b8810b3dbeb21f5746855/.oldlog
s

    124
/hbase/testtable/1d562c9c4d3b8810b3dbeb21f5746855/.oldlog
s/ \

hlog.1309812163957

    282
/hbase/testtable/1d562c9c4d3b8810b3dbeb21f5746855/.region
info

      0
/hbase/testtable/1d562c9c4d3b8810b3dbeb21f5746855/.tmp

      0
/hbase/testtable/1d562c9c4d3b8810b3dbeb21f5746855/colfam1

   11773
/hbase/testtable/1d562c9c4d3b8810b3dbeb21f5746855/colfam1
/ \

646297264540129145

      0 /hbase/testtable/66b4d2adcc25f1643da5e6260c7f7b26

    311
/hbase/testtable/66b4d2adcc25f1643da5e6260c7f7b26/.region
info

      0
/hbase/testtable/66b4d2adcc25f1643da5e6260c7f7b26/.tmp

      0
/hbase/testtable/66b4d2adcc25f1643da5e6260c7f7b26/colfam1

   7973
/hbase/testtable/66b4d2adcc25f1643da5e6260c7f7b26/colfam1
/ \

3673316899703710654
```

```
      0 /hbase/testtable/99c0716d66e536d927b479af4502bc91

    297
/hbase/testtable/99c0716d66e536d927b479af4502bc91/.region
info

      0
/hbase/testtable/99c0716d66e536d927b479af4502bc91/.tmp

      0
/hbase/testtable/99c0716d66e536d927b479af4502bc91/colfam1

   4173
/hbase/testtable/99c0716d66e536d927b479af4502bc91/colfam1
/ \

1337830525545548148

      0 /hbase/testtable/d240e0e57dcf4a7e11f4c0b106a33827

    311
/hbase/testtable/d240e0e57dcf4a7e11f4c0b106a33827/.region
info

      0
/hbase/testtable/d240e0e57dcf4a7e11f4c0b106a33827/.tmp

      0
/hbase/testtable/d240e0e57dcf4a7e11f4c0b106a33827/colfam1

   7973
/hbase/testtable/d240e0e57dcf4a7e11f4c0b106a33827/colfam1
/ \

3164171882624569 22

      0 /hbase/testtable/d9ffc3a5cd016ae58e23d7a6cb937949

    311
/hbase/testtable/d9ffc3a5cd016ae58e23d7a6cb937949/.region
info

      0
/hbase/testtable/d9ffc3a5cd016ae58e23d7a6cb937949/.tmp

      0
/hbase/testtable/d9ffc3a5cd016ae58e23d7a6cb937949/colfam1
```

```
   7973
/hbase/testtable/d9ffc3a5cd016ae58e23d7a6cb937949/colfam1
/ \

4238940159225512178
```

注：由于空间的限制，我们对输出内容进行了删减，只留下了文件大小和名称部分。你自己在集群上运行命令时可以看到更多的细节信息。

文件可以分成两类：一是直接位于 HBase 根目录下面的那些，还有就是位于 table 目录下面的那些。

3.3.1. Root Level Files

第一类文件是由 HLog 实例处理的 write-ahead log 文件，这些文件创建在 HBase 根目录下一个称为.logs 的目录。Logs 目录下包含针对每个 HRegionServer 的子目录。在每个子目录下，通常有几个 HLog 文件(因为 log 的切换而产生)。来自相同 region server 的 regions 共享同一系列的 HLog 文件。

一个有趣的现象是log file大小被报告为 0。对于最近创建的文件通常都是这样的，因为HDFS正使用一个内建的append支持来对文件进行写入，同时只有那些完整的blocks对于读取者来说才是可用的—包括 **hadoop dfs -lsr**命令。尽管put操作的数据被安全地持久化，但是当前被写入的log文件大小信息有些轻微的脱节。

等一个小时 log 文件切换后，这个时间是由配置项：
`hbase.regionserver.logroll.period`
控制的(默认设置是 60 分钟)，你就能看到现有的 log 文件的正确大小了，因为它已经被关闭了，而且 HDFS 可以拿到正确的状态了。而在它之后的那个新 log 文件大小又变成 0 了：

```
249962 /hbase/.logs/foo.internal,60020,1309812147645/ \

foo.internal%2C60020%2C1309812147645.1309812151180

      0 /hbase/.logs/foo.internal,60020,1309812147645/ \

foo.internal%2C60020%2C1309812147645.1309815751223
```

当日志文件不再需要时，因为现有的变更已经持久化到存储文件中了，它们就会被移到 HBase 根目录下的.oldlogs 目录下。这是在 log 文件达到上面的切换阈值时触发的。老的日志文件默认会在十分钟后被 master 删除，通过 hbase.master.logcleaner.ttl 设定。Master 默认每分钟会对这些文件进行检查，可

以通过 hbase.master.cleaner.interval 设定。

hbase.id 和 hbase.version 文件包含集群的唯一 ID 和文件格式版本号：

```
$ hadoop dfs -cat /hbase/hbase.id

$e627e130-0ae2-448d-8bb5-117a8af06e97

$ hadoop dfs -cat /hbase/hbase.version

7
```

它们通常是在内部使用因此通常不用关心这两个值。此外，随着时间的推进还会产生一些 root 级的目录。splitlog 和.corrupt 目录分别是 log split 进程用来存储中间 split 文件的和损坏的日志文件的。比如：

```
0 /hbase/.corrupt

     0
/hbase/splitlog/foo.internal,60020,1309851880898_hdfs%3A%
2F%2F \

localhost%2Fhbase%2F.logs%2Ffoo.internal%2C60020%2C130985
0971208%2F \

foo.internal%252C60020%252C1309850971208.1309851641956/te
sttable/ \

d9ffc3a5cd016ae58e23d7a6cb937949/recovered.edits/00000000
00000002352
```

上面的例子中没有损坏的日志文件，只有一个分阶段的split文件。关于log splitting过程参见 the section called “Replay”。

3.3.2. Table Level Files

HBase中的每个table都有它自己的目录，位于HBase根目录之下。每个table目录包含一个名为.tableinfo的顶层文件，该文件保存了针对该table的HTableDescriptor(具体细节参见 the section called “Tables”)的序列化后的内容。包含了table和column family schema信息，同时可以被读取，比如通过使用工具可以查看表的定义。.tmp目录包含一些中间数据，比如当.tableinfo被更新时该目录就会被用到。

3.3.3. Region Level Files

在每个 table 目录内，针对表的 schema 中的每个 column family 会有一个单独的目录。目录名称还包含 region name 的 MD5 hash 部分。比如通过 master 的 web UI，点击 testtable 链接后，其中 User Tables 片段的内容如下：

```
testtable,row-500,1309812163930.d9ffc3a5cd016ae58e23d7a6cb937949.
```

MD5 hash 部分是”d9ffc3a5cd016ae58e23d7a6cb937949”，它是通过对 region name 的剩余部分进行编码生成的。比如”testtable,row-500,1309812163930”。尾部的点是整个 region name 的一部分：它表示这是一种包含 hash 的新风格的名称。在 HBase 之前的版本中，region name 中并不包含 hash。

注：需要注意的是-ROOT-和.META.元数据表仍然采用老风格的格式，比如它们的 region name 不包含 hash，因此结尾就没有那个点。

.META.,,1.1028785192

对于存储在磁盘上的目录中的 region names 编码方式也是不同的：它们使用 Jenkins hash 来对 region name 编码。

Hash 是用来保证 region name 总是合法的，根据文件系统的规则：它们不能包含任何特殊字符，比如”/”，它是用来分隔路径的。这样整个的 region 文件路径就是如下形式：

```
/<hbase-root-dir>/<tablename>/<encoded-regionname>/<column-family>/<filename>
```

在每个 column-family 下可以看到实际的数据文件。文件的名字是基于 Java 内建的随机数生成器产生的任意数字。代码会保证不会产生碰撞，比如当发现新生成的数字已经存在时，它会继续寻找一个未被使用的数字。

Region 目录也包含一个.regioninfo 文件，包含了对应的 region 的 HRegionInfo 的序列化信息。类似于.tableinfo，它也可以通过外部工具来查看关于 region 的相关信息。**hbase hbck** 工具可以用它来生成丢失的 table 条目元数据。

可选的.tmp 目录是按需创建地，用来存放临时文件，比如某个 compaction 产生的重新写回的文件。一旦该过程结束，它们会被立即移入 region 目录。在极端情

况下，你可能能看到一些残留文件，在 region 重新打开时它们会被清除。

在write-ahead log replay期间，任何尚未提交的修改会写入到每个region各自对应的文件中。这是阶段 1(看下 the section called “Root Level Files”中的splitlog目录)，之后假设log splitting过程成功完成-然后会将这些文件原子性地move到recovered.edits目录下。当该region被打开时，region server能够看到这些recovery文件然后replay相应的记录。

Split vs. Split

在 write-ahead log 的 splitting 和 regions 的 splitting 之间有明显的区别。有时候，在文件系统中很难区分文件和目录的不同，因为它们两个都涉及到了 splits 这个名词。为避免错误和混淆，确保你已经理解了二者的不同。

一旦一个 region 因为大小原因而需要 split，一个与之对应的 splits 目录就会创建出来，用来筹划产生两个子 regions。如果这个过程成功了—通常只需要几秒钟或更少—之后它们会被移入 table 目录下用来形成两个新的 regions，每个代表原始 region 的一半。

换句话说，当你发现一个 region 目录下没有.tmp 目录，那么说明目前它上面没有 compaction 在执行。如果也没有 recovered.edits 目录，那么说明目前没有针对它的 write-ahead log replay。

注：在 HBase 0.90.x 版本之前，还有一些额外的文件，目前已被废弃了。其中一个是 oldlogfile.log，该文件包含了对于相应的 region 已经 replay 过的 write-ahead log edits。oldlogfile.log.old(加上一个.old 扩展名)表明在将新的 log 文件放到该位置时，已经存在一个 oldlogfile.log。另一个值得注意的是在老版 HBase 中的 compaction.dir，现在已经被.tmp 目录替换。

本节总结了下 HBase 根目录下的各种目录所包含的一系列内容。有很多是由 region split 过程产生的中间文件。在下一节里我们会分别讨论。

3.4. Region Splits

当一个 region 内的存储文件大于 hbase.hregion.max.filesize(也可能是在 column family 级别上配置的)的大小时，该 region 就需要 split 为两个。起始过程很快就完成了，因为系统只是简单地为新 regions(也称为 daughters)创建两个引用文件，每个只持有原始 region 的一半内容。

Region server 通过在 parent region 内创建 splits 目录来完成。之后，它会关闭该 region 这样它就不再接受任何请求。

Region server 然后开始准备生成新的子 regions(使用多线程)，通过在 splits 目录内设置必要的文件结构。里面包括新的 region 目录及引用文件。如果该过程成功

完成，它就会把两个新的 region 目录移到 table 目录下。.META.table 会进行更新，指明该 region 已经被 split，以及子 regions 分别是谁。这就避免了它被意外的重新打开。实例如下：

```
ow:
testtable,row-500,1309812163930.d9ffc3a5cd016ae58e23d7a6c
b937949.

  column=info:regioninfo, timestamp=1309872211559,
value=REGION => {NAME => \


'testtable,row-500,1309812163930.d9ffc3a5cd016ae58e23d7a6
cb937949. \

    TableName => 'testtable', STARTKEY => 'row-500', ENDKEY
=> 'row-700', \

    ENCODED => d9ffc3a5cd016ae58e23d7a6cb937949, OFFLINE =>
true,

    SPLIT => true,}



  column=info:splitA, timestamp=1309872211559, value=REGION
=> {NAME => \


'testtable,row-500,1309872211320.d5a127167c6e2dc5106f066c
c84506f8. \

   TableName => 'testtable', STARTKEY => 'row-500', ENDKEY
=> 'row-550', \

   ENCODED => d5a127167c6e2dc5106f066cc84506f8,}

  column=info:splitB, timestamp=1309872211559, value=REGION
=> {NAME => \


'testtable,row-550,1309872211320.de27e14ffc1f3fff65ce424f
cf14ae42. \

   TableName => [B@62892cc5', STARTKEY => 'row-550', ENDKEY
=> 'row-700', \
```

```
    ENCODED => de27e14ffc1f3fff65ce424fcf14ae42,}
```

可以看到原始的 region 在”row-550”处被分成了两个 regions。在 info:regioninfo 中的”SPLIT=>true”表面该 region 目前已经分成了两个 regions：splitA 和 splitB。

引用文件的名称是另一个随机数，但是会使用它所引用的 region 的 hash 作为后缀，比如：

```
/hbase/testtable/d5a127167c6e2dc5106f066cc84506f8/colfam1
/ \

6630747383202842155.d9ffc3a5cd016ae58e23d7a6cb937949
```

该引用文件代表了 hash 值为” d9ffc3a5cd016ae58e23d7a6cb937949”的原始 region 的一半内容。引用文件仅仅有很少量的信息：原始 region split 点的 key，引用的是前半还是后半部分。这些引用文件会通过 HalfHFileReader 类来读取原始 region 的数据文件。

现在两个子 regions 已经就绪，同时将会被同一个服务器并行打开。现在需要更新.META.table，将这两个 regions 作为可用 region 对待—看起来就像是完全独立的一样。同时会启动对这两个 regions 的 compaction—此时会异步地将存储文件从原始 region 真正地写成两半，来取代引用文件。这些都发生在子 regions 的.tmp 目录下。一旦文件生成完毕，它们就会原子性地替换掉之前的引用文件。
原始 region 最终会被清除，意味着它会从.META.table 中删除，它的所有磁盘上的文件也会被删除。最后，master 会收到关于该 split 的通知，它可以因负载平衡等原因将这些新的 regions 移动到其他服务器上。

ZooKeeper 支持

Split 中的所有相关步骤都会通过 Zookeeper 进行追踪。这就允许在服务器出错时，其他进程可以知晓该 region 的状态。

3.5. Compactions

存储文件处于严密的监控之下，这样后台进程就可以保证它们完全处于控制之中。Memstores的flush操作会逐步的增加磁盘上的文件数目。当数目足够多的时候，compaction进程会将它们合并成更少但是更大的一些文件。当这些文件中的最大的那个超过设置的最大存储文件大小时会触发一个region split过程。(see the section called “Region Splits”).

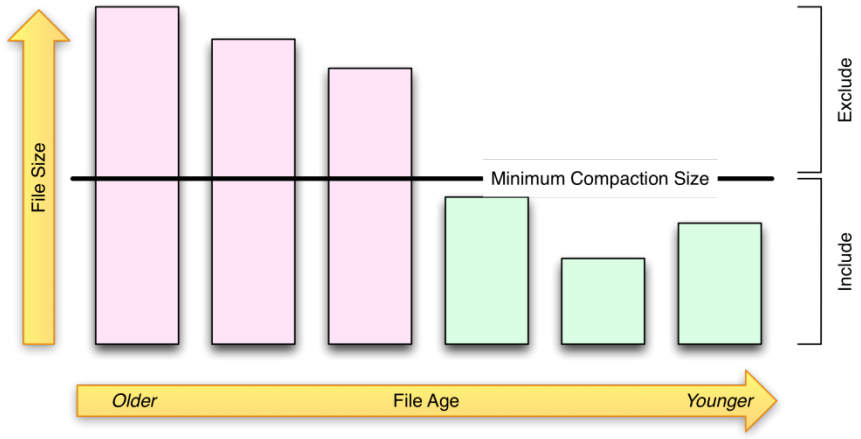
有两种类型的 Compactions：minor 和 major。Minor compaction 负责将一些小文

件合并成更大的一个文件。合并的文件数通过 hbase.hstore.compaction.min 属性进行设置(以前该参数叫做 hbase.hstore.compactionThreshold，尽管被弃用了但是目前还支持该参数)。默认该参数设为 3，同时该参数必须>=2。如果设得更大点，会延迟 minor compaction 的发生，但是一旦它启动也会需要更多的资源和更长的时间。一个 minor compaction 所包含的最大的文件数被设定为 10，可以通过 hbase.hstore.compaction.max 进行配置。

可以通过设置 hbase.hstore.compaction.min.size(设定为该 region 的对应的 memstore 的 flush size) 和 hbase.hstore.compaction.max.size(默认是 Long.MAX_VALUE)来减少需要进行 minor compaction 的文件列表。任何大于最大的 compaction size 的文件都会被排除在外。最小的 compaction size 是作为一个阈值而不是一个限制，也就是说在达到单次 compaction 允许的文件数上限之前，那些小于该阈值的文件都会被包含在内。

图 8.4 展示了一个存储文件集合的实例。所有那些小于最小的 compaction 阈值的文件都被包含进了 compaction 中。

Figure 8.4. A set of store files showing the minimum compaction threshold



该算法会使用 hbase.hstore.compaction.ratio (defaults to 1.2, or 120%)来确保总是能够选出足够的文件来进行 compaction。根据该 ratio，那些大小大于所有新于它的文件大小之和的文件也能够被选入。计算时，总是根据文件年龄从老到新进行选择，以保证老文件会先被 compacted。通过上述一系列 compaction 相关的参数可以用来控制一次 minor compaction 到底选入多少个文件。

HBase 支持的另外一种 compaction 是 major compaction：它会将所有的文件 compact 成一个。该过程的运行是通过执行 compaction 检查自动确定的。当 memstore 被 flush 到磁盘，执行了 compact 或者 major_compact 命令或者产生了相关 API 调用时，或者后台线程的运行，就会触发该检查。Region server 会通过 CompactionChecker 类实现来运行该线程。

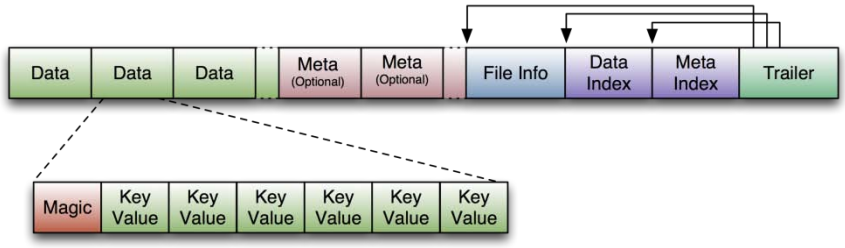
如果用户调用了 major_compact 命令或者 majorCompact()API 调用，都会强制

major compaction 运行。否则，服务端会首先检查是否该进行 major compaction，通过查看距离上次运行是否满足一定时间，比如是否达到 24 小时。

3.6. HFile 格式

实际的文件存储是通过 HFile 类实现的，它的产生只有一个目的：高效存储 HBase 数据。它基于 Hadoop 的 TFile 类，模仿了 Google 的 Bigtable 架构中使用的 SSTable 格式。之前 HBase 采用的是 Hadoop MapFile 类，实践证明性能不够高。图 8 展示了具体的文件格式：

Figure 8.5. The HFile structure



文件是变长的，定长的块只有 file info 和 trailer 这两部分。如图所示，trailer 中包含指向其他 blocks 的指针。Trailer 会被写入到文件的末尾。Index blocks 记录了 data 和 meta blocks 的偏移。data 和 meta blocks 实际上都是可选部分。但是考虑到 HBase 使用数据文件的方式，通常至少可以在文件中找到 data blocks。

Block 大小是通过 HColumnDescriptor 配置的，而它是在 table 创建时由用户指定的，或者是采用了默认的标准值。实例如下：

```
{NAME => 'testtable', FAMILIES => [{NAME => 'colfam1',

BLOOMFILTER => 'NONE', REPLICATION_SCOPE => '0', VERSIONS =>
'3',

COMPRESSION \=> 'NONE', TTL => '2147483647', BLOCKSIZE =>
'65536',

IN_MEMORY => 'false', BLOCKCACHE => 'true'}]}
```

Block 大小默认是 64KB(or 65535 bytes)。下面是 HFile JavaDoc 中的注释：
“Minimum block size。通常的使用情况下，我们推荐将最小的 block 大小设为 8KB 到 1MB。如果文件主要用于顺序访问，应该用大一点的 block 大小。但是，这会导致低效的随机访问(因为有更多的数据需要进行解压)。对于随机访问来说，小一点的 block 大小会好些，但是这可能需要更多的内存来保存 block index，同时可能在创建文件时会变慢(因为我们必须针对每个 data block 进行压缩器的 flush)。

另外，由于压缩编码器的内部缓存机制的影响，最小可能的 block 大小大概是 20KB-30KB”。

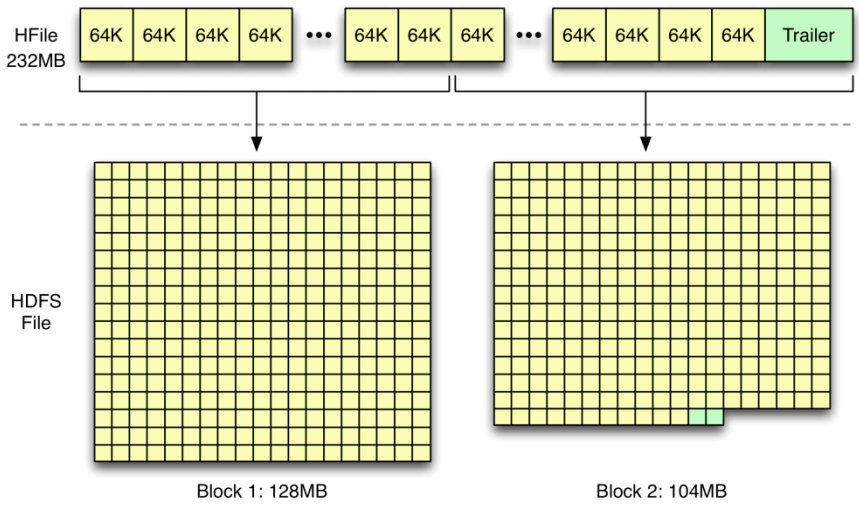
每个block包含一个magic头，一系列序列化的KeyValue实例(具体格式参见 the section called “KeyValue Format”)。在没有使用压缩算法的情况下，每个block的大小大概就等于配置的block size。并不是严格等于，因为writer需要放下用户给的任何大小数据{!如配置的block size可能是 64KB，但是用户给了一条 1MB的记录，writer也得接受它}。即使是对于比较小的值，对于block size大小的检查也是在最后一个value写入后才进行的{!不是写入前检查，而是写入后检查}，所以实际上大部分blocks大小都会比配置的大一些。另一方面，这样做也没什么坏处。

在使用压缩算法的时候，对 block 大小就更没法控制了。如果压缩编码器可以自行选择压缩的数据大小，它可能能获取更好的压缩率。比如将 block size 设为 256KB，使用 LZO 压缩，为了适应于 LZO 内部 buffer 大小，它仍然可能写出比较小的 blocks。

Writer 并不知道用户是否选择了一个压缩算法：它只是对原始数据按照设定的 block 大小限制控制写出。如果使用了压缩，那么实际存储的数据会更小。这意味着对于最终的存储文件来说与不进行压缩时的 block 数量是相同的，但是总大小要小，因为每个 block 都变小了。

你可能还注意到一个问题：默认的 HDFS block 大小是 64MB，是 HFile 默认的 block 大小的 1000 倍。这样，HBase 存储文件块与 Hadoop 的块并不匹配。实际上，两者之间根本没有关系。HBase 是将它的文件透明地存储到文件系统中的，只是 HDFS 也恰巧有一个 blocks。HDFS 本身并不知道 HBase 存储了什么，它看到的只是二进制文件。图 8.6 展示了 HFile 内容如何散布在 HDFS blocks 上。

Figure 8.6. The many smaller HFile blocks are transparently stored in two much larger HDFS blocks



有时候需要绕过 HBase 直接访问 HFile，比如健康检查，dump 文件内容。HFile.main()

提供了一些工具来完成这些事情：

```
$ ./bin/hbase org.apache.hadoop.hbase.io.hfile.HFile

usage: HFile  [-a] [-b] [-e] [-f <arg>] [-k] [-m] [-p] [-r
<arg>] [-v]

 -a,--checkfamily    Enable family check

 -b,--printblocks    Print block index meta data

 -e,--printkey       Print keys

 -f,--file <arg>     File to scan. Pass full-path; e.g.

                  hdfs://a:9000/hbase/.META./12/34

 -k,--checkrow       Enable row order check; looks for
out-of-order keys

 -m,--printmeta      Print meta data of file

 -p,--printkv        Print key/value pairs

 -r,--region <arg>   Region to scan. Pass region name; e.g.
'.META.,,1'

 -v,--verbose        Verbose output; emits file and meta data
delimiters
```

Here is an example of what the output will look like (shortened):

```
$ ./bin/hbase org.apache.hadoop.hbase.io.hfile.HFile -f \

/hbase/testtable/de27e14ffc1f3fff65ce424fcf14ae42/colfam1
/2518469459313898451 \

-v -m -p

Scanning ->
/hbase/testtable/de27e14ffc1f3fff65ce424fcf14ae42/colfam1
/2518469459313898451

K: row-550/colfam1:50/1309813948188/Put/vlen=2 V: 50

K: row-550/colfam1:50/1309812287166/Put/vlen=2 V: 50
```

```
K: row-551/colfam1:51/1309813948222/Put/vlen=2 V: 51

K: row-551/colfam1:51/1309812287200/Put/vlen=2 V: 51

K: row-552/colfam1:52/1309813948256/Put/vlen=2 V: 52

...

K: row-698/colfam1:98/1309813953680/Put/vlen=2 V: 98

K: row-698/colfam1:98/1309812292594/Put/vlen=2 V: 98

K: row-699/colfam1:99/1309813953720/Put/vlen=2 V: 99

K: row-699/colfam1:99/1309812292635/Put/vlen=2 V: 99

Scanned kv count -> 300

Block index size as per heapsize: 208

reader=/hbase/testtable/de27e14ffc1f3fff65ce424fcf14ae42/
colfam1/ \

2518469459313898451, compression=none, inMemory=false, \

firstKey=row-550/colfam1:50/1309813948188/Put, \

lastKey=row-699/colfam1:99/1309812292635/Put,
avgKeyLen=28, avgValueLen=2, \

entries=300, length=11773

fileinfoOffset=11408, dataIndexOffset=11664,
dataIndexCount=1, \

metaIndexOffset=0, metaIndexCount=0, totalBytes=11408,
entryCount=300, \

version=1

Fileinfo:

MAJOR_COMPACTION_KEY = \xFF

MAX_SEQ_ID_KEY = 2020

TIMERANGE = 1309812287166....1309813953720

hfile.AVG_KEY_LEN = 28
```

```
hfile.AVG_VALUE_LEN = 2

hfile.COMPARATOR =
org.apache.hadoop.hbase.KeyValue$KeyComparator

hfile.LASTKEY =
\x00\x07row-699\x07colfam199\x00\x00\x010\xF6\xE5|\x1B\x0
4

Could not get bloom data from meta block
```

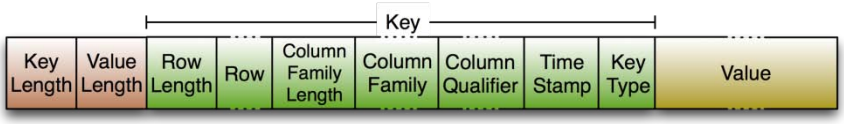
第一部分是序列化的 KeyValue 实例的实际数据。第二部分除了 trailer block 的细节信息外，还 dump 出了内部的 HFile.Reader 属性。最后一部分，以”FileInfo”开头的，是 file info block 的值。

提供的这些信息是很有价值的，比如可以确定一个文件是否进行了压缩，采用的压缩方式。它也能告诉用户存储了多少个 cell，key 和 value 的平均大小是多少。在上面的例子中，key 的长度比 value 的长度大很多。这是由于 KeyValue 类存储了很多额外数据，下面会进行解释。

3.7. KeyValue 格式

实际上 HFile 中的每个 KeyValue 就是一个简单的允许对内部数据进行 zero-copy 访问的底层字节数组，包含部分必要的解析。图 8.7 展示了内部的数据格式。

Figure 8.7. The KeyValue format



该结构以两个标识了key和value部分的大小的定长整数开始。通过该信息就可以在数组内进行一些操作，比如忽略key而直接访问value。如果要访问key部分就需要进一步的信息。一旦解析成一个KeyValue Java实例，用户就可以对内部细节信息进行访问，参见 the section called “The KeyValue Class”。

在上面的例子中 key 之所以比 value 长，就是由于 key 所包含的这些 fields 造成的：它包含一个 cell 的完整的各个维度上的信息：row key，column family name，column qualifier 等等。在处理小的 value 值时，要尽量让 key 很小。选择一个短的 row 和 column key(1 字节 family name，同时 qualifier 也要短)来控制二者的大小比例。

另一方面，压缩也有助于缓解这种问题。因为在有限的数据窗口内，如果包含的

都是很多重复性的数据那么压缩率会比较高。同时因为存储文件中的 KeyValue 都是排好序的，这样就可以让类似的 key 靠在一起(在使用多版本的情况下，value 也是这样的，多个版本的 value 也会是比较类似的)。

4. Write-Ahead Log

Region servers在未收集到足够数据flush到磁盘之前，会一直把它保存在内存中，这主要是为了避免产生太多的小文件。当数据驻留在内存中的时候，它就是不稳定的，比如可能会在服务器发生供电问题时而丢失。这是一个很典型的问题，参见 the section called “Seek vs. Transfer”。

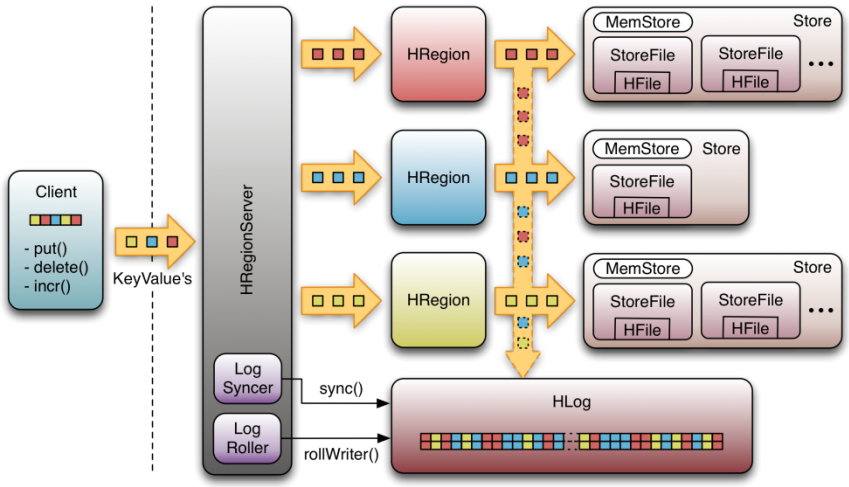
解决这个问题的通常采用 write-ahead logging 策略：每次更新(也称为”edit”)之前都先写到一个 log 里，只有当写入成功后才通知客户端该操作成功了。之后，服务端就可以根据需要在内存中对数据进行随意地进行批处理或者是聚合。

4.1. 概览

WAL 是灾难发生时的救生索。与 MySQL 中的 binary log 类似，它会记录下针对数据的所有变更。在主存产生问题的时候这是非常重要的。如果服务器 crash 了，它就可以通过重放日志让一切恢复到服务器 crash 之前的那个状态。同时这也意味着如果在记录写入到 WAL 过程中失败了，那么整个操作也必须认为是失败的。

本节主要解释下 WAL 如何融入到整个 HBase 的架构中。实际上同一个 region server 持有的所有 regions 会共享同一个 WAL，这样对于所有的修改操作来说它提供了一个集中性的 logging 支持。图 8.8 展示了这些修改操作流在 memstores 和 WAL 之间的交互。

Figure 8.8. All modifications are first saved to the WAL, then passed on the memstores



过程如下：首先客户端发起数据修改动作，比如产生一个 put()，delete()及 increment()调用(后面可能简写为 incr())。每个修改操作都会被包装为一个 KeyValue 对象实例，然后通过 RPC 调用发送出去。该调用(理想情况下是批量进行地)到达具有对应 regions 的那个 HRegionServer。

一旦 KeyValue 实例到达，它们就会被发送到给定的行所对应的 HRegion。数据就会被写入 WAL，然后被存入相应的 MemStore 中。这就是大体上的 HBase 的一个 write path。

最终，当 memstore 的达到一定大小后，或者过了特定时间段后，数据就会异步地持久化到文件系统中。在此期间数据都是保存在内存中的。WAL 可以保证数据不会丢失，即使是在服务端完全失败的情况下。需要提醒一下，log 实际上是存储在 HDFS 上的，任何其他的服务端都可以打开该日志然后 replay 其中的修改操作—这一切都不需要失败的那台物理服务器的任何参与。

4.2. HLog 类

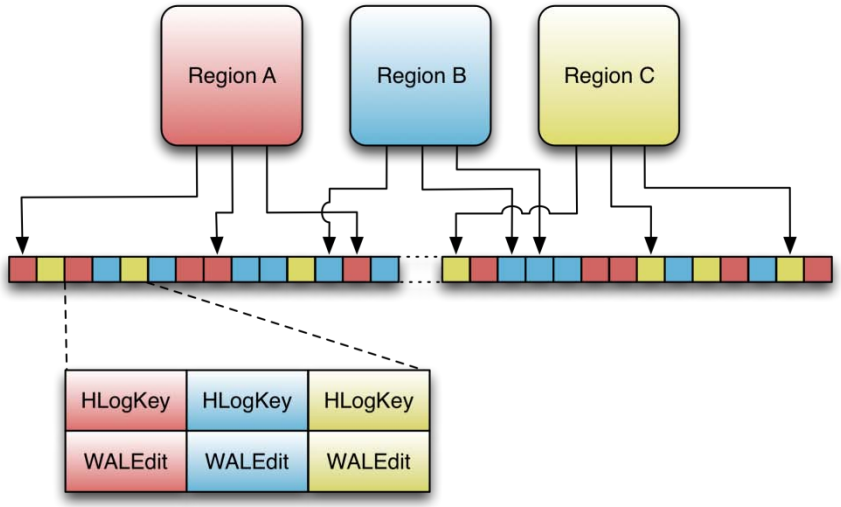
HLog 实现了 WAL。在一个 HRegion 实例化时，一个唯一的 HLog 实例会被传递给它作为构造函数参数。当一个 region 接收到更新操作时，它就可以将数据直接交给共享的 WAL 实例。

HLog 类的核心功能部分就是 append()函数，它内部又调用了 doWrite()。需要注意的是，为了性能方面的考虑，可以针对 Put，Delete 和 Increment 操作设置一个选项，通过：setWriteToWAL(false)。在执行 Put 时，如果调用了该方法，那么将不会进行 WAL 的写入。这也是上面图中指向 WAL 的朝下箭头采用了虚线形式的原因。毫无疑问，默认情况下你肯定是需要写入 WAL 的。但是比如说，你可能正在运行一个大批量数据导入的 MapReduce job，同时也可以在任何时刻重新执行它。为了性能考虑你可以关闭 WAL 写入，但是需要额外保证导入过程中不会丢失数据。

警告：强烈建议你不要轻易地关闭 WAL 的写入。如果你这样做了，那么丢数据只是早晚的事情。而且，HBase 也没法恢复那些此前未写入到日志的丢失数据。

HLog 的另一个重要功能是对变更的追踪。这是通过使用一个序列号做到的。它采用一个线程安全的内部 AtomicLong 变量，该变量要么从 0 开始，要么从已经持久化到文件系统中的最后的那个序列号开始：当 region 打开它对应的存储文件时，它会读取存在 HFile 的 meta 域里的最大序列号字段，如果它大于此前记录下的序列号，就会将它设置为 HLog 的序列号。这样，当打开所有的存储文件后，HLog 的序列号就代表了当前序列化到的位置。

Figure 8.9. The WAL saves edits in the order they arrive, spanning all regions of the same server



上图代表了同一个region server上的三个不同的regions，每个 region都包含了不同的row key range。这些regions共享同一个HLog实例。同时数据是按到达顺序写入WAL的。这就意味着在log需要进行replay时，需要做一些额外的工作(见 the section called “Replay”)。这样做的原因是replay通常很少发生，同时也是为了实现顺序化存储优化，以得到最好的IO性能。

4.3. HLogKey 类

当前的 WAL 实现采用了 Hadoop SequenceFile，它会将记录存储为一系列的 key/values。对于 WAL 来说，value 通常是客户端发送来的修改。Key 是通过一个 HLogKey 实例表示的：因为 KeyValue 仅表示了 row key，column family，column qualifier，timestamp，type 和 value；这样就需要有地方存放 KeyValue 的归属信息，比如 region 和 table 名称。这些信息会被存储在 HLogKey 中。同时上面的序列号也会被存进去，随着记录的加入，该数字会不断递增，以保存修改操作的一个顺序。

它也会记录写入时间，一个用来代表修改操作何时写入 log 的时间戳。最后，它还存储了 cluster ID，以满足用户多集群间的复制需求。

4.4. WALEdit 类

客户端发送的每个修改操作都是通过一个 WALEdit 实例进行包装。它主要用来保证 log 级的原子性。假设你正在更新某一行的十个列，每列或者说是每个 cell，都有自己的一个 KeyValue 实例。如果服务器只将它们中的五个成功写入到 WAL 然后失败了，这样最后你只得到了该变更操作的一半结果。

通过将针对多个 cells 的更新操作包装到一个单个 WALEdit 实例中，将所有的更新看做是一个原子性的操作。这样这些更新操作就是通过单个操作进行写入的了，就保证了 log 的一致性。

注：在 0.90.x 之前，HBase 确实是将这些 KeyValue 实例分别保存的。

4.5. LogSyncer 类

Table descriptor允许用户设置一个称为log flush延迟的flag，参见 the section called “Table Properties”。该值默认是false，意味着每次当一个修改操作发送给服务器的时候，它都会调用log writer的sync()方法。该调用会强制将日志更新对于文件系统可见，这样用户就得到了持久性保证。

不幸的是，该方法的调用涉及到对 N 个服务器的(N 代表了 write-ahead log 的副本数)流水线写操作。因为这是一个开销相当大的操作，因此提供给用户可以轻微地延迟该调用的机会，让它通过一个后台进程执行。需要注意的是，如果不进行 sync()调用，在服务器失败时就有可能丢数据。因此，使用该选项时一定要小心。

流水线 vs. n-路写入

当前的 sync()实现是一个流水线式的写入，这意味着当修改操作被写入时，它会首先被第一个 data node 进行持久化。成功之后，它会被该 data node 发送给另一个 data node，如此循环下去。只有当所有的三个确认了该写操作后，客户端才能继续执行。

将修改操作进行持久化的另一种方式是使用 n-路写，该写入请求同时发送给三个机器。当所有机器确认后，客户端再继续。

不同之处在于，流水线写需要更多的时间来完成，因此具有更高的延迟。但是它可以更好地利用网络带宽。n-路写具有低延迟，因为客户端只需要等待最慢的那个 data node 的确认。但是所有的写入者需要共享发送端的网络带宽，对于一个高负载系统来说这会是一个瓶颈。

目前 HDFS 已经在开展某些工作以同时支持这两种方式。这样用户就可以根据自己应用的特点选择性能最好的那种方式。

将 log flush 延迟 flag 设为 true，会导致修改操作缓存在 region server，同时 LogSyncer 会作为一个服务端线程负责每隔很短时间段调用 sync()方法，该时间段默认是 1 秒钟，可以通过 hbase.regionserver.optionallogflushinterval 配置。

需要注意的是：只对用户表应用它，所有的元数据表必须都是立即 sync 的。

4.6. LogRoller 类

写入的 log 有一个大小限制。LogRoller 类作为一个后台线程运行，负责在特定的区间内 rolling log 文件。该区间由 hbase.regionserver.logroll.period 控制，默认设

为 1 小时。

每 60 分钟当前 log 会被关闭，启动一个新的。随着时间的推移，系统积累的 log 文件数也在不断增长，这也是需要进行维护的。HLog.rollWriter()方法会被 LogRoller 调用已完成上面的当前 log 文件的切换。后面的会通过调用 HLog.cleanOldLogs()完成。

它会检查写入到存储文件中的最大序列号，因为在此序列号之前的都被持久化了。然后在检查是否有些 log 文件，它们的修改操作的序列号都小于这个序列号。如果有这样的 log 文件，它就把它们移入到.oldlogs 目录，只留下那些还需要的 log 文件。

注意，你可能在 log 中看到如下晦涩的消息

```
2011-06-15 01:45:48,427 INFO
org.apache.hadoop.hbase.regionserver.HLog: \

  Too many hlogs: logs=130, maxlogs=96; forcing flush of 8
region(s):


testtable,row-500,1309872211320.d5a127167c6e2dc5106f066cc
84506f8., ...
```

上面信息是因为当前未被持久化的 log 文件数已经超过了配置的可持有的最大 log 文件数。发生这种情况，可能是因为用户文件系统太忙了，导致数据增加的速度超过了数据持久化的速度。此外，memstore 的 flush 也与之相关。

当该信息出现时，系统将会进入一种特殊工作模式，以尽力将修改操作持久化以降低需要保存的 log 文件数。

控制 log rolling 的其他参数还有：
hbase.regionserver.hlog.blocksize(设定为文件系统默认的 block 大小，或者是 fs.local.block.size，默认是 32MB)和 hbase.regionserver.logroll.multiplier(设为 0.95)，当日志到达 block size 的 95%时会开始 rotate。这样，当日志在填满或者达到固定时间间隔后会进行切换。

4.7. Replay

Master 和 region servers 需要小心地进行 log 文件的处理，尤其是在进行服务器错误恢复时。WAL 负责安全地保存好各种操作日志，将它进行 replay 以恢复到一致性状态是一个更复杂的过程。

4.7.1. Single Log

因为每个 region server 上的所有修改操作都是写入到同一个基于 HLog 的 log 文件内的，你可能会问：为什么这样做呢？为什么不是为每个 region 单独创建一个它自己的 log 文件呢。下面是引用自 Bigtable 论文中的内容：

如果为每个 tablet 保存一个单独的 commit log，那么在 GFS 上将会有大量的文件被并发地写入。由于依赖于每个 GFS server 上的底层的文件系统，这些写入会因为需要写入到不同的物理日志文件产生大量的磁盘 seek 操作。

HBase 因为相同的原因而采取了类似策略：同时写入太多文件，再加上 log 的切换，这会降低可扩展性。所以说这个设计决定源自于底层文件系统。尽管可以替换 HBase 所依赖的文件系统，但是最常见的配置都是采用的 HDFS。

目前为止，看起来好像没什么问题。当出错的时候，问题就来了。只要所有的修改操作都及时地完成，数据安全地被持久化了，一切都很顺利。但是在服务器 crash 时，你就不得不将 log 切分成合适的片段。但是因为所有的修改操作掺杂在一块也根本没有什么索引。这样 master 必须等待该 crash 掉的服务器的所有日志都分离完毕后才能重新部署它上面的 region。而日志的数量可能会非常大，中间需要等待的时间就可能非常长。

4.7.2. Log Splitting

通常有两种情况日志文件需要进行 replay：当集群启动时，或者当服务器出错时。当 master 启动—(备份 master 转正也包括在内)—它会检查 HBase 在文件系统上的根目录下的.logs 文件是否还有一些文件，目前没有安排相应的 region server。日志文件名称不仅包含了服务器名称，而且还包含了该服务器对应的启动码。该数字在 region server 每次重启后都会被重置，这样 master 就能用它来验证某个日志是否已经被抛弃。

Log被抛弃的原因可能是服务器出错了，也可能是一个正常的集群重启。因为所有的region servers在重启过程中，它们的log文件内容都有可能未被持久化。除非用户使用了graceful stop(参见 the section called “Node Decommission”)过程，此时服务器才有机会在停止运行之前，将所有pending的修改操作flush出去。正常的停止脚本，只是简单的令服务器失败，然后在集群重启时再进行log的replay。如果不这样的话，关闭一个集群就可能需要非常长的时间，同时可能会因为memstore的并行flush引起一个非常大的IO高峰。

Master 也会使用 ZooKeeper 来监控服务器的状况，当它检测到一个服务器失败时，在将它上面的 regions 重新分配之前，它会立即启动一个所属它 log 文件的恢复过程，这发生在 ServerShutdowHandler 类中。

在 log 中的修改操作可以被 replay 之前，需要把它们按照 region 分离出来。这个

过程就是 log splitting：读取日志然后按照每条记录所属的 region 分组。这些分好组的修改操作将会保存在目标 region 附近的一个文件中，用于后续的恢复。

Logs splitting 的实现在几乎每个 HBase 版本中都有些不同：早期版本通过 master 上的单个进程读取文件。后来对它进行了优化改成了多线程的。0.92.0 版本中，最终引入了分布式 log splitting 的概念，将实际的工作从 master 转移到了所有的 region servers 中。

考虑一个具有很多 region servers 和 log 文件的大集群，在以前 master 不得不自个串行地去恢复每个日志文件—不仅 IO 超载而且内存使用也会超载。这也意味着那些具有 pending 的修改操作的 regions 必须等到 log split 和恢复完成之后才能被打开。

新的分布式模式使用 ZooKeeper 来将每个被抛弃的 log 文件分配给一个 region server。同时通过 ZooKeeper 来进行工作分配，如果 master 指出某个 log 可以被处理了，这些 region servers 为接受该任务就会进行竞争性选举。最终一个 region server 会成功，然后开始通过单个线程(避免导致 region server 过载)读取和 split 该 log 文件。

注：可用通过设置 hbase.master.distributed.log.splitting 来关闭这种分布式 log splitting 方式。将它设为 false，就是关闭，此时会退回到老的那种直接由 master 执行的方式。在非分布式模式下，writers 是多线程的，线程数由 hbase.regionserver.hlog.splitlog.writer.threads 控制，默认设为 3。如果要增加线程数，需要经过仔细的权衡考虑，因为性能很可能受限于单个 log reader 的性能限制。

Split 过程会首先将修改操作写入到 HBase 根文件夹下的 splitlog 目录下。如下：

```
   0 /hbase/.corrupt

     0
/hbase/splitlog/foo.internal,60020,1309851880898_hdfs%3A%
2F%2F \

localhost%2Fhbase%2F.logs%2Ffoo.internal%2C60020%2C130985
0971208%2F \

foo.internal%252C60020%252C1309850971208.1309851641956/te
sttable/ \

d9ffc3a5cd016ae58e23d7a6cb937949/recovered.edits/00000000
00000002352
```

为了与其他日志文件的 split 输出进行区分，该路径已经包含了日志文件名，因

该过程可能是并发执行的。同时路径也包含了 table 名称，region 名称(hash 值)，以及 recovered.edits 目录。最后，split 文件的名称就是针对相应的 region 的第一个修改操作的序列号。

.corrupt 目录包含那些无法被解析的日志文件。它会受 hbase.hlog.split.skip.errors 属性影响，如果设为 true，意味着当无法从日志文件中读出任何修改操作时，会将该文件移入.corrupt 目录。如果设为 false，那么此时会抛出一个 IOExpectation，同时会停止整个的 log splitting 过程。

一旦 log 被成功的 splitting 后，那么每个 regions 对应的文件就会被移入实际的 region 目录下。对于该 region 来说它的恢复工作现在才就绪。这也是为什么 splitting 必须要拦截那些受影响的 regions 的打开操作的原因，因为它必须要将那些 pending 的修改操作进行 replay。

4.7.3. Edits Recovery

当一个 region 被打开，要么是因为集群启动，要么是因为它从一个 region server 移到了另一个。它会首先检查 recovered.edits 目录是否存在，如果该目录存在，那么它会打开目录下的文件，开始读取文件内的修改操作。文件会根据它们的名称(名称中含有序列号)排序，这样 region 就可以按顺序恢复这些修改操作。

那些序列号小于等于已经序列化到磁盘存储中的修改操作将会被忽略，因为该修改操作已经被 apply 了。其他的修改操作将会被 apply 到该 region 对应的 memstore 中以恢复之前的状态。最后，会将 memstore 的内容强制 flush 到磁盘。

一旦 recovered.edits 中的文件被读取并持久化到磁盘后，它们就会被删除。如果某个文件无法读取，那么会根据 hbase.skip.errors 来确定如何处理：默认值是 false，会导致整个 region 恢复过程失败。如果设为 true，那么该文件会被重命名为原始名称+” .<currentTimeMillis>”。不管是哪种情况，你都需要仔细检查你的 log 文件确认问题产生的原因及如何 fix。

4.8. 持久性

无论底层采用了什么稀奇古怪的算法，用户都希望可以依赖系统来存储他们所有的数据。目前 HBase 允许用户根据需要调低 log flush 的时间或者是每次修改操作都进行 sync。但是当存储数据的 stream 被 flush 后，数据是否真的写入到磁盘了呢？我们会讨论下一些类似于 fsync 类型的问题。当前的 HBase 主要依赖于底层的 HDFS 进行持久化。

比较明确的一点是系统通过 log 来保证数据安全。一个 log 文件最好能在长时间内(比如 1 小时)一直处于打开状态。当数据到达时，一个新的 key/value 对会被写入到 SequenceFile，同时间或地被 flush 到磁盘。但是 Hadoop 并不是这样工作的，它之前提供的 API，通常都是打开一个文件，写入大量数据，立即关闭，然后产

生出一个可供其它所有人读取的不可变文件。只有当文件关闭之后，对其他人来说它才是可见的可读的。如果在写入数据到文件的过程中进程死掉通常都会有数据丢失。为了能够让日志的读取可以读到服务器 crash 时刻最后写入的那个位置，或者是尽可能接近该位置，这就需要一个 feature：append 支持。

插曲：HDFS append，hflush，hsync，sync…

在 HADOOP-1700 就已经提出，在 Hadoop 0.19.0 中，用来解决该问题的代码就已提交。但是实际情况是这样的：Hadoop 0.19.0 里的 append 实现比较糟糕，以至于 hadoop fsck 会对 HBase 打开的那些日志文件向 HDFS 报告一个数据损坏错误。

所以在该问题又在 HADOOP-4379 即 HDFS-200 被重新提出，之后实现了一个 syncFs()函数让对于一个文件的变更更可靠。有段时间我们通过客户端代码来检查 Hadoop 版本是否包含了该 API。后来就是 HDFS-265，又重新回顾了 append 的实现思路。同时也引入了 hsync()和 hflush()两个 syncable 接口。

需要注意的是 SequenceFile.Writer.sync()跟我们这里所说的 sync 方法不是一个概念：SequenceFile 中 sync 是用来写入一个同步标记，用于帮助后面的读取操作或者数据恢复。

HBase 目前会检测底层的 Hadoop 库是否支持 syncFs()或者 hflush()。如果在 log writer 中一个 sync()调用被触发，它就会调用 syncFs()或者 hflush()中的一个方法—或者是不调用任何方法，如果 HBase 工作在一个 non-durable setup 上的话。Sync()将会使用流水式的 write 过程来保证日志文件中的修改操作的持久性。当服务器 crash 的时候，系统就能安全地读取被抛弃的日志文件更新到最后的修改操作。

大体上，在Hadoop 0.21.0 版本之前，经常会碰到数据丢失。具体细节参见 the section called “Hadoop”。

5. Read Path

HBase 中的每个 column family 可能有多个文件，文件中包含实际的 cells 或者是 KeyValue 实例 当 memstore 中积累的更新被 flush 到磁盘上时这些文件就会创建出来。负责 compaction 的后台线程会通过将小文件合并成更大的文件来将文件数控制在一定水平上。Major compaction 最终会将所有的文件集合压缩成一个，之后随着 flush 的进行，小文件又会出现。

因为所有的存储文件都是不可变的，所以就没法直接将一个值从它们里面删除，也没法对某个值进行覆盖。而只能通过写入一个墓碑式的标记，来代表某个 cell 或者某几个 cell 或者是整行都被删除了。

假设今天你在给定的一行里写了一个列，之后你一直不断的添加数据，那么你可

能会为该行写入另一个列。问题是，假设最初的列值已经被持久化到了磁盘中，而新写入的列还在 memstore 中，或者已经被 flush 到了磁盘，那该行到底算存放到哪呢？换句话说，当你对该行执行一个 get 命令时，系统怎么知道该返回什么内容？作为一个客户端，你可能希望返回所有的列—看起来它们好像就是一个实体一样。但是实际的数据是存储在独立的 KeyValue 实例中的，而且可能跨越任意数目的存储文件。

如果你删除了最初的那个列值，然后再执行get操作，你希望该值已经不存在了，虽然实际上它还存在于某处，但是墓碑式的标记表明你已经把它删除了。但是该标记很有可能与你要删除的值是分开存储的。关于该架构更细节的内容参见 the section called “Seek vs. Transfer”。

该问题是通过使用 QueryMatcher 以及一个 ColumnTracker 解决的。在读取所有的存储文件以找到一个匹配的记录之前，可能会有一个快速的排除检查，可以使用时间戳或者 Bloom filter 来跳过那些肯定不包含该记录的存储文件。然后，对剩余的存储文件进行扫描以找到匹配该 key 的记录。

为何 Gets 即 Scans

在 HBase 之前的版本中，Get 方法的确是单独实现的。最近的版本进行了改变，目前它内部已经和 Scan API 使用相同的源代码。

你可能会很奇怪，按理来说一个简单的 Get 应该比 Scan 快的。把它们区分对待，更容易针对 Get 进行某些优化。实际上这是由 HBase 本身架构导致的，内部没有任何的索引文件来支持对于某个特定的行或列的直接访问。最小的访问单元就是 HFile 中的一个 block，为了找到被请求的数据，RegionServer 代码和它的底层 Store 实例必须 load 那些可能包含该数据的 blocks 然后进行扫描。实际上这就是 Scan 的操作过程。换句话说，Get 本质上就是对单个行的 Scan，就是一个从 start row 到 start row+1 的 scan。

Scan 是通过 RegionScanner 类实现的，它会每个 Store 实例(每个代表一个 column family)执行 StoreScanner 检索，如果读操作没有包含某个 column family，那么它的 Store 实例就会被略过。

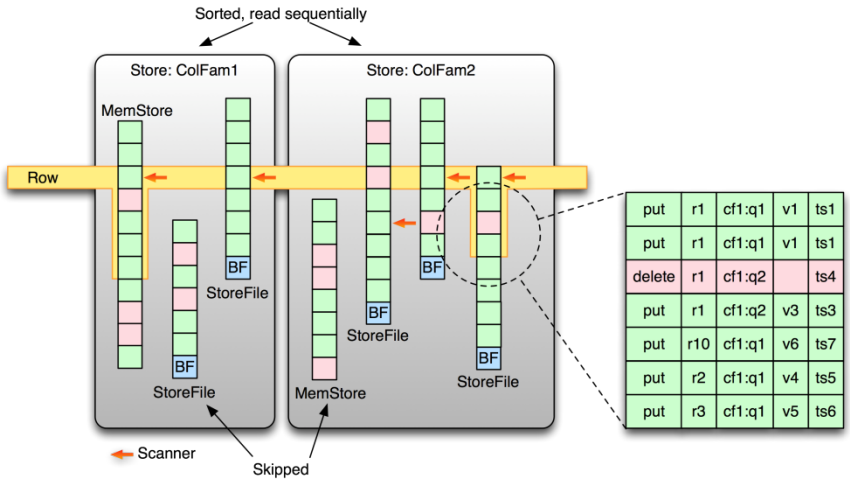
StoreScanner 会合并它所包含的存储文件和 memstore。同时这也是根据 Bloomfilter或者时间戳进行排除性检查的时候，然后你可以跳过那些不需要的存储文件。参见 the section called “Key Design”了解排除性检查的细节，以及如何利用它。

同时也是由 StoreScanner 持有 QueryMatcher(这里是 ScanQueryMatcher 类)。它会记录下那些包含在最终结果中的 KeyValue。

RegionScanner 内部会使用一个 KeyValueHeap 类来按照时间戳顺序安排所有的 Store scanners。StoreScanner 也会采用相同的方式来对存储文件进行排序。这就保证了用户可以按照正确的顺序进行 KeyValue 的读取(比如根据时间戳的降序)。

在 store scanners 被打开时，它们会将自己定位到请求的 row key 处。准备进行数据读取。

Figure 8.10. Rows are stored and scanned across different stores, on-disk or in-memory



对于一个 get()调用，所有的服务器需要做的就是调用 RegionScanner 的 next()。该调用内部会读取组成结果的所有内容。包括所有请求的版本，假设某列有三个版本，同时用户请求检索它们中所有的。这三个 KeyValue 可能分布在磁盘或内存中的存储文件。Next()调用会从所有的存储文件中读取直到读到下一行，或者直到读到足够的版本。

与此同时，它也会记录那些删除标记。当它扫描当前行的 KeyValue 时，可能会碰到这些删除标记，那些时间戳小于等于该删除标记的记录都会被认为是已经清除掉了。

图中展示了一个由一系列 KeyValue 组成的逻辑行，某些存储在相同的存储文件中，某些在其他文件上，包含了多个 column family。由于时间戳或者 Bloom filter 的排除过程，某些存储文件和 memstore 可能会被跳过。最后一个存储文件中的删除标记可能会遮蔽掉所有的记录，但是它们仍然是同一行的一部分。这些 scanners—实际上可以用一系列指向存储文件的箭头表示—要么指向文件中的第一个匹配点，要么是紧挨着所请求的 key 的那个点(如果没有直接匹配的点的话)。

在执行 next 调用时，只有那些具有匹配点的 scanners 才会被考虑。内部循环会从第一个存储文件到最后一个存储文件，按照时间地降序一个挨一个地读取其中的 KeyValue，直到超出当前请求的 key。

对于 scan 操作，则是通过在 ResultScanner 上不断的调用 next()，直到碰到表的结束行或者为当前的 batch 读取了足够多的行时。

最终的结果是一个匹配了给定的 get 或者 scan 操作的 KeyValue 的列表。它会被

发送给客户端，客户端就可以使用 API 函数来访问里面的列。

6. Region 查找

为了让客户端能够找到持有特定的 row key range 的 region server，HBase 提供了两个特殊的元数据表：-ROOT-和.META.。

-ROOT-表用于保存.META.表的所有 regions 的信息。HBase 认为只有一个 root region，同时它永不会被 split，这样就可以保证一个三层的类 B+树查找模式：第一层是存储在 ZooKeeper 上的一个保存了 root 表的 region 信息的节点，换句话说就是保存了 root region 的那个 region server 的名称。第二层需要到-ROOT-表中查找匹配的 meta region，然后第三层就是到.META.表中检索用户表的 region 信息。

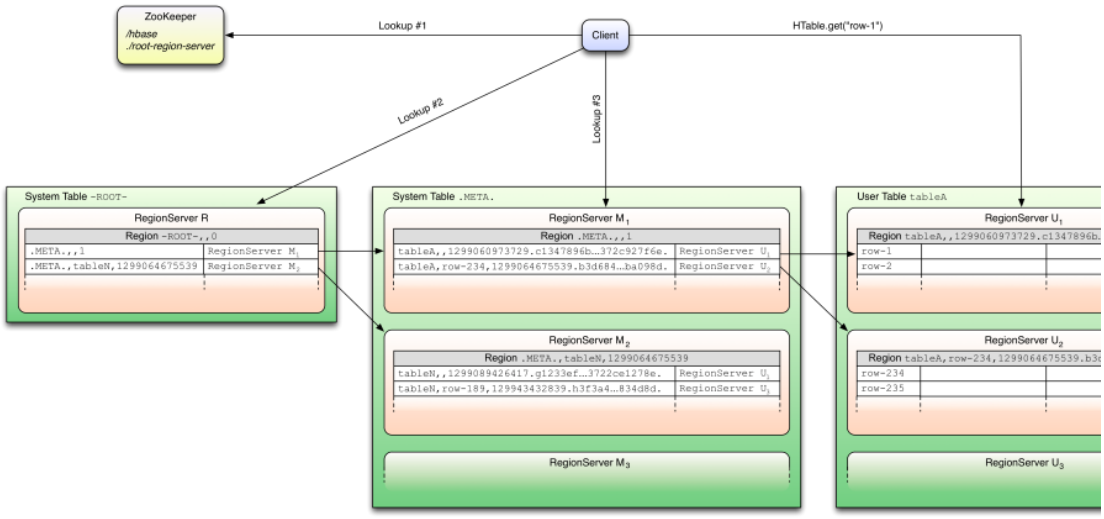
元数据表中的 row key 由每个 region 的表名，起始行，及一个 ID(通常使用当前时间，单位是毫秒)。从 HBase 0.90.0 开始，这些 key 可能会有一个额外的与之关联的 hash 值。目前只是用于用户表中。

注：Bigtable 论文指出，在.META.表的 region 大小限制在 128MB 的情况下，它可以寻址 2^34 个 regions，如果按每个 region 128MB 大小算，就是 2^61 字节大小。因为 region 大小可以增加而不会影响到定位模式，因此根据需要这个值还可以增大。

尽管客户端会缓存 region 位置信息，但是客户端在首次查询时都需要发送请求来查找特定 row key 或者一个 region 也可能会被 split，merge 或者移动，这样 cache 可能会无效。客户端库采用一种递归的方式逐层向上地找到当前的信息。它会询问与给定的 row key 匹配的.META.表 region 所属的 region server 地址。如果信息是无效的，它就退回到上层询问 root 表对应的.META. region 的位置。最后，如果也失败了，它就需要读取 Zookeeper 节点以找到 root 表 region 的位置。

最坏情况下，将会需要 6 次网络传输才能找到用户region，因为无效记录只有当查找失败时才能发现出来，当然系统假设这种情况并不经常发生。在缓冲为空的情况下，客户端需要三次网络传输来完成缓存更新。一种降低这种网络传输次数的方法是对位置信息进行预取，提前更新客户端缓存。具体细节见 the section called “Miscellaneous Features”。

Figure 8.11. Starting with an empty cache, the client has to do three lookups.



一旦用户表 region 已知之后，客户端就可以直接访问而不需要进一步的查找。图中对查找进行了标号，同时假设缓存是空的。

6.1. Region 生命周期

Region 的状态会被 master 追踪，通过使用 AssignmentManager 类。它会记下 region 从 offline 状态开始的整个生命周期。表 8.1 列出了一个 region 的所有可能状态。

Table 8.1. Possible states of a region

State	Description
Offline	The region is offline.
Pending Open	A request to open the region was sent to the server.
Opening	The server has started opening the region.
Open	The region is open and fully operational.
Pending Close	A request to close the region has been sent to the server.
Closing	The server is in the process of closing the region.
Closed	The region is closed.
Splitting	The server started splitting the region.

State	Description
Split	The region has been split by the server.

状态间的转换可能是由 master 引起，也可能是由持有它的那个 region server 引起。比如 master 可能将 region 分配给某个 server，之后它会由该 server 打开。另一方面，region server 可能会启动 split 过程，这会触发 region 打开和关闭事件。

由于这些事件本身的分布式属性，服务器使用 ZooKeeper 在一个专门的 znode 中记录各种状态。

7. ZooKeeper

从 0.20.x 开始，HBase 使用 ZooKeeper 作为它的分布式协调服务。包括 region servers 的追踪，root region 的位置及其他一些方面。0.90.x 版引入了新的 master 实现，与 ZooKeeper 有了更紧密的集成。它使得 HBase 可以去除掉 master 和 region servers 之间发送的心跳信息。这些现在都通过 ZooKeeper 完成了，当其中的某一部分发生变化时就会进行通知，而之前是通过固定的周期性检查完成。

HBase 会在它的根节点下创建一系列的 znodes。根节点默认是"/hbase"，可以通过 zookeeper.znode.parent 进行配置。下面是所包含的 znodes 节点列表及其功用：

注：下面的例子使用了 ZooKeeper 命令行接口(简称 CLI)来运行这些命令。可以通过如下命令启动 CLI：

```
$ $ZK_HOME/bin/zkCli.sh -server <quorum-server>
```

/hbase/hbaseid

包含了集群ID，跟存储在HDFS上的 hbase.id文件中的一致. 如下:

```
[zk: localhost(CONNECTED) 1] get /hbase/hbaseid

e627e130-0ae2-448d-8bb5-117a8af06e97
```

/hbase/master

包含了服务器名称, (具体参见 the section called “Cluster Status Information”). 如下:

```
[zk: localhost(CONNECTED) 2] get /hbase/master

foo.internal,60000,1309859972983
```

/hbase/replication

包含了replication的细节信息。相关细节参见 the section called “Internals”。

/hbase/root-region-server

包含了持有 -ROOT- regions 的region server的服务器名称。在region 查找过程中会用到它 (见 the section called “Region Lookups”). 如下:

```
[zk: localhost(CONNECTED) 3] get
/hbase/root-region-server

rs1.internal,60000,1309859972983
```

/hbase/rs

作为所有 region servers 的根节点，会记录它们是何时启动。用来追踪服务器的失败。每个内部的 znode 节点是临时性的，以它所代表的 region server 的服务器名称为名。比如:

```
[zk: localhost(CONNECTED) 4] ls /hbase/rs

[rs1.internal,60000,1309859972983,rs2.internal,60000
,1309859345233]
```

/hbase/shutdown

该节点用于追踪集群状态。包含集群启动时间，当集群关闭时其内容为空。比如:

```
 [zk: localhost(CONNECTED) 5] get /hbase/shutdown
```

```
Tue Jul 05 11:59:33 CEST 2011
```

/hbase/splitlog

用于所有log splitting相关协调的parent znode，细节详见 the section called “Log Splitting” 。比如：

```
[zk: localhost(CONNECTED) 6] ls /hbase/splitlog

[hdfs%3A%2F%2Flocalhost%2Fhbase%2F.logs%2Ffoo.intern
al%2C60020%2C \

1309850971208%2Ffoo.internal%252C60020%252C130985097
1208.1309851636647,

hdfs%3A%2F%2Flocalhost%2Fhbase%2F.logs%2Ffoo.interna
l%2C60020%2C \

1309850971208%2Ffoo.internal%252C60020%252C130985097
1208.1309851641956,

...

hdfs%3A%2F%2Flocalhost%2Fhbase%2F.logs%2Ffoo.interna
l%2C60020%2C \

1309850971208%2Ffoo.internal%252C60020%252C130985097
1208.1309851784396]



[zk: localhost(CONNECTED) 7] get /hbase/splitlog/ \

\hdfs%3A%2F%2Flocalhost%2Fhbase%2F.logs%2Fmemcache1.
internal%2C \

60020%2C1309850971208%2Fmemcache1.internal%252C60020
%252C1309850971208. \

1309851784396

unassigned foo.internal,60000,1309851879862



[zk: localhost(CONNECTED) 8] get /hbase/splitlog/ \
```

```
\hdfs%3A%2F%2Flocalhost%2Fhbase%2F.logs%2Fmemcache1.
internal%2C \

60020%2C1309850971208%2Fmemcache1.internal%252C60020
%252C1309850971208. \

1309851784396

owned foo.internal,60000,1309851879862



[zk: localhost(CONNECTED) 9] ls /hbase/splitlog

[RESCAN0000293834,
hdfs%3A%2F%2Flocalhost%2Fhbase%2F.logs%2Fmemcache1. \

internal%2C60020%2C1309850971208%2Fmemcache1.interna
l%252C \

60020%252C1309850971208.1309851681118,
RESCAN0000293827, RESCAN0000293828, \

RESCAN0000293829, RESCAN0000293838, RESCAN0000293837]
```

这些例子列出了很多东西:你可以看到一个未被分配的 log 是如何被 split 的，之后又如何被一个 region server 所拥有。"RESCAN"节点表示那些 workers，比如万一 log split 失败后可能被用于进一步的工作的 region server。

/hbase/table

当一个表被禁用时,它会被添加到该节点下. 表名就是新创建的 znode 的名称，内容就是"DISABLED"。比如:

```
[zk: localhost(CONNECTED) 10] ls /hbase/table

[testtable]

[zk: localhost(CONNECTED) 11] get
/hbase/table/testtable

DISABLED
```

/hbase/unassigned

该 znode 是由 AssignmentManager 用来追踪整个集群的 region 状态的。它包含了那些未被打开或者处于过渡状态的 regions 对应的 znodes，zodes 的名称就是该 region 的 hash。比如:

```
[zk: localhost(CONNECTED) 11] ls /hbase/unassigned

[8438203023b8cbba347eb6fc118312a7]
```

8. Replication

HBase replication 是在不同的 HBase 部署之间拷贝数据的一种方式。它可以作为一种灾难恢复解决方案，也可以用于提供 HBase 层的更高的可用性。同时它也能提供一些更实用的东西：比如，可以作为从面向 web 的集群中拷贝最新的更新内容到 MapReduce 集群的简单方式，然后利用 MapReduce 集群对新老数据进行处理再自动地返回结果。

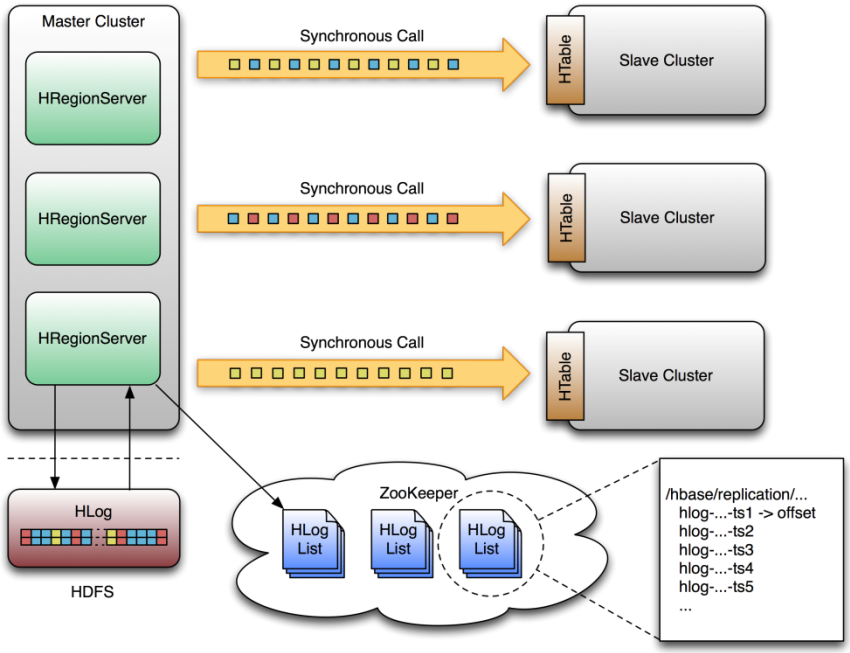
HBase replication 采用的基本架构模式是：master-push；因为每个 region server 都有自己的 write-ahead-log(即 WAL 或 HLog)，这样 就很容易记录下从上次复制之后又发生了什么，非常类似于其他一些著名的解决方案，就像 MySQL 的主从复制就只用了一个 binary log 来进行追踪。一个 master 集群可以向任意数目的 slave 集群进行复制，同时每个 region server 会参与复制它本身所对应的一系列的修改。

Replication 是异步进行的，这意味着参与的集群可能在地理位置上相隔甚远，它们之间的连接可以在某段时间内是断开的，插入到 master 集群中的那些行，在同一时间在 slave 集群上不一定是可用的(最终一致性)。

在该设计中所采用的 replication 格式在原理上类似于 MySQL 的基于状态的 replication。在这里，不是 SQL 语句，而是整个的 WALEdits(由来自客户端的 put 和 delete 操作的多个 cell inserts 组成)会被复制以维持原子性。

每个 region server 的 HLogs 是 HBase replication 的基础，同时只要这些 logs 需要复制到其他的 slave 集群上，它们就需要保存在 HDFS 上。每个 RS 会从它们需要复制的最老的 log 开始读取，同时为简化故障恢复会将当前读取位置保存到 ZooKeeper 上。对于不同的 slave 集群来说，该位置可能是不同的。参与 replication 各集群大小可能不是对称的，同时 master 集群会通过随机化来尽量保证在 slave 集群上的 replication 工作流的平衡。

Figure 8.12. Overview on the replication architecture



8.1. Life of a Log Edit

下面的几节里会描述下来自客户端的一个 edit 从与 master 集群通信开始到到达另一个 slave 集群的整个生命历程。

8.1.1. 正常处理流程

客户端会使用 HBase API 将 Put，Delete 和 Increment 发送给 region server。Key values 会被 region server 转换为一个 WALEdit 对象。该 edit 对象会被 append 到当前的 WAL 上，同时之后会被 apply 到它的 MemStore 中。

通过一个独立的线程，将该 edit 对象从 log 中读出然后只保留那些需要复制的 KeyValues(也就是说只保留那些在 family schema 中只属于 GLOBAL 作用域的 family 的成员，同时是非元数据也就是非.META 和-ROOT-)。当 buffer 被填满或者读取者读到文件末尾后，该 buffer 会被随机发送到 slave 集群上的某个 region server 上。region server 顺序地接受读到的这些 edits，同时将它们按照 table 放到不同的 buffers 中。一旦所有的 edits 读取完毕，所有的 buffer 就会通过正常的 HBase 客户端进行 flush。

回头再看 master 集群的 region server，当前复制到的 WAL 偏移位置会注册到 ZooKeeper 上。

8.1.2. Non-responding Slave Clusters

Edit 会以同样的方式进行插入。在独立的线程中，region server 像正常处理过程那样进行读取，过滤以及对 log edits 进行缓存。假设现在所联系的那个 slave 集群的 region server 不再响应 RPC 了，这样 master 集群的 region server 会进行休眠然后等待一个配置好的时间段后再进行重试。如果 slave 集群的 region server 仍然没有响应，master 集群的 region server 就会重新选择一个要复制到的 region server 子集，然后会重新尝试发送缓存的那些 edits。

与此同时，WALs 将会进行切换同时会被存储在 ZooKeeper 的一个队列中。那些被所属的 region server 归档(归档过程基本上就是把一个日志从它所属的 region server 的目录下移到一个中央的 logs 归档目录下)了的日志会更新它们在复制线程的内存队列中的路径信息。

当 slave 集群最终可用后，处理方式就又跟正常处理流程一致了。Master 集群的 region server 就又开始进行之前积压的日志的复制了。

8.2. Internals

本节会深入描述下 replication 的内部操作机制。

8.2.1. 选择复制到的目标 Region Servers

当一个 master 集群的 region server 开始作为某个 slave 集群的复制源之后，它首先会通过给定的集群 key 联系 slave 集群的 ZooKeeper。
该 key 由如下部分组成：
hbase.zookeeper.quorum
zookeeper.znode.parent
hbase.zookeeper.property.clientPort。
之后，它会扫描/hbase/rs 目录以找到所有可用的 sinks(即那些可用接收用于复制的 edits 数据流的 region servers)同时根据配置的比率(默认是 10%)来选出它们中的一个子集。比如如果 slave 集群有 150 台机器，那么将会有 15 台选定为 master 集群的 region server 将要发送的 edits 的接受者。因为复制过程中，master 的所有 region server 都会进行，这样这些 slave 集群的 region server 的负载就可能会很高，同时该方法适用于各种大小的集群。比如，一个具有 10 台机器的 master 集群向一个具有 10%比率的 5 台集群的 slave 集群进行复制。意味着 master 集群的 region servers 每次都会随机选择一台机器，这样 slave 集群的重叠和总的使用率还是很高的。

8.2.2. 日志追踪

每个master集群的region server在replication znodes体系中都有自己的节点。同时节点下针对每个集群节点还会有一个znode(如果有 5 个slave集群，就会有 5 个znode创建出来)，每个znode下又包含一个待处理的HLogs队列。这些队列是用来追踪由该region server创建的HLogs的，这些队列的大小可能有所不同。比如，如果某个slave集群某段时间不可用，那么这段时间的HLogs就不能被删除，因此它们就得呆在队列里(而其他的可能已经处理过了)。具体例子可以参考：the section called “Region Server Failover”。

当一个 source 被实例化时，它会包含 region server 当前正在写入的 HLog。在 log 切换时，新的文件在可用之前就会被添加到每个 slave 集群的 znode 的队列中。这可以让所有的 sources 在该 HLog 可以 append edits 之前就能够知道一个新 log 已经存在了，但是该操作的开销目前是很昂贵的。当 replication 线程无法从文件中读出更多的记录之后(因为它已经读到了最后一个 block)，就会将它从队列中删除，此时要求队列中的还有其他文件存在{!还有其他文件存在就意味着这个文件是一个已经写完的日志文件，而不是正在写入的那个}。这就意味着如果一个 source 已是最新状态，同时复制进程已经到了 region server 正在写入的那个 log，那么即使读到了当前文件的”end”部分，也不能将它从队列中删除{!如果该文件正在被写入，那么即使读到了末尾，也不能认为它已经结束}。

当一个 log 被归档后(因为它不再被使用或者是因为插入速度超过了 region flushing 的速度导致当前 log 文件数超过了 hbase.regionserver.maxlogs 的限制)，它会通知 source 线程该 log 的路径已经发生改变。如果某个 source 已经处理完该 log，会忽略该消息。如果它还在队列中，该路径会更新到相应的内存中。如果该 log 目前正在被复制，该变更会自动完成，读取者不需要重新打开该被移动的文件。因为文件的移动只是一个 NameNode 操作，如果读取者当前正在读取该 log 文件，它不会产生任何异常。

8.2.3. 读，过滤及发送 Edits

默认情况下，一个 source 会尽量地读取日志文件然后将日志记录尽快地发送给一个 sink。但是首先它需要对 log 记录进行过滤；只有那些具有 GLOBAL 作用域同时不属于元数据表的 KeyValues 才能保留下来。第二个限制是，附加在每个 slave 集群上所能复制的 edits 列表的大小限制，默认是 64MB。这意味着一个具有三个 slave 集群的 master 集群的 region server 最多只能使用 192MB 来存储被复制的数据。

一旦缓存的 edits 大小达到上限或者读取者读到了 log 文件末尾，source 线程将会停止读取然后随机选择一个 sink 进行复制。它会对选定的集群直接产生一个 RPC 调用，然后等待该方法返回。如果成功返回，source 会判断当前的文件是否已经读完或者还是继续从里面读。如果是前者，它会将它从 znode 的队列中删除。如果是后者，它会在该 log 的 znode 中注册一个新的 offset。如果 PRC 抛出了异

常，该 source 在寻找另一个 sink 之前会重试十次。

8.2.4. 日志清理

如果 replication 没有开启，master 的 logs 清理线程将会使用用户配置的 TTL 进行旧 logs 的删除。当使用 replication 时，这样是无法工作的，因为被归档的 log 虽然超过了它们自己的 TTL 但是仍可能在队列中。因此，需要修改默认行为，在日志超出它的 TTL 时，清理线程还要查看每个队列看能否找到该 log，如果找不到就可以将该 log 删除。查找过程中它会缓存它找到的那些 log，在下次 log 查找时，它会首先查看缓存。

8.2.5. Region Server 故障恢复

只要 region servers 没有出错，ZooKeeper 中的日志记录就不需要添加任何值。不幸的是，它们通常都会出错，这样我们就可以借助 ZooKeeper 的高可用性和它的语义来帮助我们管理队列的传输。

master 集群的所有 region servers 相互之间都有一个观察者，当其中一个死掉时，其他的都能得到通知。如果某个死掉后，它们就会通过在死掉的 region server 的 znode(该 znode 也包含它的队列)内创建一个称为 lock 的 znode 来进行竞争性选举。最终成功创建了该 znode 的 region server 会将所有的队列传输到它自己的 znode 下(逐个传输因为 ZooKeeper 并不支持 rename 操作)当传输完成后就会删掉老的那些。恢复后的队列的 znodes 将会在死掉的服务器的名称后加上 slave 集群的 id 来进行命名。

完成之后，master 集群的 region server 会对每个拷贝出的队列创建一个新的 source 线程。它们中的每一个都会遵守 read/filter/ship 模式。主要的区别是这些队列不会再有新数据因为它们不再属于它们的新 region server，同时意味着当读取者到达最后一个日志的末尾时，队列对应的 znode 就可以被删除了，同时 master 集群的 region server 将会关闭那个 replication source。

比如，考虑一个具有 3 个 region servers 的 master 集群，该集群会向一个 id 为 2 的单个 slave 集群进行复制。下面的层次结构代表了 znodes 在某个时间点上的分布。我们可以看到该 region servers 的 znodes 都包含一个具有一个队列的 peers znode。这些队列的 znodes 的在 HDFS 上的实际文件名称具有如下形式”address,port.timestamp”。

```
/hbase/replication/rs/
                  1.1.1.1,60020,123456780/
                    peers/
```

```
                          2/
                            1.1.1.1,60020.1234  (Contains a
position)
                            1.1.1.1,60020.1265
                  1.1.1.2,60020,123456790/
                    peers/
                          2/
                            1.1.1.2,60020.1214  (Contains a
position)
                            1.1.1.2,60020.1248
                            1.1.1.2,60020.1312
                  1.1.1.3,60020,   123456630/
                    peers/
                          2/
                            1.1.1.3,60020.1280  (Contains a
position)
```

现在我们假设 1.1.1.2 丢失了它的 ZK 会话，幸存者将会竞争以创建一个 lock，最后 1.1.1.3 获得了该锁。然后它开始将所有队列传输到它本地的 peers znode，同时在原有的名称上填上死掉的服务器的名称。在 1.1.1.3 清理老的 znodes 之前，节点分布如下：

```
/hbase/replication/rs/
                  1.1.1.1,60020,123456780/
                    peers/
                          2/
                            1.1.1.1,60020.1234  (Contains a
position)
                            1.1.1.1,60020.1265
```

```
                  1.1.1.2,60020,123456790/
                    lock
                    peers/
                         2/
                           1.1.1.2,60020.1214  (Contains a
position)
                           1.1.1.2,60020.1248
                           1.1.1.2,60020.1312
                  1.1.1.3,60020,123456630/
                    peers/
                         2/
                           1.1.1.3,60020.1280  (Contains a
position)


                         2-1.1.1.2,60020,123456790/
                           1.1.1.2,60020.1214  (Contains a
position)
                           1.1.1.2,60020.1248
                           1.1.1.2,60020.1312
```

一段时间后，但在 1.1.1.3 结束来自 1.1.1.2 的最后一个 HLog 的复制之前，我们假设它也死掉了(而且某些之前创建的新 logs 还在正常队列中)。最后一个 region server 会尝试锁住 1.1.1.3 的 znode 然后开始传输所有的队列。新的节点分布如下：

```
/hbase/replication/rs/
                  1.1.1.1,60020,123456780/
                    peers/
```

```
                          2/
                            1.1.1.1,60020.1378  (Contains a
position)


                          2-1.1.1.3,60020,123456630/
                            1.1.1.3,60020.1325  (Contains a
position)
                            1.1.1.3,60020.1401


2-1.1.1.2,60020,123456790-1.1.1.3,60020,123456630/
                            1.1.1.2,60020.1312  (Contains a
position)
                  1.1.1.3,60020,123456630/
                    lock
                    peers/
                          2/
                            1.1.1.3,60020.1325  (Contains a
position)
                            1.1.1.3,60020.1401


                          2-1.1.1.2,60020,123456790/
                            1.1.1.2,60020.1312  (Contains a
position)
```

Replication 目前还是一个处于实验阶段的 feature。在将它应用到你的使用场景中时需要进行仔细地评估。

9. HFile V2

最新版的 HFile 主要做了两个改进，这两个改进主要都是为了降低内存使用和启动时间，思路都是将它们切分为多个 block，填满就写出去，这也降低了 writer 的内存占用：

1. 增加了树状结构的数据块索引(data block index)支持。原因是在数据块的索引很大时，很难全部 load 到内存，比如当前的一个 data block 会在 data block indxe 区域对应一个数据项，假设每个 block 64KB，每个索引项 64Byte，这样如果每条机器上存放了 6TB 数据，那么索引数据就得有 6GB，因此这个占用的内存还是很高的。通过对这些索引以树状结构进行组织，只让顶层索引常驻内存，其他索引按需读取并通过 LRU cache 进行缓存，这样就不需要将全部索引加载到内存。

   相当于把原先平坦的索引结构以树状的结构进行分散化的组织。现在的 index block 也是与 data block 一样是散布到整个文件之中，而不再是单纯的在结尾处。同时为支持对序列化数据进行二分查找，还为非 root 的 block index 设计了新的"non-root index block"。

   具体实现来看，比如在写入 HFile 时，在内存中会存放当前的 inline block index，当 inline block index 大小达到一定阈值(比如 128KB)时就直接 flush 到磁盘，而不再是最后做一次 flush，这样就不需要在内存中一直保持所有的索引数据。当所有的 inline block index 生成之后，HFile writer 会生成更上一级的 block index，它里面的内容就是这些 inline block index 的 offset，依次递归，逐步生成更上层的 block index，上层的包含的就是下层的 offset，直到最顶层大小小于阈值时为止。所以整个过程就是自底向上的通过下层 index block 逐步构建出上层 index block。

2. 之前的 bloom filter 数据是存放在单独的一个 meta block 里，新版里它将可以被存为多个 block。优点类似于第一个改动，这就允许在处理一个查询时不必将所有的数据都 load 到内存。

https://issues.apache.org/jira/browse/HBASE-3857

Build a tree structure data block index inside of the HFile

Add Bloom Block Index Support

---

[83] See "B+ trees" on Wikipedia

[84] See LSM-Tree, O'Neil et al., 1996

[85] From "Open Source Search" by Doug Cutting, Dec. 05, 2005.

[86] See the JIRA issue HADOOP-3315 for details.

[87] For the term itself please read Write-Ahead Logging on Wikipedia.

[88] Subsequently they are referred to interchangeably as *root table* and *meta table* respectively, since for example "-ROOT-" is how the table is actually named in HBase and calling it root table is stating its purpose.

[89] See the online manual for details.