[Skip to main content](http://hbasefly.com/2016/03/25/hbase-hfile/" \l "content)

[有态度的HBase/Spark/BigData](http://hbasefly.com/)

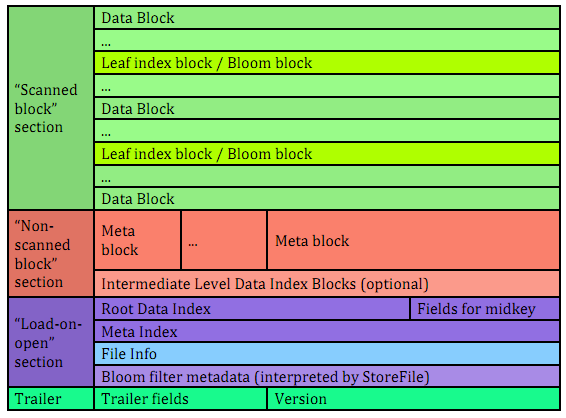
**HBase – 存储文件HFile结构解析**

[2016年3月25日](http://hbasefly.com/2016/03/25/hbase-hfile/)  [范欣欣](http://hbasefly.com/author/libisthanksgmail-com/)  [HBase](http://hbasefly.com/category/hbase/)

HFile是HBase存储数据的文件组织形式，参考BigTable的SSTable和Hadoop的TFile实现。从HBase开始到现在，HFile经历了三个版本，其中V2在0.92引入，V3在0.98引入。HFileV1版本的在实际使用过程中发现它占用内存多，HFile V2版本针对此进行了优化，HFile V3版本基本和V2版本相同，只是在cell层面添加了Tag数组的支持。鉴于此，本文主要针对V2版本进行分析，对V1和V3版本感兴趣的同学可以参考其他信息。

**HFile逻辑结构**

HFile V2的逻辑结构如下图所示：



文件主要分为四个部分：Scanned block section，Non-scanned block section，Opening-time data section和Trailer。

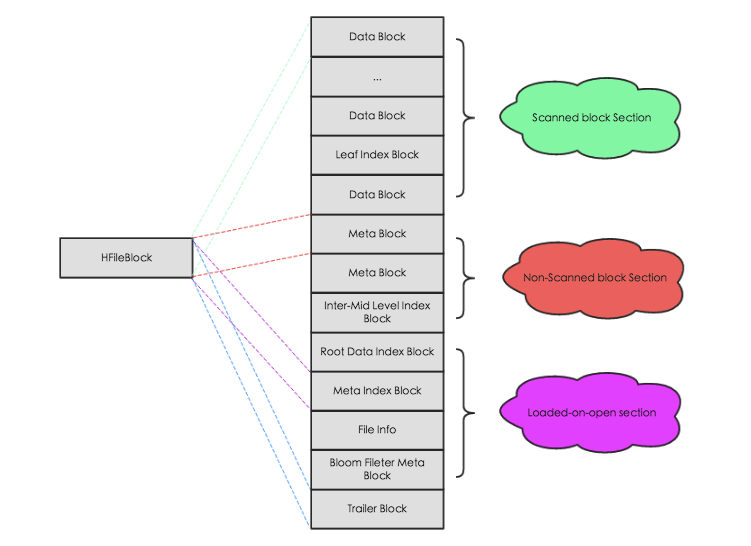
Scanned block section：顾名思义，表示顺序扫描HFile时所有的数据块将会被读取，包括Leaf Index Block和Bloom Block。

Non-scanned block section：表示在HFile顺序扫描的时候数据不会被读取，主要包括Meta Block和Intermediate Level Data Index Blocks两部分。

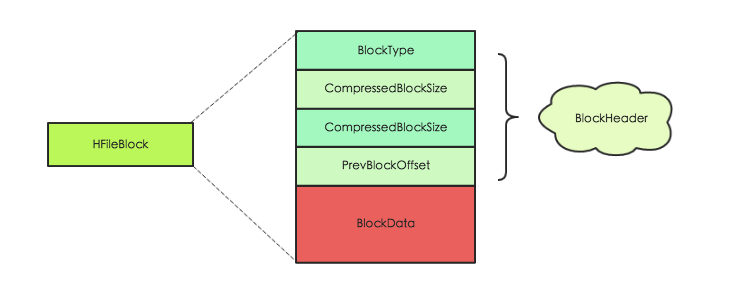
Load-on-open-section：这部分数据在HBase的region server启动时，需要加载到内存中。包括FileInfo、Bloom filter block、data block index和meta block index。

Trailer：这部分主要记录了HFile的基本信息、各个部分的偏移值和寻址信息。

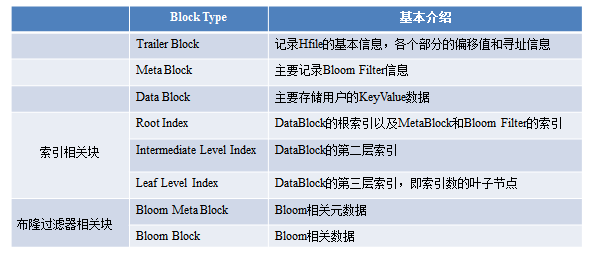
**HFile物理结构**

****

如上图所示， HFile会被切分为多个大小相等的block块，每个block的大小可以在创建表列簇的时候通过参数blocksize ＝> ‘65535’进行指定，默认为64k，大号的Block有利于顺序Scan，小号Block利于随机查询，因而需要权衡。而且所有block块都拥有相同的数据结构，如图左侧所示，HBase将block块抽象为一个统一的HFileBlock。HFileBlock支持两种类型，一种类型不支持checksum，一种不支持。为方便讲解，下图选用不支持checksum的HFileBlock内部结构：



上图所示HFileBlock主要包括两部分：BlockHeader和BlockData。其中BlockHeader主要存储block元数据，BlockData用来存储具体数据。block元数据中最核心的字段是BlockType字段，用来标示该block块的类型，HBase中定义了8种BlockType，每种BlockType对应的block都存储不同的数据内容，有的存储用户数据，有的存储索引数据，有的存储meta元数据。对于任意一种类型的HFileBlock，都拥有相同结构的BlockHeader，但是BlockData结构却不相同。下面通过一张表简单罗列最核心的几种BlockType，下文会详细针对每种BlockType进行详细的讲解：

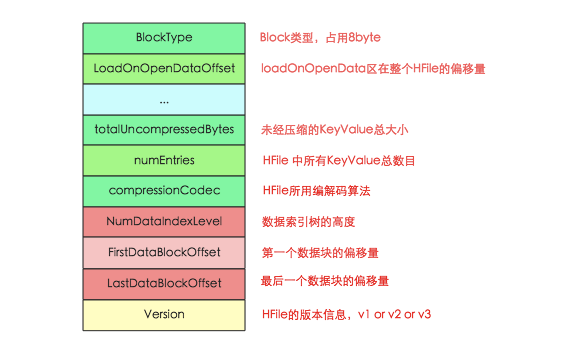


**HFile中Block块解析**

上文从HFile的层面将文件切分成了多种类型的block，接下来针对几种重要block进行详细的介绍，因为篇幅的原因，索引相关的block不会在本文进行介绍，接下来会写一篇单独的文章对其进行分析和讲解。首先会介绍记录HFile基本信息的TrailerBlock，再介绍用户数据的实际存储块DataBlock，最后简单介绍布隆过滤器相关的block。

**Trailer Block**

主要记录了HFile的基本信息、各个部分的偏移值和寻址信息，下图为Trailer内存和磁盘中的数据结构，其中只显示了部分核心字段：



HFile在读取的时候首先会解析Trailer Block并加载到内存，然后再进一步加载LoadOnOpen区的数据，具体步骤如下：

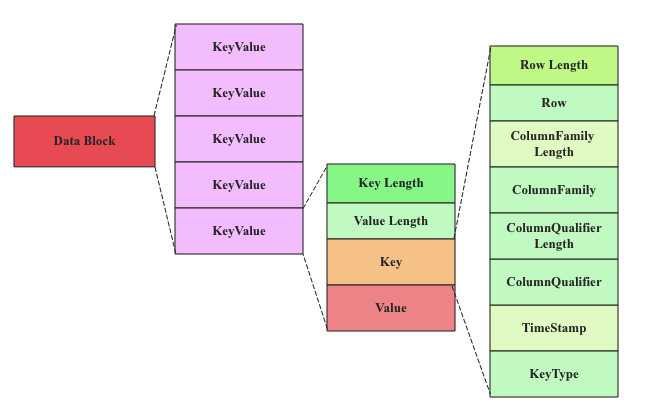
1. 首先加载version版本信息，HBase中version包含majorVersion和minorVersion两部分，前者决定了HFile的主版本： V1、V2 还是V3；后者在主版本确定的基础上决定是否支持一些微小修正，比如是否支持checksum等。不同的版本决定了使用不同的Reader对象对HFile进行读取解析

2. 根据Version信息获取trailer的长度（不同version的trailer长度不同），再根据trailer长度加载整个HFileTrailer Block

3. 最后加载load-on-open部分到内存中，起始偏移地址是trailer中的LoadOnOpenDataOffset字段，load-on-open部分的结束偏移量为HFile长度减去Trailer长度，load-on-open部分主要包括索引树的根节点以及FileInfo两个重要模块，FileInfo是固定长度的块，它纪录了文件的一些Meta信息，例如：AVG\_KEY\_LEN, AVG\_VALUE\_LEN, LAST\_KEY, COMPARATOR, MAX\_SEQ\_ID\_KEY等；索引树根节点放到下一篇文章进行介绍。

**Data Block**

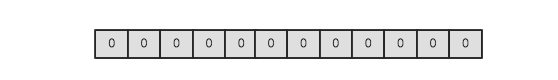
DataBlock是HBase中数据存储的最小单元。DataBlock中主要存储用户的KeyValue数据（KeyValue后面一般会跟一个timestamp，图中未标出），而KeyValue结构是HBase存储的核心，每个数据都是以KeyValue结构在HBase中进行存储。KeyValue结构在内存和磁盘中可以表示为：



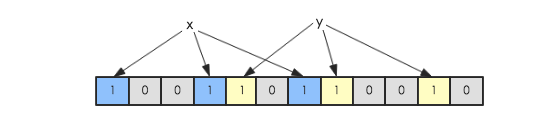
每个KeyValue都由4个部分构成，分别为key length，value length，key和value。其中key value和value length是两个固定长度的数值，而key是一个复杂的结构，首先是rowkey的长度，接着是rowkey，然后是ColumnFamily的长度，再是ColumnFamily，之后是ColumnQualifier，最后是时间戳和KeyType（keytype有四种类型，分别是Put、Delete、 DeleteColumn和DeleteFamily），value就没有那么复杂，就是一串纯粹的二进制数据。

**BloomFilter Meta Block & Bloom Block**

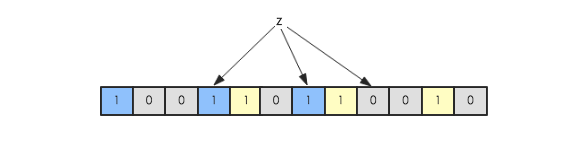
BloomFilter对于HBase的随机读性能至关重要，对于get操作以及部分scan操作可以剔除掉不会用到的HFile文件，减少实际IO次数，提高随机读性能。在此简单地介绍一下Bloom Filter的工作原理，Bloom Filter使用位数组来实现过滤，初始状态下位数组每一位都为0，如下图所示：



假如此时有一个集合S = {x1, x2, … xn}，Bloom Filter使用k个独立的hash函数，分别将集合中的每一个元素映射到｛1,…,m｝的范围。对于任何一个元素，被映射到的数字作为对应的位数组的索引，该位会被置为1。比如元素x1被hash函数映射到数字8，那么位数组的第8位就会被置为1。下图中集合S只有两个元素x和y，分别被3个hash函数进行映射，映射到的位置分别为（0，3，6）和（4，7，10），对应的位会被置为1:



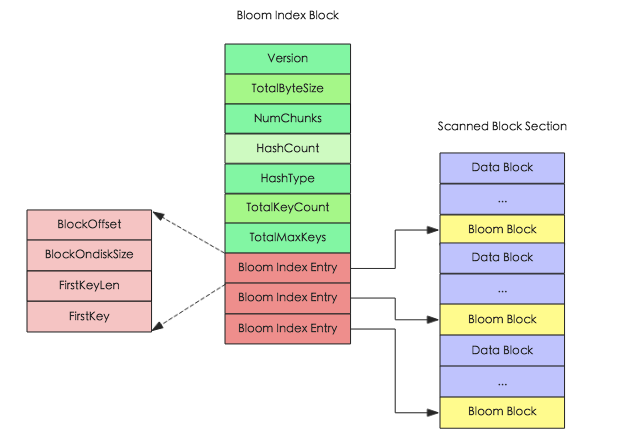
现在假如要判断另一个元素是否是在此集合中，只需要被这3个hash函数进行映射，查看对应的位置是否有0存在，如果有的话，表示此元素肯定不存在于这个集合，否则有可能存在。下图所示就表示z肯定不在集合｛x，y｝中：



HBase中每个HFile都有对应的位数组，KeyValue在写入HFile时会先经过几个hash函数的映射，映射后将对应的数组位改为1，get请求进来之后再进行hash映射，如果在对应数组位上存在0，说明该get请求查询的数据不在该HFile中。

HFile中的位数组就是上述Bloom Block中存储的值，可以想象，一个HFile文件越大，里面存储的KeyValue值越多，位数组就会相应越大。一旦太大就不适合直接加载到内存了，因此HFile V2在设计上将位数组进行了拆分，拆成了多个独立的位数组（根据Key进行拆分，一部分连续的Key使用一个位数组）。这样一个HFile中就会包含多个位数组，根据Key进行查询，首先会定位到具体的某个位数组，只需要加载此位数组到内存进行过滤即可，减少了内存开支。

在结构上每个位数组对应HFile中一个Bloom Block，为了方便根据Key定位具体需要加载哪个位数组，HFile V2又设计了对应的索引Bloom Index Block，对应的内存和逻辑结构图如下：



Bloom Index Block结构中totalByteSize表示位数组的bit数，numChunks表示Bloom Block的个数，hashCount表示hash函数的个数，hashType表示hash函数的类型，totalKeyCount表示bloom filter当前已经包含的key的数目，totalMaxKeys表示bloom filter当前最多包含的key的数目, Bloom Index Entry对应每一个bloom filter block的索引条目，作为索引分别指向’scanned block section’部分的Bloom Block，Bloom Block中就存储了对应的位数组。

Bloom Index Entry的结构见上图左边所示，BlockOffset表示对应Bloom Block在HFile中的偏移量，FirstKey表示对应BloomBlock的第一个Key。根据上文所说，一次get请求进来，首先会根据key在所有的索引条目中进行二分查找，查找到对应的Bloom Index Entry，就可以定位到该key对应的位数组，加载到内存进行过滤判断。

**总结**

这篇小文首先从宏观的层面对HFile的逻辑结构和物理存储结构进行了讲解，并且将HFile从逻辑上分解为各种类型的Block，再接着从微观的视角分别对Trailer Block、Data Block在结构上进行了解析：通过对trailer block的解析，可以获取hfile的版本以及hfile中其他几个部分的偏移量，在读取的时候可以直接通过偏移量对其进行加载；而对data block的解析可以知道用户数据在hdfs中是如何实际存储的；最后通过介绍Bloom Filter的工作原理以及相关的Block块了解HFile中Bloom Filter的存储结构。接下来会以本文为基础，再写一篇文章分析HFile中索引块的结构以及相应的索引机制。