如何从根源上解决 HDFS 小文件问题

我们知道，HDFS 被设计成存储大规模的数据集，我们可以在 HDFS 上存储 TB 甚至 PB 级别的海量数据。而这些数据的元数据（比如文件由哪些块组成、这些块分别存储在哪些节点上）全部都是由 NameNode 节点维护，为了达到高效的访问， NameNode
在启动的时候会将这些元数据全部加载到内存中。而 HDFS
中的每一个文件、目录以及文件块，在 NameNode
内存都会有记录，每一条信息大约占用150字节的内存空间。由此可见，HDFS
上存在大量的小文件（这里说的小文件是指文件大小要比一个 HDFS 块大小(在 Hadoop1.x
的时候默认块大小64M，可以通过 dfs.blocksize 来设置；但是到了 Hadoop 2.x
的时候默认块大小为128MB了，可以通过 dfs.block.size 设置)
小得多的文件。）至少会产生以下几个负面影响：

- 大量小文件的存在势必占用大量的 NameNode 内存，从而影响 HDFS 的横向扩展能力。
- 另一方面，如果我们使用 MapReduce 任务来处理这些小文件，因为每个 Map
  会处理一个 HDFS 块；这会导致程序启动大量的 Map
  来处理这些小文件，虽然这些小文件总的大小并非很大，却占用了集群的大量资源！

以上两个负面影响都不是我们想看见的。那么这么多的小文件一般在什么情况下产生？我在这里归纳为以下几种情况：

- 实时流处理：比如我们使用 Spark Streaming 从外部数据源接收数据，然后经过 ETL
  处理之后存储到 HDFS 中。这种情况下在每个 Job 中会产生大量的小文件。
- MapReduce 产生：我们使用 Hive 查询一张含有海量数据的表，然后存储在另外一张表中
  ，而这个查询只有简单的过滤条件（比如 select * from iteblog where from =
  'hadoop'），这种情况只会启动大量的 Map
  来处理，这种情况可能会产生大量的小文件。也可能 Reduce
  设置不合理，产生大量的小文件，
- 数据本身的特点：比如我们在 HDFS 上存储大量的图片、短视频、短音频等文件，由于这些文件的特点，而且数量众多，也可能给 HDFS 大量灾难。

那么针对这些小文件，现有哪几种解决方案呢？

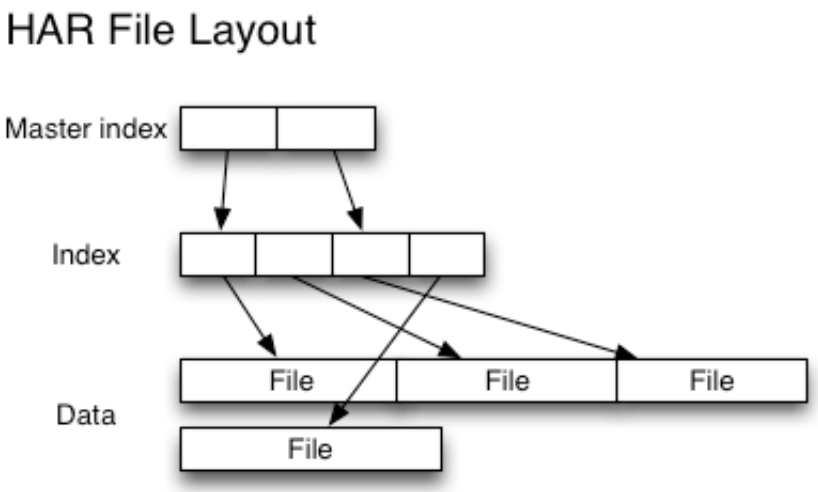
现有小文件解决方案

在本博客的《Hadoop小文件优化》文章中，翻译了 Cloudera 官方技术博客的《The Small Files Problem》文章，里面提供了两种 HDFS 小文件的解决方案。

HAR files

Hadoop Archives (HAR files)是在 Hadoop 0.18.0
版本中引入的，它的出现就是为了缓解大量小文件消耗 NameNode 内存的问题。HAR
文件是通过在 HDFS 上构建一个层次化的文件系统来工作。一个 HAR 文件是通过 hadoop 的

archive 命令来创建，而这个命令实 际上也是运行了一个 MapReduce 任务来将小文件打包成 HAR 文件。对客户端来说，使用 HAR
文件没有任何影响。所有的原始文件都可见并且可访问的（通过 har://URL）。但在 HDFS 端它内部的文件数减少了。架构如下：

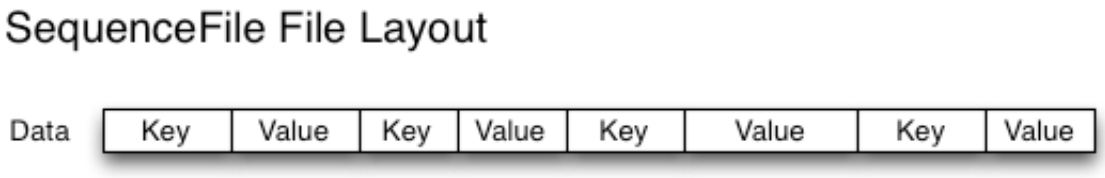


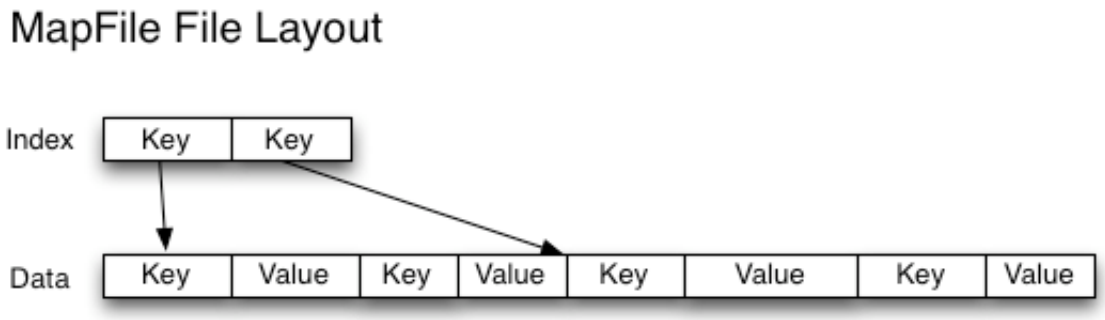
如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

从上面实现图我们可以看出，Hadoop
在进行最终文件的读取时，需要先访问索引数据，所以在效率上会比直接读取 HDFS 文件慢一些。

Sequence Files

第二种解决小文件的方法是使用 SequenceFile。这种方法使用小文件名作为 key，并且文件内容作为 value，实践中这种方式非常管用。如下图所示：





如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

和 HAR
不同的是，这种方式还支持压缩。
该方案对于小文件的存取都比较自由，不限制用户和文件的多少，但是 SequenceFile
文件不能追加写入，适用于一次性写入大量小文件的操作。

HBase

除了上面的方法，其实我们还可以将小文件存储到类似于 HBase 的 KV 数据库里面，也可以将
Key 设置为小文件的文件名，Value 设置为小文件的内容，相比使用 SequenceFile
存储小文件，使用 HBase 的时候我们可以对文件进行修改，甚至能拿到所有的历史修改版本。

从 HDFS 底层解决小文件问题

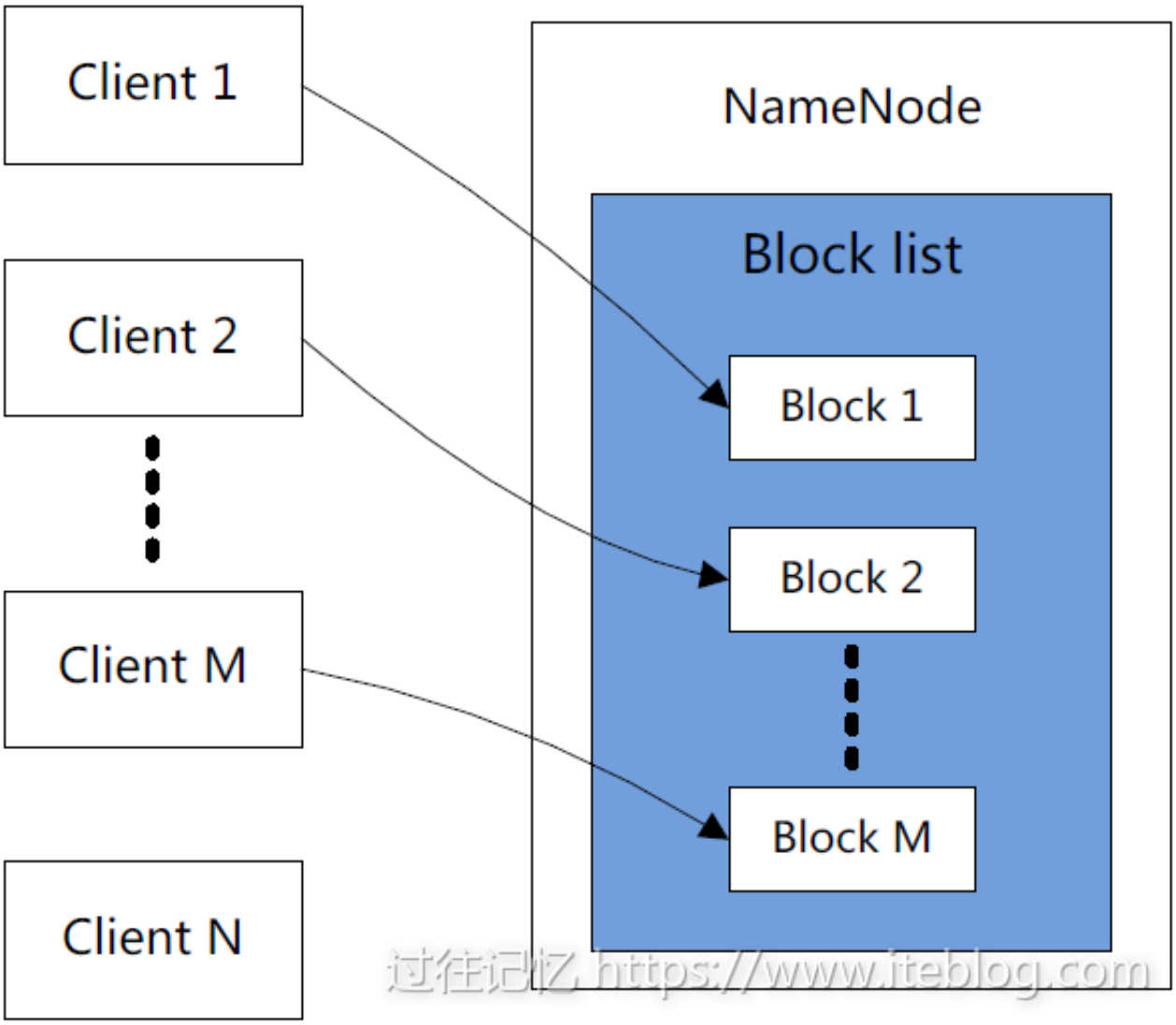
以上三种方法虽然能够解决小文件的问题，但是这些方法都有局限：HAR Files 和 Sequence Files
一旦创建，之后都不支持修改，所以这是对读场景很友好的；而使用 HBase
需要引入外部系统，维护成本很高。最后，这些方法都没有从根本上解决。那么能不能从 HDFS
底层解决这个问题呢？也就是对使用方来说，我们不需要考虑写的文件大小。目前 Hadoop
社区确实有很多相应的讨论和方案设想。下面将简要进行描述。

HDFS-8998

我们都知道，一个文件在 HDFS 上对应一个或多个 Block，每个 Block 在 NameNode（INode 和
BlocksMap）中都存在一定的元数据，而且这些数据需要占用 NameNode
一定内存。所以说，如果 HDFS 中存在大量的小文件，因为这些小文件都是小于一个 Block

大小，所以这些文件占用了一个 Block；这样，海量的小文件占用了海量的 Block 。那我们能不能把这些小 Block 合并成一个大 Block？这正是 HDFS-8998 的思想。其核心实现如下：

- 用户可以在 HDFS 上指定一个区域，用于存放小文件，这个区域称为小文件区域（Small file zone）；
- NameNode 将保存固定大小的 block 列表，列表的大小是可以配置的，比如我们配置成 M；
- 当客户端1第一次向 NameNode 发出写时，NameNode 将为客户端1创建第一个 blockid，并锁定此块；只有在关闭 OutputStream 的时候才释放这个锁；
- 当客户端2向 NameNode 发出写时，NameNode 将尝试为其分配未锁定的块（unlocked block），如果没有未锁定的块，并且现有的块数小于之前配置的大小（M），这时候 NameNode 则为客户端2创建新的 blockid 并锁定该块。
- 其余的客户端操作和这个类似；
- 客户端写数据的操作都是将数据追加到获取到的块上；
- 如果某个块被写满，也会分配新的一个块给客户端，然后写满的块将从 M 个候选块列表中移除，表示此块将不再接受写处理。
- 当 M 个块中没有未锁住的块并且 NameNode 无法再申请新块的时候，则当前客户端必须等待其它客户端操作完毕，并释放块。



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

从上图可以看出，每个 block 同时只能由一个客户端处理，但是当这个客户端写完，并释放相关锁之后，还能由其他客户端复用！直到这个 block 达到 HDFS 配置的快大小（比如 128MB）。

从上面的阐述可以看出，一个 block
将包含多个文件，那么我们需要引入额外的服务来维护各个文件在 block
中的偏移量。其余的读写删操作如下：

- 读取：关于这些文件的读取，其实和读取正常的 HDFS 文件类似。
- 删除：因为现在一个 block 包含不止一个文件，所以删除操作不能直接删除一个 block。现在的删除操作是：从 NameNode 中的 BlocksMap 删除 INode；然后当这个块中被删除的数据达到一定的阈值（这个阈值是可以配置的），对应的块对象会被重写。
- append 和 truncate：对小文件的 truncate 和 append 是不支持的，因为这些操作代价非常高。

HDFS-8286

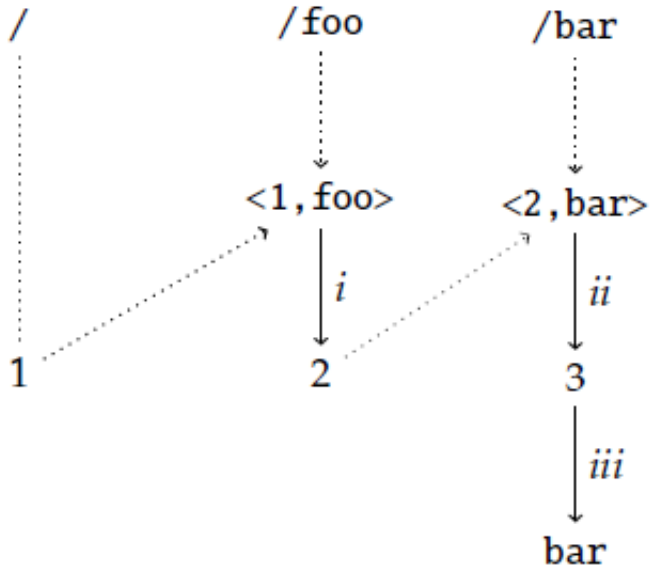
HDFS-8998 的设计目标是直接从底层的 block
做一些修改，从而减少文件元数据的条数，以此来减少 NN 的内存消耗。而 HDFS-8286
的目标是直接从解决 NN 管理的元数据入手，将 NN
管理的元数据从保存在内存转向到保存在第三方 KV 存储系统中，以此减缓 NN
的内存使用。更进一步的讲，这种方法同时也提高了 Hadoop 集群的扩展性。

现在的 HDFS 是以层次结构的形式来管理文件和目录的，所有的文件和目录都表示为 inode
对象。为了实现层次结构，一个目录需要包含对其所有子文件的引用。而 HDFS-8286 的方案采用
KV 的形式来存储元数据。下面我们来看看它是怎么实现的，我们先来了解下两条 kv
对应的规则：

- 每个 INode 对象都有对应的 inode id
- key 为 <pid ,foo> 将直接映射到 foo 对象的 inode id，而且 foo 对象的父对象的 inode id 为 pid

根据这两条规则，现在我们想基于 KV 元数据结构获取 /foo/bar 路径，解析过程如下图所示：

Key	Value
1	root
2	foo
3	bar
<1,foo>	2
<2,bar>	3



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

- 首先，root 的 inode id 为1，那么系统会将 foo 对象和其父对象的 inode id 进行组合，并得到一个 key <1,foo>；
- 第二步，系统根据上面得到的 key <1,foo>，从 KV 存储里面拿到 foo 对象的 inode id。从上图可以看出， foo 对象对应的 inode id 为2，对应上图的步骤 i；
- 第三步和第二步类似，系统需要拿到 bar 对象的 inode id，同样也是构造一个k key，得到 <2,bar>，最后从 KV 存储里面拿到 bar 对象的 inode id，这里为 3，对应上图的步骤 ii；
- 最后，系统直接根据 inode id 为 3，从 KV 存储里面拿到对应的 bar 的内容，对应图中的步骤 iii。

这个过程可以看到需要从 KV
存储里面进行多次检索，并进行解析，可能会在这里面出现一些性能问题。

Hadoop Ozone

Ozone 是 Hortonworks 基于 HDFS 实现的一个对象存储服务，旨在基于 HDFS 的 DataNode
存储，支持更大规模的数据对象存储，支持各种对象大小并且拥有 HDFS
的可靠性，一致性和可用性，对应的 issue 请参见 HDFS-7240。目前这个项目已经成为 Apache
Hadoop 的子项目，参见 ozone。Ozone 的一大目标就是扩展
HDFS，使其支持数十亿个对象的存储。关于 Ozone 的使用文档可以参见 Apache Hadoop Ozone
。

总结

社区关于 HDFS 的扩展性问题基本上都是通过解决 NameNode
元数据的存储问题，将元数据由原来的单节点存储扩展到多个；甚至直接使用外部的 KV
系统来存储一部分元数据或全部的元数据。相信不久的将来，使用 HDFS
存储小文件已经不是什么问题了。