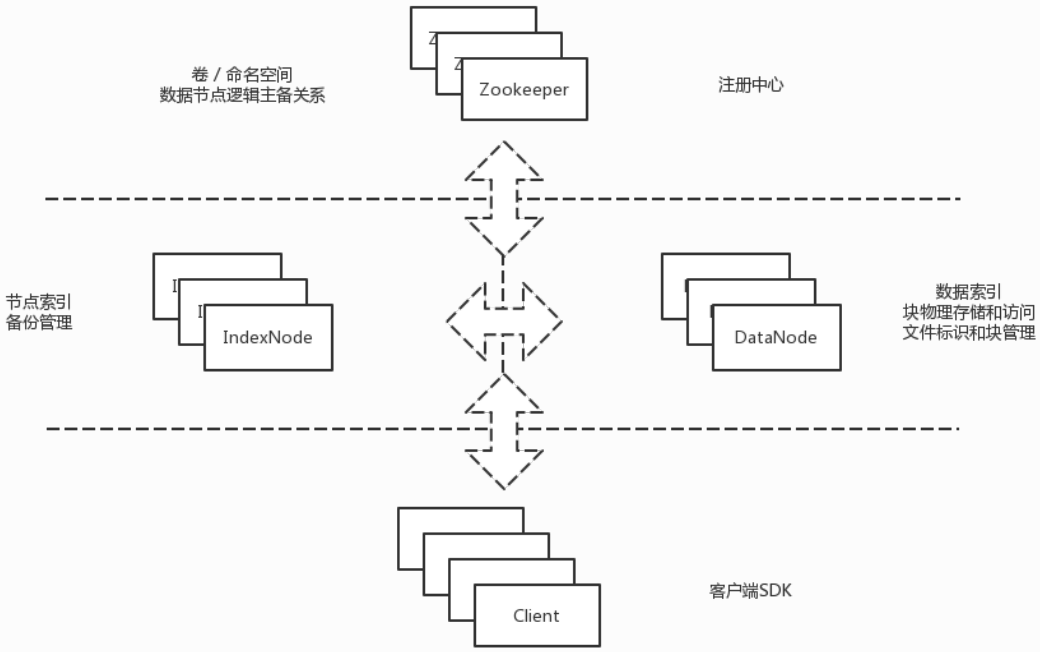
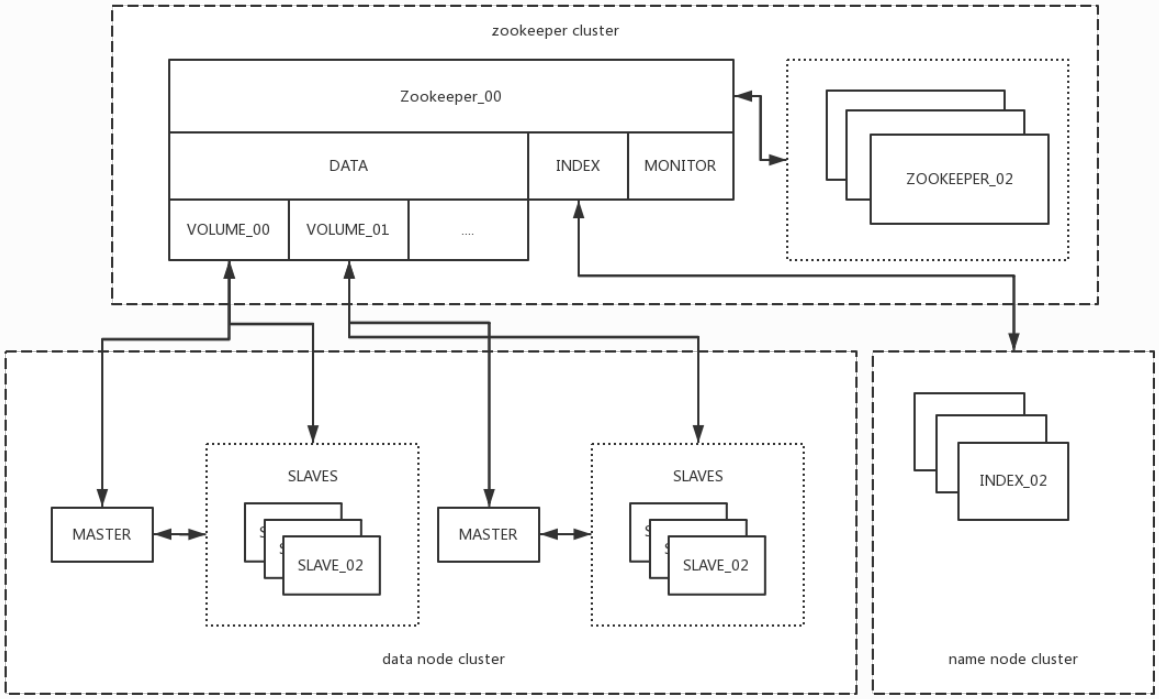
NDFS 一个分布式文件系统的设计与实现

定义

NDFS(Nero Distributed File System)通过网络，将分散的不同机器上的物理磁盘虚拟成一个或多个逻辑磁盘。


卷 / 命名空间
数据节点逻辑主备关系
Zookeeper
注册中心
节点索引
备份管理
IndexNode
DataNode
数据索引
块物理存储和访问
文件标识和块管理
Client
客户端SDK

关于HA


zookeeper cluster
Zookeeper_00
DATA
INDEX
MONITOR
VOLUME_00
VOLUME_01
....
ZOOKEEPER_02
SLAVES
MASTER
SLAVE_02
MASTER
SLAVES
SLAVE_02
data node cluster
INDEX_02
name node cluster

相对于HDFS的HA方案（在namenode里面建立映射表，为了防止namenode挂掉，所有一般会有两个namenode：namenode active 和 namenode standby。只有active对外提供服务。）这种设计的吞吐量不会很大，对于数据存储很频繁的系统，namenode的io速度会是系统瓶颈，再者 HDFS 的 Federation 方案（有多个namenode，分别负责几个 datanode。类似于美国的联邦机制）不够灵活，本质上每个联邦的namenode都存在单点故障。除非将 HA 方案和 Federation 方案结合使用，但是系统灵活性不大，运维复杂。

而NDFS 的 HA 方案是多 namenode（indexnode），多 datanode，datanode 通过 zookeeper（由Paxos/Raft支撑，不会有单点故障）发现 namenode 并向namenode更新索引，所有namenode均对外提供服务，通过LoadBalance去确定要访问的 namenode，有效提供了系统的吞吐量以及避免了单点故障。 所有datanode均为平级，只存在逻辑主备关系，所有属于同一个volume的多个datanode也均可对外提供服务，也是由LoadBalance指定。 增加了系统的灵活性，降低了运维复杂度。这种两层的负载均衡有效的提高了系统的吞吐量。

关于索引

namenode 存储{fileHash,Node}为节点的 B+-Tree，下次优化改为LSM—Tree

datanode 生成{BlockData} 为节点的索引树.

datanode 生成 {fileHash,ChunkName} 映射文件,方便文件快速定位

datanode 生成{Section,ChunkName} 的索引树，用于规划文件储存。其中Section为块内空余空间段。

这个设计相对于原来的 namenode 存储整个索引文件,减小了namenode的关于文件位置规划的计算量，索引存储量，能有效见小namenode负载，增大 namenode 吞吐量和并发。

HDFS 的 namenode运行时将元数据及其块映射关系加载到内存中，随着集群数据量的增大，namenode的内存空间也会遇到瓶颈。据实际生产经验统计如下:

文件数	数据块数	内存占用
3000万	3000万	约12G，块管理 ≈ 7.8G，包括全部块副本信息，目录树 ≈ 4.3G，目录层次结构，包含文件块列表信息
10亿	10亿	约380G，块管理 ≈ 240GB，目录树 ≈ 140GB

对于大规模系统 10 亿文件数只是时间问题。

NDFS 的namenode启动后，datanode向namenode提供索引{fileHash,nodeName}对，然后namenode将其加载到内存中，大致统计如下:

文件数	数据块数	内存占用
3000万	3000万	约1G，索引树 ≈ 1G，只包含文件存在节点位置
10亿	10亿	约32G，索引树 ≈ 32G，只包含文件存在节点位置

datanode逻辑主备关系从zookeeper获取，依datanode数量而定，本质上这个不会有多大。

HDFS将所有datanode的数据索引建立在namenode上，所以namenode占用内存较大。在集群数据量巨大，索引树较大的情况下，索引速度也不容乐观。

而且在小文件居多的情况下，这种问题更加严重，但是Hadoop目前还没有一个系统级的通用的解决HDFS小文件问题的方案。它自带的三种方案，包括Hadoop Archive，Sequence file和CombineFileInputFormat，均需要用户根据自己的需要编写程序解决小文件问题。

而NDFS则将索引分散到各个 datanode，由 namenode 索引到 datanode，再由 datanode 索引到文件块，这种分布式的二级索引方式是 namenode 内存占用量有效降低的本源。

关于储存规划

HDFS 由 namenode 规划储存路径，当小文件居多时这种方式占用较多的namenode的计算资源（要规划存储位置），对于在一个存储集群中，将规划存储位置的任务给相对于datanode要少很多的namenode显然是不合适的。

而 NDFS 由于其独特的数据块定义，使得可以由 datanode 规划存储位置，这种设计也使得NDFS的文件块可在脱离NDFS文件系统的时候也可以读取文件块内文件，也使得NDFS可以在datanode上建立文件索引，由datanode承担一部分索引任务，有效的降低的namenode的内存压力，计算压力，这部分将在后面详述。

关于储存方式

共同优势，使用块存储。将零散小文件压缩成块连续储存的方式相对于直接将零散小文件离散存储的方式，能有效的降低磁盘寻道时间。如果数据块设置过少，那需要读取的数据块就比较多，由于数据块在硬盘上非连续存储，普通硬盘因为需要移动磁头，所以随机寻址较慢，读越多的数据块（例如常规的文件存储）就增大了总的硬盘寻道时间。当硬盘寻道时间比io时间还要长的多时，那么硬盘寻道时间就成了系统的一个瓶颈。合适的块大小有助于减少硬盘寻道时间，提高系统吞吐量。一个很明显的例子：

向移动硬盘拷贝1000000个1K的小文件使用的时间要比向移动硬盘拷贝1个1G的文件使用的时间要多的多。虽然它们总大小都是1G。

同时，这种较大的文件块，在数据迁移的时候有快速的优势。

如果数据块设置过大，在读取或写入一个数据块的时候将占用更大的内存，所以HDFS和NDFS均使用的适宜的文件块大小64M/128M/256M。建议使用64M。

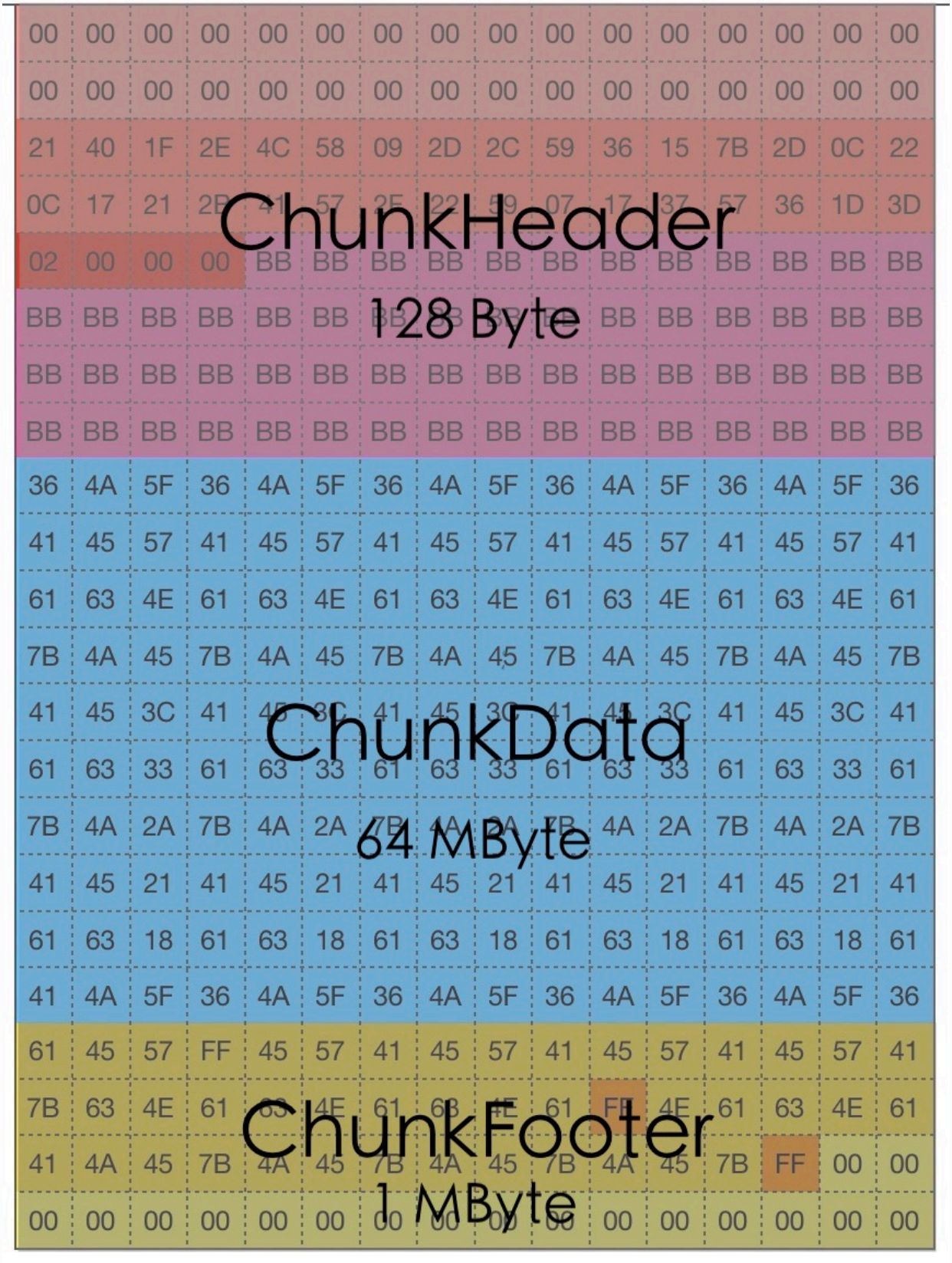
HDFS 使用块储存，对于大文件（文件大小大于文件块大小），HDFS将其拆分存到多个文件块中。

NDFS 使用块储存，将小文件压缩到文件块中，但是NDFS不适用于储存大文件（文件大小大于文件块大小），虽然将文件块配置大也可以做到。

文件块定义

NDFS文件块定义

NDFS文件块Chunk由ChunkHeader，ChunkData，ChunkFooter构成，如下：



文件块头ChunkHeader占128字节，其中文件块版本version占32字节，文件块uuid占32字节，文件块中文件数量fileCount占4字节，chainHash占32字节（用来做防篡改校验），为后续预留headerUnknown 占28字节。文件头示例：

```
00000000h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020h: 33 65 31 38 34 36 37 61 2D 32 36 37 39 2D 34 63
00000030h: 66 32 2D 39 36 62 62 2D 38 66 66 36 30 30 65 36
00000040h: 02 00 00 00 BB BB BB BB BB BB BB BB BB BB BB BB
00000050h: BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB
00000060h: BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB
00000070h: BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB BB
```

文件块数据部分ChunkData占64MByte/128MByte/256MByte/... 默认64MByte，该部分存储文件源数据，或文件压缩数据连续存储，默认使用Google Snappy压缩。示例：

```
00000080h: C7 E1 02 88 FF D8 FF E0 00 10 4A 46 49 46 00 01
00000090h: 02 00 00 64 00 64 00 00 FF EC 00 11 44 75 63 6B
000000a0h: 79 00 01 00 04 00 00 05 13 B8 EE 00 26 41 64 6F
000000b0h: 62 65 00 64 C0 00 00 00 01 03 00 15 04 03 06 0A
000000c0h: 0D 00 00 24 78 00 00 63 C7 00 00 89 8C 00 00 B0
000000d0h: C5 FF DB 00 84 00 01 01 01 9E 02 00 00 02 19 01
000000e0h: 00 03 15 01 0D 3D 04 02 01 01 1F 04 01 02 1D 1A
000000f0h: 9A 01 00 80 FF C2 00 11 08 01 F4 01 F4 03 01 11
00000100h: 00 02 11 01 03 11 01 FF C4 01 2B 00 01 00 02 03
00000110h: 00 03 01 01 00 11 01 1C 08 09 06 07 0A 02 04 05
00000120h: 01 73 0C 00 01 04 03 19 1A 01 E4 4C 05 06 08 02
00000130h: 04 07 09 10 00 00 05 03 02 05 04 02 03 01 00 03
00000140h: 01 3A F0 7D 02 03 04 05 06 00 01 07 20 08 10 30
00000150h: 50 11 35 40 33 36 37 31 34 90 21 12 22 23 24 16
00000160h: 11 00 01 03 02 04 02 03 09 0A 0B 05 05 05 09 00
00000170h: 00 02 01 03 04 11 05 00 21 12 06 31 13 41 22 14
00000180h: 20 51 61 32 42 23 74 15 B5 10 30 71 52 B2 33 73
00000190h: 24 76 07 40 50 62 72 43 53 63 B3 B4 75 16 91 C3
000001a0h: 25 86 C6 81 A1 B1 82 34 90 83 44 54 65 C1 A2 93
000001b0h: 64 C4 35 55 85 26 12 00 01 01 03 05 0A 0C 04 04
000001c0h: 05 05 05 BF F0 3C 01 02 00 11 03 20 21 31 12 04
000001d0h: 30 50 41 51 71 81 22 52 72 13 10 61 91 A1 B1 C1
000001e0h: 32 42 B2 23 73 05 F0 62 82 C2 D1 E1 D2 74 90 92
000001f0h: A2 53 14 33 43 83 C3 24 E2 54 13 01 00 01 03 02
00000200h: 03 08 03 21 60 01 C8 A0 01 11 00 21 31 41 51 61
00000210h: 71 81 10 20 30 50 F0 91 A1 B1 40 C1 E1 D1 90 F1
00000220h: 60 FF DA 00 0C 03 01 00 02 11 03 11 00 00 01 BF
```

...

文件块底部ChunkFooter，该部分记录文件块内文件列表信息，文件列表信息结构如下：

{ chunkName : String
fileName : String
fileType : String
index : Integer
fileHash : String
fileSize : Integer
compressionAlgorithm : String
compressionSize : Integer
chainHash : String del : Boolean lastMdfTime : Long
createTime : Long
}

其中chunkName为文件块名称；fileName为文件名称；fileType为文件类型，例如png；index为文件数据在data部分的起始位置；fileHash为文件的hash值，方便查找文件以及防止数据重复；fileSize为文件数据大小；compressionAlgorithm为数据压缩方式，例如zip,snappy；compressionSize为数据压缩后大小，index和compressionSize可以从data部分取出文件的压缩数据；chainHash为文件的hash链，用来做防篡改校验；del标识该文件是否已经删除；lastMdfTime为最后一次修改时间；createTime为文件创建时间。

默认将该结构对象使用Google ProtoStuff序列化后再由Google Snappy压缩，然后存放于ChunkFooter部分，每一条之间使用一个字节分隔符0xFF分隔。文件根部示例：

```
04000080h: 59 F0 58 0A 24 32 31 38 36 33 39 34 31 2D 30 39
04000090h: 65 32 2D 34 32 37 62 2D 38 30 37 31 2D 30 66 62
040000a0h: 35 36 31 34 38 35 31 38 34 12 05 73 74 65 70 73
040000b0h: 1A 03 6A 70 67 20 00 2A 05 68 61 73 68 31 30 C7
040000c0h: E1 02 3A 06 73 6E 61 70 70 79 40 E7 D4 02 48 DD
040000d0h: F5 9A 8A C4 2C 50 DD F5 9A 8A C4 2C FF 5D F0 5C
040000e0h: 0A 24 32 31 38 36 33 39 34 31 2D 30 39 65 32 2D
040000f0h: 34 32 37 62 2D 38 30 37 31 2D 30 66 62 35 36 31
04000100h: 34 38 35 31 38 34 12 07 73 74 65 70 73 2D 6F 1A
04000110h: 03 6A 70 67 20 C7 E1 02 2A 05 68 61 73 68 32 30
04000120h: E4 DE 01 3A 06 73 6E 61 70 70 79 40 F0 D0 01 48
04000130h: A0 FD 9A 8A C4 2C 50 A0 FD 9A 8A C4 2C FF 00 00
```

块存储

文件读取

文件写入，当客户端向 namenode 请求数据写入，namenode LoadBalance一个 datanode ，并向其发出位置规划请求，datanode 规划存储位置，规划算法如下：

1. 最佳适应法，最佳适应算法要求空闲区按大小递增的次序排列.在进行空间分配时,从空闲分区表首开始顺序查找,直到找到第一个能满足其大小要求的空闲区为止,如果该空闲区大于请求表中的请求长度,则将剩余空闲区留在可用表中(如果相邻有空闲区,则与之和并),然后修改相关表的表项.按这种方式为作业分配空间,就能把既满足要求又与作业大小接近的空闲分区分配给作业.如果空闲区大于该作业的大小,则与首次适应算法相同,将剩余空闲区仍留在空闲分区表中.

该算法的特点是:若存在与作业大小一致的空闲分区,则它必然被选中;若不存在与作业大小一致的空闲分区,则只划分比作业稍大的空闲分区,从而保留了大的空闲区.但空闲区一般不可能正好和作业申请的空间大小一样,因而将其分割成两部分时,往往使剩下的空闲区非常小,从而在存储器中留下许多难以利用的小空闲区(也被称为碎片).

2. 首次适应法，要求把空间中的可用分区单独组成可用分区表或可用分区自由链,按起始地址递增的次序排列.查找的方法是每次按递增的次序向后找,一旦找到大于或等于所要求空间长度的分区,则结束查找,从找到的分区中划分所要求的空间大小分配给用户,把剩余的部分进行合并(如果有相邻的空闲区存在的话),并修改可用区中的相应表项.

该算法的特点是利用空间低地址部分的空闲分区,从而保留了高地址部分的大空闲区.但由于低地址部分不断地被划分,致使低地址端留下许多难以利用的很小的空闲分区,而每次查找有都是从低地址部分开始,这无疑增加了查找可用空闲分区的开销.

3. 循环适应法，为进程分配空间空间时,不是每次从空闲分区表首开始查找,而是从上次找到的空闲分区的下一个空闲分区开始查找,直到找到第一个能满足其大小要求的空闲分区为止.然后按照作业大小,从该分区划出一块空间分配给请求者,余下的空闲分区仍留在空闲分区表中.

该算法的特点是使存储空间的利用更加均衡,分配的速度会快一些,碎片也可能会少一些,不至于使小的空闲区集中存储在存储区的一端,但会导致缺乏大的空闲分区.

4. 最坏适应法，最坏适应算法要求按空闲区大小,从大到小递减顺序组成空闲区表或自由链.寻找的方法是当用户作业或进程申请一个空闲区时,选择能满足要求的最大空闲区分配,先检查空闲区可用表或自由链的第一个空闲区的大小是否大于或等于所要求的空间长度,若满足,则分配相应的存储空间给用户,然后修改和调整空闲区可用表或自由链,否则分配失败.

目前使用最佳适应法

文件删除

将文件块Footer中del直接赋值为true，这样，在为新文件规划储存的时候会直接覆盖。

Footer整理

有时候由于Footer中del为true的文件太多，文件列表太大，Footer被填满，导致没法为新的文件创建文件信息，所以需要将del为true的文件信息从Footer删除，来整理出空间。默认在Footer装满的时候整理。

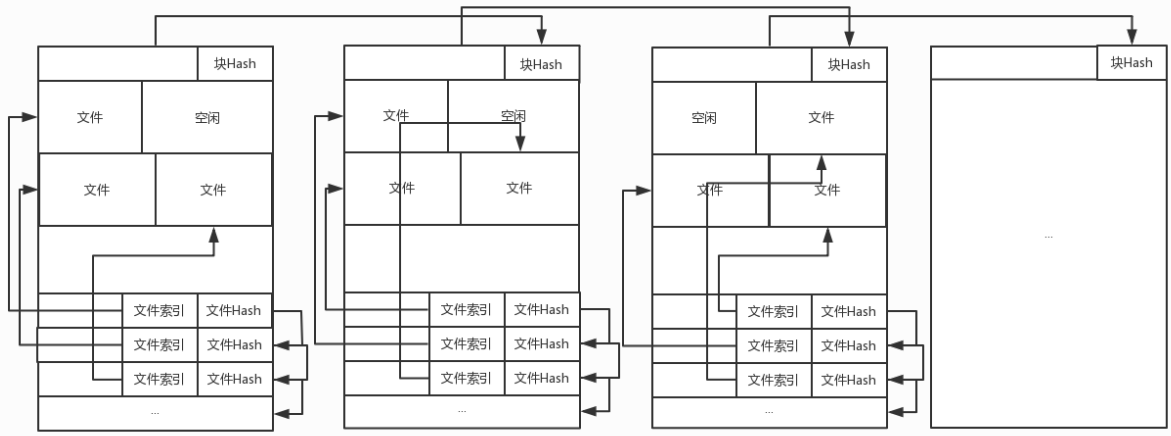
数据块碎片整理

由于采用了最佳适应法来为新的文件在文件块中分配空间导致产生的小空闲空间段，无法被有效利用。需要对其进行整理，整理默认由定时任务发起，以虚拟卷为单位进行整理。碎片整理涉及到多节点数据同步/索引同步，在后面详细述。

数据防篡改

该项配置需要在配置文件中打开，打开意味着所有对数据的修改和删除操作将失效，默认关闭.

Hash链，使用Hash算法的不可逆性，Hash链在Chunk设计中的图例：



块哈希链：每一个 chunk 的 header 的32字节chainHash由 上一个文件块的文件头+8字节时间戳 MD5 得出。chainHash = MD5(PrevChunkHeader+TimeStamp)。若该Chunk为datanode上第一个chunk，则chainHash由 128字节0xBB+时间戳 MD5 加密生成。chainHash = MD5(128byte0xBB+TimeStamp)

文件哈希链：每一个 BlockData 的chainHash由 当前块中该BlockData之前的BlockData(该chunk中最后一个BlockData)所对应的文件数据+之前BlockData的chainHash+时间戳 MD5加密得出。 chainHash = MD5(LastData+lastChianHash+TimeStamp)，若该文件为当前chunk内第一个文件，则该文件chainHash由该chunk 的 Header + 8字节时间戳 MD5加密生成。chainHash = MD5(Header+TimeStamp)

完整性校验：

文件块碎片整理

链式同步与3PC

链式同步

并行同步