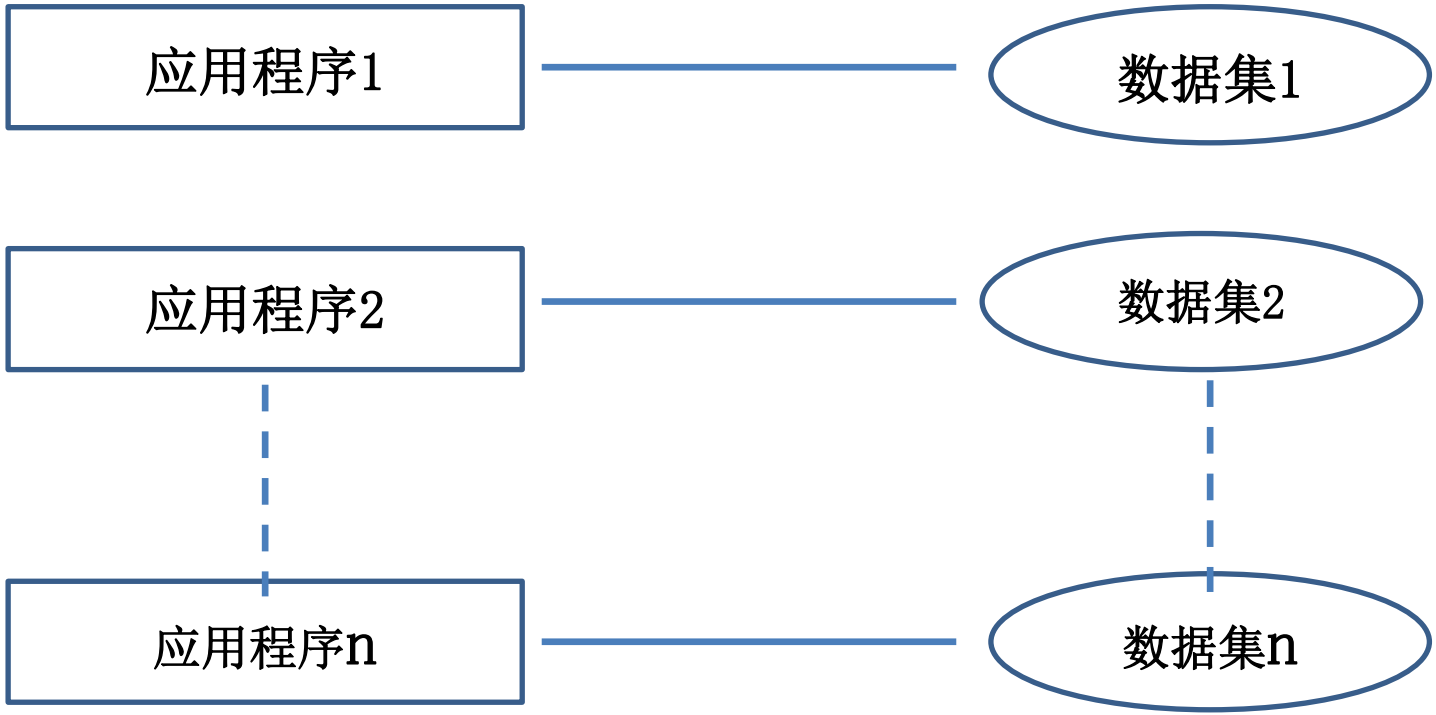
# 数据存储

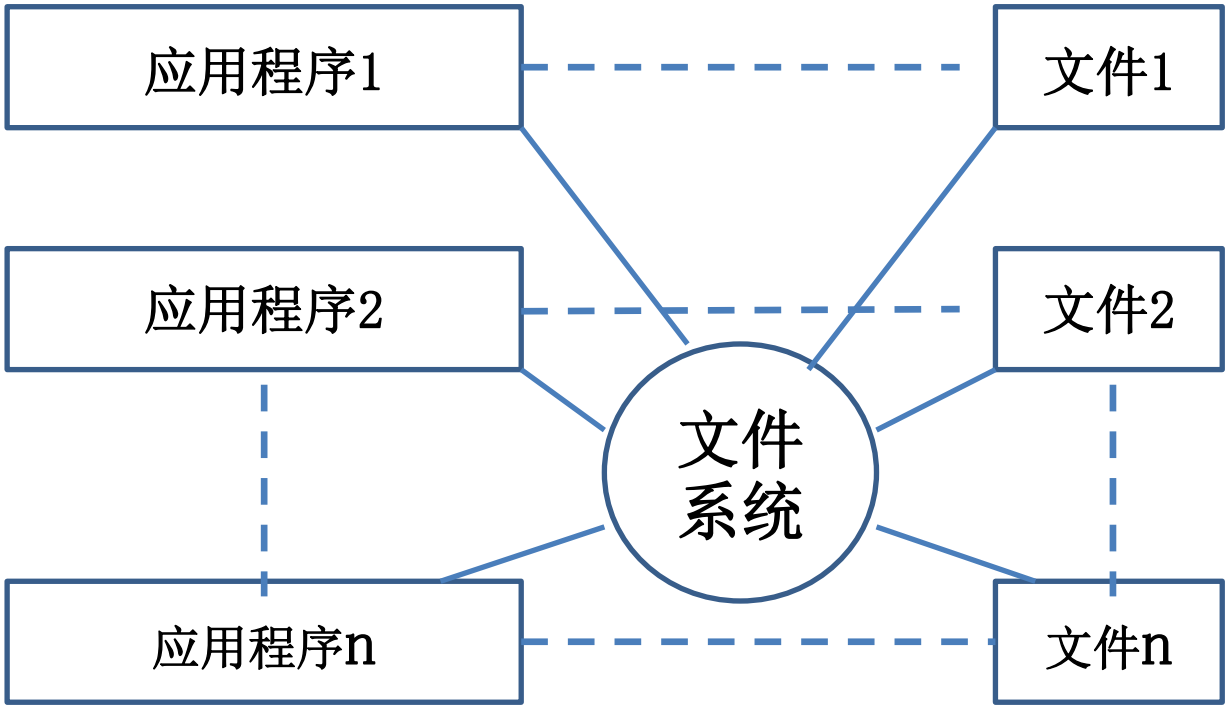
# 数据存储阶段

- 人工管理阶段
- 文件管理阶段
- 数据库管理阶段

# 人工管理阶段



# 文件系统管理阶段



# 文件系统

文件系统：操作系统用于明确磁盘或分区上的文件的方法和数据结构；磁盘上组织文件的方法。

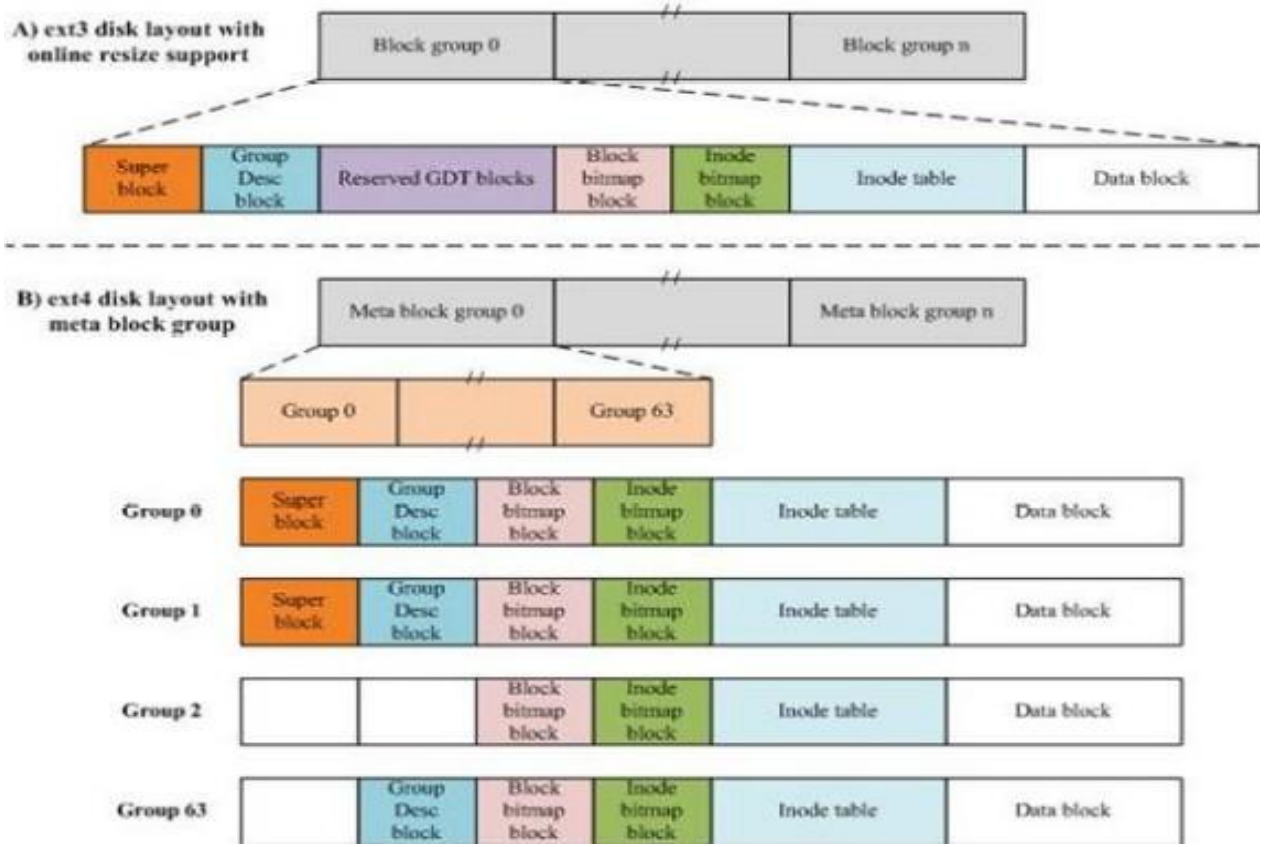
BlockSize、 inode

mkfs.ext4 -T largefile /dev/sdxx #inode_ratio 1M

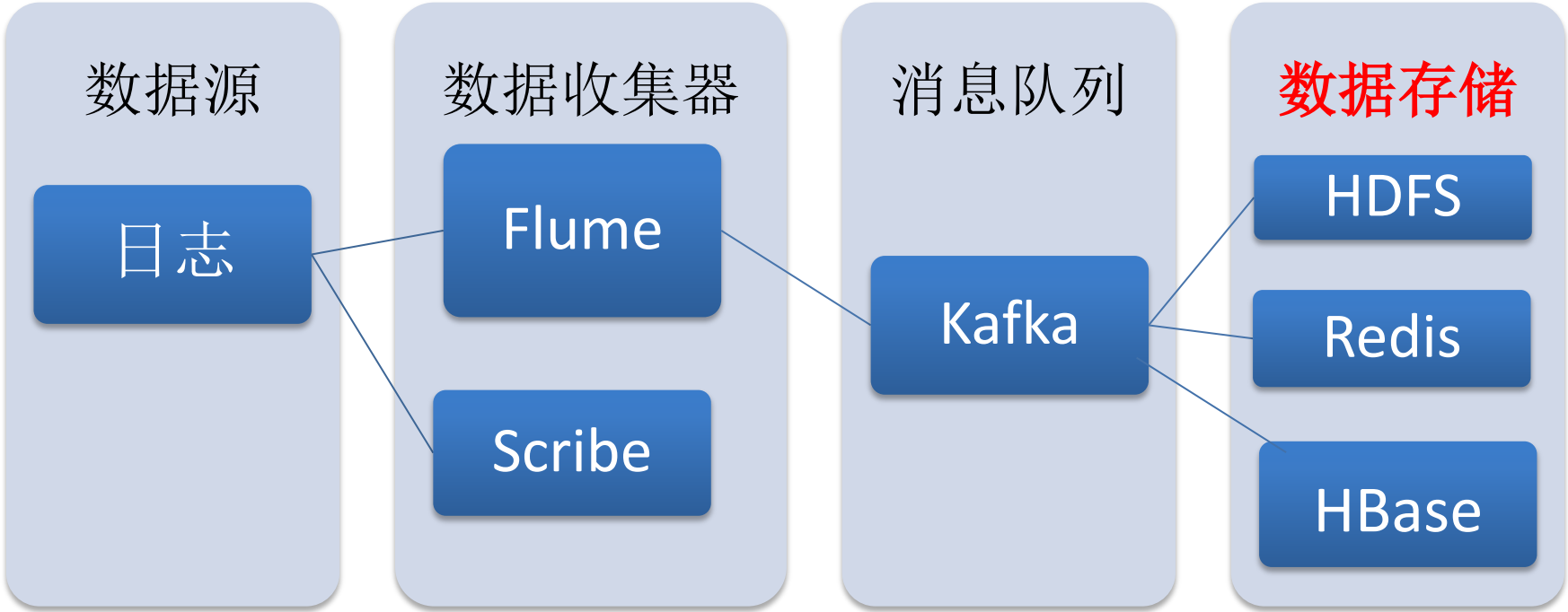
ext4 (rw,noatime,nodiratime,data=writeback,barrier=0)

Ext2/3/4; HFS; Btrf;JFS;ReiserFs; XFS;

# ext3和ext4结构



# 分布式数据存储



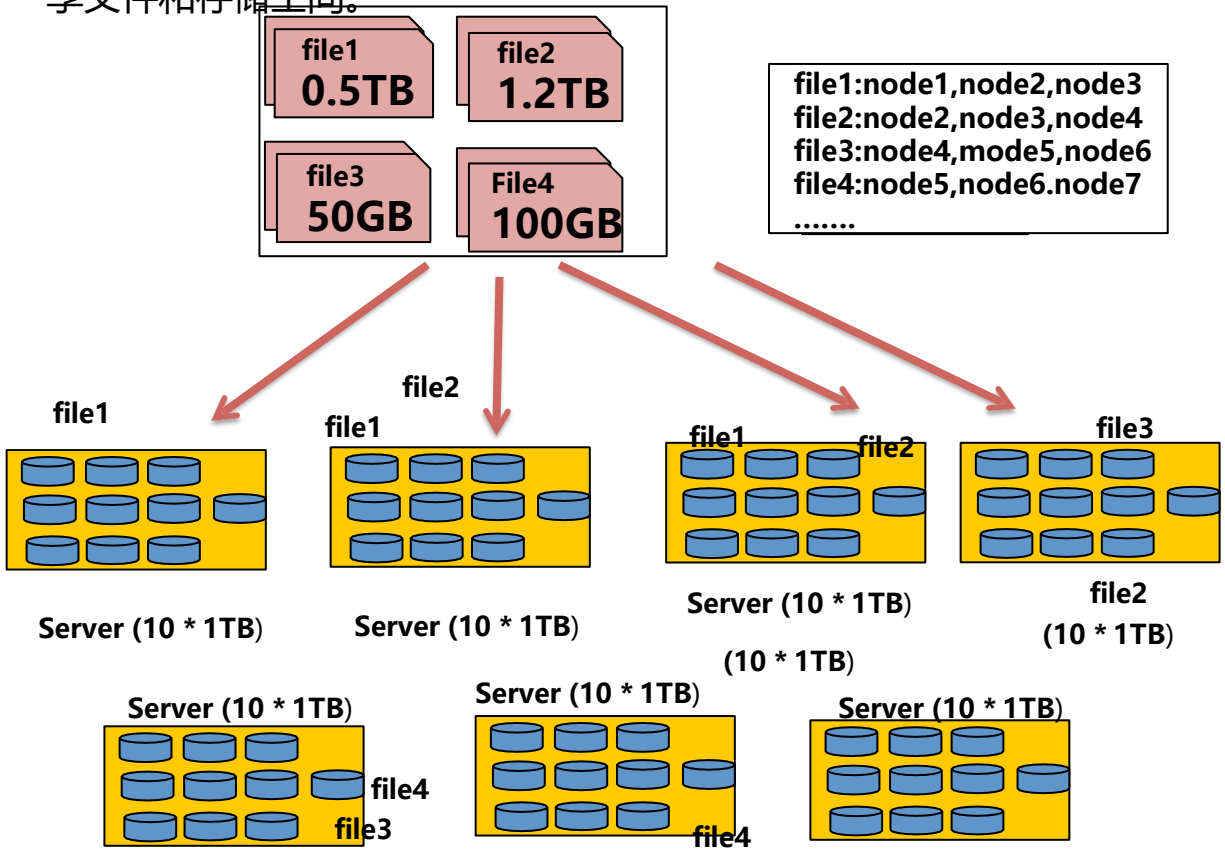
# 分布式数据存储

- 分布式文件系统：存储大量的文件、图片、音频、视频等非结构化数据，这些数据以对象的形式组织，对象之间没有关系，这数据都是二进制数据，例如GFS、HDFS等。
- 分布式Key-Value系统：用于存储关系简单的半结构化数据，提供基于Key的增删改查操作，缓存、固化存储，例如Memached、Redis、DynamoDB等。
- 分布式数据库系统： 存储结构化数据，提供SQL关系查询语言，支持多表关联，嵌套子查询等，例如MySQL Sharding集群、MongoDB等等。

# 分布式文件系统

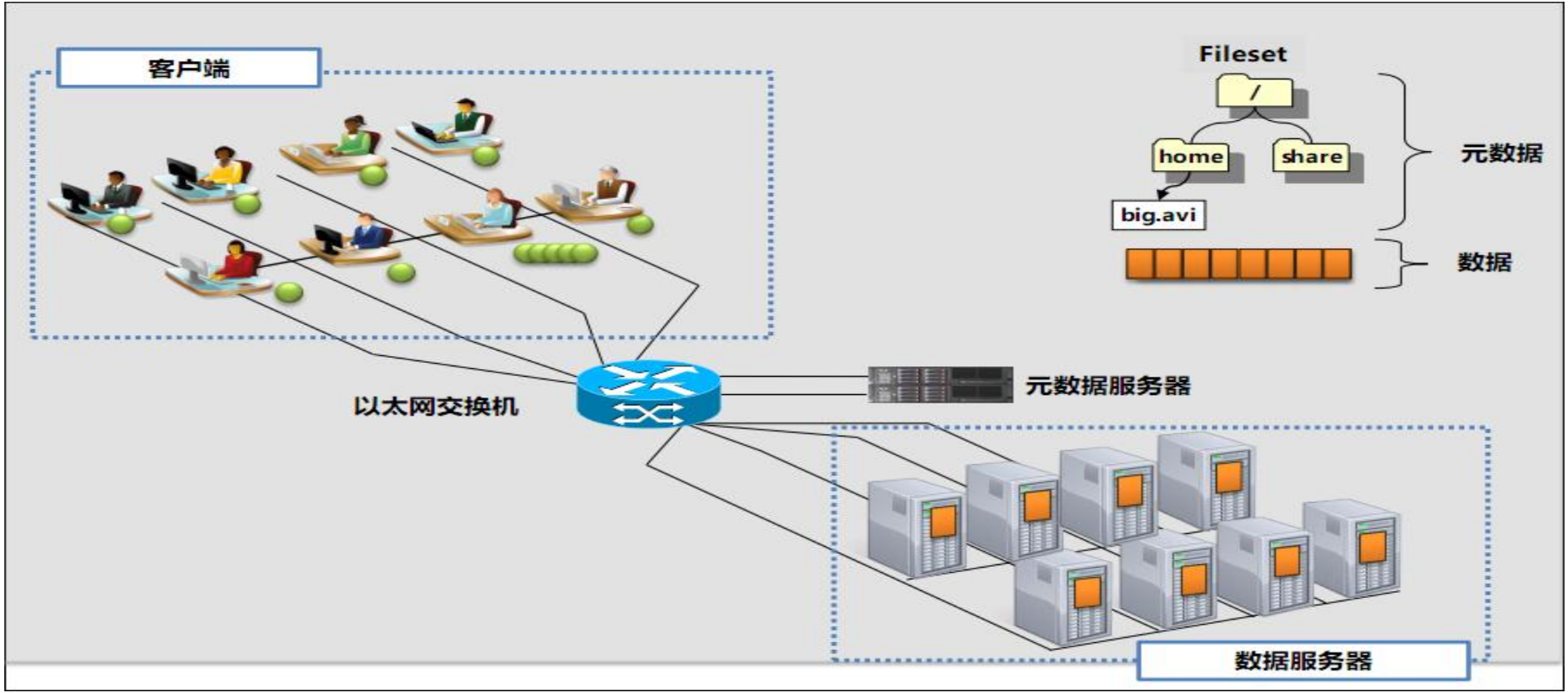
- 数据量越来越多，一个操作系统管辖的范围存不下了，分配到更多的操作系统管理的磁盘中，但是不方便管理和维护，迫切需要一种系统通过计算机网络与节点相连来管理多台机器上的文件，这就是分布式文件系统。从客户角度看一个标准的文件系统，提供了一系列API，由此进行文件或目录的创建、移动、删除，以及对文件的读写等操作。

- 从内部实现看，分布式系统文件内容和目录结构都不是存储在本地磁盘上，而是通过网络传输到远端系统上。并且同一个文件存储不只是在一台机器上，而是在一簇机器上分布式存储，协同提供服务允许文件通过网络在多台主机上分享的文件系统，可让多机器上的多用户分享文件和存储空间。

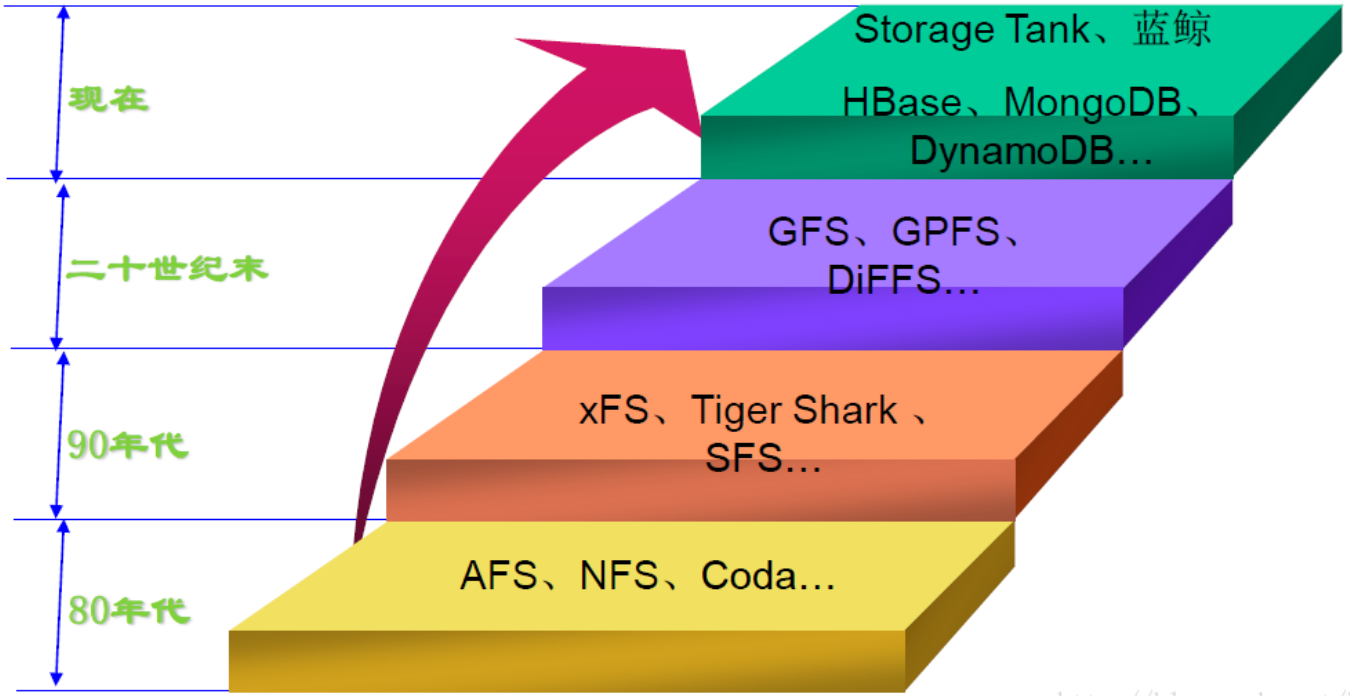


# 分布式文件系统

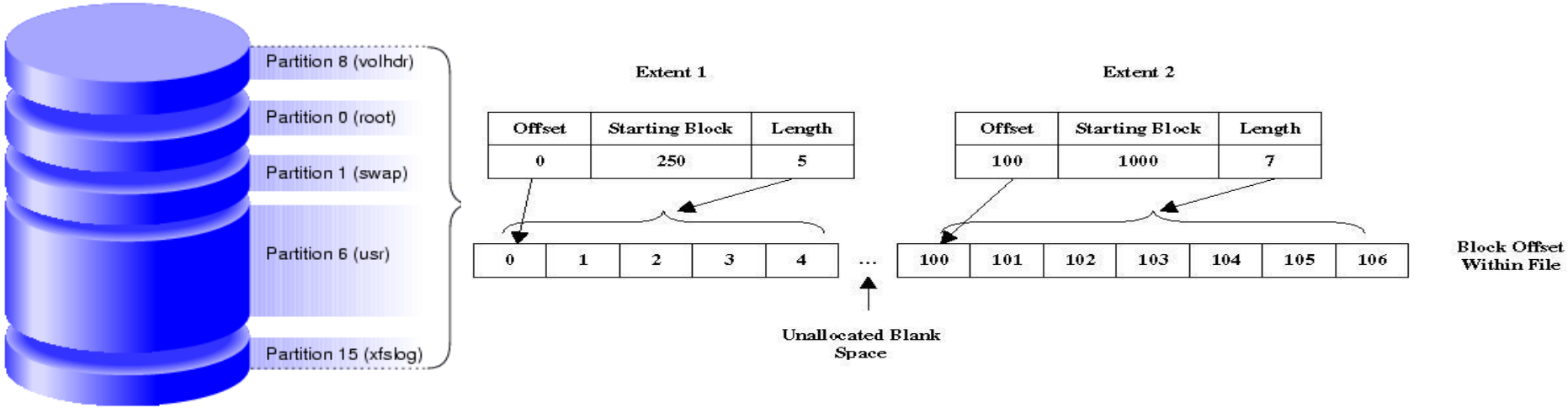
- 通透性。通过网络来访问文件就像访问本地磁盘。
- 容错。系统中有某些节点脱机，整体来说系统仍然可以持续运作而不会有数据损失。



# 分布式文件系统



# GFS文件系统结构

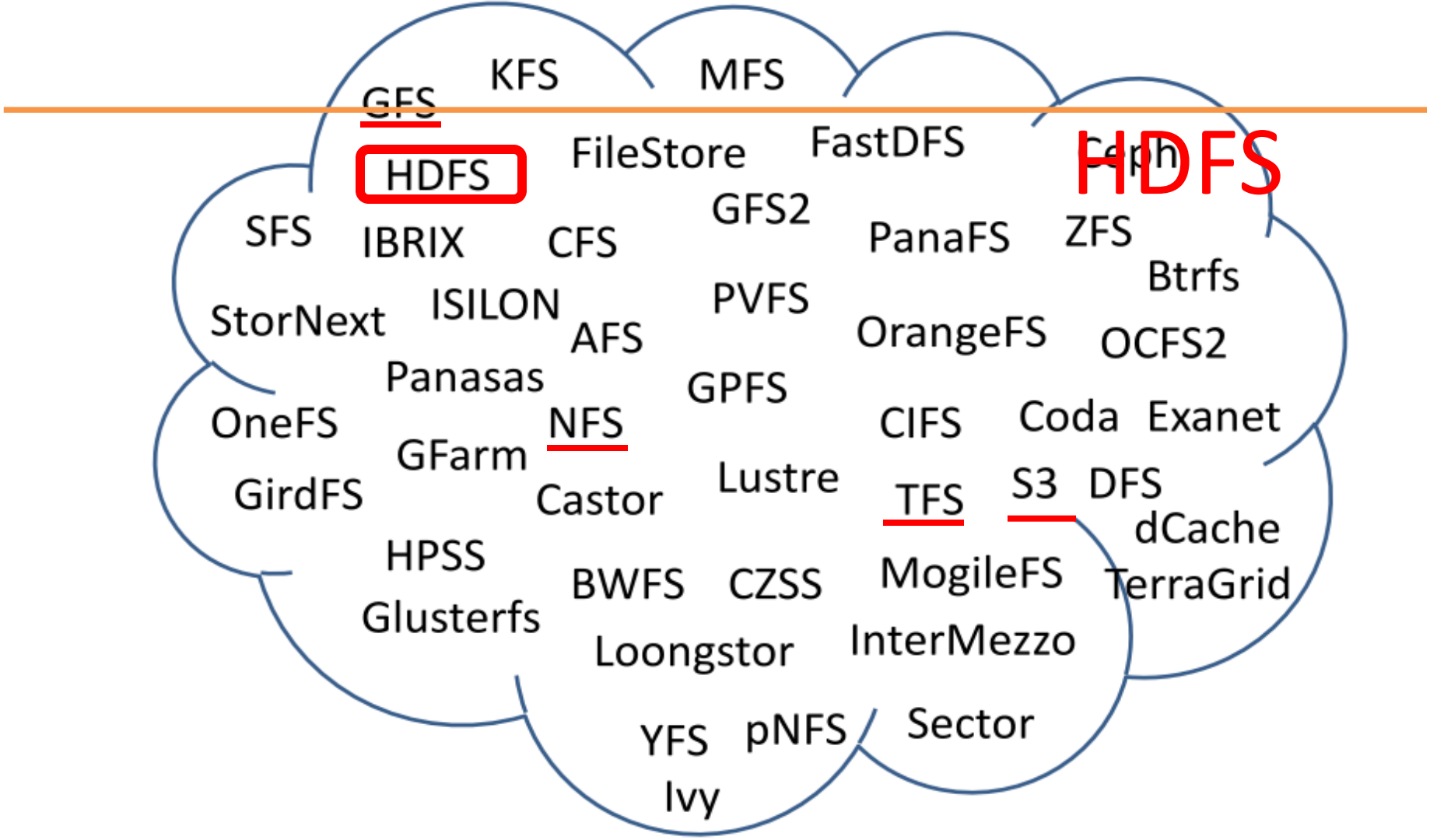



- GFS采用64MB数据块作为基本存储单元；Google业务逻辑特点决定了GFS中的文件读多写少，写主要是追加操作，基本不存在随机写操作；
- GFS的负载主要是对大文件的流式处理，追求高通量而不是低延迟，客户端缓存无意义；
- 64MB数据块降低了元数据的数量，因此系统可使用单一元数据服务器结构

# 谷歌第一个数据中心

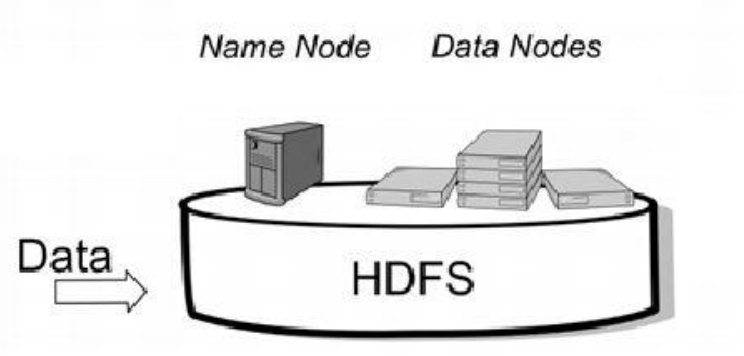
# 谷歌比利时数据中心

- 高扩展性，简单配置直接增加集群节点进行使用。低成本，用低端硬件设备组建集群，降低硬件成本。开源软件，降低软件成本。


KFS
MFS
GFS
HDFS
FileStore
FastDFS
Ceph
HDFS
GFS2
SFS
IBRIX
CFS
PanaFS
ZFS
Btrfs
StorNext
ISILON
PVFS
AFS
OrangeFS
OCFS2
Panasas
GPFS
OneFS
NFS
CIFS
Coda
Exanet
GFarm
GirdFS
Castor
Lustre
TFS
S3
DFS
dCache
HPSS
BWFS
CZSS
MogileFS
TerraGrid
Glusterfs
Loongstor
InterMezzo
YFS
pNFS
Sector
Ivy

# HDFS源于GFS

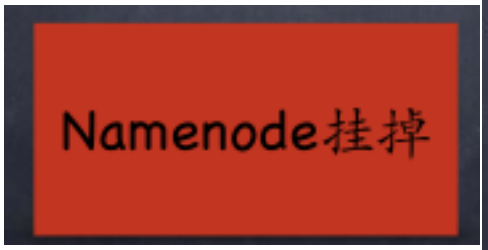
- Hadoop Distributed File System
- 源自Google的GFS论文，GFS的克隆
- 存储超过1TB以上几十TB以下数据文件时，一般采用DAS/NAS/SAN架构。若要以PB为单位，目前无法做到或者成本异常高。
- 分布式文件系统HDFS为大数据而生，整堆整堆来读处理。



# HDFS组成部分

- NameNode，命名服务器，负责保存DataNode文件元数据
- DataNode，数据节点，负责存储数据并汇报给NameNode
- SecondaryNameNode，NameNode的镜像备份节点
- Metadata，元数据：维护HDFS文件系统中文件和目录的信息
- 数据块
- 事务日志
- 映像文件
- 服务器通讯：RPC协议，心跳信息
- HDFS：分层次的文件及目录结构

# 命名服务器





- 负责所有的DataNode数据块复制。
- 负责保存DataNode数据块元数据，管理文件系统的命名空间，记录命名空间内的改动或空间本身属性的改动。
- 周期性地接受来自集群中数据节点的心跳和块报告。一个心跳的收条表示这个数据节点是健康的，是渴望服务数据的。一个块报告包括该数据节点上的所有的块列表。
- 协调客户端对文件的访问，客户端通过路径访问文件
- hdfs://namenode:port/dir-a/dir-b/dir-c/file.data

```
[hadoop@hdp-node-01 ~]$ hadoop fs -ls /
15/11/08 17:51:02 WARN util.NativeCodeLoader: Unable to load native-hadoop
Found 4 items
drwxr-xr-x   - hadoop supergroup          0 2015-11-03 03:26 /hbase
drwx-wx-wx   - hadoop supergroup          0 2015-11-04 02:09 /tmp
drwxr-xr-x   - hadoop supergroup          0 2015-11-04 06:46 /user
drwxr-xr-x   - hadoop supergroup          0 2015-11-08 08:40 /weblog
```

# 命名服务器

```
[hadoop@master current]$ pwd
/usr/local/webserver/hadoop/dfs/name/current
[hadoop@master current]$ ls -l
总用量 3124
-rw-rw-r--. 1 hadoop hadoop      42 2月  24 01:23 edits_0000000000000000001-0000000000000000002
-rw-rw-r--. 1 hadoop hadoop   12098 2月  24 02:23 edits_0000000000000000003-0000000000000000101
-rw-rw-r--. 1 hadoop hadoop      42 2月  24 03:24 edits_0000000000000000102-0000000000000000103
-rw-rw-r--. 1 hadoop hadoop 1048576 2月  24 04:18 edits_0000000000000000104-0000000000000000323
-rw-rw-r--. 1 hadoop hadoop 1048576 2月  26 21:54 edits_0000000000000000324-0000000000000000324
-rw-rw-r--. 1 hadoop hadoop 1048576 3月   5 03:53 edits_inprogress_0000000000000000325
-rw-rw-r--. 1 hadoop hadoop    5699 2月  26 21:54 fsimage_0000000000000000323
-rw-rw-r--. 1 hadoop hadoop      62 2月  26 21:54 fsimage_0000000000000000323.md5
-rw-rw-r--. 1 hadoop hadoop    5699 3月   5 03:53 fsimage_0000000000000000324
-rw-rw-r--. 1 hadoop hadoop      62 3月   5 03:53 fsimage_0000000000000000324.md5
-rw-rw-r--. 1 hadoop hadoop       4 3月   5 03:53 seen_txid
-rw-rw-r--. 1 hadoop hadoop     207 3月   5 03:53 VERSION
```

# 备份命名服务器

- NameNode的热备；
- 定期合并fsimage和 fsedits，推送给 NameNode；
- 当Active NameNode出现故障时，快速切换为新的Active NameNode。

# 数据节点

- 负责把HDFS数据块读写到本地文件系统，每台从节点都运行一个，
- 启动DN线程的时候会向NN汇报block信息，向NN发送心跳保持与其联系（3秒一次），如果NN10分钟没有收到DN心跳，则认为其已经失联，并拷贝其上的数据块到其它DN
- 存储节点，保存数据块，负责所在物理节点的存储管理
- 一次写入，多次读取（不修改），不需要考虑的读取一致性。（回滚、SCN、关系数据库大多隔离级别、锁等）。
- 文件由数据块组成，典型块大小是128MB，每个数据块就是一个blk文件
- 数据块尽量散布到各个节点，冗余效果。

# 数据节点

```
-bash-3.2$ sudo ls -l /data/cache1/dfs/dn/current/
total 594992
-rw-r--r--  1 hdfs hadoop     20495 Apr 29 21:11 blk_-1265257027172478037
-rw-r--r--  1 hdfs hadoop       171 Apr 29 21:11 blk_-1265257027172478037_1896.meta
-rw-r--r--  1 hdfs hadoop       282 Apr 29 21:10 blk_1355466200351616098
-rw-r--r--  1 hdfs hadoop        11 Apr 29 21:10 blk_1355466200351616098_1411.meta
-rw-r--r--  1 hdfs hadoop  32201859 Apr 29 21:13 blk_-1363349205939896111
-rw-r--r--  1 hdfs hadoop    251587 Apr 29 21:13 blk_-1363349205939896111_2355.meta
-rw-r--r--  1 hdfs hadoop     20349 Apr 29 21:11 blk_-1486089392719042041
-rw-r--r--  1 hdfs hadoop       167 Apr 29 21:11 blk_-1486089392719042041_1593.meta
-rw-r--r--  1 hdfs hadoop   1222666 Apr 29 21:12 blk_1841498388503862976
-rw-r--r--  1 hdfs hadoop      9563 Apr 29 21:12 blk_1841498388503862976_2595.meta
-rw-r--r--  1 hdfs hadoop   9250764 May 25 08:22 blk_1922063674315104428
-rw-r--r--  1 hdfs hadoop     72279 May 25 08:22 blk_1922063674315104428_32513112.meta
-rw-r--r--  1 hdfs hadoop   4509245 Apr 29 21:15 blk_-2039669051003294713
-rw-r--r--  1 hdfs hadoop     35239 Apr 29 21:15 blk_-2039669051003294713_4346.meta
```

# 客户端

- 文件切分
- 与NameNode交互，获取文件位置信息；
- 与 DataNode 交互 ， 读取或者写入数据；
- 管理HDFS；
- 访问HDFS。

# 事务日志和镜像文件

- NameNode两个重要文件
  - ✓ fsimage：元数据镜像文件（保存文件系统的目录树）
  - ✓ edits：元数据操作日志（针对目录树的修改操作），被写入共享存储系统中，比如NFS、JournalNode
- 元数据镜像
  - ✓ 内存中保存一份最新的
  - ✓ 内存中的镜像=fsimage+edits
- 合并fsimage与edits
  - ✓ Edits文件过大将导致NameNode重启速度慢
  - ✓ Standby Namenode负责定期合并它们

# 数据块

- 文件被切分成固定大小的数据块
  - ✓ 默认数据块大小为128MB，可配置
  - ✓ 若文件大小不到128MB，则单独存成一个block
- 为何数据块如此之大
  - ✓ 数据传输时间超过寻道时间（高吞吐率）
- 一个文件存储方式
  - ✓ 按大小被切分成若干个block，存储到不同节点上
  - ✓ 默认情况下每个block有三个副本

# 数据块

1000TB数据集群，存储在各种RAID 级别上的风险，以数据丢失的概率为衡量标准（3年内的统计）：RAID-5最不可靠，会损 失2TB数据，而三副本方案最靠谱

- ✓ 假设1组磁盘里（可能是RAID1, RAID5等）由m个磁盘组成，当m个（或更多）磁盘损坏会造成数据丢失，比如：
- ✓ RAID5只允许损坏一个磁盘，因此m=2；
- ✓ RAID6只允许损坏两个磁盘，因此m=3

| m,n | 普遍配置盘数 | 对应RAID级别 | 3年内丢数据概率P | 1组磁盘丢失的数据量 | 1PB集群丢失的数据量 |
|---|---|---|---|---|---|
| m=3 n=34 | 36盘 | RAID 6 | 0.00468% | 1.53G | 49.1G |
| m=3 n=22 | 24盘 | RAID 6 | 0.001207% | 253.0M | 12.7G |
| m=3 n=10 | 12盘 | RAID 6 | 0.000094% | 7.90M | 1011.4M |
| m=2 n=35 | 36盘 | RAID 5 | 2.31419% | 805.7G | 23.7T |
| m=2 n=23 | 24盘 | RAID 5 | 0.992204% | 223.5G | 10.2T |
| m=2 n=11 | 12盘 | RAID 5 | 0.216882% | 22.2G | 2.22T |
| m=3 n=3 | | 3 副本 | 0.000001% | 8.44K | 8.44M |
| m=2 n=2 | | 2 副本/ RAID 1 | 0.003952% | 41.4M | 41.4G |

# 数据块

假设单磁盘损坏率 = p ( 0 < p < 1 ), 单磁盘完好的概率 = (1 - p )

1个磁盘的所有可能性只有损坏, 不损坏两种状态, 概率之和是1:

$$1 = (1-p) + p$$

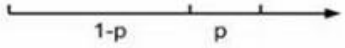
2个磁盘时,每个磁盘的2个状态两两组合, 假设2个磁盘分别是a和b, 所有可能的状态是:

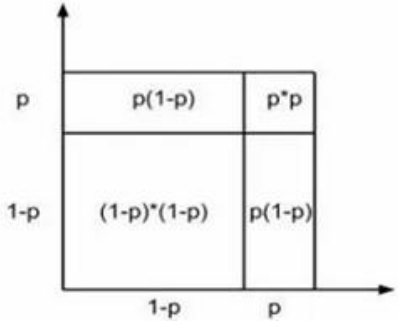
- *a,b都损坏, 概率是p*p*
- *a损坏,b完好,概率是p*(1-p)*
- *b损坏,a完好,概率是p*(1-p)*
- *a,b都完好,概率是(1-p)*(1-p)*

所有概率之和还是1:

$$1 = p*p + 2*p*(1-p) + (1-p)*(1-p)$$
$$= [(1-p) + p]^2$$

2个磁盘组合之后的所有的损坏情况由每个磁盘的每1种可能状态组合而成.
1磁盘和2磁盘的概率分布示意图:




多个磁盘的情况下,用多个维度,来覆盖所有的情况:

$$1 = [(1-p) + p]^n$$

通过朴素的2项式展开:

$$1 = C_n^0(1-p)^n p^0 + C_n^1(1-p)^{(n-1)}p^1 + C_n^2(1-p)^{(n-2)}p^2 + \ldots + C_n^n(1-p)^0 p^n$$

这个式子表示:

*所有的可能性 = 无损坏概率 + 1磁盘损坏概率 + 2磁盘损坏概率 + .... + 全部损坏的概率*

假设1组磁盘里(如能是RAID1, RAID5, RAID6等)由n个磁盘组成,当m个(或更多)磁盘损坏会造成数据丢失,如:

- RAID5里,只允许损坏1个磁盘,因此m=2
- RAID6里,只允许损坏2个磁盘,因此m=3
- 等等..

那么整个磁盘组丢失数据的概率为所有损坏磁盘数小于m的概率和:

$$P = 1 - C_n^0(1-p)^n p^0 - C_n^1(1-p)^{(n-1)}p^1 - \ldots - C_n^{m-1}(1-p)^{(n-m+1)}p^{m-1}$$

以RAID5举例,m=2,总磁盘数n:

$$P = 1 - C_n^0(1-p)^n p^0 - C_n^1(1-p)^{(n-1)}p^1$$

现在假设目前的sata盘损坏率是每年4%-7%, 折算成日损坏率是:

$$p \sim= 0.0198\%$$

这里用日损坏率来计算,是因为从经验上来讲,硬件实现的RAID5/6,rebuild数据时间大约是1天以内.
因此如果2个磁盘的故障发生在不同的2天里,故障的概率不会叠加,因为数据已经恢复了.

所以上面的式子,得出的将是1天里数据丢失的概率:

对于 10盘的RAID6, n = 10, m = 3,代入得出:

$$P = 9.41897e\text{-}10$$

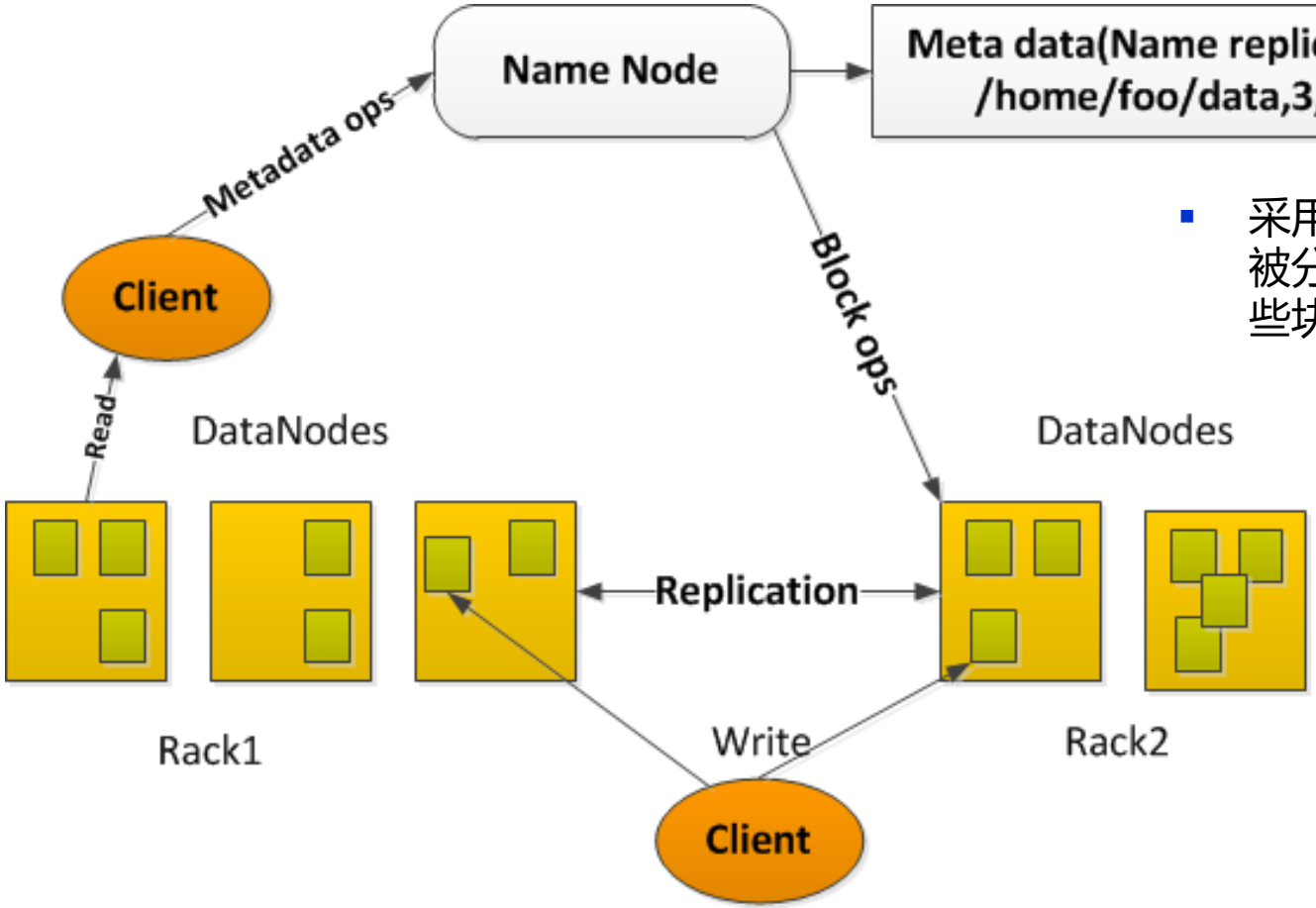
未来3年里,要保证每1天都不丢数据,才算是不丢数据.因此3年内(约1000天),数据丢失的概率是:

$$P_{3year} \sim= P * 1000 = 9.41897e\text{-}07$$

实际生产环境中:

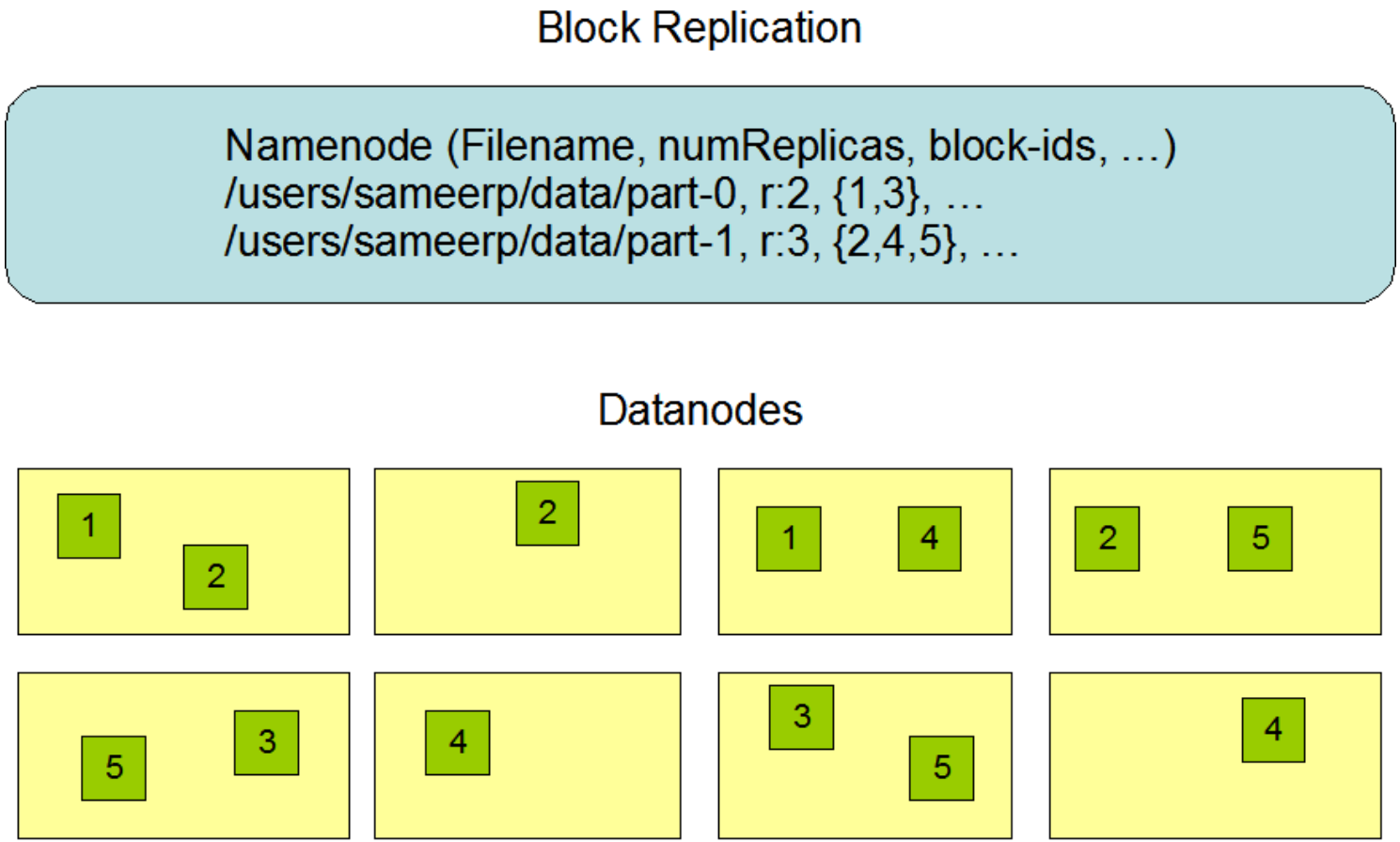
- 如果整个RAID6容量10磁盘是10T, 那么3年后丢的数据量大概是9.41M
- 如果整个RAID5容量11磁盘是10T, 那么3年后丢的数据量大概是22.2G

# HDFS架构

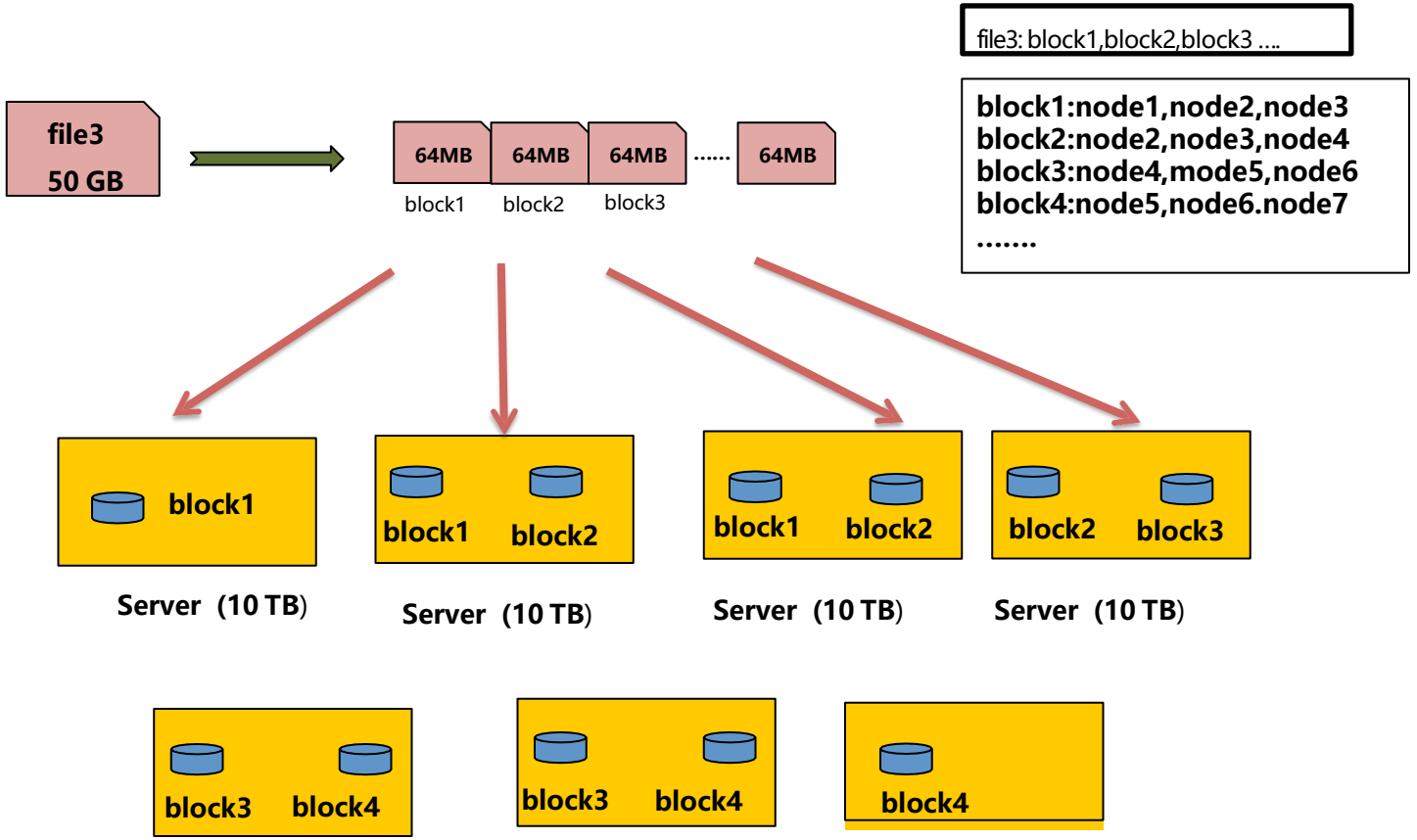



- 采用主从结构，一个文件其实被分成一个或多个数据块，这些块存储在一组Datanode上

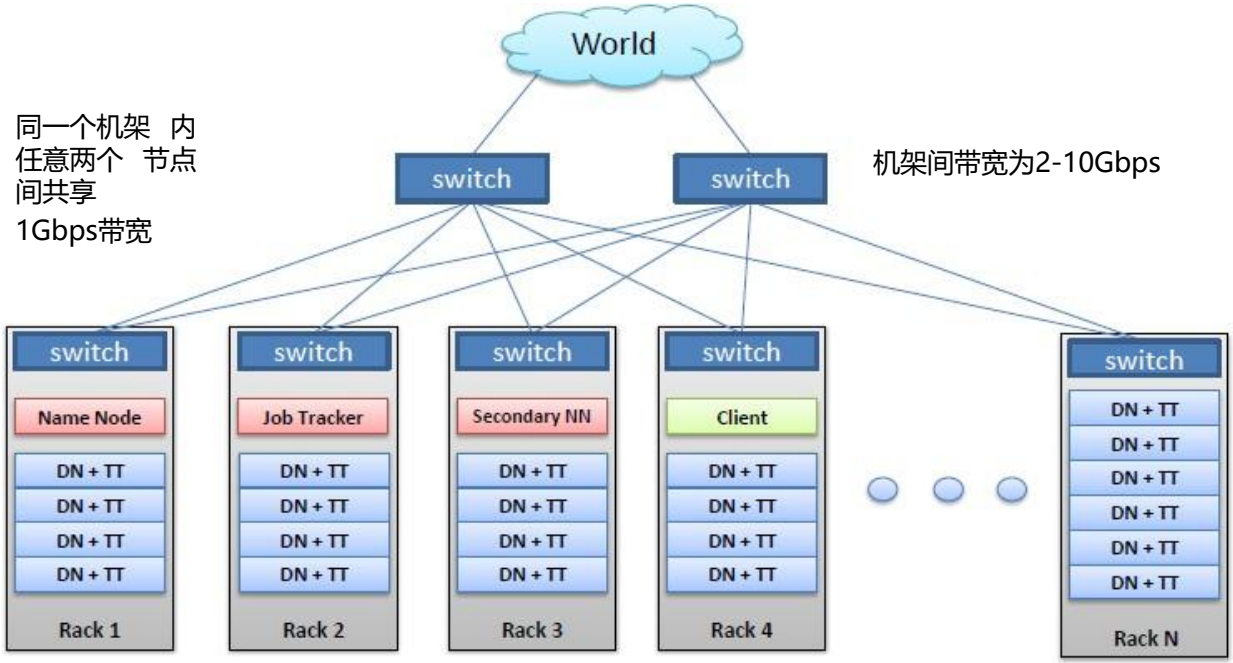
# HDFS架构


Block Replication
Namenode (Filename, numReplicas, block-ids, …)
/users/sameerp/data/part-0, r:2, {1,3}, …
/users/sameerp/data/part-1, r:3, {2,4,5}, …
Datanodes
1
2
2
1
4
2
5
5
3
4
3
5
4

# HDFS架构


file3: block1,block2,block3 ....
block1:node1,node2,node3
block2:node2,node3,node4
block3:node4,mode5,node6
block4:node5,node6.node7
.....
file3
50 GB
64MB
64MB
64MB
.....
64MB
block1
block2
block3
block1
Server（10 TB）
block1
block2
Server（10 TB）
block1
block2
Server（10 TB）
block2
block3
Server（10 TB）
block3
block4
block3
block4
block4

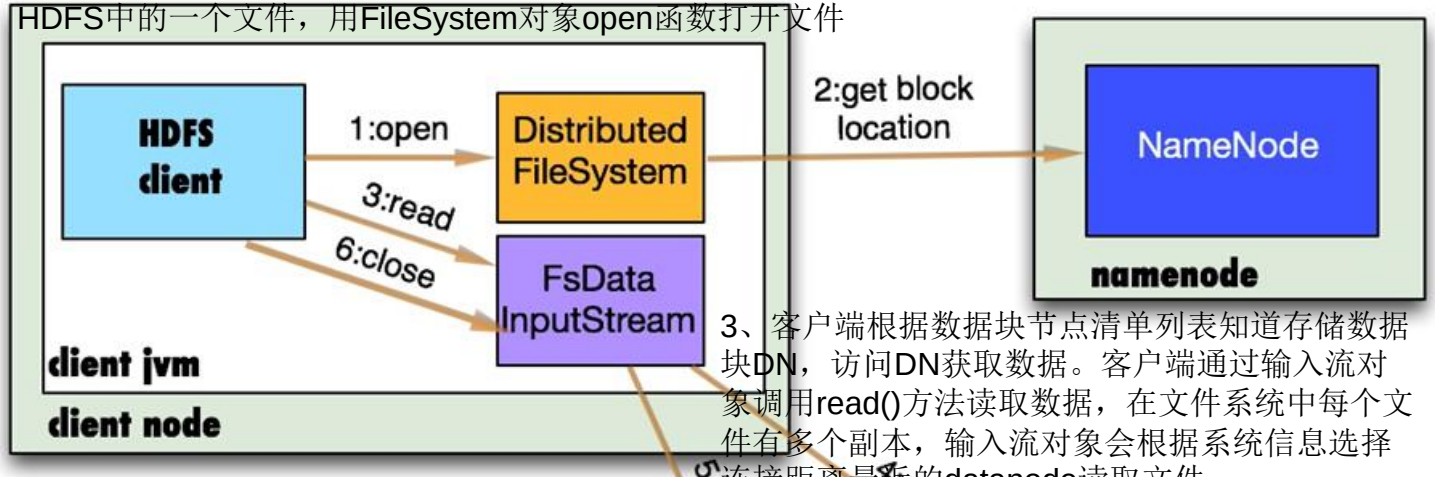
# HDFS网络拓扑




每个机架通常有16-64 个节点

# 客户端读取HDFS中的数据

1、客户端就是集群里的某一个提交作业的节点。客户端要访问HDFS中的一个文件，用FileSystem对象open函数打开文件

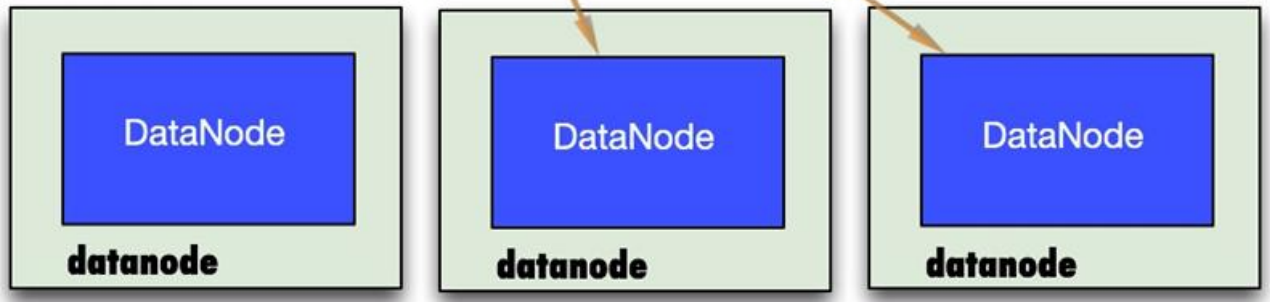



2、从NN获得组成这个文件的数据块位置列表 。NN并不参与数据实际传输，只起到查询作用。FileSystem对象通过rpc与管理节点联系获得文件的第一个block的location，同一block按照重复数会返回多个locations，这些locations按照hadoop拓扑结构排序，距离客户端近的排在前面。第1,2步会返回一个FsDataInputStream对象，该对象会被封装成DFSInputStream对象，DFSInputStream可以方便的管理datanode和namenode数据流

3、客户端根据数据块节点清单列表知道存储数据块DN，访问DN获取数据。客户端通过输入流对象调用read()方法读取数据，在文件系统中每个文件有多个副本，输入流对象会根据系统信息选择连接距离最近的datanode读取文件。

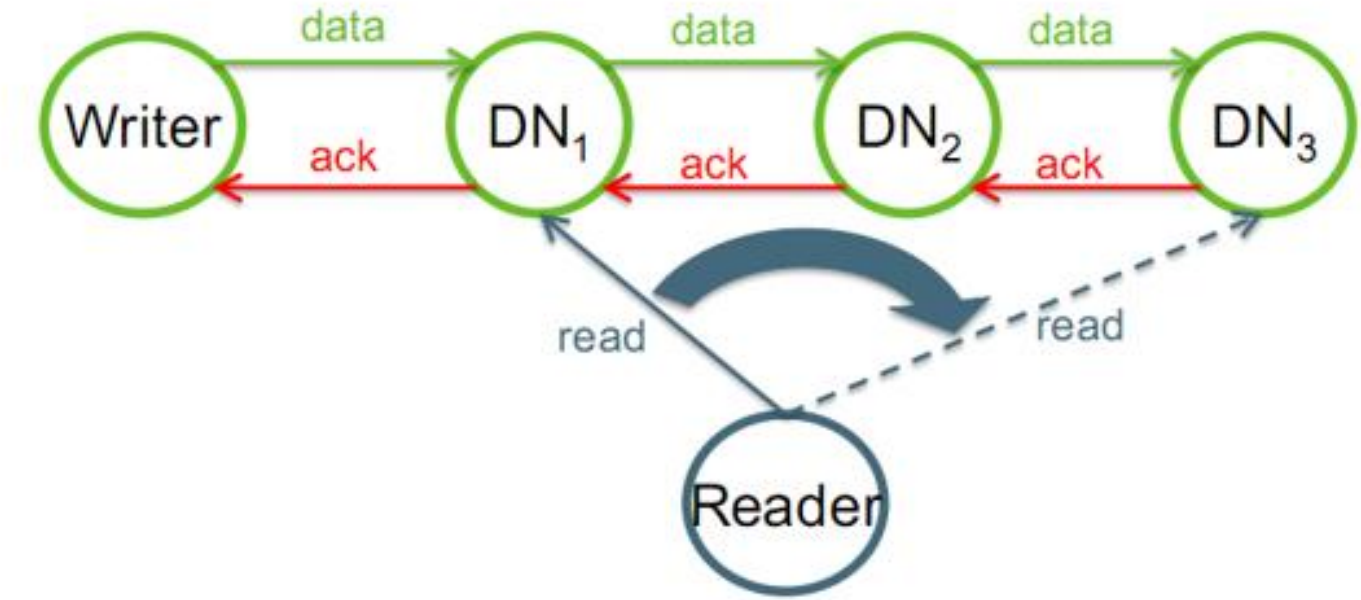
4-5、客户端通过文件类的API来读取HDFS中的文件内容，数据从datanode源源不断的流向客户端。如果第一块的数据读完了，就会关闭指向第一块的datanode连接，接着读取下一块。这些操作对客户端来说是透明的，客户端的角度看来只是读一个持续不断的流。

6、通过输入流对象关闭读进程。如果第一批block都读完了，DFSInputStream就会去namenode拿下一批blocks的location，然后继续读，如果所有的块都读完，这时就会关闭掉所有的流。



# 客户端读取HDFS中的数据

写数据的时候同时可以读取数据
—在管道中，可以通过任何一个DN读取数据
—当一个DN出现故障时，可以通过其他DN读取同样的数据

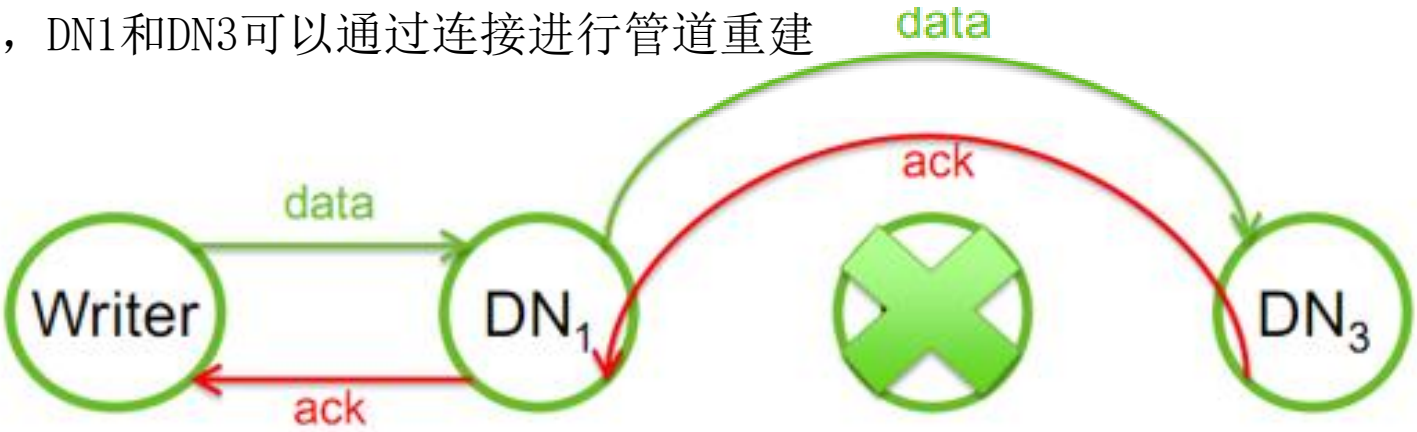


# 客户端读取HDFS中的数据

- 客户端读取损坏数据块流程：
- 由于HDFS存储每个数据块的副本，它可以通过复制完好的数据副本来修复损坏的数据块，进而得到一个新的完好无损的副本。
  （1）客户读取数据块时，检测到错误，向namenode报告已损坏的数据块以及正在尝试读取操作的datanode，最后才抛出CheckSumException异常。
  （2）namenode将这个已损坏的数据块的复本标记位损坏，这里不需要client直接和datanode联系，或尝试将这个副本复制到另一个datanode；
  （3）namenode安排这个数据块的一个副本复制到另一个datanode，如此一来，数据块的副本因子又回到期望水平。

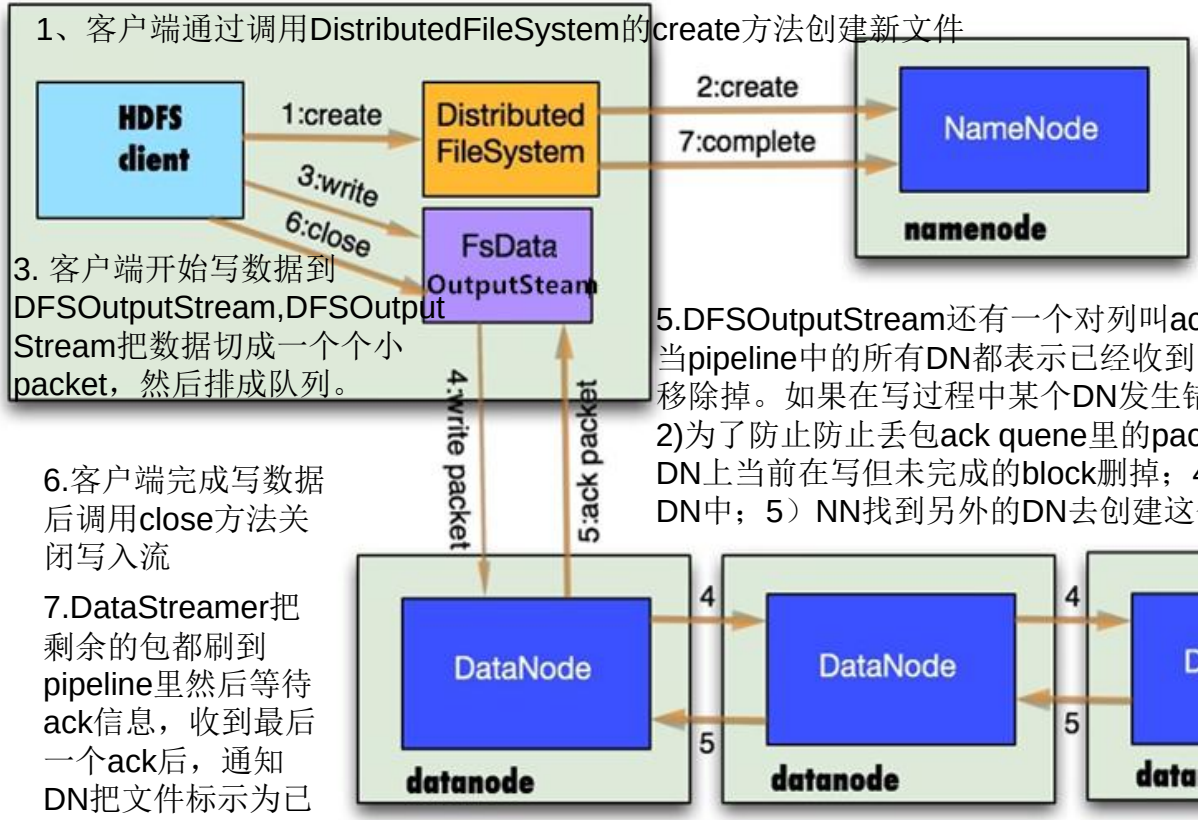
# 客户端读取HDFS中的数据

当DN2失败时，DN1和DN3可以通过连接进行管道重建




仅有DN1成功，DN2和DN3全部失败

# 客户端将数据写入HDFS




2. FileSystem通过RPC调用NN去创建一个没有blocks关联的新文件，创建前，NN会做各种校验，比如文件是否存在，客户端有无权限去创建等。如果校验通过， NN就会记录下新文件，否则就会抛出IO异常。前两步结束后会返回FSDataOutputStream的对象，与读文件的时候相似，FSDataOutputStream被封装成DFSOutputStream协调NN和DN。

5.DFSOutputStream还有一个对列叫ack quene，由packet组成，等待DN收到响应，当pipeline中的所有DN都表示已经收到的时候，这时ack quene才会把对应的packet包移除掉。如果在写过程中某个DN发生错误，会采取以下几步：1) pipeline被关闭掉；2)为了防止防止丢包ack quene里的packet会同步到data quene里；3)把产生错误的DN上当前在写但未完成的block删掉；4）block剩下的部分被写到剩下的两个正常的DN中；5）NN找到另外的DN去创建这个块的复制。这些操作对客户端来说是透明的。

4.DataStreamer会去处理接受data quene，先问询NN这个新的block最适合存储的在哪几个DN里，找到3个最适合的DN，把他们排成一个pipeline。DataStreamer把packet按队列输出到管道的第一个DN中，第一个DN又把packet输出到第二个DN中，以此类推。

# 客户端将数据写入HDFS

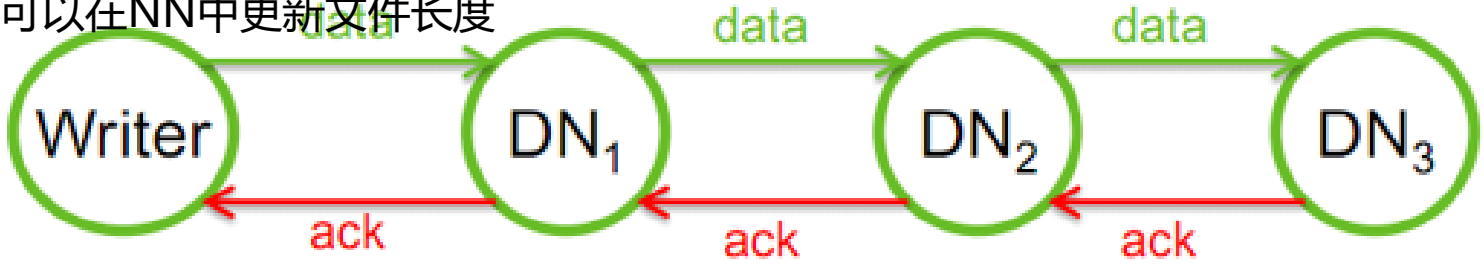
Java flush ( or C++ fflush )
——把缓冲区的数据强制输出

HDFS hflush
——将写入的数据刷到每个DN中
——确保读取数据时，数据可见
——数据可以在DN的内存中

HDFS sync
——与本地文件系统同步
——可以在NN中更新文件长度

Hadoop 1
——文件不可更改
——Write-once-read-many模式

Hadoop 2
——可以向文件中追加内容
——hflush和hsync
——读一致性
——失败时替换DN



# HDFS优点与缺点

- 高容错性
  - ✓ 数据自动保存多个副本
  - ✓ 副本丢失后，自动恢复
- 适合批处理
  - ✓ 移动计算而非数据
  - ✓ 数据位置暴露给计算框架
- 适合大数据处理
  - ✓ GB、TB、甚至PB级数据
  - ✓ 百万规模以上的文件数量
  - ✓ 10K+节点规模
- 流式文件访问
  - ✓ 一次性写入，多次读取
  - ✓ 保证数据一致性
- 可构建在廉价机器上
  - ✓ 通过多副本提高可靠性
  - ✓ 提供了容错和恢复机制

- 低延迟数据访问
  - ✓ 比如毫秒级
  - ✓ 低延迟与高吞吐率
- 小文件存取
  - ✓ 占用NameNode大量内存
  - ✓ 寻道时间超过读取时间
- 并发写入、文件随机修改
  - ✓ 一个文件只能有一个写者
  - ✓ 仅支持append

# HDFS特点

# HDFS特点

- 海量数据存储
- 数据块
  - 校验和
  - 回收站
  - 元数据保护
  - 快照
  - 心跳机制
  - 安全模式
  - 容错性
- 简单一致性模型
- 类Linux文件权限
- 流式数据访问
- 序列化
  - 联邦
  - 压缩性能

# 海量数据存储

HDFS是一个容量巨大、具有高容错性的磁盘

- HDFS支持异构存储媒介，HDFS各个存储节点可以指定若干不同的存储媒介，如HDD、SSD、内存等。

# 数据块

- 将文件切分成等大的数据块，存储到多台机器上。
- 将数据切分、容错、均衡负载等功能均透明化

- 缺省128MB，如果上传文件小于该值，仍然会占用一个Block的命名空间，物理存储上不会占用128MB的空间
- Block大小和副本数由Client端上传文件到HDFS时设置，副本数可以变更
- Block上传后不可以变更

# 类Linux文件权限

- 阻止好人做错事，而不是阻止坏人做坏事。HDFS相信，你告诉我你是谁，我就认为你是谁
- r:read; w:write; x:execute，权限x对于文件忽略，对于文件夹表示是否允许访问其内容
- Linux系统用户zhangsan使用hadoop命令创建一个文件，那么这个文件在HDFS中owner就是zhangsan

```
[root@master ~]# hadoop fs -ls /apps/hive/warehouse
Found 21 items
drwxr-xr-x   - root hdfs          0 2014-09-11 18:20 /apps/hive/warehouse/_tmp.b_qs_hbase_my
drwxr-xr-x   - root h                                                    mp.b_ry_hcy_jbxx_base
drwxr-xr-x   - root h                                                    mp.b_wj_gjwj_hbase_my
drwxr-xr-x   - root h                                                    g_lkxx_group_my
drwxr-xr-x   - root hdfs          0 2014-09-11 18:11 /apps/hive/warehouse/b_lg_lkxx_hbase_my
drwxr-xr-x   - root hdfs          0 2014-09-11 18:12 /apps/hive/warehouse/b_lg_lkxx_my
drwxr-xr-x   - root hdfs          0 2014-09-11 18:14 /apps/hive/warehouse/b_lg_lkxx_result_my
drwxr-xr-x   - root hdfs          0 2014-09-11 18:07 /apps/hive/warehouse/b_lg_lkxx_temp
```

drwxr-xr-x - root hdfs

# 简单一致性模型

- 对文件采用一次性写多次读的逻辑设计，即是文件一经写入，关闭，就再也不能修改，如果需要改的时候，先删再插入，简化了数据一致性问题，提高高吞吐率数据访问

# 序列化

- 序列化（serialization)将结构化对象转化为字节流，以便在网络上传输或写到磁盘永久存储。反序列化（deserialization)是指将字节流转回结构化对象的逆过程。
- 序列化在分布式数据处理两大领域经常出现：进程间通信(RPC调用)和永久存储
- Hadoop序列化格式Writable：Hadoop定义了自己的序列化格式Writable，Writable是Hadoop的核心，很多Mapreduce程序都会为键和值使用它

# 序列化

## 为什么没有采用Java本身的序列化机制？

相比Java的序列化而言，Hadoop的序列化更紧凑快速，从而为RPC框架提供了更高的吞吐量和更少的延时

Java本身的序列化以后的二进制编码：112字节

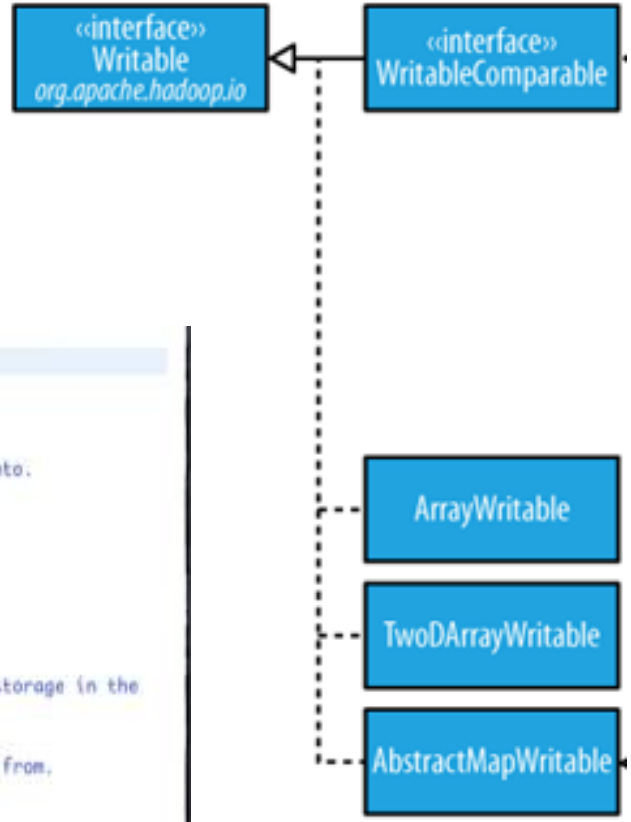
```
AC ED 00 05 73 72 00 1C  6F 72 67 2E 68 61 64 6F   ....sr.. org.hado
6F 70 69 6E 74 65 72 6E  61 6C 2E 73 65 72 2E 42   opintern al.ser.B
6C 6F 63 6B E7 80 E3 D3  A6 B6 22 53 02 00 03 4A   lock.... .."S...J
00 07 62 6C 6F 63 6B 49  64 4A 00 0F 67 65 6E 65   ..blockI dJ..gene
72 61 74 69 6F 6E 53 74  61 6D 70 4A 00 08 6E 75   rationSt ampJ..nu
6D 42 79 74 65 73 78 70  6C 55 67 95 68 E7 92 FF   mBytesxp lUg.h...
00 00 00 00 03 61 BB 8B  00 00 00 00 02 59 EC CB   .....a.. .....Y..
```

Hadoop序列化以后的二进制编码：24字节

```
AC ED 00 05 73 72 00 28  6F 72 67 2E 68 61 64 6F   ....sr.( org.hado
6F 70 69 6E 74 65 72 6E                            opintern
```

# 序列化

- Hadoop定义了自己的序列化格式Writable，Writable是Hadoop的核心，很多Mapreduce程序都会为键和值使用它，Writable接口定义了两个方法，一个将其状态写到DataOutput二进制流（往二进制流里面写数据），一个从DataInput二进制流读取其状态（从二进制流里读取数据）。




```
// 将序列化流写入DataOutput
```

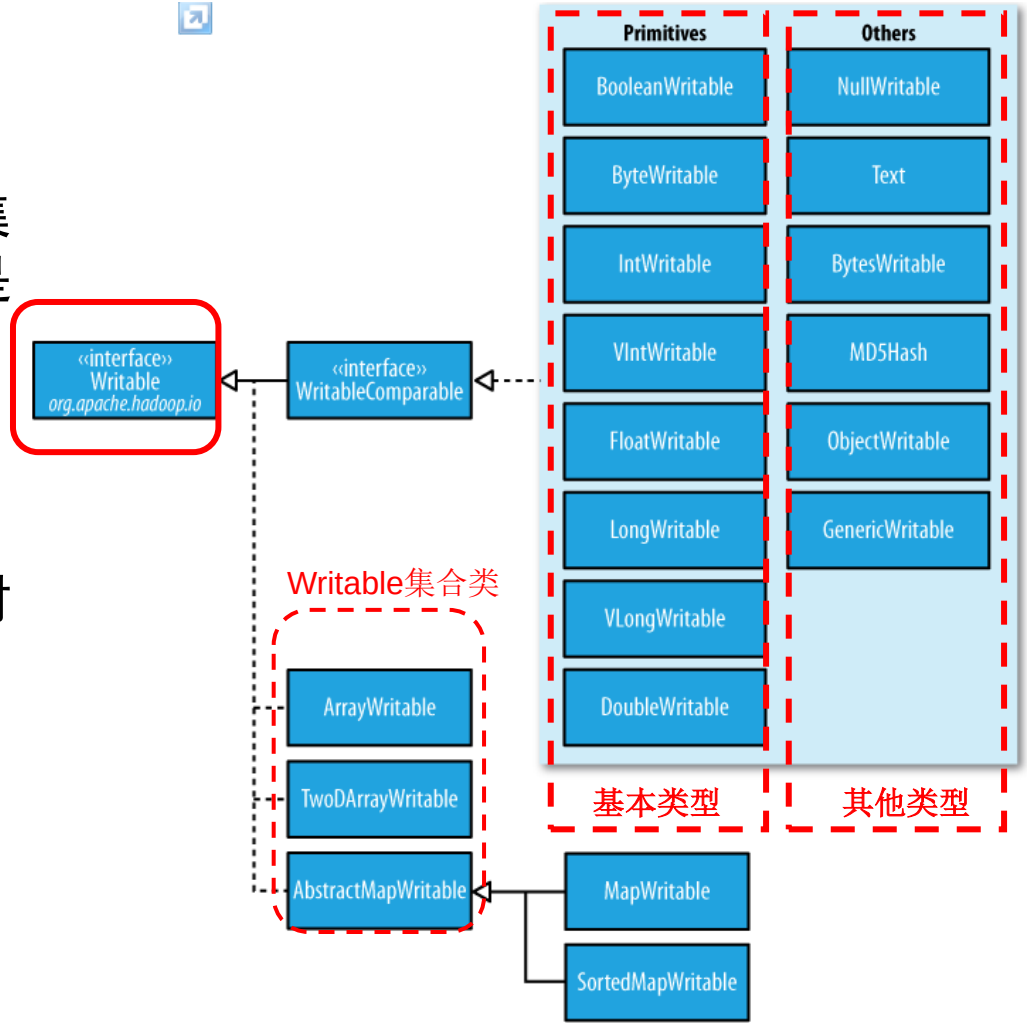
```
//从DataInput流读取二进制
```

```
public interface Writable {
  /**
   * Serialize the fields of this object to <code>out</code>.
   *
   * @param out <code>DataOuput</code> to serialize this object into.
   * @throws IOException
   */
  void write(DataOutput out) throws IOException;

  /**
   * Deserialize the fields of this object from <code>in</code>.
   *
   * <p>For efficiency, implementations should attempt to re-use storage in the
   * existing object where possible.</p>
   *
   * @param in <code>DataInput</code> to deseriablize this object from.
   * @throws IOException
   */
  void readFields(DataInput in) throws IOException;
}
```

# 序列化类

- Writerable类的层次结构
- Writable集合类，共有4个集合类，其中ArrayWritable是对Writable的数组的实现，TwoDArrayWritable 是对Writable的二维数组的实现，MapWritable是对Map的实现，SortedMapWritable是对SortedMap的实现。



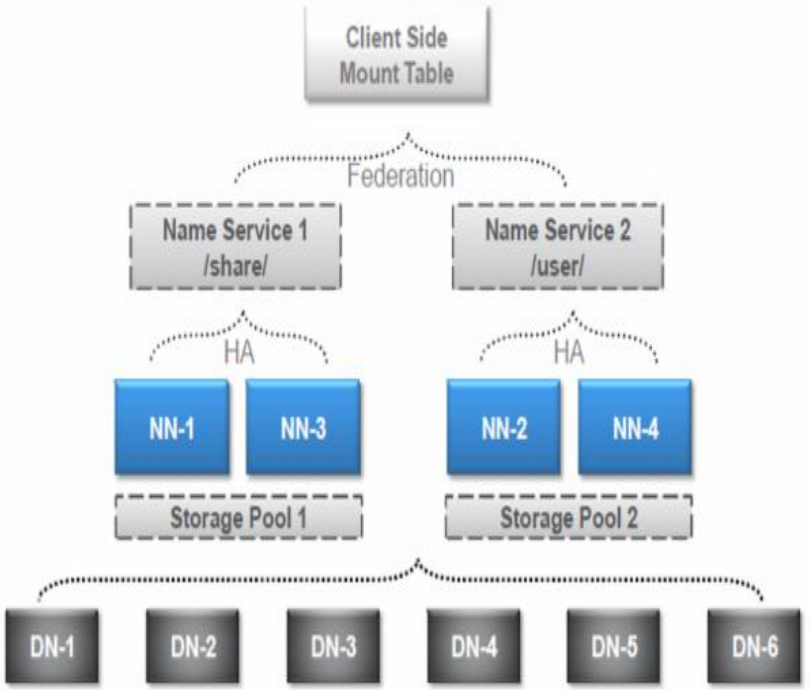
# 序列化类

- Java基本类型对应Writable类

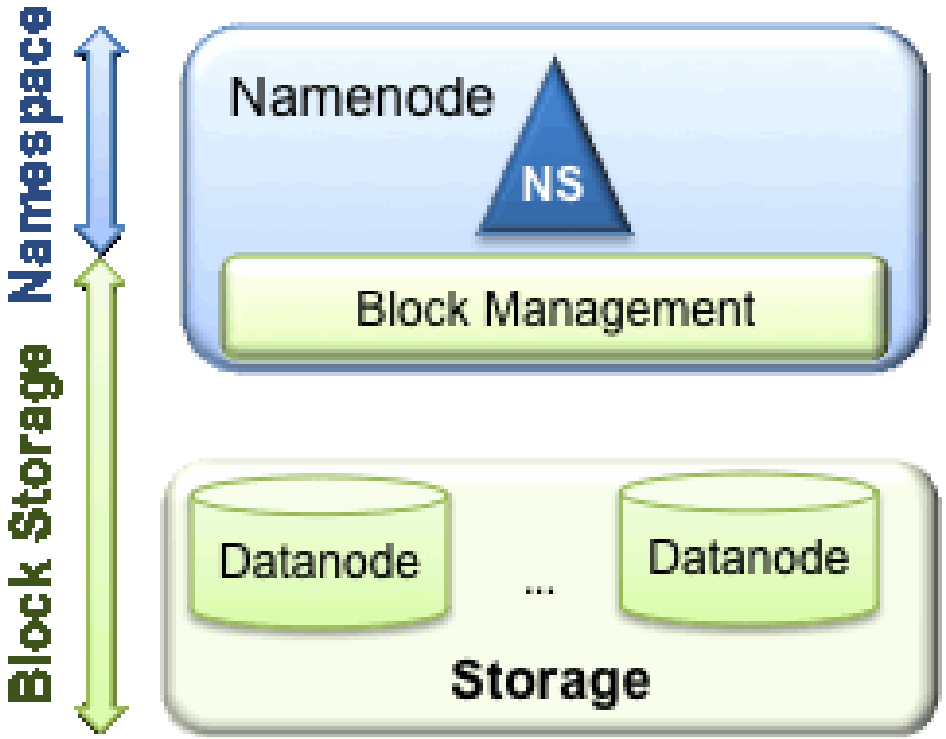
| Java 基本类型 | Writable 实现 | 序列化大小(字节) |
| --- | --- | --- |
| boolean | BooleanWritable | 1 |
| byte | ByteWritable | 1 |
| int | IntWritable | 4 |
| | VintWritable | 1~5 |
| float | FloatWritable | 4 |
| long | LongWritable | 8 |
| | VlongWritable | 1-9 |
| double | DoubleWritable | 8 |

# HDFS联邦

- 火车站售票口（NN），站内火车车厢（DN），当只有一个售票口（hadoop1），整个火车站运力受限于售票口售票情况，节假日会出现买票排长队情况。为了缓解当前情况，只有多增加售票口（HDFS联邦）部署了多个集群，集群的DN相同，每个NN存储不同的元数据。HDFS1架构只允许存在一个namespace。一个NN管理一个namespace。HDFS联邦通过增加多个NN/namespace来解决这个先前架构的限制。NN是联邦的，他们是独立的，不会要求相互协作。每个DN都在集群中的所有NN注册。
- HDFS联邦由多个NameService组成，每个NameService又由一个或两个(HA)NN组成，每个NN会定义一个存储池，单独对外提供服务,多个NN共用集群里DN上的存储资源。使用客户端挂载表把不同的目录映射到不同的NN上，通过目录自动对应NN，使联邦的配置改动对应用透明



# HDFS联邦




Namespace由目录、文件和块组成；支持所有命名空间对文件和目录的操作。

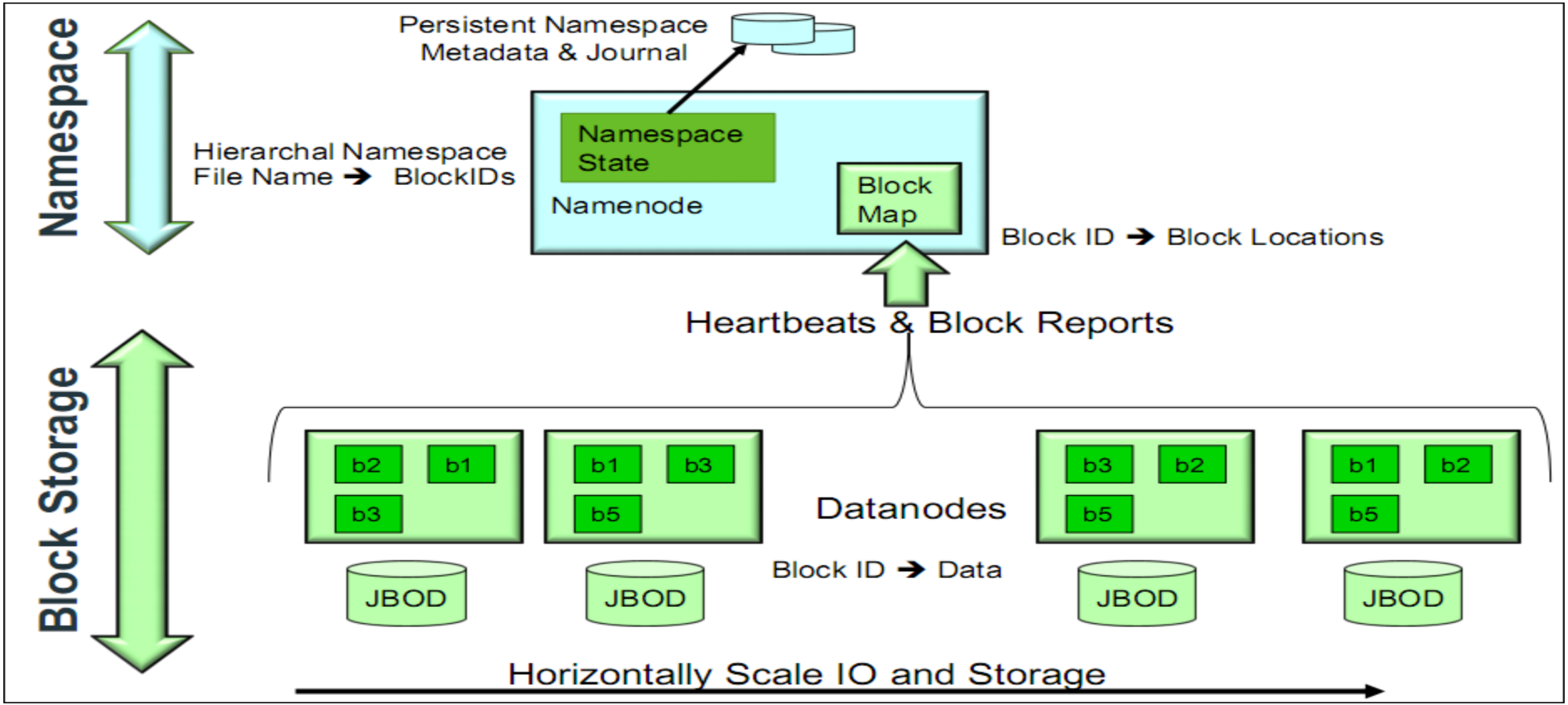
Block Storage Service由Block Management和Storage组成。

Block Management提供datanode集群成员关系，注册信息和周期性的心跳；处理块报告，维护块位置；支持块相关的操作，如创建、删除、修改等；管理副本数量、位置，删除多余副本；

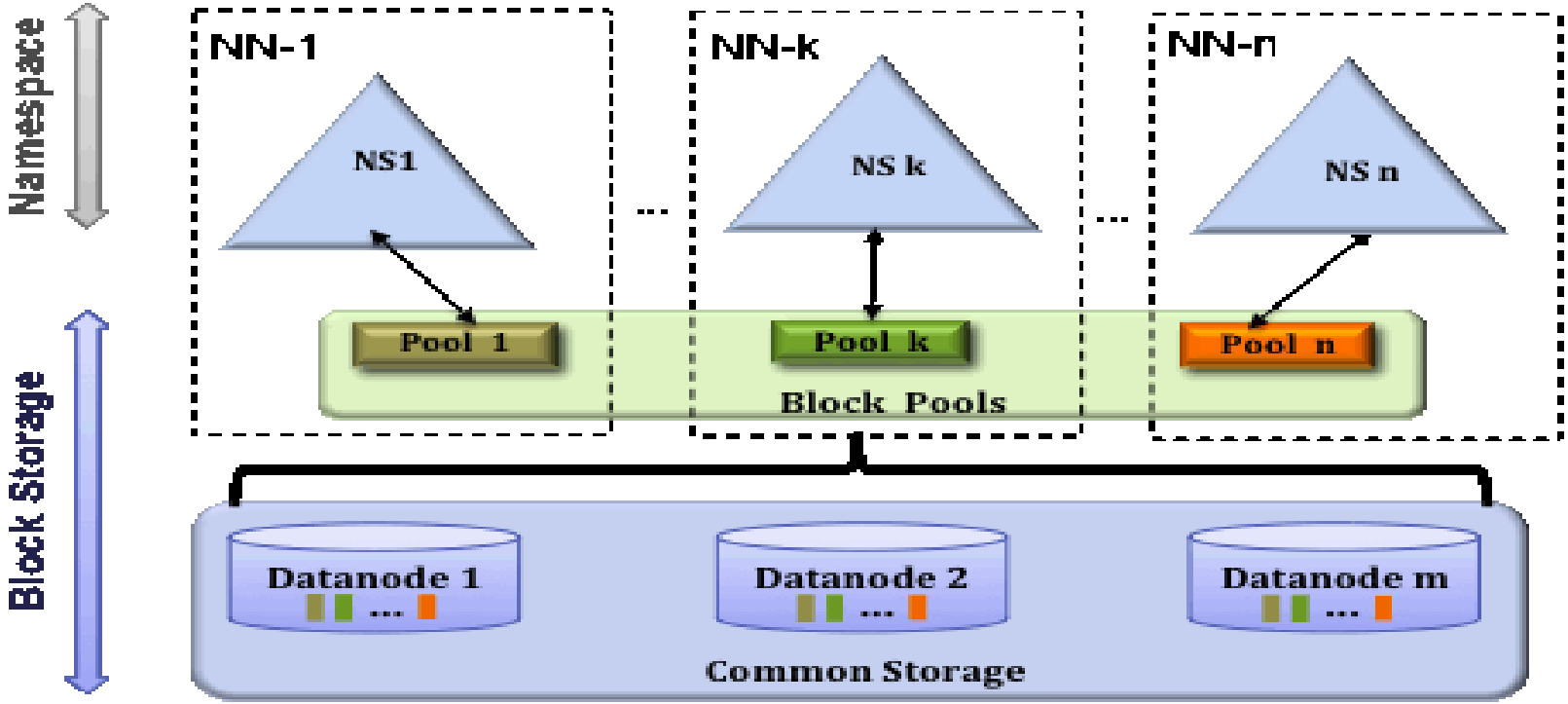
Storage是DN提供的。

# HDFS联邦

每一个命名空间使用一个块池（block pool）。



# HDFS联邦

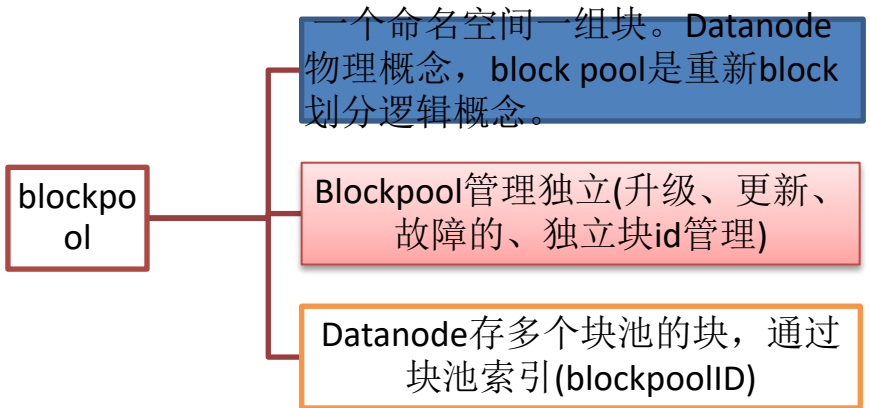

Namespace
Block Storage
NN-1
NN-k
NN-n
NS1
NS k
NS n
Pool 1
Pool k
Pool n
Block Pools
Datanode 1
Datanode 2
Datanode m
Common Storage

# HDFS联邦和HDFS1

| 类别 | hadoop1 hdfs | HDFS Federation |
| --- | --- | --- |
| 命名空间数量 | 1 | 多个 |
| 块分布 | 一组块 | 多组块，块池 |
| 组成 | 一个NN和一组DN | 多个NN和一组DN |
| DN | 为唯一的NN存储块 | 每个DN为多个块池存储块 |

# 块池

- 块池是块的集合，属于单个命名空间的一组block(块)。DN存储着集群中所有块池中的块。块池管理相互独立。一个namespace可以独立的生成块ID，不需要与其他namespace协调。一个NN失败不会导致DN的失败，这些DN还可以服务其他NN。同一个DN中可以存着属于多个块池。块池允许一个命名空间在不通知其他命名空间的情况下为一个新的block创建Block ID。一个NN失效不会影响其下的DN为其他NN的服务。
- DN与NN建立联系并开始会话后自动建立Block pool。每个block都有一个唯一的标识，称为扩展的块ID（Extended Block ID）= BlockID+BlockID。扩展的块ID在HDFS集群之间都是唯一的，为以后集群归并创造了条件。在HDFS中，所有的更新、回滚都是以NN和BlockPool为单元发生的。
- DN中的数据结构都通过块池ID索引，即DN中的BlockMap，storage等都通过BPID索引。同一HDFS Federation中不同的Namenode/BlockPool之间没有什么关系



# Namespace卷和Cluster ID

- 一个Namespace和它的Block Pool合在一起称作Namespace Volume。Namespace Volume是一个独立完整的管理单元。当一个NN/Namespace被删除，与之相对应的块池也被删除。ClusterID是用来标示集群中所有节点的。当NN格式化时，这个id会自动产生。HDFS联邦中添加了ClusterID用来区分集群中的每个节点。当格式化一个NN时，ClusterID会自动生成或者手动提供。在格式化同一集群中其他NN时会用到这个ClusterID。

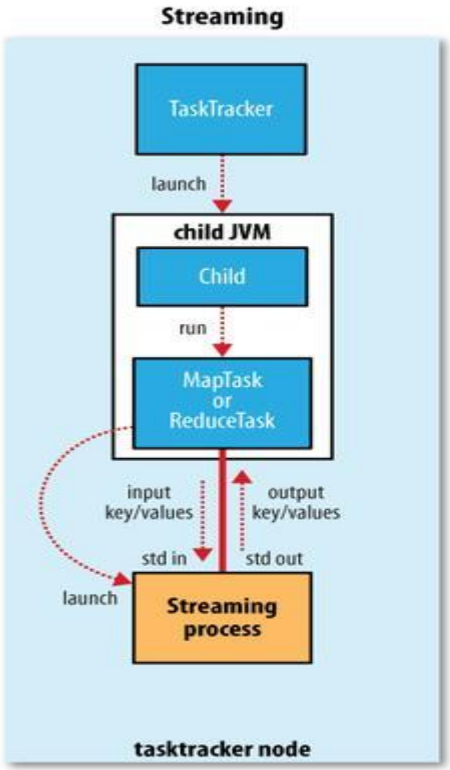
```
[hadoop@master current]$ pwd
/usr/local/webserver/hadoop/dfs/name/current
[hadoop@master current]$ cat VERSION
#Sun Mar 05 03:53:16 CST 2017
namespaceID=142655448
clusterID=CID-747fb9c5-e7ab-41fd-84f5-376a7d25c5ab
cTime=0
storageType=NAME_NODE
blockpoolID=BP-1547122488-192.168.225.136-1487870487450
layoutVersion=-63
```

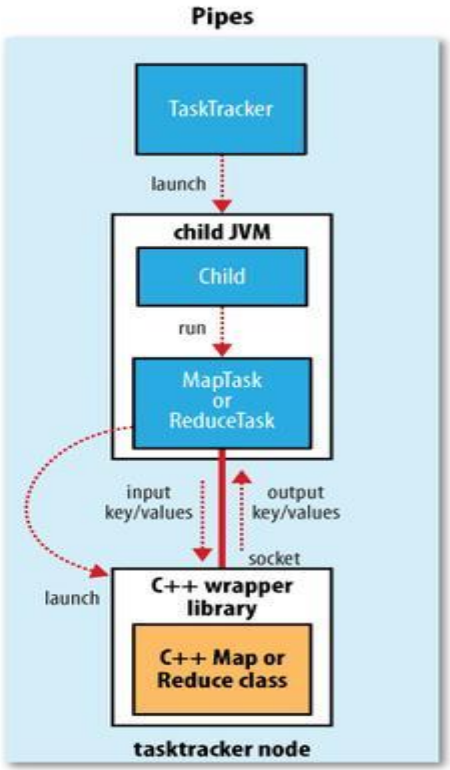
# 多Namespace卷的好处

- HDFS集群支持存储的水平扩展，但是namespace不能。对于大集群部署或者大量小文件存储时，使用多namespace会更好。
- 之前的设计中，文件系统操作效率受制于单个NN。现在，多个NN提高了文件读写操作效率。
- 一个NN在多用户环境中没有隔离性。使用多namespace，不同的应用或者用户可以隔离在不同的namespace中。
- 联邦配置是向后兼容的，之前架构下的应用不经修改的就可以工作。
- 联邦中有NameServiceID。匹配的namenode、secondary、backup、checkpointer节点，都有相同的NameServiceID.

# 流式数据访问

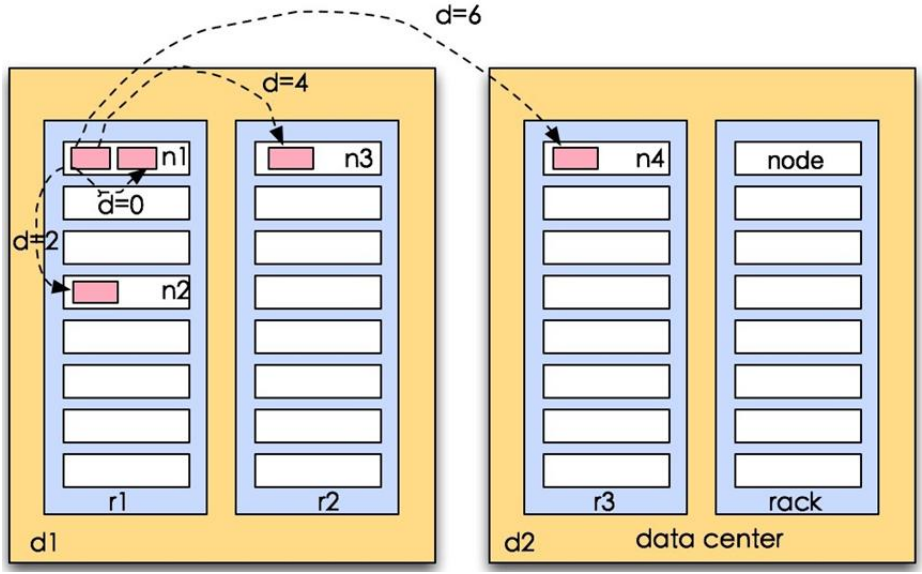
- 系统吞吐量比反应时间更重要。HDFS是设计成适合批量处理的，而不是用户交互式的。数据批量读取而非随机读写。流式数据，像流水一样，不是一次过来而是一点一点“流”过来。处理流式数据也是一点一点处理。如果是全部收到数据以后再处理，那么延迟会很大，而且在很多场合会消耗大量内存
- 流式数据被封装成了byte流（其实也是二进制的）



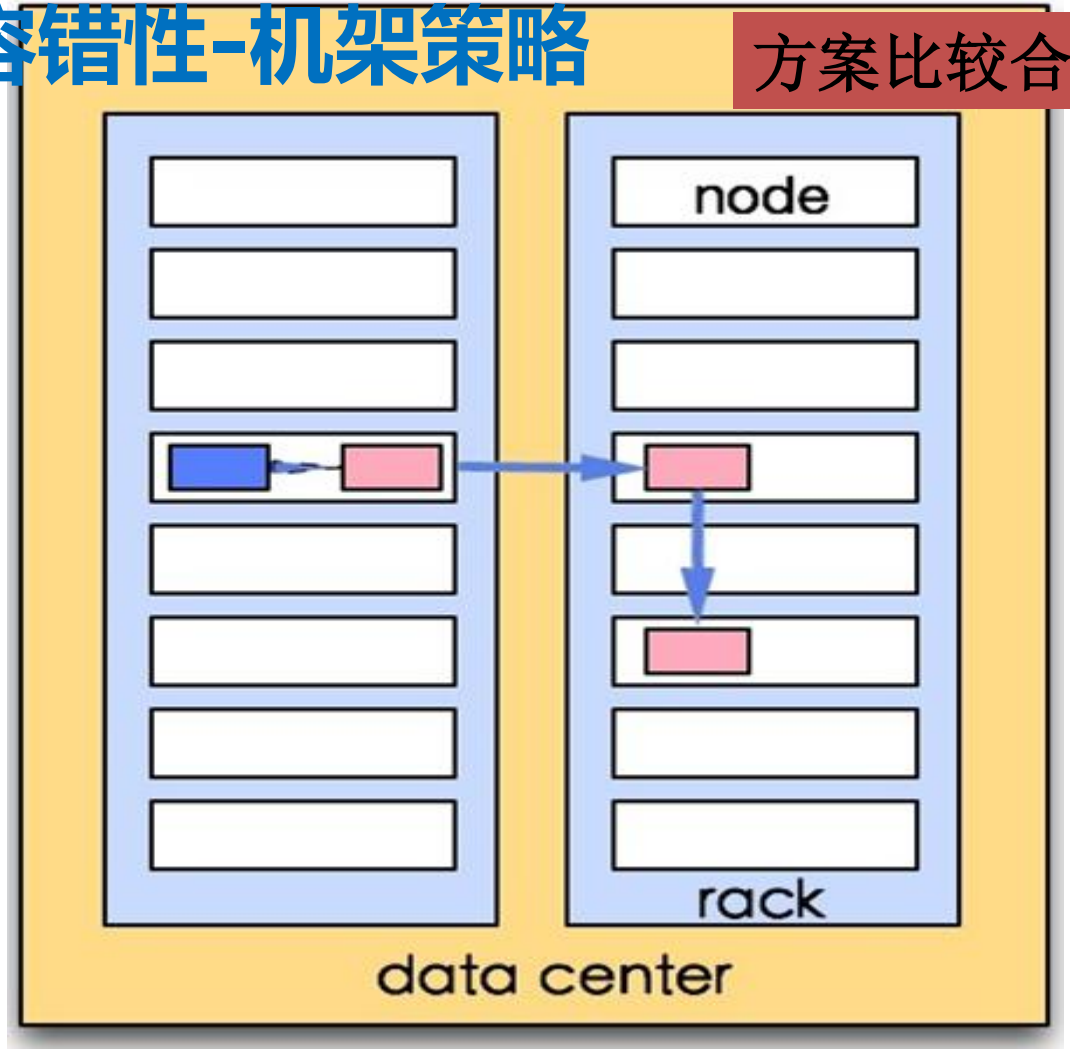
# 容错性-机架策略

- 大型HDFS集群系统运行在跨越多个机架的数据中心，不同机架上的两台机器之间的通信需要经过交换机。同一个机架内两台机器间带宽会比不同机架两台机器间带宽大。在大数量的情景中，带宽是稀缺资源，
- 计算两个节点间的间距，采用最近距离的节点进行操作，表示为tree，两个节点的距离计算就是寻找公共祖先的计算。比较典型的情景
- tree结构的节点由数据中心data center，表示为d1，d2，机架rack，表示为r1，r2，r3，还有服务器节点node，表示为n1，n2，n3，n4
- distance(d1/r1/n1,d1/r1/n1)=0 （相同节点）
- distance(d1/r1/n1,d1/r1/n2)=2 （相同机架不同节点）
- distance(d1/r1/n1,d1/r2/n3)=4 （相同数据中心不同机架）
- distance(d1/r1/n1,d2/r3/n4)=6 （不同数据中心）



# 容错性-机架策略

**方案比较合理**




✓ 可靠性：

block存储在两个机架上

✓ 写带宽：

写操作仅仅穿过一个网络交换机

✓ 读操作：

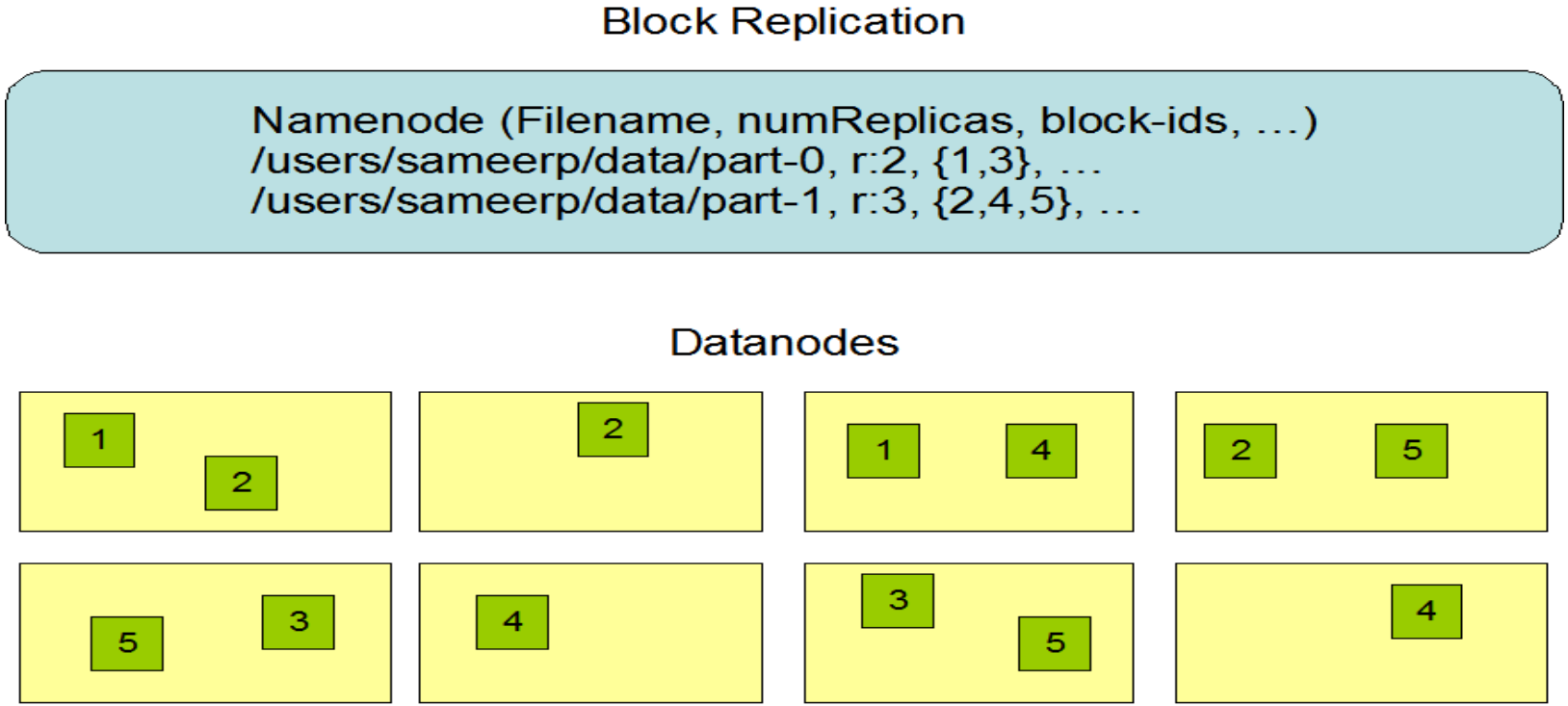
选择其中得一个机架去读

✓ block分布在整个集群上

# 容错性-冗余副本

- namenode节点选择一个datanode节点去存储block副本过程就叫做副本存放，这个过程的策略其实就是在可靠性和读写带宽间得权衡。
- 把所有的副本存放在同一个节点上，写带宽是保证了，但是这个可靠性是完全假的，一旦这个节点挂掉，数据就全没了，而且跨机架的读带宽也很低。所有副本打散在不同的节点上，可靠性提高了，但是带宽有成了问题。即使在同一个数据中心也有很多种副本存放方案。
- hadoop默认的方案：
- 把第一副本放在和客户端同一个节点上，如果客户端不在集群中，那么就会随即选一个节点存放。
- 第二个副本会再和第一个副本不同的机架上随机选一个
- 第三个副本会在第二个副本相同的机架上随机选一个不同的节点
- 剩余的副本就完全随机节点了。

# 容错性-冗余副本

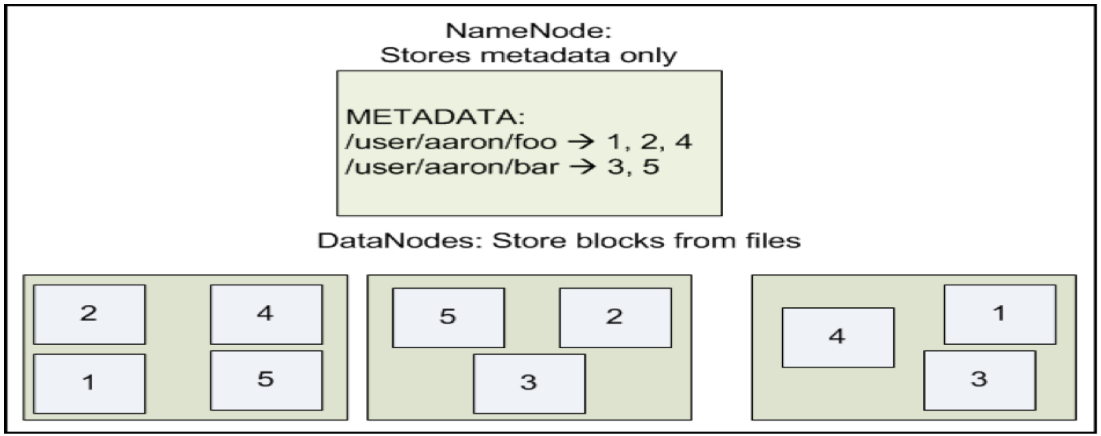

Block Replication
Namenode (Filename, numReplicas, block-ids, ...)
/users/sameerp/data/part-0, r:2, {1,3}, ...
/users/sameerp/data/part-1, r:3, {2,4,5}, ...
Datanodes
1
2
2
1
4
2
5
5
3
4
3
5
4

# 容错性-冗余副本


两个文件，一个文件156M，一个文件128M在HDFS里面怎么存储？

--Block为64MB

--rapliction默认拷贝3份



# 容错性-冗余副本

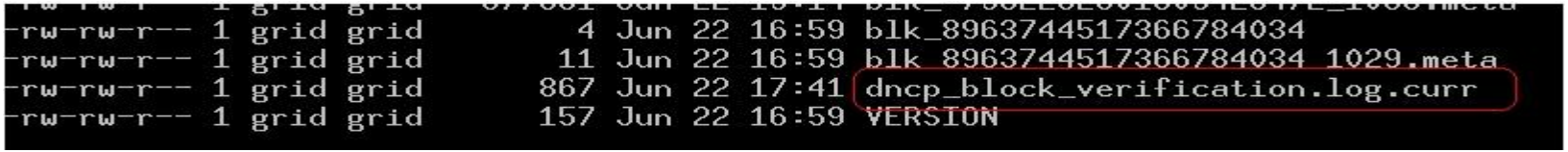
hdfs-site.xml中设置复制因子指定副本数量，fs.replication --复制因子，用来指定副本数量。数越大越安全（数大了，空间利用率会下降，影响速度）。根据集群的实际情况（性能和空间利用率的平衡）

Datanode启动时，遍历本地文件系统，产生一份hdfs数据块和本地文件的对应关系列表（blockreport）汇报给namenode。

blockreport块报告给namenode,namenode拿这个报告和元数据对照，看看数据节点的实际情况与元数据是否相同，采取相应安全措施。

# 校验和

文件创立时，每个数据块都产生校验和。
在数据节点的current目录下： dncp_block_verification.log.curr。（校验和日志），校验和在meta文件中。读数据块时，读完会把校验和算一下，与读出来的校验和看是否一致，一致的话就认为无问题，有问题就处理。将这个block标记错误，在块报告向namenode汇报等。校验和存在meta下 ，客户端获取数据时可以检查校验和是否相同，从而发现数据块是否损坏 如果正在读取的数据块损坏，则可以继续读取其它副本



# 校验和

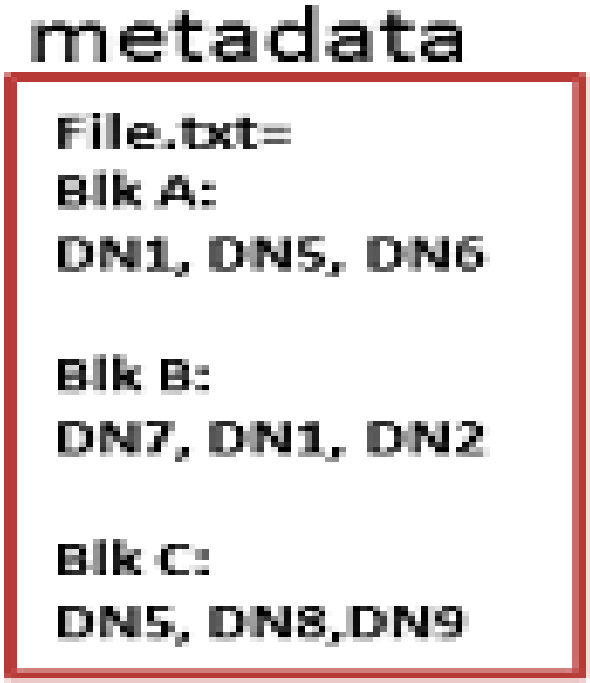
- 任何语言对IO的操作都要保持其数据的完整性。hadoop当然希望数据在存储和处理中不会丢失或损坏。检查数据完整性的常用方法是校验和checksum。
- 写入datanode验证
  HDFS会对写入的所有数据计算校验和，并在读取数据时验证校验和。datanode负责在验证收到的数据后存储数据及其校验和。正在写数据的客户端将数据及其校验和发送到一系列datanode组成的管线，管线的最后一个datanode负责验证校验和。
- 读取datanode验证
  客户端从datanode读取数据时验证校验和，将它们与datanode中的存储的校验和进行比较。Datanode保存验证的校验和日志，知道每个数据块的最后一次验证时间。客户端成功验证一个数据块后，会告诉这个datanode，datanode由此更新日志。
- Datanode块扫描器
- datanode后台进程DataBlockScanner，定期（三周）验证存储在这个datanode上的数据块

# 回收站

删除文件时，其实是放入回收站/trash
物理上没有直接删除
fs.trash.interval　　0　　默认关闭
回收站里的文件可以快速恢复
可以设置一个时间阈值，当回收站里文件的存放时间超过这个阈值，就被彻底删除，并且释放占用的数据块。

# 元数据保护

- 维护HDFS文件系统中文件和目录的信息，分为内存元数据和文件元数据两种。NameNode维护整个元数据。
- NameNode的元数据信息在启动后会加载到内存，实现对内存和I/O进行集中管理
- 元数据存储到磁盘文件名为”fsimage”（current/fsimage就是映像文件，保存元数据的目录(dfs.name.dir）。
- Block的位置信息不会保存到fsimage



# 元数据保护

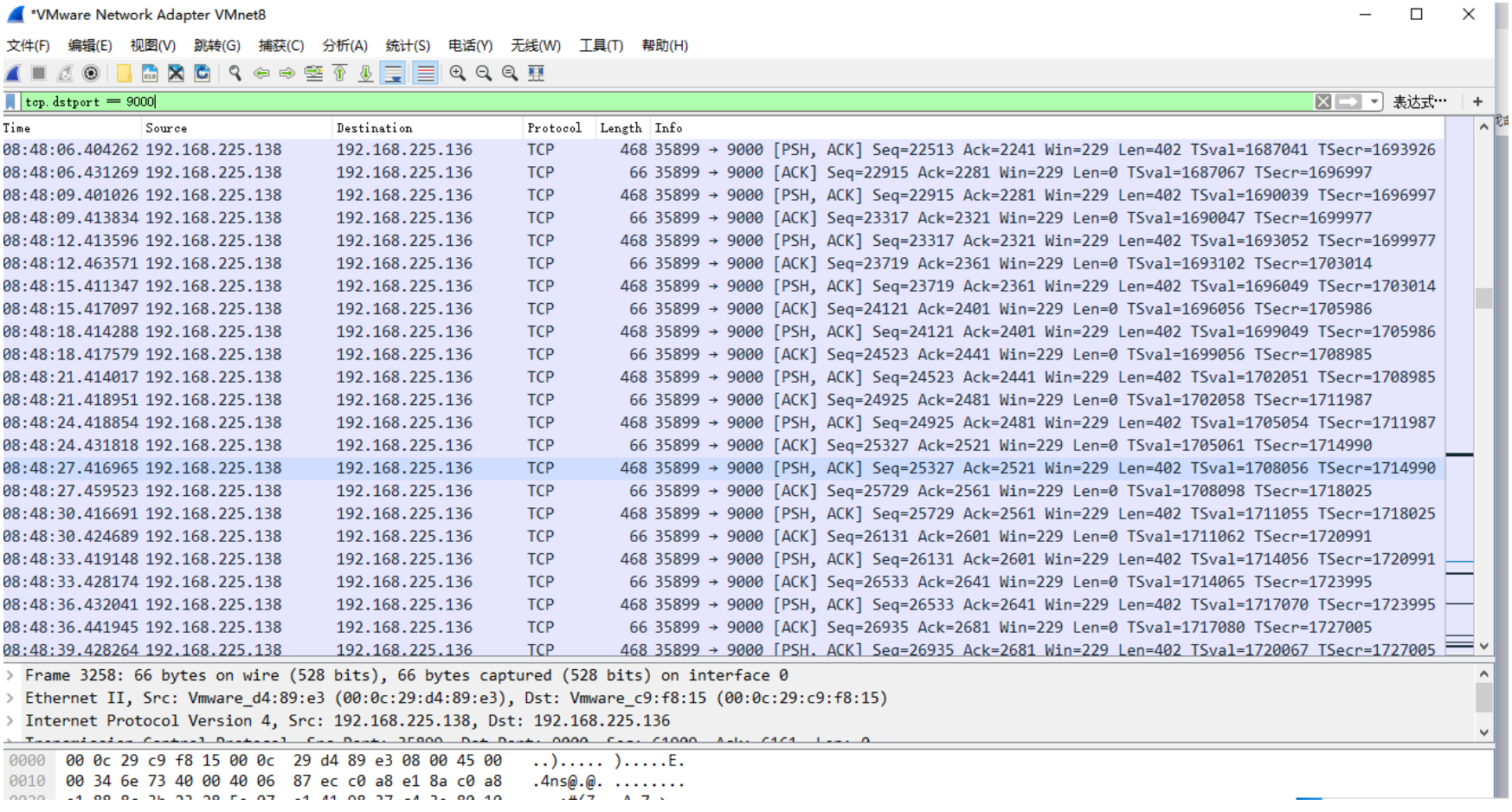


- Namenode单点，如果发生故障要手工切换。
- Namenode核心数据：映像文件和事务日志（元数据主要组成部分）。
- 可以配置为拥有多个副本。（在namenode中把这些文件复制多次，象oracle控制文件复制多次一样）

# 心跳机制

- 每个DN会去每一个NN注册，并且周期性的给所有NN发送心跳及DN的使用报告。NN周期性从DN接收心跳信号和块报告。NN根据块报告验证元数据（不停地发blockreport，根据心跳信号和块报告验证元数据有无失效，实际情况如果和NN元数据不同，进行修正或冗余策略）
- 默认节点每隔三秒发送心跳信号（heartbeats）给主节点的方式确认连接状态。心跳信号使用TCP握手来实现，主从节点间的通信端口是一致的（9000）。每10组心跳信号会组成一个Block Report记录在服务器上。
- DN还会给NN发送其所在的block pool的block report（块报告）。由于有多个NN同时存在，任何一个NN都可以随时动态加入、删除和更新

# 心跳机制


*VMware Network Adapter VMnet8
文件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)
tcp.dstport == 9000
表达式…
Time Source Destination Protocol Length Info
08:48:06.404262 192.168.225.138 192.168.225.136 TCP 468 35899 → 9000 [PSH, ACK] Seq=22513 Ack=2241 Win=229 Len=402 TSval=1687041 TSecr=1693926
08:48:06.431269 192.168.225.138 192.168.225.136 TCP 66 35899 → 9000 [ACK] Seq=22915 Ack=2281 Win=229 Len=0 TSval=1687067 TSecr=1696997
08:48:09.401026 192.168.225.138 192.168.225.136 TCP 468 35899 → 9000 [PSH, ACK] Seq=22915 Ack=2281 Win=229 Len=402 TSval=1690039 TSecr=1696997
08:48:09.413834 192.168.225.138 192.168.225.136 TCP 66 35899 → 9000 [ACK] Seq=23317 Ack=2321 Win=229 Len=0 TSval=1690047 TSecr=1699977
08:48:12.413596 192.168.225.138 192.168.225.136 TCP 468 35899 → 9000 [PSH, ACK] Seq=23317 Ack=2321 Win=229 Len=402 TSval=1693052 TSecr=1699977
08:48:12.463571 192.168.225.138 192.168.225.136 TCP 66 35899 → 9000 [ACK] Seq=23719 Ack=2361 Win=229 Len=0 TSval=1693102 TSecr=1703014
08:48:15.411347 192.168.225.138 192.168.225.136 TCP 468 35899 → 9000 [PSH, ACK] Seq=23719 Ack=2361 Win=229 Len=402 TSval=1696049 TSecr=1703014
08:48:15.417097 192.168.225.138 192.168.225.136 TCP 66 35899 → 9000 [ACK] Seq=24121 Ack=2401 Win=229 Len=0 TSval=1696056 TSecr=1705986
08:48:18.414288 192.168.225.138 192.168.225.136 TCP 468 35899 → 9000 [PSH, ACK] Seq=24121 Ack=2401 Win=229 Len=402 TSval=1699049 TSecr=1705986
08:48:18.417579 192.168.225.138 192.168.225.136 TCP 66 35899 → 9000 [ACK] Seq=24523 Ack=2441 Win=229 Len=0 TSval=1699056 TSecr=1708985
08:48:21.414017 192.168.225.138 192.168.225.136 TCP 468 35899 → 9000 [PSH, ACK] Seq=24523 Ack=2441 Win=229 Len=402 TSval=1702051 TSecr=1708985
08:48:21.418951 192.168.225.138 192.168.225.136 TCP 66 35899 → 9000 [ACK] Seq=24925 Ack=2481 Win=229 Len=0 TSval=1702058 TSecr=1711987
08:48:24.418854 192.168.225.138 192.168.225.136 TCP 468 35899 → 9000 [PSH, ACK] Seq=24925 Ack=2481 Win=229 Len=402 TSval=1705054 TSecr=1711987
08:48:24.431818 192.168.225.138 192.168.225.136 TCP 66 35899 → 9000 [ACK] Seq=25327 Ack=2521 Win=229 Len=0 TSval=1705061 TSecr=1714990
08:48:27.416965 192.168.225.138 192.168.225.136 TCP 468 35899 → 9000 [PSH, ACK] Seq=25327 Ack=2521 Win=229 Len=402 TSval=1708056 TSecr=1714990
08:48:27.459523 192.168.225.138 192.168.225.136 TCP 66 35899 → 9000 [ACK] Seq=25729 Ack=2561 Win=229 Len=0 TSval=1708098 TSecr=1718025
08:48:30.416691 192.168.225.138 192.168.225.136 TCP 468 35899 → 9000 [PSH, ACK] Seq=25729 Ack=2561 Win=229 Len=402 TSval=1711055 TSecr=1718025
08:48:30.424689 192.168.225.138 192.168.225.136 TCP 66 35899 → 9000 [ACK] Seq=26131 Ack=2601 Win=229 Len=0 TSval=1711062 TSecr=1720991
08:48:33.419148 192.168.225.138 192.168.225.136 TCP 468 35899 → 9000 [PSH, ACK] Seq=26131 Ack=2601 Win=229 Len=402 TSval=1714056 TSecr=1720991
08:48:33.428174 192.168.225.138 192.168.225.136 TCP 66 35899 → 9000 [ACK] Seq=26533 Ack=2641 Win=229 Len=0 TSval=1714065 TSecr=1723995
08:48:36.432041 192.168.225.138 192.168.225.136 TCP 468 35899 → 9000 [PSH, ACK] Seq=26533 Ack=2641 Win=229 Len=402 TSval=1717070 TSecr=1723995
08:48:36.441945 192.168.225.138 192.168.225.136 TCP 66 35899 → 9000 [ACK] Seq=26935 Ack=2681 Win=229 Len=0 TSval=1717080 TSecr=1727005
08:48:39.428264 192.168.225.138 192.168.225.136 TCP 468 35899 → 9000 [PSH, ACK] Seq=26935 Ack=2681 Win=229 Len=402 TSval=1720067 TSecr=1727005
Frame 3258: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0
Ethernet II, Src: Vmware_d4:89:e3 (00:0c:29:d4:89:e3), Dst: Vmware_c9:f8:15 (00:0c:29:c9:f8:15)
Internet Protocol Version 4, Src: 192.168.225.138, Dst: 192.168.225.136
0000 00 0c 29 c9 f8 15 00 0c 29 d4 89 e3 08 00 45 00 ..)..... ).....E.
0010 00 34 6e 73 40 00 40 06 87 ec c0 a8 e1 8a c0 a8 .4ns@.@. ........

# 心跳机制

- 没有按时发送心跳的datanode被标记为宕机，不会再给它任何I/O请求
- 引发重新复制的原因：datanode失效，数据副本本身损坏，磁盘错误，复制因子被增大等
- 如果datanode失效造成副本数量下降，并且低于预先设置的阈值，namenode会检测这些数据块，并在合适的时机进行重新复制
- dfs.replication
- dfs.replication.min
- dfs.replication.interval
- dfs.safemode.threshold.pct

# 谢谢！