# 3-HDFS入门和应用开发



## 企业存储系统

随着这两年产业互联网的推动和发展，越来越多的企业开始进行数字化转型，将传统的业务流程进行数字化改造。在进行数字化的过程中，需要数据来支撑企业的业务流程重塑，并以客户、产品为核心，以数据来支撑精细化运营。而数据分散在不同的系统中，要更充分的利用数据，需要将企业的大量数据集中存储，并进行业务化处理。此时，我们要想办法来解决大规模数据存储的问题。不管是使用哪种存储技术，都需要有存储硬件的支持。

### 硬盘



硬盘是计算机的主要存储硬件，可以用来存储大量数据。目前（2020年），市面上比较流行的硬盘多数是TB级的。

#### SATA硬盘

SATA即Serial ATA（串行ATA），是由Intel、IBM、Maxtor和Seagate等公司提出的硬盘接口规范。采用的是串行连接方式，很多时候会把SATA接口的硬盘称之为串口硬盘。



上图为希捷2TB高速机械硬盘，SATA接口，拥有256M的磁盘缓存，转速7200rpm，每秒可达90-190MB的速度。

#### SATA SSD固态硬盘



采用固态电子存储芯片阵列制作的硬盘，是以闪存作为永久性存储器的存储设备。固态硬盘的读取速度可以在500Mb/s左右。

#### RAID磁盘阵列

单个硬盘的存储能力是有限的，如果要存储更多的数据，可以通过某种技术，将若干个硬盘连接在一起，提供能耗的存储能力。我们在服务器上插更多的磁盘来提高存储容量，而服务器上的插槽是有限的，我们无法无限地增加硬盘。所以，我们可以买RAID磁盘阵列来解决数据存储速度、容错问题。



RAID可以将多块独立的硬盘组织在一起，可以将多块硬盘连接在一起，并在性能上、容错上会有一定地提升。

### 文件系统

#### 介绍

* 计算机的文件系统是一种**存储**和**组织计算机数据**的方法，它使得对其访问和查找变得容易
* 文件系统使用**文件和树形目录**的抽象逻辑概念代替了硬盘和光盘等物理设备使用数据块的概念，用户使用文件系统来保存数据不必关心数据实际保存在硬盘（或者光盘）的地址为多少的数据块上，只需要记住这个文件的所属目录和文件名。
* 在写入新数据之前，用户不必关心硬盘上的那个块地址没有被使用，硬盘上的**存储空间管理**（分配和释放）功能由文件系统自动完成，用户只需要记住数据被写入到了哪个文件中。
* 文件系统通常使用硬盘和光盘这样的存储设备，并维护文件在设备中的物理位置。但是，实际上文件系统也可能仅仅是一种访问资料的界面而已，实际的数据是通过网络协议（如NFS、SMB、9P等）提供的或者内存上，甚至可能根本没有对应的文件（如proc文件系统）。
* 严格地说，文件系统是一套实现了数据的存储、分级组织、访问和获取等操作的抽象数据类型（Abstract data type）。

#### 重要概念

* 文件系统是一种用于向用户提供底层数据访问的机制。它将设备中的空间划分为特定大小的块（或者称为簇），一般**每块512字节**。数据存储在这些块中，大小被修正为占用整数个块。由文件系统软件来负责将这些块组织为文件和目录，并记录哪些块被分配给了哪个文件，以及哪些块没有被使用。
* 不过，文件系统并不一定只在特定存储设备上出现。它是数据的组织者和提供者，至于它的底层，可以是磁盘，也可以是其它动态生成数据的设备（比如网络设备）。

#### 文件名

* 在文件系统中，文件名是用于定位存储位置。
* 大多数的文件系统对文件名的长度有限制。在一些文件系统中，文件名是大小写不敏感（如“AAA”和“aaa”指的是同一个文件）；在另一些文件系统中则大小写敏感。
* 大多现今的文件系统允许文件名包含非常多的Unicode字符集的字符。然而在大多数文件系统的界面中，会限制某些特殊字符出现在文件名中。（文件系统可能会用这些特殊字符来表示一个设备、设备类型、目录前缀、或文件类型），为方便起见，一般不建议在文件名中包含特殊字符。

#### 元数据

* 其它文件保存信息常常伴随着文件自身保存在文件系统中。
* 文件长度可能是分配给这个文件的区块数，也可能是这个文件实际的字节数。文件最后修改时间也许记录在文件的时间戳中。有的文件系统还保存文件的创建时间，最后访问时间及属性修改时间。（不过大多数早期的文件系统不记录文件的时间信息）其它信息还包括文件设备类型（如：区块数，字符集，套接口，子目录等等），文件所有者的ID，组ID，还有访问权限（如：只读，可执行等等）。

|  |
| --- |
| [root@node2 ~]# ll  total 32  -rw-------. 1 root root 2117 Apr 8 2020 anaconda-ks.cfg  drwxr-xr-x. 2 root root 6 Aug 30 09:33 Desktop  drwxr-xr-x. 2 root root 6 Aug 30 09:33 Documents  drwxr-xr-x. 2 root root 6 Aug 30 09:33 Downloads  -rw-r--r--. 1 root root 2165 Apr 8 2020 initial-setup-ks.cfg  drwxr-xr-x. 2 root root 6 Aug 30 09:33 Music  drwxr-xr-x. 2 root root 6 Aug 30 09:33 Pictures  -rw-r--r--. 1 root root 3158 Oct 10 22:53 profile  drwxr-xr-x. 2 root root 6 Aug 30 09:33 Public  drwxr-xr-x. 2 root root 6 Aug 30 09:33 Templates  drwxr-xr-x. 2 root root 6 Aug 30 09:33 Videos  -rw-r--r--. 1 root root 18234 Oct 10 18:04 zookeeper.out |

#### 文件系统分类

##### 磁盘文件系统

磁盘文件系统是一种设计用来利用数据存储设备来保存计算机文件的文件系统，最常用的数据存储设备是磁盘驱动器，可以直接或者间接地连接到计算机上。例如：FAT、exFAT、NTFS、HFS、HFS+、ext2、ext3、ext4、ODS-5、btrfs、XFS、UFS、ZFS。



**Windows支持的文件系统**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Windows** | **FAT12/FAT16** | **FAT32/VFAT** | **FAT64/exFAT** | **NTFS** |
| Windows 3.x或更早 （MS-DOS 6.22） （PC-DOS 7.0） | 可读/可写 | 不支持 | 不支持 | 不支持 |
| Windows 95 | 可读/可写 | 不支持 | 不支持 | 不支持 |
| Windows 95（OSR2以后） Windows 98（含SE） ME | 可读/可写 | 可读/可写 | 不支持 | 不支持 |
| Windows NT | 可读/可写 | 不支持 | 不支持 | 可读/可写 |
| 2000 Windows XP Windows Vista Server 2003 Server 2008 (R2) Windows 7 Windows 8 Server 2012 (R2) Windows 8.1  Windows 10 | 可读/可写 | 可读/可写 | 可读/可写 | 可读/可 |



**Linux支持的文件系统**

随着Linux的不断发展，它所支持的文件系统也在迅速扩充，Linux系统核心可以支持十多种文件系统类型：**Btrfs、JFS、ReiserFS、exFAT、ext、ext2、ext3、ext4、XFS、ISO 9660、Minix、MSDOS、UMSDOS、VFAT**、NTFS（Linux Kernel内置的NTFS驱动程序，写入功能不稳定）、**HPFS、NFS、SMB、SysV、PROC**等。

注意：部分Linux发行版的Kernel默认不编译Kernel内置的NTFS文件系统支持，常见的在Linux下读写NTFS的解决方法是安装NTFS-3G或ufsd等NTFS驱动程序。部分Linux发行版对NTFS的支持度并不高。



**UNIX及BSD操作系统下的文件系统**

柏克莱加州大学开发早期的伯克利快速文件系统（Berkeley Fast File System），再由各UNIX厂商开发不同的文件系统，包括IRIX上的XFS、IBM AIX的JFS、HP HP-UNIX的VxFS及Solaris的ZFS。



**macOS（Mac OS X）的文件系统**

从1998年到2016年间使用HFS+，再早采用HFS。从2016年发布的macOS Sierra起，使用苹果文件系统（APFS）。

##### 光盘

ISO 9660和UDF被用于CD、DVD与蓝光光盘。

##### 网络文件系统

网络文件系统（NFS，Network File System）是一种将远程主机上的分区（目录）经网络挂载到本地系统的一种机制。

### 面临海量数据存储的问题

#### 成本高

传统存储硬件通用性差，设备投资加上后期维护、升级扩容的成本非常高。



#### 性能低

单节点I/O性能瓶颈无法逾越，容量和性能都不易扩展，难以支撑海量数据的高并发高吞吐场景。

#### 可扩展性差

无法实现快速部署和弹性扩展。

#### 支持大数据分析、AI

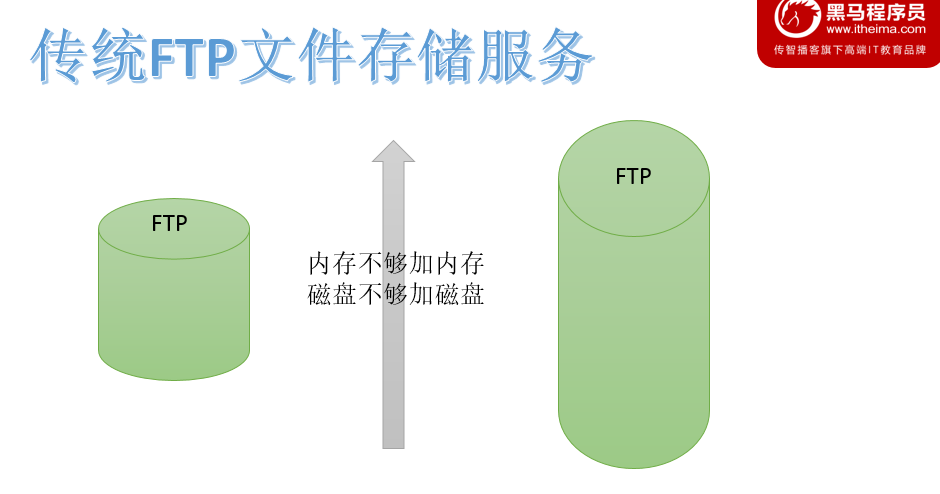
传统存储与Spark等大数据分析平台对接是否有难度，一套存储能否满足企业数据存储、管理和挖掘的需求。

## 场景案例：如何模拟实现分布式存储？

### 如何解决海量数据存的下问题

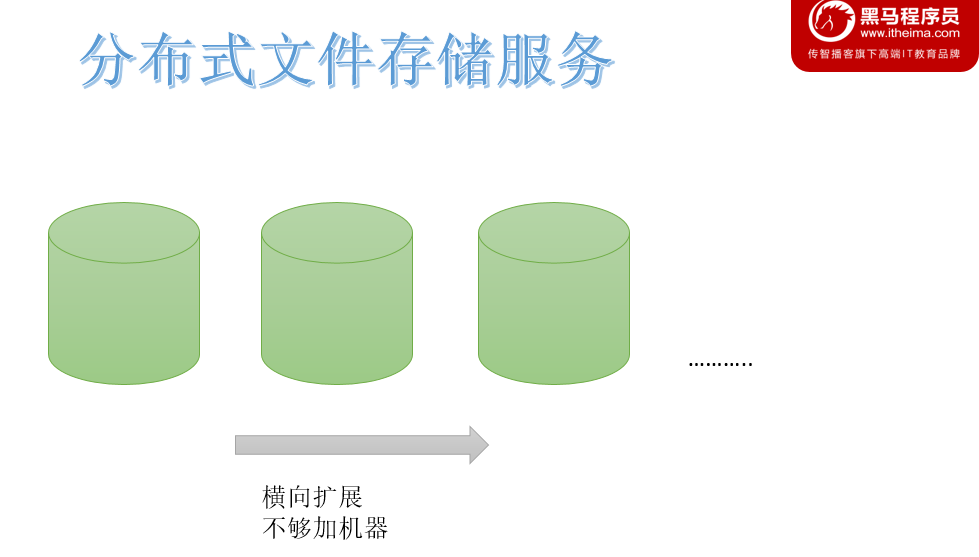
#### 传统存储方式

应对文件存储服务，传统做法是在服务器上部署文件服务比如FTP。但是随着数据变多，会遇到存储瓶颈。此时，本能的操作反应是：内存不够加内存，磁盘不够加磁盘—单机纵向扩展。但是单机能够扩展的内存磁盘是有上限的，不能无限制下去。



#### 分布式存储方式

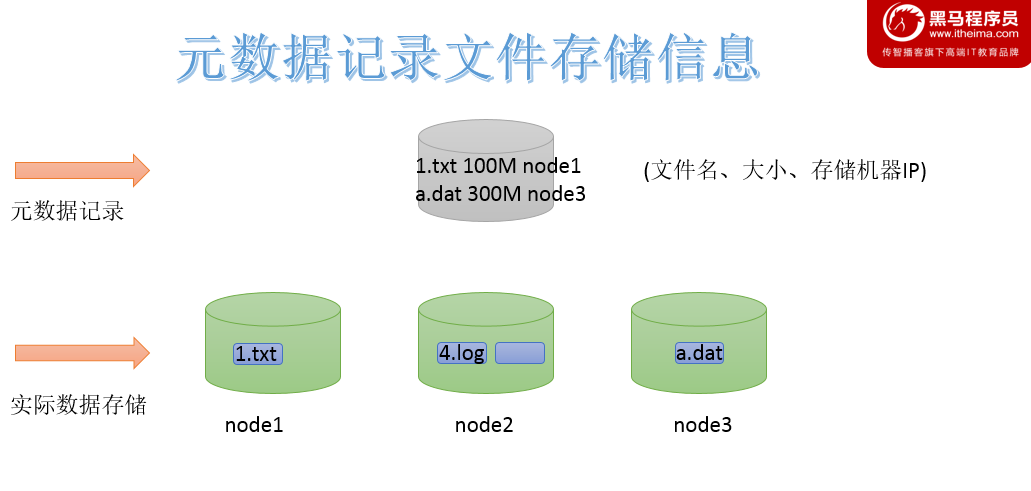
纵向扩展有上限，自然想到横向扩展。所谓横向指的是采用多台机器存储，一台不够就多台一起存储，不够就加机器。



理论上，可以横向无限制下去。因此海量数据如何存储的下的问题解决方式就是采用多台机器存储—即**分布式存储**。

### 如何解决数据查询便捷问题

当文件被分布式存储在多台机器之后，后续获取文件的时候如何能快速找到文件位于哪台机器上呢。一台一台查询过来也不靠谱。因此可以借助于**元数据记录**来解决这个问题。把文件和其存储的机器的位置信息记录下来，类似于图书馆查阅图书系统，这样就可以快速定位文件存储在哪一台机器上了。



### 如何解决大文件传输效率慢问题

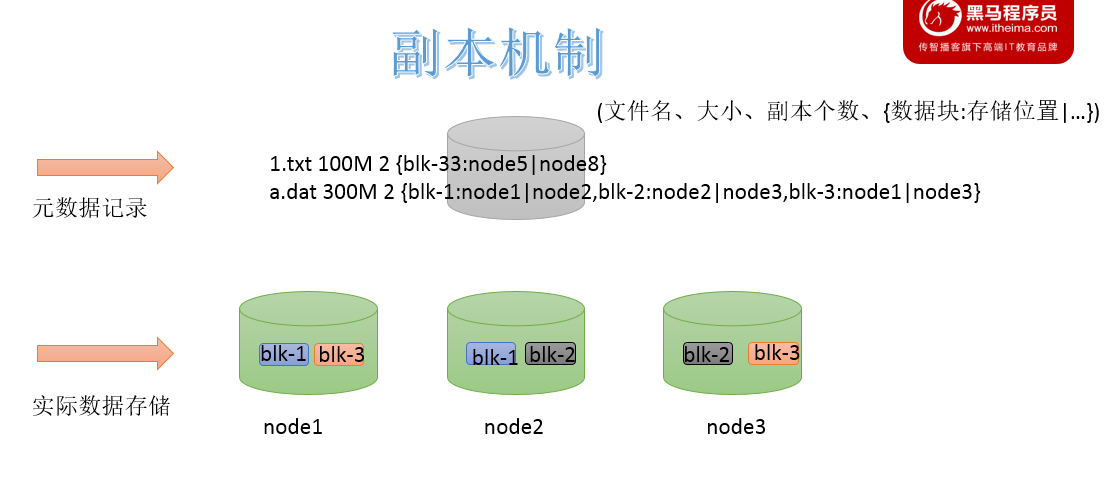
大数据使用场景下，GB、TP级别的大文件是常见的。当单个文件过大的时候，如何提高传输效率？通常的做法是**分块存储**：把大文件拆分成若干个小块（**block** 简写blk），分别存储在不同机器上，并行操作提高效率。

此外分块存储还可以解决数据存储负载均衡问题。此时元数据记录信息也应该更加详细：文件分了几块，分别位于哪些机器上。



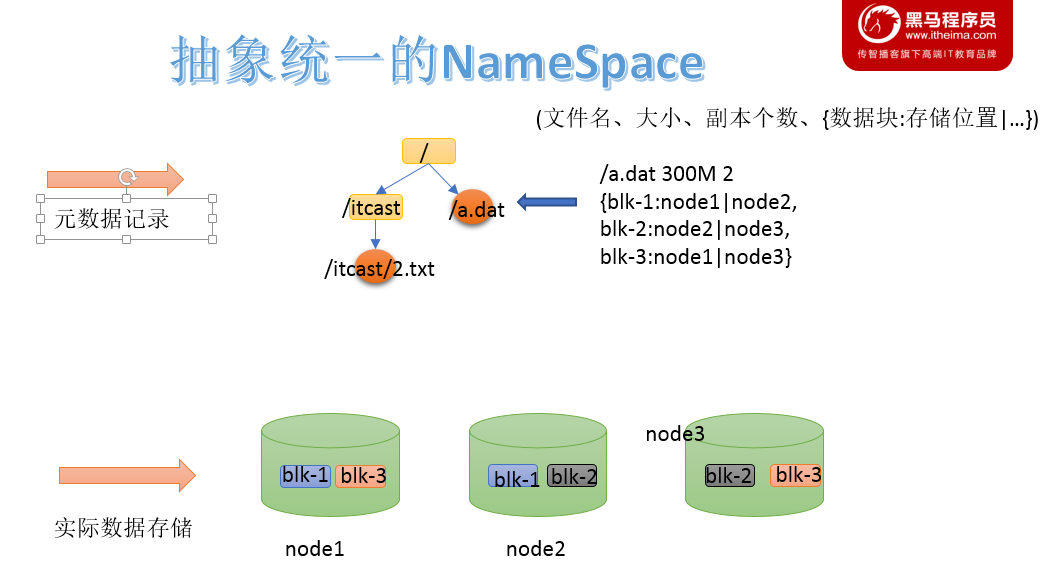
### 如何解决数据丢失问题

机器、磁盘等硬件出现故障是难以避免的事情，如何保证数据存储的安全性。如果某台机器故障，数据块丢失，对于文件来说整体就是不完整的。冗余存储是个不错的选择。采用**副本机制**。副本越多，数据越安全，当然冗余也会越多。通过“不要把鸡蛋放在一个篮子里”的思想，可以把数据丢失的风险分散到各个机器上。



### 如何解决用户查询视角统一问题

随着存储的进行，数据文件越来越多，与之对应元数据信息也越来越多，如何让用户视觉层面感觉不到元数据的凌乱，同时也与传统的文件系统操作体验保持一致？传统的文件系统拥有所谓的目录树结构，带有层次感的namespace（命名空间），因此可以把分布式文件系统的元数据记录这一块也**抽象成统一的目录树结构**。



### 小结

通过上述场景式分析，可以得出要想实现一个分布式文件系统，是需要多方面综合考虑的。通常来说一个分布式文件系统需要具备：**分布式特性**、**分块存储**、**副本机制**、**元数据记录**、**抽象目录树**、**统一namespace命名空间**。

## 分布式文件HDFS

### HDFS简介

* HDFS（Hadoop Distributed File System）是 Apache Hadoop 项目的一个子项目，它的设计初衷是为了能够支持高吞吐和超大文件读写操作
* HDFS是一种能够在普通硬件上运行的分布式文件系统，它是高度容错的，适应于具有大数据集的应用程序，它非常适于存储大型数据 (比如 TB 和 PB)
* HDFS使用多台计算机存储文件, 并且提供统一的访问接口, 像是访问一个普通文件系统一样使用分布式文件系统

### HDFS发展历史

1. Doug Cutting 在做 Lucene 的时候, 需要编写一个爬虫服务, 这个爬虫写的并不顺利, 遇到 了一些问题, 诸如: 如何存储大规模的数据, 如何保证集群的可伸缩性, 如何动态容错等
2. 200x年的时候, Google 发布了三篇论文, 被称作为三驾马车, 其中有一篇叫做 GFS
3. GFS是描述了 Google 内部的一个叫做 GFS 的分布式大规模文件系统, 具有强大的可伸缩性和容错
4. Doug Cutting后来根据 GFS 的论文, 创造了一个新的文件系统, 叫做 HDFS

### HDFS设计目标

1. HDFS集群由很多的服务器组成，而每一个机器都与可能会出现故障。HDFS为了能够进行故障检测、快速恢复等。
2. HDFS主要适合去做批量数据出来，相对于数据请求时的反应时间，HDFS更倾向于保障吞吐量。
3. 典型的HDFS中的文件大小是GB到TB，HDFS比较适合存储大文件
4. HDFS很多时候是以： Write-One-Read-Many来应用的，一旦在HDFS创建一个文件，写入完后就不需要修改了

### HDFS应用场景

#### 适合的应用场景

* 存储非常大的文件：这里非常大指的是几百M、G、或者TB级别，需要高吞吐量，对延时没有要求。
* 基于流的数据访问方式: 即一次写入、多次读取，数据集经常从数据源生成或者拷贝一次，然后在其上做很多分析工作 ，且不支持文件的随机修改。
* 正因为如此，HDFS适合用来做大数据分析的底层存储服务，并不适合用来做网盘等应用，因为，修改不方便，延迟大，网络开销大，成本太高。
* 运行于商业硬件上: Hadoop不需要特别贵的机器，可运行于普通廉价机器，可以处节约成本
* 需要高容错性
* 为数据存储提供所需的扩展能力

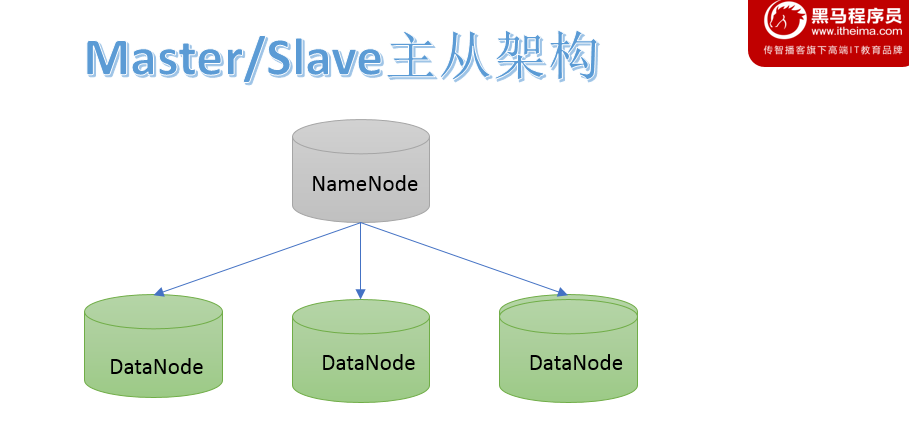
#### 不适合的应用场景

* **低延时**的数据访问 对延时要求在毫秒级别的应用，不适合采用HDFS。HDFS是为高吞吐数据传输设计的,因此可能牺牲延时
* **大量小文件**的元数据保存在NameNode的内存中， 整个文件系统的文件数量会受限于NameNode的内存大小。 经验而言，一个文件/目录/文件块一般占有150字节的元数据内存空间。如果有100万个文件，每个文件占用1个文件块，则需要大约300M的内存。因此十亿级别的文件数量在现有商用机器上难以支持
* **多方读写**，需要任意的文件修改 HDFS采用追加（append-only）的方式写入数据。不支持文件任意offset的修改，HDFS适合用来做大数据分析的底层存储服务，并不适合用来做.网盘等应用，因为，修改不方便，延迟大，网络开销大，成本太高。

### HDFS重要特性

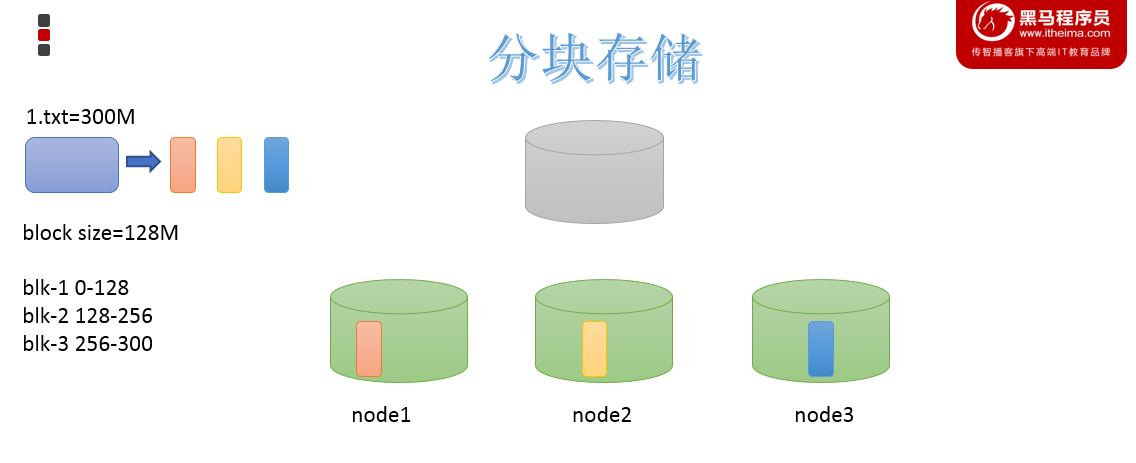
#### 主从架构

HDFS采用master/slave架构。一般一个HDFS集群是有一个Namenode和一定数目的Datanode组成。Namenode是HDFS主节点，Datanode是HDFS从节点，两种角色各司其职，共同协调完成分布式的文件存储服务。



#### 分块机制

HDFS中的文件在物理上是分块存储（block）的，块的大小可以通过配置参数来规定，参数位于hdfs-site.xml中：dfs.blocksize。默认大小是**128M**（134217728）。



#### 副本机制

为了容错，文件的所有block都会有副本。每个文件的block大小（dfs.blocksize）和副本系数（dfs.replication）都是可配置的。应用程序可以指定某个文件的副本数目。副本系数可以在文件创建的时候指定，也可以在之后通过命令改变。

默认**dfs.replication的值是3**，也就是会额外再复制2份，连同本身总共3份副本。



#### Namespace

HDFS支持传统的**层次型文件组织结构**。用户可以创建目录，然后将文件保存在这些目录里。文件系统名字空间的层次结构和大多数现有的文件系统类似：用户可以创建、删除、移动或重命名文件。

Namenode负责维护文件系统的namespace名称空间，任何对文件系统名称空间或属性的修改都将被Namenode记录下来。

HDFS会给客户端提供一个**统一的抽象目录树**，客户端通过路径来访问文件，形如：hdfs://namenode:port/dir-a/dir-b/dir-c/file.data。

#### 元数据管理

在HDFS中，Namenode管理的元数据具有两种类型：

* **文件自身属性信息**

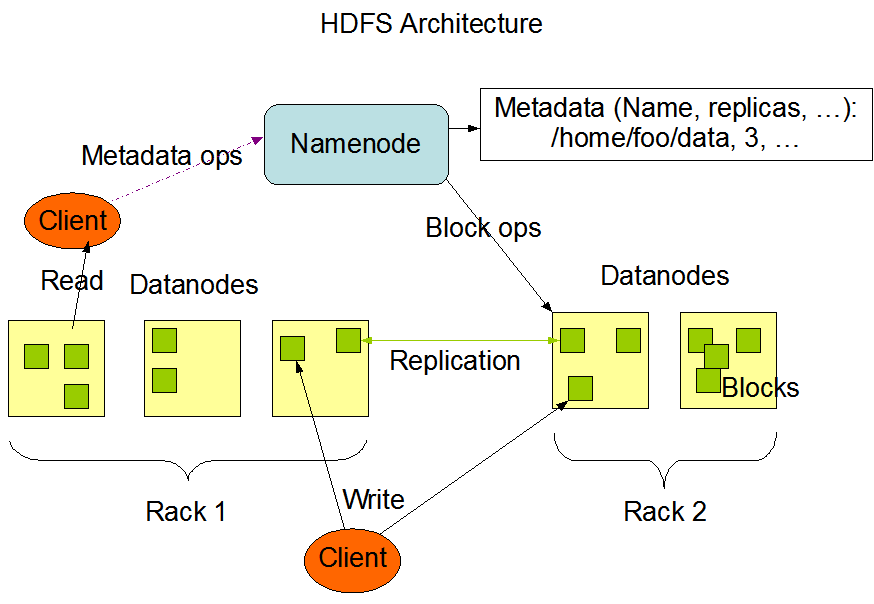
文件名称、权限，修改时间，文件大小，复制因子，数据块大小。

* **文件块位置映射信息**

记录文件块和DataNode之间的映射信息，即哪个块位于哪个节点上。

#### 数据块存储

文件的各个block的具体存储管理由DataNode节点承担。每一个block都可以在多个DataNode上存储。



## HDFS的Shell命令

HDFS是存取数据的分布式文件系统，那么对HDFS的操作，就是文件系统的基本操作，比如文件的创建、修改、删除、修改权限等，文件夹的创建、删除、重命名等。对HDFS的操作命令类似于Linux的shell对文件的操作，如ls、mkdir、rm等。

Hadoop提供了文件系统的shell命令行客户端，使用方法如下：

|  |
| --- |
| hadoop fs <args> |

文件系统shell包括与Hadoop分布式文件系统（HDFS）以及Hadoop支持的其他文件系统（如本地FS，HFTP FS，S3 FS等）直接交互的各种类似shell的命令。

所有FS shell命令都将路径URI作为参数。URI格式为scheme://authority/path。对于HDFS，该scheme是hdfs，对于本地FS，该scheme是file。scheme和authority是可选的。如果未指定，则使用配置中指定的默认方案。

对于HDFS,命令示例如下：

|  |
| --- |
| hadoop fs -ls  hdfs://namenode:port/parent/child  hadoop fs -ls  /parent/child #core-site.xml中的fs.defaultFS中有配置 |

对于本地文件系统，命令示例如下：

|  |
| --- |
| hadoop fs -ls file:///root/ |

如果使用的文件系统是HDFS，则也可使用hdfs dfs <args>命令。

### Shell命令选项

|  |  |  |
| --- | --- | --- |
| **选项名称** | **使用格式** | **含义** |
| -ls | -ls <路径> | 查看指定路径的当前目录结构 |
| -lsr | -lsr <路径> | 递归查看指定路径的目录结构 |
| -du | -du <路径> | 统计目录下个文件大小 |
| -dus | -dus <路径> | 汇总统计目录下文件(夹)大小 |
| -count | -count [-q] <路径> | 统计文件(夹)数量 |
| -mv | -mv <源路径> <目的路径> | 移动 |
| -cp | -cp <源路径> <目的路径> | 复制 |
| -rm | -rm [-skipTrash] <路径> | 删除文件/空白文件夹 |
| -rmr | -rmr [-skipTrash] <路径> | 递归删除 |
| -put | -put <多个linux上的文件> <hdfs路径> | 上传文件 |
| -copyFromLocal | -copyFromLocal <多个linux上的文件> <hdfs路径> | 从本地复制 |
| -moveFromLocal | -moveFromLocal <多个linux上的文件> <hdfs路径> | 从本地移动 |
| -getmerge | -getmerge <源路径> <linux路径> | 合并到本地 |
| -cat | -cat <hdfs路径> | 查看文件内容 |
| -text | -text <hdfs路径> | 查看文件内容 |
| -copyToLocal | -copyToLocal [-ignoreCrc] [-crc] [hdfs源路径] [linux目的路径] | 从本地复制 |
| -moveToLocal | -moveToLocal [-crc] <hdfs源路径> <linux目的路径> | 从本地移动 |
| -mkdir | -mkdir <hdfs路径> | 创建空白文件夹 |
| -touchz | -touchz <文件路径> | 创建空白文件 |
| -stat | -stat [format] <路径> | 显示文件统计信息 |
| -tail | -tail [-f] <文件> | 查看文件尾部信息 |
| -chmod | -chmod [-R] <权限模式> [路径] | 修改权限 |
| -chown | -chown [-R] [属主][:[属组]] 路径 | 修改属主 |
| -chgrp | -chgrp [-R] 属组名称 路径 | 修改属组 |
| -help | -help [命令选项] | 帮助 |

### 常用的Shell命令

**-ls**

|  |
| --- |
| 格式：  hadoop fs **-**ls URI  作用：类似于Linux的ls命令，显示文件列表  hadoop fs -ls / |

**-lsr**

|  |
| --- |
| 格式 **:**   hdfs dfs **-**lsr URI  作用 **:** 在整个目录下递归执行ls**,** 与UNIX中的ls-R类似  hadoop fs   -lsr / |

**-mkdir**

|  |
| --- |
| 格式 ： hdfs dfs [-p] -mkdir <paths>  作用 :   以<paths>中的URI作为参数，创建目录。使用-p参数可以递归创建目录  hadoop fs -mkdir /dir1  hadoop fs -mkdir /dir2  hadoop fs -p -mkdir /aaa/bbb/ccc |

**-put**

|  |
| --- |
| 格式   ： hadoop fs -put <localsrc > ... <dst>  作用 ： 将单个的源文件src或者多个源文件srcs从本地文件系统拷贝到目标文件系统中（<dst>对应的路径）。也可以从标准输入中读取输入，写入目标文件系统中  echo “Hello HDFS” >> /root/1.txt  hadoop fs -put /root/1.txt /dir1 |

**-moveFromLocal**

|  |
| --- |
| 格式： hdfs dfs **-**moveFromLocal **<**localsrc**>**   **<**dst**>**  作用**:**   和put命令类似，但是源文件localsrc拷贝之后自身被删除  echo “Hello HDFS” >> /root/2.txt  hdfs dfs -moveFromLocal /root/2.txt / |

**-moveToLocal**

未实现

**-get**

|  |
| --- |
| 格式   hadoop fs  -get [-ignorecrc ] [-crc] <src> <localdst>  ​  作用：将文件拷贝到本地文件系统。 CRC 校验失败的文件通过-ignorecrc选项拷贝。 文件和CRC校验和可以通过-CRC选项拷贝  hadoop fs  -get   /2.txt /export/data |

**-getmerge**

|  |
| --- |
| 格式: hadoop fs -getmerge -nl < hdfs dir > < local file >  功能：合并下载多个文件  参数: 加上nl后，合并到local file中的hdfs文件之间会空出一行  示例：比如hdfs的目录 /aaa/下有多个文件:log.1, log.2,log.3,...  hadoop fs -getmerge /aaa/log.\* ./log.sum |

**-mv**

|  |
| --- |
| 格式 ： hdfs dfs -mv URI   <dest>  作用： 将hdfs上的文件从原路径移动到目标路径（移动之后文件删除），该命令不能夸文件系统    hdfs dfs  -mv /dir1/a.txt   /dir2 |

**-rm**

|  |
| --- |
| 格式： hadoop fs -rm [-r] 【-skipTrash】 URI 【URI 。。。】  作用：   删除参数指定的文件和目录，参数可以有多个，删除目录需要加-r参数  如果指定-skipTrash选项，那么在回收站可用的情况下，该选项将跳过回收站而直接删除文件；  否则，在回收站可用时，在HDFS Shell 中执行此命令，会将文件暂时放到回收站中。    hadoop fs -rm /2.txt #删除文件  hadoop fs -rm  -r /dir1 #删除目录 |

**-cp**

|  |
| --- |
| 格式:     hdfs  dfs  -cp URI [URI ...] <dest>  作用：    将文件拷贝到目标路径中。如果<dest>  为目录的话，可以将多个文件拷贝到该目录下。  -f 选项将覆盖目标，如果它已经存在。  -p 选项将保留文件属性（时间戳、所有权、许可、ACL、XAttr）。    hadoop fs -cp /dir1/1.txt /dir2/2.txt |

**-cat**

|  |
| --- |
| hadoop fs  -cat URI [uri ...]  作用：将参数所指示的文件内容输出到控制台    hadoop fs  -cat /dir2/2.txt |

**-du**

|  |
| --- |
| hadoop fs  -cat URI  功能：显示目录中所有文件大小，当只指定一个文件时，显示此文件的大小。  hadoop fs -du / |

**-chmod**

|  |
| --- |
| 格式:      hadoop fs  -chmod  [-R]  URI[URI  ...] 作用：    改变文件权限。如果使用  -R 选项，则对整个目录有效递归执行。使用这一命令的用户必须是文件的所属用户，或者超级用户。  例如:可以创建一个用户hadoop，将/a.txt的所属用户和所属用户组修改为hadoop hadoop fs -chmod -R 777 /dir1 |

**-chown**

|  |
| --- |
| 格式:      hdfs   dfs  -chmod  [-R]  URI[URI  ...]  作用：    改变文件的所属用户和用户组。如果使用  -R 选项，则对整个目录有效递归执行。使用这一命令的用户必须是文件的所属用户，或者超级用户。  hadoop fs  -chown  -R hadoop:hadoop /a.txt |

**-appendToFile**

|  |
| --- |
| 格式: hadoop fs -appendToFile <localsrc> ... <dst>  作用: 追加一个或者多个文件到hdfs指定文件中.也可以从命令行读取输入.  cd /export/server/hadoop2.7.5/etc/hadoop/  hadoop fs -appendToFile \*.xml /big.xml |

**-setrep 命令**

HDFS中默认每个block会保存三个副本，同样一份数据需要存3份。假设，此处我们需要将 /source/weibo/start/comment\_log/20190811\_node1 因为已经过去了很久，我们对该目录下的文件容错要求较低、而且数据使用频率也较低，所以，我们可以将它的副本数调整为2，此时我们需要使用setrep命令。

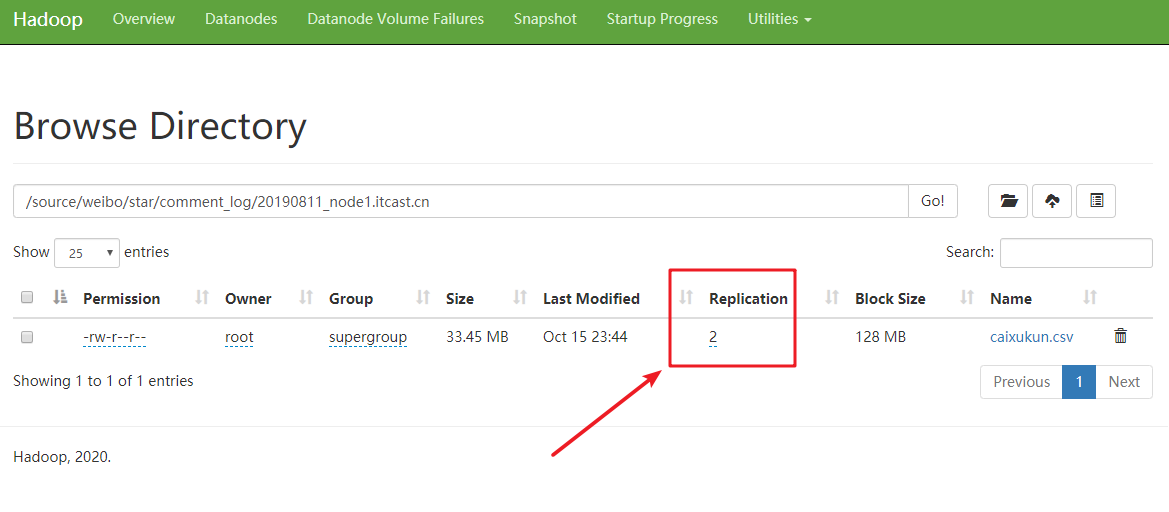
hdfs dfs -setrep [-R] [-w] <numReplicas> <path>

更改文件的副本因子。 如果path是目录，则该命令以递归方式更改以path为根的目录树下所有文件的复制因子。

参数：

|  |
| --- |
| -w：标志请求命令等待复制完成。 这可能会花费很长时间。  -R：标志是为了向后兼容。 没有作用。 |

|  |
| --- |
| [root@node1 ~]# **hdfs dfs -setrep -w 2 /dir/a.txt**  Replication 2 set: /source/weibo/star/comment\_log/20190811\_node1/caixukun.csv  Waiting for /source/weibo/star/comment\_log/20190811\_node1/caixukun.csv ...  WARNING: the waiting time may be long for DECREASING the number of replications.  . done |



## HDFS的安全模式

安全模式是hadoop的一种**保护机制**，用于保证集群中的数据块的安全性。当集群启动的时候，会首先进入安全模式。当系统处于安全模式时会检查数据块的完整性。

在NameNode启动过程中，等待DataNodes汇报可用的block信息。在此期间，NameNode保持在安全模式。随着DataNode的block汇报持续进行，当整个系统达到安全标准时，HDFS自动离开安全模式。在NameNode Web主页上会显示安全模式是打开还是关闭。

如果HDFS处于安全模式下，不允许HDFS客户端进行任何修改文件的操作,包括上传文件，删除文件，重命名，创建文件夹,修改副本数等操作。

假设我们设置的副本数（即参数dfs.replication）是3，那么在datanode上就应该有3个副本存在，假设只存在2个副本，那么比例就是2/3=0.666。hdfs默认的副本率0.999。我们的副本率0.666明显小于0.999，因此系统会自动的复制副本到其他dataNode，使得副本率不小于0.999。如果系统中有5个副本，超过我们设定的3个副本，那么系统也会删除多于的2个副本。

**在安全模式状态下，文件系统只接受读数据请求，而不接受删除、修改等变更请求**。在当整个系统达到安全标准时，HDFS自动离开安全模式。

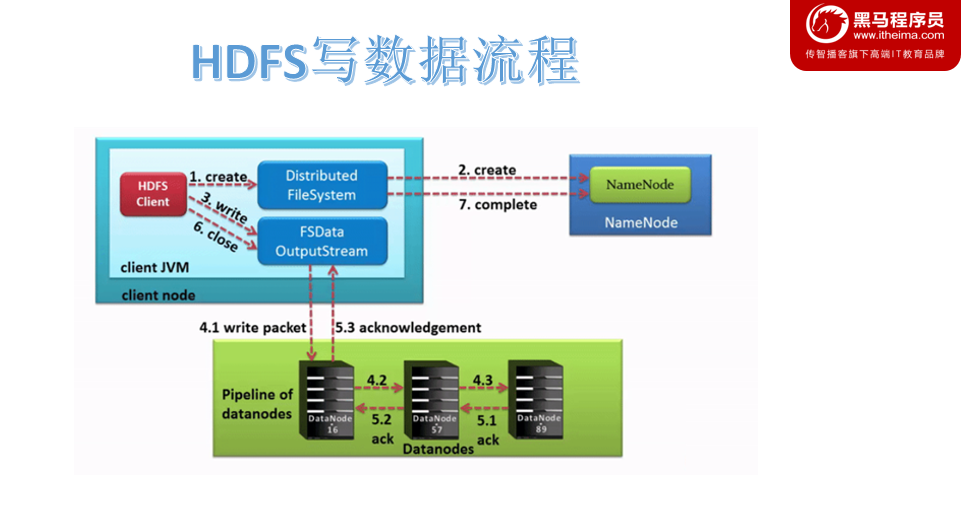
安全模式操作命令

|  |
| --- |
| hdfs dfsadmin -safemode get #查看安全模式状态  hdfs dfsadmin -safemode enter #进入安全模式  hdfs dfsadmin -safemode leave #离开安全模式 |

## HDFS的数据读写流程

因为namenode维护管理了文件系统的元数据信息，这就造成了不管是读还是写数据都是基于NameNode开始的，也就是说**NameNode成为了HDFS访问的唯一入口**。入口地址是：<http://nn_host:8020>。

### 写数据流程

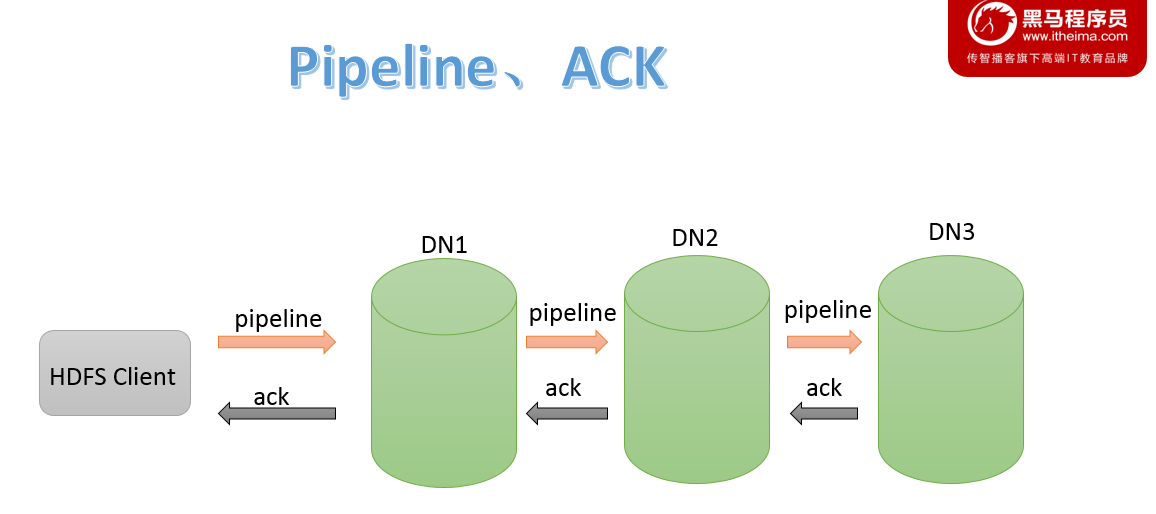


#### Pipeline管道、ACK应答响应

**Pipeline**，中文翻译为管道。这是HDFS在上传文件写数据过程中采用的一种数据传输方式。客户端将数据块写入第一个数据节点，第一个数据节点保存数据之后再将块复制到第二个数据节点，后者保存后将其复制到第三个数据节点。通俗描述pipeline的过程就是：**Client🡪A🡪B🡪C。**

为什么datanode之间采用pipeline**线性传输**，而不是一次给三个datanode拓扑式传输呢？因为数据以管道的方式，顺序的沿着一个方向传输，这样能够充分利用每个机器的带宽，避免网络瓶颈和高延迟时的连接，最小化推送所有数据的延时。在线性推送模式下，每台机器所有的出口宽带都用于以最快的速度传输数据，而不是在多个接受者之间分配宽带。

**ACK** (Acknowledge character）即是确认字符，在数据通信中，接收方发给发送方的一种传输类控制字符。表示发来的数据已确认接收无误。在pipeline管道传输数据的过程中，传输的反方向会进行ACK校验，确保数据传输安全。

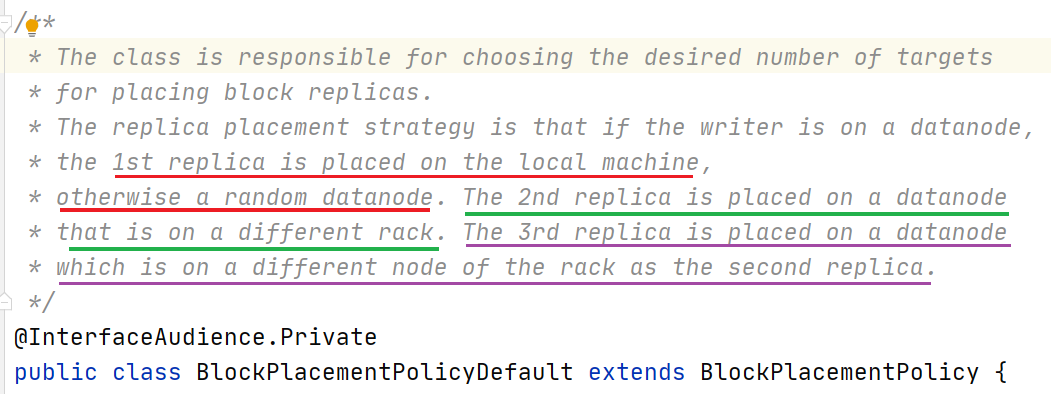


#### 具体流程

* HDFS客户端通过对DistributedFileSystem 对象调用**create()**请求创建文件。
* DistributedFileSystem对namenode进行**RPC调用**，请求上传文件。namenode执行各种检查判断：目标文件是否存在、父目录是否存在、客户端是否具有创建该文件的权限。检查通过，namenode就会为创建新文件记录一条记录。否则，文件创建失败并向客户端抛出一个IOException。
* DistributedFileSystem为客户端返回FSDataOutputStream输出流对象。由此客户端可以开始写入数据。FSDataOutputStream是一个包装类，所包装的是DFSOutputStream。
* 在客户端写入数据时，DFSOutputStream将它分成一个个数据包（**packet** 默认64kb）,并写入一个称之为数据队列（data queue）的内部队列。DFSOutputStream有一个内部类做DataStreamer，用于请求NameNode挑选出适合存储数据副本的一组DataNode。这一组DataNode采用**pipeline机制**做数据的发送。默认是3副本存储。
* DataStreamer将数据包流式传输到pipeline的第一个datanode,该DataNode存储数据包并将它发送到pipeline的第二个DataNode。同样，第二个DataNode存储数据包并且发送给第三个（也是最后一个）DataNode。
* DFSOutputStream也维护着一个内部数据包队列来等待DataNode的收到确认回执，称之为确认队列（ack queue）,收到pipeline中所有DataNode确认信息后，该数据包才会从确认队列删除。
* 客户端完成数据写入后，将在流上调用close()方法关闭。该操作将剩余的所有数据包写入DataNode pipeline，并在联系到NameNode告知其文件写入完成之前，等待确认。
* 因为namenode已经知道文件由哪些块组成（DataStream请求分配数据块），因此它仅需等待最小复制块即可成功返回。
* 数据块最小复制是由参数dfs.namenode.replication.min指定，默认是1.

#### 默认3副本存储策略

默认副本存储策略是由BlockPlacementPolicyDefault指定。策略如下：

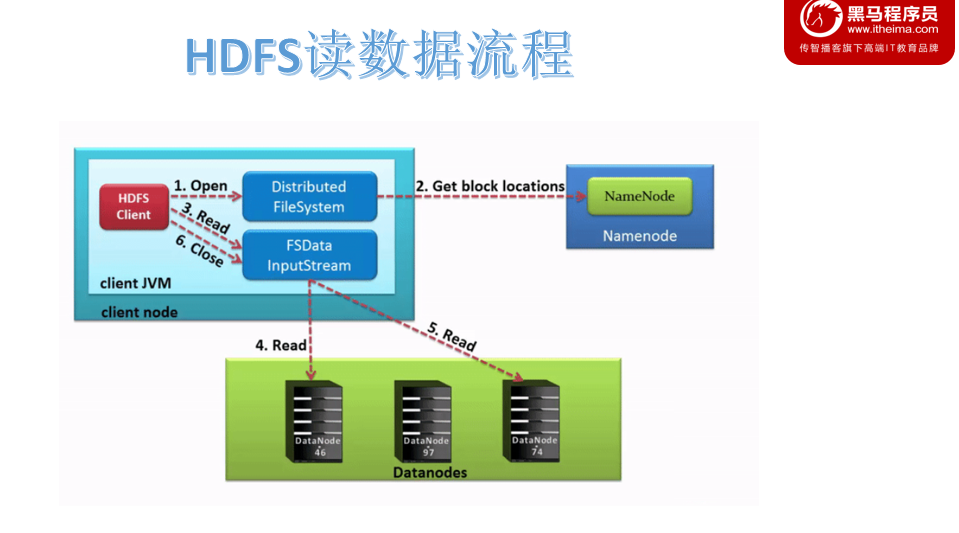


第一块副本：优先客户端本地，否则随机

第二块副本：不同于第一块副本的不同机架。

第三块副本：第二块副本相同机架不同机器。

### 读数据流程



#### 具体流程

* 客户端通过调用DistributedFileSystem对象上的open()来打开希望读取的文件。
* DistributedFileSystem使用**RPC调用**namenode来确定文件中**前几个块**的块位置。对于每个块，namenode返回具有该块副本的datanode的地址，并且datanode根据块与客户端的距离进行排序。注意此距离指的是**网络拓扑中的距离**。比如客户端的本身就是一个DataNode，那么从本地读取数据明显比跨网络读取数据效率要高。
* DistributedFileSystem将FSDataInputStream（支持文件seek定位读的输入流）返回到客户端以供其读取数据。FSDataInputStream类转而封装为DFSInputStream类，DFSInputStream管理着datanode和namenode之间的IO。
* 客户端在流上调用read()方法。然后，已存储着文件前几个块DataNode地址的DFSInputStream随即连接到文件中第一个块的最近的DataNode节点。通过对数据流反复调用read()方法，可以将数据从DataNode传输到客户端。
* 当该块快要读取结束时，DFSInputStream将关闭与该DataNode的连接，然后寻找下一个块的最佳datanode。这些操作对用户来说是透明的。所以用户感觉起来它一直在读取一个连续的流。
* 客户端从流中读取数据时，块是按照打开DFSInputStream与DataNode新建连接的顺序读取的。它也会根据需要询问NameNode来检索下一批数据块的DataNode位置信息。一旦客户端完成读取，就对FSDataInputStream调用close()方法。
* 如果DFSInputStream与DataNode通信时遇到错误，它将尝试该块的下一个最接近的DataNode读取数据。并将记住发生故障的DataNode，保证以后不会反复读取该DataNode后续的块。此外，DFSInputStream也会通过校验和（checksum）确认从DataNode发来的数据是否完整。如果发现有损坏的块，DFSInputStream会尝试从其他DataNode读取该块的副本，也会将被损坏的块报告给namenode 。

## HDFS的元数据辅助管理

### 元数据是什么

**元数据（Metadata）**，又称中介数据，为描述[数据](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE/5947370)的数据（data about data），主要是描述数据[属性](https://baike.baidu.com/item/%E5%B1%9E%E6%80%A7/1405051)（property）的[信息](https://baike.baidu.com/item/%E4%BF%A1%E6%81%AF/111163)，用来支持如指示存储位置、[历史](https://baike.baidu.com/item/%E5%8E%86%E5%8F%B2/360)数据、[资源](https://baike.baidu.com/item/%E8%B5%84%E6%BA%90/9089683)查找、文件记录等功能。

在HDFS中，元数据主要指的是**文件相关的元数据**，由NameNode管理维护。从广义的角度来说，因为NameNode还需要管理众多DataNode节点，因此DataNode的位置和健康状态信息也属于元数据。

### 元数据管理概述

在HDFS中，文件相关元数据具有两种类型：

* **文件自身属性信息**

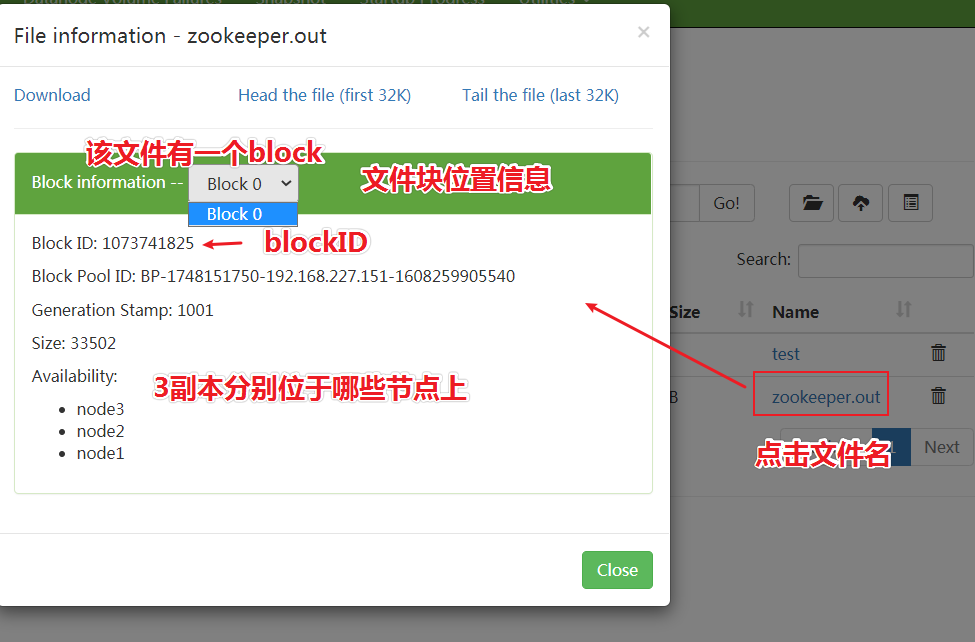
文件名称、权限，修改时间，文件大小，复制因子，数据块大小。



* **文件块位置映射信息**

记录文件块和DataNode之间的映射信息，即哪个块位于哪个节点上。

按存储形式分为内存元数据和元数据文件两种，分别存在内存和磁盘上。



#### 内存元数据

为了保证用户操作元数据交互高效，延迟低，NameNode把所有的元数据都存储在内存中，我们叫做内存元数据。内存中的元数据是最完整的，包括文件自身属性信息、文件块位置映射信息。

但是内存的致命问题是，断点数据丢失，数据不会持久化。因此NameNode又辅佐了元数据文件来保证元数据的安全完整。

#### 磁盘元数据文件

##### fsimage 内存镜像文件

是内存元数据的一个持久化的检查点。但是**fsimage中仅包含Hadoop文件系统中文件自身属性相关的元数据信息，但不包含文件块位置的信息**。文件块位置信息只存储在内存中，时由datanode启动加入集群的时候，向namenode进行数据块的汇报得到的，并且后续间断指定时间进行数据块报告。

持久化的动作是一种数据从内存到磁盘的IO过程。会对namenode正常服务造成一定的影响，不能频繁的进行持久化。

##### Edits log编辑日志

为了避免两次持久化之间数据丢失的问题，又设计了Edits log编辑日志文件。**文件中记录的是HDFS所有更改操作（文件创建，删除或修改）的日志**，文件系统客户端执行的更改操作首先会被记录到edits文件中。

#### 加载元数据顺序

fsimage和edits文件都是经过序列化的，在**NameNode启动的时候，它会将fsimage文件中的内容加载到内存中，之后再执行edits文件中的各项操作，使得内存中的元数据和实际的同步**，存在内存中的元数据支持客户端的读操作，也是最完整的元数据。

当客户端对HDFS中的文件进行新增或者修改操作，操作记录首先被记入edits日志文件中，当客户端操作成功后，相应的元数据会更新到内存元数据中。因为fsimage文件一般都很大（GB级别的很常见），如果所有的更新操作都往fsimage文件中添加，这样会导致系统运行的十分缓慢。

HDFS这种设计实现着手于：一是内存中数据更新、查询快，极大缩短了操作响应时间；二是内存中元数据丢失风险颇高（断电等），因此辅佐元数据镜像文件（fsimage）+编辑日志文件（edits）的备份机制进行确保元数据的安全。

NameNode维护整个文件系统元数据。因此，元数据的准确管理，影响着HDFS提供文件存储服务的能力。

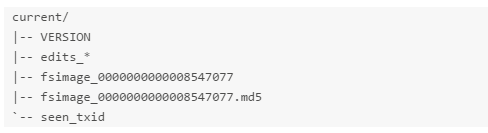
### 元数据管理相关目录文件

#### 元数据存储目录

在Hadoop的HDFS首次部署好配置文件之后，并不能马上启动使用，而是先要对文件系统进行格式化操作:hdfs namenode -format

在这里要注意两个概念，一个是format之前，HDFS在物理上还不存在；二就是此处的**format**并不是指传统意义上的本地磁盘格式化，而是一些清除与准备工作。其中就会创建元数据本地存储目录和一些初始化的元数据相关文件。

namenode元数据存储目录由参数：dfs.namenode.name.dir指定，格式化完成之后，将会在$dfs.namenode.name.dir/current目录下创建如下的文件：



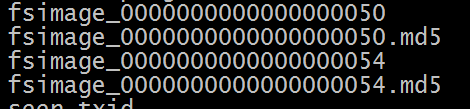
其中的dfs.namenode.name.dir是在hdfs-site.xml文件中配置的，默认值如下：



**dfs.namenode.name.dir属性可以配置多个目录**，各个目录存储的文件结构和内容都完全一样，相当于备份，这样做的好处是当其中一个目录损坏了，也不会影响到hadoop的元数据，特别是当其中一个目录是NFS（网络文件系统Network File System，NFS）之上，即使你这台机器损坏了，元数据也得到保存。

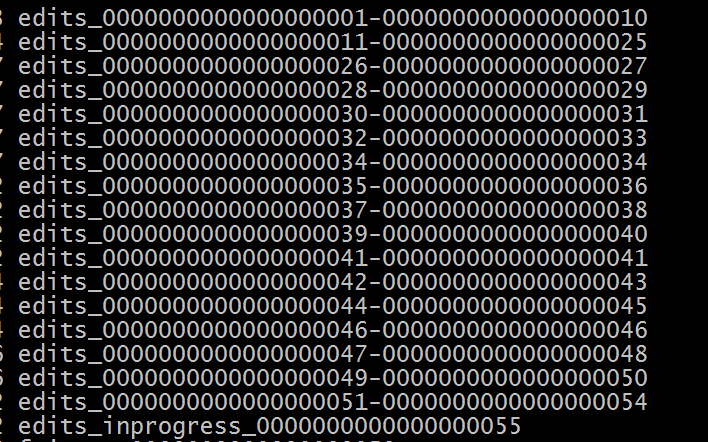
#### 元数据相关文件

##### fsimage相关



元数据镜像文件。每个fsimage文件还有一个对应的.md5文件，其中包含MD5校验和，HDFS使用该文件来防止磁盘损坏文件异常。

##### Edits log相关



已完成且不可修改的编辑日志。这些文件中的每个文件都包含文件名定义的范围内的所有编辑日志事务。在HA高可用性部署中，主备namenode之间可以通过edits log进行数据同步。

#### Fsimage、editslog查看

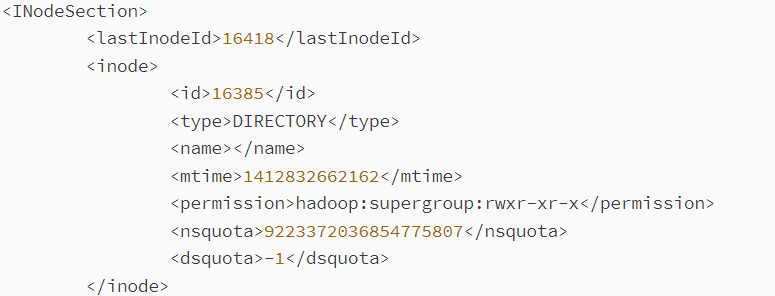
##### Fsimage

**fsimage**文件是Hadoop文件系统元数据的一个永久性的检查点，包含Hadoop文件系统中的所有目录和文件idnode的序列化信息；对于文件来说，包含的信息有修改时间、访问时间、块大小和组成一个文件块信息等；而对于目录来说，包含的信息主要有修改时间、访问控制权限等信息。

**oiv是offline image viewer**的缩写，用于将fsimage文件的内容转储到指定文件中以便于阅读，该工具还提供了只读的WebHDFS API以允许离线分析和检查hadoop集群的命名空间。

oiv在处理非常大的fsimage文件时是相当快的，如果该工具不能够处理fsimage，它会直接退出。该工具不具备向后兼容性，比如使用hadoop-2.4版本的oiv不能处理hadoop-2.3版本的fsimage，只能使用hadoop-2.3版本的oiv。就像它的名称所提示的（offline），oiv不需要hadoop集群处于运行状态。

命令：hdfs oiv -i fsimage\_0000000000000000050 -p XML -o fsimage.xml



##### edits log

**edits log**文件存放的是Hadoop文件系统的所有更新操作记录日志，文件系统客户端执行的所有写操作首先会被记录到edits文件中。

NameNode起来之后，HDFS中的更新操作会重新写到edits文件中，因为fsimage文件一般都很大（GB级别的很常见），如果所有的更新操作都往fsimage文件中添加，这样会导致系统运行的十分缓慢，但是如果往edits文件里面写就不会这样，每次执行写操作之后，且在向客户端发送成功代码之前，edits文件都需要同步更新。如果一个文件比较大，使得写操作需要向多台机器进行操作，只有当所有的写操作都执行完成之后，写操作才会返回成功，这样的好处是任何的操作都不会因为机器的故障而导致元数据的不同步。

**oev是offline edits viewer**（离线edits查看器）的缩写，该工具不需要hadoop集群处于运行状态。

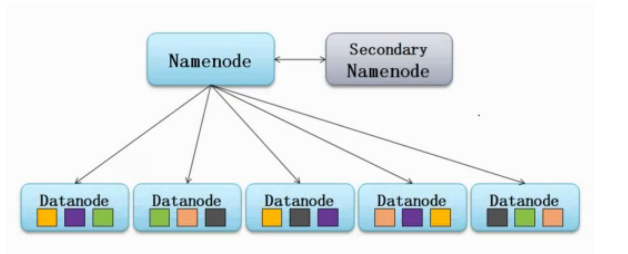
命令：hdfs oev -i edits\_0000000000000000011-0000000000000000025 -o edits.xml

在输出文件中，每个RECORD记录了一次操作,示例如下：



### SecondaryNamenode

#### SNN职责概述



NameNode职责是管理元数据信息，DataNode的职责是负责数据具体存储，那么SecondaryNameNode的作用是什么？对很多初学者来说是非常迷惑的。它为什么会出现在HDFS中。从它的名字上看，它给人的感觉就像是NameNode的备份。但它实际上却不是。

当HDFS集群运行一段事件后，就会出现下面一些问题：

* edits logs会变的很大，fsimage将会变得很旧；
* namenode重启会花费很长时间，因为有很多改动要合并到fsimage文件上；
* 如果频繁进行fsimage持久化，又会影响NameNode正常服务，毕竟IO操作是一种内存到磁盘的耗精力操作

因此为了克服这个问题，需要一个易于管理的机制来帮助我们**减小edit logs文件的大小和得到一个最新的fsimage文件**，这样也会减小在NameNode上的压力。

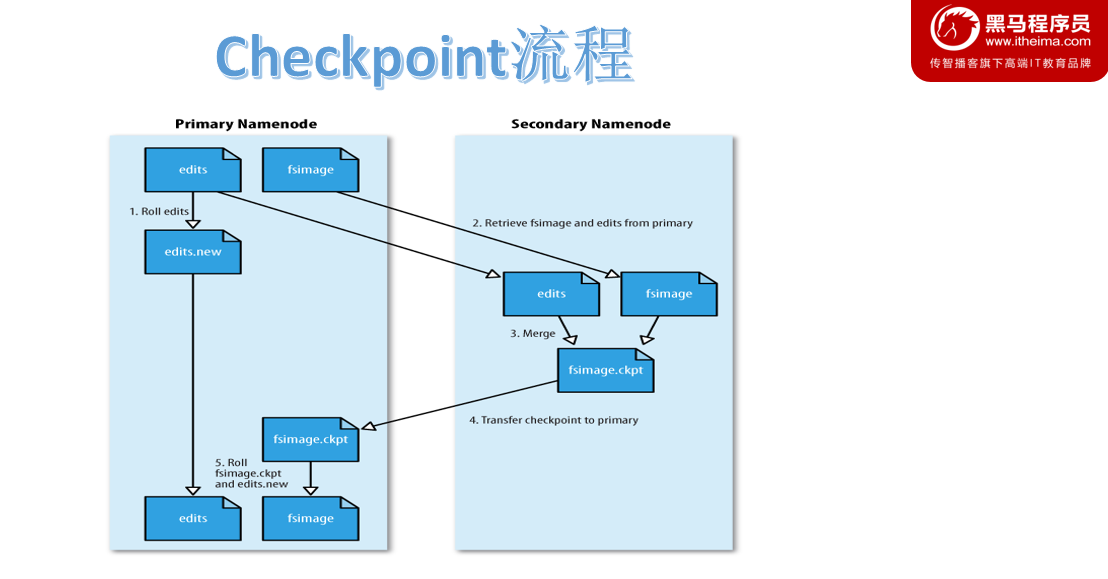
**SecondaryNameNod**e就是来帮助解决上述问题的，它的职责是**合并NameNode的edit logs到fsimage文件中**。

#### SNN checkpoint机制

##### 概述

**Checkpoint**核心是把fsimage与edits log合并以生成新的fsimage的过程。此过程有两个好处：fsimage版本不断更新不会太旧、edits log文件不会太大。

##### 流程



* 当触发checkpoint操作条件时，SNN发生请求给NN滚动edits log。然后NN会生成一个新的编辑日志文件：edits new，便于记录后续操作记录。
* 同时SNN会将edits文件和fsimage复制到本地（使用HTTP GET方式）。
* SNN首先将fsimage载入到内存，然后一条一条地执行edits文件中的操作，使得内存中的fsimage不断更新，这个过程就是edits和fsimage文件合并。合并结束，SNN将内存中的数据dump生成一个新的fsimage文件。
* SNN将新生成的Fsimage new文件复制到NN节点。
* 至此刚好是一个轮回，等待下一次checkpoint触发SecondaryNameNode进行工作，一直这样循环操作。

##### 触发机制

Checkpoint触发条件受两个参数控制，可以通过**core-site.xml**进行配置：

|  |
| --- |
| dfs.namenode.checkpoint.period=3600 //两次连续的checkpoint之间的时间间隔。默认1小时  dfs.namenode.checkpoint.txns=1000000 //最大没有执行checkpoint事务的数量，满足将强制执行紧急checkpoint，即使尚未达到检查点周期。默认100万事务数量。 |

从上面的描述我们可以看出，**SecondaryNamenode根本就不是Namenode的一个热备，只是将fsimage和edits合并**。

### Namenode 元数据恢复

#### NameNode存储多目录

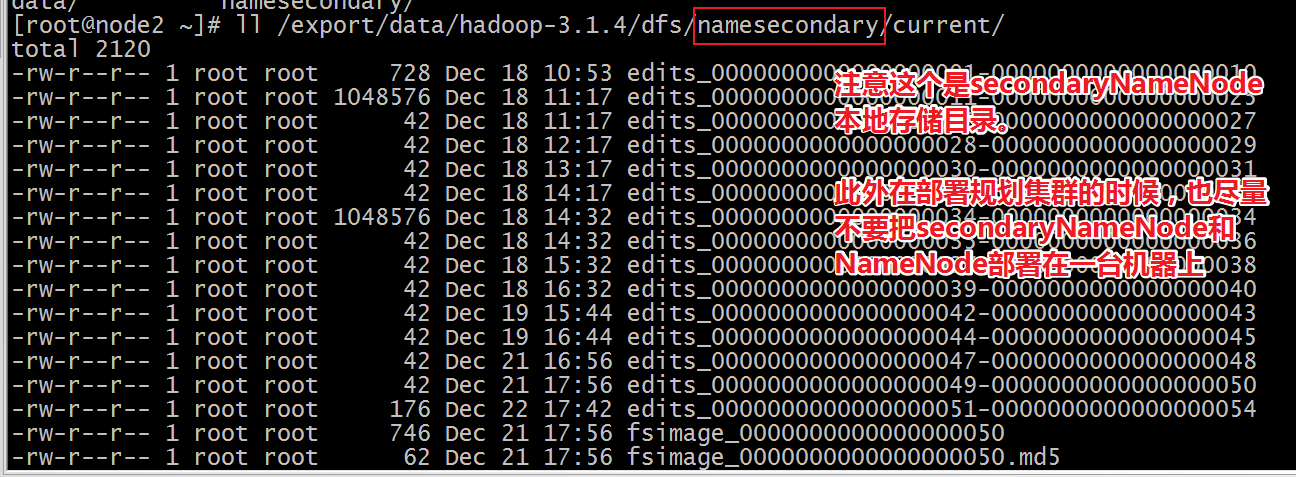
namenode元数据存储目录由参数：dfs.namenode.name.dir指定。

**dfs.namenode.name.dir属性可以配置多个目录**，各个目录存储的文件结构和内容都完全一样，相当于备份，这样做的好处是当其中一个目录损坏了，也不会影响到hadoop的元数据，特别是当其中一个目录是NFS（网络文件系统Network File System，NFS）之上，即使你这台机器损坏了，元数据也得到保存。

#### 从SecondaryNameNode恢复

SecondaryNameNode在checkpoint的时候会将fsimage和edits log下载到自己的本机上本地存储目录下。并且在checkpoint之后也不会进行删除。

如果NameNode中的fsimage真的出问题了，还是可以用SecondaryNamenode中的fsimage替换一下NameNode上的fsimage，虽然已经不是最新的fsimage，但是我们可以将损失减小到最少！



## 使用文件系统方式访问数据

### HDFS操作 - Java客户端

#### 介绍

HDFS在生产应用中主要是客户端的开发，其核心步骤是从HDFS提供的api中构造一个HDFS的访问客户端对象，然后通过该客户端对象操作（增删改查）HDFS上的文件。

#### 配置Windows下Hadoop环境

**问题**

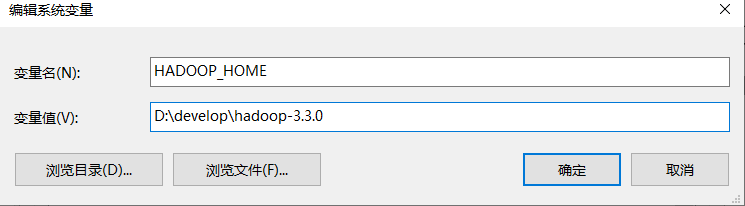
在windows上做HDFS客户端应用开发，需要设置Hadoop环境,而且要求是windows平台编译的Hadoop,不然会报以下的错误:

|  |
| --- |
| **缺少winutils.exe**  Could not locate executable null \bin\winutils.exe in the hadoop binaries  **缺少hadoop.dll**  Unable to load native-hadoop library for your platform… using builtin-Java classes where applicable |

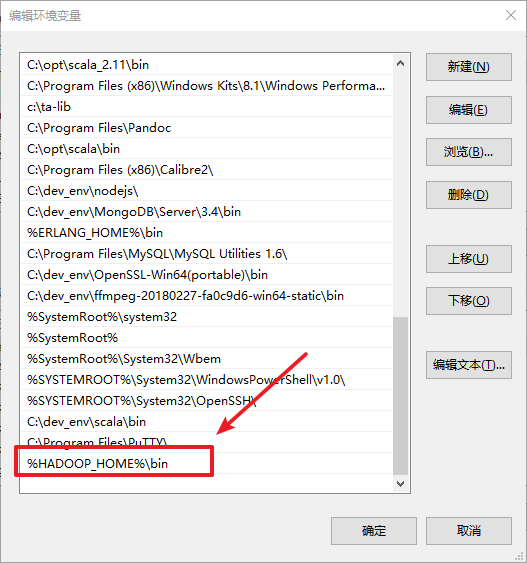
**搭建步骤**

1. 将已经编译好的Windows版本Hadoop解压到到一个没有中文没有空格的路径下面
2. 在windows上面配置hadoop的环境变量： HADOOP\_HOME，并将%HADOOP\_HOME%\bin添加到path中

配置HADOOP\_HOME



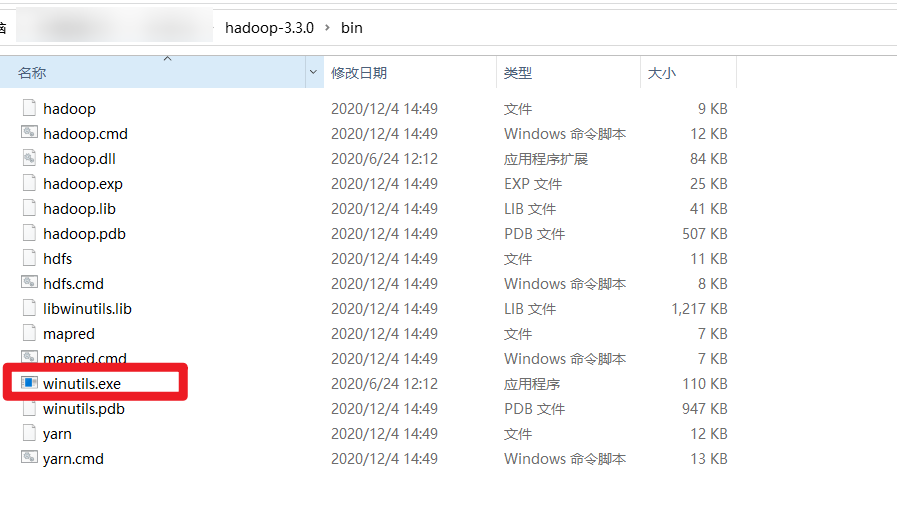
配置PATH目录



1. 把hadoop3.3.0文件夹中bin目录下的hadoop.dll文件放到系统盘: C:\Windows\System32 目录
2. 重启windows系统

5. 验证系统是否兼容hadoop

双击bin目录下的：winutils.exe文件，如果一闪而过则正常，否则找老师



### 创建项目

在IDEA中创建项目：

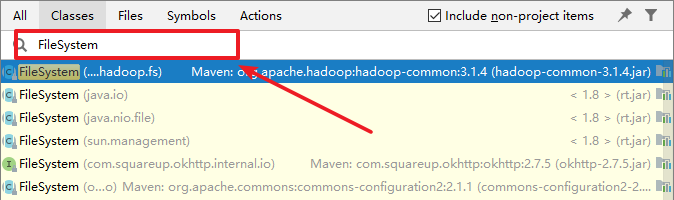
|  |  |
| --- | --- |
| groupid | cn.itcast |
| artifactid | day07\_hdfs\_api |

#### 导入Maven依赖

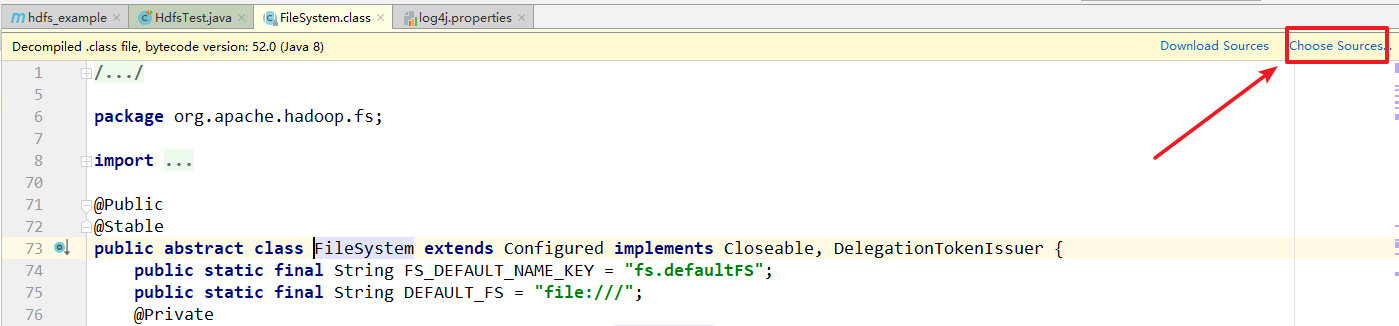
|  |
| --- |
| <**repositories**>  <**repository**>  <**id**>cental</**id**>  <**url**>http://maven.aliyun.com/nexus/content/groups/public//</**url**>  <**releases**>  <**enabled**>true</**enabled**>  </**releases**>  <**snapshots**>  <**enabled**>true</**enabled**>  <**updatePolicy**>always</**updatePolicy**>  <**checksumPolicy**>fail</**checksumPolicy**>  </**snapshots**>  </**repository**> </**repositories**>  <**dependencies**>  <**dependency**>  <**groupId**>org.apache.hadoop</**groupId**>  <**artifactId**>hadoop-common</**artifactId**>  <**version**>3.3.0</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.hadoop</**groupId**>  <**artifactId**>hadoop-client</**artifactId**>  <**version**>3.3.0</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.hadoop</**groupId**>  <**artifactId**>hadoop-hdfs</**artifactId**>  <**version**>3.3.0</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.hadoop</**groupId**>  <**artifactId**>hadoop-mapreduce-client-core</**artifactId**>  <**version**>3.3.0</**version**>  </**dependency**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.13</**version**>  </**dependency**>   <!-- Google Options -->  <**dependency**>  <**groupId**>com.github.pcj</**groupId**>  <**artifactId**>google-options</**artifactId**>  <**version**>1.0.0</**version**>  </**dependency**>  <**dependency**>  <**groupId**>commons-io</**groupId**>  <**artifactId**>commons-io</**artifactId**>  <**version**>2.6</**version**>  </**dependency**> </**dependencies**> |

#### 关联Hadoop源代码

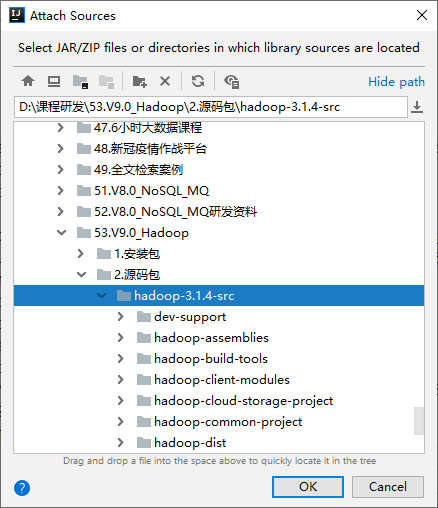
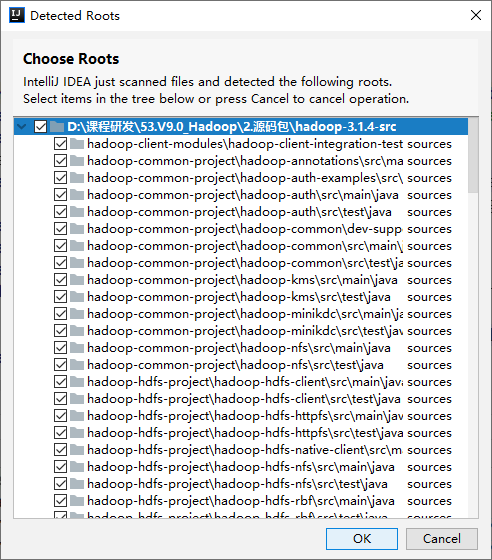
1. 点击IDEA中的【Navigate】菜单，再点击【Class…】菜单
2. 输入【FileSystem】搜索



1. 点击【Choose Sources…】



1. 找到源代码目录，关联即可。

### 实现HDFS操作类

#### HDFS API介绍

**涉及的主要类**

在Java中操作HDFS，主要涉及以下Class：

* **Configuration：**该类的对象封转了客户端或者服务器的配置
* **FileSystem：**该类的对象是一个文件系统对象，可以用该对象的一些方法来对文件进行操作，通过FileSystem的静态方法get获得该对象。

FileSystem fs = FileSystem.*get*(conf);

* get方法从conf中的一个参数 fs.defaultFS的配置值判断具体是什么类型的文件系统。如果我们的代码中没有指定fs.defaultFS，并且工程classpath下也没有给定相应的配置，conf中的默认值就来自于hadoop的jar包中的core-default.xml，默认值为： file:///，则获取的将不是一个DistributedFileSystem的实例，而是一个本地文件系统的客户端对象。

Java API官方文档：

[https://hadoop.apache.org/docs/r3.3.0/api/index.html](https://hadoop.apache.org/docs/r3.1.4/api/index.html)

##### 获取FileSystem方式

### 使用文件系统方式访问数据

#### 涉及的主要类

在java中操作HDFS，主要涉及以下Class：

**Configuration：**该类的对象封转了客户端或者服务器的配置;

**FileSystem：**该类的对象是一个文件系统对象，可以用该对象的一些方法来对文件进行操作，通过FileSystem的静态方法get获得该对象。

|  |
| --- |
| FileSystem fs = FileSystem.*get*(conf); |

get方法从conf中的一个参数 fs.defaultFS的配置值判断具体是什么类型的文件系统。如果我们的代码中没有指定fs.defaultFS，并且工程classpath下也没有给定相应的配置，conf中的默认值就来自于hadoop的jar包中的core-default.xml，默认值为： file:///，则获取的将不是一个DistributedFileSystem的实例，而是一个本地文件系统的客户端对象。

#### 获取FileSystem方式

|  |
| --- |
| @Test **public void** getFileSystem2() **throws** Exception{  FileSystem fileSystem = FileSystem.*get*(**new** URI(**"hdfs://node1:8020"**), **new** Configuration());  System.***out***.println(**"fileSystem:"**+fileSystem); } |

#### 遍历HDFS中所有文件

|  |
| --- |
| @Test **public void** listMyFiles()**throws** Exception{  *//获取fileSystem类* FileSystem fileSystem = FileSystem.*get*(**new** URI(**"hdfs://node1:8020"**), **new** Configuration());  *//获取RemoteIterator 得到所有的文件或者文件夹，第一个参数指定遍历的路径，第二个参数表示是否要递归遍历* RemoteIterator<LocatedFileStatus> locatedFileStatusRemoteIterator = fileSystem.listFiles(**new** Path(**"/"**), **true**);  **while** (locatedFileStatusRemoteIterator.hasNext()){  LocatedFileStatus next = locatedFileStatusRemoteIterator.next();  System.***out***.println(next.getPath().toString());  }  fileSystem.close(); } |

#### HDFS上创建文件夹

|  |
| --- |
| @Test **public void** mkdirs() **throws** Exception{  FileSystem fileSystem = FileSystem.*get*(**new** URI(**"hdfs://node1:8020"**), **new** Configuration());  **boolean** mkdirs = fileSystem.mkdirs(**new** Path(**"/hello/mydir/test"**));  fileSystem.close(); } |

#### 下载文件-方式1

|  |
| --- |
| @Test **public void** getFileToLocal()**throws** Exception{  FileSystem fileSystem = FileSystem.*get*(**new** URI(**"hdfs://node1:8020"**), **new** Configuration());  FSDataInputStream inputStream = fileSystem.open(**new** Path(**"/timer.txt"**));  FileOutputStream outputStream = **new** FileOutputStream(**new** File(**"e:\\timer.txt"**));  IOUtils.*copy*(inputStream,outputStream );  IOUtils.*closeQuietly*(inputStream);  IOUtils.*closeQuietly*(outputStream);  fileSystem.close(); } |

#### 下载文件-方式2

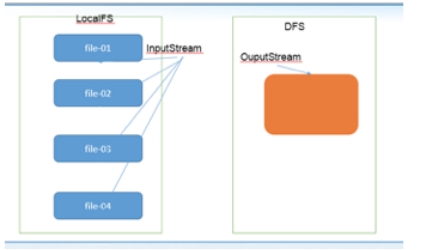
|  |
| --- |
| @Test  public void downLoadFile**()** **throws** URISyntaxException**,** IOException**,** InterruptedException **{**  //1:获取FiletSystem对象  FileSystem fileSystem **=** FileSystem**.**get**(new** URI**(**"hdfs://node1:8020"**),** **new** Configuration**());**  //2:实现文件下载  fileSystem**.**copyToLocalFile**(new** Path**(**"/anaconda-ks.cfg"**),** **new** Path**(**"E:\\test"**));**  //3:释放资源  fileSystem**.**close**();**  **}** |

#### 上传文件

|  |
| --- |
| @Test **public void** putData() **throws** Exception{  FileSystem fileSystem = FileSystem.*get*(**new** URI(**"hdfs://node1:8020"**), **new** Configuration());  fileSystem.copyFromLocalFile(**new** Path(**"file:///c:\\install.log"**),**new** Path(**"/hello/mydir/test"**));  fileSystem.close(); } |

#### 小文件合并

由于 Hadoop 擅长存储大文件，因为大文件的元数据信息比较少，如果 Hadoop 集群当中有大量的小文件，那么每个小文件都需要维护一份元数据信息，会大大的增加集群管理元数据的内存压力，所以在实际工作当中，如果有必要一定要将小文件合并成大文件进行一起处理,可以在上传的时候将小文件合并到一个大文件里面去



|  |
| --- |
| @Test **public void** mergeFile() **throws** Exception{  *//获取分布式文件系统* FileSystem fileSystem = FileSystem.*get*(**new** URI(**"hdfs://node1:8020"**), **new** Configuration(),**"root"**);  FSDataOutputStream outputStream = fileSystem.create(**new** Path(**"/bigfile.txt"**));  *//获取本地文件系统* LocalFileSystem local = FileSystem.*getLocal*(**new** Configuration());  *//通过本地文件系统获取文件列表，为一个集合* FileStatus[] fileStatuses = local.listStatus(**new** Path(**"file:///E:\\input"**));  **for** (FileStatus fileStatus : fileStatuses) {  FSDataInputStream inputStream = local.open(fileStatus.getPath());  IOUtils.*copy*(inputStream,outputStream);  IOUtils.*closeQuietly*(inputStream);  }  IOUtils.*closeQuietly*(outputStream);  local.close();  fileSystem.close(); } |

## 集群之间的数据复制

在我们实际工作当中，极有可能会遇到将测试集群的数据拷贝到生产环境集群，或者将生产环境集群的数据拷贝到测试集群，那么就需要我们在多个集群之间进行数据的远程拷贝，hadoop自带也有命令可以帮我们实现这个功能。

### 集群内部文件拷贝scp

### 本地复制到远程

方式1：指定用户名，命令执行后需要再输入密码；

scp -r local\_folder remote\_username@remote\_ip:remote\_folder

方式2:没有指定用户名，命令执行后需要输入用户名和密码；

scp -r local\_folder remote\_ip:remote\_folder

注意，如果实现了ssh免密登录之后，则不需要输入密码即可拷贝。

实例:

|  |
| --- |
| #**复制文件**-将 /root/test.txt 拷贝到 192.168.88.161 的 /root/ 目录下，文件名还是 text.txt，使用 root 用户，此时会提示输入远程 root 用户的密码。  scp **/**root**/**test.txt root**@**192**.**168**.**88**.**161**:/**root**/**  #**复制文件并重命名**-将 /root/test.txt 拷贝到 192.168.88.161 的 /root/ 目录下，文件名还是 text1.txt，使用 root 用户，此时会提示输入远程 root 用户的密码。  scp **/**root**/**test.txt root**@**192**.**168**.**88**.**161**:/**root**/**test1.txt  #**复制目录**-将整个目录 /root/test/ 复制到 192.168.88.161 的 /root/ 下，即递归的复制，使用 root 用户，此时会提示输入远程 root 用户的密码。  scp -r **/**root**/**test**/** root**@**192**.**168**.**88**.**161**:/**root**/** |

### 远程复制到本地

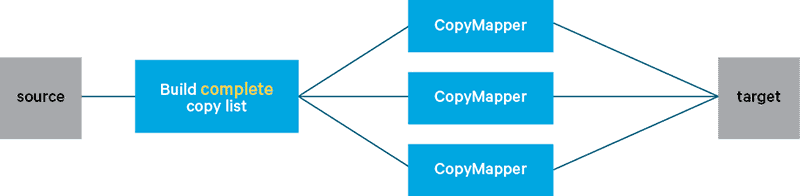
远程复制到本地 与 从本地复制到远程命令类似，不同的是 远程文件作为源文件在前，本地文件作为目标文件在后。

|  |
| --- |
| #复制文件-将192.168.88.162的/root目录下的test.txt拷贝到当前主机的/root/目录下，文件名不变  scp root**@**192**.**168**.**88**.**162**:/**root**/**test.txt **/**root**/**test.txt |

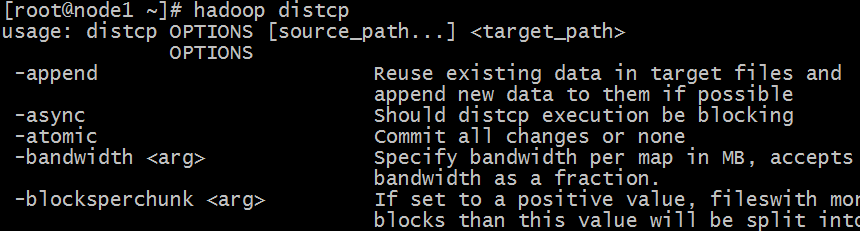
### HDFS分布式拷贝工具：DistCp

#### DistCp介绍

DistCp是Apache Hadoop中的一种流行工具，在hadoop-tools工程下，作为独立子工程存在。其定位就是用于数据迁移的，定期在集群之间和集群内部备份数据。（在备份过程中，每次运行DistCp都称为一个备份周期。）尽管性能相对较慢，但它的普及程度已经越来越高。



#### DistCp命令



|  |
| --- |
| $ hadoop distcp  usage: distcp OPTIONS [source\_path...] <target\_path>    -append //拷贝文件时支持对现有文件进行追加写操作  -async //异步执行distcp拷贝任务  -bandwidth <arg> //对每个Map任务的带宽限速  -delete //删除相对于源端,目标端多出来的文件  -diff <arg> //通过快照diff信息进行数据的同步  -overwrite //以覆盖的方式进行拷贝，如果目标端文件已经存在，则直接覆盖  -p <arg> //拷贝数据时,扩展属性信息的保留，包括权限信息、块大小信息等等  -skipcrccheck //拷贝数据时是否跳过cheacksum的校验  -update //拷贝数据时,只拷贝相对于源端 ，目标端不存在的文件数据 |

其中 **source\_path 、target\_path 需要带上地址前缀以区分不同的集群**：

例如 ：hadoop distcp hdfs://nnl:8020/foo/a hdfs://nn2:8020/bar/foo

上面的命令表示从nnl集群拷贝／foo/a 路径下的数据到nn2集群的／bar/foo 路径下。

|  |
| --- |
| **cd** **/**export**/**servers**/**hadoop-2.7.5**/**  bin**/**hadoop distcp hdfs://node1:8020/jdk-8u241-linux-x64.tar.gz hdfs://cluster2:8020/ |

#### 同步集群配置文件

|  |
| --- |
| scp -r /export/server/hadoop-3.3.0/etc/hadoop/core-site.xml node2:/export/server/hadoop-3.3.0/etc/hadoop/  scp -r /export/server/hadoop-3.3.0/etc/hadoop/core-site.xml node3:/export/server/hadoop-3.3.0/etc/hadoop/ |

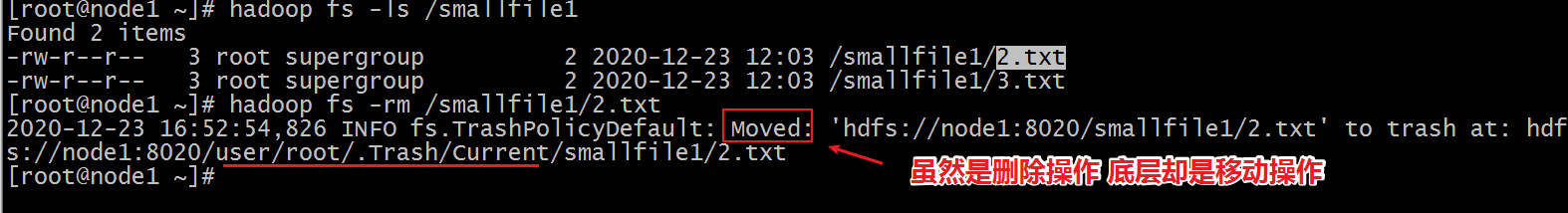
#### 启动HDFS集群

在node1节点上，执行一键启动HDFS集群命令：start-dfs.sh。

### 功能使用

#### 删除文件到Trash

开启Trash功能后，正常执行删除操作，文件实际并不会被直接删除，而是被移动到了垃圾回收站。



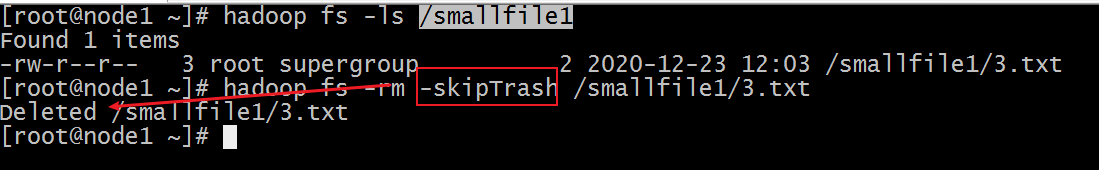
当然也可以去Trash回收站下面查看一下：



#### 删除文件跳过Trash

有的时候，我们希望直接把文件删除，不需要再经过Trash回收站了，可以在执行删除操作的时候添加一个参数：**-skipTrash.**

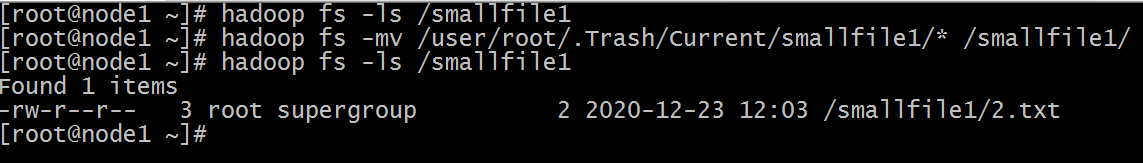
hadoop fs -rm -skipTrash /smallfile1/3.txt



#### 从Trash中恢复文件

回收站里面的文件，在到期被自动删除之前，都可以通过命令恢复出来。使用mv、cp命令把数据文件从Trash目录下复制移动出来就可以了。

hadoop fs -mv /user/root/.Trash/Current/smallfile1/\* /smallfile1/



#### 清空Trash

除了fs.trash.interval参数控制到期自动删除之外，用户还可以通过命令手动清空回收站，释放HDFS磁盘存储空间。

首先想到的是删除整个回收站目录，将会清空回收站,这是一个选择。此外。HDFS提供了一个命令行工具来完成这个工作：

hadoop fs -expunge

该命令立即从文件系统中删除过期的检查点。

## Archive档案的使用

HDFS并不擅长存储小文件，因为每个文件最少一个block，每个block的元数据都会在NameNode占用内存，如果存在大量的小文件，它们会吃掉NameNode节点的大量内存。

Hadoop Archives可以有效的处理以上问题，它可以把多个文件归档成为一个文件，归档成一个文件后还可以透明的访问每一个文件。

### 如何创建Archive

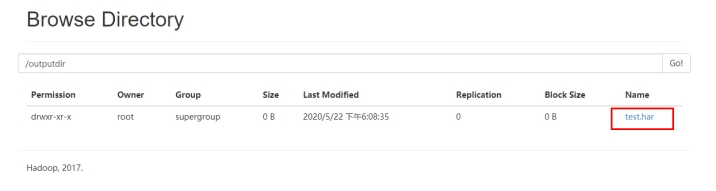
Usage: hadoop archive -archiveName name -p <parent> <src>\* <dest>

其中-archiveName是指要创建的存档的名称。比如test.har，archive的名字的扩展名应该是\*.har。 -p参数指定文件存档文件（src）的相对路径。

例如：如果你只想存档一个目录/config下的所有文件:

|  |
| --- |
| hadoop archive **-**archiveName test.har -p /config **/**outputdir |

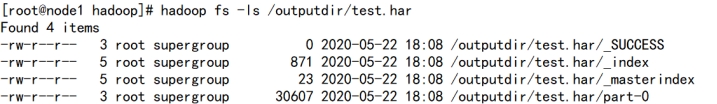
这样就会在/outputdir目录下创建一个名为test.har的存档文件。



### 如何查看Archive

首先我们来看下创建好的har文件。使用如下的命令：

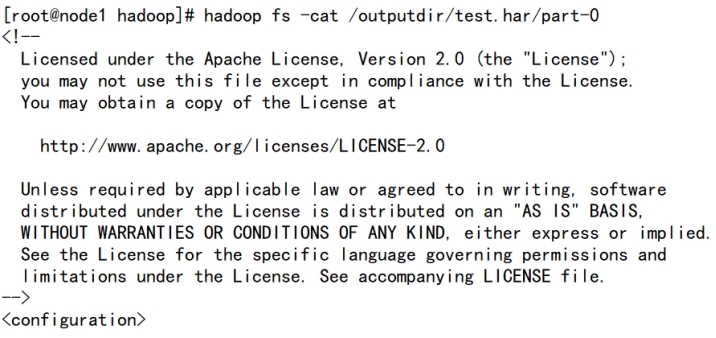
|  |
| --- |
| hadoop fs **-**ls **/**outputdir**/**test.har |



这里可以看到har文件包括：两个索引文件，多个part文件（本例只有一个）以及一个标识成功与否的文件。part文件是多个原文件的集合，根据index文件去找到原文件。

例如上述的/input目录下有很多小的xml文件。进行archive操作之后，这些小文件就归档到test.har里的part-0一个文件里。

|  |
| --- |
| hadoop fs **-**cat **/**outputdir**/**test.har**/**part-0 |



archive作为文件系统层暴露给外界。所以所有的fs shell命令都能在archive上运行，但是要使用不同的URI。Hadoop Archives的URI是：

har://scheme-hostname:port/archivepath/fileinarchive

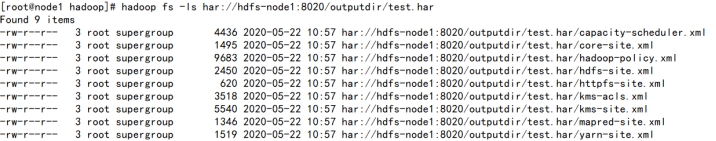
scheme-hostname格式为hdfs-域名:端口，如果没有提供scheme-hostname，它会使用默认的文件系统。这种情况下URI是这种形式：

har:///archivepath/fileinarchive

如果用har uri去访问的话，索引、标识等文件就会隐藏起来，只显示创建档案之前的原文件：

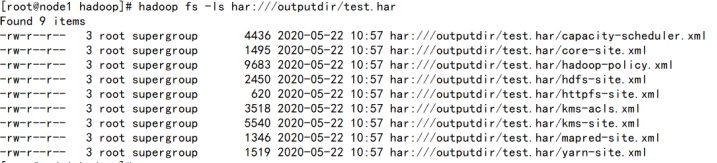
查看归档文件中的小文件,使用har uri

|  |
| --- |
| hadoop fs -ls har://hdfs-node1:8020/outputdir/test.har |



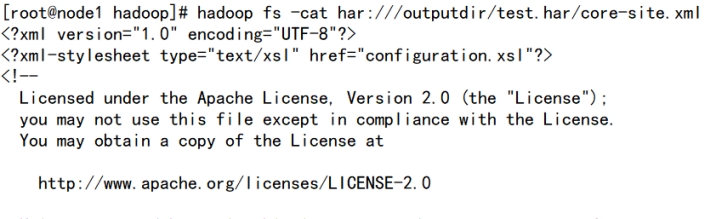
查看归档文件中的小文件,不使用har uri

|  |
| --- |
| hadoop fs -ls har:///outputdir/test.har |



查看har归档文件中小文件的内容

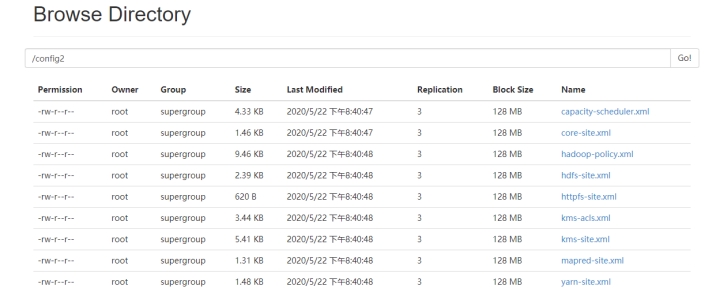
|  |
| --- |
| hadoop fs **-**cat har:///outputdir/test.har/core-site.xml |



### 如何解压Archive

|  |
| --- |
| hadoop fs -mkdir /config2  hadoop fs **-**cp har:///outputdir/test.har/**\*** /config2 |

查看HDFS页面，发现/config2目录中已经有解压后的小文件了



### Archive注意事项

1. Hadoop archives是特殊的档案格式。一个Hadoop archive对应一个文件系统目录。Hadoop archive的扩展名是\*.har；
2. 创建archives本质是运行一个Map/Reduce任务，所以应该在Hadoop集群上运行创建档案的命令，要提前启动Yarn集群；
3. 创建archive文件要消耗和原文件一样多的硬盘空间；
4. archive文件不支持压缩，尽管archive文件看起来像已经被压缩过；
5. archive文件一旦创建就无法改变，要修改的话，需要创建新的archive文件。事实上，一般不会再对存档后的文件进行修改，因为它们是定期存档的，比如每周或每日；
6. 当创建archive时，源文件不会被更改或删除；

## HDFS权限管理

### 总览概述

作为分布式文件系统，HDFS也集成了一套兼容POSIX的权限管理系统。客户端在进行每次文件操时，系统会从**用户身份认证**和**数据访问授权**两个环节进行验证： 客户端的操作请求会首先通过本地的用户身份验证机制来获得“凭证”（类似于身份证书），然后系统根据此“凭证”分辨出合法的用户名，再据此查看该用户所访问的数据是否已经授权。一旦这个流程中的某个环节出现异常，客户端的操作请求便会失败。



### UGO权限管理

#### 介绍

HDFS的文件权限与Linux/Unix系统的UGO模型类型类似，可以简单描述为：每个文件和目录都与一个所有者和一个组相关联。该文件或目录对作为**所有者（USER）**的用户，作为该**组成员的其他用户（GROUP）**以及对所有**其他用户（OTHER）**具有单独的权限。



在HDFS中，对于文件，需要r权限才能读取文件，而w权限才能写入或追加到文件。没有x可执行文件的概念。

对于目录，需要r权限才能列出目录的内容，需要w权限才能创建或删除文件或目录，并且需要x权限才能访问目录的子级。

#### UGO权限相关命令

hadoop fs -**chmod** 750 /user/itcast/foo //变更目录或文件的权限位

hadoop fs -**chown** :portal /user/itcast/foo //变更目录或文件的属主或用户组

hadoop fs -**chgrp** itcast \_group1 /user/itcast/foo //变更用户组

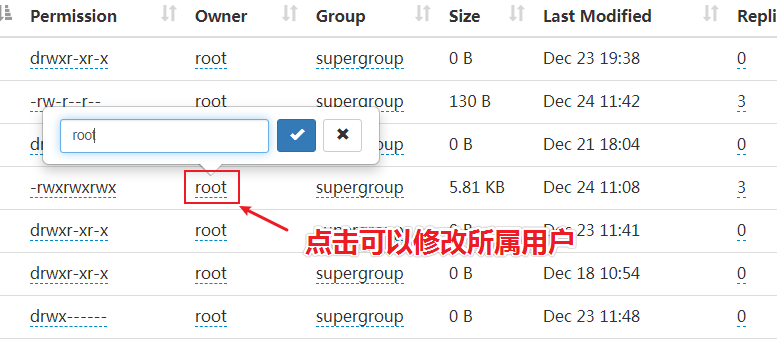
需要注意的是，使用这个命令的用户必须是超级用户，或者是该文件的属主，同时也是该用户组的成员。

#### Web页面修改UGO权限

Hadoop3.0之后，支持在HDFS **Web页面上使用鼠标修改**。



粘滞位（Sticky bit）用法在目录上设置，如此以来，只有目录内文件的所有者或者root才可以删除或移动该文件。如果不为目录设置粘滞位，任何具有该目录写和执行权限的用户都可以删除和移动其中的文件。实际应用中，粘滞位一般用于/tmp目录，以防止普通用户删除或移动其他用户的文件。



## HDFS动态节点管理

### 背景

已有HDFS集群容量已经不能满足存储数据的需求，需要在原有集群基础上动态添加新的DataNode节点。就是俗称的**动态扩容**。

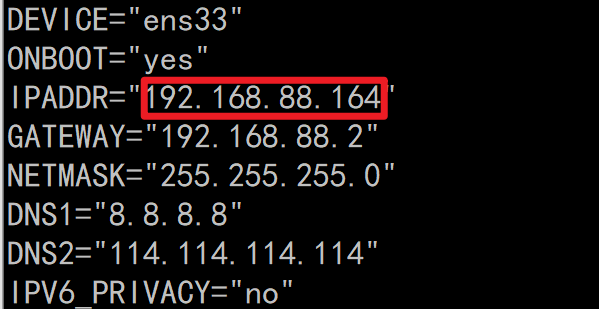
旧的服务器需要进行退役更换，暂停服务，需要在当下的集群中停止某些机器上HDFS的服务，俗称**动态缩容**。

### 动态扩容、节点上线

#### 新机器基础环境准备

##### 主机名、IP

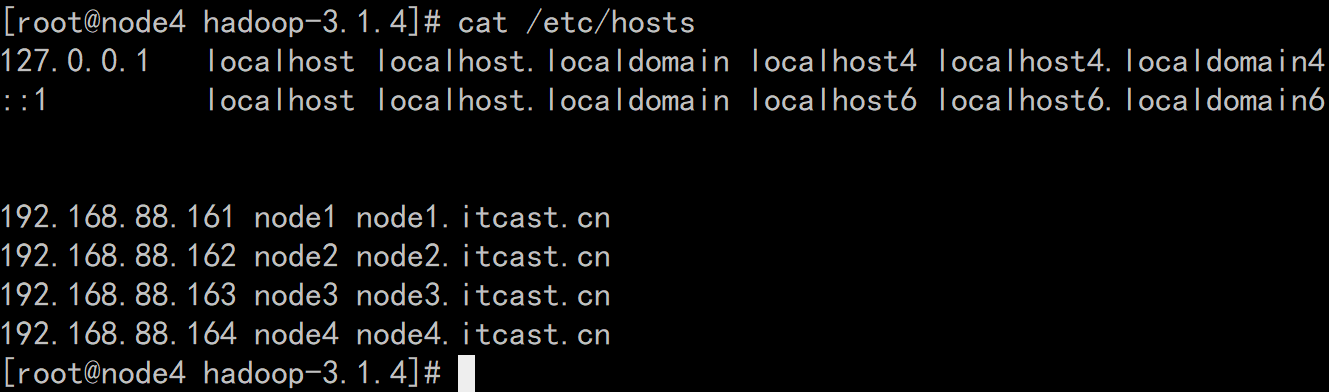
确保新机器**IP**和已有HDFS集群所属同一网段



新机器系统**hostname**



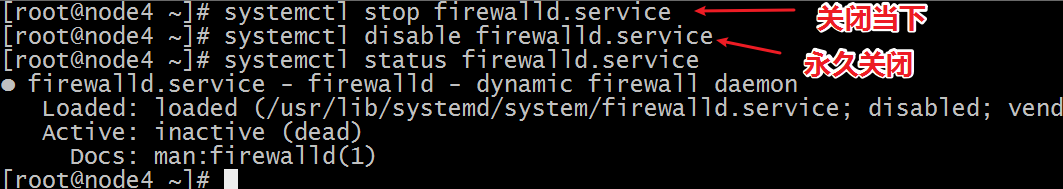
##### Hosts映射



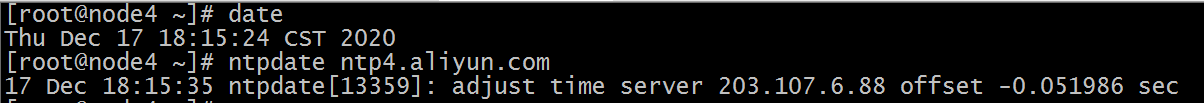
集群所有节点保持hosts文件统一

##### 防火墙、时间同步

关闭防火墙



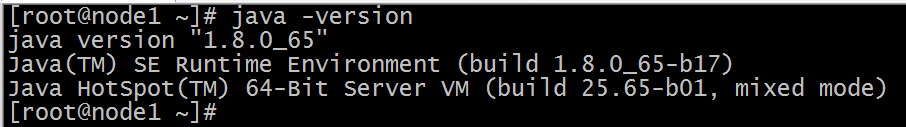
集群时间同步



##### SSH免密登录

为了后续脚本一键启动关闭集群方便，设置NameNode到新机器的免密登录

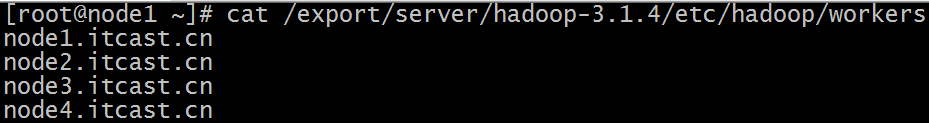
##### JDK环境配置



#### Hadoop配置

##### NameNode节点配置

修改namenode节点workers配置文件，增加新节点主机名，便于后续一键启停。



##### 新机器配置

从namenode节点复制hadoop安装包到新节点，**注意不包括hadoop.tmp.dir指定的数据存储目录**。



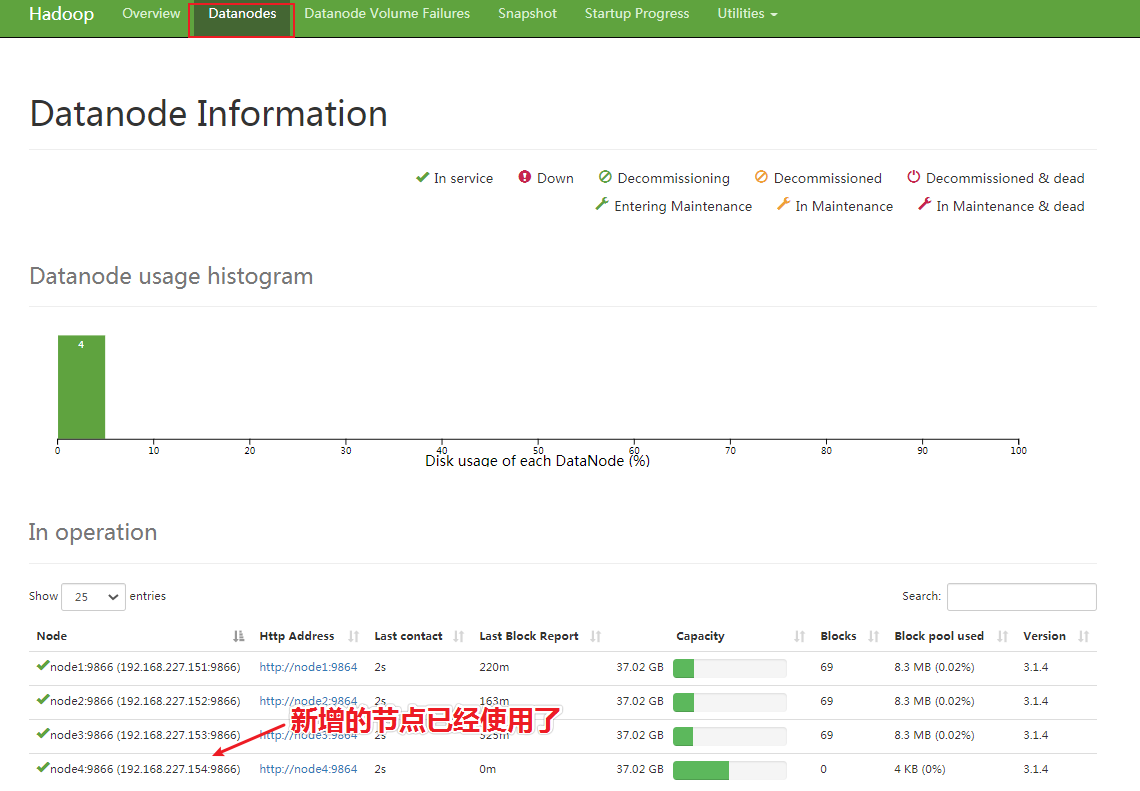
新机器上配置hadoop环境变量

|  |
| --- |
| vim /etc/profile  export HADOOP\_HOME=/export/server/hadoop-3.3.0  export PATH=$PATH:$HADOOP\_HOME/bin:$HADOOP\_HOME/sbin  source /etc/profile |

#### 手动启动DataNode进程



#### Hadoop Web页面查看



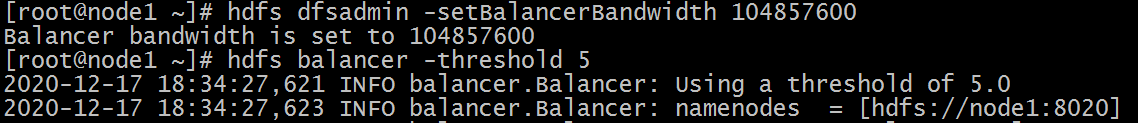
#### DataNode负载均衡服务

新加入的节点，没有数据块的存储，使得集群整体来看负载不均衡。因此最后还需要对hdfs负载设置均衡。首先设置数据传输带宽。

hdfs dfsadmin -setBalancerBandwidth 104857600

然后启动Balancer，等待集群自均衡完成即可。

hdfs balancer -threshold 5



### 动态缩容、节点下线

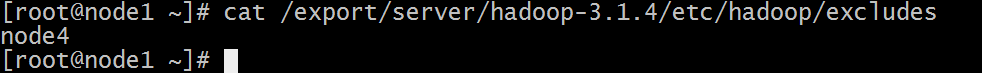
#### 添加退役节点

在namenode机器的hdfs-site.xml配置文件中需要提前配置dfs.hosts.exclude属性，该属性指向的文件就是所谓的黑名单列表，会被namenode排除在集群之外。如果文件内容为空，则意味着不禁止任何机器。

提前配置好的目的是让namenode启动的时候就能加载到该属性，只不过还没有指定任何机器。否则就需要重启namenode才能加载，因此这样的操作我们称之为**具有前瞻性的操作**。

|  |
| --- |
| <property>          <name>dfs.hosts.exclude</name>          <value>/export/server/hadoop-3.3.0/etc/hadoop/excludes</value>  </property> |

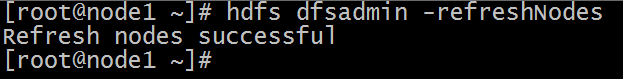
编辑dfs.hosts.exclude属性指向的excludes文件，添加需要退役的主机名称。



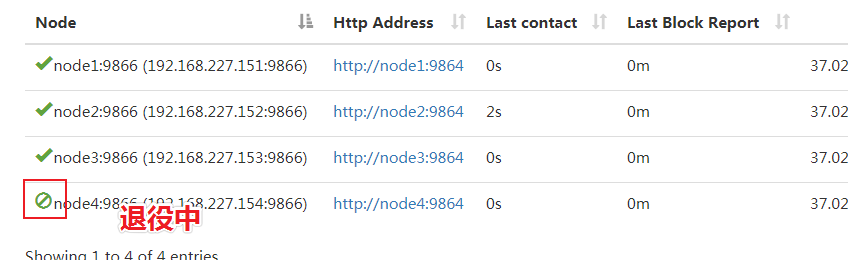
注意：**如果副本数是3，服役的节点小于等于3，是不能退役成功的，需要修改副本数后才能退役。**

#### 刷新集群

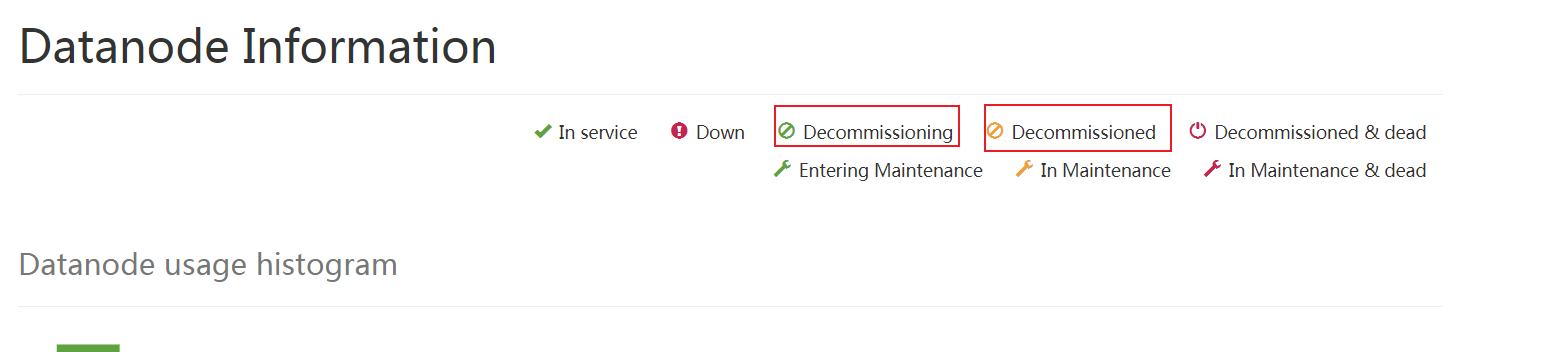
在namenode所在的机器刷新节点：hdfs dfsadmin -refreshNodes



等待退役节点状态为decommissioned（所有块已经复制完成）







#### 手动关闭DataNode进程

hdfs --daemon stop datanode



#### DataNode负载均衡服务

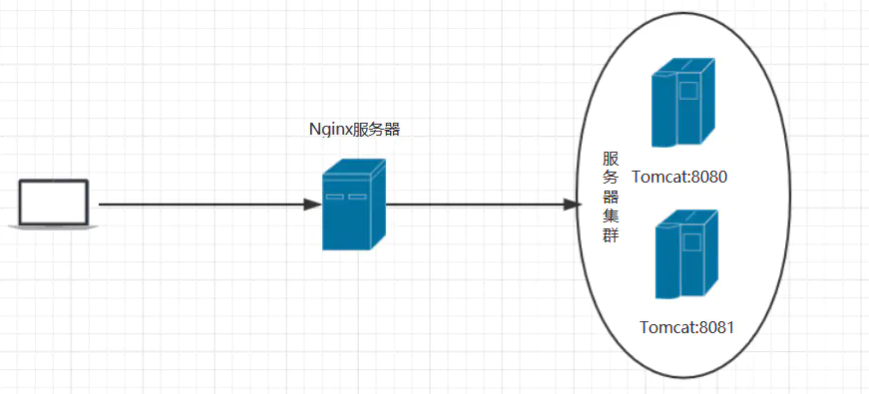
如果需要可以对已有的HDFS集群进行负载均衡服务。

hdfs balancer -threshold 5

## HDFS High Availability（HA）高可用

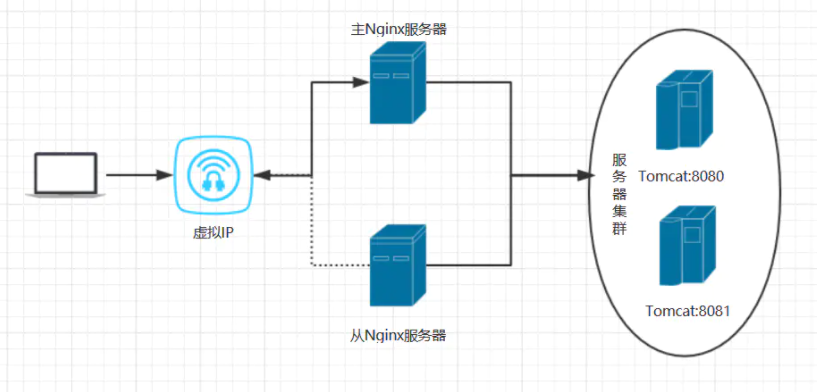
### High Availability背景知识

#### 单点故障、高可用

**单点故障**（英语：single point of failure，缩写**SPOF**）是指系统中某一点一旦失效，就会让整个系统无法运作，换句话说，单点故障即会整体故障。

**高可用性**（英语：high availability，缩写为 **HA**），IT术语，指系统无中断地执行其功能的能力，代表系统的可用性程度。是进行系统设计时的准则之一。高可用性系统意味着系统服务可以更长时间运行，通常通过提高系统的容错能力来实现。

高可用性或者高可靠度的系统不会希望有单点故障造成整体故障的情形。一般可以透过冗余的方式增加多个相同机能的部件，只要这些部件没有同时失效，系统（或至少部分系统）仍可运作，这会让可靠度提高。



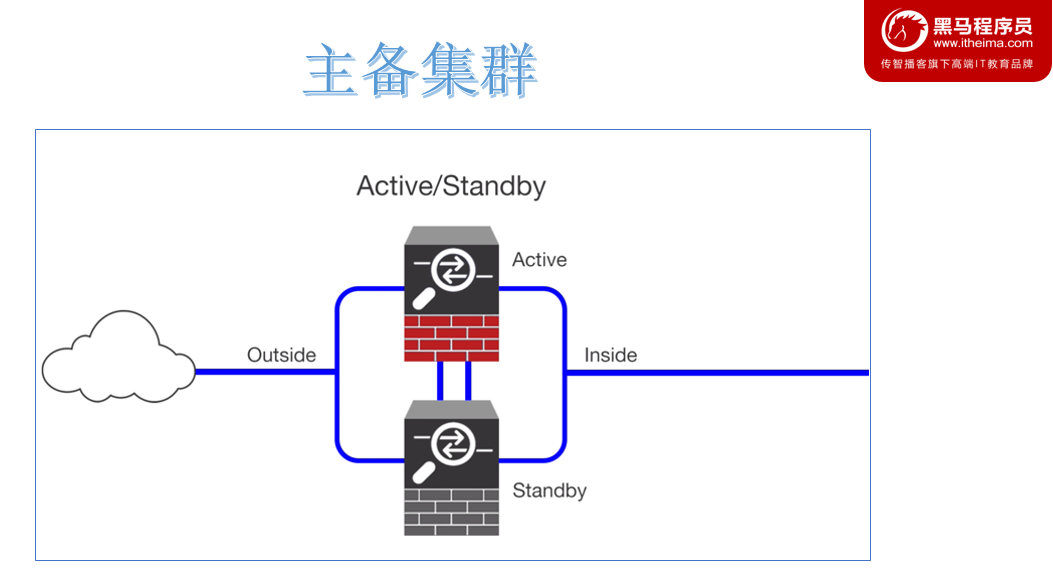
#### 高可用如何实现

##### 主备集群

解决单点故障，实现系统服务高可用的核心并不是让故障永不发生，而是让故障的发生对业务的影响降到最小。因为软硬件故障是难以避免的问题。

当下企业中成熟的做法就是给单点故障的位置设置备份，形成主备架构。通俗描述就是**当主挂掉，备份顶上**，短暂的中断之后继续提供服务。

常见的是**一主一备**架构，当然也可以一主多备。备份越多，容错能力越强，与此同时，冗余也越大，浪费资源。



##### Active、Standby

**Active**：主角色。活跃的角色，代表正在对外提供服务的角色服务。任意时间有且只有一个active对外提供服务。

**Standby**：备份角色。需要和主角色保持数据、状态同步，并且时刻准备切换成主角色（当主角色挂掉或者出现故障时），对外提供服务，保持服务的可用性。

#### 可用性评判标准—x个9

在系统的高可用性里有个衡量其可靠性的标准——**X个9**，这个X是代表数字3-5。X个9表示在系统1年时间的使用过程中，系统可以正常使用时间与总时间（1年）之比。

* **3个9**：(1-99.9%)\*365\*24=8.76小时，表示该系统在连续运行1年时间里最多可能的业务中断时间是8.76小时。
* **4个9**：(1-99.99%)\*365\*24=0.876小时=52.6分钟，表示该系统在连续运行1年时间里最多可能的业务中断时间是52.6分钟。
* **5个9**：(1-99.999%)\*365\*24\*60=5.26分钟，表示该系统在连续运行1年时间里最多可能的业务中断时间是5.26分钟。

可以看出，9越多，系统的可靠性越强，能够容忍的业务中断时间越少，但是要付出的成本更高。



#### HA系统设计核心问题

##### 脑裂问题

**脑裂**(split-brain)是指“大脑分裂”,本是医学名词。在HA集群中，脑裂指的是当联系主备节点的"心跳线"断开时(即两个节点断开联系时)，本来为一个整体、动作协调的HA系统，就分裂成为两个独立的节点。由于相互失去了联系，主备节点之间像"裂脑人"一样，使得整个集群处于混乱状态。脑裂的严重后果：

1）**集群无主**：都认为对方是状态好的，自己是备份角色，后果是无服务；

2）**集群多主**：都认为对方是故障的，自己是主角色。相互争抢共享资源，结果会导致系统混乱，数据损坏。此外对于客户端访问也是一头雾水，找谁呢？

避免脑裂问题的核心是：保持任意时刻系统有且只有一个主角色提供服务。

##### 数据同步问题

主备切换保证服务持续可用性的前提是主备节点之间的状态、数据是一致的，或者说准一致的。如果说备用的节点和主节点之间的数据差距过大，即使完成了主备切换的动作，那也是没有意义的。

数据同步常见做法是：**通过日志重演操作记录**。主角色正常提供服务，发生的事务性操作通过日志记录，备用角色读取日志重演操作。

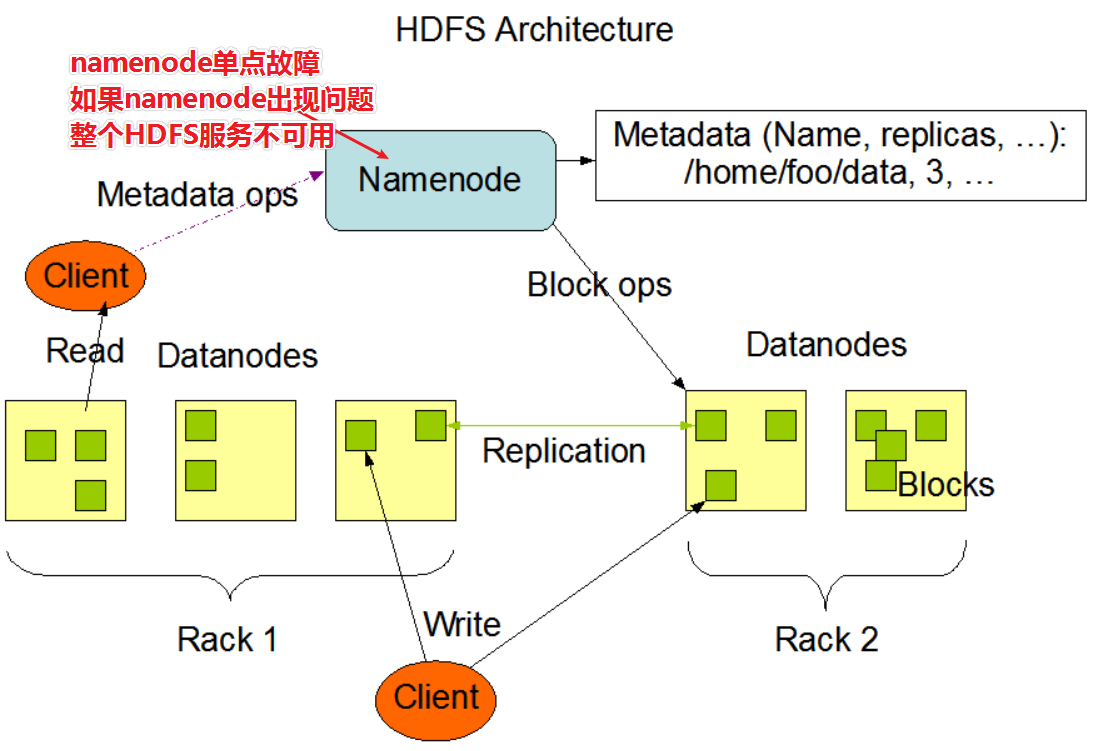
### HDFS NAMENODE单点故障问题

在Hadoop 2.0.0之前，**NameNode是HDFS集群中的单点故障（SPOF）**。每个群集只有一个NameNode，如果该计算机或进程不可用，则整个群集在整个NameNode重新启动或在另一台计算机上启动之前将不可用。

NameNode的单点故障从两个方面影响了HDFS群集的总可用性：

* 如果发生意外事件（例如机器崩溃），则在重新启动NameNode之前，群集将不可用。
* 计划内的维护事件，例如NameNode计算机上的软件或硬件升级，将导致群集停机时间的延长。

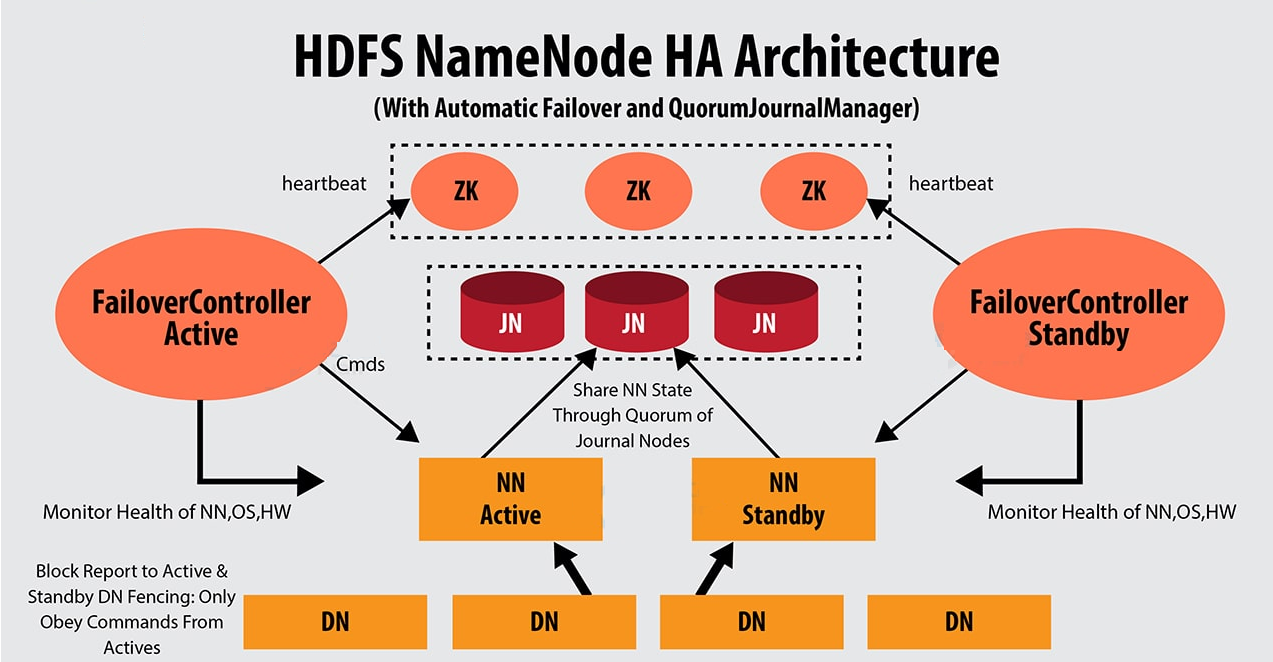
HDFS高可用性解决方案：在同一群集中运行两个（从3.0.0起，超过两个）冗余NameNode。这样可以在机器崩溃的情况下快速故障转移到新的NameNode，或者出于计划维护的目的由管理员发起的正常故障转移。



### HDFS HA解决方案—QJM

QJM全称**Quorum Journal Manager**，由cloudera公司提出，是Hadoop官方推荐的HDFS HA解决方案之一。

QJM中，使用zookeeper中ZKFC来实现主备切换；使用Journal Node（JN）集群实现edits log的共享以达到数据同步的目的。



#### QJM—主备切换、脑裂问题解决

##### ZKFailoverController（zkfc）

Apache ZooKeeper是一款高可用分布式协调服务软件，用于维护少量的协调数据。 Zookeeper的下列特性功能参与了HDFS的HA解决方案中：

* 临时znode

如果一个znode节点是临时的，那么该znode的生命周期将和创建它的客户端的session绑定。客户端断开连接session结束，znode将会被自动删除。

* Path路径唯一性

zookeeper中维持了一份类似目录树的数据结构。每个节点称之为Znode。Znode具有唯一性，不会重名。也可以理解为排他性。

* 监听机制

客户端可以针对znode上发生的事件设置监听，当事件发生触发条件，zk服务会把事件通知给设置监听的客户端。

**ZKFailoverController**（ZKFC）是一个新组件，它是一个ZooKeeper客户端。运行NameNode的每台计算机也都运行ZKFC，ZKFC的主要职责：

* 监视和管理NameNode健康状态

ZKFC通过命令定期ping本地负责监视的NameNode节点。

* 维持和ZooKeeper集群联系

如果本地NameNode运行状况良好，并且ZKFC看到当前没有其他节点持有锁znode，它将自己尝试获取该锁。如果成功，则表明它“赢得了选举”，并负责运行故障转移以使其本地NameNode处于Active状态。如果已经有其他节点持有锁，zkfc选举失败，则会对该节点注册监听，等待下次继续选举。

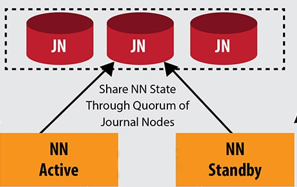
##### Fencing隔离机制

故障转移过程也就是俗称的主备角色切换的过程，切换过程中最怕的就是脑裂的发送。因此需要**Fencing机制**来避免，将先前的Active节点隔离，然后将本地NameNode转换为Active状态。

Hadoop公共库中对外提供了两种fenching实现，分别是sshfence和shellfence（缺省实现），其中sshfence是指通过ssh登陆目标节点上，使用命令fuser将进程杀死（通过tcp端口号定位进程pid，该方法比jps命令更准确），shellfence是指执行一个用户事先定义的shell命令（脚本）完成隔离。

#### QJM—主备数据同步问题解决

**Journal Node（JN）集群**是轻量级分布式系统，主要用于高速读写数据、存储数据。通常使用**2N+1**台JournalNode存储共享Edits Log（编辑日志）。



任何修改操作在 Active NN上执行时，JournalNode进程同时也会记录edits log到**至少半数**以上的JN中，这时 Standby NN 监测到JN 里面的同步log发生变化了会读取JN里面的edits log，然后重演操作记录同步到自己的目录镜像树里面，

当发生故障Active NN挂掉后，Standby NN 会在它成为Active NN 前，读取所有的JN里面的修改日志，这样就能高可靠的保证与挂掉的NN的目录镜像树一致，然后无缝的接替它的职责，维护来自客户端请求，从而达到一个高可用的目的。

### HDFS HA环境搭建

HA集群搭建的难度主要在于配置文件的编写，**心细，心细，心细**！

#### 集群基础环境准备

1.修改Linux主机名 /etc/hostname

2.修改IP /etc/sysconfig/network-scripts/ifcfg-ens33

3.修改主机名和IP的映射关系 /etc/hosts

4.关闭防火墙

5.ssh免登陆

6.安装JDK，配置环境变量等 /etc/profile

7.集群时间同步

8.配置主备NN之间的互相免密登录

#### HA集群规划

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **node1** | namenode | resourcemanager | zkfc | nodemanager | datanode | zookeeper | journal node |
| **node2** | namenode | resourcemanager | zkfc | nodemanager | datanode | zookeeper | journal node |
| **node3** |  |  |  | nodemanager | datanode | zookeeper | journal node |

#### 上传解压Hadoop安装包

hadoop-3.3.0-bin-snappy-CentOS7.tar.gz

tar zxvf hadoop-3.3.0-bin-snappy-CentOS7.tar.gz -C /export/server/

#### 配置Hadoop环境变量

export HADOOP\_HOME=/export/server/hadoop-3.3.0

export PATH=$PATH:$HADOOP\_HOME/bin:$HADOOP\_HOME/sbin

#### 修改Hadoop配置文件

##### hadoop-env.sh

|  |
| --- |
| cd /export/server/hadoop-3.3.0/etc/Hadoop  vim hadoop-env.sh  export JAVA\_HOME=/export/server/jdk1.8.0\_65  export HDFS\_NAMENODE\_USER=root  export HDFS\_DATANODE\_USER=root  export HDFS\_JOURNALNODE\_USER=root  export HDFS\_ZKFC\_USER=root |

##### core-site.xml

|  |
| --- |
| vim core-site.xml  <configuration>  <!-- HA集群名称，该值要和hdfs-site.xml中的配置保持一致 -->  <property>  <name>fs.defaultFS</name>  <value>hdfs://mycluster</value>  </property>  <!-- hadoop本地磁盘存放数据的公共目录 -->  <property>  <name>hadoop.tmp.dir</name>  <value>/export/data/ha-hadoop</value>  </property>  <!-- ZooKeeper集群的地址和端口-->  <property>  <name>ha.zookeeper.quorum</name>  <value>node1:2181,node2:2181,node3:2181</value>  </property>  </configuration> |

##### hdfs-site.xml

|  |
| --- |
| <configuration>  <!--指定hdfs的nameservice为mycluster，需要和core-site.xml中的保持一致 -->  <property>  <name>dfs.nameservices</name>  <value>mycluster</value>  </property>    <!-- mycluster下面有两个NameNode，分别是nn1，nn2 -->  <property>  <name>dfs.ha.namenodes.mycluster</name>  <value>nn1,nn2</value>  </property>  <!-- nn1的RPC通信地址 -->  <property>  <name>dfs.namenode.rpc-address.mycluster.nn1</name>  <value>node1:8020</value>  </property>  <!-- nn1的http通信地址 -->  <property>  <name>dfs.namenode.http-address.mycluster.nn1</name>  <value>node1:9870</value>  </property>  <!-- nn2的RPC通信地址 -->  <property>  <name>dfs.namenode.rpc-address.mycluster.nn2</name>  <value>node2:8020</value>  </property>    <!-- nn2的http通信地址 -->  <property>  <name>dfs.namenode.http-address.mycluster.nn2</name>  <value>node2:9870</value>  </property>  <!-- 指定NameNode的edits元数据在JournalNode上的存放位置 -->  <property>  <name>dfs.namenode.shared.edits.dir</name>  <value>qjournal://node1:8485;node2:8485;node3:8485/mycluster</value>  </property>    <!-- 指定JournalNode在本地磁盘存放数据的位置 -->  <property>  <name>dfs.journalnode.edits.dir</name>  <value>/export/data/journaldata</value>  </property>  <!-- 开启NameNode失败自动切换 -->  <property>  <name>dfs.ha.automatic-failover.enabled</name>  <value>true</value>  </property>  <!-- 指定该集群出故障时，哪个实现类负责执行故障切换 -->  <property>  <name>dfs.client.failover.proxy.provider.mycluster</name>  <value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider</value>  </property>  <!-- 配置隔离机制方法-->  <property>  <name>dfs.ha.fencing.methods</name>  <value>sshfence</value>  </property>  <!-- 使用sshfence隔离机制时需要ssh免登陆 -->  <property>  <name>dfs.ha.fencing.ssh.private-key-files</name>  <value>/root/.ssh/id\_rsa</value>  </property>    <!-- 配置sshfence隔离机制超时时间 -->  <property>  <name>dfs.ha.fencing.ssh.connect-timeout</name>  <value>30000</value>  </property>  </configuration> |

##### workers

|  |
| --- |
| node1  node2  node3 |

#### 集群同步安装包

在node1上：

|  |
| --- |
| cd /export/server  scp -r hadoop-3.3.0 root@node2:$PWD  scp -r hadoop-3.3.0 root@node3:$PWD |

#### HA集群初始化

##### 启动zk集群

/export/server/zookeeper-3.4.6/bin/zkServer.sh start

##### 手动启动JN集群

hdfs --daemon start journalnode

##### Format namenode

在node1执行格式化namenode

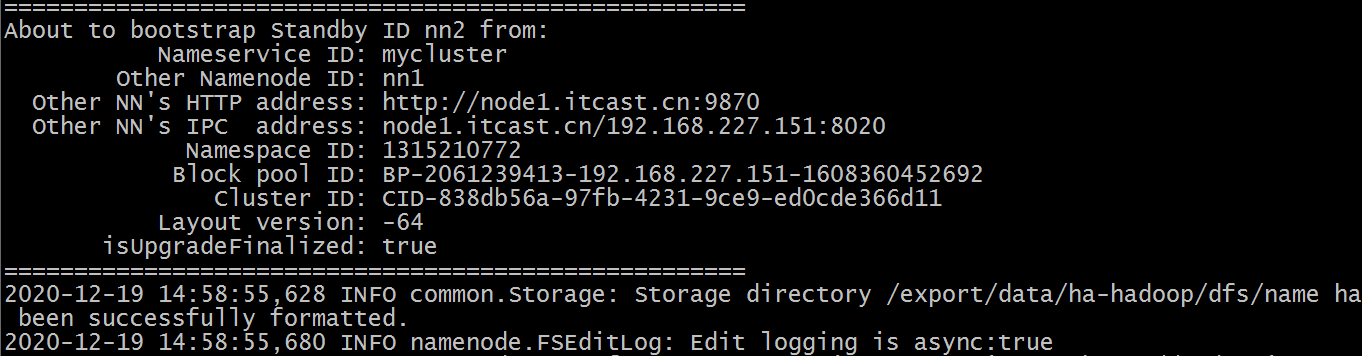
hdfs namenode –format

在node1启动namenode进程

hdfs --daemon start namenode

在node2上进行namenode元数据同步

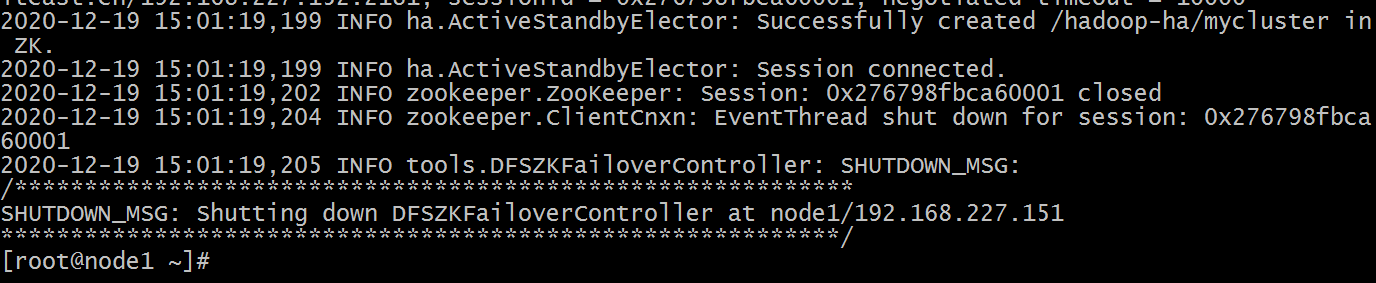
hdfs namenode –bootstrapStandby



##### 格式化zkfc

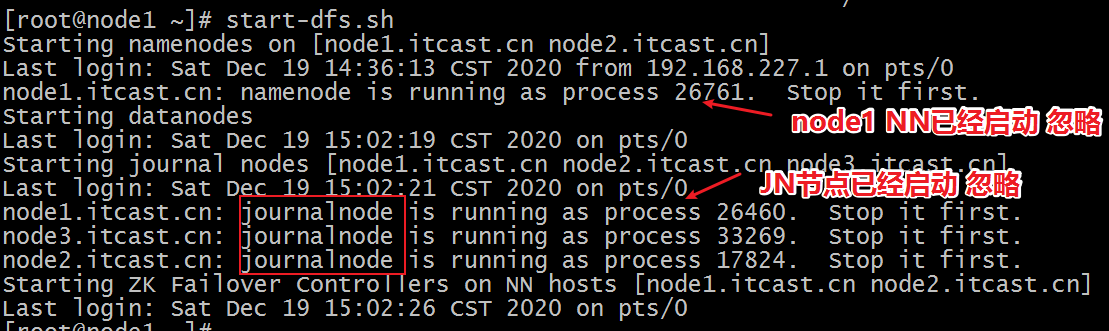
注意，在哪台机器上执行，哪台机器就将成为第一次的Active NN

hdfs zkfc –formatZK



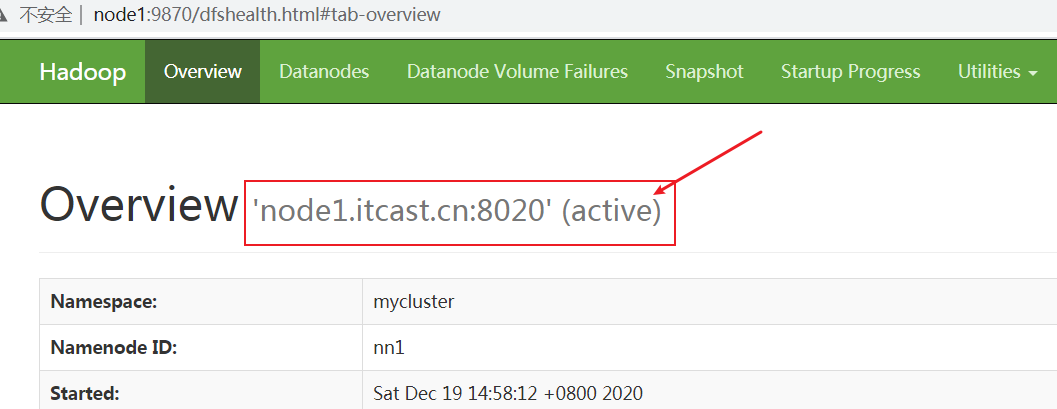
#### HA集群启动

在node1上启动HDFS集群

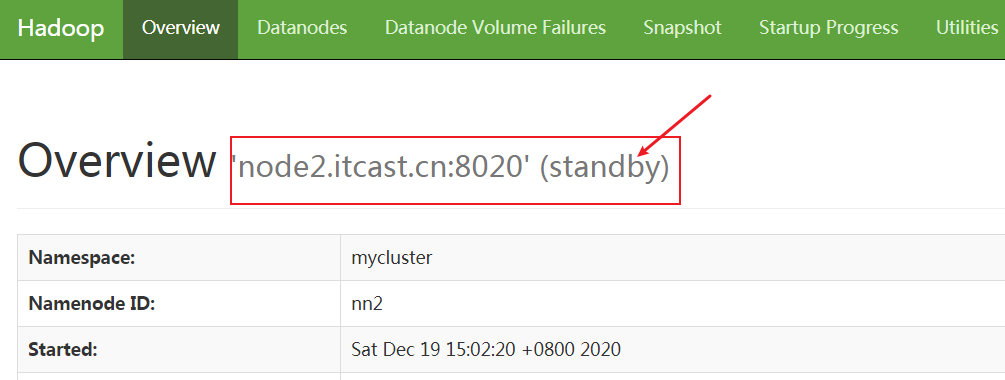


### HDFS HA效果演示

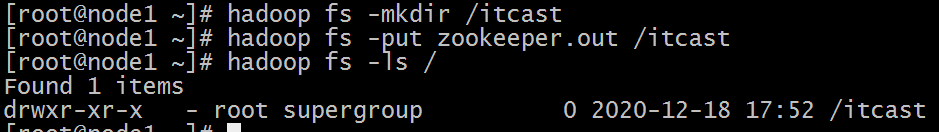
在node1上，显示namenode是active状态

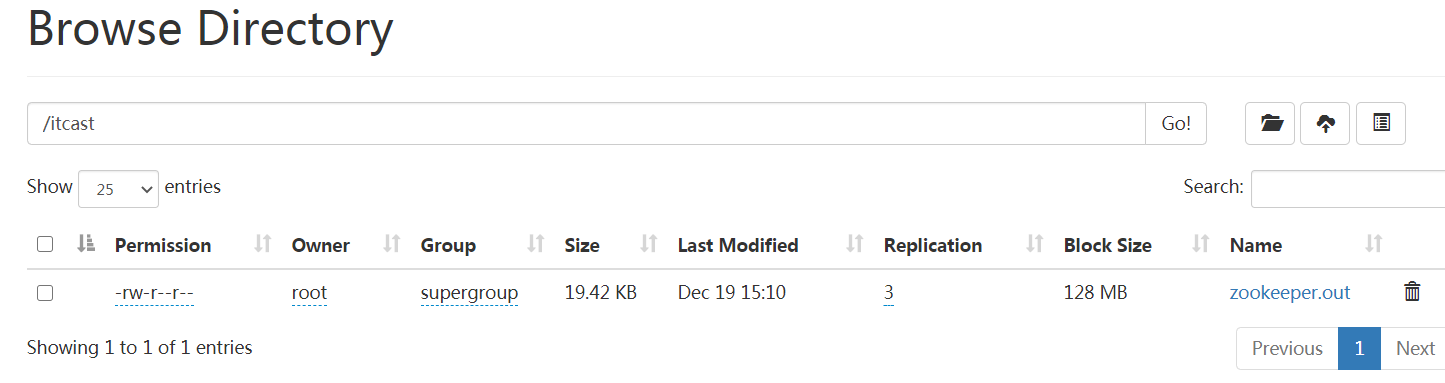


在node2上，显示namenode是standby状态

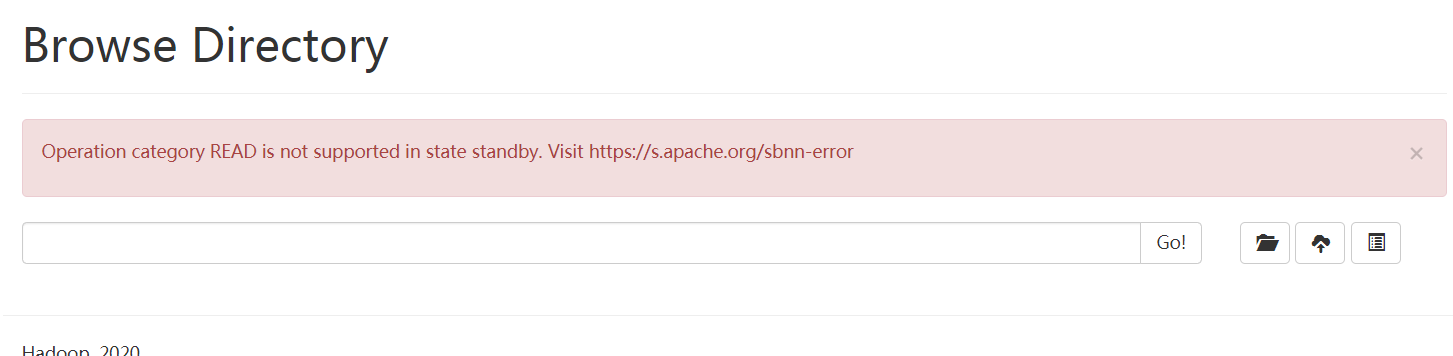


##### 正常操作



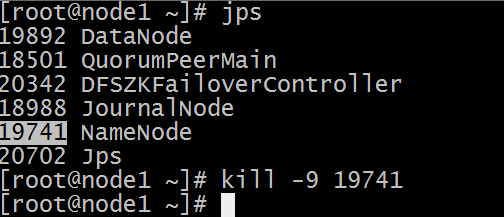


Node2无法浏览，想一想为什么？



##### 模拟故障出现

在node1，手动kill杀死namenode进程。

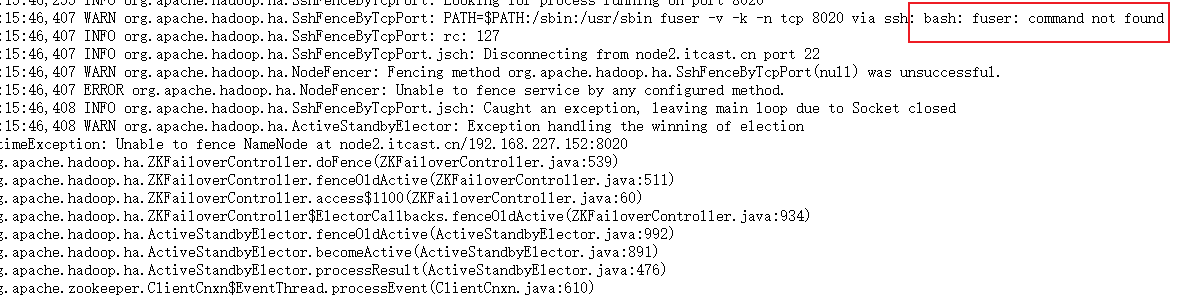


此时发现node2上的namenode切换成为Active状态 hdfs服务正常可用。

##### HA切换失败—错误解决

使用kill -9模拟JVM崩溃。或者重新启动计算机电源或拔出其网络接口以模拟另一种故障。另一个NameNode应在几秒钟内自动变为活动状态。检测故障并触发故障转移所需的时间取决于ha.zookeeper.session-timeout.ms的配置，但默认值为5秒。

如果测试不成功，检查**zkfc守护程序以及NameNode守护程序的日志**，以便进一步诊断问题。如果错误信息如下：



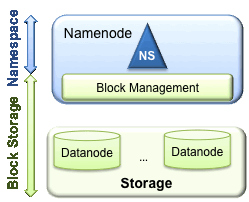
提示未找到fuser程序，导致无法进行fence，所以可以通过如下命令来安装，Psmisc软件包中包含了fuser程序（两个NN机器上都需要进行安装）

yum install psmisc -y

## HDFS Federation联邦机制

### 当前HDFS体系架构

#### 简介



当前的HDFS架构有两个主要的层：

* **命名空间（namespace）**

HDFS体系结构中的命名空间层由文件，块和目录组成。该层支持与名称空间相关的文件系统操作，例如创建，删除，修改和列出文件和目录。

* **块存储层（Block Storage）**

块存储层包括两个部分：

块管理： NameNode执行块管理。块管理通过处理注册和定期心跳来提供DataNode群集成员身份。它处理块报告并支持与块相关的操作，如创建，删除，修改或获取块位置。它还维护块的位置，副本位置。为未复制的块管理块复制，并在已复制的块中删除。

存储： DataNode通过在本地文件系统上存储块并提供读/写访问权限来管理存储空间。

#### 局限性

当下的HDFS体系结构仅允许单个NameNode维护文件系统名称空间。注意HA体系中虽然说允许多个NameNode，但是他们所维护的是同一套文件系统名称空间。这种体系目前存在着一些弊端和局限性：

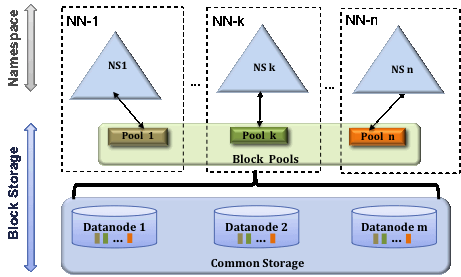
* DataNode磁盘存储空间不够增加节点，NameNode内存不够是否可以无限扩容。一种是DataNode横向扩展机器增加节点，一种是纵向扩展单机加内存。
* 由于名称空间和存储层的紧密耦合，NameNode的替代实现很困难。这限制了其他服务直接使用块存储。唯一的NameNode成了唯一入口。
* 文件系统的操作还限于NameNode一次处理的任务数。因此，群集的性能取决于NameNode吞吐量。
* 同样，由于使用单个名称空间，因此使用群集的占用者组织之间没有隔离。

### HDFS Federation架构

#### 简介

**Federation**中文意思为联邦,联盟，是NameNode之间的Federation,也就是集群中会有多个NameNode。多个NameNode的情况意味着有多个namespace。注意，这区别于HA模式下的多NameNode，HA中它们是拥有着同一个namespace。

Federation体系中多个namenode之间相互独立且不需要互相协调，各自分工，管理自己的区域。每个DataNode要向集群中所有的namenode注册，且周期性地向所有namenode发送心跳和块报告，并执行来自所有namenode的命令。



上图中，有多个NameNode，分别表示为NN1，NN2，.. NNn。NS1，NS2等是由它们各自的NameNode管理的多个名称空间。

每个名称空间都有其自己的块池（block pool）（NS1具有Pool1，NS2具有Pool2，依此类推）。每个DataNode存储集群中所有块池的块。

HDFS Federation体系结构中的**块池是属于单个名称空间的块的集合**。每个块池彼此独立地进行管理。在删除NameNode或名称空间时，DataNode中存在的相应块池也将被删除。在升级群集时，每个名称空间卷都作为一个单元进行升级。

#### 好处

* 命名空间可伸缩性

使用Federation，可以水平扩展名称空间。这对大型群集或包含太多小文件的群集有利，因为向群集添加了更多的NameNode。

* 性能

由于文件系统操作不受单个NameNode吞吐量的限制，因此可以提高文件系统的性能。

* 隔离

由于有多个名称空间，它可以为使用群集的占用者组织提供隔离。