

# Hadoop 集群（第 8 期）

## ——HDFS 初探之旅

### 1、HDFS简介

HDFS（Hadoop Distributed File System）是 Hadoop 项目的核心子项目，是分布式计算中数据存储管理的基础，是基于流数据模式访问和处理超大文件的需求而开发的，可以运行于廉价的商用服务器上。它所具有的高容错、高可靠性、高可扩展性、高获得性、高吞吐率等特征为海量数据提供了不怕故障的存储，为超大数据集（Large Data Set）的应用处理带来了便利。

Hadoop 整合了众多文件系统，在其中有一个综合性的文件系统抽象，它提供了文件系统实现的各类接口，HDFS 只是这个抽象文件系统的一个实例。提供了一个高层的文件系统抽象类 org.apache.hadoop.fs.FileSystem，这个抽象类展示了一个分布式文件系统，并有几个具体实现，如下表 1-1 所示。

表 1-1 Hadoop 的文件系统

文件系统	URI 方案	Java 实现 (org.apache.hadoop)	定义
Local	file	fs.LocalFileSystem	支持有客户端校验和本地文件系统。带有校验和的本地系统文件在 fs.RawLocalFileSystem 中实现。
HDFS	hdfs	hdfs.DistributionFileSystem	Hadoop 的分布式文件系统。
HFTP	hftp	hdfs.HftpFileSystem	支持通过 HTTP 方式以只读的方式访问 HDFS，distcp 经常用在不同的 HDFS 集群间复制数据。
HSFTP	hsftp	hdfs.HsftpFileSystem	支持通过 HTTPS 方式以只读的方式访问 HDFS。
HAR	har	fs.HarFileSystem	构建在 Hadoop 文件系统之上，对文件进行归档。Hadoop 归档文件主要用来减少 NameNode 的内存使用。
KFS	kfs	fs.kfs.KosmosFileSystem	Cloudstore（其前身是 Kosmos 文件系统）文件系统是类似于 HDFS 和 Google 的 GFS 文件系统，使用 C++编写。
FTP	ftp	fs.ftp.FtpFileSystem	由 FTP 服务器支持的文件系统。
S3（本地）	s3n	fs.s3native.NativeS3FileSystem	基于 Amazon S3 的文件系统。
S3（基于块）	s3	fs.s3.NativeS3FileSystem	基于 Amazon S3 的文件系统，以块格式存储解决了 S3 的 5GB 文件大小的限制。

Hadoop 提供了许多文件系统的接口，用户可以使用 URI 方案选取合适的文件系统来实现交互。

## 2、HDFS基础概念

### 2.1 数据块（block）

- HDFS(Hadoop Distributed File System)默认的最基本的存储单位是 64M 的数据块。
- 和普通文件系统相同的是，HDFS 中的文件是被分成 64M 一块的数据块存储的。
- 不同于普通文件系统的是，HDFS 中，如果一个文件小于一个数据块的大小，并不占用整个数据块存储空间。

### 2.2 NameNode和DataNode

HDFS 体系结构中有两类节点，一类是 NameNode，又叫“元数据节点”；另一类是 DataNode，又叫“数据节点”。这两类节点分别承担 Master 和 Worker 具体任务的执行节点。

#### 1) 元数据节点用来管理文件系统的命名空间

- 其将所有的文件和文件夹的元数据保存在一个文件系统树中。
- 这些信息也会在硬盘上保存成以下文件：命名空间镜像(namespace image)及修改日志(edit log)
- 其还保存了一个文件包括哪些数据块，分布在哪些数据节点上。然而这些信息并不存储在硬盘上，而是在系统启动的时候从数据节点收集而成的。

#### 2) 数据节点是文件系统中真正存储数据的地方。

- 客户端(client)或者元数据信息(namenode)可以向数据节点请求写入或者读出数据块。
- 其周期性的向元数据节点回报其存储的数据块信息。

#### 3) 从元数据节点（secondary namenode）

- 从元数据节点并不是元数据节点出现问题时候的备用节点，它和元数据节点负责不同的事情。
- 其主要功能就是周期性将元数据节点的命名空间镜像文件和修改日志合并，以防日志文件过大。这点在下面会相信叙述。
- 合并过后的命名空间镜像文件也在从元数据节点保存了一份，以防元数据节点失败的时候，可以恢复。

### 2.3 元数据节点目录结构

```
${dfs.name.dir}/current/VERSION
                        /edits
                        /fsimage
                        /fstime
```

VERSION 文件是 java properties 文件，保存了 HDFS 的版本号。

- layoutVersion 是一个负整数，保存了 HDFS 的持续化在硬盘上的数据结构的格式版本号。
- namespaceID 是文件系统的唯一标识符，是在文件系统初次格式化时生成的。
- cTime 此处为 0
- storageType 表示此文件夹中保存的是元数据节点的数据结构。

```
namespaceID=1232737062
cTime=0
storageType=NAME_NODE
layoutVersion=-18
```

## 2.4 数据节点的目录结构

```
${dfs.data.dir}/current/VERSION
    /blk_<id_1>
    /blk_<id_1>.meta
    /blk_<id_2>
    /blk_<id_2>.meta
    /...
    /blk_<id_64>
    /blk_<id_64>.meta
    /subdir0/
    /subdir1/
    /...
    /subdir63/
```

- 数据节点的 VERSION 文件格式如下：

```
namespaceID=1232737062
storageID=DS-1640411682-127.0.1.1-50010-1254997319480
cTime=0
storageType=DATA_NODE
layoutVersion=-18
```

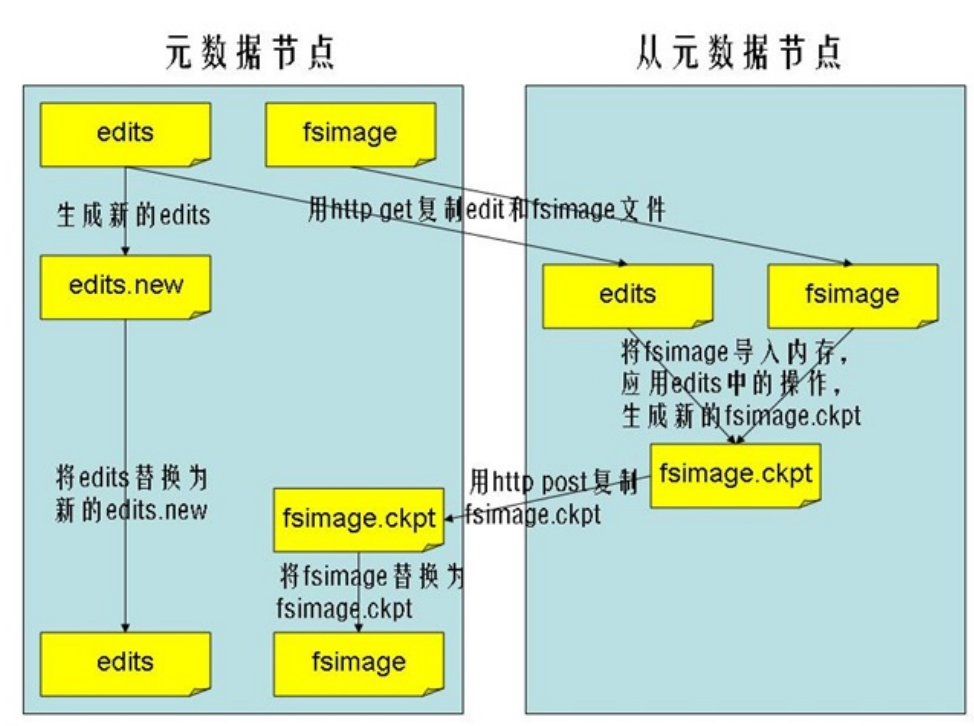
- blk\_<id> 保存的是 HDFS 的数据块，其中保存了具体的二进制数据。
- blk\_<id>.meta 保存的是数据块的属性信息：版本信息，类型信息，和 checksum
- 当一个目录中的数据块到达一定数量的时候，则创建子文件夹来保存数据块及数据块属性信息。

## 2.5 文件系统命名空间映像文件及修改日志

- 当文件系统客户端(client)进行写操作时，首先把它记录在修改日志中(edit log)
- 元数据节点在内存中保存了文件系统的元数据信息。在记录了修改日志后，元数据节点

则修改内存中的数据结构。

- 每次的写操作成功之前，修改日志都会同步(sync)到文件系统。
- fsimage 文件，也即命名空间映像文件，是内存中的元数据在硬盘上的 checkpoint，它是一种序列化的格式，并不能够在硬盘上直接修改。
- 同数据的机制相似，当元数据节点失败时，则最新 checkpoint 的元数据信息从 fsimage 加载到内存中，然后逐一重新执行修改日志中的操作。
- 从元数据节点就是用来帮助元数据节点将内存中的元数据信息 checkpoint 到硬盘上的
- checkpoint 的过程如下：
  - ◆ 从元数据节点通知元数据节点生成新的日志文件，以后的日志都写到新的日志文件中。
  - ◆ 从元数据节点用 http get 从元数据节点获得 fsimage 文件及旧的日志文件。
  - ◆ 从元数据节点将 fsimage 文件加载到内存中，并执行日志文件中的操作，然后生成新的 fsimage 文件。
  - ◆ 从元数据节点将新的 fsimage 文件用 http post 传回元数据节点
  - ◆ 元数据节点可以将旧的 fsimage 文件及旧的日志文件，换为新的 fsimage 文件和新的日志文件(第一步生成的)，然后更新 fstime 文件，写入此次 checkpoint 的时间。
  - ◆ 这样元数据节点中的 fsimage 文件保存了最新的 checkpoint 的元数据信息，日志文件也重新开始，不会变的很大了。



### 3、HDFS体系结构

HDFS 是一个主/从 (Master/Slave) 体系结构，从最终用户的角度来看，它就像传统的文件系统一样，可以通过目录路径对文件执行 CRUD (Create、Read、Update 和 Delete) 操作。但由于分布式存储的性质，HDFS 集群拥有一个 NameNode 和一些 DataNode。NameNode

管理文件系统的元数据，DataNode 存储实际的数据。客户端通过同 NameNode 和 DataNodes 的交互访问文件系统。客户端联系 NameNode 以获取文件的元数据，而真正的文件 I/O 操作是直接和 DataNode 进行交互的。

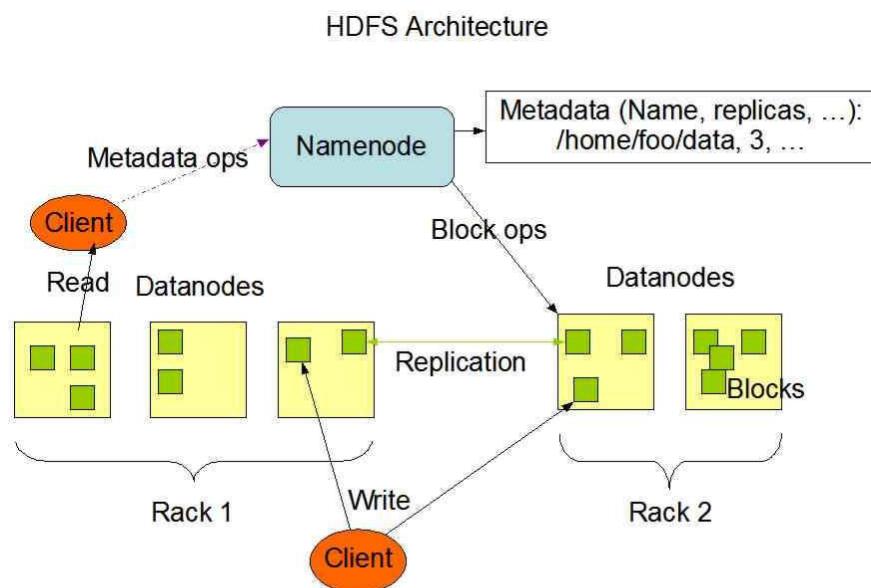


图 3.1 HDFS 总体结构示意图

### 1) NameNode、DataNode 和 Client

- NameNode 可以看作是分布式文件系统的管理者，主要负责管理文件系统的命名空间、集群配置信息和存储块的复制等。NameNode 会将文件系统的 Meta-data 存储在内存中，这些信息主要包括了文件信息、每一个文件对应的文件块的信息和每一个文件块在 DataNode 的信息等。
- DataNode 是文件存储的基本单元，它将 Block 存储在本地文件系统中，保存了 Block 的 Meta-data，同时周期性地将所有存在的 Block 信息发送给 NameNode。
- Client 就是需要获取分布式文件系统文件的应用程序。

### 2) 文件写入

- Client 向 NameNode 发起文件写入的请求。
- NameNode 根据文件大小和文件块配置情况，返回给 Client 它所管理部分 DataNode 的信息。
- Client 将文件划分为多个 Block，根据 DataNode 的地址信息，按顺序写入到每一个 DataNode 块中。

### 3) 文件读取

- Client 向 NameNode 发起文件读取的请求。
- NameNode 返回文件存储的 DataNode 的信息。
- Client 读取文件信息。

HDFS **典型的部署**是在一个专门的机器上运行 NameNode，集群中的其他机器各运行一个 DataNode；也可以在运行 NameNode 的机器上同时运行 DataNode，或者一台机器上运行多个 DataNode。一个集群只有一个 NameNode 的设计大大简化了系统架构。

## 4、HDFS的优缺点

### 4.1 HDFS的优点

#### 1) 处理超大文件

这里的超大文件通常是指百 MB、设置数百 TB 大小的文件。目前在实际应用中，HDFS 已经能用来存储管理 PB 级的数据了。

#### 2) 流式的访问数据

HDFS 的设计建立在更多地响应“一次写入、多次读写”任务的基础上。这意味着一个数据集一旦由数据源生成，就会被复制分发到不同的存储节点中，然后响应各种各样的数据分析任务请求。在多数情况下，分析任务都会涉及数据集中的大部分数据，也就是说，对 HDFS 来说，请求读取整个数据集要比读取一条记录更加高效。

#### 3) 运行于廉价的商用机器集群上

Hadoop 设计对硬件需求比较低，只须运行在低廉的商用硬件集群上，而无需昂贵的高可用性机器上。廉价的商用机也就意味着大型集群中出现节点故障情况的概率非常高。这就要求设计 HDFS 时要充分考虑数据的可靠性，安全性及高可用性。

### 4.2 HDFS的缺点

#### 1) 不适合低延迟数据访问

如果要处理一些用户要求时间比较短的低延迟应用请求，则 HDFS 不适合。HDFS 是为了处理大型数据集分析任务的，主要是为达到高的数据吞吐量而设计的，这就可能要求以高延迟作为代价。

**改进策略：**对于那些有低延时要求的应用程序，HBase 是一个更好的选择。通过上层数据管理项目来尽可能地弥补这个不足。在性能上有了很大的提升，它的口号就是 goes real time。使用缓存或多 master 设计可以降低 client 的数据请求压力，以减少延时。还有就是对 HDFS 系统内部的修改，这就得权衡大吞吐量与低延时了，HDFS 不是万能的银弹。

#### 2) 无法高效存储大量小文件

因为 Namenode 把文件系统的元数据放置在内存中，所以文件系统所能容纳的文件数目是由 Namenode 的内存大小来决定。一般来说，每一个文件、文件夹和 Block 需要占据 150 字节左右的空间，所以，如果你有 100 万个文件，每一个占据一个 Block，你就至少需要 300MB 内存。当前来说，数百万的文件还是可行的，当扩展到数十亿时，对于当前的硬件水平来说就没法实现了。还有一个问题就是，因为 Map task 的数量是由 splits 来决定的，所以用 MR 处理大量的小文件时，就会产生过多的 Maptask，线程管理开销将会增加作业时间。举个例子，处理 10000M 的文件，若每个 split 为 1M，那就会有 10000 个 Maptasks，会有很大的线程开销；若每个 split 为 100M，则只有 100 个 Maptasks，每个 Maptask 将会有更多的事情做，而线程的管理开销也将减小很多。

**改进策略：**要想让 HDFS 能处理好小文件，有不少方法。

- 利用 SequenceFile、MapFile、Har 等方式归档小文件，这个方法的原理就是把小文件归档起来管理，HBase 就是基于此的。对于这种方法，如果想找回原来的小文件内容，那就必须得知道与归档文件的映射关系。
- 横向扩展，一个 Hadoop 集群能管理的小文件有限，那就把几个 Hadoop 集群拖在



一个虚拟服务器后面，形成一个大的 Hadoop 集群。google 也是这么干过的。

- 多 Master 设计，这个作用显而易见了。正在研发中的 GFS II 也要改为分布式多 Master 设计，还支持 Master 的 Failover，而且 Block 大小改为 1M，有意要调优处理小文件啊。
- 附带个 Alibaba DFS 的设计，也是多 Master 设计，它把 Metadata 的映射存储和管理分开了，由多个 Metadata 存储节点和一个查询 Master 节点组成。

### 3) 不支持多用户写入及任意修改文件

在 HDFS 的一个文件中**只有**一个写入者，而且写操作**只能**在文件**末尾**完成，即**只能执行追加操作**。目前 HDFS 还**不支持多个用户对同一文件的写操作**，以及在文件任意位置进行修改。

## 5、HDFS常用操作

先说一下“**hadoop fs** 和 **hadoop dfs** 的区别”，看两本 Hadoop 书上各有用到，但效果一样，求证与网络发现下面一解释比较中肯。

粗略的讲，fs 是个比较抽象的层面，在分布式环境中，fs 就是 dfs，但在本地环境中，fs 是 local file system，这个时候 dfs 就不能用。

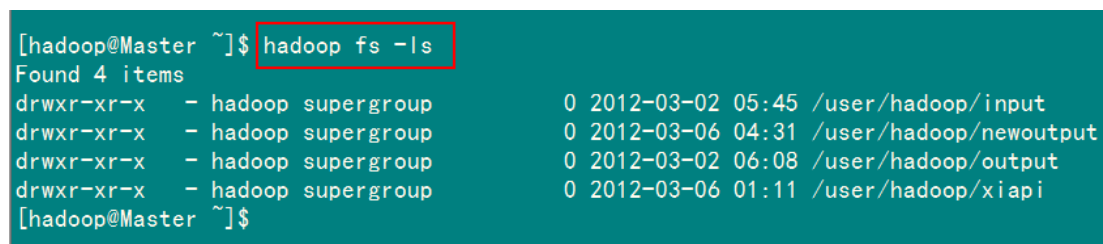
### 5.1 文件操作

#### 1) 列出 HDFS 文件

此处为你展示如何通过“-ls”命令列出 HDFS 下的文件：

```
hadoop fs -ls
```

执行结果如图 5-1-1 所示。在这里需要注意：在 HDFS 中未带参数的“-ls”命名没有返回任何值，它**默认**返回 HDFS 的“**home**”目录下的内容。在 HDFS 中，**没有当前目录**这样一个概念，也**没有 cd** 这个命令。



```
[hadoop@Master ~]$ hadoop fs -ls
Found 4 items
drwxr-xr-x - hadoop supergroup          0 2012-03-02 05:45 /user/hadoop/input
drwxr-xr-x - hadoop supergroup          0 2012-03-06 04:31 /user/hadoop/newoutput
drwxr-xr-x - hadoop supergroup          0 2012-03-02 06:08 /user/hadoop/output
drwxr-xr-x - hadoop supergroup          0 2012-03-06 01:11 /user/hadoop/xiapi
[hadoop@Master ~]$
```

图 5-1-1 列出 HDFS 文件

#### 2) 列出 HDFS 目录下某个文档中的文件

此处为你展示如何通过“-ls 文件名”命令浏览 HDFS 下名为“input”的文档中文件：

```
hadoop fs -ls input
```

执行结果如图 5-1-2 所示。

```
[hadoop@Master ~]$ hadoop fs -ls input
Found 2 items
-rw-r--r-- 1 hadoop supergroup          12 2012-03-02 05:45 /user/hadoop/input/file1.txt
-rw-r--r-- 1 hadoop supergroup          13 2012-03-02 05:45 /user/hadoop/input/file2.txt
[hadoop@Master ~]$
```

图 5-1-2 列出 HDFS 下名为 input 的文档下的文件

### 3) 上传文件到 HDFS

此处为你展示如何通过“-put 文件 1 文件 2”命令将“**Master.Hadoop**”机器下的“/home/hadoop”目录下的 **file** 文件上传到 HDFS 上并**重命名**为 **test**:

```
hadoop fs -put ~/file test
```

执行结果如图 5-1-3 所示。在执行“-put”时**只有两种**可能，即是**执行成功**和**执行失败**。在上传文件时，文件首先复制到 **DataNode** 上。**只有**所有的 DataNode 都**成功**接收完数据，文件上传**才是**成功的。其他情况（如文件上传终端等）对 HDFS 来说都是做了无用功。

```
[hadoop@Master ~]$ ll
总用量 141380
drwxrwxr-x. 2 hadoop hadoop    4096 3月 2 05:36 file
drwxr-xr-x. 16 hadoop hadoop    4096 3月 5 00:19 hadoop
-rw-r--r--. 1 hadoop hadoop 59468784 2月 27 03:31 hadoop-1.0.0.tar.gz
-rw-r--r--. 1 hadoop hadoop 85292206 2月 27 03:29 jdk-6u31-linux-i586.bin
[hadoop@Master ~]$ hadoop fs -put ~/file test
[hadoop@Master ~]$ hadoop fs -ls
Found 5 items
drwxr-xr-x - hadoop supergroup          0 2012-03-02 05:45 /user/hadoop/input
drwxr-xr-x - hadoop supergroup          0 2012-03-06 04:31 /user/hadoop/newoutput
drwxr-xr-x - hadoop supergroup          0 2012-03-02 06:08 /user/hadoop/output
drwxr-xr-x - hadoop supergroup          0 2012-03-07 18:45 /user/hadoop/test
drwxr-xr-x - hadoop supergroup          0 2012-03-06 01:11 /user/hadoop/xiapi
[hadoop@Master ~]$
```

图 5-1-3 成功上传 file 到 HDFS

### 4) 将 HDFS 中文件复制到本地系统中

此处为你展示如何通过“-get 文件 1 文件 2”命令将 HDFS 中的“output”文件复制到本地系统并命名为“getout”。

```
hadoop fs -get output getout
```

执行结果如图 5-1-4 所示。

```
[hadoop@Master ~]$ hadoop fs -get output getout
[hadoop@Master ~]$ ll | grep getout
drwxrwxr-x. 3 hadoop hadoop    4096 3月 7 18:54 getout
[hadoop@Master ~]$
```

图 5-1-4 成功将 HDFS 中 output 文件复制到本地系统



**备注：**与“-put”命令一样，“-get”操作既可以操作文件，也可以操作目录。

### 5) 删除 HDFS 下的文档

此处为你展示如何通过“-rmr 文件”命令删除 HDFS 下名为“newoutput”的文档：

```
hadoop fs -rmr newoutput
```

执行结果如图 5-1-5 所示。

```
[hadoop@Master ~]$ hadoop fs -ls
Found 5 items
drwxr-xr-x - hadoop supergroup      0 2012-03-02 05:45 /user/hadoop/input
drwxr-xr-x - hadoop supergroup      0 2012-03-06 04:31 /user/hadoop/newoutput
drwxr-xr-x - hadoop supergroup      0 2012-03-02 06:08 /user/hadoop/output
drwxr-xr-x - hadoop supergroup      0 2012-03-07 18:45 /user/hadoop/test
drwxr-xr-x - hadoop supergroup      0 2012-03-06 01:11 /user/hadoop/xiapi
[hadoop@Master ~]$ hadoop fs -rmr newoutput
Deleted hdfs://192.168.1.2:9000/user/hadoop/newoutput
[hadoop@Master ~]$ hadoop fs -ls
Found 4 items
drwxr-xr-x - hadoop supergroup      0 2012-03-02 05:45 /user/hadoop/input
drwxr-xr-x - hadoop supergroup      0 2012-03-02 06:08 /user/hadoop/output
drwxr-xr-x - hadoop supergroup      0 2012-03-07 18:45 /user/hadoop/test
drwxr-xr-x - hadoop supergroup      0 2012-03-06 01:11 /user/hadoop/xiapi
[hadoop@Master ~]$
```

图 5-1-5 成功删除 HDFS 下的 newoutput 文档

### 6) 查看 HDFS 下某个文件

此处为你展示如何通过“-cat 文件”命令查看 HDFS 下 input 文件中内容：

```
hadoop fs -cat input/*
```

执行结果如图 5-1-6 所示。

```
[hadoop@Master ~]$ hadoop fs -cat input/*
Hello World
Hello Hadoop
[hadoop@Master ~]$
```

图 5-1-6 HDFS 下 input 文件的内容

“hadoop fs”的命令远不止这些，本小节介绍的命令已可以在 HDFS 上完成大多数常规操作。对于其他操作，可以通过“-help commandName”命令所列出的清单来进一步学习与探索。

## 5.2 管理与更新

### 1) 报告 HDFS 的基本统计情况

此处为你展示通过“-report”命令如何查看 HDFS 的基本统计信息：

```
hadoop dfsadmin -report
```

执行结果如图 5-2-1 所示。

```
[hadoop@Master ~]$ hadoop dfsadmin -report
Configured Capacity: 158534062080 (147.65 GB)
Present Capacity: 144828329984 (134.88 GB)
DFS Remaining: 144828059648 (134.88 GB)
DFS Used: 270336 (264 KB)
DFS Used%: 0%
Under replicated blocks: 2
Blocks with corrupt replicas: 0
Missing blocks: 0

-----
Datanodes available: 3 (3 total, 0 dead)

Name: 192.168.1.4:50010
Decommission Status : Normal
Configured Capacity: 52844687360 (49.22 GB)
DFS Used: 81920 (80 KB)
Non DFS Used: 4576182272 (4.26 GB)
DFS Remaining: 48268423168 (44.95 GB)
DFS Used%: 0%
DFS Remaining%: 91.34%
Last contact: Wed Mar 07 22:28:44 CST 2012
```

图 5-2-1 HDFS 基本统计信息

## 2) 退出安全模式

NameNode 在启动时会自动进入安全模式。安全模式是 NameNode 的一种状态,在这个阶段,文件系统不允许有任何修改。安全模式的目的在系统启动时检查各个 DataNode 上数据块的有效性,同时根据策略对数据块进行必要的复制或删除,当数据块最小百分比数满足的最小副本数条件时,会自动退出安全模式。

系统显示 “Name node is in safe mode”, 说明系统正处于安全模式,这时只需要等待 17 秒即可,也可以通过下面的命令退出安全模式:

```
hadoop dfsadmin -safemode enter
```

成功退出安全模式结果如图 5-2-2 所示。

```
[hadoop@Master ~]$ hadoop dfsadmin -safemode leave
Safe mode is OFF
[hadoop@Master ~]$
```

图 5-2-2 成功退出安全模式

3) 进入安全模式

在必要情况下，可以通过以下命令把 HDFS 置于安全模式：

```
hadoop dfsadmin -safemode enter
```

执行结果如图 5-2-3 所示。

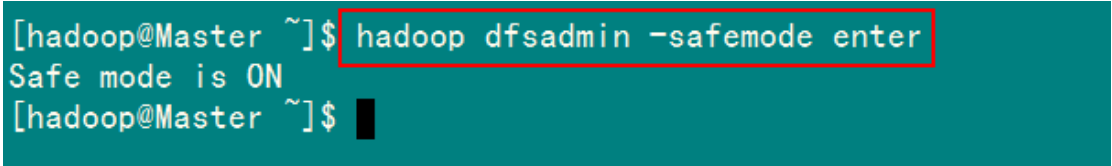


图 5-2-3 进入 HDFS 安全模式

4) 添加节点

**可扩展性**是 HDFS 的一个重要特性，向 HDFS 集群中添加节点是很容易实现的。添加一个新的 DataNode 节点，首先在新加节点上安装好 Hadoop，要和 NameNode 使用相同的配置（可以直接从 NameNode 复制），修改 “/usr/hadoop/conf/master” 文件，加入 **NameNode** 主机名。然后在 NameNode 节点上修改 “/usr/hadoop/conf/slaves” 文件，加入新节点主机名，再建立到新加节点无密码的 SSH 连接，运行启动命令：

```
start-all.sh
```

5) 负载均衡

HDFS 的数据在各个 DataNode 中的分布可能很不均匀，尤其是在 DataNode 节点**出现故障或新增** DataNode 节点时。新增数据块时 NameNode 对 DataNode 节点的**选择策略**也有可能 导致数据块分布的不均匀。用户可以使用命令重新平衡 DataNode 上的数据块的分布：

```
start-balancer.sh
```

执行命令前，DataNode 节点上数据分布情况如图 5-2-4 所示。

NameNode 'Master.Hadoop:9000'

Started: Mon Mar 05 23:03:10 CST 2012

Version: 1.0.0, r1214675

Compiled: Thu Dec 15 16:36:35 UTC 2011 by hortonfo

Upgrades: There are no upgrades in progress.

[Browse the filesystem](#)

[Namenode Logs](#)

[Go back to DFS home](#)

---

Live Datanodes : 3

Node	Last Contact	Admin State	Configured Capacity (GB)	Used (GB)	Non DFS Used (GB)	Remaining (GB)	Used (%)	Used (%)	Remaining (%)	Blocks
Slave1	0	In Service	49.22	1.78	4.26	43.17	3.61	<div></div>	87.72	36
Slave2	1	In Service	49.22	2.19	4.26	42.76	4.46	<div></div>	86.88	40
Slave3	1	In Service	49.22	2.55	4.22	42.44	5.18	<div></div>	86.24	47

This is Apache Hadoop release 1.0.0

负载均衡完毕后，DataNode 节点上数据的分布情况如图 5-2-5 所示。

NameNode 'Master.Hadoop:9000'

Started: Mon Mar 05 23:03:10 CST 2012  
Version: 1.0.0, r1214675  
Compiled: Thu Dec 15 16:36:35 UTC 2011 by hortonfo  
Upgrades: There are no upgrades in progress.

[Browse the filesystem](#)  
[Namenode Logs](#)  
[Go back to DFS home](#)

Live Datanodes : 3

Node	Last Contact	Admin State	Configured Capacity (GB)	Used (GB)	Non DFS Used (GB)	Remaining (GB)	Used (%)	Used (%)	Remaining (%)	Blocks
Slave1	0	In Service	49.22	1.78	4.26	43.17	3.61	<div></div>	87.72	38
Slave2	2	In Service	49.22	2.19	4.26	42.76	4.46	<div></div>	86.88	40
Slave3	1	In Service	49.22	2.53	4.24	42.44	5.15	<div></div>	86.24	48

This is [Apache Hadoop](#) release 1.0.0

执行负载均衡命令如图 5-2-6 所示。

```
[hadoop@Master ~]$ start-balancer.sh
starting balancer, logging to /usr/hadoop/libexec/./logs/hadoop-hadoop-balancer-Master.Hadoop.out
Time Stamp      Iteration#  Bytes Already Moved  Bytes Left To Move  Bytes Being Moved
The cluster is balanced. Exiting...
Balancing took 391.0 milliseconds
[hadoop@Master ~]$
```

6、HDFS API详解

Hadoop 中关于文件操作类基本上全部是在“[org.apache.hadoop.fs](#)”包中，这些 API 能够支持的操作包含：打开文件，读写文件，删除文件等。

Hadoop 类库中最终面向用户提供的接口类是 [FileSystem](#)，该类是个抽象类，只能通过来类的 get 方法得到具体类。get 方法存在几个重载版本，常用的是这个：

```
static FileSystem get(Configuration conf);
```

该类封装了几乎所有的文件操作，例如 mkdir，delete 等。综上基本上可以得出操作文件的程序库框架：

```
operator()
{
    得到 Configuration 对象
    得到 FileSystem 对象
    进行文件操作
}
```

## 6.1 上传本地文件

通过“`FileSystem.copyFromLocalFile(Path src, Path dst)`”可将本地文件上传到 HDFS 的制定位置上，其中 src 和 dst 均为文件的完整路径。具体事例如下：

```
package com.hebut.file;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class CopyFile {
    public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);

        //本地文件
        Path src =new Path("D:\\HebutWinOS");
        //HDFS 为止
        Path dst =new Path("/");

        hdfs.copyFromLocalFile(src, dst);
        System.out.println("Upload to"+conf.get("fs.default.name"));

        FileStatus files[]=hdfs.listStatus(dst);
        for(FileStatus file:files){
            System.out.println(file.getPath());
        }
    }
}
```

运行结果可以通过控制台、项目浏览器和 SecureCRT 查看，如图 6-1-1、图 6-1-2、图 6-1-3 所示。

### 1) 控制台结果

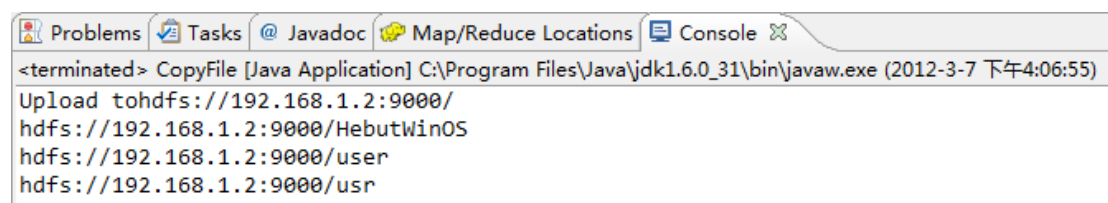


图 6-1-1 运行结果（1）

## 2) 项目浏览器

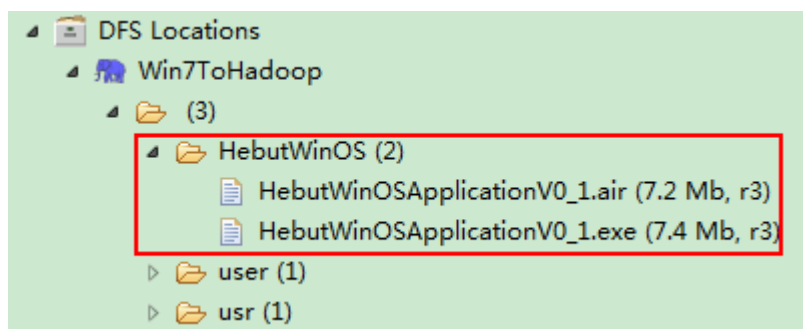


图 6-1-2 运行结果 (2)

## 3) SecureCRT 结果

```
[hadoop@Master ~]$ hadoop fs -ls /
Found 3 items
drwxr-xr-x - hadoop supergroup          0 2012-03-08 00:10 /HebutWinOS
drwxr-xr-x - hadoop supergroup          0 2012-03-05 19:29 /user
drwxr-xr-x - hadoop supergroup          0 2012-03-05 19:43 /usr
[hadoop@Master ~]$ hadoop fs -ls /HebutWinOS
Found 2 items
-rw-r--r--  3 hadoop supergroup    7541853 2012-03-08 00:10 /HebutWinOS/HebutWinOSApplicationV0_1.air
-rw-r--r--  3 hadoop supergroup    7768212 2012-03-08 00:10 /HebutWinOS/HebutWinOSApplicationV0_1.exe
[hadoop@Master ~]$
```

图 6-1-3 运行结果 (3)

# 6.2 创建HDFS文件

通过“**FileSystem.create (Path f)**”可在 HDFS 上创建文件，其中 f 为文件的完整路径。具体实现如下：

```
package com.hebut.file;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class CreateFile {

    public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);

        byte[] buff="hello hadoop world!\n".getBytes();

        Path dfs=new Path("/test");
```



```

        FSDDataOutputStream outputStream=hdfs.create(dfs);
        outputStream.write(buff,0,buff.length);

    }
}

```

运行结果如图 6-2-1 和图 6-2-2 所示。

#### 1) 项目浏览器



图 6-2-1 运行结果 (1)

#### 2) SecureCRT 结果

```

[hadoop@Master ~]$ hadoop fs -ls /
Found 4 items
drwxr-xr-x  - hadoop supergroup      0 2012-03-08 00:10 /HebutWinOS
-rw-r--r--  3 hadoop supergroup    20 2012-03-08 01:27 /test
drwxr-xr-x  - hadoop supergroup      0 2012-03-05 19:29 /user
drwxr-xr-x  - hadoop supergroup      0 2012-03-05 19:43 /usr
[hadoop@Master ~]$ hadoop fs -cat /test
hello hadoop world!
[hadoop@Master ~]$

```

图 6-2-2 运行结果 (2)

## 6.3 创建HDFS目录

通过“**FileSystem.mkdirs (Path f)**”可在 HDFS 上创建文件夹, 其中 f 为文件夹的完整路径。具体实现如下:

```

package com.hebut.dir;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class CreateDir {

```

```

public static void main(String[] args) throws Exception{
    Configuration conf=new Configuration();
    FileSystem hdfs=FileSystem.get(conf);

    Path dfs=new Path("/TestDir");

    hdfs.mkdirs(dfs);

}
}

```

运行结果如图 6-3-1 和图 6-3-2 所示。

#### 1) 项目浏览器

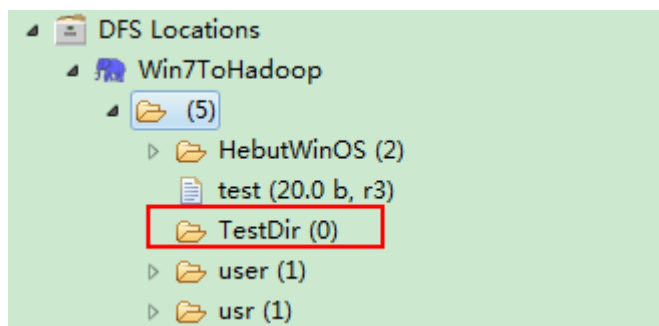


图 6-3-1 运行结果 (1)

#### 2) SecureCRT 结果

```

[hadoop@Master ~]$ hadoop fs -ls /
Found 5 items
drwxr-xr-x - hadoop supergroup 0 2012-03-08 00:10 /HebutWinOS
drwxr-xr-x - hadoop supergroup 0 2012-03-08 04:49 /TestDir
-rw-r--r-- 3 hadoop supergroup 20 2012-03-08 01:27 /test
drwxr-xr-x - hadoop supergroup 0 2012-03-05 19:29 /user
drwxr-xr-x - hadoop supergroup 0 2012-03-05 19:43 /usr
[hadoop@Master ~]$

```

图 6-3-2 运行结果 (2)

## 6.4 重命名HDFS文件

通过“**FileSystem.rename (Path src, Path dst)**”可为指定的 HDFS 文件重命名，其中 src 和 dst 均为文件的完整路径。具体实现如下：

```

package com.hebut.file;

import org.apache.hadoop.conf.Configuration;

```

```

import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class Rename{
    public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);

        Path frpaht=new Path("/test"); //旧的文件名
        Path topath=new Path("/test1"); //新的文件名

        boolean isRename=hdfs.rename(frpaht, topath);

        String result=isRename?"成功":"失败";
        System.out.println("文件重命名结果为: "+result);

    }
}

```

运行结果如图 6-4-1 和图 6-4-2 所示。

#### 1) 项目浏览器

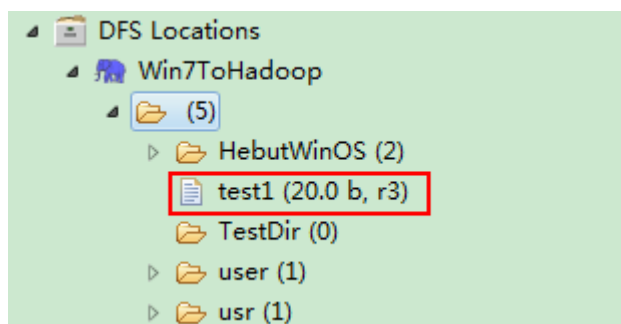


图 6-4-1 运行结果 (1)

#### 2) SecureCRT 结果

```

[hadoop@Master ~]$ hadoop fs -ls /
Found 5 items
drwxr-xr-x  - hadoop supergroup          0 2012-03-08 00:10 /HebutWinOS
drwxr-xr-x  - hadoop supergroup          0 2012-03-08 04:49 /TestDir
-rw-r--r--  3 hadoop supergroup         20 2012-03-08 05:06 /test1
drwxr-xr-x  - hadoop supergroup          0 2012-03-05 19:29 /user
drwxr-xr-x  - hadoop supergroup          0 2012-03-05 19:43 /usr
[hadoop@Master ~]$

```

图 6-4-2 运行结果 (2)

## 6.5 删除HDFS上的文件

通过“**FileSystem.delete (Path f, Boolean recursive)**”可删除指定的 HDFS 文件，其中 f 为需要删除文件的完整路径，recursive 用来确定是否进行递归删除。具体实现如下：

```
package com.hebut.file;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class DeleteFile {

    public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);

        Path delef=new Path("/test1");

        boolean isDeleted=hdfs.delete(delef,false);
        //递归删除
        //boolean isDeleted=hdfs.delete(delef,true);
        System.out.println("Delete?" + isDeleted);
    }
}
```

运行结果如图 6-5-1 和图 6-5-2 所示。

### 1) 控制台结果

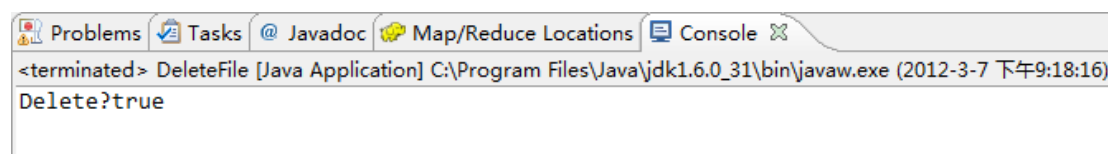


图 6-5-1 运行结果（1）

### 2) 项目浏览器

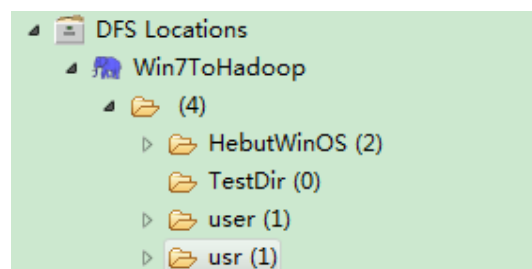


图 6-5-2 运行结果（2）

## 6.6 删除HDFS上的目录

同删除文件代码一样，只是换成删除目录路径即可，如果目录下有文件，要进行递归删除。

## 6.7 查看某个HDFS文件是否存在

通过“`FileSystem.exists (Path f)`”可查看指定 HDFS 文件是否存在，其中 f 为文件的完整路径。具体实现如下：

```
package com.hebut.file;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class CheckFile {
    public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);
        Path findf=new Path("/test1");
        boolean isExists=hdfs.exists(findf);
        System.out.println("Exist?" + isExists);
    }
}
```

运行结果如图 6-7-1 和图 6-7-2 所示。

### 1) 控制台结果

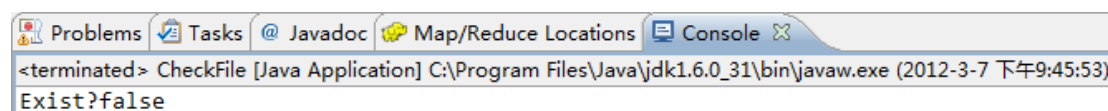


图 6-7-1 运行结果（1）

### 2) 项目浏览器

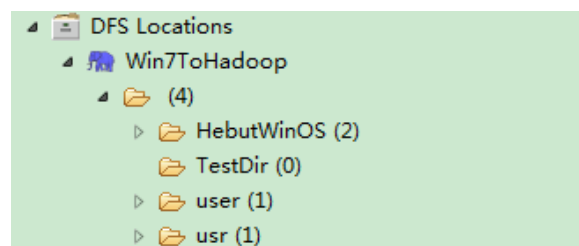


图 6-7-2 运行结果（2）

## 6.8 查看HDFS文件的最后修改时间

通过“**FileSystem.getModificationTime()**”可查看指定 HDFS 文件的修改时间。具体实现如下：

```
package com.hebut.file;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class GetLTime {

    public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);

        Path fpath =new Path("/user/hadoop/test/file1.txt");

        FileStatus fileStatus=hdfs.getFileStatus(fpath);
        long modiTime=fileStatus.getModificationTime();

        System.out.println("file1.txt 的修改时间是"+modiTime);
    }
}
```

运行结果如图 6-8-1 所示。

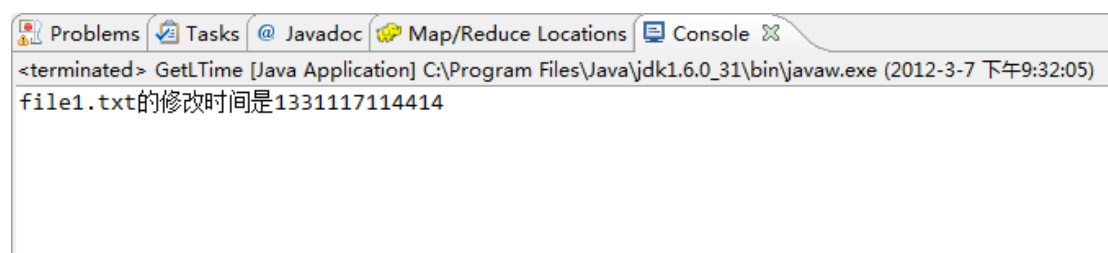


图 6-8-1 控制台结果

## 6.9 读取HDFS某个目录下的所有文件

通过“**FileStatus.getPath()**”可查看指定 HDFS 中某个目录下所有文件。具体实现如下：



```

package com.hebut.file;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class ListAllFile {
    public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);

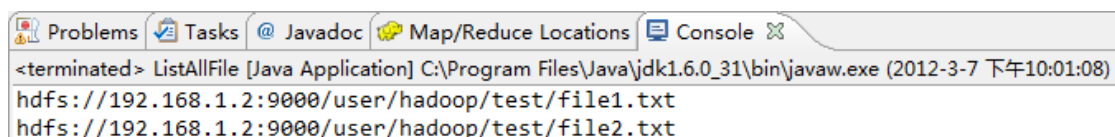
        Path listf =new Path("/user/hadoop/test");

        FileStatus stats[]=hdfs.listStatus(listf);
        for(int i = 0; i < stats.length; ++i)
        {
            System.out.println(stats[i].getPath().toString());
        }
        hdfs.close();
    }
}

```

运行结果如图 6-9-1 和图 6-9-2 所示。

### 1) 控制台结果



```

<terminated> ListAllFile [Java Application] C:\Program Files\Java\jdk1.6.0_31\bin\javaw.exe (2012-3-7 下午10:01:08)
hdfs://192.168.1.2:9000/user/hadoop/test/file1.txt
hdfs://192.168.1.2:9000/user/hadoop/test/file2.txt

```

图 6-9-1 运行结果 (1)

### 2) 项目浏览器

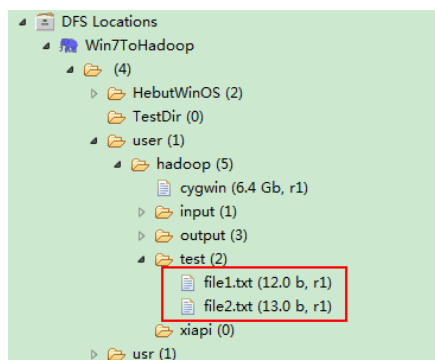


图 6-9-2 运行结果 (2)

## 6.10 查找某个文件在HDFS集群的位置

通过“**FileSystem.getFileBlockLocation (FileStatus file, long start, long len)**”可查找指定文件在 HDFS 集群上的位置，其中 file 为文件的完整路径，start 和 len 来标识查找文件的路径。具体实现如下：

```
package com.hebut.file;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.BlockLocation;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class FileLoc {
    public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);
        Path fpath=new Path("/user/hadoop/cygwin");

        FileStatus filestatus = hdfs.getFileStatus(fpath);
        BlockLocation[] blkLocations = hdfs.getFileBlockLocations(
            filestatus, 0, filestatus.getLen());

        int blockLen = blkLocations.length;
        for(int i=0;i<blockLen;i++){
            String[] hosts = blkLocations[i].getHosts();
            System.out.println("block_"+i+"_location:"+hosts[0]);
        }
    }
}
```

运行结果如图 6-10-1 和 6.10.2 所示。

### 1) 控制台结果



```
<terminated> FileLoc [Java Application] C:\Program Files\Java\jdk1.6.0_31\bin\javaw.exe (2012-3-8 上午9:51:49)
block_0_location:Slave2.Hadoop
block_1_location:Slave2.Hadoop
block_2_location:Slave1.Hadoop
block_3_location:Slave3.Hadoop
block_4_location:Slave1.Hadoop
block_5_location:Slave1.Hadoop
block_6_location:Slave1.Hadoop
block_7_location:Slave2.Hadoop
block_8_location:Slave3.Hadoop
block_9_location:Slave2.Hadoop
```

图 6-10-1 运行结果（1）

## 2) 项目浏览器

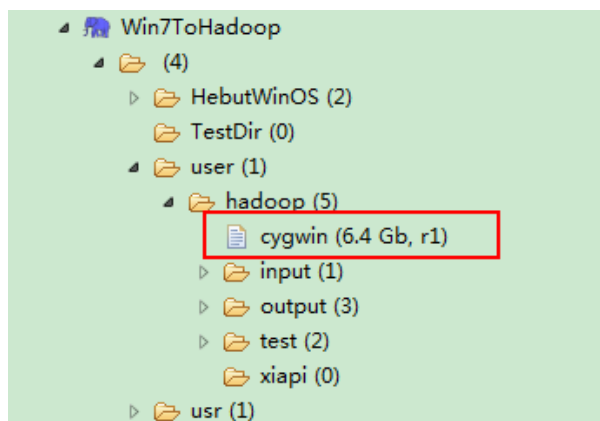


图 6-10-2 运行结果（2）

## 6.11 获取HDFS集群上所有节点名称信息

通过“`DatanodeInfo.getHostName()`”可获取 HDFS 集群上的所有节点名称。具体实现如下：

```
package com.hebut.file;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.hdfs.DistributedFileSystem;
import org.apache.hadoop.hdfs.protocol.DatanodeInfo;

public class GetList {

    public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem fs=FileSystem.get(conf);

        DistributedFileSystem hdfs = (DistributedFileSystem)fs;
        DatanodeInfo[] dataNodeStats = hdfs.getDataNodeStats();

        for(int i=0;i<dataNodeStats.length;i++){
            System.out.println("DataNode_"+i+"_Name:"
                               +dataNodeStats[i].getHostName());
        }
    }
}
```

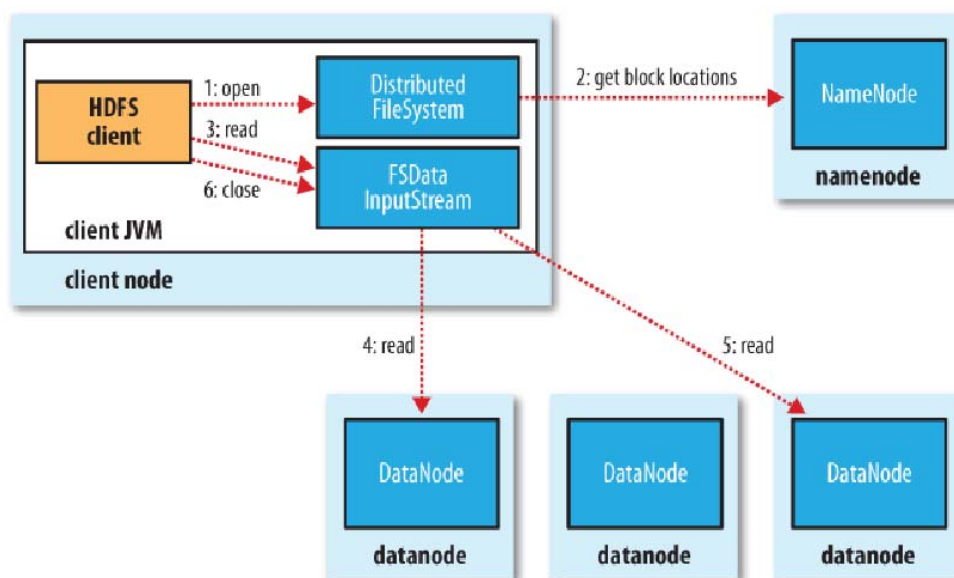
运行结果如图 6-11-1 所示。

```
<terminated> GetList [Java Application] C:\Program Files\Java\jdk1.6.0_31\bin\javaw.exe (2012-3-8 上午10:09:16)
DataNode_0_Name:Slave2.Hadoop
DataNode_1_Name:Slave3.Hadoop
DataNode_2_Name:Slave1.Hadoop
```

图 6-11-1 控制台结果

## 7、HDFS的读写数据流

### 7.1 文件的读取剖析



文件读取的过程如下：

#### 1) 解释一

- 客户端(client)用 FileSystem 的 open()函数打开文件。
- DistributedFileSystem 用 RPC 调用元数据节点，得到文件的数据块信息。
- 对于每一个数据块，元数据节点返回保存数据块的数据节点的地址。
- DistributedFileSystem 返回 FSDataInputStream 给客户端，用来读取数据。
- 客户端调用 stream 的 read()函数开始读取数据。
- DFSInputStream 连接保存此文件第一个数据块的最近的数据节点。
- Data 从数据节点读到客户端(client)。
- 当此数据块读取完毕时，DFSInputStream 关闭和此数据节点的连接，然后连接此文件下一个数据块的最近的数据节点。
- 当客户端读取完毕数据的时候，调用 FSDataInputStream 的 close 函数。
- 在读取数据的过程中，如果客户端在与数据节点通信出现错误，则尝试连接包含此

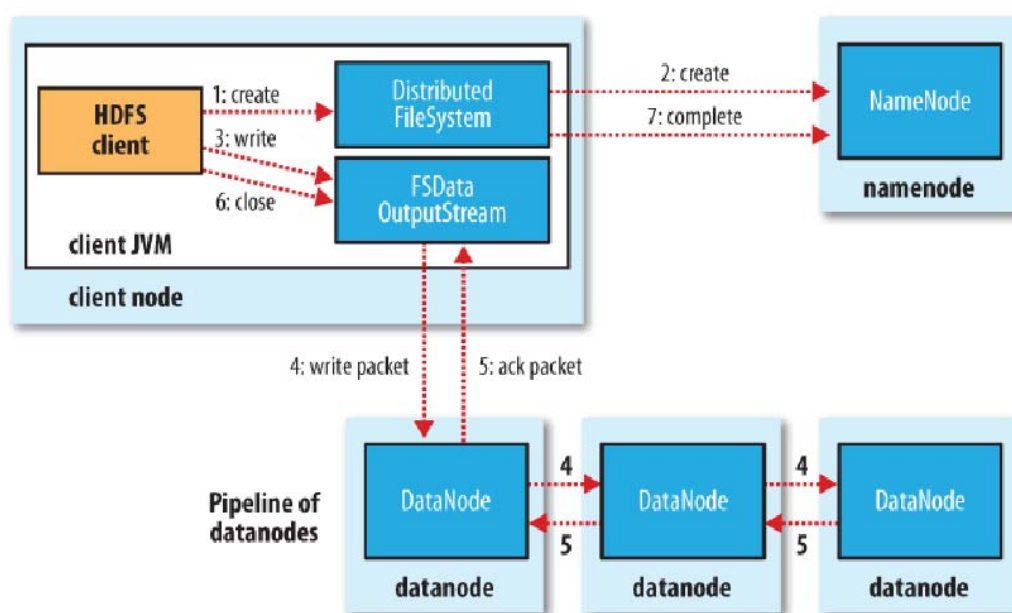
数据块的下一个数据节点。

- 失败的数据节点将被记录，以后不再连接。

## 2) 解释二

- 使用 HDFS 提供的客户端开发库，向远程的 Namenode 发起 RPC 请求；
- Namenode 会视情况返回文件的部分或者全部 block 列表，对于每个 block，Namenode 都会返回有该 block 拷贝的 datanode 地址；
- 客户端开发库会选取离客户端最近的数据节点来读取 block；
- 读取完当前 block 的数据后，关闭与当前的 datanode 连接，并为读取下一个 block 寻找最佳的 datanode；
- 当读完列表的 block 后，且文件读取还没有结束，客户端开发库会继续向 Namenode 获取下一批的 block 列表。
- 读取完一个 block 都会进行 checksum 验证，如果读取 datanode 时出现错误，客户端会通知 Namenode，然后再从下一个拥有该 block 拷贝的 datanode 继续读。

## 7.2 文件的写入剖析



写入文件的过程比读取较为复杂：

## 1) 解释一

- 客户端调用 create() 来创建文件
- DistributedFileSystem 用 RPC 调用元数据节点，在文件系统的命名空间中创建一个新的文件。
- 元数据节点首先确定文件原来不存在，并且客户端有创建文件的权限，然后创建新文件。
- DistributedFileSystem 返回 DFSOutputStream，客户端用于写数据。
- 客户端开始写入数据，DFSOutputStream 将数据分成块，写入 data queue。
- Data queue 由 Data Streamer 读取，并通知元数据节点分配数据节点，用来存储数据

块(每块默认复制 3 块)。分配的数据节点放在一个 pipeline 里。

- Data Streamer 将数据块写入 pipeline 中的第一个数据节点。第一个数据节点将数据块发送给第二个数据节点。第二个数据节点将数据发送给第三个数据节点。
- DFSOutputStream 为发出去的数据块保存了 ack queue，等待 pipeline 中的数据节点告知数据已经写入成功。
- 如果数据节点在写入的过程中失败：
  - 关闭 pipeline，将 ack queue 中的数据块放入 data queue 的开始。
  - 当前的数据块在已经写入的数据节点中被元数据节点赋予新的标示，则错误节点重启后能够察觉其数据块是过时的，会被删除。
  - 失败的数据节点从 pipeline 中移除，另外的数据块则写入 pipeline 中的另外两个数据节点。
  - 元数据节点则被通知此数据块是复制块数不足，将来会再创建第三份备份。
- 当客户端结束写入数据，则调用 stream 的 close 函数。此操作将所有的数据块写入 pipeline 中的数据节点，并等待 ack queue 返回成功。**最后**通知元数据节点写入完毕。

## 2) 解释二

- 使用 HDFS 提供的客户端开发库，向远程的 Namenode 发起 RPC 请求；
- Namenode 会检查要创建的文件是否已经存在，创建者是否有权限进行操作，成功则会为文件创建一个记录，否则会让客户端抛出异常；
- 当客户端开始写入文件的时候，开发库会将文件切分成多个 packets，并在内部以”data queue”的形式管理这些 packets，并向 Namenode 申请新的 blocks，获取用来存储 replicas 的合适的 datanodes 列表，列表的大小根据在 Namenode 中对 replication 的设置而定。
- 开始以 pipeline（管道）的形式将 packet 写入所有的 replicas 中。开发库把 packet 以流的方式写入第一个 datanode，该 datanode 把该 packet 存储之后，再将其传递给在此 pipeline 中的下一个 datanode，直到最后一个 datanode，这种写数据的方式呈流水线的形式。
- 最后一个 datanode 成功存储之后会返回一个 ack packet，在 pipeline 里传递至客户端，在客户端的开发库内部维护着”ack queue”，成功收到 datanode 返回的 ack packet 后会从”ack queue”移除相应的 packet。
- 如果传输过程中，有某个 datanode 出现了故障，那么当前的 pipeline 会被关闭，出现故障的 datanode 会从当前的 pipeline 中移除，剩余的 block 会继续剩下的 datanode 中继续以 pipeline 的形式传输，同时 Namenode 会分配一个新的 datanode，保持 replicas 设定的数量。



## 参考文献

---

感谢以下文章的编写作者，没有你们的铺路，我或许会走得很艰难，参考不分先后，贡献同等珍贵。

**【1】Hadoop 实战——陆嘉恒——机械工业出版社**

**【2】实战 Hadoop——刘鹏——电子工业出版社**