

# 第 4 章 分布式存储系统——HDFS

HDFS 作为分布式文件管理系统, Hadoop 的基础。

以下是本章学习要点:

- 分布式文件系统与 HDFS
- HDFS 体系结构与基本概念★★★
- HDFS 的 shell 操作★★★
- 搭建 eclipse 开发环境★★
- java 接口及常用 api★★★
- hadoop 的 RPC 机制★
- hadoop 读写数据的过程分析★★

## 4.1. 分布式文件系统与 HDFS

数据量越来越大, 在一个操作系统管辖的范围存不下了, 那么就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, 迫切需要一种系统来管理多台机器上的文件, 这就是分布式文件管理系统—。

学术二点的定义就是: 分布式文件系统是一种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。

分布式文件管理系统很多, hdfsHDFS 只是其中一种。适用于一次写入、多次查询的情况, 不支持并发写情况, 小文件不合适。

**小提示:** 如何在刚开始学习的时候, 形象化的理解什么是 HDFS 呢? 我们可以把 HDFS 看做是Windows的文件系统。在Windows的文件系统维护着有一套很多层次的文件夹目录, 这么复杂的目录层次是为了在文件夹中分门别类的存放文件。我们经常做的操作是创建文件夹、创建文件、移动文件、复制文件、删除文件、编辑文件、查找文件等。HDFS 与 Windows中的文件系统类似, 看到的和操作的也类似。读者可以把 HDFS 理解为分 Windows文件系统。

## 4.2. HDFS 的 shell 操作

既然 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。

我们执行以下操作的时候, 一定要确定 hadoop 是正常运行的, 使用 jps 命令确保看到各个 hadoop 进程。

我们执行命令 hadoop fs, 如图 4-1 所示。

```

root@itcast225:~ [52x20]
Connection Edit View Window Option Help
[root@itcast225 ~]# hadoop fs
Warning: $HADOOP_HOME is deprecated.

Usage: java FsShell
      [-ls <path>]
      [-lsr <path>]
      [-du <path>]
      [-dus <path>]
      [-count[-q] <path>]
      [-mv <src> <dst>]
      [-cp <src> <dst>]
      [-rm [-skipTrash] <path>]
      [-rmr [-skipTrash] <path>]
      [-expunge]
      [-put <localsrc> ... <dst>]
      [-copyFromLocal <localsrc> ... <dst>]
      [-moveFromLocal <localsrc> ... <dst>]
      [-get [-ignoreCrc] [-crc] <src> <localdst>
>           [-getmerge <src> <localdst> [-addall]]

```

图 4-1

图中显示了很多命令选项信息。以上截图不全，我在表格 4-1 中完整地列出了支持的命令选项。

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
-setrep	-setrep [-R] [-w] <副本数> <路径>	修改副本数量
-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

注意：以上表格中**对于路径，包括 hdfs 中的路径和 linux 中的路径**。对于容易产生歧义的地方，会特别指出“linux 路径”或者“hdfs 路径”。如果没有明确指出，意味着是 hdfs 路径。

下面我们讲述每个命令选项的用法。

## ● -ls 显示当前目录结构

该命令选项表示查看指定路径的当前目录结构，后面跟 hdfs 路径，如图 4-1 所示。

```
[root@hadoop0 Desktop]# hadoop fs -ls /
Found 6 items
drwxr-xr-x  - root supergroup          0 2013-07-24 09:17 /abc
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /hbase
-rw-r--r--  1 root supergroup      6176 2013-07-24 07:48 /hbase-env.sh
-rw-r--r--  1 root supergroup     37645 2013-07-24 07:52 /install.log
drwxr-xr-x  - root supergroup          0 2013-07-24 14:39 /user
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /usr
```

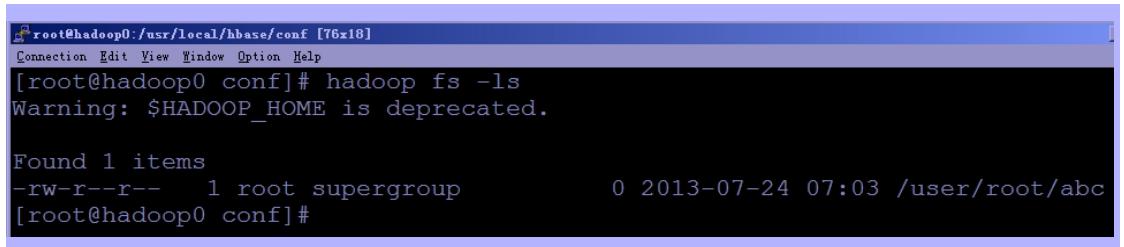
图 4-1

上图中的路径是 hdfs 根目录，显示的内容格式与 linux 的命令 ls -l 显示的内容格式非常相似，现在解析每一行的内容格式：

- 首字母表示文件夹(如果是“d”)还是文件(如果是“-”);
- 后面的 9 位字符表示权限;
- 后面的数字或者“-”表示副本数。如果是文件，使用数字表示副本数；文件夹没有副本；
- 后面的“root”表示属主；
- 后面的“supergroup”表示属组；
- 后面的“0”、“6176”、“37645”表示文件大小，单位是字节；
- 后面的时间表示修改时间，格式是年月日时分；
- 最后一项表示文件路径。

可见根目录下面有四个文件夹、两个文件。

如果该命令选项后面没有路径，那么就会访问 /user/<当前用户> 目录。我们使用 root 用户登录，因此会访问 hdfs 的 /user/root 目录，如图 4-2 所示。



```
[root@hadoop0 conf]# hadoop fs -ls
Warning: $HADOOP_HOME is deprecated.

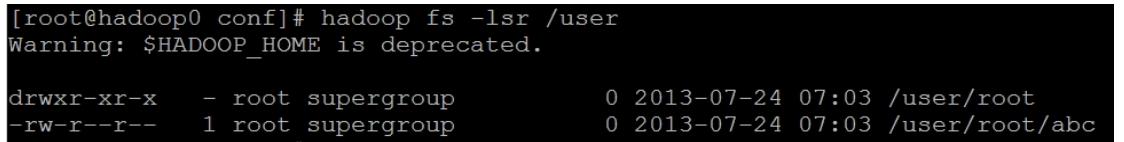
Found 1 items
-rw-r--r-- 1 root supergroup          0 2013-07-24 07:03 /user/root/abc
[root@hadoop0 conf]#
```

图 4-2

如果没有这个目录/user/root，会提示文件不存在的错误。

## ● **-lsr** 递归显示目录结构

该命令选项表示递归显示当前路径的目录结构，后面跟 hdfs 路径。如图 4-3 所示。



```
[root@hadoop0 conf]# hadoop fs -lsr /user
Warning: $HADOOP_HOME is deprecated.

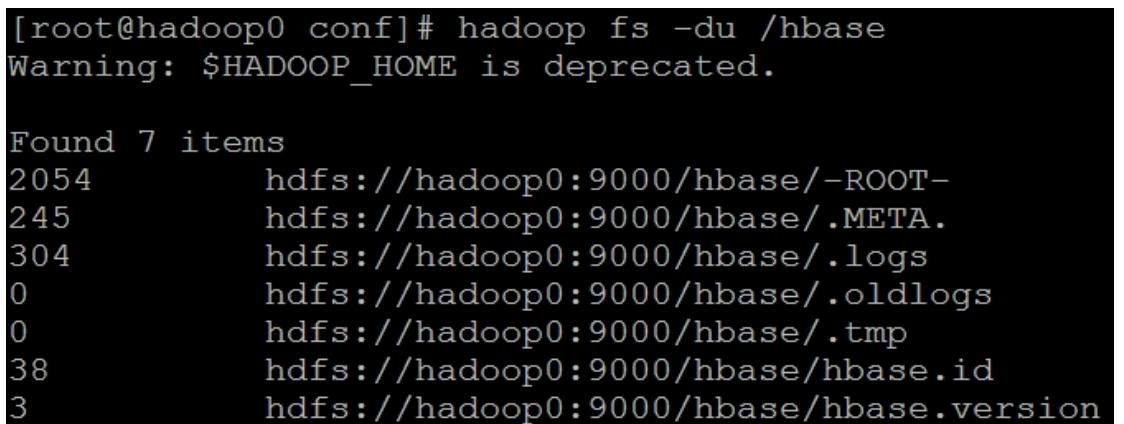
drwxr-xr-x  - root supergroup          0 2013-07-24 07:03 /user/root
-rw-r--r--  1 root supergroup          0 2013-07-24 07:03 /user/root/abc
[root@hadoop0 conf]#
```

图 4-3

显示/user 目录下有个 root 目录，root 目录下有文件 abc。

## ● **-du** 统计目录下各文件大小

该命令选项显示指定路径下的文件大小，单位是字节，如图 4-4 所示。



```
[root@hadoop0 conf]# hadoop fs -du /hbase
Warning: $HADOOP_HOME is deprecated.

Found 7 items
2054      hdfs://hadoop0:9000/hbase/-ROOT-
245       hdfs://hadoop0:9000/hbase/.META.
304       hdfs://hadoop0:9000/hbase/.logs
0         hdfs://hadoop0:9000/hbase/.oldlogs
0         hdfs://hadoop0:9000/hbase/.tmp
38        hdfs://hadoop0:9000/hbase/hbase.id
3         hdfs://hadoop0:9000/hbase/hbase.version
[root@hadoop0 conf]#
```

图 4-4

## ● **-dus** 汇总统计目录下文件大小

该命令选项显示指定路径的文件大小，单位是字节，如图 4-5 所示。

```
[root@hadoop0 conf]# hadoop fs -dus /hbase
Warning: $HADOOP_HOME is deprecated.

hdfs://hadoop0:9000/hbase          2644
```

图 4-5

请读者比较图 4-4 与图 4-5 的区别，体会两个命令选项的不同含义。

## ● -count 统计文件(夹)数量

该命令选项显示指定路径下的文件夹数量、文件数量、文件总大小信息，如图 4-6 所示。

```
[root@hadoop0 conf]# hadoop fs -count /usr
       6      1      4 hdfs://hadoop0:9000/usr
[root@hadoop0 conf]# hadoop fs -lsr /usr
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /usr/local
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /usr/local/hadoop
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /usr/local/hadoop/tmp
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /usr/local/hadoop/tmp/mapred
drwx-----  - root supergroup          0 2013-07-24 06:54 /usr/local/hadoop/tmp/mapred/system
-rw-----  1 root supergroup          4 2013-07-24 06:54 /usr/local/hadoop/tmp/mapred/system/jobtracker.info
```

图 4-6

在图 4-6 中有两条命令，下面的命令是为了佐证上面命令的正确性的。

## ● -mv 移动

该命令选项表示移动 hdfs 的文件到指定的 hdfs 目录中。后面跟两个路径，第一个表示源文件，第二个表示目的目录，如图 4-7 所示。

```
[root@hadoop0 conf]# hadoop fs -lsr /user
drwxr-xr-x  - root supergroup          0 2013-07-24 07:03 /user/root
-rw-r--r--  1 root supergroup          0 2013-07-24 07:03 /user/root/abc
[root@hadoop0 conf]# hadoop fs -mv /user/root/abc /user
[root@hadoop0 conf]# hadoop fs -lsr /user
-rw-r--r--  1 root supergroup          0 2013-07-24 07:03 /user/abc
drwxr-xr-x  - root supergroup          0 2013-07-24 07:19 /user/root
```

图 4-7

在图 4-7 中有三条命令，是为了体现移动前后的变化情况。

## ● -cp 复制

该命令选项表示复制 hdfs 指定的文件到指定的 hdfs 目录中。后面跟两个路径，第

一个是被复制的文件，第二个是目的地，如图 4-8 所示。

```
[root@hadoop0 conf]# hadoop fs -lsr /user
-rw-r--r-- 1 root supergroup          0 2013-07-24 07:03 /user/abc
drwxr-xr-x - root supergroup          0 2013-07-24 07:19 /user/root
[root@hadoop0 conf]# hadoop fs -cp /user/abc /user/root
[root@hadoop0 conf]#
[root@hadoop0 conf]# hadoop fs -lsr /user
-rw-r--r-- 1 root supergroup          0 2013-07-24 07:03 /user/abc
drwxr-xr-x - root supergroup          0 2013-07-24 07:24 /user/root
-rw-r--r-- 1 root supergroup          0 2013-07-24 07:24 /user/root/abc
```

图 4-8

在图 4-8 中有三条命令，是为了体现复制前后的变化情况。

## ● -rm 删 除文件/空 白文件夹

该命令选项表示删除指定的文件或者空目录，如图 4-9 所示。

```
[root@hadoop0 conf]# hadoop fs -lsr /user
-rw-r--r-- 1 root supergroup          0 2013-07-24 07:03 /user/abc
drwxr-xr-x - root supergroup          0 2013-07-24 07:24 /user/root
-rw-r--r-- 1 root supergroup          0 2013-07-24 07:24 /user/root/abc
[root@hadoop0 conf]#
[root@hadoop0 conf]# hadoop fs -rm /user/abc
Deleted hdfs://hadoop0:9000/user/abc
[root@hadoop0 conf]#
[root@hadoop0 conf]# hadoop fs -lsr /user
drwxr-xr-x - root supergroup          0 2013-07-24 07:24 /user/root
-rw-r--r-- 1 root supergroup          0 2013-07-24 07:24 /user/root/abc
[root@hadoop0 conf]#
[root@hadoop0 conf]# hadoop fs -rm /user/root
rm: Cannot remove directory "hdfs://hadoop0:9000/user/root", use -rmdir instead
```

图 4-9

在图 4-9 中，前三条命令是为了体现执行前后的变化情况。第四条命令是删除非空的 “/user/root” 目录，操作失败，表明不能删除非空目录。

## ● -rmdir 递归删除

该命令选项表示递归删除指定目录下的所有子目录和文件，如图 4-10 所示。

```
[root@hadoop0 conf]# hadoop fs -lsr /user
drwxr-xr-x - root supergroup          0 2013-07-24 07:24 /user/root
-rw-r--r-- 1 root supergroup          0 2013-07-24 07:24 /user/root/abc
[root@hadoop0 conf]#
[root@hadoop0 conf]# hadoop fs -rmdir /user
Deleted hdfs://hadoop0:9000/user
```

图 4-10

## ● -put 上传文件

该命令选项表示把 linux 上的文件复制到 hdfs 中，如图 4-11 所示。

```
[root@hadoop0 conf]# ls
hadoop-metrics.properties  hbase-policy.xml  log4j.properties
hbase-env.sh                hbase-site.xml    regionserver
[root@hadoop0 conf]#
[root@hadoop0 conf]# hadoop fs -put hbase-env.sh /
[root@hadoop0 conf]#
[root@hadoop0 conf]# hadoop fs -ls /
Found 3 items
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /hbase
-rw-r--r--  1 root supergroup      6176 2013-07-24 07:48 /hbase-env.sh
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /usr
```

图 4-11

## ● -copyFromLocal 从本地复制

操作与-put一致，不再举例。

## ● -moveFromLocal 从本地移动

该命令表示把文件从 linux 上移动到 hdfs 中，如图 4-12 所示。

```
[root@hadoop0 conf]# hadoop fs -ls /
Found 3 items
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /hbase
-rw-r--r--  1 root supergroup      6176 2013-07-24 07:48 /hbase-env.sh
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /usr
[root@hadoop0 conf]# cd
[root@hadoop0 ~]# ls
anaconda-ks.cfg  Desktop   Downloads   install.log       metastore_db  Pictures  Templates
derby.log        Documents  hdfs-namenode  install.log.syslog  Music       Public    Videos
[root@hadoop0 ~]#
[root@hadoop0 ~]# hadoop fs -moveFromLocal install.log /
[root@hadoop0 ~]#
[root@hadoop0 ~]# ls
anaconda-ks.cfg  Desktop   Downloads   install.log.syslog  Music       Public    Videos
derby.log        Documents  hdfs-namenode  metastore_db      Pictures  Templates
[root@hadoop0 ~]#
[root@hadoop0 ~]# hadoop fs -ls /
Found 4 items
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /hbase
-rw-r--r--  1 root supergroup      6176 2013-07-24 07:48 /hbase-env.sh
-rw-r--r--  1 root supergroup     37645 2013-07-24 07:52 /install.log
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /usr
```

图 4-12

## ● getmerge 合并到本地

该命令选项的含义是把 hdfs 指定目录下的所有文件内容合并到本地 linux 的文件中，如图 4-13 所示。

```
[root@hadoop0 Desktop]# ls
[root@hadoop0 Desktop]# hadoop fs -getmerge /hbase abc
13/07/24 07:59:42 INFO util.NativeCodeLoader: Loaded the native-hadoop library
[root@hadoop0 Desktop]# ls
abc
```

图 4-13

### ● **-cat** 查看文件内容

该命令选项是查看文件内容，如图 4-14 所示。

```
[root@hadoop0 Desktop]# hadoop fs -cat /hbase-env.sh
#
# /**
# * Copyright 2007 The Apache Software Foundation
# *
# * Licensed to the Apache Software Foundation (ASF) under one
# * or more contributor license agreements. See the NOTICE file
# * distributed with this work for additional information
# * regarding copyright ownership. The ASF licenses this
# * to you under the Apache License, Version 2.0 (the
# * "License"); you may not use this file except in
# * accordance with the License. You may obtain a copy of the
# * License at
# *
# *     http://www.apache.org/licenses/LICENSE-2.0
# *
```

图 4-14

### ● **-text** 查看文件内容

该命令选项可以认为作用和用法与-cat 相同，此处略。

### ● **-mkdir** 创建空白文件夹

该命令选项表示创建文件夹，后面跟的路径是在 hdfs 将要创建的文件夹，如图 4-15 所示。

```
[root@hadoop0 Desktop]# hadoop fs -ls /
Found 4 items
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /hbase
-rw-r--r--  1 root supergroup      6176 2013-07-24 07:48 /hbase-env.sh
-rw-r--r--  1 root supergroup    37645 2013-07-24 07:52 /install.log
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /usr
[root@hadoop0 Desktop]# hadoop fs -mkdir /abc
[root@hadoop0 Desktop]#
[root@hadoop0 Desktop]# hadoop fs -ls /
Found 5 items
drwxr-xr-x  - root supergroup          0 2013-07-24 09:17 /abc
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /hbase
-rw-r--r--  1 root supergroup      6176 2013-07-24 07:48 /hbase-env.sh
-rw-r--r--  1 root supergroup    37645 2013-07-24 07:52 /install.log
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /usr
```

图 4-15

## ● -setrep 设置副本数量

该命令选项是修改已保存文件的副本数量，后面跟副本数量，再跟文件路径，如图 4-16 所示。

```
[root@hadoop0 Desktop]# hadoop fs -ls /install.log
Found 1 items
-rw-r--r-- 1 root supergroup      37645 2013-07-24 07:52 /install.log
[root@hadoop0 Desktop]#
[root@hadoop0 Desktop]# hadoop fs -setrep 2 /install.log
Replication 2 set: hdfs://hadoop0:9000/install.log
[root@hadoop0 Desktop]#
[root@hadoop0 Desktop]# hadoop fs -ls /install.log
Found 1 items
-rw-r--r-- 2 root supergroup      37645 2013-07-24 07:52 /install.log
```

图 4-16

在图 4-16 中，我们修改了文件/install.log 的副本数，由 1 修改为 2，意味着多了一个副本，HDFS 会自动执行文件的复制工作，产生新的副本。

如果最后的路径表示文件夹，那么需要跟选项-R，表示对文件夹中的所有文件都修改副本，如图 4-17 所示。.

```
[root@hadoop0 Desktop]# hadoop fs -lsr /user
drwxr-xr-x - root supergroup          0 2013-07-24 14:39 /user/root
-rw-r--r-- 1 root supergroup          0 2013-07-24 14:39 /user/root/abc
[root@hadoop0 Desktop]#
[root@hadoop0 Desktop]# hadoop fs -setrep -R 2 /user
Replication 2 set: hdfs://hadoop0:9000/user/root/abc
[root@hadoop0 Desktop]#
[root@hadoop0 Desktop]# hadoop fs -lsr /user
drwxr-xr-x - root supergroup          0 2013-07-24 14:39 /user/root
-rw-r--r-- 2 root supergroup          0 2013-07-24 14:39 /user/root/abc
```

图 4-17

在图 4-17 中，我们对/user 文件夹进行的操作，使用了选项-R，那么/user/root 下的文件 abc 的副本数发生了改变。

还有一个选项是-w，表示等待副本操作结束才退出命令，如图 4-18 所示。.

```
[root@hadoop0 Desktop]# hadoop fs -setrep -R -w 1 /user
Replication 1 set: hdfs://hadoop0:9000/user/root/abc
Waiting for hdfs://hadoop0:9000/user/root/abc ... done
```

图 4-18

请读者自己比较以上两图中使用-q 前后执行结果的变化情况。

## ● -touchz 创建空白文件

该命令选项是在 hdfs 中创建空白文件，如图 4-19 所示。.

```
[root@hadoop0 Desktop]# hadoop fs -ls /
Found 6 items
drwxr-xr-x  - root supergroup          0 2013-07-24 09:17 /abc
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /hbase
-rw-r--r--  1 root supergroup       6176 2013-07-24 07:48 /hbase-env.sh
-rw-r--r--  2 root supergroup      37645 2013-07-24 07:52 /install.log
drwxr-xr-x  - root supergroup          0 2013-07-24 14:39 /user
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /usr
[root@hadoop0 Desktop]#
[root@hadoop0 Desktop]# hadoop fs -touchz /emptyfile
[root@hadoop0 Desktop]#
[root@hadoop0 Desktop]# hadoop fs -ls /
Found 7 items
drwxr-xr-x  - root supergroup          0 2013-07-24 09:17 /abc
-rw-r--r--  1 root supergroup          0 2013-07-24 14:54 /emptyfile
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /hbase
-rw-r--r--  1 root supergroup       6176 2013-07-24 07:48 /hbase-env.sh
-rw-r--r--  2 root supergroup      37645 2013-07-24 07:52 /install.log
drwxr-xr-x  - root supergroup          0 2013-07-24 14:39 /user
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /usr
```

图 4-19

## ● **-stat** 显示文件的统计信息

该命令选项显示文件的一些统计信息，如图 4-20 所示。

```
[root@hadoop0 Desktop]# hadoop fs -stat '%b %n %o %r %Y' /install.log
37645 install.log 67108864 1 1374677532018
```

图 4-20

在图 4-20 中，命令选项后面可以有格式，使用引号表示。示例中的格式“%b %n %o %r %Y”依次表示文件大小、文件名称、块大小、副本数、访问时间。

## ● **-tail** 查看文件尾部内容

该命令选项显示文件最后 1K 字节的内容。一般用于查看日志。如果带有选项-f，那么当文件内容变化时，也会自动显示→，如图 4-21 所示。

```
[root@hadoop0 Desktop]# hadoop fs -tail /install.log
ibootmgr-0.5.4-10.el6.i686
Installing xvattr-1.3-18.el6.i686
Installing irqbalance-0.55-34.el6.i686
Installing cyrus-sasl-plain-2.1.23-13.el6.i686
Installing mlocate-0.22.2-3.el6.i686
Installing patch-2.6-6.el6.i686
Installing lsof-4.82-4.el6.i686
Installing rsync-3.0.6-9.el6.i686
Installing unzip-6.0-1.el6.i686
Installing make-3.81-20.el6.i686
Installing ed-1.1-3.3.el6.i686
Installing nano-2.0.9-7.el6.i686
Installing time-1.7-37.1.el6.i686
Installing attr-2.4.44-7.el6.i686
Installing scl-utils-20120423-2.el6.i686
Installing wdaemon-0.17-2.el6.i686
Installing mtr-0.75-5.el6.i686
Installing traceroute-2.0.14-2.el6.i686
Installing setserial-2.17-25.el6.i686
Installing vconfig-1.9-8.1.el6.i686
Installing rfkill-0.3-4.el6.i686
Installing rdate-1.4-16.el6.i686
Installing zip-3.0-1.el6.i686
Installing bridge-utils-1.2-9.el6.i686
Installing eject-2.1.5-17.el6.i686
Installing ledmon-0.32-1.el6.i686
Installing strace-4.5.19-1.11.el6_2.1.i686
Installing b43-fwcutter-012-2.2.el6.i686
*** FINISHED INSTALLING PACKAGES ***[root@hadoop0 Desktop]#
```

图 4-21

## ● **-chmod** 修改文件权限

该命令选项的使用类似于 linux 的 shell 中的 chmod 用法，作用是修改文件的权限，如图 4-22 所示。

```
[root@hadoop0 Desktop]# hadoop fs -ls /
Found 7 items
drwxr-xr-x  - root supergroup          0 2013-07-24 09:17 /abc
-rw-r--r--  1 root supergroup          0 2013-07-24 14:54 /emptyfile
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /hbase
-rw-r--r--  1 root supergroup         6176 2013-07-24 07:48 /hbase-env.sh
-rw-r--r--  2 root supergroup        37645 2013-07-24 07:52 /install.log
drwxr-xr-x  - root supergroup          0 2013-07-24 14:39 /user
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /usr
[root@hadoop0 Desktop]#
[root@hadoop0 Desktop]# hadoop fs -chmod 755 /emptyfile
[root@hadoop0 Desktop]#
[root@hadoop0 Desktop]# hadoop fs -ls /
Found 7 items
drwxr-xr-x  - root supergroup          0 2013-07-24 09:17 /abc
-rw-r--r--  1 root supergroup          0 2013-07-24 14:54 /emptyfile
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /hbase
-rw-r--r--  1 root supergroup         6176 2013-07-24 07:48 /hbase-env.sh
-rw-r--r--  2 root supergroup        37645 2013-07-24 07:52 /install.log
drwxr-xr-x  - root supergroup          0 2013-07-24 14:39 /user
drwxr-xr-x  - root supergroup          0 2013-07-24 06:54 /usr
```

图 4-22

在图 4-22 中，修改了文件/emptyfile 的权限。

如果加上选项-R，可以对文件夹中的所有文件修改权限，如图 4-23 所示。-

```
[root@hadoop0 Desktop]# hadoop fs -lsr /user
drwxr-xr-x  - root supergroup          0 2013-07-24 14:39 /user/root
-rw-r--r--  1 root supergroup          0 2013-07-24 14:39 /user/root/abc
[root@hadoop0 Desktop]#
[root@hadoop0 Desktop]# hadoop fs -chmod -R 700 /user
[root@hadoop0 Desktop]#
[root@hadoop0 Desktop]# hadoop fs -lsr /user
drwx-----  - root supergroup          0 2013-07-24 14:39 /user/root
-rw-----  1 root supergroup          0 2013-07-24 14:39 /user/root/abc
```

图 4-23

## ● -chown 修改属主

该命令选项表示修改文件的属主，如图 4-24 所示。

```
[root@hadoop0 etc]# hadoop fs -ls /emptyfile
Found 1 items
-rw-r--r--  1 root supergroup          0 2013-07-24 14:54 /emptyfile
[root@hadoop0 etc]# hadoop fs -chown itcast /emptyfile
[root@hadoop0 etc]#
[root@hadoop0 etc]# hadoop fs -ls /emptyfile
Found 1 items
-rw-r--r--  1 itcast supergroup        0 2013-07-24 14:54 /emptyfile
```

图 4-24

上图中把文件/emptyfile 的属主由 root 修改为 itcast。

也可以同时修改属组，如图 4-25 所示。

```
[root@hadoop0 etc]# hadoop fs -chown itcast:itcast /emptyfile
[root@hadoop0 etc]#
[root@hadoop0 etc]# hadoop fs -ls /emptyfile
Found 1 items
-rw-r--r-- 1 itcast itcast 0 2013-07-24 14:54 /emptyfile
```

图 4-25

在图 4-25 中，把文件 /emptyfile 的属主和属组都修改为 itcast，如果只修改属组，可以使用 “: itcast”。

如果带有选项-R，意味着可以递归修改文件夹中的所有文件的属主、属组信息。

## ● -chgrp 修改属组

该命令的作用是修改文件的属组，该命令相当于“chown :属组”的用法，如图 4-26 所示。

```
[root@hadoop0 etc]# hadoop fs -chgrp supergroup /emptyfile
[root@hadoop0 etc]#
[root@hadoop0 etc]# hadoop fs -ls /emptyfile
Found 1 items
-rw-r--r-- 1 itcast supergroup 0 2013-07-24 14:54 /emptyfile
```

图 4-26

## ● -help 帮助

该命令选项会显示帮助信息，后面跟上需要查询的命令选项即可，如图 4-27 所示。

```
[root@hadoop0 etc]# hadoop fs -help rm
-rm [-skipTrash] <src>: Delete all files that match the specified file pattern.
      Equivalent to the Unix command "rm <src>""
      -skipTrash option bypasses trash, if enabled, and immediately
deletes <src>
```

图 4-27

在图 4-27 中，查询的 rm 的用法。

该命令选项显示的内容并非完全准确，比如查询 count 的结果就不准确，而是把所有命令选项的用法都显示出来，如图 4-28 所示。希望新的版本以后改进。

```
[root@hadoop0 etc]# hadoop fs -help count
hadoop fs is the command to execute fs commands. The full syntax is:

hadoop fs [-fs <local | file system URI>] [-conf <configuration file>]
           [-D <property=value>] [-ls <path>] [-lsr <path>] [-du <path>]
           [-dus <path>] [-mv <src> <dst>] [-cp <src> <dst>] [-rm [-skipTrash] <s
           [-rmr [-skipTrash] <src>] [-put <localsrc> ... <dst>] [-copyFromLocal <
]>]
           [-moveFromLocal <localsrc> ... <dst>] [-get [-ignoreCrc] [-crc] <src>
           [-getmerge <src> <localdst> [addnl]] [-cat <src>]
           [-copyToLocal [-ignoreCrc] [-crc] <src> <localdst>] [-moveToLocal <src>
           [-mkdir <path>] [-report] [-setrep [-R] [-w] <rep> <path/file>]
           [-touchz <path>] [-test -[ezd] <path>] [-stat [format] <path>]
           [-tail [-f] <path>] [-text <path>]
           [-chmod [-R] <MODE>... | OCTALMODE> PATH...]
           [-chown [-R] [OWNER] [:GROUP] PATH...]
           [-chgrp [-R] GROUP PATH...]
           [-count [-q] <path>]
           [-help [cmd]]]

-fs [local | <file system URI>]:          Specify the file system to use.
                                         If not specified, the current configuration is used,
                                         taken from the following, in increasing precedence:
                                         core-default.xml inside the hadoop jar file
                                         core-site.xml in $HADOOP_CONF_DIR
                                         'local' means use the local file system as your DFS.
                                         <file system URI> specifies a particular file system to
                                         contact. This argument is optional but if used must appear
```

图 4-28

读者短时间无法掌握全部命令用法，请重点掌握 ls(r)、rm(r)、mkdir、put、get 的使用。

## 4.3. HDFS 体系结构与基本概念

我们通过 hadoop shell 上传的文件是存放在 DataNode 的 block 中，通过 linux shell 是看不到文件的，只能看到 block。

可以一句话描述 HDFS：把客户端的大文件存放在很多节点的数据块中。在这里，出现了三个关键词：文件、节点、数据块。HDFS 就是围绕着这三个关键词设计的，我们在学习的时候也要紧抓住这三个关键词来学习。

### 4.3.1. NameNode

### 4.3.1.1. 作用

[NameNode 的作用](#)是管理文件目录结构，是管理数据节点的。名字节点维护两套数据，一套是文件目录与数据块之间的关系，[另一套](#)是数据块与节点之间的关系。前一套数据是静态的，是存放在磁盘上的，通过 `fsimage` 和 `edits` 文件来维护；后一套数据是动态的，不持久化到磁盘的，每当集群启动的时候，会自动建立这些信息。

### 4.3.1.2. 目录结构

既然 NameNode 维护了这么多的信息，那么这些信息都存放在哪里[呢](#)?在 hadoop 源代码中有个文件叫做 `core-default.xml`，如图 4-51 所示。

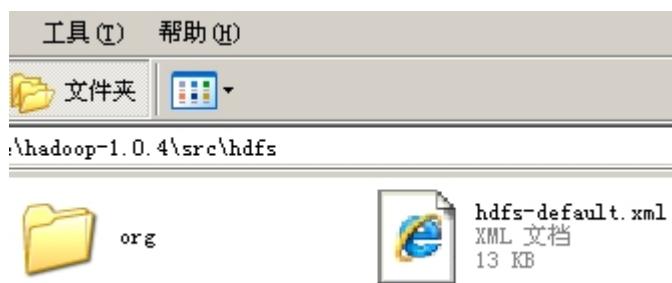


图 4-51

打开这个文件，在第 149 行和第 158 行，有两个配置信息，一个是 `dfs.name.dir`，[另](#)一个是 `dfs.name.edits.dir`。这两个文件表示的是 NameNode 的核心文件 `fsimage` 和 `edits` 的存放位置，如图 4-52 所示。

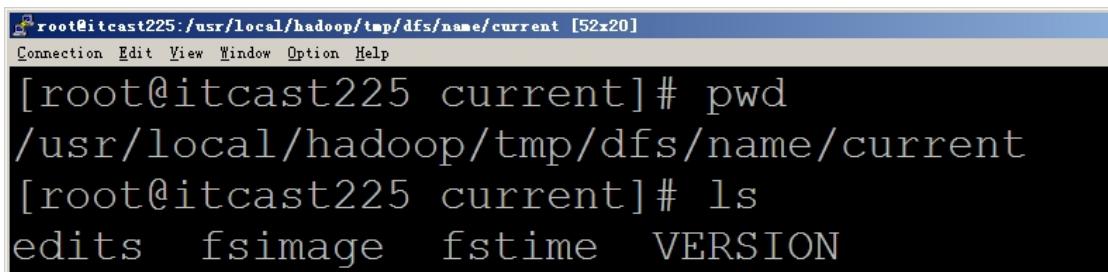
```
147
148<property>
149<name>dfs.name.dir</name>
150<value>${hadoop.tmp.dir}/dfs/name</value>
151<description>Determines where on the local filesystem the DFS name node
152    should store the name table(fsimage). If this is a comma-delimited list
153    of directories then the name table is replicated in all of the
154    directories, for redundancy. </description>
155</property>
156
157<property>
158<name>dfs.name.edits.dir</name>
159<value>${dfs.name.dir}</value>
160<description>Determines where on the local filesystem the DFS name node
161    should store the transaction (edits) file. If this is a comma-delimited list
162    of directories then the transaction file is replicated in all of the
163    directories, for redundancy. Default value is same as dfs.name.dir
164</description>
165</property>
```

The image shows a code editor window with the file 'hdfs-default.xml' open. The code is an XML configuration file for HDFS. Lines 149 and 158 are highlighted in blue, indicating they are the properties being discussed. Line 149 defines 'dfs.name.dir' with a value of '\${hadoop.tmp.dir}/dfs/name'. Line 158 defines 'dfs.name.edits.dir' with a value of '\${dfs.name.dir}'. Both lines have detailed descriptions explaining their purpose in determining where the DFS name node should store the name table and transaction file respectively.

图 4-52

在对应配置的 `value` 值有 \${ }，这是变量的表示方式，在程序读取文件时，会把变量的值读取出来。那么，第 150 行的变量 `hadoop.tmp.dir` 的值是在我们上一章的配置文件 `core-site.xml` 中配置的，值是 /usr/local/hadoop/tmp。可以看出，这两个文件的存储位置是在 linux 文件系统的 /usr/local/hadoop/tmp/dfs/name 目录下。

我们进入 linux 文件系统，可以看到如图 4-53 所示的目录结构。



```
[root@itcast225 current]# pwd  
/usr/local/hadoop/tmp/dfs/name/current  
[root@itcast225 current]# ls  
edits  fsimage  fstime  VERSION
```

图 4-53

### 4.3.2. DataNode

#### 4.3.2.1. 作用

DataNode 的作用—是 HDFS 中真正存储数据的。

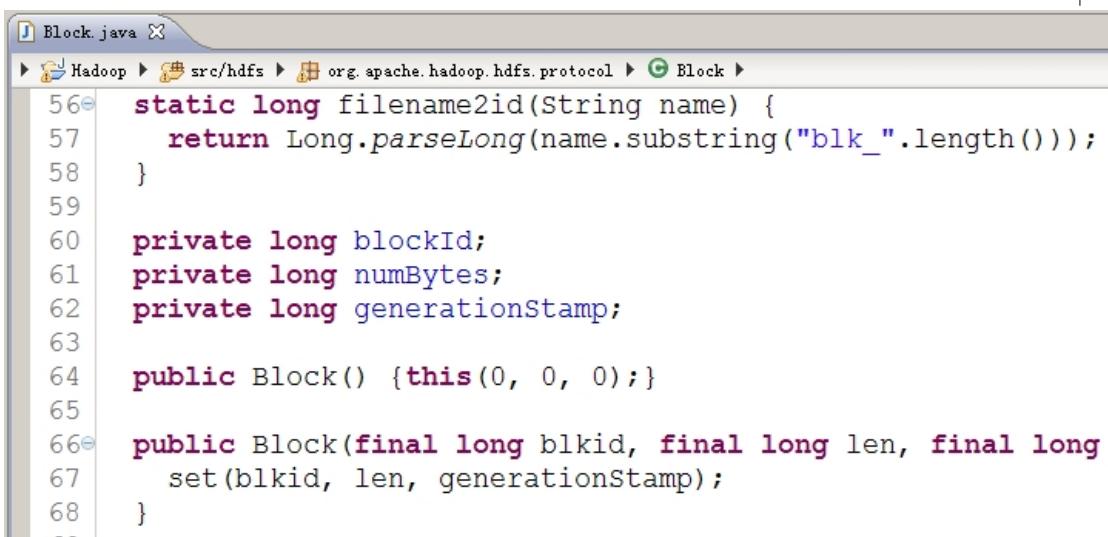
#### 4.3.2.2. block

如果一个文件非常大，比如 100GB，那么是怎么存储在 DataNode 中哪呢？DataNode 在存储数据的时候是按照 **block** 为单位读写数据的。**block** 是 hdfs 读写数据的基本单位。

假设文件大小是 100GB，从字节位置 0 开始，每 64MB 字节划分为一个 **block**，以此类推，可以划分出很多的 **block**。每个 **block** 就是 64MB 大小。

**block** 本质上是一个逻辑概念，意味着 **block** 里面不会真正的存储数据，只是划分文件的。

我们看一下 `org.apache.hadoop.hdfs.protocol.Block` 类，这里面的属性有以下几个，如图 4-54 所示。

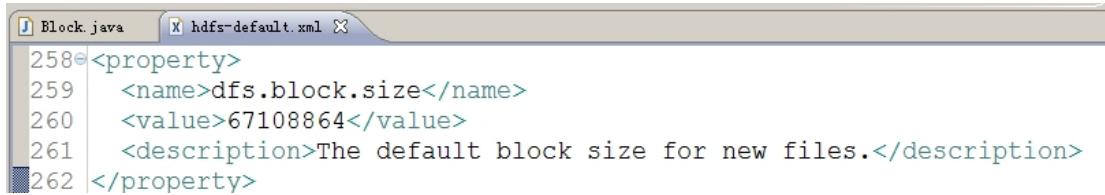


```
56  static long filename2id(String name) {  
57      return Long.parseLong(name.substring("blk_".length()));  
58  }  
59  
60  private long blockId;  
61  private long numBytes;  
62  private long generationStamp;  
63  
64  public Block() {this(0, 0, 0);}  
65  
66  public Block(final long blkid, final long len, final long  
67      set(blkid, len, generationStamp);  
68  }
```

图 4-54

类中的属性没有一个是可以存储数据的。

为什么一定要划分为 64MB 大小哪呢？因为这是在默认配置文件中设置的，我们查看 core-default.xml 文件，如图 4-55 所示。



```
258<property>
259  <name>dfs.block.size</name>
260  <value>67108864</value>
261  <description>The default block size for new files.</description>
262 </property>
```

图 4-55

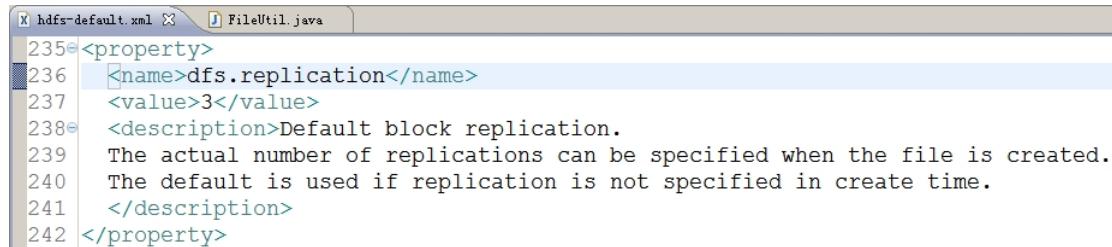
上图中的参数 `dfs.block.name` 指的就是 block 的大小，值是 67108864 字节，可以换算为 64MB。如果我们不希望使用 64MB 大小，可以在 `core-site.xml` 中覆盖该值。注意单位是字节。

### 4.3.2.3. 副本

副本就是备份，目的当时是为了安全。正是因为集群环境的不可靠，所以才使用副本机制来保证数据的安全性。

副本的缺点就是会占用大量的存储空间。副本越多，占用的空间越多。相比数据丢失的风险，存储空间的花费还是值得的。

那么，一个文件有几个副本合适哪呢？我们查看 `hdfs-default.xml` 文件，如图 4-551 所示。



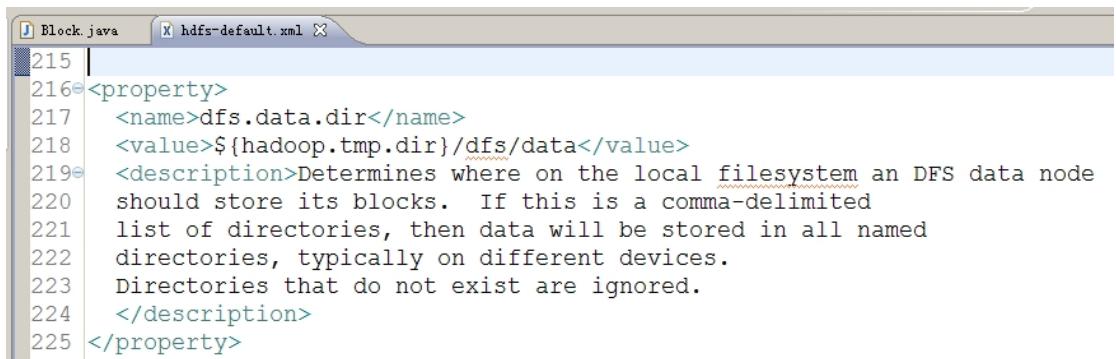
```
235<property>
236  <name>dfs.replication</name>
237  <value>3</value>
238  <description>Default block replication.
239  The actual number of replications can be specified when the file is created.
240  The default is used if replication is not specified in create time.
241 </description>
242 </property>
```

图 4-551

从图 4-551 中可以看到，默认的副本数量是 3。意味着 HDFS 中的每个数据块都有 3 份。当然，每一份肯定会尽力分配在不同的 DataNode 服务器中。试想：如果备份的 3 份数据都在同一台服务器上，那么这台服务器停机了，是不是所有的数据都丢了啊？

### 4.3.2.4. 目录结构

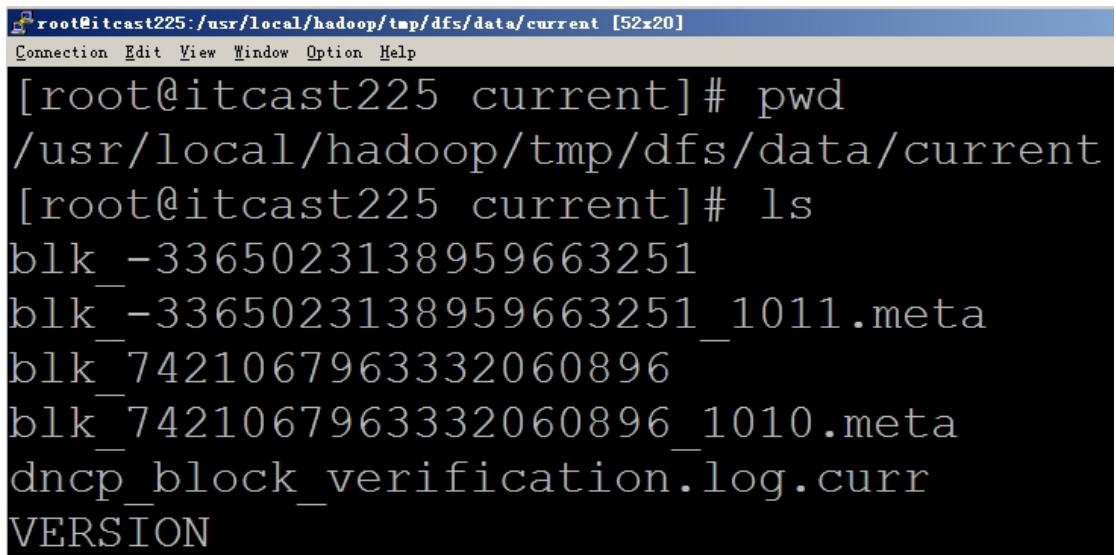
既然 DataNode 的 block 是划分文件并，那么划分后的文件到底存放在哪里哪？我们查看文件 `core-default.xml`，如图 4-56 所示。



```
215 |
216<property>
217  <name>dfs.data.dir</name>
218  <value>${hadoop.tmp.dir}/dfs/data</value>
219<description>Determines where on the local filesystem an DFS data node
220 should store its blocks. If this is a comma-delimited
221 list of directories, then data will be stored in all named
222 directories, typically on different devices.
223 Directories that do not exist are ignored.
224 </description>
225</property>
```

图 4-56

参数 `dfs.data.dir` 的值就是 `block` 存放在 `linux` 文件系统中的位置。变量 `hadoop.tmp.dir` 的值前面已经介绍了，是 `/usr/local/hadoop/tmp`，那么 `dfs.data.dir` 的完整路径是 `/usr/local/hadoop/tmp/dfs/data`。通过 `linux` 命令查看，结果如图 4-57 所示。



```
[root@itcast225 current]# pwd
/usr/local/hadoop/tmp/dfs/data/current [52x20]
Connection Edit View Window Option Help
[root@itcast225 current]# ls
blk_-3365023138959663251
blk_-3365023138959663251_1011.meta
blk_7421067963332060896
blk_7421067963332060896_1010.meta
dncp_block_verification.log.curr
VERSION
```

图 4-57

上图中以“`blk_`”开头的文件就是存储数据的 `block`。这里的命名是有规律的，除了 `block` 文件外，还有后缀是“`meta`”的文件，这是 `block` 的源数据文件，存放一些元数据信息。因此，上图中只有 2 个 `block` 文件。

注意：我们从 `linux` 磁盘上传一个完整的文件到 `hdfs` 中，这个文件在 `linux` 是可以看到的，但是上传到 `hdfs` 后，就不会有一个对应的文件存在，而是被划分成很多的 `block` 存在的。

### 4.3.3. SecondaryNameNode

SNN 只有一个职责，就是合并 `NameNode` 中的 `edits` 到 `fsimage` 中。

#### 4.3.3.1. 合并原理

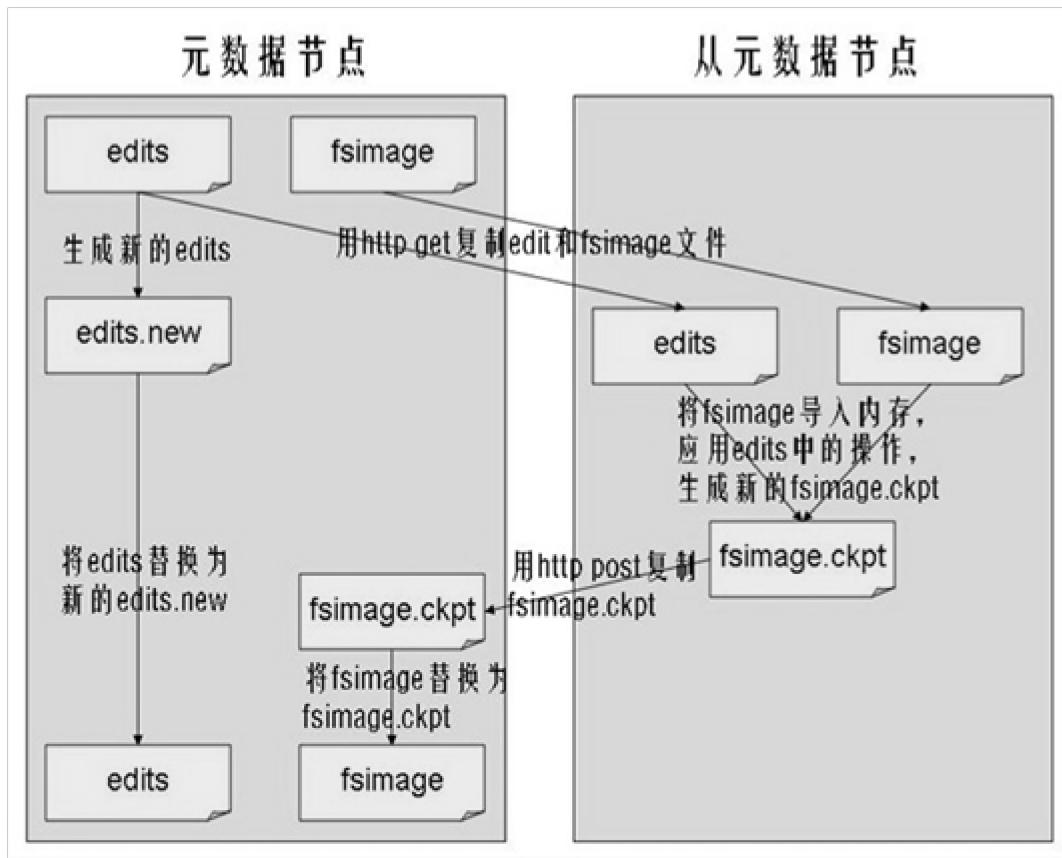


图 4-58

#### 4.4. HDFS 的 web 接口

HDFS 对外提供了可供访问的 http server，开放了很多端口，下面介绍常用的几个端口。

- 50070 端口，查看 NameNode 状态，如图 4-59 所示。

**NameNode 'hadoop0:9000'**

**Started:** Wed Jul 24 06:53:48 PDT 2013  
**Version:** 1.1.2, r1440782  
**Compiled:** Thu Jan 31 02:03:24 UTC 2013 by hortonfo  
**Upgrades:** There are no upgrades in progress.

[Browse the filesystem](#)  
[Namenode Logs](#)

### Cluster Summary

<b>39 files and directories, 12 blocks = 51 total. Heap Size</b>	:	
<b>Configured Capacity</b>	:	18.41 GB
<b>DFS Used</b>	:	168 KB
<b>Non DFS Used</b>	:	5.24 GB
<b>DFS Remaining</b>	:	13.17 GB
<b>DFS Used%</b>	:	0 %
<b>DFS Remaining%</b>	:	71.55 %
<b>Live Nodes</b>	:	1
<b>Dead Nodes</b>	:	0
<b>Decommissioning Nodes</b>	:	0
<b>Number of Under-Replicated Blocks</b>	:	2

图 4-59

该端口的定义位于 core-default.xml 中，如图 4-60 所示，读者可以在 core-site.xml 中自行修改。

```
60<property>
61  <name>dfs.http.address</name>
62  <value>0.0.0.0:50070</value>
63<description>
64  The address and the base port where the dfs namenode web ui will listen on.
65  If the port is 0 then the server will start on a free port.
66</description>
67</property>
```

图 4-60

如果读者通过该端口看着这个页面，以为着 NameNode 节点是存活的。

- 50075 端口，查看 DataNode 的，如图 4-61 所示。

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
<a href="#">abc</a>	dir				2013-07-24 09:17	rwxr-xr-x	root	supergroup
<a href="#">emptyfile</a>	file	0 KB	1	64 MB	2013-07-24 14:54	rw-r--r--	itcast	supergroup
<a href="#">hbase</a>	dir				2013-07-24 06:54	rwxr-xr-x	root	supergroup
<a href="#">hbase-env.sh</a>	file	6.03 KB	1	64 MB	2013-07-24 07:48	rw-r--r--	root	supergroup
<a href="#">install.log</a>	file	36.76 KB	2	64 MB	2013-07-24 07:52	rw-r--r--	root	supergroup
<a href="#">user</a>	dir				2013-07-24 14:39	rwx-----	root	supergroup
<a href="#">usr</a>	dir				2013-07-24 06:54	rwxr-xr-x	root	supergroup

图 4-61

该地址和端口的定义位于 hdfs-default.xml 中, 如图 4-62 所示, 读者可以在 hdfs-site.xml 中自行修改。

```

36<property>
37  <name>dfs.datanode.http.address</name>
38  <value>0.0.0.0:50075</value>
39<description>
40      The datanode http server address and port.
41      If the port is 0 then the server will start on a free port.
42  </description>
43</property>
44

```

图 4-62

- 50090 端口, 查看 SecondaryNameNode 的
- 50030 端口, 查看 JobTracker 状态的, 如图 4-63 所示。

**hadoop0 Hadoop Map/Reduce Administration**

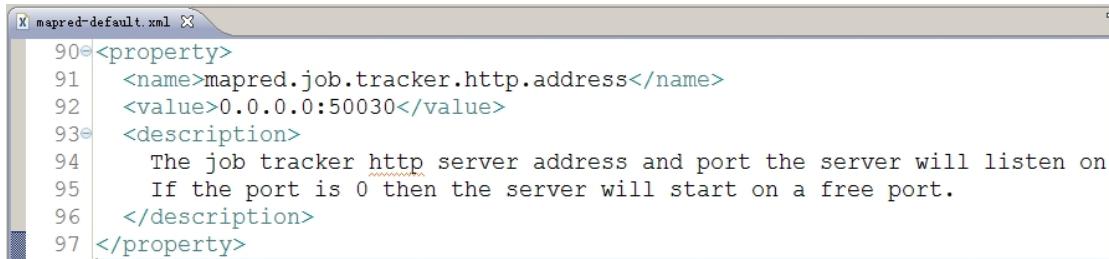
**State:** RUNNING  
**Started:** Wed Jul 24 15:40:49 PDT 2013  
**Version:** 1.1.2, r1440782  
**Compiled:** Thu Jan 31 02:03:24 UTC 2013 by hortonfo  
**Identifier:** 201307241540  
**SafeMode:** OFF

**Cluster Summary (Heap Size is 7.56 MB/966.69 MB)**

Running Map Tasks	Running Reduce Tasks	Total Submissions	Nodes	Occupied Map Slots	Occupied Reduce Slots	Reserved Map Slots	Reserved Reduce Slots	Map Task Capacity
0	0	0	1	0	0	0	0	2

图 4-63

该端口定义在 mapred-default.xml 中，如图 4-64，读者可以在 mapred-site.xml 中自行修改



```
90<property>
91    <name>mapred.job.tracker.http.address</name>
92    <value>0.0.0.0:50030</value>
93    <description>
94        The job tracker http server address and port the server will listen on
95        If the port is 0 then the server will start on a free port.
96    </description>
97</property>
```

图 4-64

- 50060 端口，查看 TaskTracker，如图 4-65 所示。

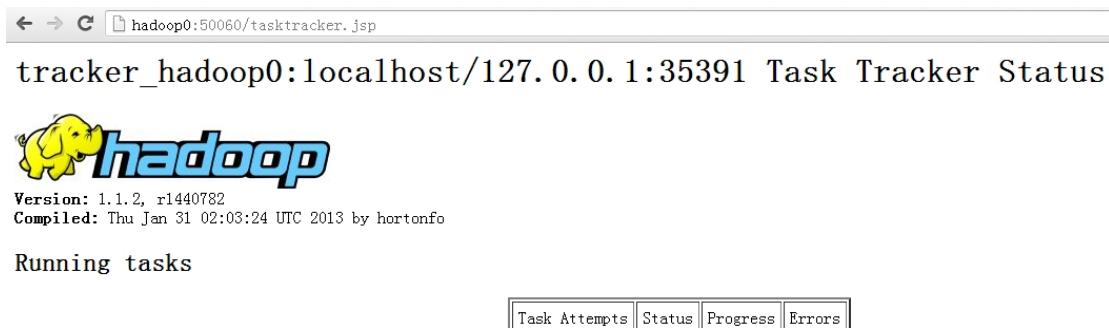
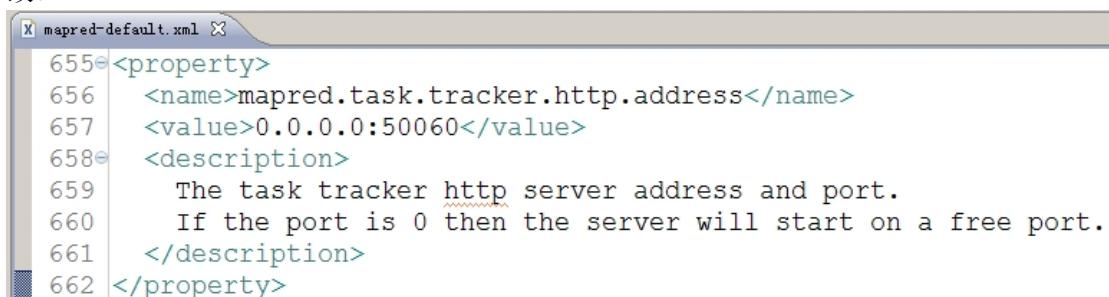


图 4-65

该端口定义位于 mapred-default.xml，如图 4-66，读者可以在 mapred-site.xml 中自行修改。



```
655<property>
656    <name>mapred.task.tracker.http.address</name>
657    <value>0.0.0.0:50060</value>
658    <description>
659        The task tracker http server address and port.
660        If the port is 0 then the server will start on a free port.
661    </description>
662</property>
```

图 4-66

## 4.5. HDFS 的 java 访问接口

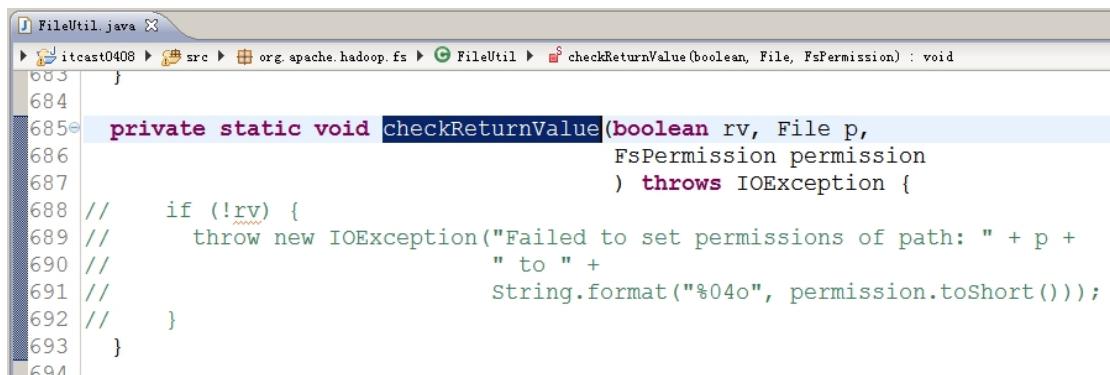
### 4.5.1. 搭建 Hadoop 开发环境

我们在工作中写完的各种代码是在服务器中运行的，HDFS 的操作代码也不例外。在开发阶段，我们使用 windows 下的 eclipse 作为开发环境，访问运行在虚拟机中的 HDFS。也就

是通过在本地的 eclipse 中的 java 代码访问远程 linux 中的 hdfs。

要使用宿主机中的 java 代码访问客户机中的 hdfs，需要保证以下几点：

- 确保宿主机与客户机的网络是互通的
- 确保宿主机和客户机的防火墙都关闭，因为很多端口需要通过，为了减少防火墙配置，直接关闭
- 确保宿主机与客户机使用的 jdk 版本一致。如果客户机使用 jdk6，宿主机使用 jdk7，那么代码运行时会报不支持的版本的错误
- 宿主机的登录用户名必须与客户机的用户名一致。比如我们 linux 使用的是 root 用户，那么 windows 也要使用 root 用户，否则会报权限异常
- 在 eclipse 项目中覆盖 hadoop 的 org.apache.hadoop.fs.FileUtil 类的 checkReturnValue 方法，如图 4-661，目的是为了避免权限错误



```
FileUtil.java
private static void checkReturnValue(boolean rv, File p,
                                      FsPermission permission
) throws IOException {
    if (!rv) {
        throw new IOException("Failed to set permissions of path: " + p +
                              " to " +
                              String.format("%04o", permission.toShort()));
    }
}
```

图 4-661

如果读者在开发过程中出现权限等问题，请按照本节的提示检查自己的环境。

## 4.5.2. 使用 FileSystem api 读写数据

在 hadoop 的 HDFS 操作中，有个非常重要的 api，是 org.apache.hadoop.fs.FileSystem，这是我们用户代码操作 HDFS 的直接入口，该类含有操作 HDFS 的各种方法，类似于 jdbc 中操作数据库的直接入口是 Connection 类。

那我们怎么获得一个 FileSystem 对象哪？

```
String uri = "hdfs://192.168.1.240:9000/";
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(URI.create(uri), configuration);
```

以上代码中，要注意调用的是 FileSystem 的静态方法 get，传递两个值给形式参数，第一个访问的 HDFS 地址，该地址的协议是 hdfs，ip 是 192.168.1.240，端口是 9000。这个地址的完整信息是在配置文件 core-site.xml 中指定的，读者可以使用自己环境的配置文件中的设置。第二个参数是一个配置对象。

### 4.5.2.1. 创建文件夹

使用 HDFS 的 shell 命令查看一下根目录下的文件情况，如图 4-67 所示。

```
[root@hadoop0 ~]# hadoop fs -ls /
Found 2 items
drwxr-xr-x  - root supergroup          0 2013-07-24 15:35 /hbase
drwxr-xr-x  - root supergroup          0 2013-07-25 15:21 /tmp
```

图 4-67

我们在 HDFS 的根目录下创建文件夹，代码如下

```
final String pathString = "/d1";
boolean exists = fs.exists(new Path(pathString));
if(!exists){
    boolean result = fs.mkdirs(new Path(pathString));
    System.out.println(result);
}
```

以上代码中，我们决定创建的文件夹完整路径是“/d1”。第二行代码是使用方法 `existst` 判断文件夹是否存在；如果不存在，执行创建操作。创建文件夹，调用的是 `mkdirs` 方法，返回值是布尔值，如果是 `true`，表示创建成功；如果是 `false`，表示创建失败。

现在查看一下是否成功了，如图 4-68，可见创建成功了。

```
[root@hadoop0 ~]# hadoop fs -ls /
Found 3 items
drwxr-xr-x  - root supergroup          0 2013-07-25 18:52 /d1
drwxr-xr-x  - root supergroup          0 2013-07-24 15:35 /hbase
drwxr-xr-x  - root supergroup          0 2013-07-25 15:21 /tmp
```

图 4-68

### 4.5.2.2. 写文件

我们可以向 HDFS 写入文件，代码如下：

```
final String pathString = "/d1/f1";
final FSDataOutputStream fsDataOutputStream = fs.create(new Path(pathString));
IOUtils.copyBytes(new ByteArrayInputStream("my name is WU CHAO".getBytes()),
fsDataOutputStream, configuration, true);
```

第一行代码表示创建的文件是在刚才创建的 `d1` 文件夹下的文件 `f1`；

第二行是调用 `create` 方法创建一个通向 HDFS 的输出流；

第三行是通过调用 `hadoop` 的一个工具类 `IOUtils` 的静态方法 `copyBytes` 把一个字符串发送给输出流中。该静态方法有四个参数，第一个参数输入流，第二个参数是输出流，第三个参数是配置对象，第四个参数是布尔值，如果是 `true` 表示数据传输完毕后关闭流。

现在看一下是否创建成功了，如图 4-69 所示。

```
[root@hadoop0 ~]# hadoop fs -text /d1/f1
my name is WU CHAO[root@hadoop0 ~]#
```

图 4-69

### 4.5.2.3. 读文件

现在我们把刚才写入到 HDFS 的文件 “/d1/f1” 读出来，代码如下：

```
final String pathString = "/d1/f1";
final FSDataInputStream fsDataInputStream = fs.open(new Path(pathString));
IOUtils.copyBytes(fsDataInputStream, System.out, configuration, true);
```

第二行表示调用方法 open 打开一个指定的文件，返回值是一个通向该文件的输入流；

第三行还是调用 IOUtils.copyBytes 方法，输出的目的地是控制台。见图 4-70

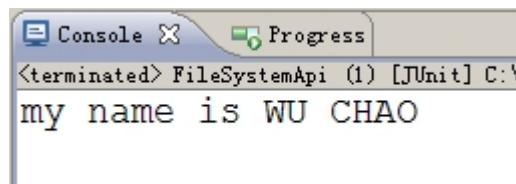


图 4-70

### 4.5.2.4. 查看目录列表和文件详细信息

我们可以把根目录下的所有文件和目录显示出来，代码如下

```
final String pathString = "/";
final FileStatus[] listStatus = fs.listStatus(new Path(pathString));
for (FileStatus fileStatus : listStatus) {
    final String type = fileStatus.isDir()?"目录":"文件";
    final short replication = fileStatus.getReplication();
    final String permission = fileStatus.getPermission().toString();
    final long len = fileStatus.getLength();
    final Path path = fileStatus.getPath();
    System.out.println(type+"\t"+permission+"\t"+replication+"\t"+len+"\t"+path);
}
```

调用listStatus方法会得到一个指定路径下的所有文件和文件夹，每一个用FileStatus表示。我们使用for循环显示每一个FileStatus对象。FileStatus对象表示文件的详细信息，里面含有类型、副本数、权限、长度、路径等很多信息，我们只是显示了一部分。结果如图4-71所示。

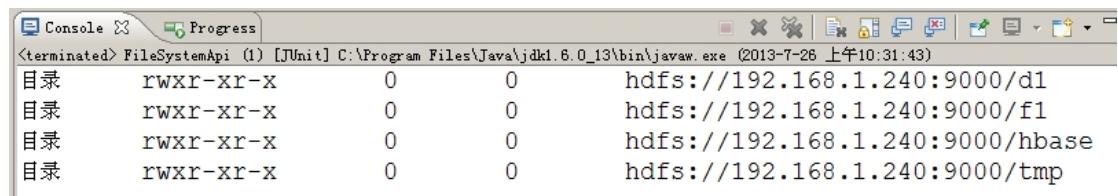


图 4-71

### 4.5.2.5. 删 除文件或目录

我们可以删除某个文件或者路径，代码如下

```
final String pathString = "/d1/f1";
//fs.delete(new Path("/d1"), true);
fs.deleteOnExit(new Path(pathString));
```

第三行代码表示删除文件“/d1/f1”，注释掉的第二行代码表示递归删除目录“/d1”及下面的所有内容。

除了上面列出的 fs 的方法外，还有很多方法，请读者自己查阅 api。

## 4.6. HDFS 的 RPC 机制

RPC 是远程过程调用(Remote Procedure Call)，即远程调用其他虚拟机中运行的 java object。RPC 是一种客户端/服务器模式，那么在使用时包括服务端代码和客户端代码，还有我们调用的远程过程对象。

HDFS 的运行就是建立在此基础之上的。本章通过分析实现一个简单的 RPC 程序来分析 HDFS 的运行机理。本节难度偏大，读者可以在第二、三遍阅读本书时掌握即可。

下面的代码是服务端代码—。

```
public class MyServer {
    public static final int SERVER_PORT = 12345;
    public static final String SERVER_ADDRESS = "localhost";
    public static void main(String[] args) throws IOException {
        final Server server = RPC.getServer(new MyBiz(), SERVER_ADDRESS,
            SERVER_PORT, new Configuration());
        server.start();
    }
}
```

核心在于第 5 行的 RPC.getServer 方法，该方法有四个参数，第一个参数是被调用的 java 对象，第二个参数是服务器的地址，第三个参数是服务器的端口。获得服务器对象后，启动服务器。这样，服务器就在指定端口监听客户端的请求。

下面的代码是被调用的远程对象类—。

```
public class MyBiz implements MyBizable{
    public static long BIZ_VERSION = 2345234L;
    @Override
    public String hello(String name){
        System.out.println("我被调用了");
        return "hello "+name;
    }

    @Override
    public long getProtocolVersion(String protocol, long clientVersion)
```

```
    throws IOException {
        return BIZ_VERSION;
    }
}
```

---

被调用的远程对象实现了接口 MyBizable，这里面有两个方法被实现，一个就是 hello 方法，**另一个是 getProtocolVersion 方法**。这个 hello 方法内部有个输出语句。

下面的代码是远程调用类的接口定义。

```
public interface MyBizable extends VersionedProtocol{
    public abstract String hello(String name);
}
```

---

这个接口中的方法就是刚才的 Biz 中实现的方法。接口继承的 VersionedProtocol，是 hadoop 的 RPC 的接口，所有的 RPC 通信必须实现这个一接口，用于保证客户端和服务端的端口一致。服务端被调用的类必须继承这个接口 VersionedProtocol。

下面是客户端代码，这里使用的调用对象的接口。

```
public class MyClient {
    public static void main(String[] args) throws Exception {
        final MyBizable proxy = (MyBizable)RPC.getProxy(MyBizable.class,
MyBiz.BIZ_VERSION, new InetSocketAddress(MyServer.SERVER_ADDRESS,
MyServer.SERVER_PORT), new Configuration());
        //调用接口中的方法
        final String result = proxy.hello("world");
        System.out.println(result);
        //本质是关闭网络连接
        RPC.stopProxy(proxy);
    }
}
```

---

以上代码中核心在于 RPC.getProxy()，该方法有四个参数，第一个参数是被调用的接口类，第二个是客户端版本号，**第三个是服务端地址**。返回的代理对象，就是服务端对象的代理，内部就是使用 java.lang.Proxy 实现的。

运行时，先启动服务端，再启动客户端。读者可以服务端和客户端输出信息。

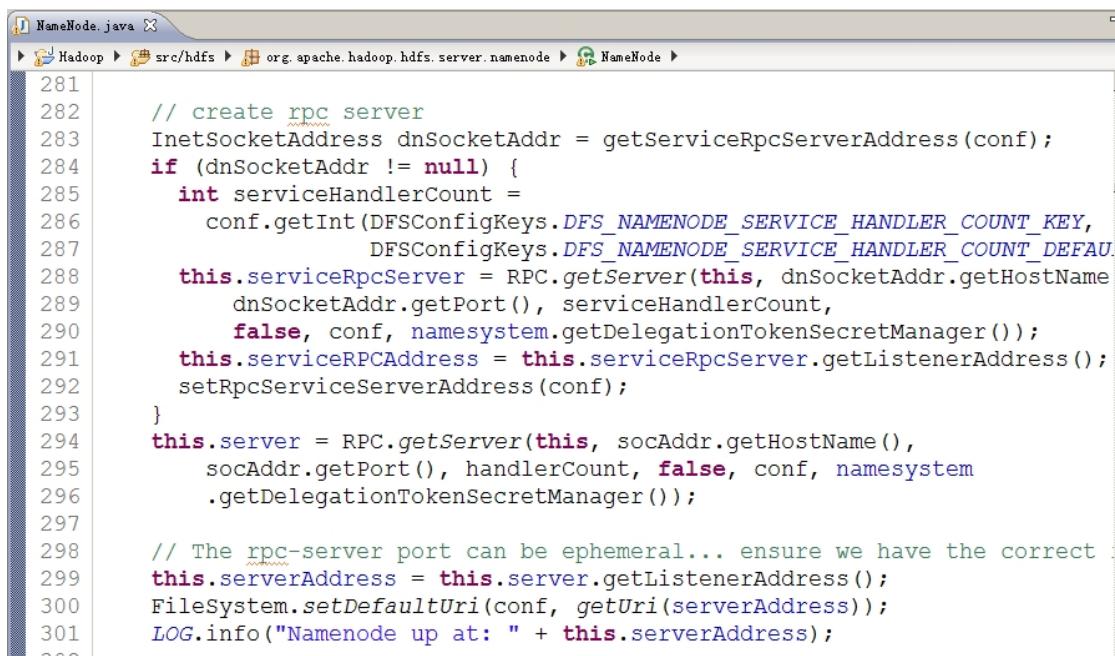
从上面的 RPC 调用中，可以看出：在客户端调用的业务类的方法是定义在业务类的接口中的。该接口实现了 VersionedProtocol 接口。

现在我们在命令行执行 jps 命令，查看输出信息，如图 5-1 所示。

```
C:\Documents and Settings\Administrator>jps
7180 Jps
7044 MyServer
5904
```

图 5-1

可以看到一个 java 进程，是“MyServer”，该进程正是我们刚刚运行的 rpc 的服务端类 MyServer。大家可以联想到我们搭建 hadoop 环境时，也执行过该命令用来判断 hadoop 的进程是否全部启动，[如图 3](#)。那么可以判断，hadoop 启动时产生的 5 个 java 进程也应该是 RPC 的服务端。我们观察 NameNode 的源代码，如图 5-2，可以看到 NameNode 确实创建了 RPC 的服务端。



```

281
282     // create rpc server
283     InetSocketAddress dnSocketAddr = getServiceRpcServerAddress(conf);
284     if (dnSocketAddr != null) {
285         int serviceHandlerCount =
286             conf.getInt(DFSConfigKeys.DFS_NAMENODE_SERVICE_HANDLER_COUNT_KEY,
287                         DFSConfigKeys.DFS_NAMENODE_SERVICE_HANDLER_COUNT_DEFAULT);
288         this.serviceRpcServer = RPC.getServer(this, dnSocketAddr.getHostName(),
289                                               dnSocketAddr.getPort(), serviceHandlerCount,
290                                               false, conf, namesystem.getDelegationTokenSecretManager());
291         this.serviceRPCAddress = this.serviceRpcServer.getListenerAddress();
292         setRpcServiceServerAddress(conf);
293     }
294     this.server = RPC.getServer(this, socAddr.getHostName(),
295                                 socAddr.getPort(), handlerCount, false, conf, namesystem
296                                 .getDelegationTokenSecretManager());
297
298     // The rpc-server port can be ephemeral... ensure we have the correct
299     // server address
300     this.serverAddress = this.server.getListenerAddress();
301     FileSystem.setDefaultUri(conf, getUri(serverAddress));
302     LOG.info("Namenode up at: " + this.serverAddress);
303
304 }
```

图 5-2

## 4.7. NameNode 的接口分析

由 5.1 节分析，可以看到 NameNode 本身就是一个 java 进程。观察图 5-2 中 `RPC.getServer()` 方法的第一个参数，发现是 `this`，说明 NameNode 本身就是一个位于服务端的被调用对象，即 NameNode 中的方法是可以被客户端代码调用的。根据 RPC 运行原理可知，NameNode 暴露给客户端的方法是位于接口中的。

我们查看 NameNode 的源码，如图 5-3 所示。

```
126  ****
127 public class NameNode implements ClientProtocol, DatanodeProtocol,
128             NamenodeProtocol, FSConstants,
129             RefreshAuthorizationPolicyProtocol,
130             RefreshUserMappingsProtocol {
131     static{
132         Configuration.addDefaultResource("hdfs-default.xml");
133         Configuration.addDefaultResource("hdfs-site.xml");
134     }
135
136     public long getProtocolVersion(String protocol,
137             long clientVersion) throws IOException {
138         if (protocol.equals(ClientProtocol.class.getName())) {
139             return ClientProtocol.versionID;
140         } else if (protocol.equals(DatanodeProtocol.class.getName())){
141             return DatanodeProtocol.versionID;
142         } else if (protocol.equals(NamenodeProtocol.class.getName())){
143             return NamenodeProtocol.versionID;
144         } else if (protocol.equals(RefreshAuthorizationPolicyProtocol.class.getName())){
145             return RefreshAuthorizationPolicyProtocol.versionID;
146         } else if (protocol.equals(RefreshUserMappingsProtocol.class.getName())){
147             return RefreshUserMappingsProtocol.versionID;
148     }
149 }
```

图 5-3

可以看到 NameNode 实现了 ClientProtocol、DatanodeProtocol、NamenodeProtocol 等接口。下面我们逐一分析这些接口。

## ● DFSClient 调用 ClientProtocol

- 这个接口是供客户端调用的。这里的客户端不是指的我们自己写的代码，而是 hadoop 的一个类叫做 DFSClient。在 DFSClient 中会调用 ClientProtocol 中的方法，完成一些操作。
- 该接口中的方法大部分是对 HDFS 的操作，如 create、delete、mkdirs、rename 等。

## ● DataNode 调用 DatanodeProtocol

- 这个接口是供 DataNode 调用的。DataNode 调用该接口中的方法向 NameNode 报告本节点的状态和 block 信息。
- NameNode 不能向 DataNode 发送消息，只能通过该接口中方法的返回值向 DataNode 传递消息。

## ● SecondaryNameNode 调用 NamenodeProtocol

- ——这个接口是供 SecondaryNameNode 调用的。SecondaryNameNode 是专门做 NameNode 中 edits 文件向 fsimage 合并数据的。

## 4.8. DataNode 的接口分析

按照分析 NameNode 的思路，看一下 DataNode 的源码接口，如图 5-4 所示。

```
164 public class DataNode extends Configured
165     implements InterDatanodeProtocol, ClientDatanodeProtocol, FSConstants,
166     Runnable, DataNodeMXBean {
167     public static final Log LOG = LogFactory.getLog(DataNode.class);
168
169     static{
170         Configuration.addDefaultResource("hdfs-default.xml");
171         Configuration.addDefaultResource("hdfs-site.xml");
172     }

```

图 5-4

这里有两个接口，分别是 `InterDatanodeProtocol`、`ClientDatanodeProtocol`。这里就不展开分析了。读者可以根据上面的思路独立分析。

## 4.9. HDFS 的写数据过程分析

我们通过 `FileSystem` 类可以操控 HDFS，那我们就从这里开始分析写数据到 HDFS 的过程。

在我们向 HDFS 写文件的时候，调用的是 `FileSystem.create(Path path)` 方法，我们查看这个方法的源码，通过跟踪内部的重载方法，可以找到如图 5-5 所示的调用。

```
573     public abstract FSDataOutputStream create(Path f,
574             FsPermission permission,
575             boolean overwrite,
576             int bufferSize,
577             short replication,
578             long blockSize,
579             Progressable progress) throws IOException;
```

图 5-5

这个方法是抽象类，没有实现。那么我们只能向他的子类寻找实现。`FileSystem` 有个子类是 `DistributedFileSystem`，在我们的伪分布环境下使用的就是这个类。我们可以看到 `DistributedFileSystem` 的这个方法的实现，如图 5-6 所示。

```
179
180     public FSDataOutputStream create(Path f, FsPermission permission,
181             boolean overwrite, int bufferSize, short replication,
182             long blockSize, Progressable progress) throws IOException {
183
184         statistics.incrementWriteOps(1);
185         return new FSDataOutputStream(dfs.create(getPathName(f), permission,
186             overwrite, true, replication, blockSize, progress, bufferSize),
187             statistics);
188     }

```

图 5-6

在图 5-6 中，注意第 185 行的返回值 `FSDataOutputStream`。这个返回值对象调用了自己的构造方法，构造方法的第一个参数是 `dfs.create()` 方法。我们关注一下这里的 `dfs` 对象是谁，`create` 方法做了什么事情。现在进入这个方法的实现，如图 5-7 所示。

```
FileSystem.java DistributedFileSystem.java DFSClient.java
Hadoop src/hdfs org.apache.hadoop.hdfs DFSClient recoverLease(String) : boolean
698 public OutputStream create(String src,
699             FsPermission permission,
700             boolean overwrite,
701             boolean createParent,
702             short replication,
703             long blockSize,
704             Progressable progress,
705             int bufferSize
706             ) throws IOException {
707     checkOpen();
708     if (permission == null) {
709         permission = FsPermission.getDefault();
710     }
711     FsPermission masked = permission.applyUMask(FsPermission.getUMask(conf));
712     LOG.debug(src + ": masked=" + masked);
713     OutputStream result = new DFSOutputStream(src, masked,
714         overwrite, createParent, replication, blockSize, progress, bufferSize,
715         conf.getInt("io.bytes.per.checksum", 512));
716     leasechecker.put(src, result);
717     return result;
718 }
```

图 5-7

在图 5-7 中，返回值正是第 713 行创建的对象。这个类有什么神奇的地方吗？我们看一下他的源码，如图 5-8 所示。

```
FileSystem.java DistributedFileSystem.java DFSClient.java
Hadoop src/hdfs org.apache.hadoop.hdfs DFSClient DFSOutputStream String int Progressable LocatedBlock HdfsFileStatus
3237 * Create a new output stream to the given DataNode.
3238 * @see ClientProtocol#create(String, FsPermission, String, boolean, short, long)
3239 */
3240 DFSOutputStream(String src, FsPermission masked, boolean overwrite,
3241     boolean createParent, short replication, long blockSize, Progressable progress,
3242     int bufferSize, int bytesPerChecksum) throws IOException {
3243     this(src, blockSize, progress, bytesPerChecksum, replication);
3244
3245     computePacketChunkSize(writePacketSize, bytesPerChecksum);
3246
3247     try {
3248         namenode.create(
3249             src, masked, clientName, overwrite, createParent, replication, blockSize);
3250     } catch(RemoteException re) {
3251         throw re.unwrapRemoteException(AccessControlException.class,
3252                                         FileAlreadyExistsException.class,
3253                                         FileNotFoundException.class,
3254                                         NSQuotaExceededException.class,
3255                                         DSQuotaExceededException.class);
3256     }
3257     streamer.start();
3258 }
3259 }
```

图 5-8

在图 5-8 中，可以看到，这个类是 DFSClient 的内部类。在类内部通过调用 namenode.create()方法创建了一个输出流。我们再看一下 namenode 对象是什么类型，如图 5-9 所示。

```
68 *
69 * Hadoop DFS users should obtain an instance of
70 * DistributedFileSystem, which uses DFSClient to handle
71 * filesystem tasks.
72 *
73 ****
74 public class DFSClient implements FSConstants, java.io.Closeable {
75     public static final Log LOG = LogFactory.getLog(DFSClient.class);
76     public static final int MAX_BLOCK_ACQUIRE_FAILURES = 3;
77     private static final int TCP_WINDOW_SIZE = 128 * 1024; // 128 KB
78     public final ClientProtocol namenode;
79     private final ClientProtocol rpcNamenode;
```

图 5-9

在图 5-9 中，可以看到 namenode 其实是 ClientProtocol 接口。那么，这个对象是什么时候创建的那？如图 5-10 所示。

```
209 */
210 DFSClient(InetSocketAddress nameNodeAddr, ClientProtocol rpcNamenode,
211             Configuration conf, FileSystem.Statistics stats)
212     throws IOException {
213     this.conf = conf;
214     this.stats = stats;
215     this.nnAddress = nameNodeAddr;
216     this.socketTimeout = conf.getInt("dfs.socket.timeout",
217                                     HdfsConstants.READ_TIMEOUT);
218     this.datanodeWriteTimeout = conf.getInt("dfs.datanode.socket.write.timeout",
219                                             HdfsConstants.WRITE_TIMEOUT);
220     this.timeoutValue = this.socketTimeout;
221     this.socketFactory = NetUtils.getSocketFactory(conf, ClientProtocol.class);
222     // dfs.write.packet.size is an internal config variable
223     this.writePacketSize = conf.getInt("dfs.write.packet.size", 64*1024);
224     this.maxBlockAcquireFailures = getMaxBlockAcquireFailures(conf);
225
226     ugi = UserGroupInformation.getCurrentUser();
227
228     String taskId = conf.get("mapred.task.id");
229     if (taskId != null) {
230         this.clientName = "DFSClient_" + taskId;
231     } else {
232         this.clientName = "DFSClient_" + r.nextInt();
233     }
234     defaultBlockSize = conf.getLong("dfs.block.size", DEFAULT_BLOCK_SIZE);
235     defaultReplication = (short) conf.getInt("dfs.replication", 3);
236
237     if (nameNodeAddr != null && rpcNamenode == null) {
238         this.rpcNamenode = createRPCNamenode(nameNodeAddr, conf, ugi);
239         this.namenode = createNamenode(this.rpcNamenode);
```

图 5-10

可以，namenode 对象是在 DFSClient 的构造函数调用时创建的，即当 DFSClient 对象存在的时候，namenode 对象已经存在了。

至此，我们可以看到，使用 FileSystem 对象的 api 操纵 HDFS，其实是通过 DFSClient 对象访问 NameNode 中的方法操纵 HDFS 的。这里的 DFSClient 是 RPC 机制的客户端，NameNode 是 RPC 机制的服务端的调用对象，整个调用过程如图 5-11 所示。

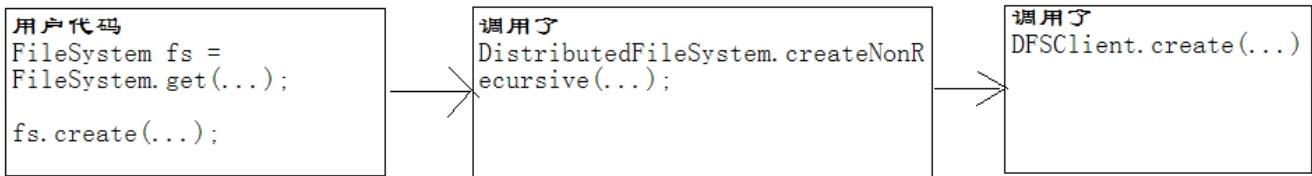


图 5-11

在整个过程中，`DFSClient` 是个很重要的类，从名称就可以看出，他表示 HDFS 的 Client，是整个 HDFS 的 RPC 机制的客户端部分。我们对 HDFS 的操作，是通过 `FileSsytem` 调用的 `DFSClient` 里面的方法。`FileSystem` 是封装了对 `DFSClient` 的操作，提供给用户使用的。

## 4.10. HDFS 的读数据过程分析

我们继续在 `FileSystem` 类分析，读数据使用的是 `open(...)`方法，我们可以看到源码，如图 5-12 所示。

```

153
154     public FSDataInputStream open(Path f, int bufferSize) throws IOException {
155         statistics.incrementReadOps(1);
156         return new DFSClent.DFSDataInputStream(dfs.open(getPathName(f),
157             bufferSize, verifyChecksum, statistics));
158     }
159

```

图 5-12

在图 5-12 中，返回的是 `DFSClient` 类中 `DFSDataInputStream` 类，显而易见，这是一个内部类。这个内部类的构造函数，有两个形参，第一个参数是 `dfs.open(...)`创建的对象。我们看一下方法的源码，如图 5-13 所示。

```

567 /**
568 * Create an input stream that obtains a nodelist from the
569 * namenode, and then reads from all the right places. Creates
570 * inner subclass of InputStream that does the right out-of-band
571 * work.
572 */
573 public DFSDataInputStream open(String src, int bufferSize, boolean verifyChecksum,
574     FileSystem.Statistics stats
575 ) throws IOException {
576     checkOpen();
577     // Get block info from namenode
578     return new DFSDataInputStream(src, bufferSize, verifyChecksum);
579 }

```

图 5-13

在图 5-13 的实现中，返回的是一个 `DFSDataInputStream` 对象。该对象中含有 `NameNode` 中的数据块信息。我们看一下这个类的构造方法源码，如图 5-14 所示。

```
1828     DFSInputStream(String src, int buffersize, boolean verifyChecksum
1829                     ) throws IOException {
1830         this.verifyChecksum = verifyChecksum;
1831         this.buffersize = buffersize;
1832         this.src = src;
1833         prefetchSize = conf.getLong("dfs.read.prefetch.size", prefetchSize);
1834         openInfo();
1835     }
1836
1837     /**
1838      * Grab the open-file info from namenode
1839     */
1840     synchronized void openInfo() throws IOException {
1841         LocatedBlocks newInfo = callGetBlockLocations(namenode, src, 0, prefetchSize
1842             );
1843         if (newInfo == null) {
1844             throw new FileNotFoundException("File does not exist: " + src);
1845         }
1846     }

```

图 5-14

在图 5-14 中，这个构造方法中最重要的语句是第 1834 行，打开信息，从第 1840 行开始是 openInfo() 的源代码，截图显示不全。注意第 1841 行，是获取数据块的信息的。我们查看这一行的源代码，如图 5-15 所示。

```
533     }
534
535     static LocatedBlocks callGetBlockLocations(ClientProtocol namenode,
536         String src, long start, long length) throws IOException {
537         try {
538             return namenode.getBlockLocations(src, start, length);
539         } catch (RemoteException re) {
540             throw re.unwrapRemoteException(AccessControlException.class,
541                                         FileNotFoundException.class);
542         }
543     }

```

图 5-15

从图 5-15 中可以看到，获取数据块信息的方法也是通过调用 namenode 取得的。这里的 namenode 属性还是位于 DFSClient 中的。通过前面的分析，我们已经知道，在 DFSClient 类中的 namenode 属性是 ClientProtocol。

至此，读者应该能够自己画出类之间的调用过程了。

## 4.11.本章小结

## 4.12.思考题