分布式文件系统 UNIT06

HDFS实现原理

文件的写入刨析

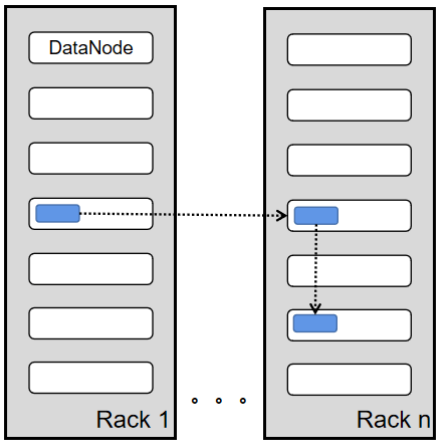
- **数据存放策略：**

  为了提高数据的可靠性与系统的可用性，以及充分利用网络带宽，HDFS采用了以机架（Rack）为基础的数据存放策略。一个HDFS集群通常包含多个机架，不同机架之间的数据通信需要经过交换机或者路由器，同一机架中的不同机器之间的通信则不需要经过交换机或者路由器，这意味着同一机架中不同机器之间的通信要比不同机架之间机器的通信带宽大。

  NameNode如何选择在哪个DataNode存储副本？这里需要对可靠性、写入带宽和读取带宽进行权衡。例如，把所有副本都复制在一个节点中，此时损失的写入带宽最小（因为复制管线都在同一节点上运行），但是这并不提供真实的冗余（如果节点发生故障，那么该块中的数据会丢失）。另外一个极端，把副本放在不同机架的服务器上，虽然最大限度的提高了冗余，但是带宽的损耗非常大。

  HDFS的默认布局策略是：在运行客户端的节点上放第一个副本（如果客户端运行集群之外，就随机选择一个 节点，不过系统会避免挑选那些存储太满或太忙的节点）。第二个副本放在与第一个副本不同且随机的机架中的节点上。第三个副本与第二个副本放在同一机架，且随机选择另一个节点。如果还有其他副本，则放在集群中随机选择的节点上，不过系统会尽量避免在同一个机架上放太多的副本。

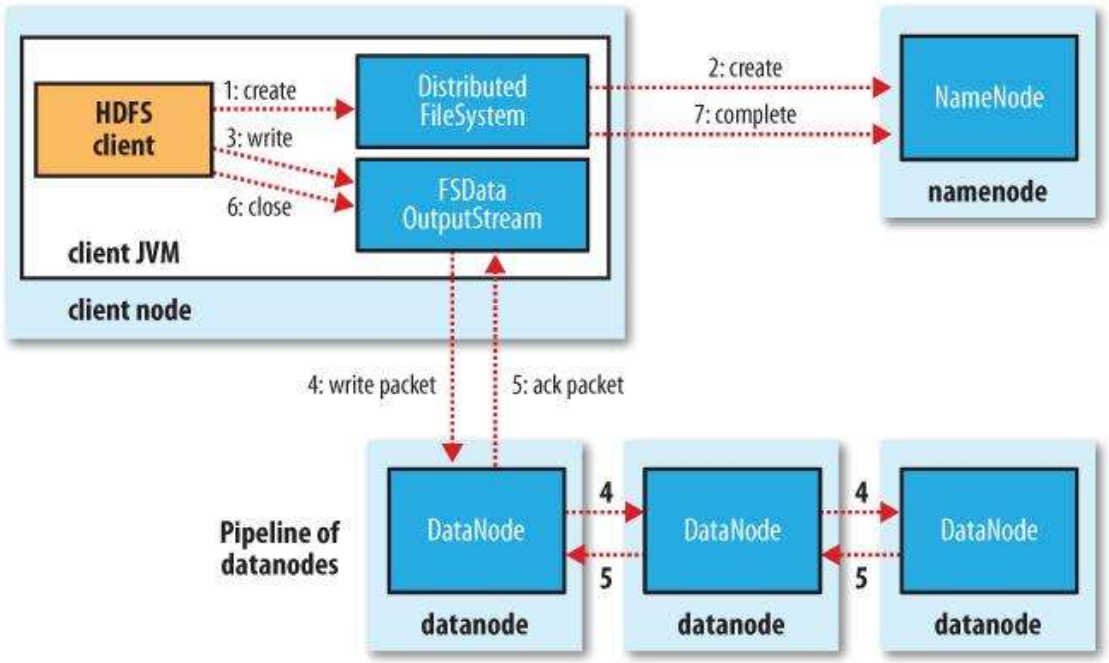
  一旦选定副本的放置位置，就根据网络拓扑创建一个管线，如果副本数量为3，则创建如下图所示的管线：

  总的来说，这一方法不仅提供了很好的稳定性（数据块存储在两个机架中），并实现很好的负载均衡，包括写入带宽（写入操作只需要遍历一个交换机）、读取性能（可以从两个机架中选择读取）和集群中块的均匀分布（客户端只在本地机架上写入一个块）。

- **写入流程分析：** 如何新建一个文件，把数据数据写入文件，最后关闭该文件。

  数据写入流程图如下所示：



4

（1）客户端通过对 `DistributedFileSystem` 对象调用 `create()` 方法来新建文件。

（2） `DistributedFileSystem` 对 `namenode` 创建一个 `RPC` 调用，在文件系统的命名空间中新建一个文件，此时该文件中还没有相应的数据块。 `namenode` 执行各种不同的检查以确保这个文件不存在以及客户端有新建该文件的权限。如果这些检查均通过， `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` 异常。检查通过之后， `DistributedFileSystem` 向客户端返回一个 `FSDataOutputStream` 对象，由此客户端可以开始写入数据。 `FSDataOutputStream` 封装了一个 `DFSoutPutStream` 对象，该对象负责处理 `datanode` 和 `namenode` 之间的通信。

（3）在客户端写入数据时， `DFSOutputStream` 将它分为一个个的数据包，并写入内部队列，称为“数据队列”（data queue）。 `DataStreamer` 处理数据队列，它的责任是挑选出适合存储数据副本的一组 `datanode` ，并据此来要求 `namenode` 分配新的数据块。

（4）这一组 `datanode` 构成一个管线---我们假设副本数为3，所以管线中有3个节点。 `DataStreamer` 将数据包流式传输到管线中第1个 `datanode` ，该 `datanode` 存储数据包并将它发送到管线中的第2个 `datanode` ，同样，第2个 `datanode` 存储该数据包并且发送给管线的第3个 `datanode` （也是最后一个）。

（5） `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（确认信息），称为“确认队列”（ack queue）。确认回执沿着数据流管道逆流而上，从数据管道一次经过各个 `datanode` 并最终发往客户端，收到管道中所有 `datanode` 确认信息后，该数据包才会从确认队列删除。不断执行（3）~（5）步，知道数据全部写完。

如果任何datanode在数据写入期间发生故障，则执行以下操作：

（1）首先关闭管线，确认把队列中的所有数据包都添加回 `DSFOutputStream` 队列的最前端，已确保故障节点下游的datanode不会漏掉任何一个数据包。

（2）为存储在另一个正常datanode的当前数据块指定一个新的标识，并将该标识传送个namenode，以便故障datanode在恢复后可以删除存储的部分数据块。

（3）从管线中删除故障datanode，基于两个正常的datanode构建一条新的管线。余下的数据块写入到管线中正常的datanode。

（4）namenode注意到块副本量不足时，会在另一个节点上创建一个新的副本，后续的数据块继续正常接收处理。

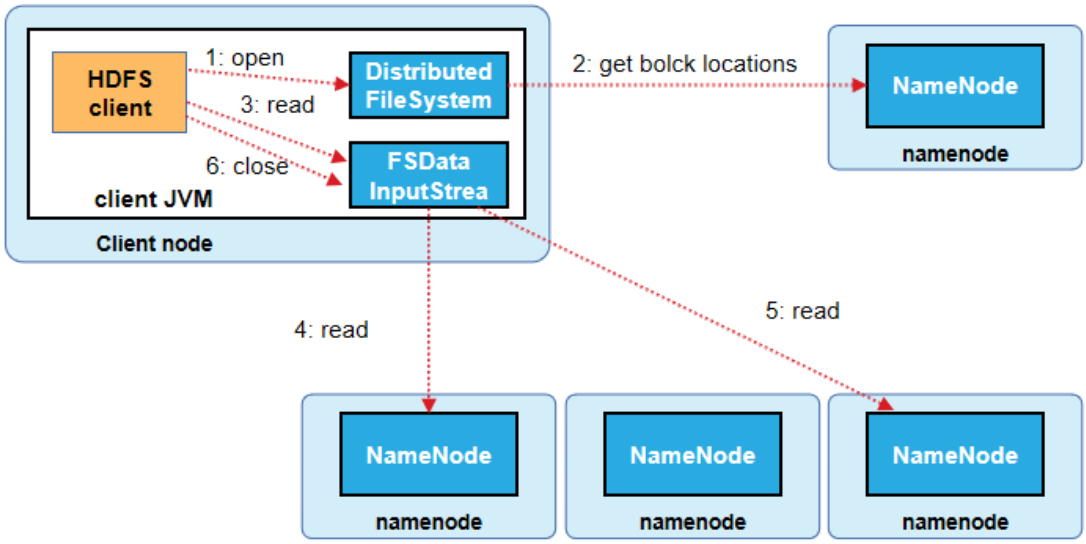
在一个块被写入的期间可能会有多个datanode同时发生故障，但非常少见。只要写入的了`dfs.namenode.replication.min`的副本数（最小副本数默认为1），写操作就会认为是成功的，并且这个块可以在集群中异步复制，直到达到其目标副本数（`dfs.replication`的默认值为3）。

（6）客户端完成数据的写入后，对数据流调用`close()`方法。该操作将剩余的所有数据包写入`datanode`管线，并在等待确认信息。

（7）收到确认回执信息之后，联系namenode，告知其文件写入完成。此时namenode已经知道文件由哪些数据块组成（因为DataStreamer请求分配数据块），所以它在返回成功之前只需要等待数据块进行最小量的复制（只要收到最小副本数写入完成，就可以向namenode发送写入完成信息了）。

文件的读取刨析

- **数据的读取策略：**HDFS提供了一个API可以确定一个数据节点所属的机架ID,客户端也可以调用API获取自己所属的机架ID。当客户端读取数据时，从名称节点获得数据块不同副本的存放位置列表，列表中包含了副本所在的数据节点，可以调用API来确定客户端和这些数据节点所属的机架ID。当发现某个数据块副本对应的机架ID和客户端对应的机架ID相同时，就优先选择该副本读取数据，如果没有发现，就随机选择一个副本读取数据。
- **数据读取分析：**如下图所示，显示在读取文件时事件的发生顺序。



（1）客户端通过调用`FileSystem`对象的`open()`方法来打开希望读取的文件，对于HDFS来说，这个对象是`DistributedFileSystem`的一个实例。

（2）`DistributedFileSystem`通过使用远程过程调用（RPC）来调用`namenode`，以确定文件起始块的位置。对于每一个块，namenode返回存有该块副本的所有datanode地址，此外，这些datanode根据它们与客户端的距离来排序。如果该客户端本身就是与一个datanode（比如在MapReduce任务中），那么客户端将会从保存相应数据块副本的本地datanode读取数据。

（3）`DistributedFileSystem`类返回一个`FSDataInputStream`对象（一个支持文件定位的输入流）给客户端以便读取数据。`FSDataInputStream`类转而封装`DFSInputStream`对象，该对象管理着datanode和namendoe的I/O。

接着，客户端对这个输入流调用`read()`方法。存储着文件起始几个块的datanode地址的`DFSInputStream`随即连接距离最近的文件中的第一个块所在的datanode。

（4）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.

（5）到达块的末端时，`DFSInputStream`关闭与该datanode的连接，然后寻找下一个块的最佳datanode。

所有这些对于客户端都是透明的，在客户端看来它一直在读取一个连续的流。

（6）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来检索下一批数据库的datanode的位置。一旦客户端完成读取，就对`FSDataInputStream`调用`close()`方法。

在读取数据的时候，如果`DFSInputStream`在与datanode通信时遇到错误，会尝试从这个块的另外一个最邻近的datanode读取数据。它也记住那个故障datanode，以保证以后不会反复读取该节点上后续的块。`DFSInputStream`也会通过校验和确认从datanode发来的数据是否完整。如果发现有损坏的块，`DFSInputStream`会试图从其他datanode读取其副本，也会将被损坏的块通知给namenode。

这个设计的一个重点是，客户端可以直接连接到datanode检索数据，且namenode告知客户端每个块所在的最佳datanode。由于数据流分散在集群中的所有datanode，所以这种设计能使HDFS扩展到大量的并发客户端。同时，namenode只需要响应块位置的请求（这些信息存储在内存中，因为非常高效），无需响应数据请求，否则随着客户端数量的增长，namenode会很快成为瓶颈。

网络拓扑与Hadoop

在本地网络中，两个节点被称为“彼此邻近”是什么意思？在海量数据处理中，其主要限制因素是节点之间的数据传输速率 -- 带宽很稀缺。这里的想法是将两个节点之间的带宽作为距离的衡量标准。

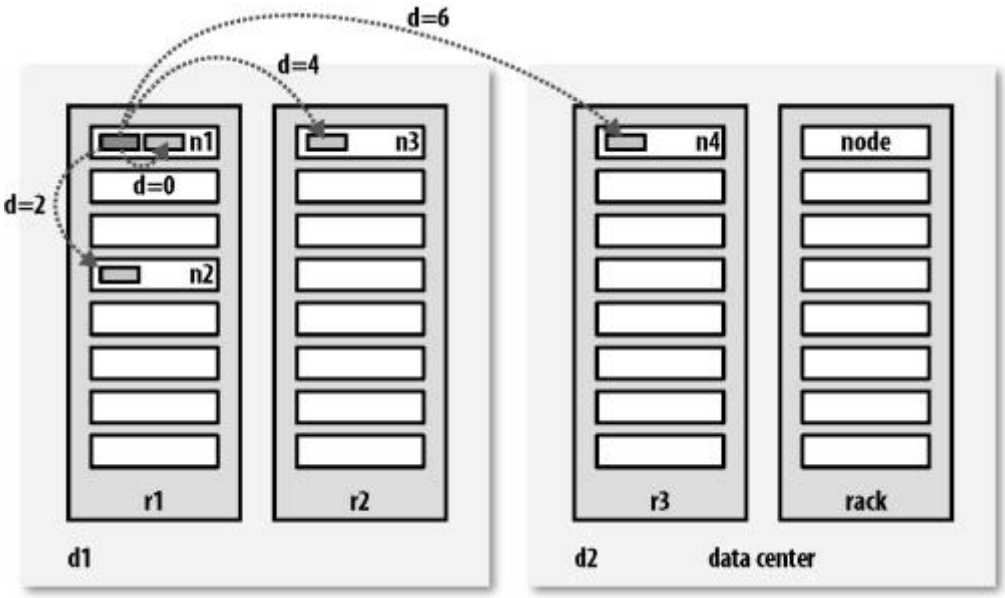
Hadoop采用一个简单的方法：把网络看作一棵树，两个节点之间的距离是它们到最近共同祖先的距离总和。该树中的层次是没有预先设定的，但是相对于数据中心、机架和正在运行的节点，通常可以设定等级。具体想法是针对以下每个场景，可用带宽依次递减：

- 同一节点上的进程
- 同一机架上的不同节点
- 同一数据中心中不同机架上的节点
- 不同数据中心中的节点

例如，假设有数据中心d1，机架r1中的节点n1.该节点可以表示为/d1/r1/n1. 利用这种标记，这里给出四种距离描述：

- distance(/d1/r1/n1, /d1/r1/n1)=0（同一节点上的进程）
- distance（/d1/r1/n1， /d1/r1/n2）=2（同一机架上的不同节点）
- distance（/d1/r1/n1， /d1/r2/n3）=4（同一数据中心不同机架上的节点）
- distance（/d1/r1/n1， /d2/r3/n4）=6（不同数据中心中的节点）

示意如下图所示：



HDFS编程实践

HDFS编程实践主要包括：Linux操作系统中关于HDFS分布式文件系统常用的shell 命令，以及利用Hadoop提供的Java API进行基本的文件操作(HDFS的操作都是在启动HDFS集群的前提下进行的)。

HDFS指令操作

Hadoop文件系统的shell指令可以和Hadoop支持的所有分布式文件进行交互，如本地文件系统、HDFS文件系统、S3、HFTP文件系统等。文件系统的shell指令调用如下所示：

```
{HADOOP_HOME}/bin/hadoop fs <args
```

所有文件系统的shell命令都是用 `路径的URIs` 作为参数。 URI 的格式为 `scheme://authority/path` 。对于HDFS分布式文件系统，scheme为 `hdfs` ， 对于本地文件，scheme为 `file` 。 scheme和authority是可选的。如果为指定，则使用在 `core-site.xml` 配置文件中指定的配置的scheme（ `fs.defaultFS` 值所表示的Hadoop默认文件系统）。 如：在 `core-site.xml` 文件中指定的 `fs.defaultFS` 的值为 `hdfs://namenodehost:port` ，那么表示Hadoop默认使用的文件系统为hdfs分布式文件系统，此时在指令中要使用HDFS分布式文件时，既可以指定路径为： `hdfs://namenodehost:port/parent/child` ，也可以直接指定为 `/parent/child` 。

如果用户在Hadoop中配置的文件系统为HDFS文件系统（在 `core-site.xml` 文件中配置为 `fs.defaultFS` 的值为 `hdfs://namenodehost:port` ），那么使用 `hadoop fs -command` 与使用 `hdfs dfs -command` 效果是相同的。

- 查看Hadoop中文件系统可以使用的指令信息

  ```
  hadoop  fs
  ```

  如果Hadoop中使用HDFS文件系统,也可以使用:

  ```
  hdfs  dfs
  ```

显示内容如下所示:

```
[root@hadoop ~]# hdfs dfs
Usage: hadoop fs [generic options]
        [-appendToFile <localsrc> ... <dst>]
        [-cat [-ignoreCrc] <src> ...]
        [-checksum <src> ...]
        [-chgrp [-R] GROUP PATH...]
        [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
        [-chown [-R] [OWNER][:[GROUP]] PATH...]
        [-copyFromLocal [-f] [-p] [-l] [-d] <localsrc> ... <dst>]
        [-copyToLocal [-f] [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-count [-q] [-h] [-v] [-t [<storage type>]] [-u] [-x] <path> ...]
        [-cp [-f] [-p | -p[topax]] [-d] <src> ... <dst>]
        [-createSnapshot <snapshotDir> [<snapshotName>]]
        [-deleteSnapshot <snapshotDir> <snapshotName>]
        [-df [-h] [<path> ...]]
        [-du [-s] [-h] [-x] <path> ...]
        [-expunge]
        [-find <path> ... <expression> ...]
        [-get [-f] [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-getfacl [-R] <path>]
        [-getfattr [-R] {-n name | -d} [-e en] <path>]
        [-getmerge [-nl] [-skip-empty-file] <src> <localdst>]
        [-help [cmd ...]]
        [-ls [-C] [-d] [-h] [-q] [-R] [-t] [-S] [-r] [-u] [<path> ...]]
        [-mkdir [-p] <path> ...]
        [-moveFromLocal <localsrc> ... <dst>]
        [-moveToLocal <src> <localdst>]
        [-mv <src> ... <dst>]
        [-put [-f] [-p] [-l] [-d] <localsrc> ... <dst>]
        [-renameSnapshot <snapshotDir> <oldName> <newName>]
        [-rm [-f] [-r|-R] [-skipTrash] [-safely] <src> ...]
        [-rmdir [--ignore-fail-on-non-empty] <dir> ...]
        [-setfacl [-R] [{-b|-k} {-m|-x <acl_spec>} <path>]|[--set <acl_spec> <path>]]
        [-setfattr {-n name [-v value] | -x name} <path>]
        [-setrep [-R] [-w] <rep> <path> ...]
        [-stat [format] <path> ...]
        [-tail [-f] <file>]
        [-test -[defsz] <path>]
        [-text [-ignoreCrc] <src> ...]
        [-touchz <path> ...]
        [-truncate [-w] <length> <path> ...]
        [-usage [cmd ...]]
```

(1)文件/目录的创建

- `mkdir` : 创建目录

  使用形式:

  ```
  hdfs dfs -mkdir [-p] <path ...
  ```

  参数:

  - -p : 添加`-p`参数之后, 可以创建不存在的多级目录

  案例:

  ```
  hdfs dfs -mkdir /part01                  #创建 /part01 目录
  hdfs dfs -mkdir -p /part02/part03        #创建 /part02/part03目录,即使part02不
  存在也可以创建成功
  hdfs dfs -mkdir /part04 /part05          #创建/part04和/part05目录
  ```

- `touchz` : 创建指定目录的空文件, 如果文件不为空, 则报错

  使用形式:

  ```
  hdfs dfs -touchz <path ...
  ```

案例:

```
hdfs dfs -touchz /a.txt                    #创建 /a.txt空文件
hdfs dfs -touchz /b.txt /c.txt             #创建 /b.txt和/c.txt两个空文件
```

(2)文件/目录的上传、下载

1. 第一种方式:
   - `copyFromLocal` : 文件/目录上传

     使用形式:

     ```
     hdfs dfs -copyFormLocal [-f] [-p] [-l] [-d] <localsrc ...  <hdfsdst
     ```

     参数:

     - -f : 如果目标路径中已存在上传文件, 则直接覆盖
     - -p : 上传文件保留原来的访问、修改时间、所有权和权限
     - -l : 允许DataNode延迟持久化文件到磁盘，并强制副本数量为1（强制复制因子为1，不建议使用）
     - -d : 上传过程中跳过创建后缀为 ._COPYING_的临时文件

     案例:

     ```
     hdfs dfs -copyFromLocal dept.txt /dept.txt  #将dept.txt文件上传到HDFS的根目录下
     # 再次执行该指令会报错, 并且通过浏览器或-ls指令查看发现HDFS中的dept.txt文件的属性信息与Linux中的属性信息不同
     hdfs dfs -copyFromLocal -f -d dept.txt /dept.txt    # 如果在HDFS根目录中有/dept.txt文件,则覆盖该文件,并保留原来Linux操作系统中该文件的属性信息
     # 此时执行不会报错,并且通过浏览器或-ls指令查看发现: HDFS中的dept.txt文件的属性信息与源文件属性信息相同
     ```

   - `copyToLocal` : 文件下载

     使用形式:

     ```
     hdfs dfs -copyToLocal [-f] [-p] <hdfssrc ...  <localdst
     ```

     参数:

     - -f : 如果目标路径中已存在下载文件, 则直接覆盖
     - -p : 下载文件时保留原来的访问、修改时间、所有权和权限

     案例:

     ```
     hdfs dfs -copyToLocal /dept.txt dept.txt        #将HDFS根目录下的/dept.txt下载到本地
     # 再次执行会报错,通过Linux指令ls查看,发现本地的dept.txt文件的属性信息通过HDFS中的属性信息不同
     hdfs dfs -copyToLocal -f -p /dept.txt dept.txt  #将强制下载HDFS中的/dept.txt文件到本地,并保留属性信息
     # 此时执行不会报错,并通过Linux指令 ls 查看,发现本地和HDFS中的dept.txt文件属性信息相同
     ```

2. 第二种方式:

- `put` : 文件上传

  使用形式:

  ```
  hdfs dfs -put [-f] [-p] [-l] [-d] <localsrc ...  <hdfsdst
  ```

  参数:

  - -f : 如果目标路径中已存在上传文件, 则直接覆盖
  - -p : 上传文件保留原来的访问、修改时间、所有权和权限
  - -l : 允许DataNode延迟持久化文件到磁盘，并强制副本数量为1（强制复制因子为1，不建议使用）
  - -d： 上传过程中跳过创建后缀为 ._COPYING_的临时文件

  案例:

  ```
  hdfs dfs -put -f -p emp.txt dept.txt /
  # 将emp.txt和dept.txt文件, 强制上传(如果原来存储则覆盖)到的HDFS的根目录下,并保留文件的属性信息
  ```

- `get` : 文件下载

  使用形式:

  ```
  hdfs dfs -get [-f] [-p] <hdfssrc ...  <localdst
  ```

  参数:

  - -f : 如果目标路径中已存在下载文件, 则直接覆盖
  - -p : 下载文件时保留原来的访问、修改时间、所有权和权限

  案例:

  ```
  hdfs dfs -get -f -p /dept.txt /emp.txt /opt
  # 将HDFS中的文件/dept.txt和/emp.txt文件强制下载(如果目标路径中存在则覆盖)到Linux的/opt目录中,并保留源文件的属性信息
  ```

3. 第三种方式:

- `moveFromLocal` : 文件上传, 并删除本地源文件

  使用形式:

  ```
  hdfs dfs -moveFromLocal <localsrc ... <hdfsdst
  ```

  案例:

  ```
  hdfs dfs -moveFromLocal emp.txt dept.txt /
  # 将本地的emp.txt文件和dept.txt上传到HDFS的根目录中,并删除本地源文件
  ```

- `moveToLocal` : 文件下载, 并删除HDFS中的源文件

  使用形式:

  ```
  hdfs dfs -moveToLocal <hdfssrc <localdst
  ```

  该指令尚未实现。

(3)目录内容查询

- `ls`： 查看指定目录下的内容

  使用形式:

  ```
  hdfs dfs -ls [-C] [-d] [-h] [-R] [-t] [-S] [-r] <path ...
  ```

  参数:

  - -C : 只显示文件和目录的路径
  - -d : 只显示指定目录的属性信息
  - -h : 将文件大小以人类认识的方式显示(如64.0M,而不是67108864)
  - -R : 递归显示子孙目录内容
  - -t : 按照修改时间对输出排序
  - -S : 按照文件的大小对输出排序
  - -r : 以倒序的方式显示

  案例:

  ```
  hdfs dfs -ls -h -R -t /          #显示HDFS根目录下的内容,文件以人类可读的方式显示,递归显示所有子孙目录下的内容,并且按照修改时间排序显示
  ```

- `count`：计算指定目录下的目录个数、文件个数和所占字节数。（HD'FS高级指令）

  使用形式:

  ```
  hdfs dfs -count [-q] [-h] [-v] [-t] [-u] <path ...
  ```

  参数:

  - -q : 显示所有列的内容。列包含：QUOTA（当前目录的文件限额，包括目录和文件的总个数，如果没有指定过配额，则显示none）、REMAINING_QUOTA（剩余的文件限额，即还可以创建多少个文件或文件夹，如果没有指定过配额，则显示inf）、SPACE_QUOTA（当前目录中的空间限额，如果没有指定过配额，则显示none）、REMAINING_SPACE_QUOTA（剩余的空间限额，如果没有指定过配额，则显示inf）、DIR_COUNT（当前目录中的目录个数）、FILE_COUNT（当前目录中的文件个数）、CONTENT_SIZE（当前目录的总大小）、PATHNAME（当前目录的路径）。
  - -u：不显示目录的个数、文件的个数以及目录总大小（DIR_COUNT、FILE_COUNT、CONTENT_SIZE）
  - -t：显示每种存储类型的配额和使用情况。如果在参数中没有指定 -q或者-u，但是指定了-t，则忽略参数-t。存储类型有：all（所有）、ssd、ram_disk、disk或archive
  - -h：以人类可读的格式显示大小
  - -v：显示标题。

  案例:

```
hdfs dfs -count /out         #查看out目录中,目录的个数\文件的个数\所占字节数
#输出结果为:
    5       6       23123    /out

hdfs dfs -count -q -v -h /GSOD #查看GSOD目录的配额，显示所有列

#输出结果如下：
QUOTA REM_QUOTA SPACE_QUOTA REM_SPACE_QUOTA DIR_COUNT FILE_COUNT
CONTENT_SIZE PATHNAME
none     inf       none           inf            5          6             3.8
M /GSOD
```

(4)文件内容查询

- `cat`：查看指定文件的内容

  使用形式:

  ```
  hdfs dfs -cat <src ...
  ```

  案例:

  ```
  hdfs dfs -cat /out1 /out2        #显示/out1和/out2文件中的内容
  ```

- `tail`：显示文件的最后一个千字节（1KB）的数据

  使用形式:

  ```
  hdfs dfs -tail [-f] <file
  ```

  参数:

  - -f : 当文件内容更新时, 输出将会改变, 具有实时性.

  案例:

  ```
  hdfs dfs -tail /out1         #查看/out1文件最后1KB的内容

  hdfs dfs -tail -f /out1      #查看/out1文件最后1KB的内容,此时命令窗口进入等待状态
  # 如果通过其他指令或API向/out1文件中添加数据成功之后,会在立即更新 tail -f 输出的内容
  ```

- `text`：将HDFS中文件以文本形式输出(包括zip包、jar包等形式中的内容)

  使用形式:

  ```
  hdfs dfs -text <src ...
  ```

  案例:

  ```
  hdfs dfs -text a.jar         #将a.jar包中的内容以文本形式输出
  ```

- `checksum`：返回文件的校验信息

  使用形式:

```
hdfs dfs -checksum <src ...
```

案例:

```
hdfs dfs -checksum /GSOD.jar        #查看GSOD.jar包的校验信息

# 输出结果为:
/GSOD.jar MD5-of-0MD5-of-512CRC32C
000002000000000000000000de583ac39098a2947bf6b22d9f8f0757
```

(5)文件/目录查找

- `find` : 查找满足表达式的文件和文件夹。没有配置path的话，默认的就是全部目录/；如果表达式没有配置，则默认为-print。

  使用形式:

  ```
  hdfs dfs -find <path ... <expression ...
  ```

  下面是常用的表达式:

  - -name pattern
  - -iname pattern

  根据文件的名称匹配，其中iname表示不区分大小写匹配

  案例:

  ```
  hdfs dfs -find / -name *.txt        #在HDFS的根目录下查找以.txt结尾的所有的文件
  ```

(6)文件/目录的删除

- `rm` : 删除指定的文件/目录

  使用形式:

  ```
  hdfs dfs -rm [-f] [-r|-R] <src ...
  ```

  参数:

  - -f : 如果要 删除的文件/目录不存在,不会报错,也不会有任何提示信息
  - -r | -R : 递归删除该目录下的所有内容

  案例:

  ```
  hdfs dfs -rm -f -R /aaa     #如果/aaa存在,则删除/aaa及子孙文件/目录
  ```

(7)文件/目录的修改

- `appendToFile` : 将本地文件系统中的一个或多个文件中的内容追加到目标文件中

  使用形式:

  ```
  hdfs dfs -appendToFile <localsrc ... <dst
  ```

  案例:

```
hdfs dfs -appendToFile a.txt /a.txt
#将本地文件系统中的a.txt文件的内容追加到HDFS文件系统中的/a.txt文件中
```

- `mv`：重命名或剪切

  使用形式:

  ```
  hdfs dfs -mv <src ... <dst
  ```

- `cp`：拷贝

  使用形式:

  ```
  hdfs dfs -cp [-f] [-p | -p[topax]] [-d] <src ... <dst
  ```

  参数:

  - -f：如果目标路径中存在, 则 -f 选项将覆盖目标目录中存在的文件/目录
  - -p：保留文件属性(时间戳、所有权、权限、ACL等)

- `setrep`：修改文件的副本数量

  使用形式：

  ```
  hdfs dfs -setrep <rep <path ...
  ```

  案例：

  ```
  hdfs dfs -setrep 1 /a.txt       #将/a.txt文件的副本数量修改为1
  ```

- `chgrp`：更改文件所属群组(用户必须文件的所属者或者超级用户)

  使用形式:

  ```
  hdfs dfs -chgrp [-R] GROUP PATH...
  ```

  参数:

  - -R：递归更改目录下的所有子孙目录及文件

  案例:

  ```
  hdfs dfs -chgrp -R root /   #将HDFS根目录的所属群组修改root
  ```

- `chmod`：更改文件的权限(用户必须是文件的所有者或者超级用户)

  HDFS中的文件/目录对于所属者用户、群组内的其他用户、非群组内的用户都有单独的权限。

  对于文件：要求有`r`权限才能读取文件，而有`w`权限才能写入或追加内容到文件。(文件没有可执行的概念)

  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。

  在设置权限的过程中, 可以使用数值代表权限: 如 r--4, w--2, x--1。如某一目录对所属者有`rwx`权限, 对所属者群组内的其他用户有`r-x`权限, 对非群组内的其他用户只具有`r--`权限, 在设置的过程中可使用754来表示。

  使用形式:

```
hdfs dfs -chmod [-R] <MODE[,MODE]... | OCTALMODE PATH...
```

参数:

- -R : 递归更改目录下的所有子孙目录及文件

案例:

```
hdfs dfs -chmode -R 754 /out    #修改/out文件/目录的权限,对所属者为rwx,本群组内的其他用户r-x,对非群组的其他用户:r--
```

- `chown` : 更改文件的所属者(用户必须是文件的所有者或者超级用户)

  使用形式:

  ```
  hdfs dfs -chown [-R] [OWNER][:[GROUP]] PATH...
  ```

  参数:

  - -R : 递归更改目录下的所有子孙目录及文件

  案例:

  ```
  hdfs dfs -chown -R admin:root /a.txt
  # 修改/a.txt文件的所属者为root群组中的admin用户.(将文件所属群组也给修改了)
  ```

HDFS Java API操作

Hadoop主要是使用Java语言编写的, Hadoop不同的文件系统之间通过调用Java API进行交互。上面介绍的shell命令，本质上就是Java API的应用。这里将介绍HDFS文件系统中常用的操作。

在操作过程中，使用maven项目来实现。在`pom.xml`文件中需要添加如下所示的依赖信息:

```
<dependency
    <groupIdorg.apache.hadoop</groupId
    <artifactIdhadoop-common</artifactId
    <version2.8.5</version
</dependency
<dependency
    <groupIdorg.apache.hadoop</groupId
    <artifactIdhadoop-client</artifactId
    <version2.8.5</version
</dependency
<dependency
    <groupIdorg.apache.hadoop</groupId
    <artifactIdhadoop-hdfs</artifactId
    <version2.8.5</version
</dependency
```

(1)常用API介绍

- `org.apache.hadoop.conf.Configuration` : 该对象封装了客户端或服务器的配置。注意：配置信息都是hadoop默认的配置。如果想要将配置内容更改为用户自定义的配置，可以通过两种方式来设置:
  - （1）通过设置配置文件读取类路径来实现。将HDFS相关的配置文件（core-site.xml和hdfs-site.xml）添加到当前项目的类路径中（如在IDEA的Maven项目，需要添加到

`src/main/resources`目录中）。
    - （2）通过在该对象中通过<属性名：属性值的方式（k-v）——指定

```
// 创建配置信息对象
Configuration conf = new Configuration();
// 设置文件系统为HDFS文件系统，并指定存储的副本为1
conf.set("fs.defaultFS", "hdfs://192.168.56.100:9000");
conf.set("dfs.replication", "1");
```

- `org.apache.hadoop.fs.FileSystem`：一个通用文件系统的抽象基类，可以被分布式文件系统继承。所有可能使用Hadoop文件系统的代码，都要用到这个类。虽然Hadoop为`FileSystem`提供了多种具体的实现，但是还是建议使用该抽象类来完成关于文件系统的操作，这样在不同文件系统中可移植。

  Hadoop中文件系统对象的创建通常使用如下三种方式来实现：

```
FileSystem fs = FileSystem.get(URI uri, Configuration conf, String user);
FileSystem fs = FileSystem.get(URI uri, Configuration conf);
// 上面这两种方式，主要用于在conf（配置信息对象）中未指定Hadoop使用的文件系统时，使用URI指定文件系统
// 如： URI hdfsUri = new URI("hdfs://192.168.56.100:9000");
// 第一种方式最后的String user为指定的用户名（Hadoop允许指定其他名称的用户名）

// 第三种方式，主要用于在配置信息对象中已经指定Hadoop使用的文件系统
FileSystem fs = FileSystem.get(Configuration conf);
```

  对Hadoop文件系统的操作，如增删改查，都可以通过`FileSystem`对象来实现。

- `org.apache.hadoop.fs.Path`：用于表示Hadoop文件系统中的一个文件或一个目录的路径对象。可以通过

```
Path path = new Path("/a.txt");
```

  的方式创建该对象。

- `org.apache.hadoop.fs.FileStatus`：一个用于向客户端展示系统中文件和目录的元数据的接口。展示的信息主要包括：文件大小、块大小、副本信息、所有者、修改时间等，可以通过

```
FileSystem.listStatus()
```

  方法获得具体的实例对象。

- `org.apache.hadoop.fs.FSDataInputStream`：文件输入流，用于读取Hadoop文件。可以通过

```
FileSystem.open()
```

  方法返回该对象

- `org.apache.hadoop.fs.FSDataOutputStream`：文件输出流，用于写入数据到Hadoop文件中。可以通过

```
FileSystem.create()
```

  方法返回该对象

(2)数据读取

可以使用`FileSystem.open()`方法来返回一个读取HDFS文件系统文件的输入流。获取得到输入流之后，可以将流中的数据写入到本地文件系统（文件下载），也可以直接输出到控制台（查看HDFS文件内容）。下面是文件下载和查看HDFS文件内容的完整案例代码。

- 文件下载

```
static Configuration conf = new Configuration();
static FileSystem fs = null;
static{
    conf.set("fs.defaultFS", "hdfs://192.168.56.100:9000");
    conf.set("dfs.replication", "1");
    try {
        fs = FileSystem.get(conf);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) throws IOException {
    // 获取读取HDFS文件系统中文件的数据输入流
    FSDataInputStream in = fs.open(new Path("/a.txt"));
    File f = new File("a.txt");
    // 获取写出到本地文件系统文件的数据输出流
    OutputStream out = new FileOutputStream(f);
    // 交换输入、输出流中的内容
    IOUtils.copy(in, out);
    // 关闭资源
    out.close();
    in.close();
}
```

- 查看HDFS文件内容

```
static Configuration conf = new Configuration();
static FileSystem fs = null;
static{
    conf.set("fs.defaultFS", "hdfs://192.168.56.100:9000");
    conf.set("dfs.replication", "1");
    try {
        fs = FileSystem.get(conf);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) throws IOException {
    // 获取读取HDFS文件系统中文件的数据输入流
    FSDataInputStream in = fs.open(new Path("/a.txt"));
    BufferedReader read = new BufferedReader(new InputStreamReader(in));
    String line = null;
    while((line = read.readLine()) != null){
        System.out.println(line);
    }
    read.close();
    in.close();
```

```
}
```

(3)数据写入

将本地文件写入到HDFS文件系统中（文件上传或者追加数据），需要先获取本地文件的输入流，可以通过下面的方式获取：

```
InputStream in = new FileInputStream(new File("本地文件路径"));
```

还需要获取写出到HDFS文件系统的输出流，可以通过下面的方式获取：

```
// 文件上传
FSDataOutputStream out = FileSystem.create(new Path("HDFS文件路径"));

// 文件追加
FSDataOutputStream out = FileSystem.append(new Path("HDFS文件路径"));
```

然后将输入流中的内容交换给输出流：

```
IOUtils.copy(in, out);
```

- 文件上传

  下面是完整的代码：

  ```
  static Configuration conf = new Configuration();
  static FileSystem fs = null;
  static{
      conf.set("fs.defaultFS", "hdfs://192.168.56.100:9000");
      conf.set("dfs.replication", "1");
      try {
          fs = FileSystem.get(conf);
      } catch (IOException e) {
          e.printStackTrace();
      }
  }

  public static void main(String[] args) throws IOException {
      File file = new File("src/main/resources/core-site.xml");
      InputStream in = new FileInputStream(file);
      OutputStream out = fs.create(new Path("/core-site.xml"));
      IOUtils.copy(in, out);
  }
  ```

- 文件追加

  下面是完整代码：

  ```
  static Configuration conf = new Configuration();
  static FileSystem fs = null;
  static{
      conf.set("fs.defaultFS", "hdfs://192.168.56.100:9000");
      conf.set("dfs.replication", "1");
      try {
          fs = FileSystem.get(conf);
  ```

```
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) throws IOException {

    File file = new File("src/main/resources/core-site.xml");
    InputStream in = new FileInputStream(file);
    OutputStream out = fs.append(new Path("/core-site.xml"));
    IOUtils.copy(in, out);
}
```

- 创建目录

  `FileSystem`实例提供了创建目录的方法：

  ```
  FileSystem.mkdirs(Path path);
  ```

  这个方法可以一次性创建所有必要但没有的父目录。如果创建成功，则返回true。

  通常，用户不需要显示创建一个目录，因为调用`FileSystem.create()`方法写入文件时会自动创建父目录。

(4)查询文件系统

通过`FileSystem.listStatus()`方法，会返回一个保存指定路径下所有的`FileStatus[]`数组实例。HDFS中对该方法有如下几个实现：

```
FileStatus[] listStatus(Path path);
FileStatus[] listStatus(Path f, PathFilter filter);
FileStatus[] listStatus(Path[] files);
FileStatus[] listStatus(Path[] files, PathFilter filter);
```

其中传入的path对象，用来指定HDFS文件/目录的路径，可以指定多个，如第三个和第四个方法。

PathFiter对象用来限制匹配的文件和目录。

- 查询某一目录下的所有文件或目录

  下面是完整代码：

  ```
  static Configuration conf = new Configuration();
  static FileSystem fs = null;
  static{
      conf.set("fs.defaultFS", "hdfs://192.168.56.100:9000");
      conf.set("dfs.replication", "1");
      try {
          fs = FileSystem.get(conf);
      } catch (IOException e) {
          e.printStackTrace();
      }
  }

  public static void main(String[] args) throws IOException {
      FileStatus[] fileStatuses = fs.listStatus(new Path("/GSOD"));
      for (FileStatus fileStatus : fileStatuses) {
          System.out.println(fileStatus);
  ```

```
    }
}
```

- 通配符及含义：在单个操作中处理一批文件是一个很常见的需求。在一个表达式中使用通配符来匹配多个文件是比较方便的，无需列举每个文件和目录来指定输入，该操作称为“通配（globbing）”，Hadoop为执行通配提供了两个方法：

```
FileStatus[] globStatus(Path pathPattern);
FileStatus[] globStatus(Path pathPattern, PathFilter filter);
```

`globStatus()`方法返回路径格式与指定模式匹配的所有FileStatus对象组成的数组，并按路径排序。PathFilter命令作为可选项可以进一步对匹配结构进行限制。

Hadoop支持的通配符与Unix bash shell支持的相同，如图所示：

通配符	匹配
*	匹配0或多个字符
?	匹配单一字符
[ab]	匹配{a，b}集合中的一个字符
[^ab]	匹配非{a，b}集合中的一个字符
[a-b]	匹配一个在{a，b}范围内的字符（包括ab）
[^a-b]	匹配一个不在{a，b}范围内的字符（包括ab）
{a，b}	匹配包含a或b中的一个表达式
\c	转义字符

在HDFS根目录下匹配以`.txt`结尾的文件，下面是代码案例：

```
static Configuration conf = new Configuration();
static FileSystem fs = null;
static{
    conf.set("fs.defaultFS", "hdfs://192.168.56.100:9000");
    conf.set("dfs.replication", "1");
    try {
        fs = FileSystem.get(conf);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) throws IOException {
    // 在根目录下匹配以 .txt 结尾的文件
    FileStatus[] fileStatuses = fs.globStatus(new Path("/*.txt"));
    for (FileStatus fileStatus : fileStatuses) {
        System.out.println(fileStatus);
    }
}
```

- `PathFilter`对象：通配符模式并不总能够精确地描述我们想要访问的文件。比如，使用通配符格式排除一个特定的文件就不太可能。`FileSystem`中的`listStatus()`和`globStatus()`方法中提供了可选的`PathFilter`对象，以编程方式控制通配符。

  下面是一个自定义的通过正则表达式的方式匹配文件的`PathFilter`。

```
class RegexPathFilter implements PathFilter{
    private  final String regex;

    public RegexPathFilter(String regex) {
        this.regex = regex;
    }

    @Override
    public boolean accept(Path path) {
        return path.toString().matches(regex);
    }
}
```

  此时就可以在代码中通过正则表达式进行匹配：

```
public static void main(String[] args) throws IOException {
    // 匹配以 .txt结尾的文件
    FileStatus[] fileStatuses = fs.listStatus(new Path("/"), new
RegexPathFilter("^.*(.txt)+$"));
    for (FileStatus fileStatus : fileStatuses) {
        System.out.println(fileStatus);
    }
}
```

(5)删除数据

使用`FileSystem.delete()`方法可以永久性删除文件或目录。

```
boolean delete(Path path, boolean recursive)
```

如果指定的路径为一个文件或空目录，那么`recursive`参数值就会被忽略。但如果指定的为非空目录时，`recursive`为true时，成功删除非空目录，`recursive`为false时，会抛出`IOException`异常。