第2章

分布式文件系统 HDFS

本章重点

※ 使用 HDFS 存储大数据文件
※ HDFS 基本概念及体系结构
※ HDFS shell 操作 HDFS 文件
※ Java API 操作 HDFS 文件

本章目标

※ 掌握 HDFS 文件系统的访问方式
※ 掌握 HDFS 的体系结构
※ 掌握 HDFS 数据的读写流程
※ 了解 HDFS 的序列化使用

本章任务

学习本章，需要完成以下 4 个工作任务。请记录下来学习过程中所遇到的问题，可以通过自己的努力或访问 kgc.cn 解决。

任务 1：初识 HDFS

了解 HDFS 的产生背景、HDFS 文件系统是什么以及特点和设计目标，掌握 HDFS 文件系统的架构组成。

任务 2：HDFS 操作

掌握使用 HDFS shell 和 Java API 操作 HDFS 文件系统。

任务 3：HDFS 运行机制

掌握 HDFS 文件的读写流程、副本摆放策略、认知 HDFS 数据负载均衡和机架感知。

任务 4：HDFS 进阶

了解 Hadoop 的序列化操作，掌握 SquenceFile 和 MapFile 的常用操作。

任务 1　初识 HDFS

关键步骤如下：

- 认知文件系统以及 HDFS 文件系统。
- 了解 HDFS 文件系统存储的优缺点。
- 认识 HDFS 的基本概念。
- 掌握 HDFS 的体系架构。

2.1.1　HDFS 概述

1. HDFS 产生背景

我们在大数据概述章节中已经了解到当今产生的数据量越来越多，那么与之相对应的需要存储和处理的数据量也就越来越多。我们平时使用的操作系统的存储空间是有限的，存储不了那么多的数据，有小伙伴们想到是否能把多个操作系统综合成为一个大的操作系统呢？然后把数据存储在这个大的系统中，这种方法是可行的，但是却不方便管理和维护。因此，迫切需要一种系统来管理分散存储在多台机器上的文件，于是就产生了分布式文件管理系统，英文名称为 DFS（Distributed File System）。

那么到底什么是分布式文件系统？它是允许将一个文件通过网络在多台主机上以多副本（提高容错性）的方式进行存储。分布式文件系统实际上是通过网络来访问文件，

用户和程序看起来就像是访问本地的磁盘一样。

2. HDFS 简介

HDFS（Hadoop Distributed File System）是 Hadoop 项目的核心子项目，用于分布式计算中的数据存储。Hadoop 官方给的描述是：HDFS 可以运行在廉价的服务器上，为存储海量数据提供了高容错、高可靠性、高可扩展性、高获得性、高吞吐率等特征。说到 HDFS，那就不得不提 Google 的 GFS，HDFS 就是基于它做的开源实现。

Hadoop 整合了众多的底层文件系统，比如本地文件系统、HDFS 文件系统、S3 文件系统，在 Hadoop 中提供了一个文件系统抽象类 org.apache.hadoop.fs.FileSystem，对应的具体实现类如表 2-1 所示。

表 2-1　Hadoop 的文件系统

文件系统	URI 方案	Java 实现	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
FTP	ftp	fs.ftp.FtpFileSystem	由 FTP 服务器支持的文件系统
S3（本地）	s3a	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。

3. HDFS 的优点

（1）处理超大文件：这里的超大文件通常是指 MB 到 TB 级别的数据文件，Hadoop 中并不怕文件大，相反，如果 HDFS 文件系统中存在众多的小文件，那么对于集群的性能反而有所下降的。

（2）运行于廉价机器上：Hadoop 集群可以部署在普通的廉价的机器之上，无需部署在价格昂贵的小型机上，这可以降低公司的运营成本。那么有些小伙伴可能就会

问了，要是运行在廉价的机器上，出现故障该怎么办？这就要求 HDFS 自身要做到高可用、高可靠。

（3）流式地访问数据：HDFS 提供一次写入，多次读取的服务。比如你在 HDFS 上存储了一个要处理的问题，后续可能会有多个作业都会使用到这份数据，那么只需要通过集群来读取前面已经存储好的数据即可。HDFS 设计之初是不支持对文件追加内容的，后来随着 Hadoop 社区的发展，现在已支持对已有文件进行内容的追加。

4. HDFS 的缺点

（1）不适合低延迟数据访问

HDFS 本身是为存储大数据而设计的，如果你在工作中遇到的需求是要求时间低延时的应用请求，那么 HDFS 不适合。实时性、低延迟的查询使用 HBase 会是更好的选择，但是 HBase 中的 rowkey 设计的是否合适也会决定你的查询性能的高低。

（2）无法高效存储大量小文件

在 HDFS 文件系统中的元数据（元数据信息包括：文件和目录自身的属性信息，例如文件名、目录名、父目录信息、文件大小、创建时间、修改时间等；记录文件内容存储相关信息，例如文件块情况、副本个数、每个副本存放在哪等）是存放在 NameNode 的内存中，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。一旦集群中的小文件过多，会导致 NameNode 的压力陡增，进而影响到集群的性能。我们可以采用 SequenceFile 等方式对小文件进行合并，或者是使用 NameNode Federation 的方式来改善。

5. HDFS 的设计目标

HDFS 的设计目标详细描述可以参考 Hadoop 的官方文档的描述：http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html#Introduction。本节我们就挑选几个重要的设计目标进行讲解。

（1）硬件错误

硬件错误是常态而不是异常。HDFS 可能由成百上千的服务器所构成，每个服务器上存储着文件系统的部分数据。事实上构成系统的组件数目是巨大的，而且任一组件都有可能失效，这意味着总是有一部分 HDFS 的组件是不工作的。因此错误检测和快速、自动的恢复是 HDFS 最核心的架构目标。

（2）大规模数据集

运行在 HDFS 上的应用具有很大的数据集。HDFS 上的一个典型文件，大小一般都在 GB 至 TB。因此，需要调节 HDFS 以支持大文件存储。HDFS 应该能提供整体较高的数据传输带宽，能在一个集群里扩展到数百个节点。一个单一的 HDFS 实例应该能支撑千万计的文件。

（3）移动计算代价比移动数据代价低

一个作业的计算，离它操作的数据越近就越高效，在数据达到海量级别的时候更是如此。因为这样能降低网络阻塞的影响，提高系统数据的吞吐量。将计算移动到数

据附近，比将数据移动到应用所在显然更好。HDFS 为应用提供了将计算移动到数据附近的接口。

（4）其他

请参考 Hadoop 的官方文档描述。

2.1.2　HDFS 基本概念

1. 数据块（Block）

HDFS 默认的最基本的存储单位是数据块（Block），默认的块大小（Block Size）是 64M（有些发布版本为 128M）。HDFS 中的文件是被分成以 Block Size 为大小的数据块存储的。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，文件大小是多大就占用多少存储空间。HDFS 与 Block 的关系如图 2.1 所示。

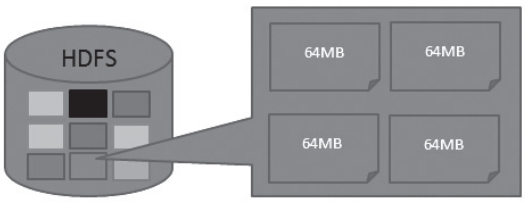

图 2.1　HDFS 与 Block 的关系

2. 元数据节点（NameNode）

NameNode 的职责是管理文件系统的命名空间，它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，至于一个文件包括哪些数据块，分布在哪些数据节点上，这些信息都存储下来。

NameNode 目录结构如下所示：

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

目录结构描述：

（1）VERSION 文件是存放版本的文件，保存了 HDFS 的版本号。

（2）edits：当文件系统客户端进行写操作时，首先把它记录在修改日志中，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，修改日志都会同步到文件系统。

（3）fsimage 文件即命名空间文件。

3. 数据节点（DataNode）

DataNode 是文件系统中真正存储数据的地方，一个文件被拆分成多个 Block 后，会将这些 Block 存储在对应的数据节点上。客户端向 NameNode 发起请求然后到对应

的数据节点上写入或者读出对应的数据 Block。

DataNode 目录结构如下所示：

```
${dfs.name.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/
```

目录结构描述：

（1）blk_<id> 保存的是 HDFS 的数据块，其中保存了具体的二进制数据。

（2）blk_<id>.meta 保存的是数据块的属性信息：版本信息、类型信息和校验和。

（3）subdirxx: 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。

4. 从元数据节点（Secondary NameNode）

Secondary NameNode 并不是 NameNode 节点出现问题时的备用节点，它和元数据节点负责不同的功能。其主要功能就是周期性将 NameNode 的 namespace image 和 edit log 合并，以防日志文件过大。合并过后的 namespace image 也在元数据节点保存了一份，以防在 NameNode 失败的时候进行恢复。

Secondary NameNode 目录结构如下所示：

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

Secondary NameNode 用来帮助 NameNode 将内存中的元数据信息 checkpoint 到硬盘上。

2.1.3 HDFS 体系结构

1. 体系架构概述

HDFS 体系架构详细描述参见：http://hadoop.apache.org/docs/current/hadoop-project-

dist/hadoop-hdfs/HdfsDesign.html#NameNode_and_DataNodes。

HDFS 采用 master/slave 的架构。一个 HDFS 集群是由一个 NameNode 和一定数量的 DataNodes 组成。NameNode 是一个中心服务器，负责管理文件系统的名字空间（namespace）以及客户端对文件的访问。集群中的 DataNode 一般是一个节点对应一个，负责管理它所在节点上的存储数据。HDFS 暴露了文件系统的名字空间，用户能够以文件的形式在上面存储数据。从内部看，一个文件其实被分成一个或多个数据块，这些块存储在一组 DataNode 上。NameNode 执行文件系统的名字空间操作，比如打开、关闭、重命名文件或目录，它也负责确定数据块到具体 DataNode 节点的映射。DataNode 负责处理文件系统客户端的读写请求。在 NameNode 的统一调度下进行数据块的创建、删除和复制。HDFS 架构如图 2.2 所示：

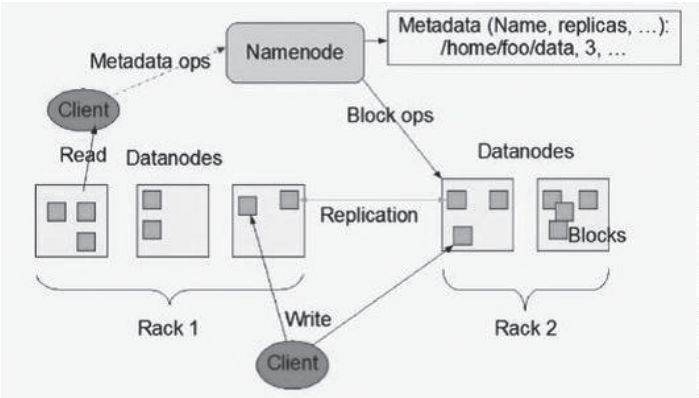

图 2.2　HDFS 体系结构

2. 架构组件功能

NameNode 和 DataNode 被设计成可以在普通的商用机器上运行。这些机器一般运行着 GNU/Linux 操作系统（OS）。由于采用了可移植性极强的 Java 语言，使得 HDFS 可以部署到多种类型的机器上，任何支持 Java 的机器都可以部署 NameNode 或 DataNode。一个典型的部署场景是一台机器上只运行一个 NameNode 实例，而集群中的其他机器分别运行一个 DataNode 实例。这种架构并不排斥在一台机器上运行多个 DataNode，只不过这样的情况比较少见。

文件系统的名字空间（namespace）：HDFS 支持传统的层次型文件组织结构。用户或者应用程序可以创建目录，然后将文件保存在这些目录里。文件系统名字空间的层次结构和大多数现有的文件系统类似，用户可以创建、删除、移动或重命名文件。当前，HDFS 不支持用户磁盘配额和访问权限控制，也不支持硬链接和软链接。但是 HDFS 架构并不妨碍实现这些特性。

NameNode 负责维护文件系统的名字空间，任何对文件系统名字空间或属性的修改都将被 NameNode 记录下来。应用程序可以设置 HDFS 保存的文件的副本数目，文件副本的数目称为文件的副本系数，这个信息也是由 NameNode 保存的。

其他的功能可以查看 Hadoop 官方文档。至此，在学习了以上相关知识后，任务 1 就可以完成了。

任务 2　HDFS 操作

关键步骤如下：

- 掌握使用 shell 访问 HDFS 文件系统。
- 掌握使用 Java API 访问 HDFS 文件系统。
- 掌握 DataFrame 的常用操作。
- 掌握 RDD 和 DataFrame 互操作。

2.2.1　HDFS shell 访问

1. 概述

HDFS 文件系统为使用者提供了基于 shell 操作命令来管理 HDFS 上的数据。这些 shell 命令设计的和 Linux 的命令十分类似，这样设计的好处是让已经熟悉 Linux 的用户可以更加快速的对 HDFS 文件系统的数据进行操作，减少学习所需要的时间。

注意：

使用 HDFS shell 之前需要先启动 Hadoop。

HDFS 的基本命令格式为：

```
bin/hdfs dfs -cmd <args>
```

注意：

cmd 就是具体的命令，cmd 前面的“-”千万不要省略。

2. 列出文件目录

命令：hadoop fs -ls 目录路径

示例：查看 HDFS 根目录下的文件：hdfs dfs -ls / 如下所示：

```
[hadoop@hadoop000 ~]$ hadoop fs -ls /
Found 4 items
-rw-r--r--   1 hadoop supergroup        159 2017-01-15 05:11 /README.html
drwxr-xr-x   - hadoop supergroup          0 2017-01-15 05:15 /data
drwxr-xr-x   - hadoop supergroup          0 2017-01-15 05:31 /datas
-rw-r--r--   1 hadoop supergroup         40 2017-01-15 05:13 /text.log
```

如果想递归查看文件，使用 -ls -R 命令，即该命令不仅会打印出目录路径下的文件，

而且还会打印出其子目录和子目录的文件。例如我想查看 /data 下的所有文件：hadoop fs -ls -R /data 如下所示：

```
[hadoop@hadoop000 ~]$ hadoop fs -ls -R /data
drwxr-xr-x   - hadoop supergroup          0 2017-01-15 05:12 /data/input
-rw-r--r--   1 hadoop supergroup   21102856 2017-01-15 05:12 /data/input/src.zip
-rw-r--r--   1 hadoop supergroup         40 2017-01-15 05:15 /data/text.log
```

3. 在 HDFS 中创建文件夹

命令：hadoop fs -mkdir 文件夹名称

示例：在 HDFS 的根目录下创建名为 datatest 的文件夹：hadoop fs -mkdir /datatest

```
[hadoop@hadoop000 ~]$ hadoop fs -mkdir /datatest
[hadoop@hadoop000 ~]$ hadoop fs -ls /
Found 5 items
-rw-r--r--   1 hadoop supergroup        159 2017-01-15 05:11 /README.html
drwxr-xr-x   - hadoop supergroup          0 2017-01-15 05:15 /data
drwxr-xr-x   - hadoop supergroup          0 2017-01-15 05:31 /datas
drwxr-xr-x   - hadoop supergroup          0 2017-01-17 03:39 /datatest
-rw-r--r--   1 hadoop supergroup         40 2017-01-15 05:13 /text.log
```

如果我们想级联创建一个文件夹，需要在 -mkdir 命令后指定 -p 参数。例如，我们想在 HDFS 上创建这样一个目录：/datatest/mr/input，而 mr 目录在之前是不存在的，所以如果想一次性创建成功，必须加上 -p 参数，否则会报错，命令为：hadoop fs -mkdir -p /datatest /mr/input。

```
[hadoop@hadoop000 ~]$ hadoop fs -mkdir -p /datatest/mr/input
[hadoop@hadoop000 ~]$ hadoop fs -ls /datatest
Found 1 items
drwxr-xr-x  - hadoop supergroup  0 2017-01-17 03:41 /datatest/mr

[hadoop@hadoop000 ~]$ hadoop fs -ls /datatest/mr
Found 1 items
drwxr-xr-x  - hadoop supergroup  0 2017-01-17 03:41 /datatest/mr/input
```

4. 上传文件至 HDFS

命令：hadoop fs -put 源路径 目标存放路径

示例：将本地 Linux 文件系统目录下 /home/hadoop/data/ 的 input.txt 文件上传至 HDFS 文件目录 /datatest 下

```
[hadoop@hadoop000 ~]$ hdfs dfs -put /home/hadoop/data/input.txt /datatest

[hadoop@hadoop000 ~]$ hdfs dfs -ls /datatest
Found 2 items
-rw-r--r--   1 hadoop supergroup        343 2017-01-17 03:44 /datatest/input.txt
drwxr-xr-x   - hadoop supergroup          0 2017-01-17 03:41 /datatest/mr
```

5. 从 HDFS 上下载文件

命令：hdfs dfs -get HDFS 文件路径 本地存放路径

示例：将刚刚上传的 input.txt 文件下载到本地用户的目录下

```
[hadoop@hadoop000~]$ hdfs dfs -get /datatest/input.txt /home/hadoop/app
[hadoop@hadoop000~]$ ll
-rw-r--r--.  1 hadoop hadoop  343 Jan 17 03:48 input.txt
```

6. 查看 HDFS 上某个文件的内容

命令：hadoop fs -text(cat) HDFS 上的文件存放路径

示例：查看刚刚上传的 input.txt 文件

```
[hadoop@hadoop000~]$ hdfs dfs -text /datatest/input.txt
spark hadoop spark hadoop hive
hadoop kafka Hbase spark Hadoop
spark hive cisco ES hadoop flume
```

注意，text 命令和 cat 命令都可以用来查看文件内容，这里只演示了 text 命令，cat 命令请读者自己动手操作。

7. 统计目录下各文件的大小

命令：hdfs dfs -du 目录路径

示例：查看 /datas/mr/input/ 目录下各个文件的大小

```
[hadoop@hadoop000 ~]$ hdfs dfs -du /datatest/
96  96  /datatest/input.txt
0   0   /datatest/mr
```

注意：

统计目录下文件大小的单位是字节。

8. 删除 HDFS 上的某个文件或者文件夹

命令：hdfs dfs -rm(r) 文件存放路径

示例：删除刚刚上传的 input.txt 文件

```
[hadoop@hadoop000 ~]$ hdfs dfs -rm /datatest/input.txt
Deleted /datatest/input.txt

[hadoop@hadoop000 ~]$ hdfs dfs -ls /datatest/
Found 1 items
drwxr-xr-x   - hadoop supergroup          0 2017-01-17 03:41 /datatest/mr
```

在 HDFS 中删除文件与删除目录所使用命令的区别，一个是 -rm，表示删除指定的文件或者空目录；一个是 -rmr，表示递归删除指定目录下的所有子目录和文件。如下面的命令是删除 output 下的所有子目录和文件。

```
hdfs dfs -rmr /test/output
```

注意：

在生产环境中要慎用 -rmr，容易引起误删除操作。

9. 使用 help 命令寻求帮助

命令：hdfs dfs -help 命令

示例：查看 rm 命令的帮助

```
[hadoop@hadoop000 ~]$ hdfs dfs -help rm
-rm [-f] [-r|-R] [-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>
  -f          If the file does not exist, do not display a diagnostic message or
              modify the exit status to reflect an error.
  -[rR]       Recursively deletes directories
```

如上列出来的命令是我们在工作中使用频次较高的，所以大家务必要熟练掌握。如果还想学习其他 shell 命令操作，可以访问官网（http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html）、使用 help 帮助、或者在网络上搜寻，在此不再赘述。

2.2.2　Java API 访问

1. 概述

我们除使用 HDFS shell 的方式来访问 HDFS 上的数据，Hadoop 还提供以 Java API 的方式来操作 HDFS 上的数据，在生产上我们开发的大数据应用都是以代码的方式提交的，所以在代码中使用 API 的方式来操作 HDFS 数据也必须要掌握。下面介绍如何使用 Java API 对 HDFS 中的文件进行操作。

2. 搭建开发环境

我们使用 Maven 来构建 Java 应用程序，所以需要添加 maven 的依赖包如下。

代码 2.1　Maven pom 文件

```
<properties>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   <hadoop.version>2.6.0-cdh5.7.0</hadoop.version>
</properties>


<dependencies>
   <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-common</artifactId>
      <version>${hadoop.version}</version>
   </dependency>

   <dependency>
```

```
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-hdfs</artifactId>
        <version>${hadoop.version}</version>
    </dependency>

    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.10</version>
    </dependency>
</dependencies>
```

注意：

在执行单元测试之前，需要在 $HADOOP_HOME/etc/hadoop/hdfs-site.xml 中添加如下配置，并重启 HDFS 集群。

```
<property>
    <name>dfs.permissions</name>
    <value>false</value>
</property>
```

3. 单元测试的 setUp 和 tearDown 方法

在单元测试中，一般将初始化的操作放在 setUp 方法中完成，将关闭资源的操作一般放在 tearDown 方法中完成。那么我们在测试 HDFS 文件系统时，打开文件系统的操作就可以放在 setUp 中，而关闭文件系统的操作就可以放在 tearDown 中。

代码 2.2　单元测试 setup 和 teardown 方法

```
package com.kgc.bigdata.hadoop.hdfs.api;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import java.net.URI;

/**
 * HDFS Java API 操作
 */
public class HDFSApp {

    public static final String HDFS_PATH = "hdfs://hadoop000:8020";

    Configuration configuration = null;
    FileSystem fileSystem = null;
```

```
    @Before
    public void setUp() throws Exception{
        System.out.println("HDFSApp.setUp()");
        configuration = new Configuration();
        fileSystem = FileSystem.get(new URI(HDFS_PATH), configuration);
    }

    @After
    public void tearDown() throws Exception{
        fileSystem = null;
        configuration = null;
        System.out.println("HDFSApp.tearDown()");
    }
}
```

4. 使用 Java API 操作 HDFS 的常用操作

代码 2.3　Java API 操作 HDFS 文件

```
/**
 * 创建目录
 */
@Test
public void mkdir() throws Exception {
    fileSystem.mkdirs(new Path("/hdfsapi/test"));
}

/**
 * 创建文件
 */
@Test
public void create() throws Exception {
    FSDataOutputStream output = fileSystem.create(new Path("/hdfsapi/test/a.txt"));
    output.write("hello world".getBytes());
    output.flush();
    output.close();
}

/**
 * 重命名
 */
@Test
public void rename() throws Exception {
    Path oldPath = new Path("/hdfsapi/test/a.txt");
    Path newPath = new Path("/hdfsapi/test/b.txt");
    System.out.println(fileSystem.rename(oldPath, newPath));
}
```

```
/**
 * 上传本地文件到 HDFS
 */
@Test
public void copyFromLocalFile() throws Exception {
    Path src = new Path("/home/hadoop/data/hello.txt");
    Path dist = new Path("/hdfsapi/test/");
    fileSystem.copyFromLocalFile(src, dist);
}

/**
 * 查看某个目录下的所有文件
 */
@Test
public void listFiles() throws Exception {
    FileStatus[] listStatus = fileSystem.listStatus(new Path("/hdfsapi/test"));
    for (FileStatus fileStatus : listStatus) {
        String isDir = fileStatus.isDirectory() ? " 文件夹 " : " 文件 ";  // 文件 / 文件夹
        String permission = fileStatus.getPermission().toString(); // 权限
        short replication = fileStatus.getReplication(); // 副本系数
        long len = fileStatus.getLen(); // 长度
        String path = fileStatus.getPath().toString(); // 路径
        System.out.println(isDir + "\t" + permission + "\t" + replication + "\t" + len + "\t" + path);
    }
}

/**
 * 查看文件块信息
 */
@Test
public void getFileBlockLocations() throws Exception {
    FileStatus fileStatus = fileSystem.getFileStatus(new Path("/hdfsapi/test/b.txt"));
    BlockLocation[] blocks = fileSystem.getFileBlockLocations(fileStatus, 0, fileStatus.getLen());
    for (BlockLocation block : blocks) {
        for (String host : block.getHosts()) {
            System.out.println(host);
        }
    }
}
```

至此，在学习了以上相关知识后，任务 2 就可以完成了。

任务 3　HDFS 运行机制

关键步骤如下：

- 掌握 HDFS 文件读写流程。

- 认知 HDFS 的副本机制。
- 了解 HDFS 文件数据的负载均衡和机架感知。

2.3.1　HDFS 文件读写流程

1. HDFS 文件读流程

客户端读取数据过程如下：

（1）客户端通过调用 FileSystem 的 open 方法获取所需要读取的数据文件，对于 HDFS 来说该 FileSystem 就是 DistributeFileSystem；

（2）DistributeFileSystem 通过 RPC 来调用 NameNode，获取到要读取的数据文件对应的 Block 存储在哪些 DataNode 之上；

（3）客户端调用 DFSInputStream 的 read 方法，先到最佳位置（距离最近）的 DataNode，通过对数据反复调用 read 方法，可以将数据从 DataNode 传递到客户端；

（4）当读取完所有的数据之后，DFSInputStream 会关闭与 DataNode 的连接，然后寻找下一块的最佳位置，客户端只需要读取连续的流；

（5）一旦客户端完成读取操作后，就对 DFSInputStream 调用 close 方法来完成资源的关闭操作。

过程如图 2.3 所示。

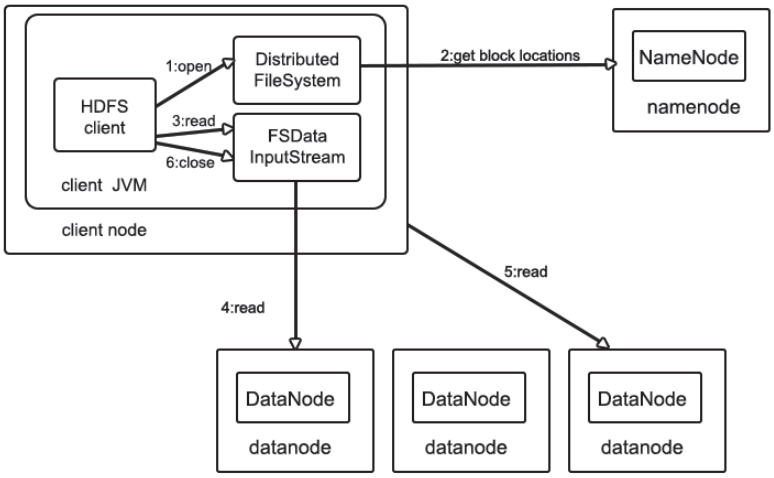

图 2.3　HDFS 数据读流程

2. HDFS 文件写流程

客户端写数据过程如下：

（1）客户端通过调用 DistributeFileSystem 的 create 方法来创建一个文件；

（2）DistributeFileSystem 会对 NameNode 发起 RPC 请求，在文件系统的命名空间中创建一个新的文件，此时会进行各种检查，比如我们要创建的文件是否已经存在，

如果该文件不存在，NameNode 就会为该文件创建一条元数据记录；

（3）客户端调用 FSDataOutputStream 的 write 方法将数据写到一个内部队列中。假设副本系数为 3，那么将队列中的数据写到各个副本对应存储的 DataNode 上；

（4）FSDataOutputStream 内部维护着一个确认队列，当接收到所有 DataNode 确认写完的消息后，该数据才会从确认队列中删除；

（5）当客户端完成数据的写入后，会对数据流调用 close 方法来关闭相关资源。

过程如图 2.4 所示。

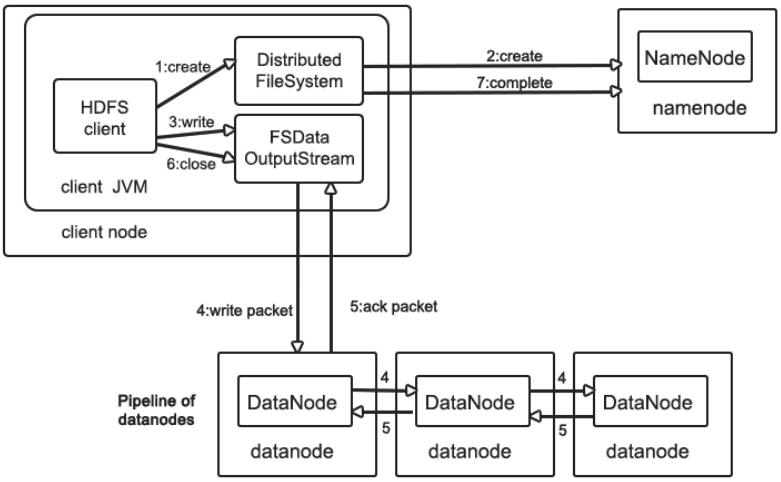

图 2.4　HDFS 数据写流程

2.3.2　HDFS 副本机制

HDFS 上的文件对应的 Block 保存多个副本，且提供容错机制，副本丢失或宕机自动恢复。默认存 3 份副本。HDFS 副本摆放机制如图 2.5 所示。

1. 副本摆放策略

第一副本：放置在上传文件的 DataNode 上；如果是集群外提交，则随机挑选一台磁盘不太慢、CPU 不太忙的节点。

第二副本：放置在与第一个副本不同的机架的节点上。

第三副本：与第二个副本相同机架的不同节点上。

如果还有更多的副本：随机放在节点中。

2. 副本系数

（1）对于上传文件到 HDFS 时，当时 Hadoop 的副本系数是几，那么这个文件的块副本数就有几份，无论以后怎么更改系统副本系数，这个文件的副本数都不会改变。也就是说上传到 HDFS 系统的文件副本数是由当时的系统副本数决定的，不会受副本系数修改影响。

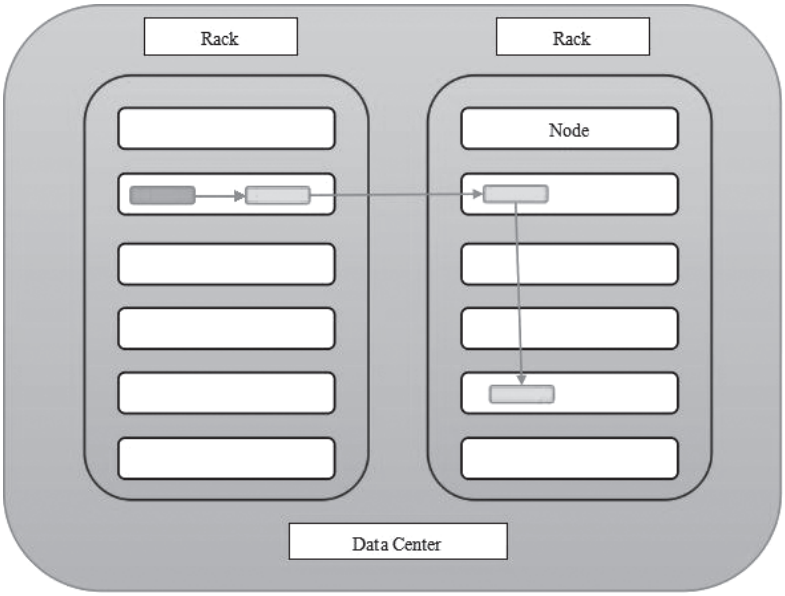

图 2.5　HDFS 副本摆放机制

（2）在上传文件时可以指定副本系数，dfs.replication 是客户端属性，不指定具体的 replication 时采用默认副本数；文件上传后，备份数已经确定，修改 dfs.replication 不会影响以前的文件，也不会影响后面指定备份数的文件，只会影响后面采用默认备份数的文件。

（3）replication 默认是由客户端决定的，如果客户端未设置才会去配置文件中读取。

（4）如果在 hdfs-site.xml 中设置了 dfs.replication=1，这也并不一定就是块的备份数是 1，因为可能没把 hdfs-site.xml 加入到工程的 classpath 里，那么我们的程序运行时取的 dfs.replication 可能是 hdfs-default.xml 中 dfs.replication 的默认值 3；可能这就是造成为什么 dfs.replication 总是 3 的原因。

```
hadoop fs setrep 3 test/test.txt
hadoop fs -ls test/test.txt
```

此时 test.txt 的副本系数就是 3 了，但是重新 put 一个到 hdfs 系统中，备份块数还是 1（假设默认 dfs.replication 的值为 1）。

2.3.3　数据负载均衡

HDFS 的架构支持数据均衡策略。如果某个 DataNode 节点上的空闲空间低于特定的临界点，按照均衡策略系统就会自动地将数据从这个 DataNode 移动到其他空闲的 DataNode。当对某个文件的请求突然增加，那么也可能启动一个计划创建该文件新的副本，并且同时重新平衡集群中的其他数据。当 HDFS 负载不均衡时，需要对 HDFS 进行数据的负载均衡调整，即对各节点机器上数据的存储分布进行调整，从而让数据

均匀的分布在各个 DataNode 上，以均衡 IO 性能、平衡 IO、平均数据、平衡集群，防止热点的发生；

在 Hadoop 中，包含一个 start-balancer.sh 脚本，通过运行这个工具，启动 HDFS 数据均衡服务。$HADOOP_HOME/bin 目录下的 start-balancer.sh 脚本就是该任务的启动脚本。启动命令为：

```
$HADOOP_HOME$HADOOP_HOME/bin/start-balancer.sh –threshold
```

影响 Balancer 的几个参数：

（1）-threshold

默认设置：10，参数取值范围：0-100

参数含义：判断集群是否平衡的阈值。理论上，该参数值越小整个集群就越平衡

（2）dfs.balance.bandwidthPerSec

默认设置：1048576（1M/S）

参数含义：Balancer 运行时允许占用的带宽

示例如下：

```
# 启动数据均衡，默认阈值为 10%
$HADOOP_HOME/bin/start-balancer.sh

# 启动数据均衡，阈值 5%
$HADOOP_HOME/bin/start-balancer.sh –threshold 5

# 停止数据均衡
$HADOOP_HOME/bin/stop-balancer.sh
```

在 hdfs-site.xml 文件中可以设置数据均衡占用的网络带宽限制

```
<property>
  <name>dfs.balance.bandwidthPerSec</name>
  <value>1048576</value>
   <description> Specifies the maximum bandwidth that each datanode can utilize for the balancing purpose in term of the number of bytes per second. </description>
</property>
```

2.3.4 机架感知

通常大型 Hadoop 集群是以机架的形式来组织的，同一个机架上的不同节点间的网络状况比不同机架之间的更为理想，NameNode 设法将数据块副本保存在不同的机架上以提高容错性。

HDFS 不能够自动判断集群中各个 DataNode 的网络拓扑情况，Hadoop 允许集群的管理员通过配置 dfs.network.script 参数来确定节点所处的机架，配置文件提供了 ip 到 rackid 的翻译。NameNode 通过这个配置知道集群中各个 DataNode 机器的 rackid。如果 topology.script.file.name 没有设定，则每个 ip 都会被翻译成 /default-rack。机架感知如图 2.6 所示。

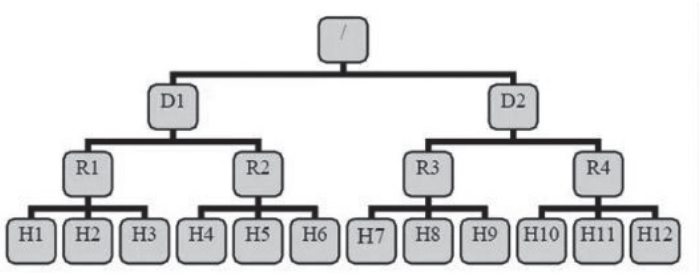

图 2.6　机架感知

图中 D 和 R 是交换机，H 是 DataNode。则 H1 的 rackid=/D1/R1/H1，有了 rackid 信息（这些 rackid 信息可以通过 topology.script.file.name 配置）就可以计算出任意两台 DataNode 之间的距离。

distance(/D1/R1/H1,/D1/R1/H1) = 0 相同的 DataNode
distance(/D1/R1/H1,/D1/R1/H2) = 2 同 rack 下的不同 DataNode
distance(/D1/R1/H1,/D1/R1/H4) = 4 同 IDC 下的不同 DataNode
distance(/D1/R1/H1,/D1/R1/H7) = 6 不同 IDC 下的 DataNode

说明：

（1）当没有配置机架信息时，所有的机器 Hadoop 都在同一个默认的机架下，名为“/default-rack”，这种情况的任何一台 DataNode 机器，不管物理上是否属于同一个机架，都会被认为是在同一个机架下。

（2）一旦配置 topology.script.file.name，就按照网络拓扑结构来寻找 DataNode；topology.script.file.name 这个配置选项的 value 指定为一个可执行程序，通常为一个脚本。

至此，在学习了以上相关知识后，任务 3 就可以完成了。

任务 4　HDFS 进阶

关键步骤如下：

- Hadoop 的序列化机制。
- SequeceFile 的使用。
- MapFile 的使用。

2.4.1　Hadoop 序列化

1. 什么是序列化和反序列化

序列化：将对象转化为字节流，以便在网络上传输或者写在磁盘上进行永久存储。

反序列化：将字节流转回成对象。

序列化在分布式数据处理的两个领域经常出现：进程间通信和永久存储。

Hadoop 中多个节点进程间通信通过远程过程调用（Remote Procedure Call，RPC）实现。

2. Hadoop 的序列化

Hadoop 的序列化不采用 Java 的序列化，而是实现了自己的序列化机制。在 Hadoop 的序列化机制中，用户可以复用对象，这就减少了 Java 对象的分配和回收，提高了应用效率。

Hadoop 通过 Writable 接口实现序列化机制，但没有提供比较功能，所以和 Java 中的 Comparable 接口合并，提供一个接口 WritableComparable。

```
public interface Writable {
  void write(DataOutput out) throws IOException;     // 状态写入到 DataOutput 二进制流
  void readFields(DataInput in) throws IOException;  // 从 DataInput 二进制流中读取状态
}

public interface WritableComparable<T> extends Writable, Comparable<T> {
}
```

3. Hadoop 的序列化案例

功能需求：序列化 Person 对象，并将序列化的对象反序列化出来。

实现步骤：

（1）序列化对象类 implements WritableComparable

（2）属性：既可以采用 Hadoop 的类型，也可以采用 Java 的类型

（3）实现 write 方法接口：将对象转换为字节流并写入到输出流 out 中

　　（a）如果属性是 Hadoop 类型的：name.write(out)。

　　（b）如果属性是 Java 类型的：out.write(name)。

（4）实现 readFields 接口：从输入流 in 中读取字节流并反序列化对象

　　（a）如果属性是 Hadoop 类型的：name.readFields(in)。

　　（b）如果属性是 Java 类型的： in.readFields(name)。

（5）实现 compare To 方法接口

（6）setter 方法：采用 Java 类型的属性，方便客户端操作

（7）构造方法

（8）实现 toString/hashCode/equals 方法

实现代码：

代码 2.4　Hadoop 序列化操作

```
package com.kgc.bigdata.hadoop.hdfs.io;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.WritableComparable;
```

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

/**
 * 序列化实体类
 */
public class Person implements WritableComparable<Person> {

    private Text name = new Text();
    private IntWritable age = new IntWritable();
    private Text sex = new Text();

    public Person(String name, int age, String sex){
        this.name.set(name);
        this.age.set(age);
        this.sex.set(sex);
    }

    public Person(Text name, IntWritable age, Text sex) {
        this.name = name;
        this.age = age;
        this.sex = sex;
    }

    public Person() {
    }

    public void set(String name, int age, String sex){
        this.name.set(name);
        this.age.set(age);
        this.sex.set(sex);
    }

    @Override
    public void write(DataOutput out) throws IOException {
        name.write(out);
        age.write(out);
        sex.write(out);
    }

    @Override
    public void readFields(DataInput in) throws IOException {
        name.readFields(in);
        age.readFields(in);
        sex.readFields(in);
```

```
}

// 比较规则：姓名相同比年龄，年龄相同比性别
@Override
public int compareTo(Person o) {
    int result = 0;

    int comp1 = name.compareTo(o.name);
    if (comp1 != 0) {
        return comp1;
    }

    int comp2 = age.compareTo(o.age);
    if (comp2 != 0) {
        return comp2;
    }

    int comp3 = sex.compareTo(o.sex);
    if (comp3 != 0) {
        return comp3;
    }
    return result;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((age == null) ? 0 : age.hashCode());
    result = prime * result + ((name == null) ? 0 : name.hashCode());
    result = prime * result + ((sex == null) ? 0 : sex.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Person other = (Person) obj;
    if (age == null) {
        if (other.age != null)
            return false;
```

```
        } else if (!age.equals(other.age))
            return false;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        if (sex == null) {
            if (other.sex != null)
                return false;
        } else if (!sex.equals(other.sex))
            return false;
        return true;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + ", sex=" + sex + "]";
    }
}
```

序列化工具类：

```
package com.kgc.bigdata.hadoop.hdfs.io;

import org.apache.hadoop.io.Writable;

import java.io.*;

/**
 * 序列化操作
 */
public class HadoopSerializationUtil {

    public static byte[] serialize(Writable writable) throws IOException {
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        DataOutputStream dataout = new DataOutputStream(out);
        writable.write(dataout);
        dataout.close();
        return out.toByteArray();
    }

    public static void deserialize(Writable writable, byte[] bytes)
            throws Exception {
        ByteArrayInputStream in = new ByteArrayInputStream(bytes);
        DataInputStream datain = new DataInputStream(in);
        writable.readFields(datain);
```

```
            datain.close();
        }

    }
```

测试类：

```
package com.kgc.bigdata.hadoop.hdfs.io;

public class Test {
    public static void main(String[] args) throws Exception {
        // 测试序列化
        Person person = new Person("zhangsan", 27, "man");
        byte[] values = HadoopSerializationUtil.serialize(person);

        // 测试反序列化
        Person p = new Person();
        HadoopSerializationUtil.deserialize(p, values);
        System.out.println(p);
    }
}
```

输出：

```
Person [name= zhangsan, age=27, sex=man]
```

2.4.2 基于文件的数据结构 SequenceFile

1. SequenceFile 概述

SequenceFile 是 Hadoop 提供的一种对二进制文件的支持。二进制文件直接将<Key, Value>对序列化到文件中。HDFS 文件系统是适合存储大文件的，很小的文件如果很多的话对于 NameNode 的压力会非常大，因为每个文件都会有一条元数据信息存储在 NameNode 上，当小文件非常多也就意味着在 NameNode 上存储的元数据信息就非常多。Hadoop 是适合存储大数据的，所以我们可以通过 SequenceFile 将小文件合并起来，可以获得更高效率的存储和计算。SequenceFile 中的 key 和 value 可以是任意类型的 Writable 或者自定义 Writable 类型。

注意：

对于一定大小的数据，比如说 100GB，如果采用 SequenceFile 进行存储的话占用的空间是大于 100GB 的，因为 SequenceFile 的存储中为了查找方便添加了一些额外的信息。

2. SequenceFile 特点

（1）支持压缩：可定制为基于 Record（记录）和 Block（块）压缩。

无压缩类型：如果没有启动压缩（默认设置），那么每个记录就由它的记录长

度（字节数）、键的长度，键和值组成，长度字段为 4 字节。SequenceFile 内部结构如图 2.7 所示。

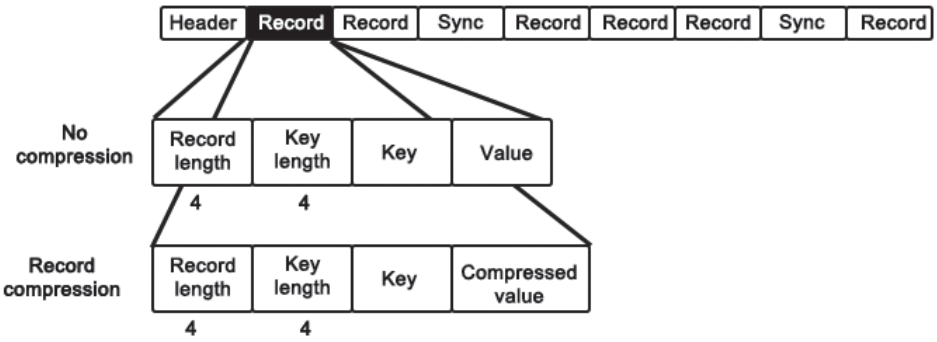

图 2.7　SequenceFile 内部结构

Record 针对行压缩，只压缩 Value 部分不压缩 Key；Block 对 Key 和 Value 都压缩。

（2）本地化任务支持：因为文件可以被切分，因此在运行 MapReduce 任务时数据的本地化情况应该是非常好的；尽可能多地发起 Map Task 来进行并行处理进而提高作业的执行效率。

（3）难度低：因为是 Hadoop 框架提供的 API，业务逻辑侧的修改比较简单。

3. SequenceFile 写操作

实现步骤：

（1）设置 Configuration

（2）获取 FileSystem

（3）设置文件输出路径

（4）SequenceFile.createWriter() 创建 SequenceFile.Write 写入

（5）调用 SequenceFile.Write.append 追加写入

（6）关闭流

代码实现：

代码 2.5　SequenceFile 文件写操作

```
package com.kgc.bigdata.hadoop.hdfs.sequence;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.SequenceFile;
import org.apache.hadoop.io.Text;
```

```
import java.net.URI;

/**
 * SequenceFile 写操作
 */
public class SequenceFileWriter {
   private static Configuration configuration = new Configuration();
   private static String url = "hdfs://hadoop000:8020";

   private static String[] data ={"a,b,c,d,e,f,g","e,f,g,h,i,j,k","l,m,n,o,p,q,r,s","t,u,v,w,x,y,z"};

   public static void main(String[] args) throws Exception{
      FileSystem fs = FileSystem.get(URI.create(url), configuration);
      Path outputPath = new Path("MySequenceFile.seq");

      IntWritable key = new IntWritable();
      Text value = new Text();

      SequenceFile.Writer writer = SequenceFile.createWriter(fs, configuration, outputPath, IntWritable.
class, Text.class);

      for (int i = 0; i < 10; i++) {
         key.set(10-i);
         value.set(data[i%data.length]);
         writer.append(key, value);
      }

      IOUtils.closeStream(writer);
   }
}
```

文件的默认 HDFS 写出路径为：/user/<user>/

4. SequenceFile 读操作

实现步骤：

（1）设置 Configuration

（2）获取 FileSystem

（3）设置文件输出路径

（4）调用 SequenceFile.Reader() 创建读取类 SequenceFile.Reader

（5）拿到 key 和 value 的 class

（6）读取

（7）关闭流

代码实现：

代码 2.6　SequenceFile 文件读操作

```
package com.kgc.bigdata.hadoop.hdfs.sequence;

import org.apache.hadoop.conf.Configuration;
```

```
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.SequenceFile;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.util.ReflectionUtils;

import java.net.URI;

/**
 * SequenceFile 读操作
 */
public class SequenceFileReader {

    static Configuration configuration = new Configuration();
    private static String url = "hdfs://hadoop000:8020";

    public static void main(String[] args) throws Exception{
        FileSystem fs = FileSystem.get(URI.create(url), configuration);
        Path inputPath = new Path("MySequenceFile.seq");

        SequenceFile.Reader reader = new SequenceFile.Reader(fs,inputPath,configuration);

        Writable keyClass = (Writable) ReflectionUtils.newInstance(reader.getKeyClass(), configuration);
            Writable valueClass = (Writable) ReflectionUtils.newInstance(reader.getValueClass(),
configuration);
        while(reader.next(keyClass, valueClass)){
            System.out.println("key : " + keyClass);
            System.out.println("value : " + valueClass);
            System.out.println("position : " + reader.getPosition());
        }

        IOUtils.closeStream(reader);
    }
}
```

5. SequenceFile 写操作使用压缩

SequenceFile 写操作的压缩支持 Record 和 Block 两种，在读取时能够自动解压。代码实现：

代码 2.7　SequenceFile 文件使用压缩写操作

```
package com.kgc.bigdata.hadoop.hdfs.sequence;


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

```
import org.apache.hadoop.io.*;
import org.apache.hadoop.io.compress.BZip2Codec;
import org.apache.hadoop.util.ReflectionUtils;

import java.net.URI;

/**
 * SequenceFile 压缩方式写操作
 */
public class SequenceFileCompression {

    static Configuration configuration = null;
    private static String url = "hdfs://hadoop000:8020";

    static {
        configuration = new Configuration();
    }

    private static String[] data = {"a,b,c,d,e,f,g", "e,f,g,h,i,j,k",
            "l,m,n,o,p,q,r,s", "t,u,v,w,x,y,z"};

    public static void main(String[] args) throws Exception {
        FileSystem fs = FileSystem.get(URI.create(url), configuration);
        Path outputPath = new Path("MySequenceFileCompression.seq");

        IntWritable key = new IntWritable();
        Text value = new Text();

        SequenceFile.Writer writer = SequenceFile.createWriter(fs,
                configuration, outputPath, IntWritable.class, Text.class,
                SequenceFile.CompressionType.RECORD, new BZip2Codec());

        for (int i = 0; i < 10; i++) {
            key.set(10 - i);
            value.set(data[i % data.length]);
            writer.append(key, value);
        }

        IOUtils.closeStream(writer);


        Path inputPath = new Path("MySequenceFileCompression.seq");
        SequenceFile.Reader reader = new SequenceFile.Reader(fs,inputPath,configuration);

        Writable keyClass = (Writable) ReflectionUtils.newInstance(reader.getKeyClass(), configuration);
            Writable valueClass = (Writable) ReflectionUtils.newInstance(reader.getValueClass(),
configuration);
        while(reader.next(keyClass, valueClass)){
```

```
            System.out.println("key : " + keyClass);
            System.out.println("value : " + valueClass);
            System.out.println("position : " + reader.getPosition());
        }

        IOUtils.closeStream(reader);
    }
}
```

2.4.3　基于文件的数据结构 MapFile

1. MapFile 概述

MapFile 是排序过后的 SequenceFile，由两部分构成，分别是 data 和 index。index 作为文件的数据索引，主要记录了每个 Record 的 key 值，以及该 Record 在文件中的偏移位置。在 MapFile 被访问的时候，索引文件会先被加载到内存，通过 index 映射关系可迅速定位到指定 Record 所在文件位置，因此，相对 SequenceFile 而言，MapFile 的检索效率更高，缺点是会消耗一部分内存来存储 index 数据。

2. MapFile 写操作

实现步骤：

（1）设置 Configuration

（2）获取 FileSystem

（3）设置文件输出路径

（4）MapFile. Writer() 创建 MapFile.Write 写入

（5）调用 MapFile.Write.append 追加写入

（6）关闭流

代码实现：

代码 2.8　MapFile 写操作

```
package com.kgc.bigdata.hadoop.hdfs.mapfile;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.MapFile;
import org.apache.hadoop.io.Text;

import java.net.URI;

/**
 * MapFile 写文件
```

```
 */
public class MapFileWriter {

    static Configuration configuration = new Configuration();
    private static String url = "hdfs://hadoop000:8020";

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

        FileSystem fs = FileSystem.get(URI.create(url), configuration);
        Path outPath = new Path("MyMapFile.map");

        Text key = new Text();
        key.set("mymapkey");
        Text value = new Text();
        value.set("mymapvalue");

        MapFile.Writer writer = new MapFile.Writer(configuration, fs,
                outPath.toString(), Text.class, Text.class);

        writer.append(key, value);
        IOUtils.closeStream(writer);
    }
}
```

3. MapFile 读操作

实现步骤：

（1）设置 Configuration

（2）获取 FileSystem

（3）设置文件输出路径

（4）MapFile. Reader() 创建 MapFile.Reader 写入

（5）拿到 Key 与 Value 的 class

（6）读取

（7）关闭流

代码实现：

代码 2.9　MapFile 读操作

```
package com.kgc.bigdata.hadoop.hdfs.mapfile;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.MapFile;
```

```
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.util.ReflectionUtils;

import java.net.URI;

/**
 * MapFile 读文件
 */
public class MapFileReader {

    static Configuration configuration = new Configuration();
    private static String url = "hdfs://hadoop000:8020";

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

        FileSystem fs = FileSystem.get(URI.create(url), configuration);
        Path inPath = new Path("MyMapFile.map");

        MapFile.Reader reader = new MapFile.Reader(fs, inPath.toString(),
                configuration);

        Writable keyclass = (Writable) ReflectionUtils.newInstance(
                reader.getKeyClass(), configuration);
        Writable valueclass = (Writable) ReflectionUtils.newInstance(
                reader.getValueClass(), configuration);

        while (reader.next((WritableComparable) keyclass, valueclass)) {
            System.out.println(keyclass);
            System.out.println(valueclass);
        }
        IOUtils.closeStream(reader);
    }
}
```

至此，在学习了以上相关知识后，任务 4 就可以完成了。

本章总结

本章学习了以下知识点：

- HDFS 文件产生背景、设计目标及特点。
- HDFS 的基本概念。
- HDFS 的体系结构及各组件功能。
- 使用 HDFS shell 以及 Java API 操作 HDFS 文件系统。

- HDFS 文件数据读写流程。
- HDFS 副本机制、数据负载均衡以及机架感知。
- HDFS 序列化。
- SequenceFile 以及 MapFile 的读写操作。

本章作业

在生产环境中，输入数据往往是由许多小文件组成，这里的小文件指小于 HDFS 系统块大小的文件，然而每一个存储在 HDFS 中的文件、目录和块都映射为一个对象，存储在 NameNode 服务器内存中，通常占用 150 个字节。如果有 1 千万个文件，就需要消耗大约 3G 的内存空间，如果是 10 亿个文件呢，简直不可想象。所以在项目开始前，我们选择一种适合的方案来解决小文件问题。请开发一个应用程序来对小文件进行合并。