大数据技术

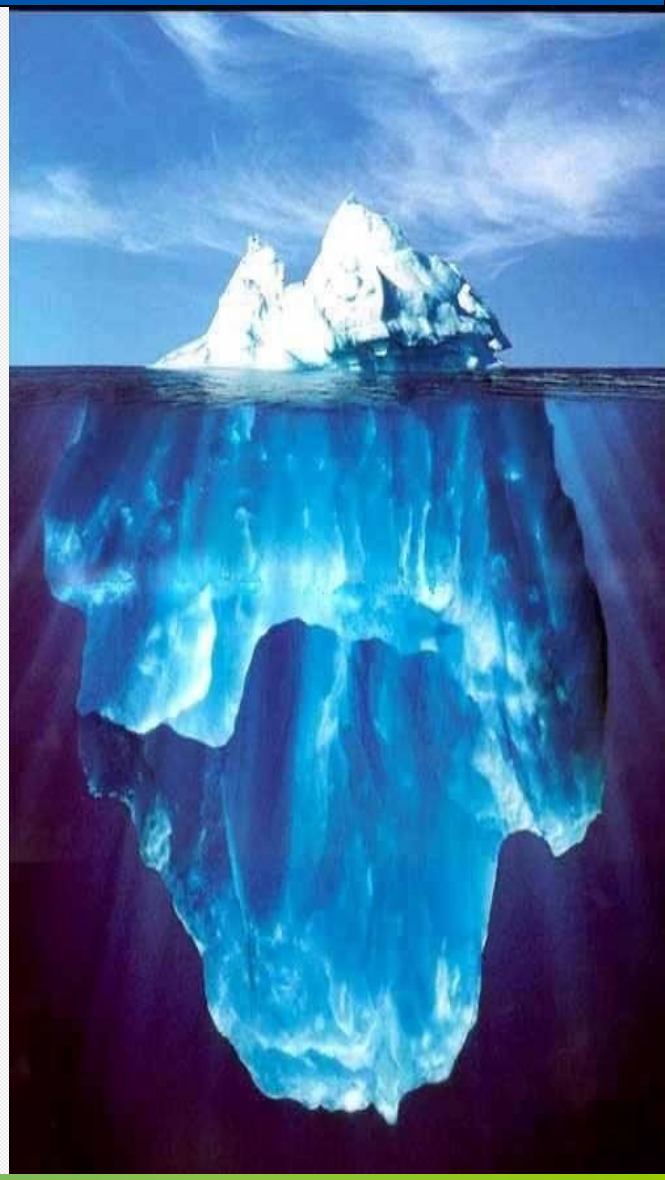
第3章 分布式文件系统-HDFS

洪韬

E-mail: 2993400893@qq.com

提纲

- 3.1 分布式文件系统
- 3.2 HDFS简介
- 3.3 HDFS相关概念
- 3.4 HDFS体系结构
- 3.5 HDFS存储原理
- 3.6 HDFS数据读写过程
- 3.7 HDFS编程实践



3.1 分布式文件系统

分布式系统 (distributed system) 是建立在网络之上的软件系统；

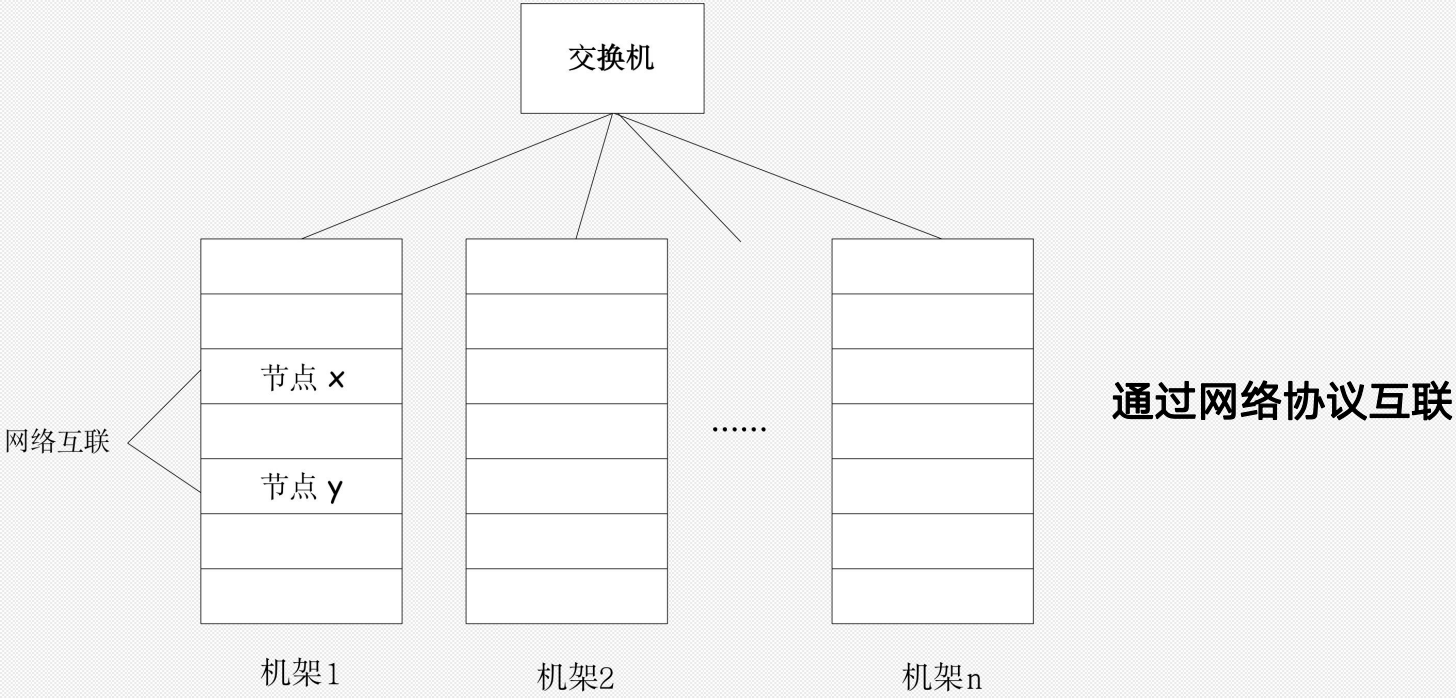
文件系统 (file system) 是操作系统用于明确存储设备（磁盘，或者基于NAND Flash的固态硬盘）或分区上的文件的方法和数据结构，即在存储设备上组织文件的方法。

分布式文件系统（distributed file system）是一种通过网络实现文件在多台主机上进行分布式存储的文件系统；

3.1.1 计算机集群结构

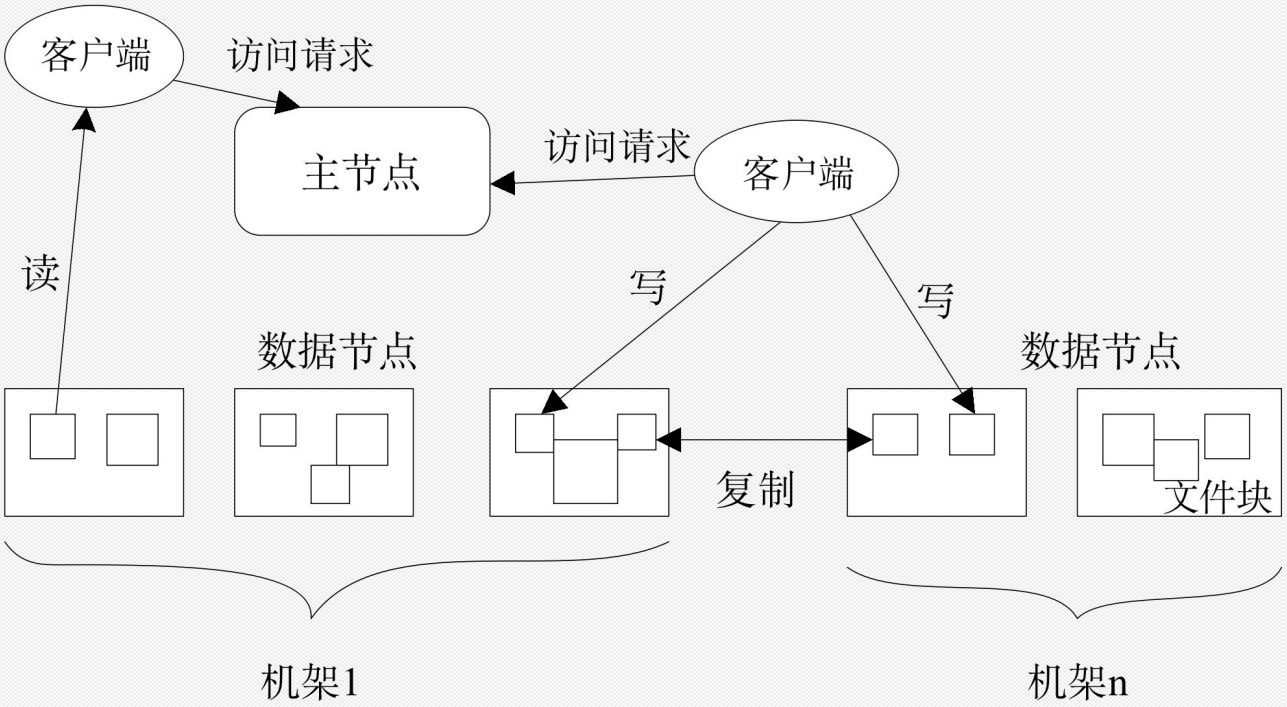
•分布式文件系统把文件分布存储到多个计算机节点上，成千上万的计算机节点构成计算机集群

•与之前**使用多个处理器和专用高级硬件的并行化处理装置**不同的是，目前的分布式文件系统所采用的计算机集群，都是由普通硬件构成的，这就大大降低了硬件上的开销



3.1.2 分布式文件系统的结构

分布式文件系统在物理结构上是由计算机集群中的多个节点构成的，这些节点分为两类，一类叫“**主节点**”(Master Node)或者也被称为“**名称结点**”(NameNode)，另一类叫“**从节点**”（Slave Node）或者也被称为“**数据节点**”(DataNode)



3.2 HDFS简介

总体而言，HDFS(Hadoop Distributed File System)要实现以下目标：

- 兼容廉价的硬件设备
- 流数据读写
- 大数据集
- 简单的文件模型
- 强大的跨平台兼容性

- 降低数据存储成本，方便伸缩扩容
- 只能从指定位置开始读写
- 支持GB、TB级别等大文件
- 一次写入，多次读取
- Java跨平台特性

HDFS特殊的设计，在实现上述优良特性的同时，也使得自身具有一些应用局限性，主要包括以下几个方面：

- 不适合低延迟数据访问
- 无法高效存储大量小文件
- 不支持多用户写入及任意修改文件

？

- 数据交互依赖网络
- 文件和数据块之间映射关系随文件数目成正相关
- 并发控制简单，同一时间一个文件只能有一个写入者

机制相对简单、廉价部署、具有跨平台和容错能力的大规模分布式集群文件系统。

3.3.1HDFS相关概念-块

传统的文件系统中，一般以数据块为单位，一般在几千个字节左右。以块(block)作为存储单位，可以最小化寻址开销，HDFS默认一个块128MB，大小通过配置可调；对于存储空间未达到数据块大小的文件，这个文件也不会占用整个数据块的存储空间；

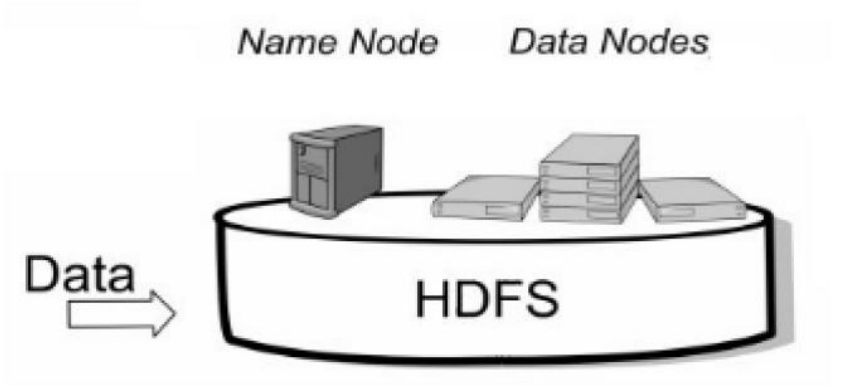
HDFS采用抽象的块概念可以带来以下几个好处：

- **支持大规模文件存储**：文件以块为单位进行存储，一个大规模文件可以被分拆成若干个文件块，不同的文件块可以被分发到不同的节点上，因此，一个文件的大小不会受到单个节点的存储容量的限制，可以远远大于网络中任意节点的存储容量
- **简化系统设计**：首先，大大简化了存储管理，因为文件块大小是固定的，这样就可以很容易计算出一个节点可以存储多少文件块；其次，方便了元数据的管理，元数据不需要和文件块一起存储，可以由其他系统负责管理元数据
- **适合数据备份**：每个文件块都可以冗余存储到多个节点上，大大提高了系统的容错性和可用性

3.3.2HDFS相关概念-名称节点和数据节点

从类型上讲，**元数据**有三类重要信息：
第一类是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**等。
第二类记录文件内容存储相关信息，例如**文件块情况、副本个数**、副本所在的Data Node信息等。
第三类用来记录HDFS中所有**Data Node信息**，用于Data Node管理。

HDFS主要组件的功能

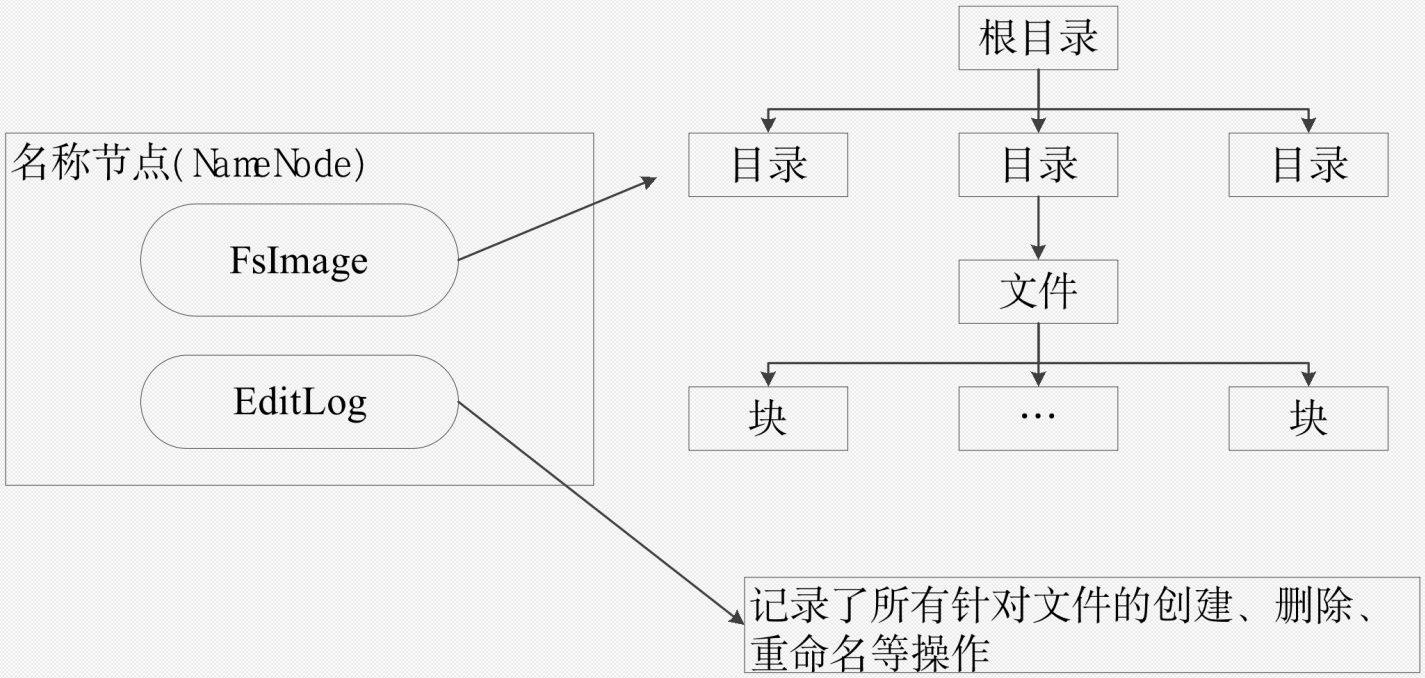


NameNode	DataNode
存储元数据	存储文件内容
元数据在内存中加载	文件内容保存在磁盘
保存文件、block、datanode之间的映射关系	负责处理客户端的读写请求，根据namenode执行块的创建、删除和复制操作

3.3.2HDFS相关概念-名称节点和数据节点

名称节点的数据结构

- 在HDFS中，**名称节点（NameNode）**负责管理分布式文件系统的命名空间（Namespace），保存了两个核心的数据结构，即FsImage和EditLog
  - FsImage用于维护文件系统树以及文件树中所有的文件和文件夹的元数据
  - 操作日志文件EditLog中记录了所有针对文件的创建、删除、重命名等操作
- 名称节点记录了每个文件中各个块所在的数据节点的位置信息



3.3.2HDFS相关概念-名称节点和数据节点

FsImage文件

•FsImage文件包含文件系统中所有目录和文件inode的序列化形式。每个inode是一个文件或目录的元数据的内部表示，并包含此类信息：文件的复制等级、修改和访问时间、访问权限、块大小以及组成文件的块。对于目录，则存储修改时间、权限和配额元数据

•FsImage文件持久化存储着块、文件以及所存放DataNode节点之间的映射关系。这些信息在HDFS启动后被加载到内存中。

3.3.2 名称节点和数据节点

名称节点的启动

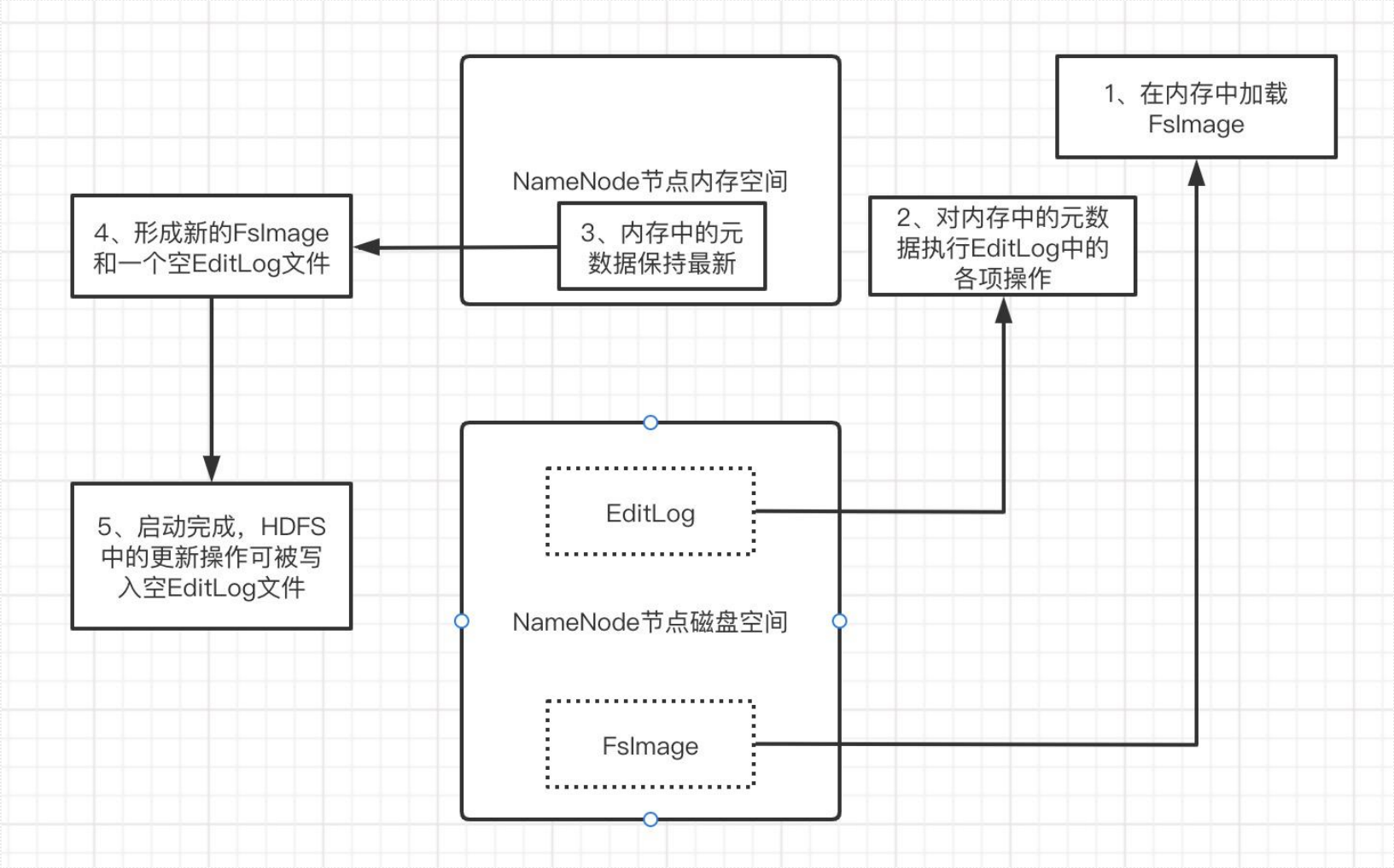
•在名称节点启动的时候，它会将FsImage文件中的内容加载到内存中，之后再执行EditLog文件中的各项操作，使得内存中的元数据和实际的同步，名称节点在启动过程中处于“**安全模式**”,只对外提供**读操作**

•一旦在内存中成功建立文件系统元数据的映射，则创建一个新的FsImage文件和一个空的EditLog文件

•名称节点启动过之后，HDFS中的更新操作会重新写到EditLog文件中，因为FsImage文件一般都很大（GB级别的很常见），如果所有的更新操作都往FsImage文件中添加，这样会导致系统运行的十分缓慢，但是，如果往EditLog文件里面写就不会这样，因为EditLog 要小很多。

3.3.2 名称节点和数据节点

名称节点的启动



3.3.2 名称节点和数据节点

名称节点运行期间EditLog不断变大的问题

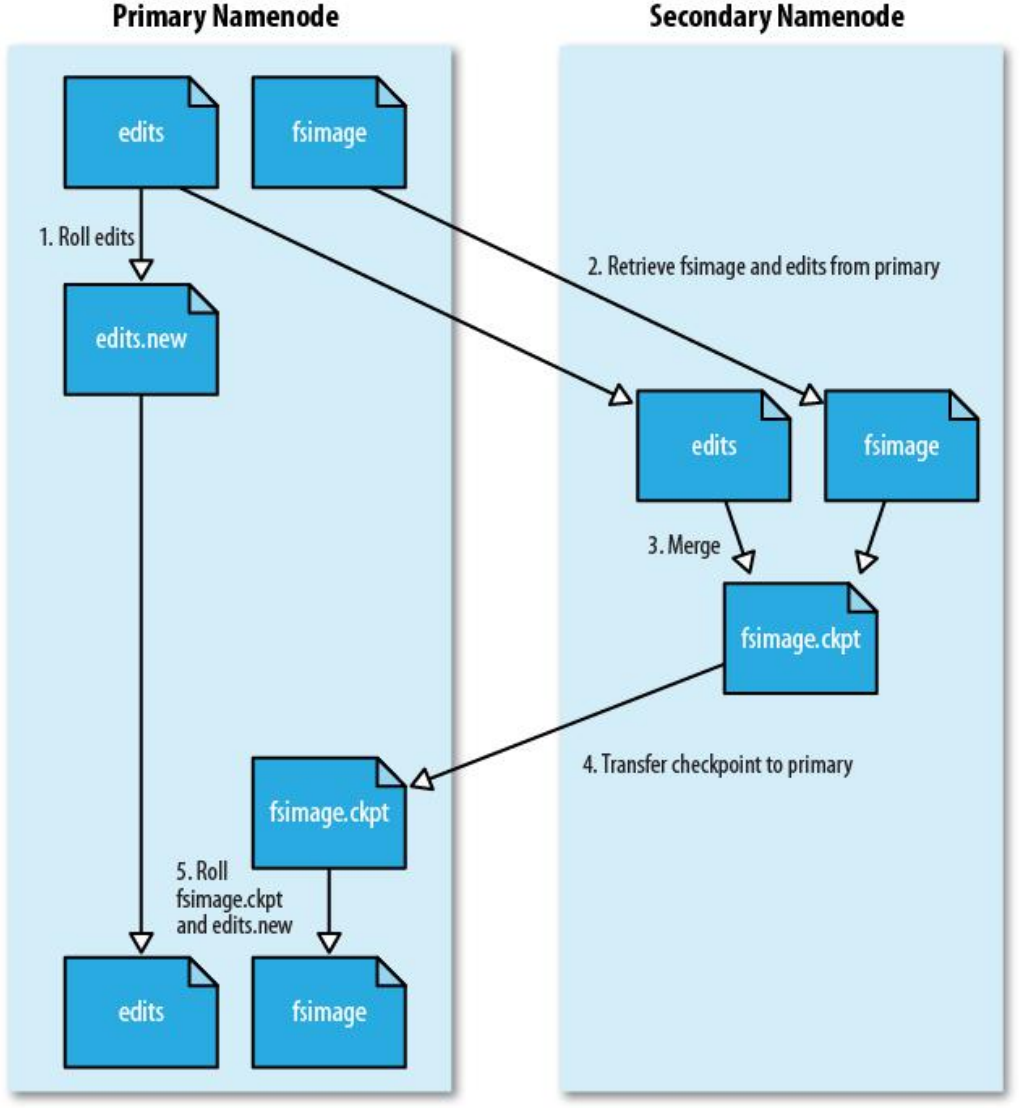
•在名称节点运行期间，HDFS的所有更新操作都是直接写到EditLog中，久而久之， EditLog文件将会变得很大

•虽然这对名称节点运行时候是没有什么明显影响的，但是，当名称节点重启的时候，名称节点需要先将FsImage里面的所有内容映像到内存中，然后再一条一条地执行EditLog中的记录，当EditLog文件非常大的时候，会导致名称节点启动操作非常慢，而在这段时间内HDFS系统处于“安全模式”，一直无法对外提供写操作，影响了用户的使用

如何解决？答案是：SecondaryNameNode第二名称节点

第二名称节点是HDFS架构中的一个组成部分，它是用来保存名称节点中对HDFS 元数据信息的备份，并减少名称节点重启的时间。
SecondaryNameNode一般是单独运行在一台机器上

3.3.2 名称节点和数据节点



SecondaryNameNode的工作情况：

（1）SecondaryNameNode会定期和NameNode通信，请求其停止使用EditLog文件，暂时将新的写操作写到一个新的文件edit.new上来，这个操作是瞬间完成，上层写日志的函数完全感觉不到差别；

（2）SecondaryNameNode通过HTTP GET方式从NameNode上获取到FsImage和EditLog文件，并下载到本地的相应目录下；

（3）SecondaryNameNode将下载下来的FsImage载入到内存，然后一条一条地执行EditLog文件中的各项更新操作，使得内存中的FsImage保持最新；这个过程就是EditLog和FsImage文件合并；

（4）SecondaryNameNode执行完（3）操作之后，会通过post方式将新的FsImage文件发送到NameNode节点上

（5）NameNode将从SecondaryNameNode接收到的新的FsImage替换旧的FsImage文件，同时将edit.new替换EditLog文件，通过这个过程EditLog就变小了

3.3.2 名称节点和数据节点

3.3.2 名称节点和数据节点

3.3.2 名称节点和数据节点

3.3.2 名称节点和数据节点

数据节点（DataNode）

•数据节点是分布式文件系统HDFS的工作节点，负责数据的存储和读取，会根据客户端或者是名称节点的调度来进行数据的存储和检索

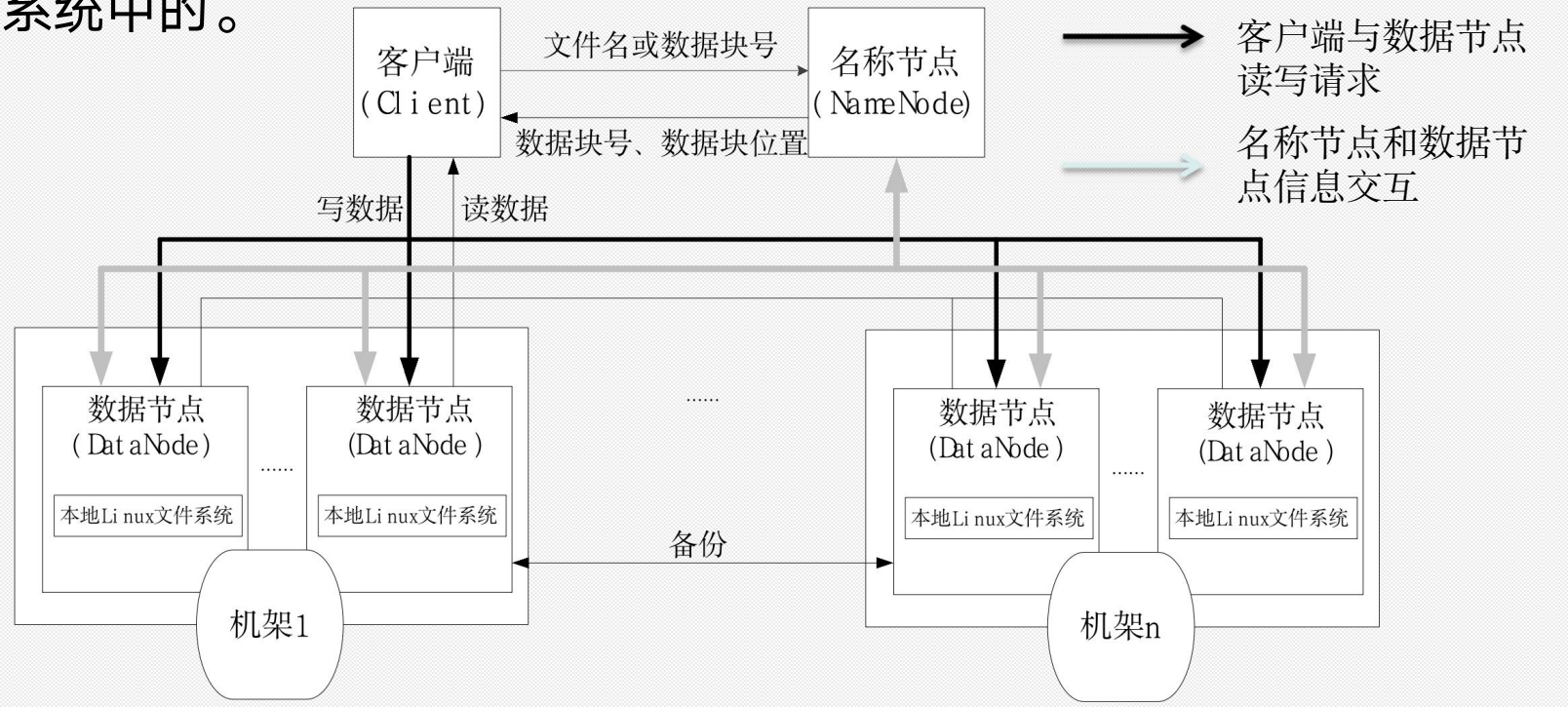
•每个数据节点中的数据会被保存在各自节点的本地文件系统中

3.4 HDFS体系结构

- 3.4.1 HDFS体系结构概述
- 3.4.2 HDFS命名空间管理
- 3.4.3 通信协议
- 3.4.4 客户端
- 3.4.5 HDFS体系结构的局限性

3.4.1 HDFS体系结构概述

HDFS采用了主从（Master/Slave）结构模型，一个HDFS集群包括一个名称节点（NameNode）和若干个数据节点（DataNode）（如下图所示）。名称节点作为**中心服务器**，负责管理文件系统的命名空间及客户端对文件的访问。集群中的数据节点一般是一个节点运行一个数据节点进程，负责处理文件系统客户端的读/写请求，在名称节点的统一调度下进行数据块的创建、删除和复制等操作。每个数据节点的数据实际上是保存在本地文件系统中的。



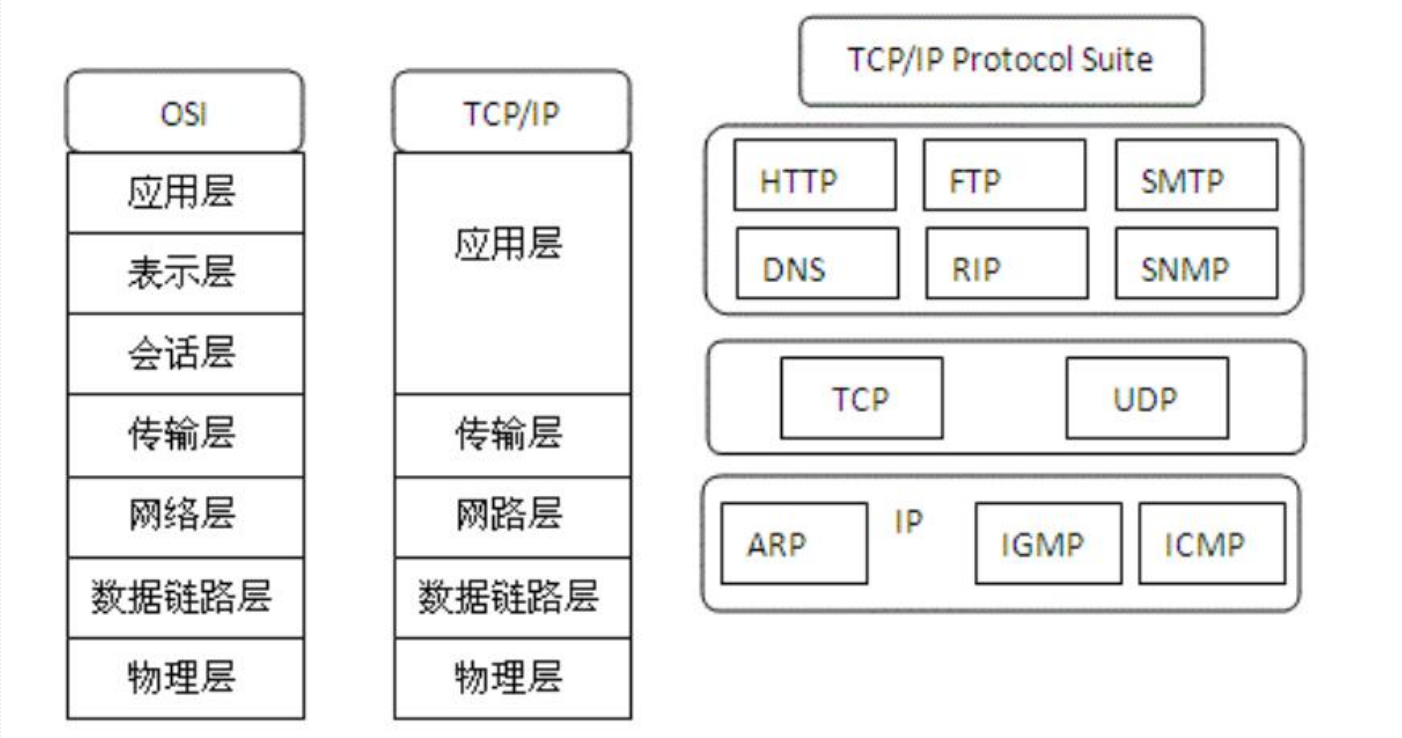
3.4.2 HDFS命名空间管理

- HDFS的命名空间包含目录、文件和块
- 在HDFS1.0体系结构中，在整个HDFS集群中只有一个命名空间，并且只有唯一一个名称节点，该节点负责对这个命名空间进行管理
- HDFS使用的是传统的分级文件体系，因此，用户可以像使用普通文件系统一样，创建、删除目录和文件，在目录间转移文件，重命名文件等

3.4.3 通信协议

- HDFS是一个部署在集群上的分布式文件系统，因此，很多数据需要通过网络进行传输
- 所有的HDFS通信协议都是构建在TCP/IP协议基础之上的
- 客户端通过一个可配置的端口向名称节点主动发起TCP连接，并使用客户端协议与名称节点进行交互
- 名称节点和数据节点之间则使用数据节点协议进行交互
- 客户端与数据节点的交互是通过RPC（Remote Procedure Call）来实现的。在设计上，名称节点不会主动发起RPC，而是响应来自客户端和数据节点的RPC请求

3.4.3 通信协议



网络模型

3.4.3 通信协议

早期单机时代，一台电脑上运行多个进程，大家各干各的，老死不相往来。假如A进程需要一个画图的功能，B进程也需要一个画图的功能，程序员就必须为两个进程都写一个画图的功能。这不是整人么？于是就出现了IPC（Inter-process communication，单机中运行的进程之间的相互通信）。OK，现在A既然有了画图的功能，B就调用A进程上的画图功能好了，程序员终于可以偷下懒了。

到了网络时代，大家的电脑都连起来了。以前程序只能调用自己电脑上的进程，能不能调用其他机器上的进程呢？于是就程序员就把IPC扩展到网络上，这就是RPC（远程过程调用）了。现在不仅单机上的进程可以相互通信，多机器中的进程也可以相互通信了。

3.4.4 客户端

- 客户端是用户操作HDFS最常用的方式，HDFS在部署时都提供了客户端
- HDFS客户端是一个库，暴露了HDFS文件系统接口，这些接口隐藏了HDFS实现中的大部分复杂性
- 严格来说，客户端并不算是HDFS的一部分
- 客户端可以支持打开、读取、写入等常见的操作，并且提供了类似Shell的命令行方式来访问HDFS中的数据
- 此外，HDFS也提供了Java API，作为应用程序访问文件系统的客户端编程接口

3.4.5 HDFS体系结构的局限性

HDFS只设置唯一一个名称节点，这样做虽然大大简化了系统设计，但也带来了一些明显的**局限性**，具体如下：

（1）**命名空间的限制**：名称节点是保存在内存中的，因此，名称节点能够容纳的对象（文件、块）的个数会受到内存空间大小的限制。

（2）**性能的瓶颈**：整个分布式文件系统的吞吐量，受限于单个名称节点的吞吐量。

（3）**隔离问题**：由于集群中只有一个名称节点，只有一个命名空间，因此，无法对不同应用程序进行隔离。

（4）**集群的可用性**：一旦这个唯一的名称节点发生故障，会导致整个集群变得不可用。

3.5 HDFS存储原理

- 3.5.1 冗余数据保存
- 3.5.2 数据存取策略
- 3.5.3 数据错误与恢复

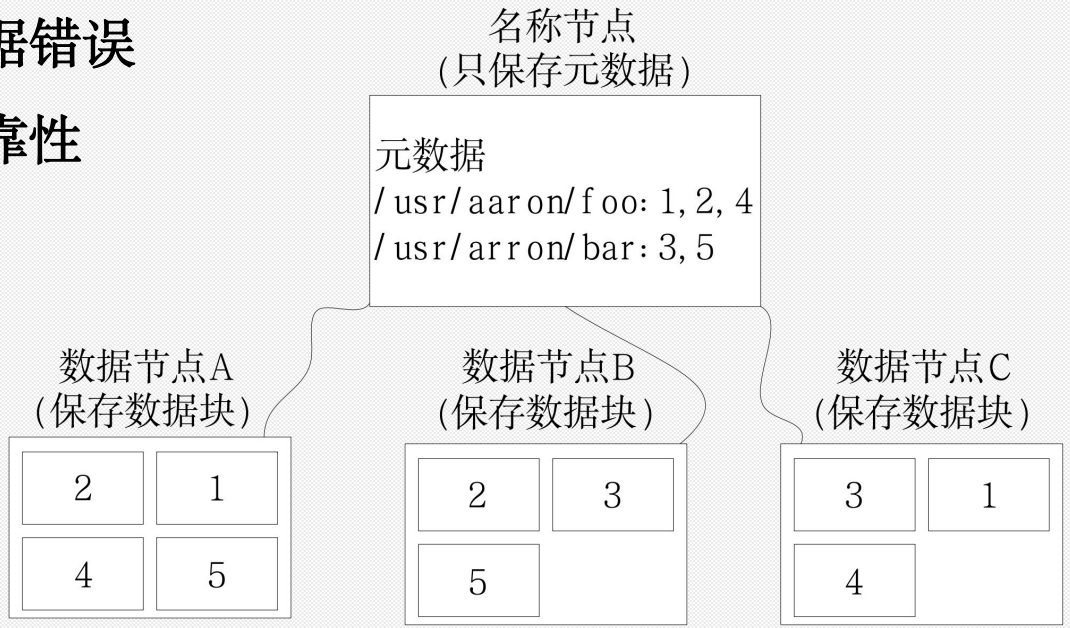
3.5.1冗余数据保存

作为一个分布式文件系统，为了保证系统的容错性和可用性，HDFS采用了多副本方式对数据进行冗余存储，通常一个数据块的多个副本会被分布到不同的数据节点上，如下图所示，数据块1被分别存放到数据节点A和C上，数据块2被存放在数据节点A和B上。这种多副本方式具有以下几个优点：

(1) **加快数据传输速度**

(2) **容易检查数据错误**

(3) **保证数据可靠性**

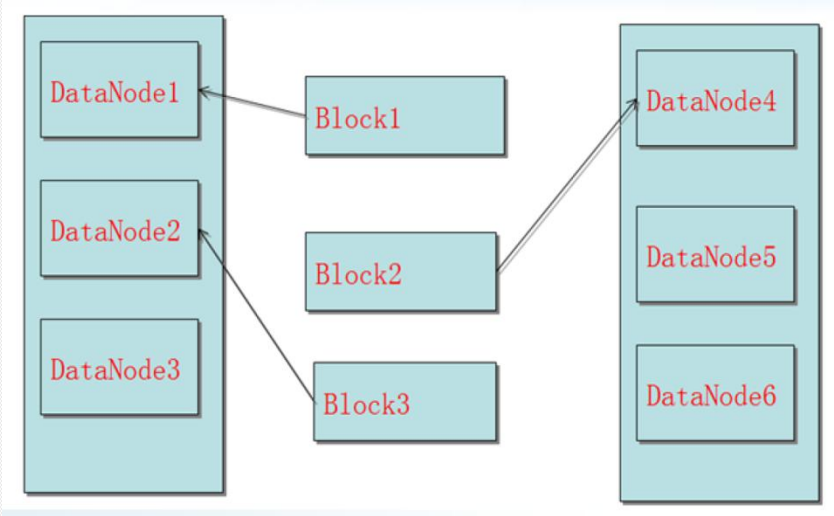


3.5.2数据存取策略

1.数据存放

•第一个副本：集群内的数据写请求，则放置在上传文件的数据节点；如果是集群外数据写请求，则随机挑选一台磁盘不太满、CPU不太忙的节点
•第二个副本：放置在与第一个副本不同的机架的节点上
•第三个副本：与第一个副本相同机架的其他节点上
•更多副本：随机节点

Block的副本放置策略



3.5.2数据存取策略

2. 数据读取

•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

3.5.3数据错误与恢复

HDFS具有较高的容错性，可以兼容廉价的硬件，它把硬件出错看作一种常态，而不是异常，并设计了相应的机制检测数据错误和进行自动恢复，主要包括以下几种情形：名称节点出错、数据节点出错和数据出错。

1. 名称节点出错

名称节点保存了所有的元数据信息，其中，最核心的两大数据结构是FsImage和Editlog，如果这两个文件发生损坏，那么整个HDFS实例将失效。因此，HDFS设置了备份机制，把这些核心文件同步复制到备份服务器SecondaryNameNode上。当名称节点出错时，就可以根据备份服务器SecondaryNameNode中的FsImage和Editlog数据进行恢复。同时还有一种将NN上的所有元数据同步存储到其他文件系统的方式。由于SecondaryNameNode可能与NN有一定的数据偏差，所以一般会将两种方式结合使用，将远程文件系统备份的元数据放到第二名称节点上进行恢复，并将第二名称节点作为名称节点使用。

3.5.3数据错误与恢复

2. 数据节点出错

•每个数据节点会定期向名称节点发送“心跳”信息，向名称节点报告自己的状态
•当数据节点发生故障，或者网络发生断网时，名称节点就无法收到来自一些数据节点的心跳信息，这时，这些数据节点就会被标记为“宕机”，节点上面的所有数据都会被标记为“不可读”，名称节点不会再给它们发送任何I/O请求
•这时，有可能出现一种情形，即由于一些数据节点的不可用，会导致一些数据块的副本数量小于冗余因子
•名称节点会定期检查这种情况，一旦发现某个数据块的副本数量小于冗余因子，就会启动数据冗余复制，为它生成新的副本
•HDFS和其它分布式文件系统的最大区别就是可以调整冗余数据的位置

3.5.3数据错误与恢复

3. 数据出错

- 网络传输和磁盘错误等因素，都会造成数据错误
- 客户端在读取到数据后，会采用md5和sha1对数据块进行校验，以确定读取到正确的数据
- 在文件被创建时，客户端就会对每一个文件块进行信息摘录，并把这些信息写入到同一个路径的隐藏文件里面
- 当客户端读取文件的时候，会先读取该信息文件，然后，利用该信息文件对每个读取的数据块进行校验，如果校验出错，客户端就会请求到另外一个数据节点读取该文件块，并且向名称节点报告这个文件块有错误，名称节点会定期检查并且重新复制这个块。

课后作业

- 作业一

结合课堂所讲等分布式文件系统以及HDFS特性、优缺点。如果让你设计一个分布式文件系统，你会有哪些考虑，设置那些属性并如何统筹这些属性？

- 作业二

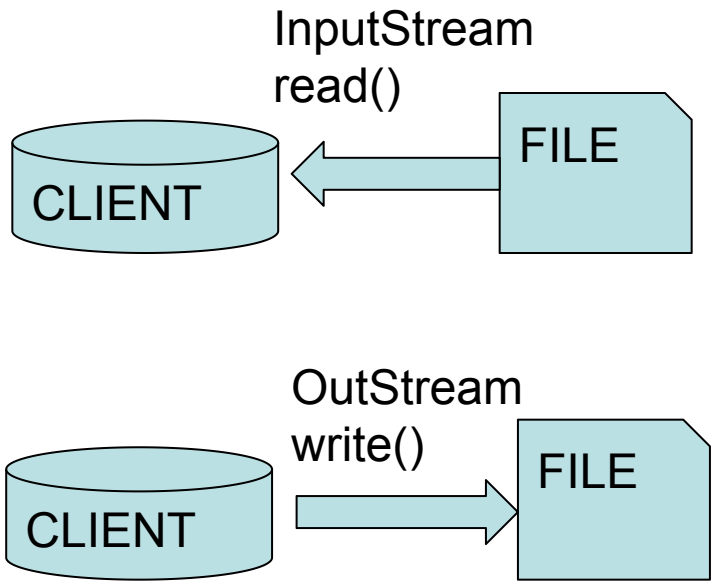
巩固复习HDFS主节点数据结构和第二名称节点工作流程。要求自己画出HDFS结构以及第二名称节点工作流程。

3.6 HDFS数据读写过程

- 3.6.1 读数据的过程
- 3.6.2 写数据的过程

3.6 HDFS数据读写过程

流式读取



3.6 HDFS数据读写过程

读取文件

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.FSDataInputStream;
public class Chapter3 {
        public static void main(String[] args) {
                try {
                        Configuration conf = new Configuration();
                        conf.set("fs.defaultFS","hdfs://localhost:9000");
                        conf.set("fs.hdfs.impl","org.apache.hadoop.hdfs.DistributedFileSystem");
                        FileSystem fs = FileSystem.get(conf);
                        Path file = new Path("test");
                        FSDataInputStream getIt = fs.open(file);
                        BufferedReader d = new BufferedReader(new InputStreamReader(getIt));
                        String content = d.readLine(); //读取文件一行
                        System.out.println(content);
                        d.close(); //关闭文件
                        fs.close(); //关闭hdfs
                } catch (Exception e) {
                        e.printStackTrace();
                } }} }
```

3.6 HDFS数据读写过程

写入文件

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.Path;
public class Chapter3 {
    public static void main(String[] args) {
        try {
            Configuration conf = new Configuration();
            conf.set("fs.defaultFS","hdfs://localhost:9000");
            conf.set("fs.hdfs.impl","org.apache.hadoop.hdfs.DistributedFileSystem");
            FileSystem fs = FileSystem.get(conf);
            byte[] buff = "Hello world".getBytes(); // 要写入的内容
            String filename = "test"; //要写入的文件名
            FSDataOutputStream os = fs.create(new Path(filename));
            os.write(buff,0,buff.length);
            System.out.println("Create:"+ filename);
            os.close();
            fs.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    } }
```

3.6 HDFS数据读写过程

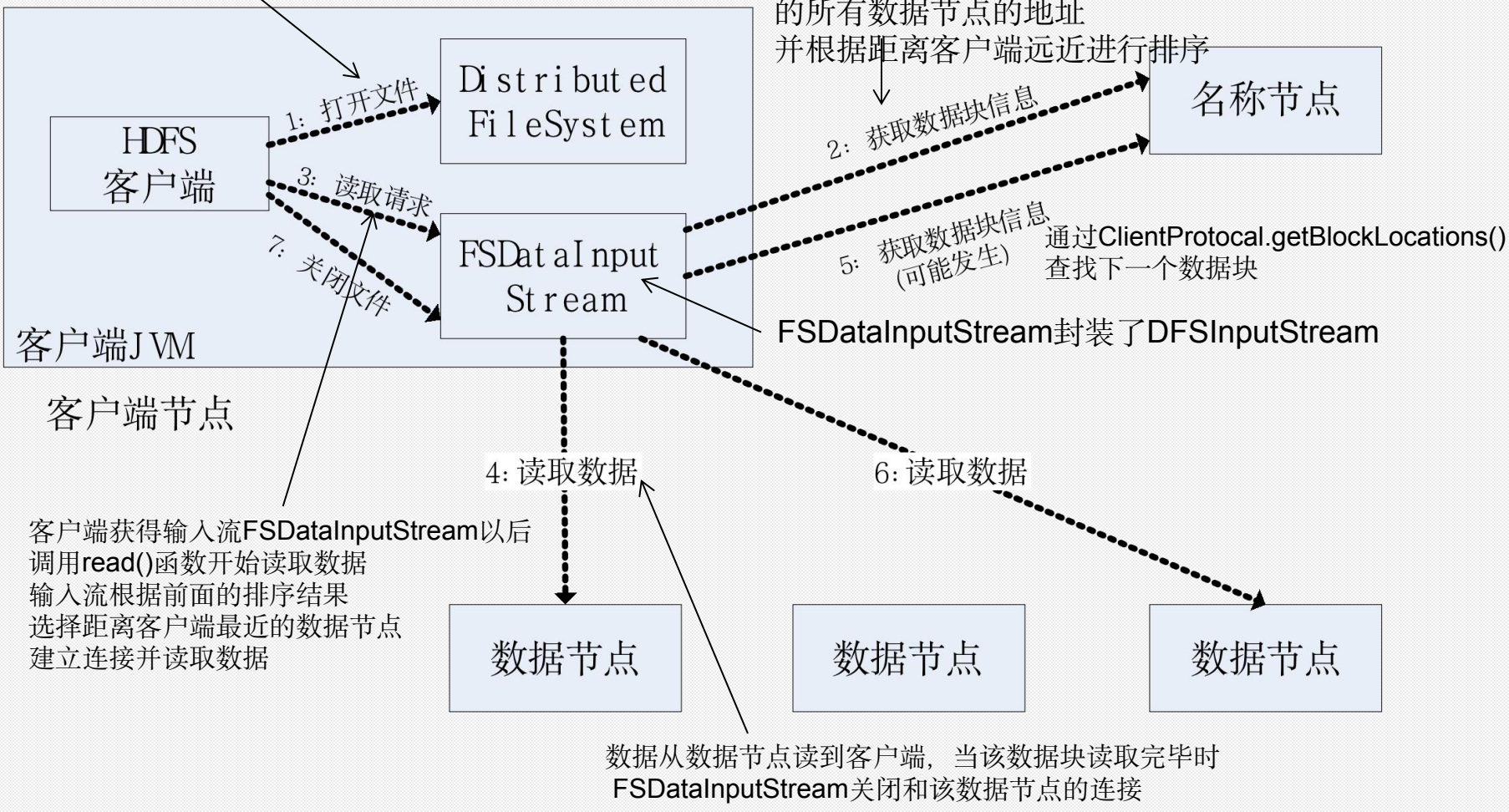
•FileSystem是一个通用文件系统的抽象基类，可以被分布式文件系统继承，所有可能使用Hadoop文件系统的代码，都要使用这个类
•Hadoop为FileSystem这个抽象类提供了多种具体实现
•DistributedFileSystem就是FileSystem在HDFS文件系统中的具体实现
•FileSystem的open()方法返回的是一个输入流FSDataInputStream对象，在HDFS文件系统中，具体的输入流就是DFSInputStream；FileSystem中的create()方法返回的是一个输出流FSDataOutputStream对象，在HDFS文件系统中，具体的输出流就是DFSOutputStream。

```
Configuration conf = new Configuration();
conf.set("fs.defaultFS","hdfs://localhost:9000");
conf.set("fs.hdfs.impl","org.apache.hadoop.hdfs.DistributedFileSystem");
FileSystem fs = FileSystem.get(conf);
FSDataInputStream in = fs.open(new Path(uri));
FSDataOutputStream out = fs.create(new Path(uri));
```

3.6.1读数据的过程

```
import org.apache.hadoop.fs.FileSystem
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
FSDataInputStream in = fs.open(new Path(uri));
```

DFSInputStream的构造方法中通过ClientProtocal.getBlockLocations()
远程调用名称节点，获得文件开始部分数据块的位置
对于该数据块，名称节点返回保存该数据块的所有数据节点的地址
并根据距离客户端远近进行排序

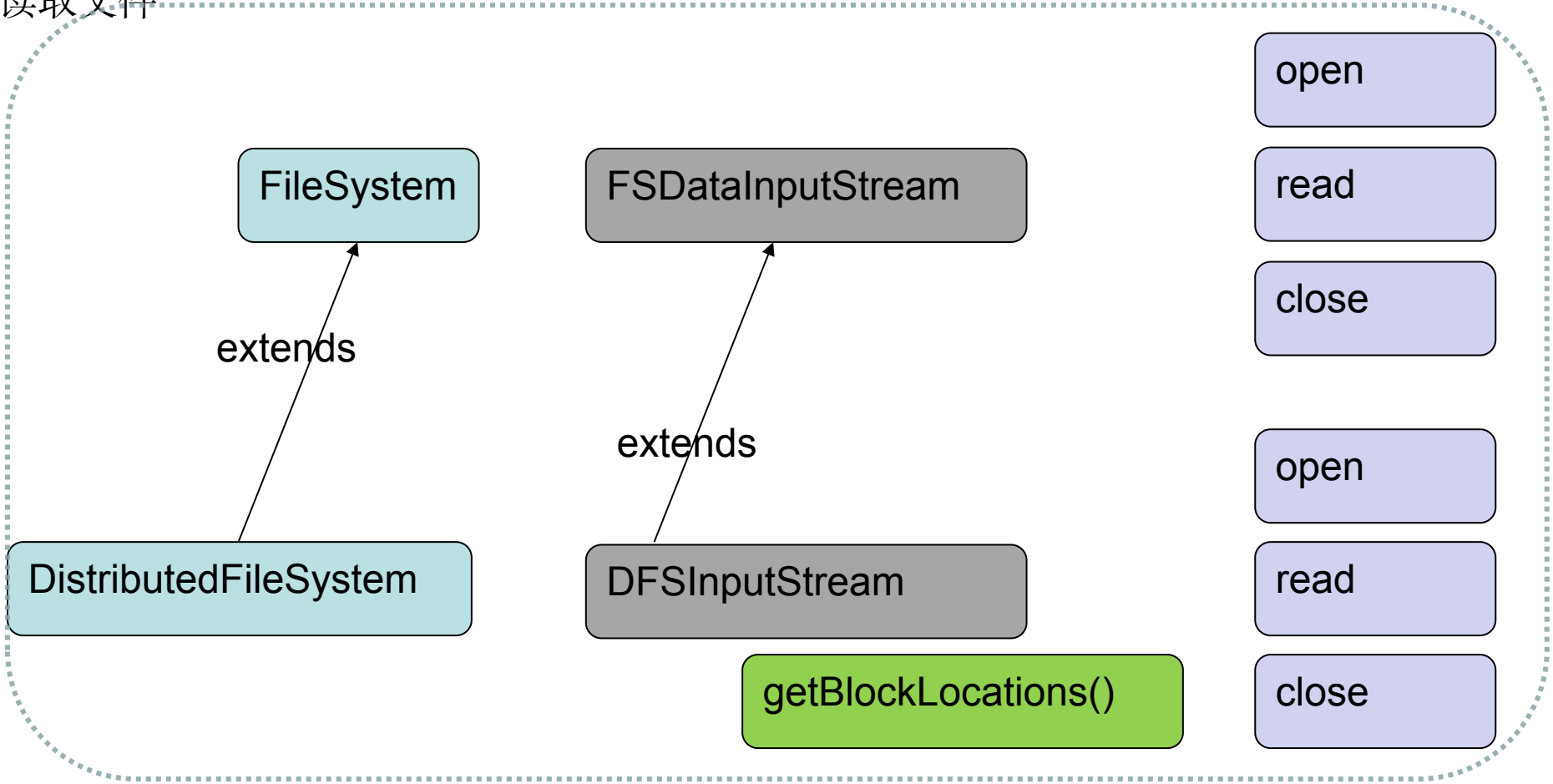


客户端获得输入流FSDataInputStream以后
调用read()函数开始读取数据
输入流根据前面的排序结果
选择距离客户端最近的数据节点
建立连接并读取数据

数据从数据节点读到客户端，当该数据块读取完毕时
FSDataInputStream关闭和该数据节点的连接

3.6.1 读数据过程

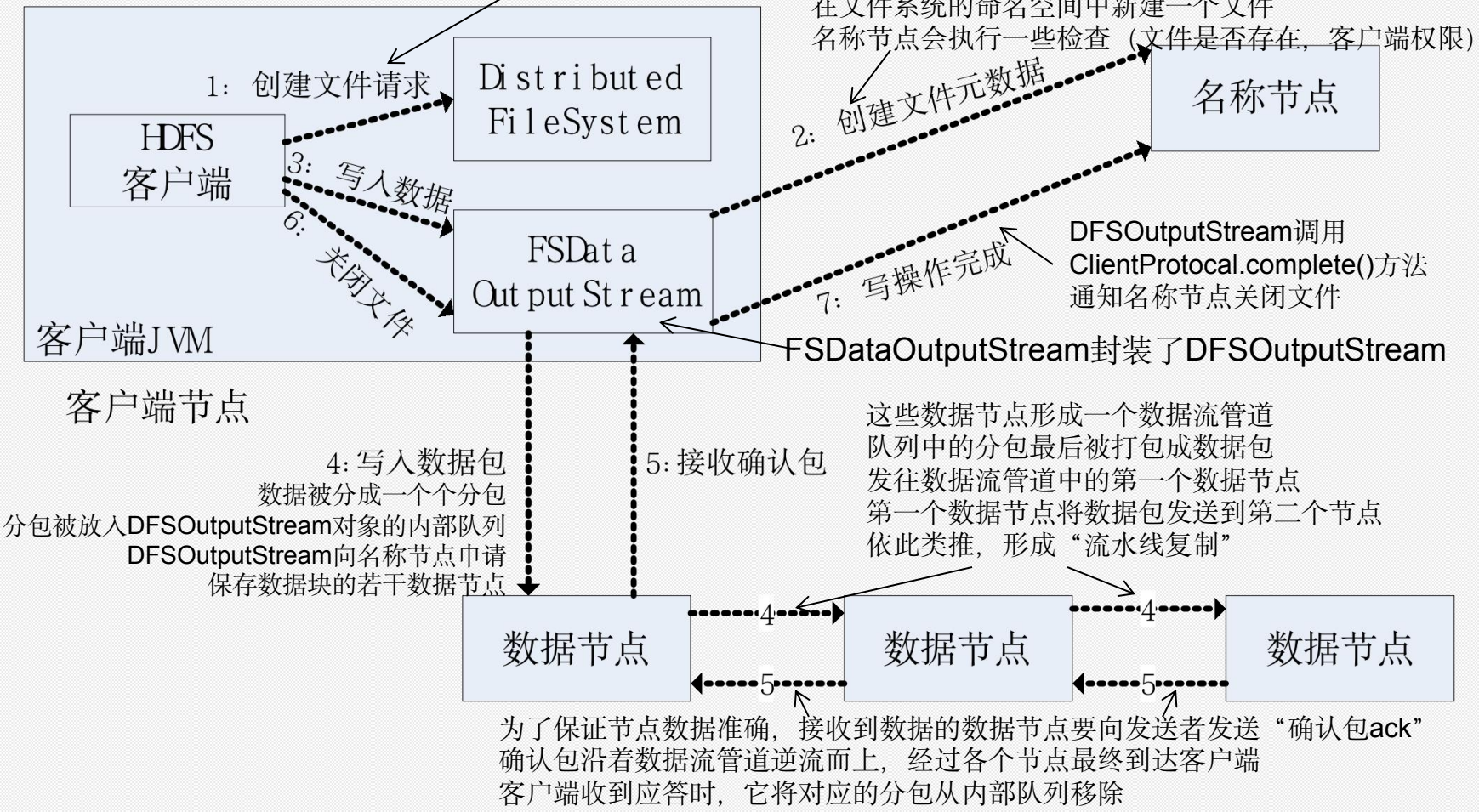
读取文件



一些细节的过程，比如获取文件对应块的存储信息、存储节点的选择，这些过程被代码隐藏了。

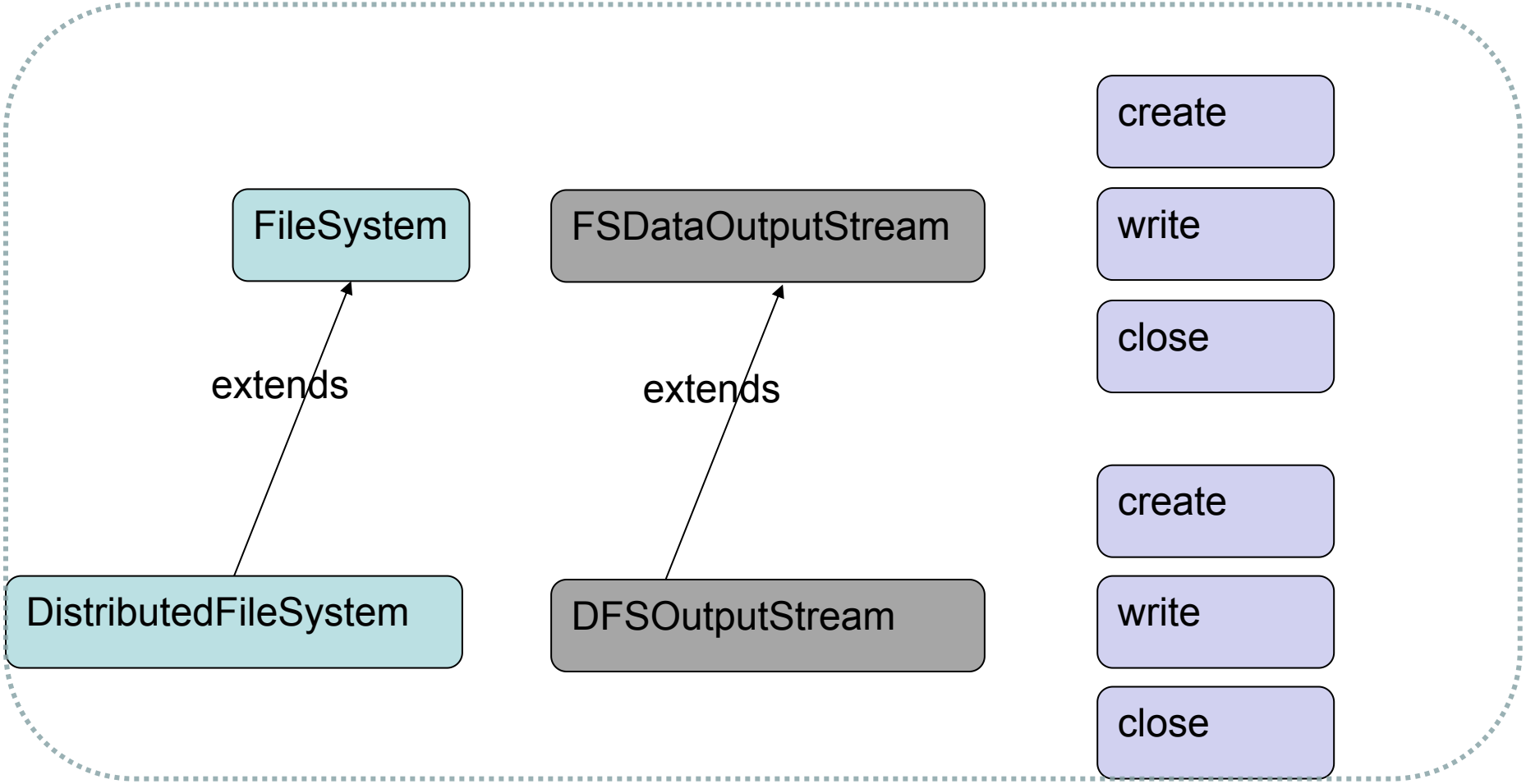
3.6.2 写数据的过程

```
import org.apache.hadoop.fs.FileSystem
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
FSDataOutputStream out = fs.create(new Path(uri));
```



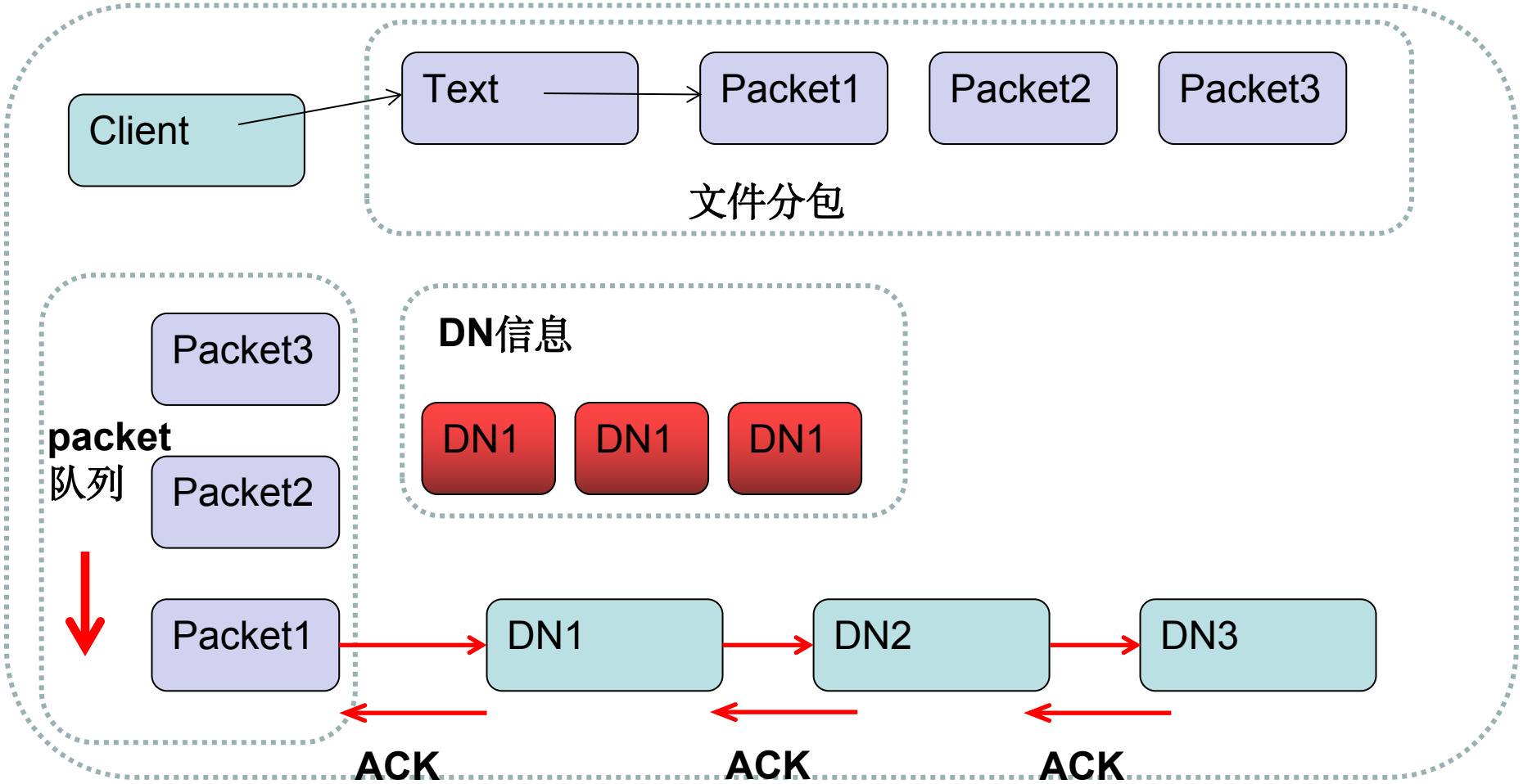
3.6.2 写数据的过程

写入文件



3.6.2 写数据的过程

写入过程示意图



3.7 HDFS编程实践

Hadoop提供了关于HDFS在Linux操作系统上进行文件操作的常用Shell命令以及Java API。同时还可以利用Web界面查看和管理Hadoop文件系统

备注：Hadoop安装成功后，已经包含HDFS和MapReduce，不需要额外安装。而HBase等其他组件，则需要另外下载安装。

在学习HDFS编程实践前，我们需要启动Hadoop。执行如下命令：

```
$ cd /usr/local/hadoop
$ ./bin/hdfs namenode -format    #格式化hadoop的hdfs文件系统
$ ./sbin/start-dfs.sh #启动hadoop
```

3.7.1HDFS常用命令

HDFS有很多shell命令，其中，fs命令可以说是HDFS最常用的命令

利用该命令可以查看HDFS文件系统的目录结构、上传和下载数据、创建文件等。该命令的用法为：

hadoop fs [genericOptions] [commandOptions]

备注：Hadoop中有三种Shell命令方式：

•hadoop fs适用于任何不同的文件系统，比如本地文件系统和HDFS文件系统

hadoop dfs只能适用于HDFS文件系统

hdfs dfs跟hadoop dfs的命令作用一样，也只能适用于HDFS文件系统

3.7.1HDFS常用命令

实例：

hadoop fs -ls <path>:显示<path>指定的文件的详细信息

hadoop fs -mkdir <path>:创建<path>指定的文件夹

```
hadoop@ubuntu:/usr/local/hadoop$ ./bin/hadoop fs -mkdir
hdfs://localhost:9000/tempDir
hadoop@ubuntu:/usr/local/hadoop$ ./bin/hadoop fs -ls hdf
s://localhost:9000/
Found 6 items
drwxr-xr-x   - hadoop supergroup          0 2020-02-06 1
0:04 hdfs://localhost:9000/bigdatacase
drwxr-xr-x   - hadoop supergroup          0 2020-02-19 1
9:39 hdfs://localhost:9000/hbase
drwxr-xr-x   - hadoop supergroup          0 2020-02-27 1
0:57 hdfs://localhost:9000/tempDir
drwx-wx-wx   - hadoop supergroup          0 2020-01-30 0
9:38 hdfs://localhost:9000/tmp
drwxr-xr-x   - hadoop supergroup          0 2020-01-30 1
9:13 hdfs://localhost:9000/user
drwxr-xr-x   - hadoop supergroup          0 2020-01-30 1
9:21 hdfs://localhost:9000/usr
```

3.7.1HDFS常用命令

实例：

hadoop fs -cat <path>:将<path>指定的文件的内容输出到标准输出（stdout）

hadoop fs -copyFromLocal <localsrc> <dst>:将本地源文件<localsrc>复制到路径<dst>指定的文件或文件夹中

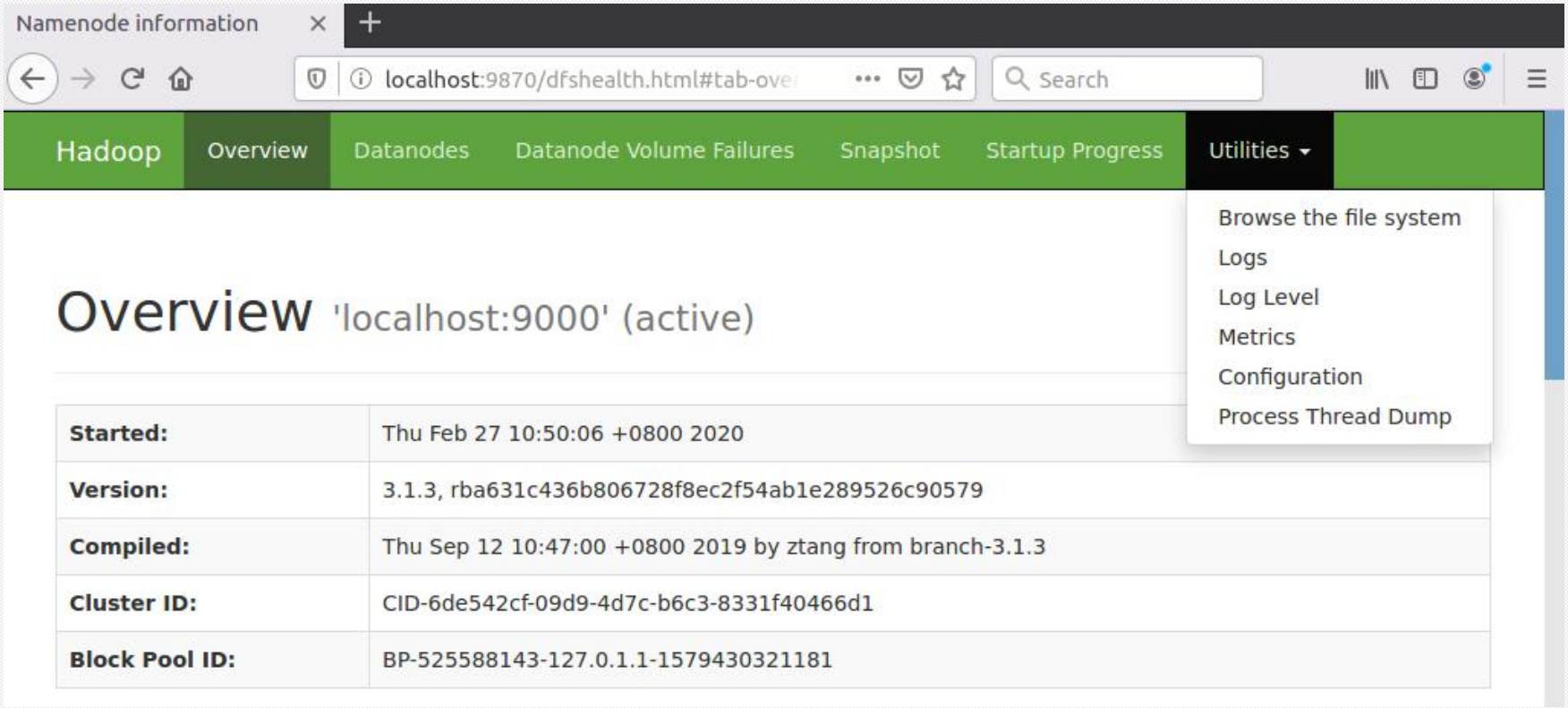
```
hadoop@ubuntu:/usr/local/hadoop$ ./bin/hadoop fs -copyFromLocal /home/ha
doop/mergefile/* hdfs://localhost:9000/tempDir
```

```
hadoop@ubuntu:/usr/local/hadoop$ ./bin/hadoop fs -cat hdfs://localhost:9
000/tempDir/*2020-02-27 11:08:25,938 INFO sasl.SaslDataTransferClient: S
ASL encryption trust check: localHostTrusted = false, remoteHostTrusted
= false
this is file1.txt
this is file2.txt
this is file3.txt
this is file4.abc
this is file5.abc
```

3.7.2HDFS的Web界面

在配置好Hadoop集群之后，可以通过浏览器登录“http://localhost:9870”访问HDFS文件系统

通过Web界面的”Utilities”菜单下面的“Browse the filesystem”查看文件



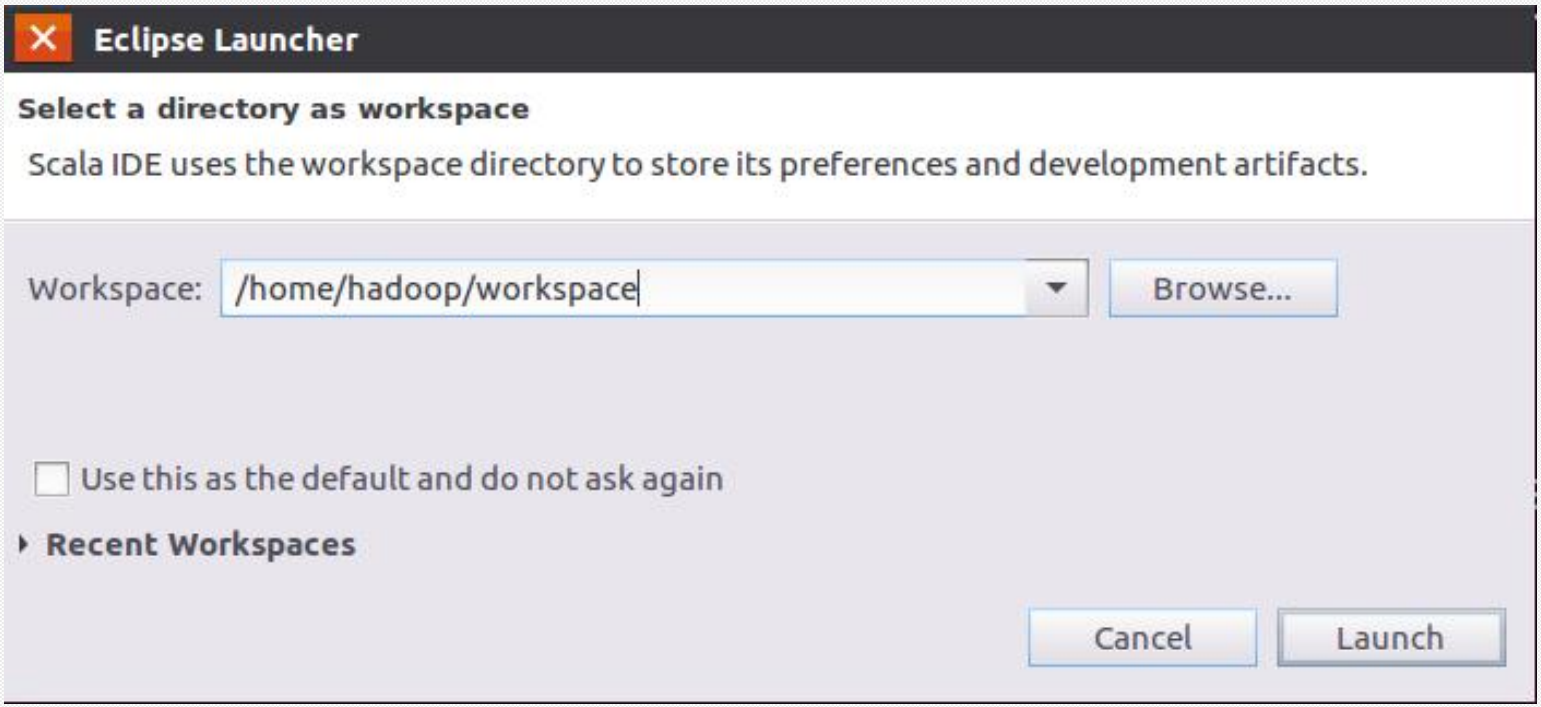
3.7.3HDFS常用Java API及应用实例

现在要执行的任务是：假设在目录“hdfs://localhost:9000/user/hadoop”下面有几个文件，分别是file1.txt、file2.txt、file3.txt、file4.abc和file5.abc，这里需要从该目录中过滤出所有后缀名不为“.abc”的文件，对过滤之后的文件进行读取，并将这些文件的内容合并到文件“hdfs://localhost:9000/user/hadoop/merge.txt”中。

3.7.3HDFS常用Java API及应用实例

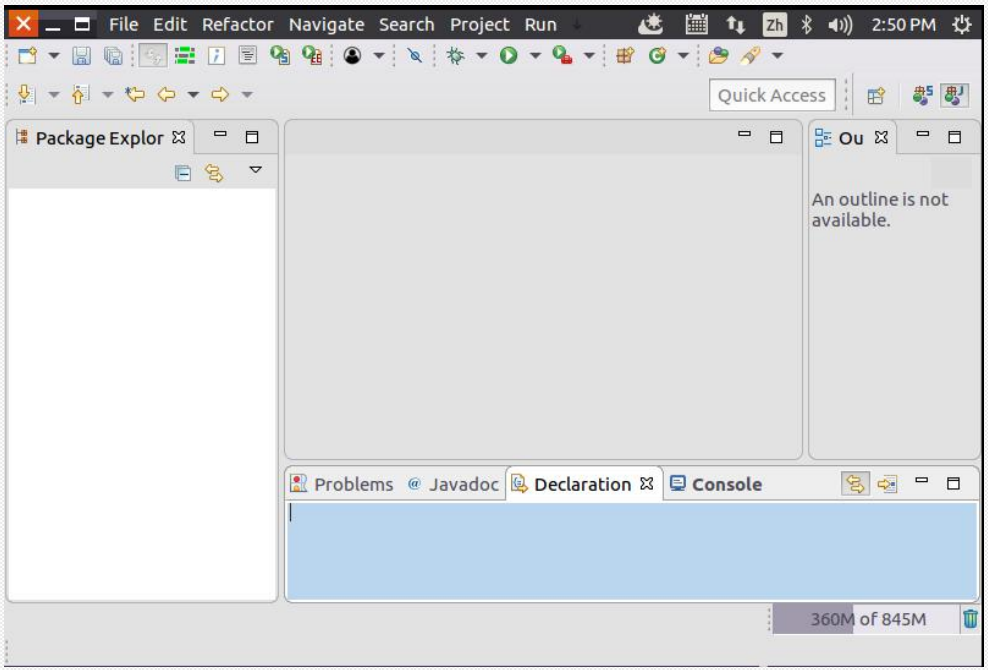
一、在Eclipse中创建项目

启动Eclipse。当Eclipse启动以后，会弹出如下图所示界面，提示设置工作空间（workspace）。



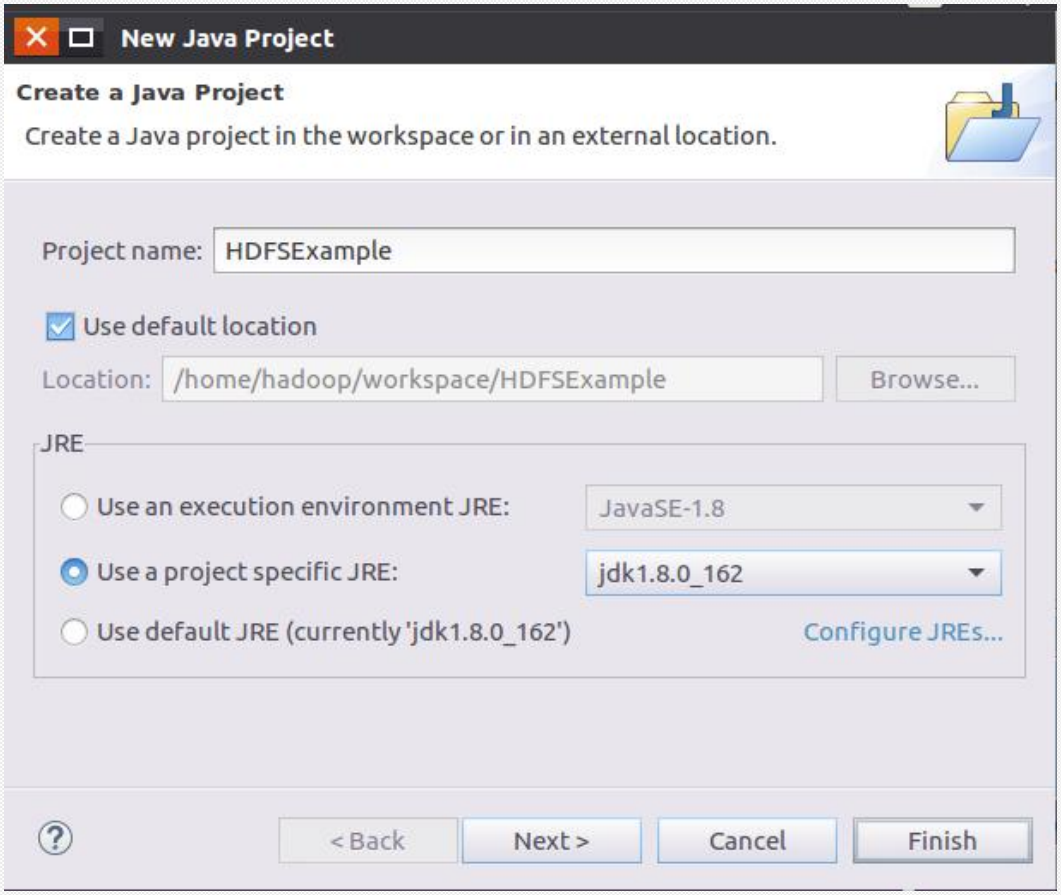
3.7.3HDFS常用Java API及应用实例

可以直接采用默认的设置“/home/hadoop/workspace”，点击“Launch”按钮。可以看出，由于当前是采用hadoop用户登录了Linux系统，因此，默认的工作空间目录位于hadoop用户目录“/home/hadoop”下。
Eclipse启动以后，会呈现如下图所示的界面。



3.7.3HDFS常用Java API及应用实例

选择“File->New->Java Project”菜单，开始创建一个Java工程，会弹出如下图所示界面。



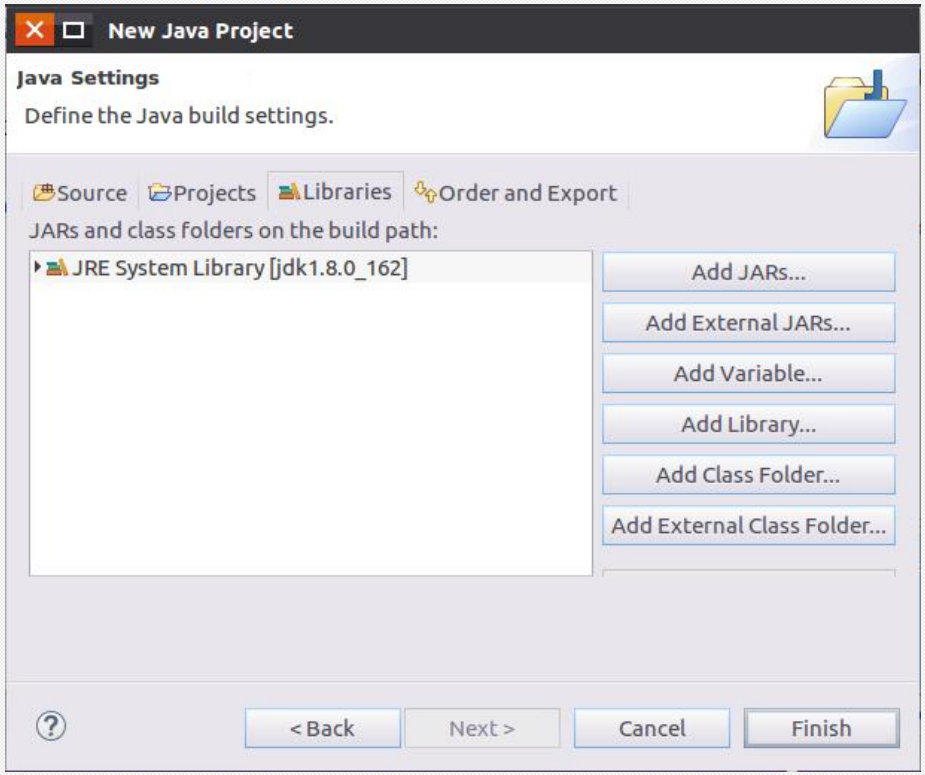
3.7.3HDFS常用Java API及应用实例

在“Project name”后面输入工程名称“HDFSExample”，选中“Use default location”，让这个Java工程的所有文件都保存到“/home/hadoop/workspace/HDFSExample”目录下。在“JRE”这个选项卡中，可以选择当前的Linux系统中已经安装好的JDK，比如jdk1.8.0_162。然后，点击界面底部的“Next>”按钮，进入下一步的设置。

3.7.3HDFS常用Java API及应用实例

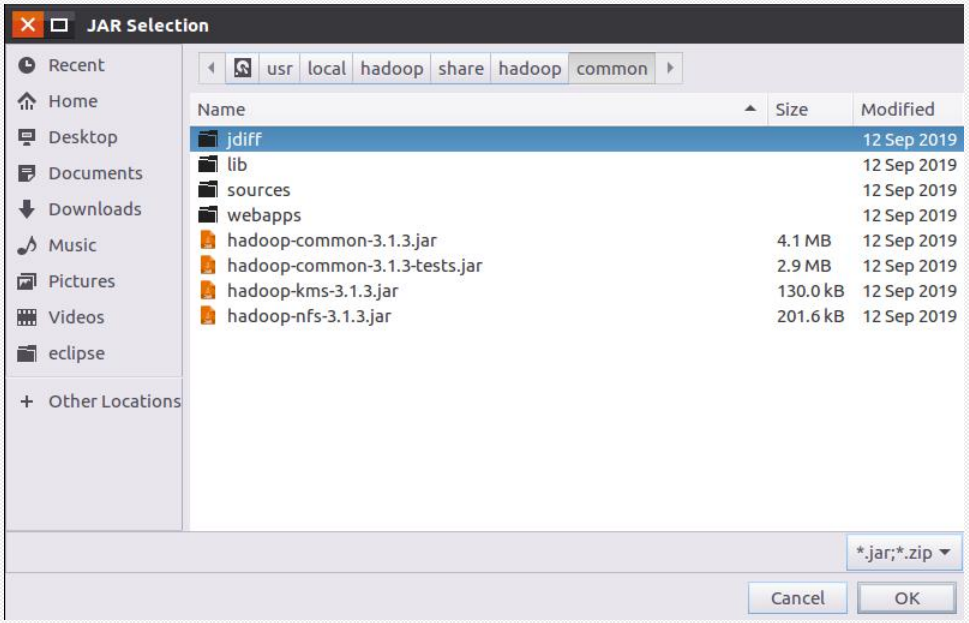
二、为项目添加需要用到的JAR包

进入下一步的设置以后，会弹出如下图所示界面。



3.7.3HDFS常用Java API及应用实例

需要在这个界面中加载该Java工程所需要用到的JAR包，这些JAR包中包含了可以访问HDFS的Java API。这些JAR包都位于Linux系统的Hadoop安装目录下，对于本教程而言，就是在“/usr/local/hadoop/share/hadoop”目录下。点击界面中的“Libraries”选项卡，然后，点击界面右侧的“Add External JARs…”按钮，会弹出如下图所示界面。



3.7.3HDFS常用Java API及应用实例

在该界面中，上面的一排目录按钮（即“usr”、“local”、“hadoop”、“share”、“hadoop”和“common”），当点击某个目录按钮时，就会在下面列出该目录的内容。

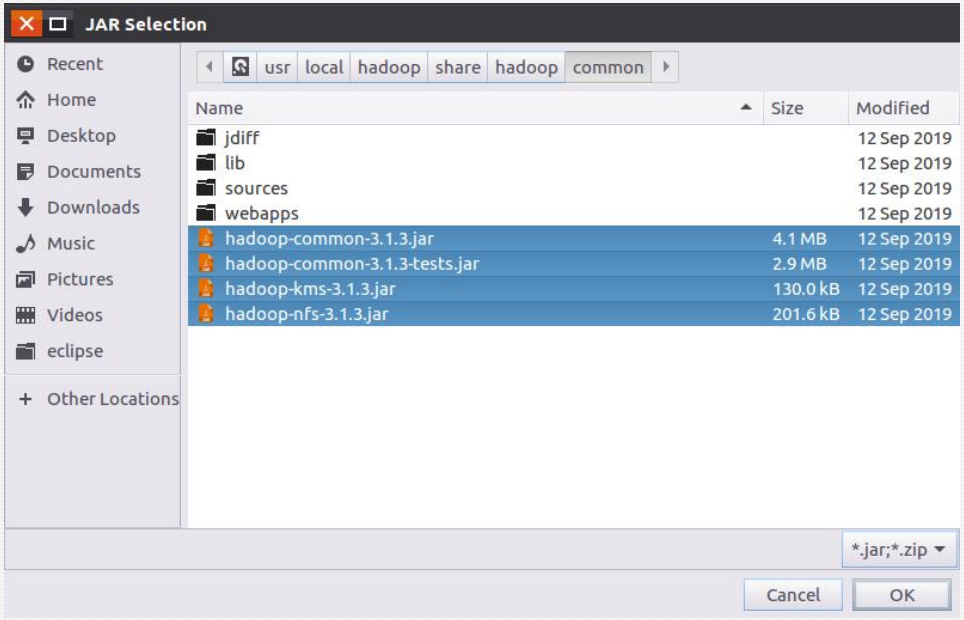
为了编写一个能够与HDFS交互的Java应用程序，一般需要向Java工程中添加以下JAR包：

- “/usr/local/hadoop/share/hadoop/common”目录下的所有JAR包，包括hadoop-common-3.1.3.jar、hadoop-common-3.1.3-tests.jar、haoop-nfs-3.1.3.jar和haoop-kms-3.1.3.jar，注意，不包括目录jdiff、lib、sources和webapps；
- “/usr/local/hadoop/share/hadoop/common/lib”目录下的所有JAR包；
- “/usr/local/hadoop/share/hadoop/hdfs”目录下的所有JAR包，注意，不包括目录jdiff、lib、sources和webapps；

“/usr/local/hadoop/share/hadoop/hdfs/lib”目录下的所有JAR包。

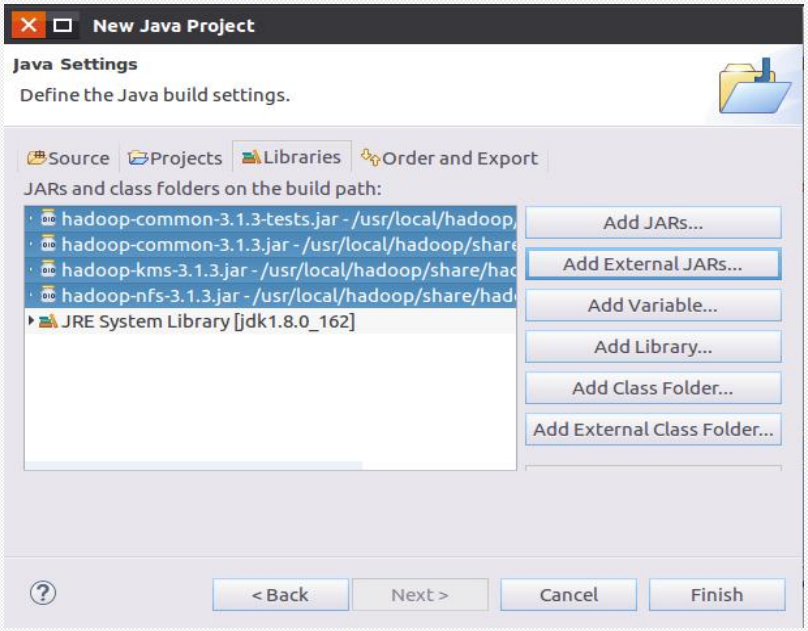
3.7.3HDFS常用Java API及应用实例

比如，如果要把“/usr/local/hadoop/share/hadoop/common”目录下的hadoop-common-3.1.3.jar、hadoop-common-3.1.3-tests.jar、haoop-nfs-3.1.3.jar和haoop-kms-3.1.3.jar添加到当前的Java工程中，可以在界面中点击目录按钮，进入到common目录，然后，界面会显示出common目录下的所有内容（如下图所示）。



3.7.3HDFS常用Java API及应用实例

请在界面中用鼠标点击选中hadoop-common-3.1.3.jar、hadoop-common-3.1.3-tests.jar、haoop-nfs-3.1.3.jar和haoop-kms-3.1.3.jar（不要选中目录jdiff、lib、sources和webapps），然后点击界面右下角的“确定”按钮，就可以把这两个JAR包增加到当前Java工程中，出现的界面如下图所示。



3.7.3HDFS常用Java API及应用实例

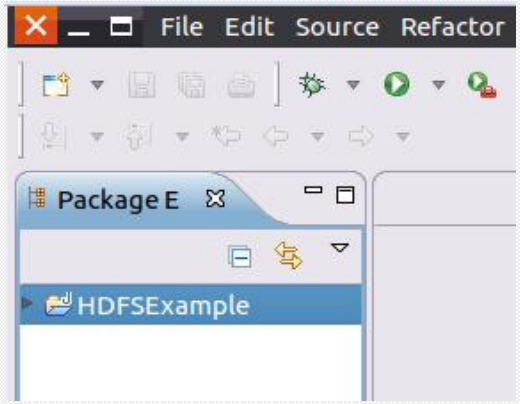
从这个界面中可以看出，hadoop-common-3.1.3.jar、hadoop-common-3.1.3-tests.jar、haoop-nfs-3.1.3.jar和haoop-kms-3.1.3.jar已经被添加到当前Java工程中。然后，按照类似的操作方法，可以再次点击“Add External JARs…”按钮，把剩余的其他JAR包都添加进来。需要注意的是，当需要选中某个目录下的所有JAR包时，可以使用“Ctrl+A”组合键进行全选操作。全部添加完毕以后，就可以点击界面右下角的“Finish”按钮，完成Java工程HDFSExample的创建。

3.7.3HDFS常用Java API及应用实例

三、编写Java应用程序

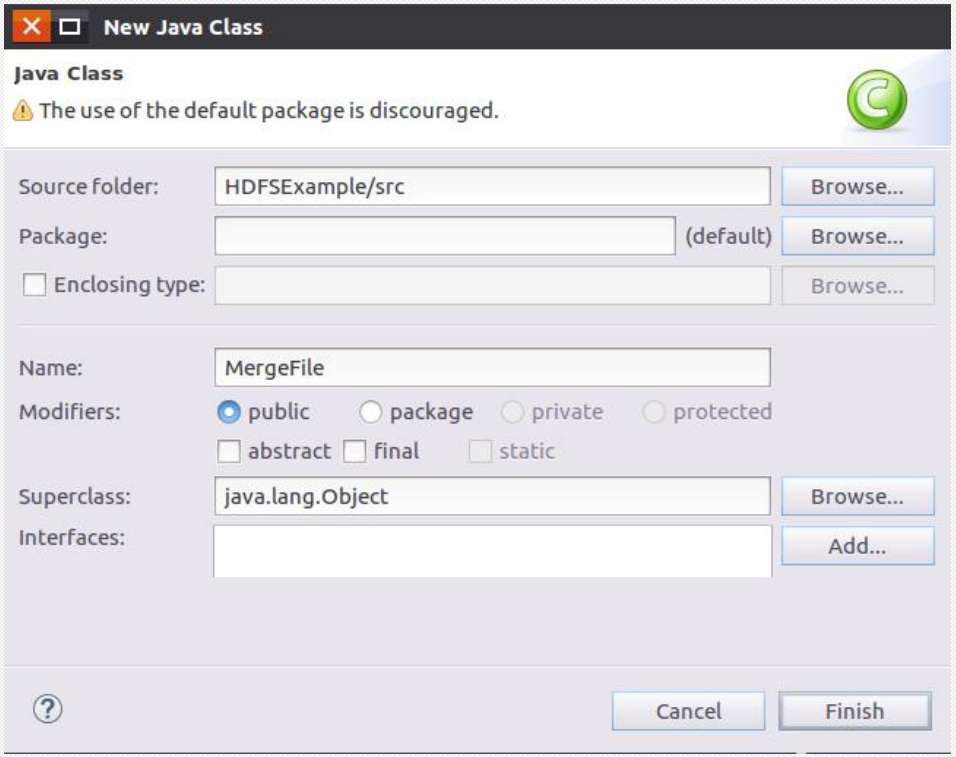
下面编写一个Java应用程序，用来检测HDFS中是否存在一个文件。

请在Eclipse工作界面左侧的“Package Explorer”面板中（如下图所示），找到刚才创建好的工程名称“HDFSExample”，然后在该工程名称上点击鼠标右键，在弹出的菜单中选择“New→Class”菜单。



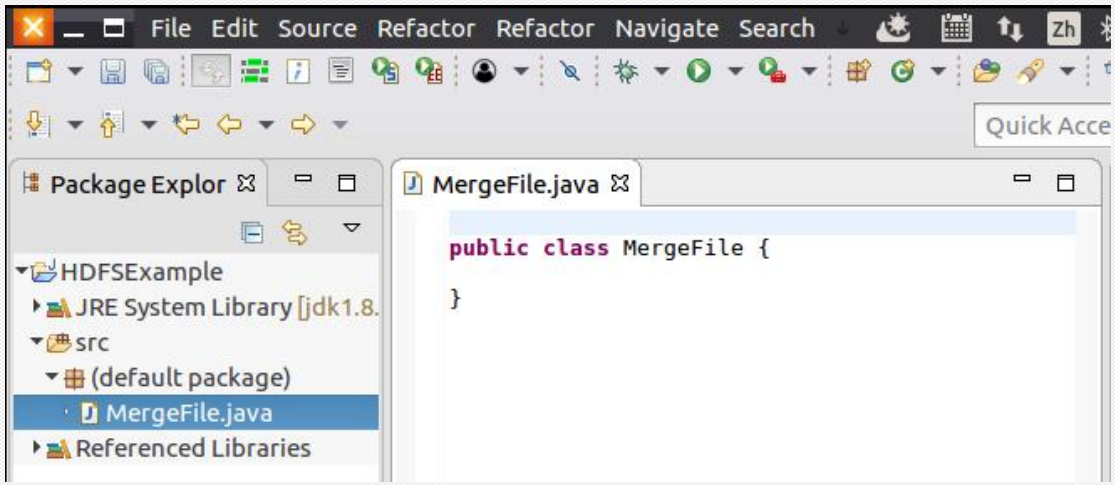
3.7.3HDFS常用Java API及应用实例

选择“New→Class”菜单以后会出现如下图所示界面。



3.7.3HDFS常用Java API及应用实例

在该界面中，只需要在“Name”后面输入新建的Java类文件的名称，这里采用名称“MergeFile”，其他都可以采用默认设置，然后，点击界面右下角“Finish”按钮，出现如下图所示界面。



3.7.3HDFS常用Java API及应用实例

可以看出，Eclipse自动创建了一个名为“MergeFile.java”的源代码文件，请在该文件中输入以下代码：

```
import java.io.IOException;
import java.io.PrintStream;
import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.*;

/**
 * 过滤掉文件名满足特定条件的文件
 */
class MyPathFilter implements PathFilter {
    String reg = null;
    MyPathFilter(String reg) {
        this.reg = reg;
    }
    public boolean accept(Path path) {
                                                    if (!(path.toString().matches(reg)))
                                                                                    return true;
                                                    return false;
                                    }
}
/***
 * 利用FSDataOutputStream和FSDataInputStream合并HDFS中的文件
 */
public class MergeFile {
                                    Path inputPath = null; //待合并的文件所在的目录的路径
                                    Path outputPath = null; //输出文件的路径
                                    public MergeFile(String input, String output) {
                                                    this.inputPath = new Path(input);
                                                    this.outputPath = new Path(output);
                                    }
                                    public void doMerge() throws IOException {
                                                    Configuration conf = new Configuration();
                                                    conf.set("fs.defaultFS","hdfs://localhost:9000");
        conf.set("fs.hdfs.impl","org.apache.hadoop.hdfs.DistributedFileSystem");
                                                    FileSystem fsSource = FileSystem.get(URI.create(inputPath.toString()), conf);
                                                    FileSystem fsDst = FileSystem.get(URI.create(outputPath.toString()), conf);
                                                                                    //下面过滤掉输入目录中后缀为.abc的文件
                                                    FileStatus[] sourceStatus = fsSource.listStatus(inputPath,
                                                                                    new MyPathFilter(".*\\.abc"));
                                                    FSDataOutputStream fsdos = fsDst.create(outputPath);
                                                    PrintStream ps = new PrintStream(System.out);
                                                    //下面分别读取过滤之后的每个文件的内容，并输出到同一个文件中
                                                    for (FileStatus sta : sourceStatus) {
                                                                                    //下面打印后缀不为.abc的文件的路径、文件大小
                                                                                    System.out.print("路径：" + sta.getPath() + "    文件大小：" + sta.getLen()
                                                                                                                    + "   权限：" + sta.getPermission() + "   内容：");
                                                                                    FSDataInputStream fsdis = fsSource.open(sta.getPath());
                                                                                    byte[] data = new byte[1024];
                                                                                    int read = -1;

                                                                                    while ((read = fsdis.read(data)) > 0) {
                                                                                                                    ps.write(data, 0, read);
                                                                                                                    fsdos.write(data, 0, read);
                                                                                    }
                                                                                    fsdis.close();
                                                    }
                                                    ps.close();
                                                    fsdos.close();
                                    }
                                    public static void main(String[] args) throws IOException {
                                                    MergeFile merge = new MergeFile(
                                                                                                                    "hdfs://localhost:9000/user/hadoop/",
                                                                                                                    "hdfs://localhost:9000/user/hadoop/merge.txt");
                                                    merge.doMerge();
                                    }
}
```

3.7.3HDFS常用Java API及应用实例

四、编译运行程序

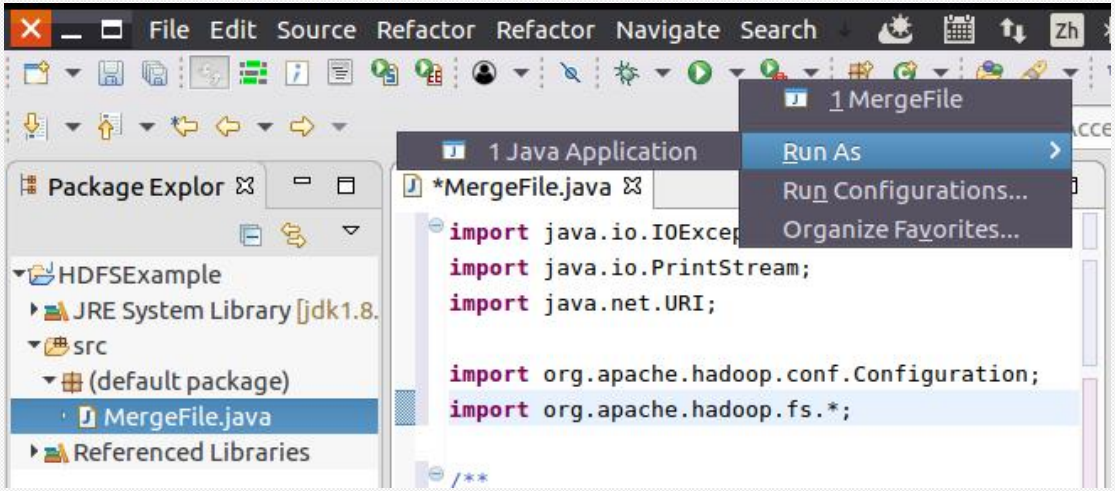
在开始编译运行程序之前，请一定确保Hadoop已经启动运行

然后，要确保HDFS的“/user/hadoop”目录下已经存在file1.txt、file2.txt、file3.txt、file4.abc和file5.abc，每个文件里面有内容。这里，假设文件内容如下表所示。

文件名称	文件内容
file1.txt	this is file1.txt
file2.txt	this is file2.txt
file3.txt	this is file3.txt
file4.abc	this is file4.abc
file5.abc	this is file5.abc

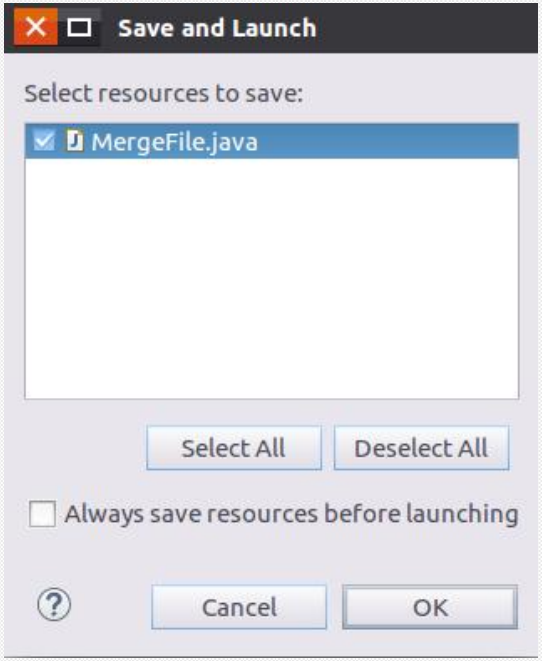
3.7.3HDFS常用Java API及应用实例

现在就可以编译运行上面编写的代码。可以直接点击Eclipse工作界面上部的运行程序的快捷按钮，当把鼠标移动到该按钮上时，在弹出的菜单中选择“Run As”，继续在弹出来的菜单中选择“Java Application”，如下图所示。



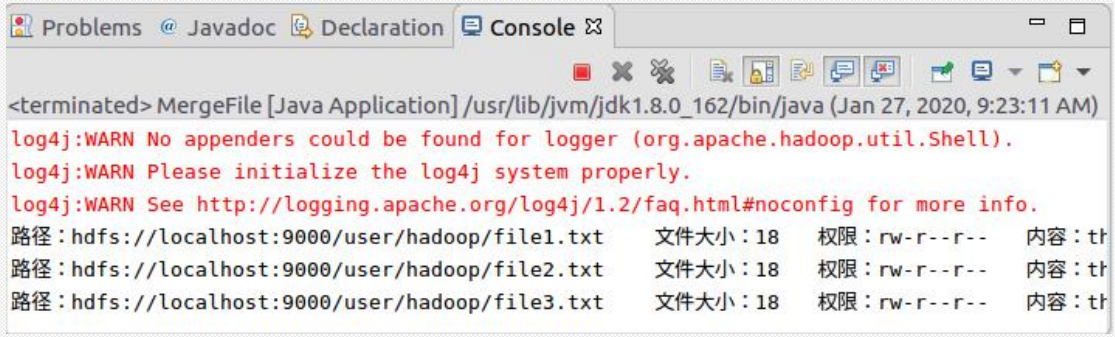
3.7.3HDFS常用Java API及应用实例

然后，会弹出如下图所示界面。



3.7.3HDFS常用Java API及应用实例

在该界面中，点击界面右下角的“OK”按钮，开始运行程序。程序运行结束后，会在底部的“Console”面板中显示运行结果信息（如下图所示）。同时，“Console”面板中还会显示一些类似“log4j:WARN…”的警告信息，可以不用理会。



3.7.3HDFS常用Java API及应用实例

如果程序运行成功，这时，可以到HDFS中查看生成的merge.txt文件，比如，可以在Linux终端中执行如下命令：

```
$ cd /usr/local/hadoop
$ ./bin/hdfs dfs -ls /user/hadoop
$ ./bin/hdfs dfs -cat /user/hadoop/merge.txt
```

可以看到如下结果：

```
this is file1.txt
this is file2.txt
this is file3.txt
```

3.7.3HDFS常用Java API及应用实例

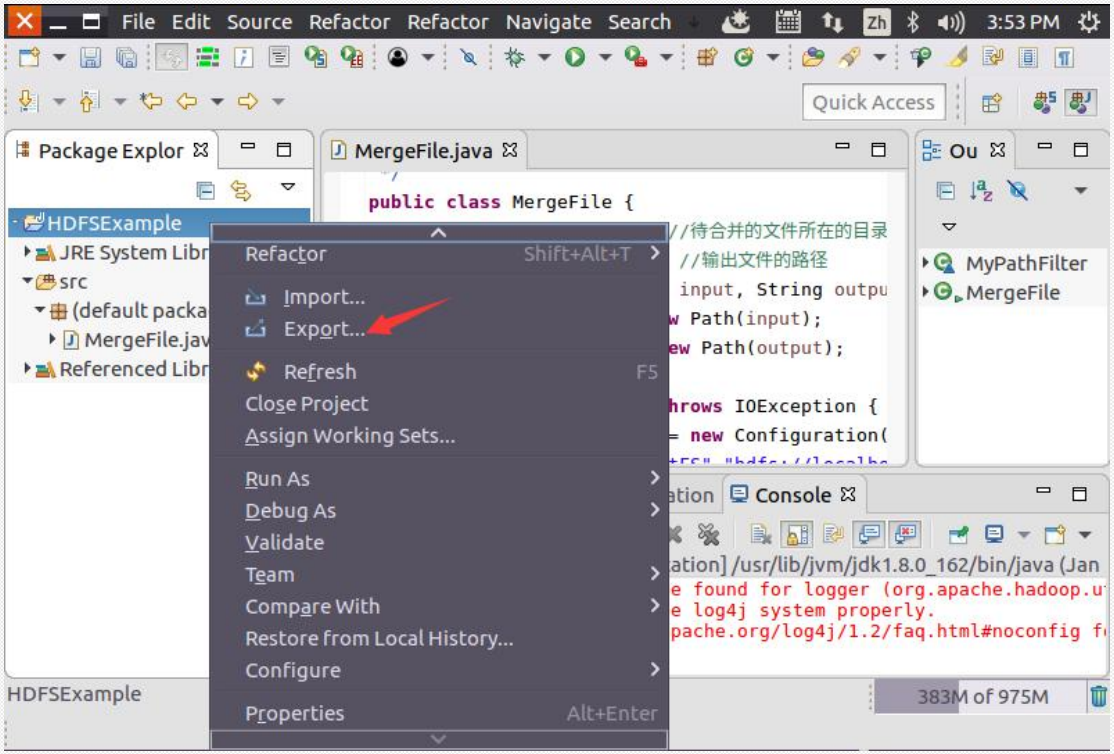
四、应用程序的部署

下面介绍如何把Java应用程序生成JAR包，部署到Hadoop平台上运行。首先，在Hadoop安装目录下新建一个名称为myapp的目录，用来存放我们自己编写的Hadoop应用程序，可以在Linux的终端中执行如下命令：

```
$ cd /usr/local/hadoop
$ mkdir myapp
```

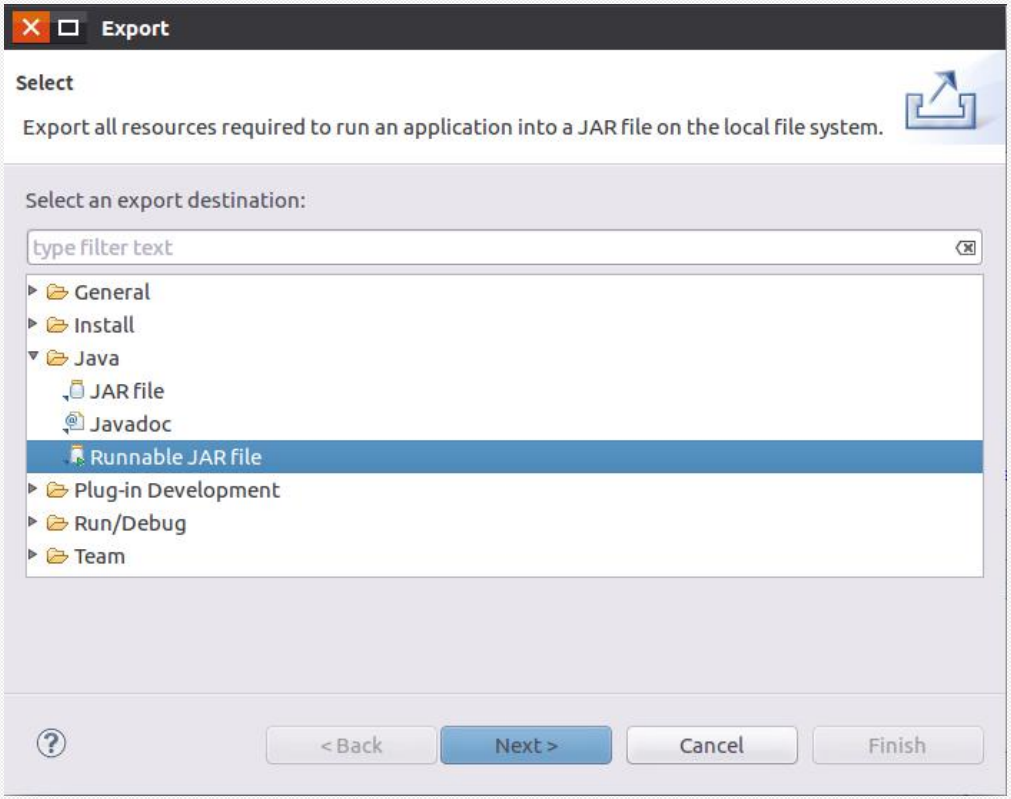
3.7.3HDFS常用Java API及应用实例

然后，请在Eclipse工作界面左侧的“Package Explorer”面板中，在工程名称“HDFSExample”上点击鼠标右键，在弹出的菜单中选择“Export”，如下图所示。



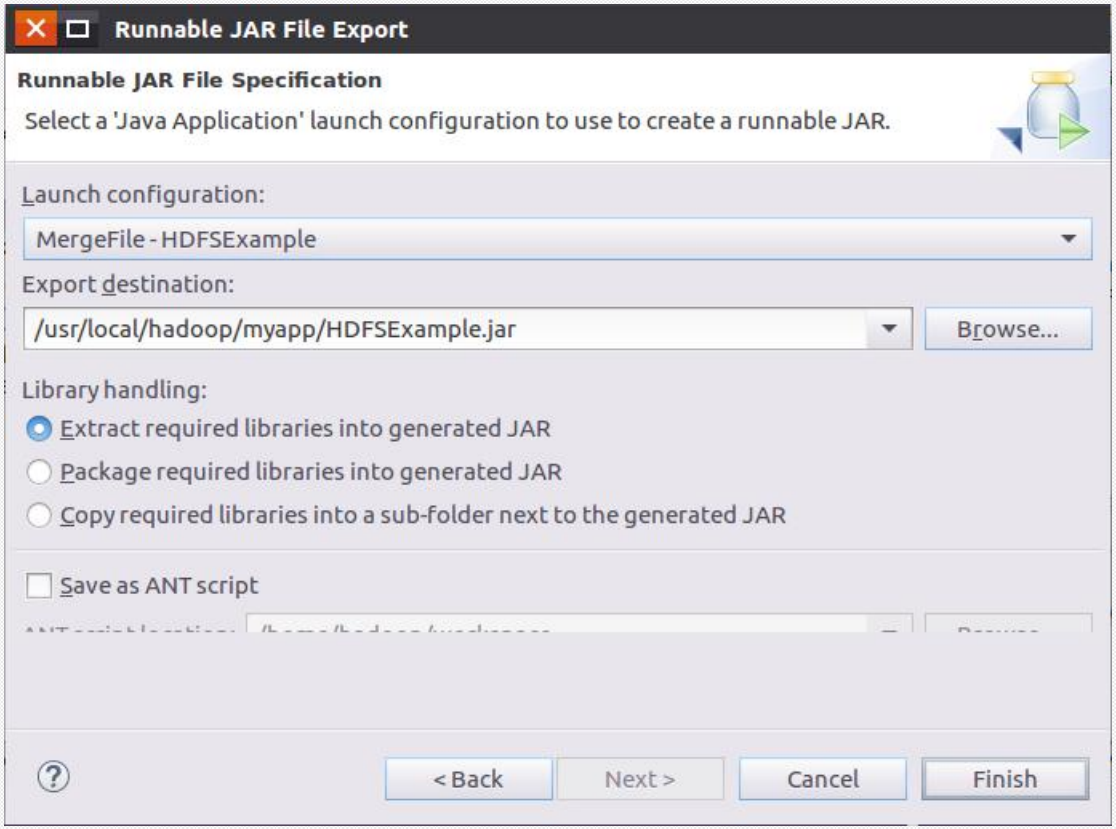
3.7.3HDFS常用Java API及应用实例

然后，会弹出如下图所示界面。



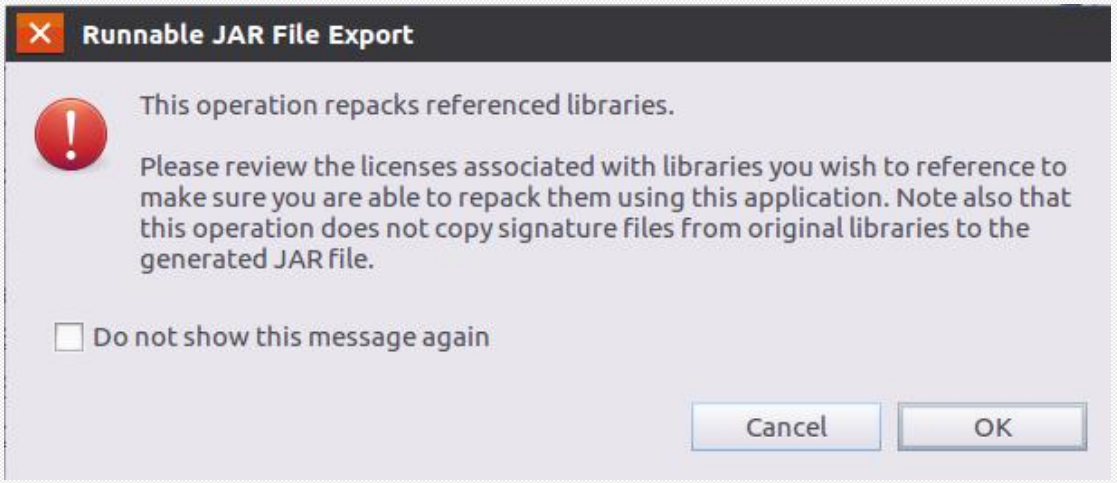
3.7.3HDFS常用Java API及应用实例

在该界面中，选择“Runnable JAR file”，然后，点击“Next>”按钮，弹出如下图所示界面。



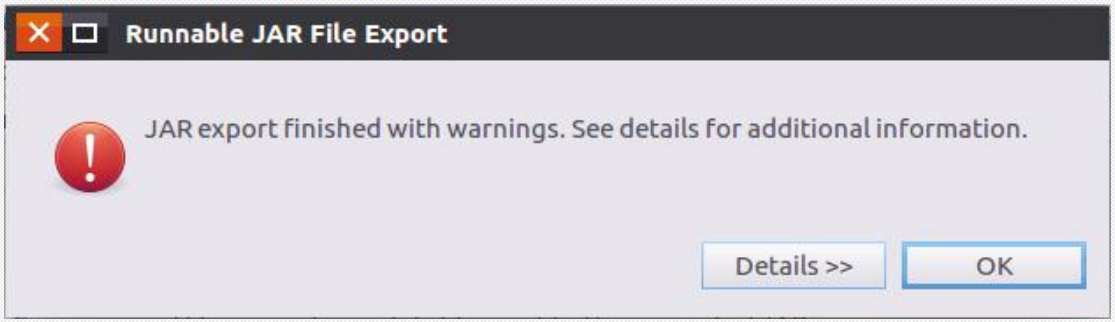
3.7.3HDFS常用Java API及应用实例

在该界面中，“Launch configuration”用于设置生成的JAR包被部署启动时运行的主类，需要在下拉列表中选择刚才配置的类“MergeFile-HDFSExample”。在“Export destination”中需要设置JAR包要输出保存到哪个目录，比如，这里设置为“/usr/local/hadoop/myapp/HDFSExample.jar”。在“Library handling”下面选择“Extract required libraries into generated JAR”。然后，点击“Finish”按钮，会出现如下图所示界面。



3.7.3HDFS常用Java API及应用实例

可以忽略该界面的信息，直接点击界面右下角的“OK”按钮，启动打包过程。打包过程结束后，会出现一个警告信息界面，如下图所示。



3.7.3HDFS常用Java API及应用实例

可以忽略该界面的信息，直接点击界面右下角的“OK”按钮。至此，已经顺利把HDFSExample工程打包生成了HDFSExample.jar。可以到Linux系统中查看一下生成的HDFSExample.jar文件，可以在Linux的终端中执行如下命令：

```
$ cd /usr/local/hadoop/myapp
$ ls
```

可以看到，“/usr/local/hadoop/myapp”目录下已经存在一个HDFSExample.jar文件。

3.7.3HDFS常用Java API及应用实例

由于之前已经运行过一次程序，已经生成了merge.txt，因此，需要首先执行如下命令删除该文件：

```
$ cd /usr/local/hadoop
$ ./bin/hdfs dfs -rm /user/hadoop/merge.txt
```

现在，就可以在Linux系统中，使用hadoop jar命令运行程序，命令如下：

```
$ cd /usr/local/hadoop
$ ./bin/hadoop jar ./myapp/HDFSExample.jar
```

3.7.3HDFS常用Java API及应用实例

上面程序执行结束以后，可以到HDFS中查看生成的merge.txt文件，比如，可以在Linux终端中执行如下命令：

```
$ cd /usr/local/hadoop
$ ./bin/hdfs dfs -ls /user/hadoop
$ ./bin/hdfs dfs -cat /user/hadoop/merge.txt
```

可以看到如下结果：

```
this is file1.txt
this is file2.txt
this is file3.txt
```

本章小结

- 分布式文件系统是大数据时代解决大规模数据存储问题的有效解决方案，HDFS开源实现了GFS，可以利用由廉价硬件构成的计算机集群实现海量数据的分布式存储
- HDFS具有兼容廉价的硬件设备、流数据读写、大数据集、简单的文件模型、强大的跨平台兼容性等特点。但是，也要注意到，HDFS也有自身的局限性，比如不适合低延迟数据访问、无法高效存储大量小文件和不支持多用户写入及任意修改文件等
- 块是HDFS核心的概念，一个大的文件会被拆分成很多个块。HDFS采用抽象的块概念，具有支持大规模文件存储、简化系统设计、适合数据备份等优点
- HDFS采用了主从（Master/Slave）结构模型，一个HDFS集群包括一个名称节点和若干个数据节点。名称节点负责管理分布式文件系统的命名空间；数据节点是分布式文件系统HDFS的工作节点，负责数据的存储和读取
- HDFS采用了冗余数据存储，增强了数据可靠性，加快了数据传输速度。HDFS还采用了相应的数据存放、数据读取和数据复制策略，来提升系统整体读写响应性能。HDFS把硬件出错看作一种常态，设计了错误恢复机制
- 本章最后介绍了HDFS的数据读写过程以及HDFS编程实践方面的相关知识


Thank You!