

《大数据基础》

第7章 MapReduce

洪韬

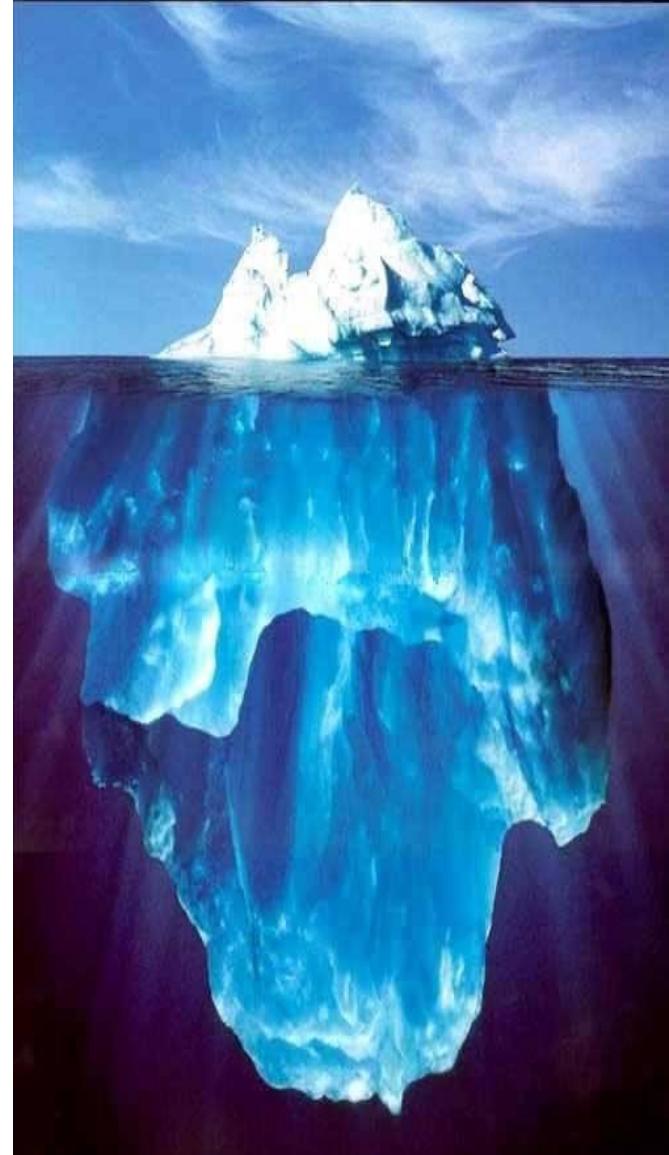
E-mail: 2990893@qq.com





提纲

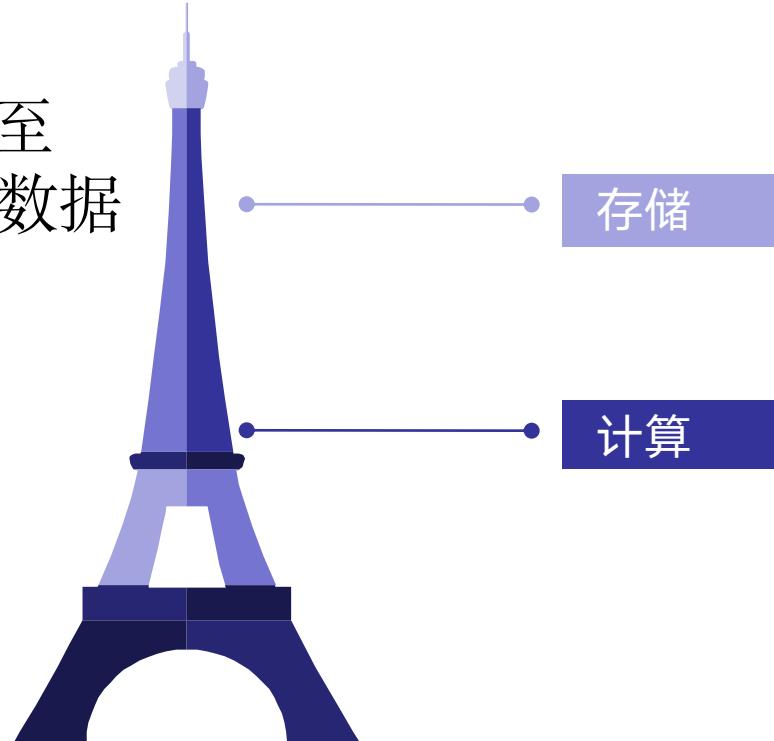
- 7.1 概述
- 7.2 MapReduce体系结构
- 7.3 MapReduce工作流程
- 7.4 实例分析：WordCount
- 7.5 MapReduce的具体应用
- 7.6 MapReduce编程实践





7.1 概述

TB级别甚至
PB级别的数据



HDFS



7.1.1 分布式并行编程

- “摩尔定律”， CPU性能大约每隔18个月翻一番
- 从2005年开始摩尔定律逐渐失效， 需要处理的数据量快速增加， 人们开始借助于分布式并行编程来提高程序性能
- 分布式程序运行在大规模计算机集群上， 可以并行执行大规模数据处理任务， 从而获得海量的计算能力
- 谷歌公司最先提出了分布式并行编程模型**MapReduce**， Hadoop MapReduce是它的开源实现， 后者比前者使用门槛低很多



7.1.1 分布式并行编程

问题：在MapReduce出现之前，已经有像MPI这样非常成熟的并行计算框架了，那么为什么Google还需要MapReduce？MapReduce相较于传统的并行计算框架有什么优势？

	传统并行计算框架	MapReduce
集群架构/容错性	共享式(共享内存/共享存储)，容错性差	非共享式，容错性好
硬件/价格/扩展性	刀片服务器、高速网、SAN，价格贵，扩展性差	普通PC机，便宜，扩展性好
编程/学习难度	what-how，难	what，简单
适用场景	实时、细粒度计算、计算密集型	批处理、非实时、数据密集型

分布式存储问题、工作调度问题、负载均衡问题、容错处理以及网络通信等问题



7.1.2 MapReduce模型简介

- MapReduce将复杂的、运行于大规模集群上的并行计算过程高度地抽象到了两个函数：**Map**和**Reduce**





7.1.2 MapReduce模型简介

函数	输入	输出	说明
Map	$\langle k_1, v_1 \rangle$ 如： \langle 行号,"a b c" \rangle	List($\langle k_2, v_2 \rangle$) 如： \langle "a",1 \rangle \langle "b",1 \rangle \langle "c",1 \rangle	1. 将小数据集进一步解析成一批 \langle key,value \rangle 对， 输入Map函数中进 行处理 2. 每一个输入的 $\langle k_1, v_1 \rangle$ 会输出一批 $\langle k_2, v_2 \rangle$ 。 $\langle k_2, v_2 \rangle$ 是计算的中间结果
Reduce	$\langle k_2, \text{List}(v_2) \rangle$ 如： \langle "a", \langle 1,1,1 \rangle \rangle	$\langle k_3, v_3 \rangle$ \langle "a",3 \rangle	输入的中间结果 $\langle k_2, \text{List}(v_2) \rangle$ 中的 $\text{List}(v_2)$ 表示是一批属于同一个 k_2 的 value



7.1.2 MapReduce模型简介

Reduce函数

输入

`<key, value-list>`

也就是说一个键以及一堆值的列表

`value-list`是值的列表，比如前面`key`是`a`

后面一堆值列表`<1, 1, 1>`就是一堆的值，很

多的值构成一个列表`list`

这叫`value-list` 是一个值的列表

输出

对`value-list`进行一个`reduce`的结果

对这些值进行一个汇总求和

求完以后就生成一个`<key, value>`

输入是一个`<key, value-list>`

输出是一个`<key, value>`



7.1.2 MapReduce模型简介

MapReduce的策略





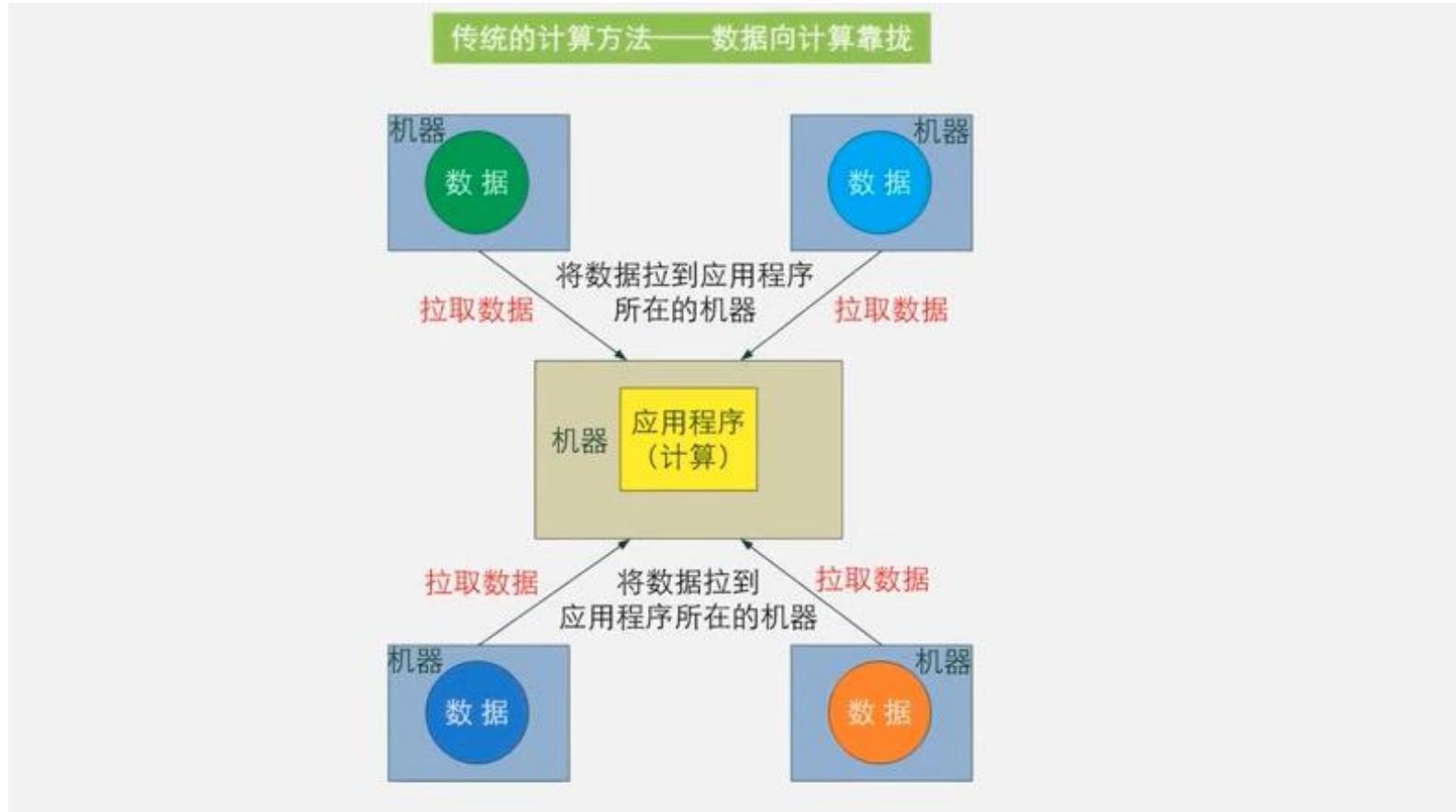
7.1.2 MapReduce模型简介

MapReduce的理念

- 计算向数据靠拢而不是数据向计算靠拢
- 要完成一次数据分析时，选择一个计算节点，把运行数据分析的程序放到计算节点上运行
- 然后把它所涉及的数据，全部从各个不同节点上面拉过来，传输到计算发生的地方

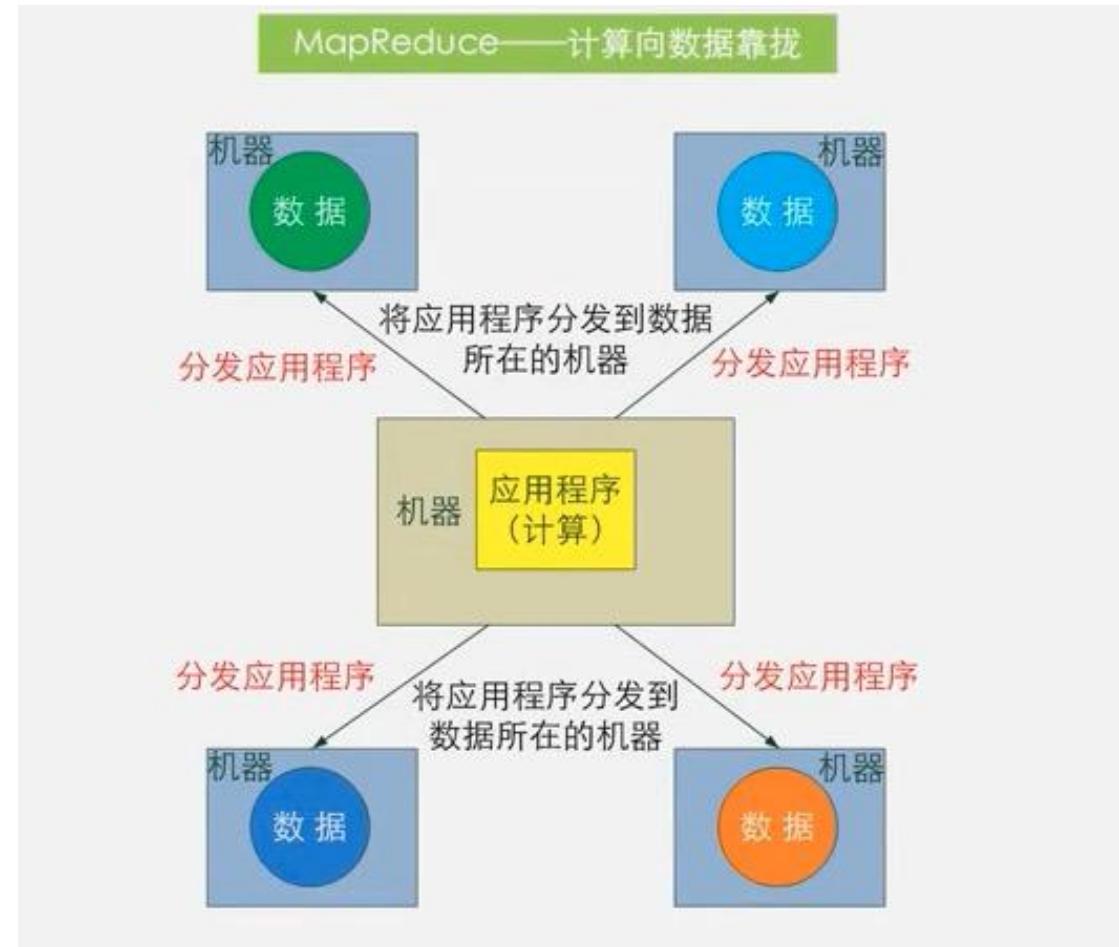


7.1.2 MapReduce模型简介





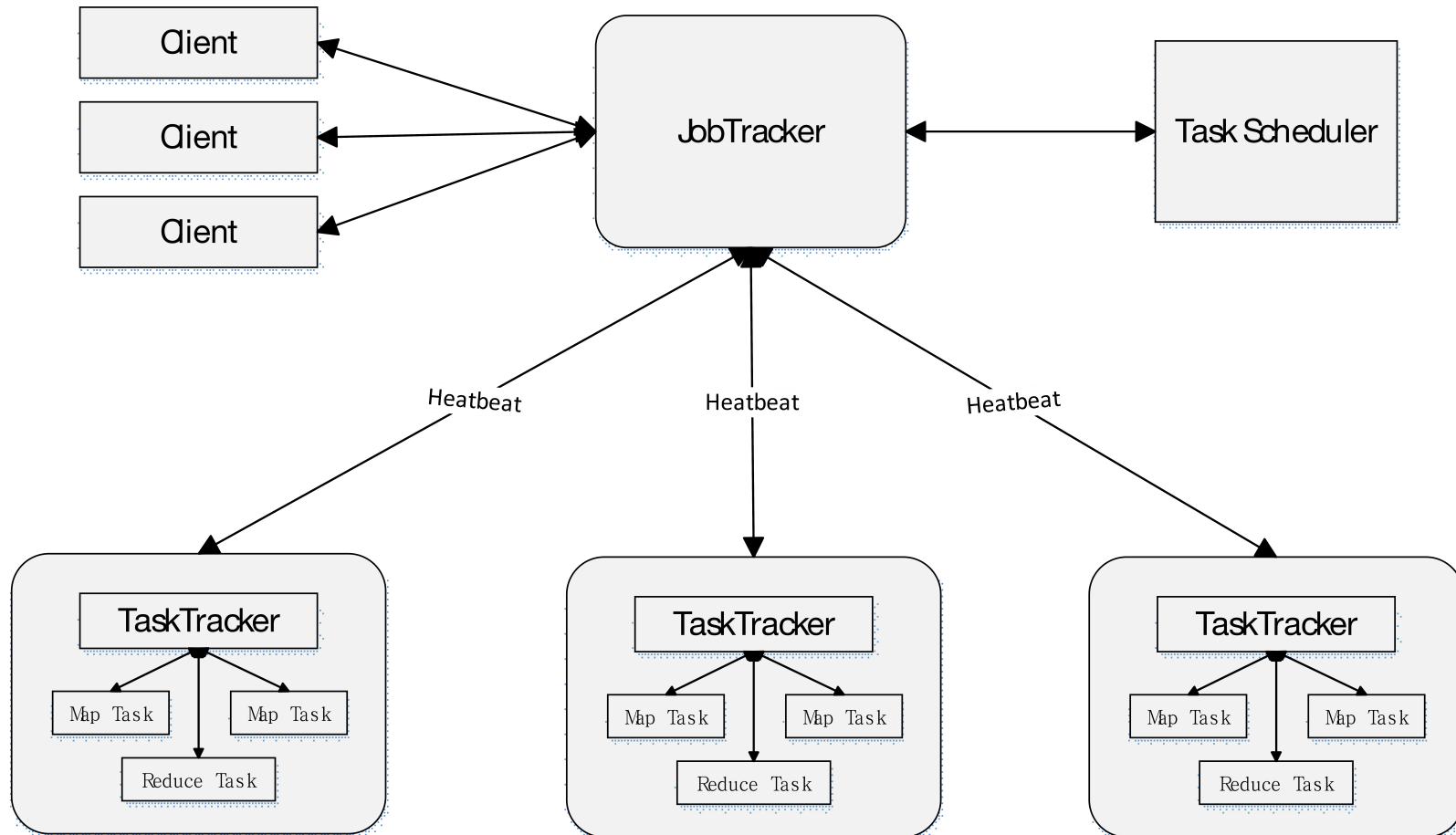
7.1.2 MapReduce模型简介





7.2 MapReduce的体系结构

MapReduce体系结构主要由四个部分组成，分别是：Client、JobTracker、TaskTracker以及Task





7.2 MapReduce的体系结构





7.2 MapReduce的体系结构

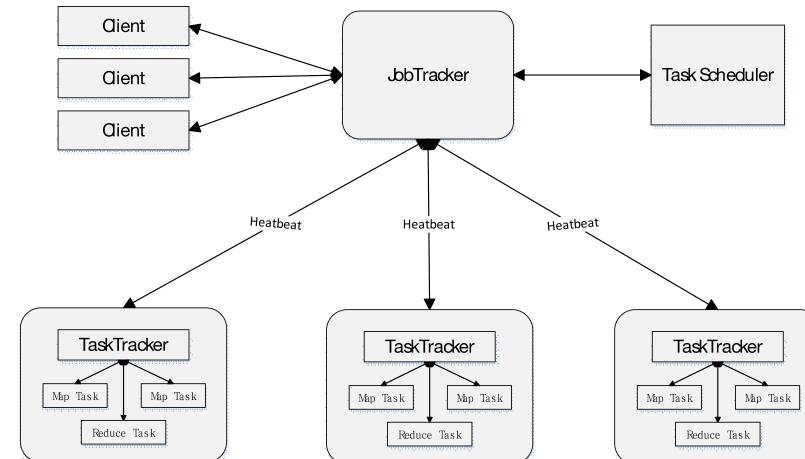
MapReduce主要有以下4个部分组成:

1) Client

- 用户编写的MapReduce程序通过Client提交到JobTracker端
- 用户可通过Client提供的一些接口查看作业运行状态

2) JobTracker

- JobTracker负责资源监控和作业调度
- JobTracker 监控所有TaskTracker与Job的健康状况，一旦发现失败，就将相应的任务转移到其他节点
- JobTracker 会跟踪任务的执行进度、资源使用量等信息，并将这些信息告诉任务调度器（TaskScheduler），而调度器会在资源出现空闲时，选择合适的任务去使用这些资源





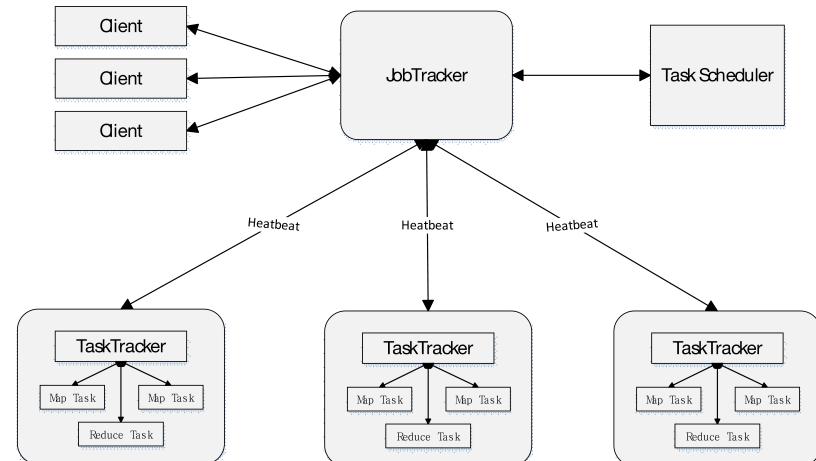
7.2 MapReduce的体系结构

3) TaskTracker

- TaskTracker 会周期性地通过“心跳”将本节点上资源的使用情况和任务的运行进度汇报给JobTracker，同时接收JobTracker 发送过来的命令并执行相应的操作（如启动新任务、杀死任务等）
- TaskTracker 使用“slot”等量划分本节点上的资源量（CPU、内存等）。一个Task 获取到一个slot 后才有机会运行，而Hadoop调度器的作用就是将各个TaskTracker上的空闲slot分配给Task使用。slot 分为Map slot 和Reduce slot 两种，分别供MapTask 和Reduce Task 使用

4) Task

Task 分为Map Task 和Reduce Task 两种，均由TaskTracker 启动



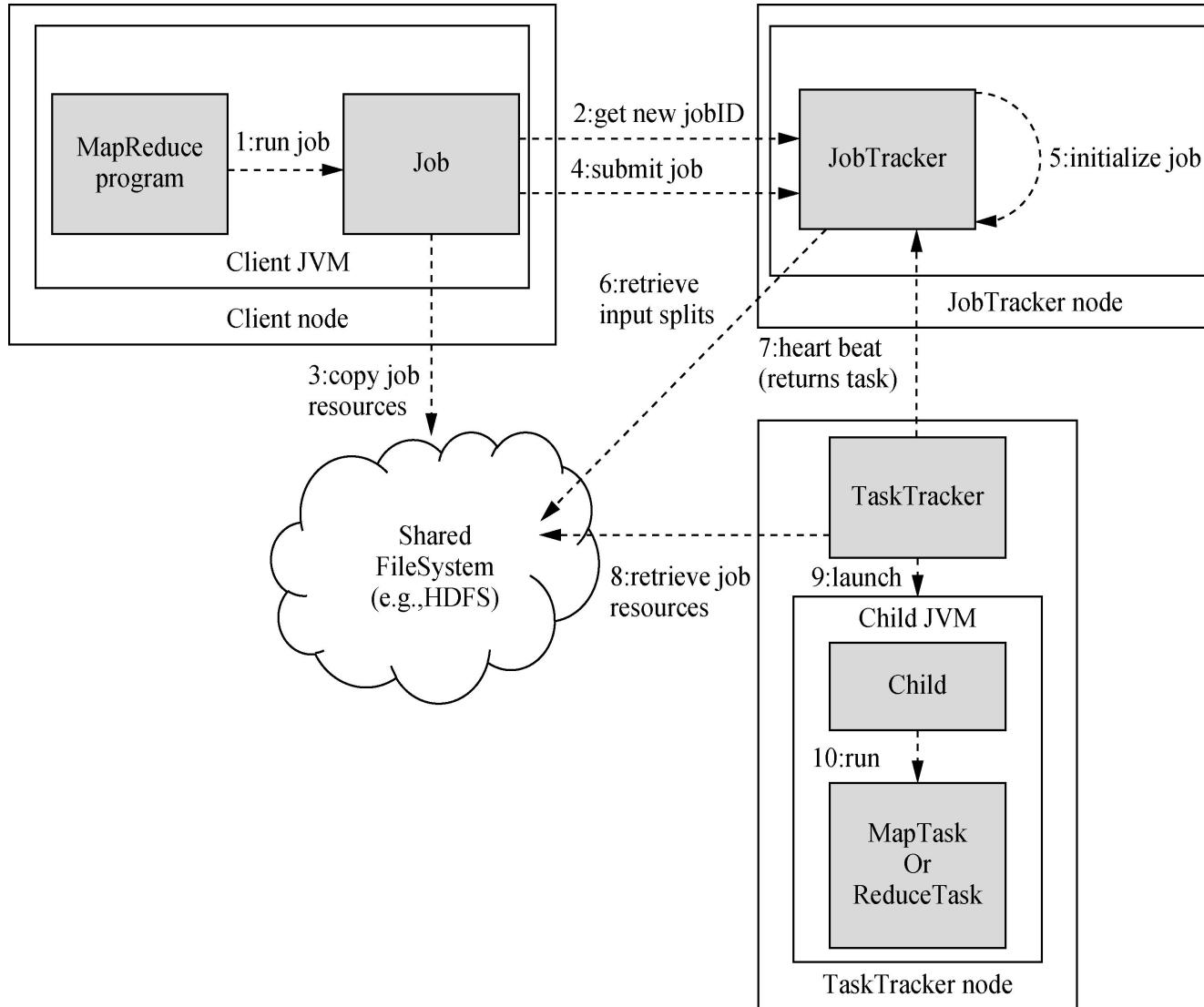


7.3 MapReduce工作流程

- 7.3.1 工作流程概述
- 7.3.2 MapReduce各个执行阶段
- 7.3.3 Shuffle过程详解

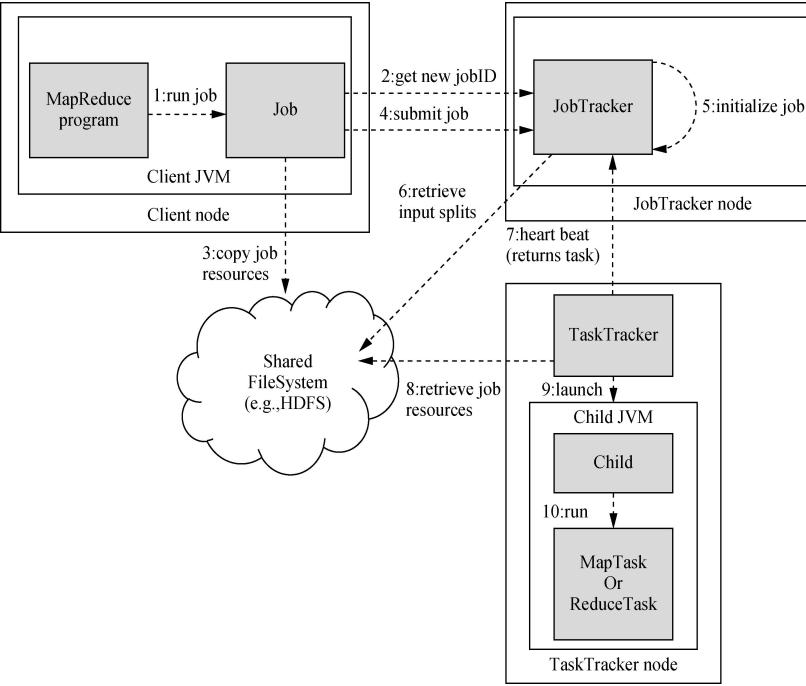


7.3.1 MapReduce工作流程





7.3.1 MapReduce工作流程



JobTracker称为Master，TaskTracker称为Slave。

用户提交的需要计算的作业称为Job（作业），每一个Job会被划分成若干个Tasks（任务）。

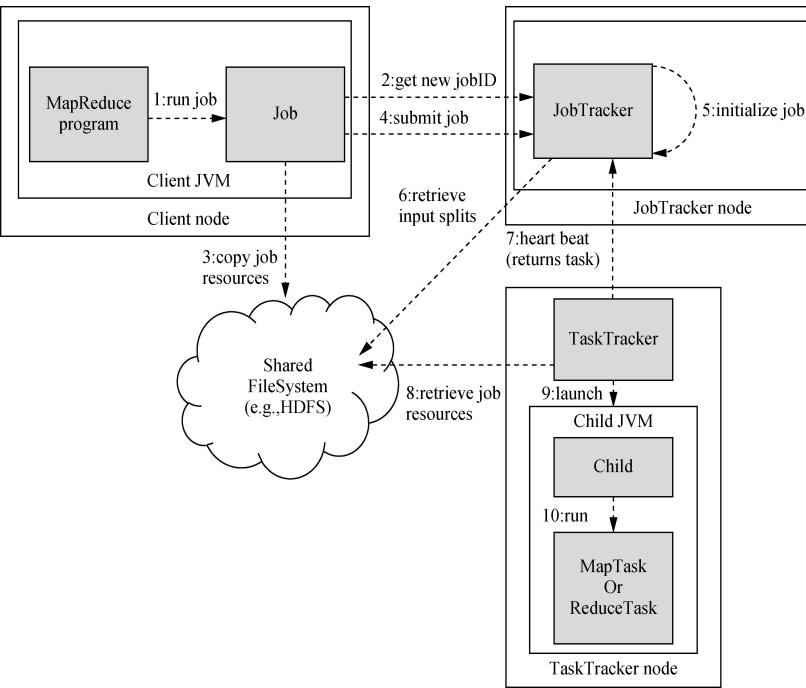
JobTracker负责Job和Tasks的调度，而TaskTracker负责执行Tasks。

MapReduce架构由4个独立的节点（Node）组成，分别为Client、JobTracker、TaskTracker和HDFS，分别介绍如下：

- (1) Client: 用来提交MapReduce作业;
- (2) JobTracker: 用来初始化作业、分配作业并与TaskTracker通信并协调整个作业;
- (3) TaskTracker: 将分配过来的数据片段执行MapReduce任务，并保持与JobTracker通信;
- (4) HDFS: 用来在其他节点间共享作业文件。



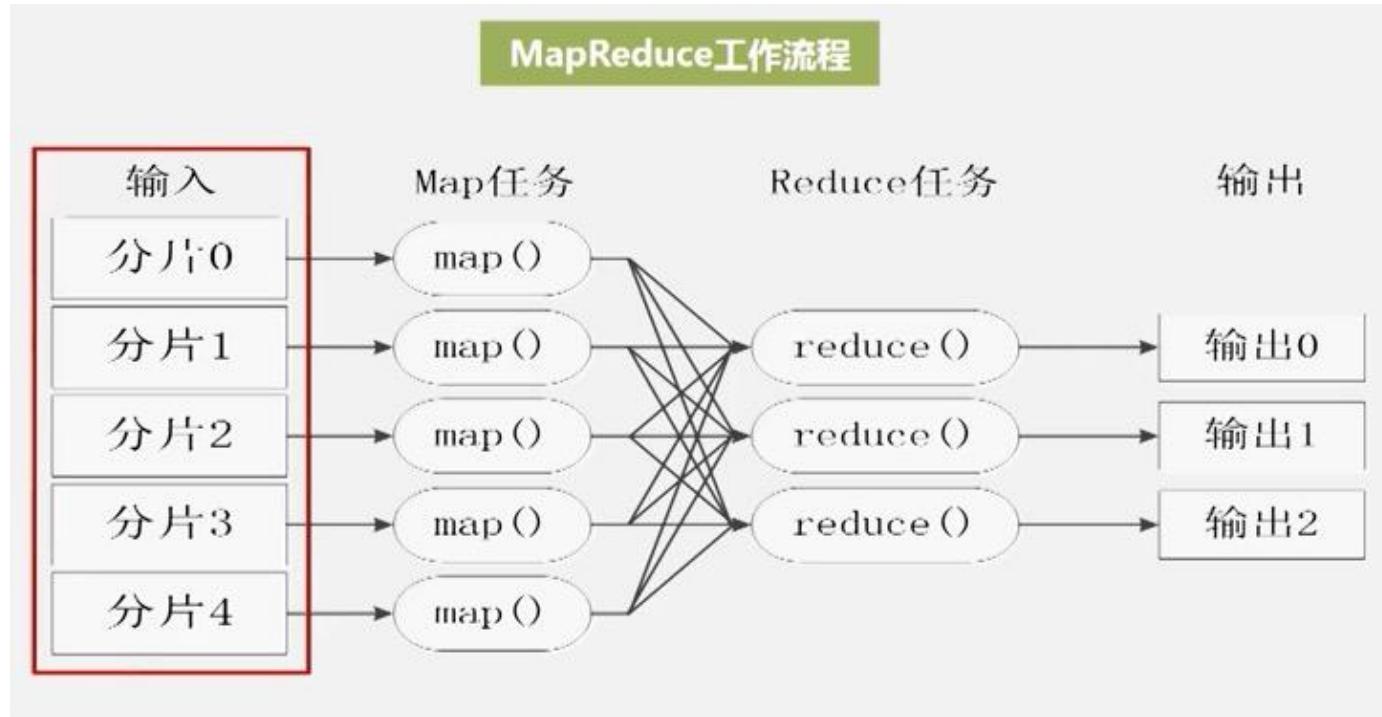
7.3.1 MapReduce工作流程



- (1) MapReduce在客户端启动一个作业;
- (2) Client向JobTracker请求一个JobID;
- (3) Client将需要执行的作业资源复制到HDFS上;
- (4) Client将作业提交给JobTracker;
- (5) JobTracker在本地初始化作业;
- (6) JobTracker从HDFS作业资源中获取作业输入的分割信息，根据这些信息将作业分割成多个任务;
- (7) JobTracker把多个任务分配给在与JobTracker心跳通信中请求任务的TaskTracker;
- (8) TaskTracker接收到新的任务之后会首先从HDFS上获取作业资源，包括作业配置信息和本作业分片的输入;
- (9) TaskTracker在本地登录子JVM;
- (10) TaskTracker启动一个JVM并执行任务，并将结果写回HDFS。



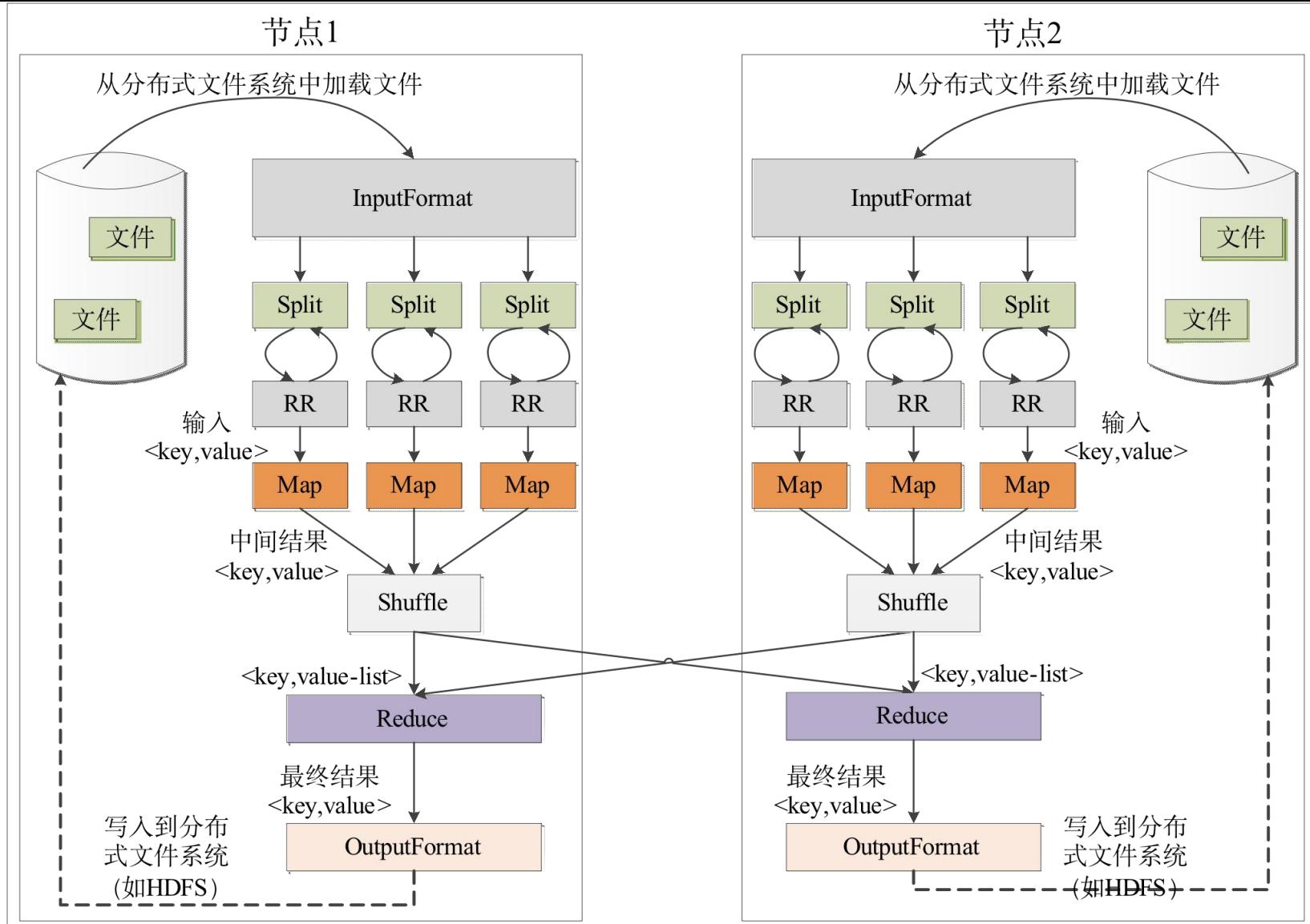
7.3.1 工作流程概述



- 不同的**Map**任务之间不会进行通信
- 不同的**Reduce**任务之间也不会发生任何信息交换
- 用户不能显式地从一台机器向另一台机器发送消息
- 所有的数据交换都是通过**MapReduce**框架自身去实现的



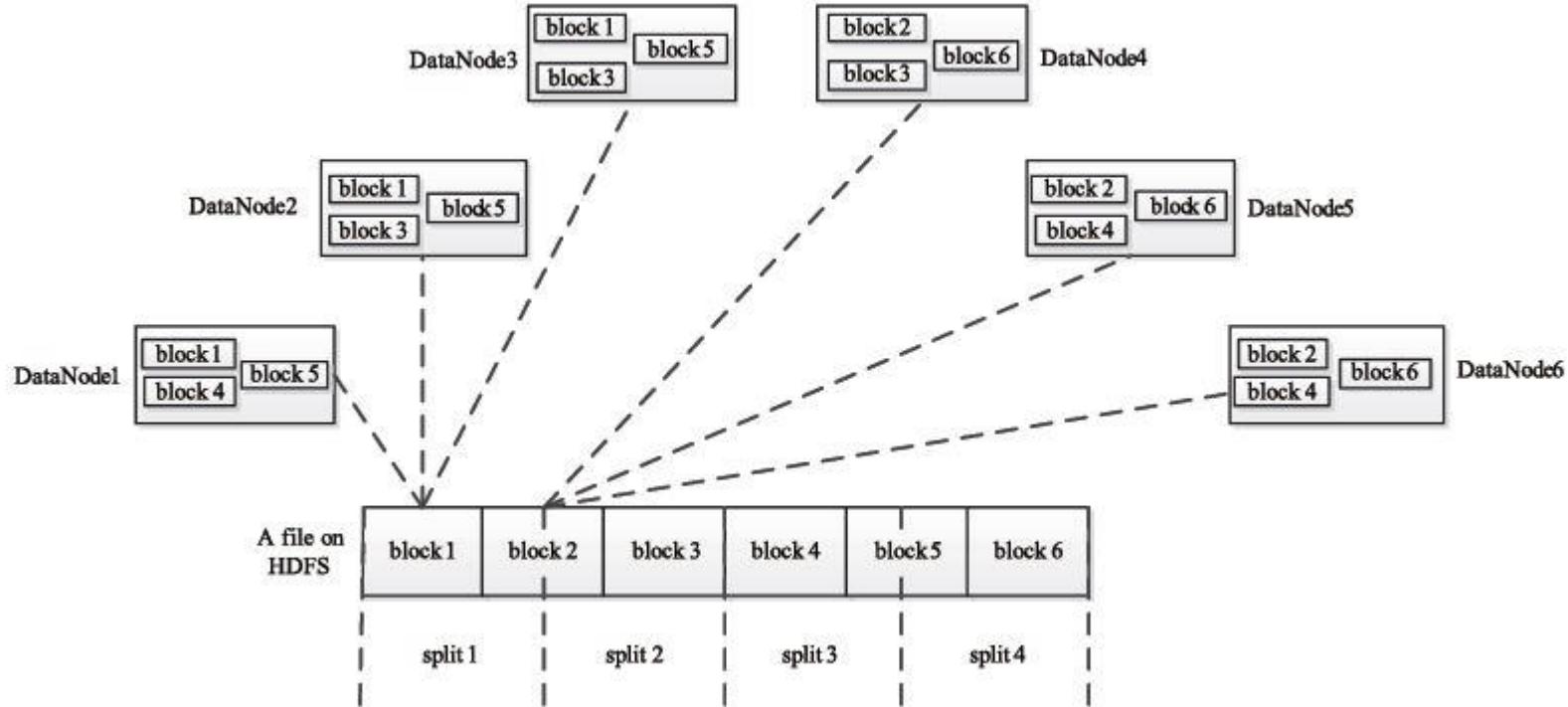
7.3.2 MapReduce各个执行阶段





7.3.2 MapReduce各个执行阶段

关于Split (分片)



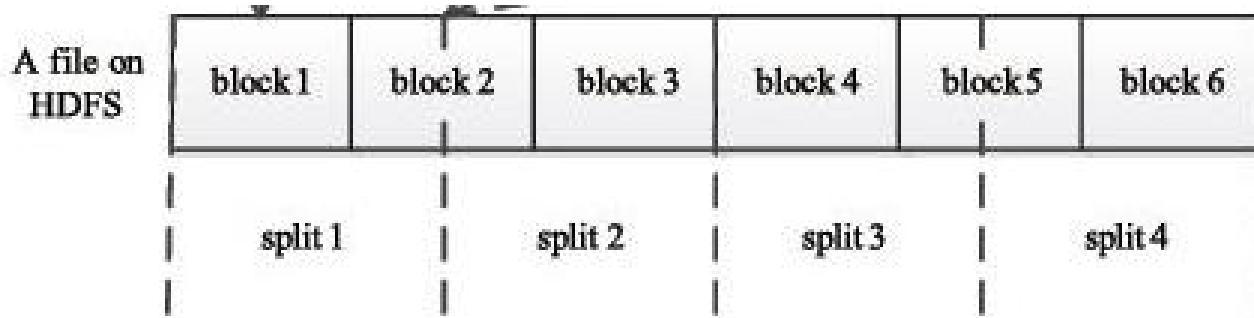
HDFS 以固定大小的 **block** 为基本单位存储数据，而对于 MapReduce 而言，其处理单位是 **split**。split 是一个逻辑概念，它只包含一些元数据信息，比如数据起始位置、数据长度、数据所在节点等。它的划分方法完全由用户自己决定。



7.3.2 MapReduce各个执行阶段

Map任务的数量

- Hadoop为每个split创建一个Map任务，split 的多少决定了Map任务的数目。大多数情况下，理想的分片大小是一个HDFS块



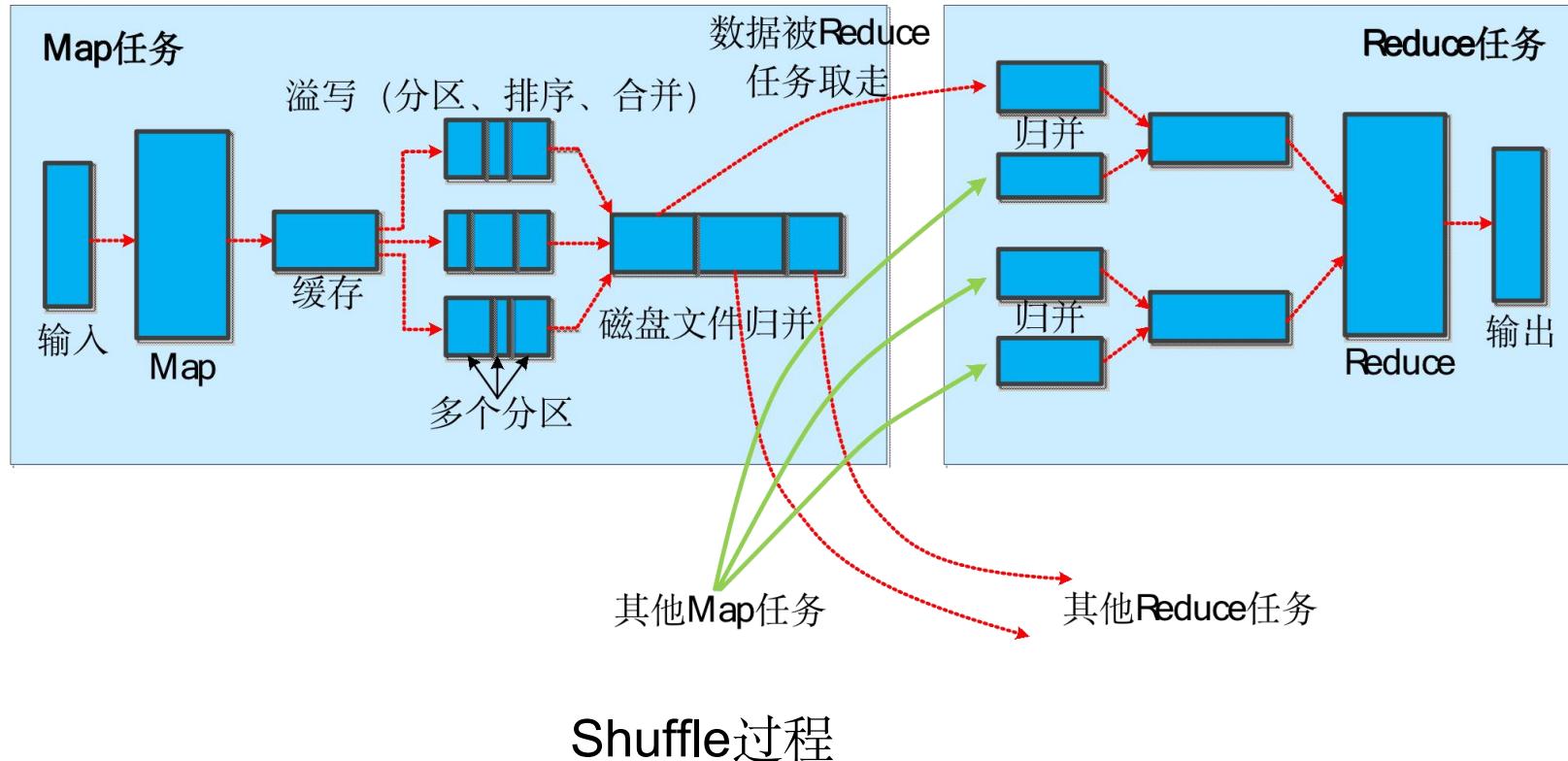
Reduce任务的数量

- 最优的Reduce任务个数取决于集群中可用的reduce任务槽(slot)的数目
- 通常设置比reduce任务槽数目稍微小一些的Reduce任务个数 (这样可以预留一些系统资源处理可能发生的错误)



7.3.3 Shuffle过程详解

1. Shuffle过程简介

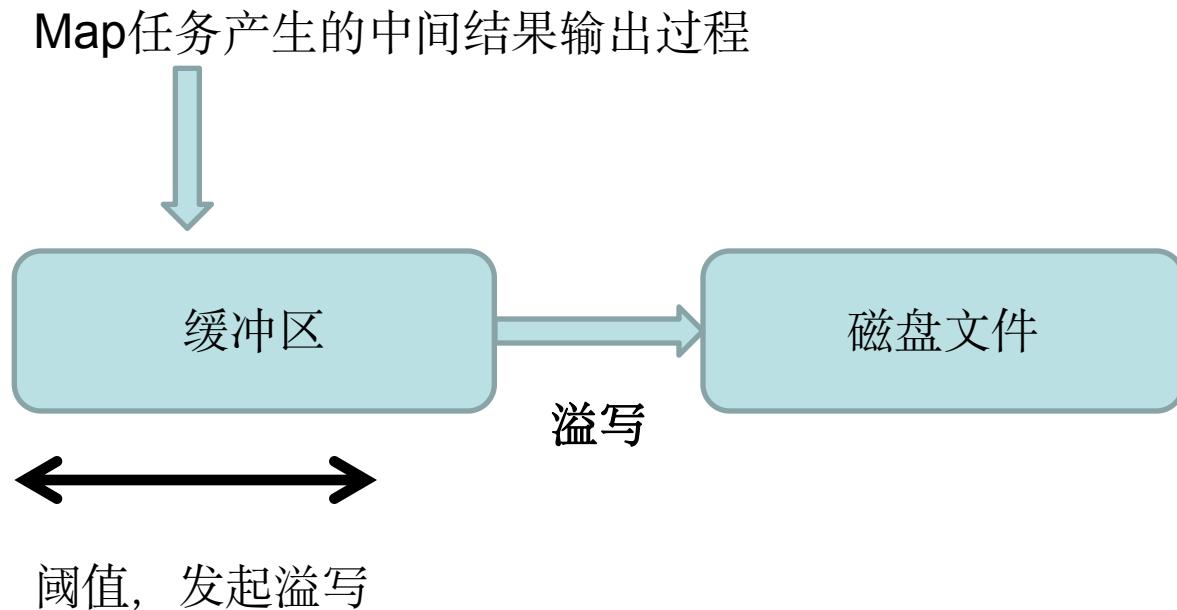


Shuffle过程



7.3.3 Shuffle过程详解

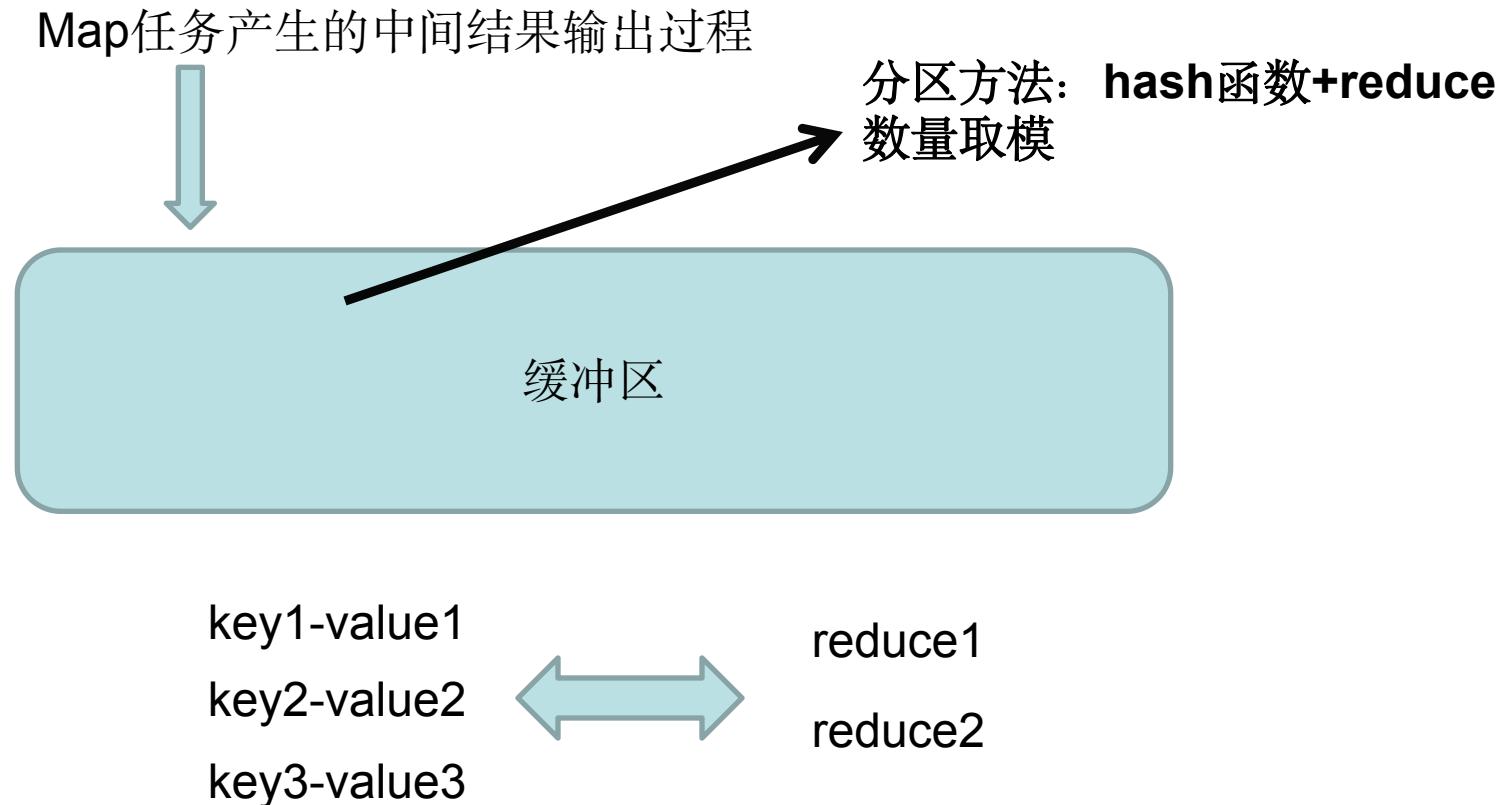
1. 1 溢写





7.3.3 Shuffle过程详解

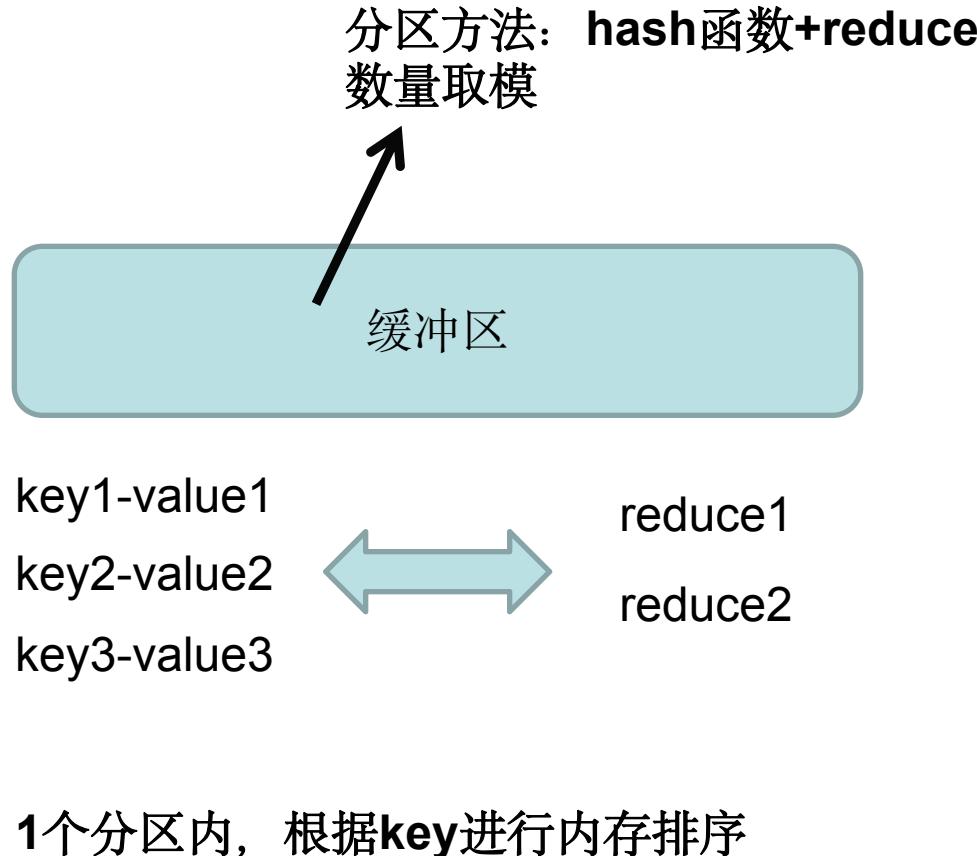
1. 2 分区、排序





7.3.3 Shuffle过程详解

1. 2 分区、排序



哈希: $f(x) = y$

比如对任意输入得到一个整形输出
 $abc \rightarrow 1$
 $abcd \rightarrow 2$
并尽量避免冲突

reduce任务数目取模 (取余)
 $key1 \rightarrow 1$
 $key2 \rightarrow 2$
 $key3 \rightarrow 3$

reduce数目为2

$key1 \rightarrow \text{reduce2}$
 $key2 \rightarrow \text{reduce1}$
 $key3 \rightarrow \text{reduce2}$



7.3.3 Shuffle过程详解

1. 3 合并

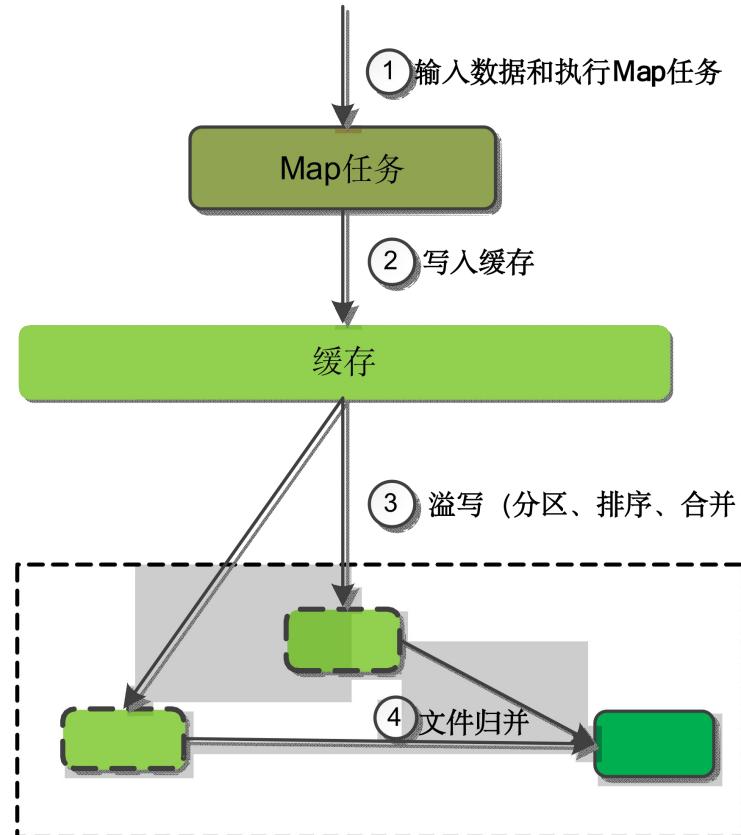
合并是对**key**相同的结果进行**value**相加运算

这个方法，可以用户自定义，也可以在**reduce**过程再进行；



7.3.3 Shuffle过程详解

2. Map端的Shuffle过程



- 每个Map任务分配一个缓存
- MapReduce默认100MB缓存

- 设置溢写比例0.8
- 分区默认采用哈希函数
- 排序是默认的操作
- 排序后可以合并 (Combine)
- 合并不能改变最终结果

- 在Map任务全部结束之前进行归并
- 归并得到一个大的文件，放在本地磁盘
- 文件归并时，如果溢写文件数量大于预定值（默认是3）则可以再次启动Combiner，少于3不需要
- JobTracker会一直监测Map任务的执行，并通知Reduce任务来领取数据

合并 (Combine) 和归并 (Merge) 的区别：

两个键值对<“a”,1>和<“a”,1>，如果合并，会得到<“a”,2>，如果归并，会得到<“a”,<1,1>>



7.3.3 Shuffle过程详解

3. Reduce端的Shuffle过程

- Reduce任务通过RPC向JobTracker询问Map任务是否已经完成，若完成，则领取数据
- Reduce领取数据先放入缓存，来自不同Map机器，先归并，再合并，写入磁盘
- 多个溢写文件归并成一个或多个大文件，文件中的键值对是排序的
- 当数据很少时，不需要溢写到磁盘，直接在缓存中归并，然后输出给Reduce

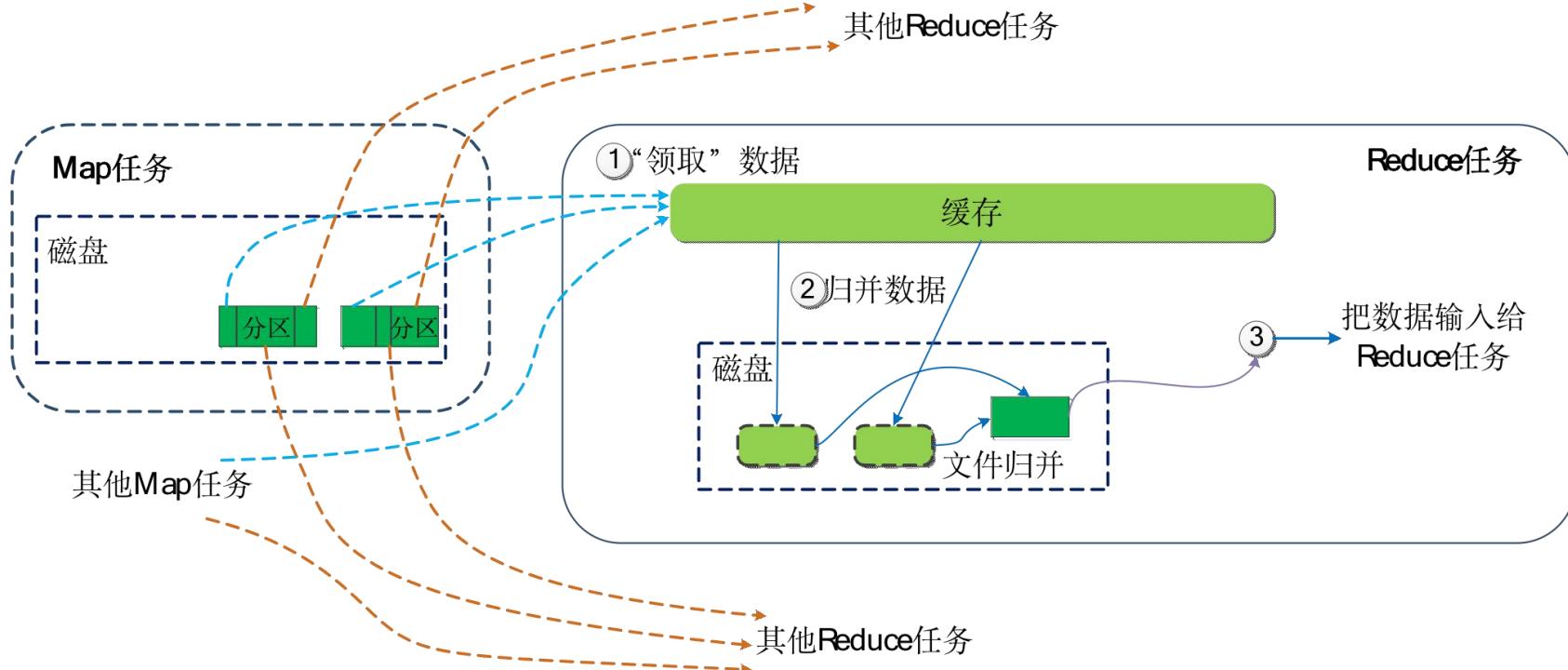
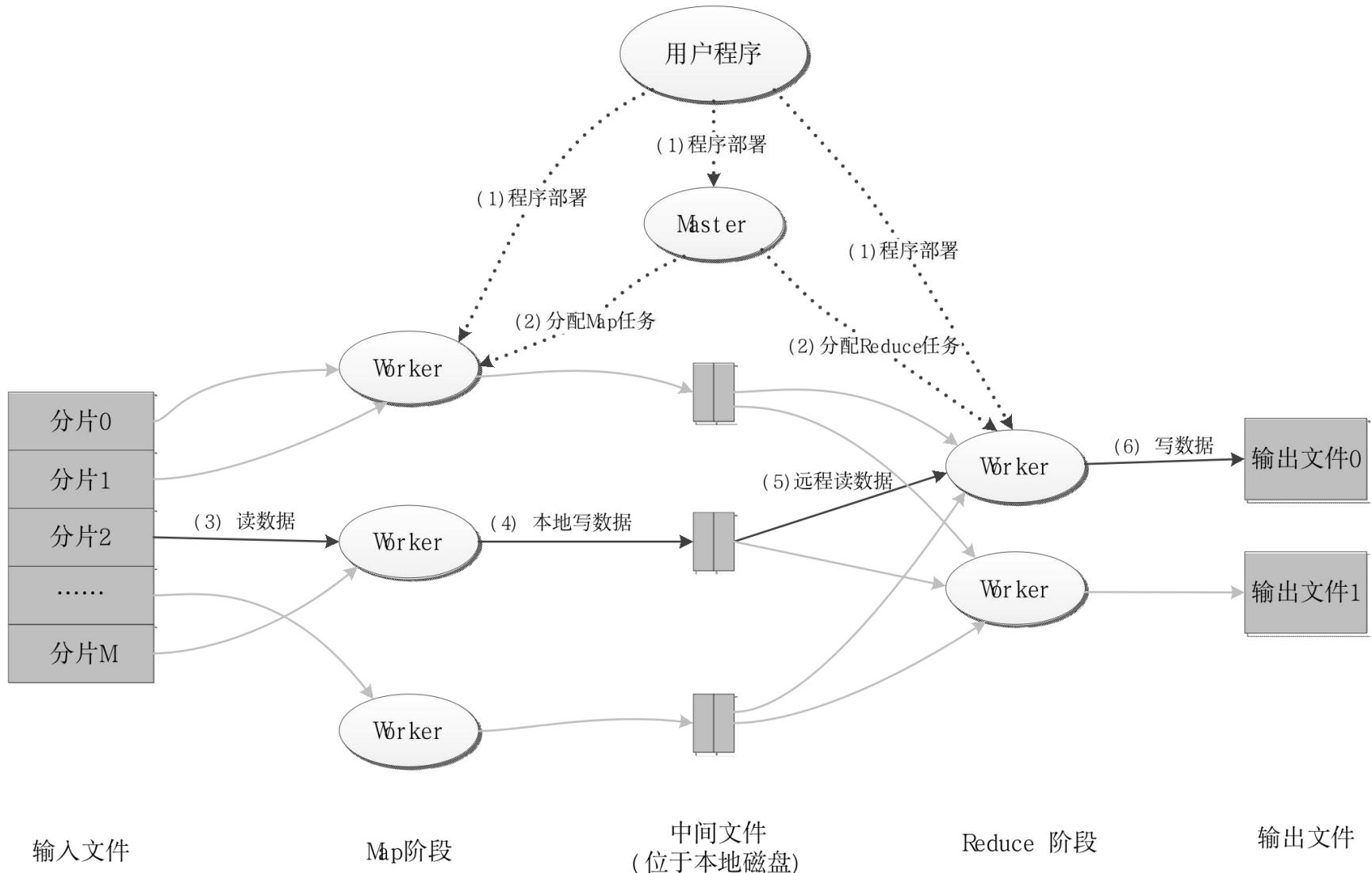


图7-5 Reduce端的Shuffle过程



7.3.4 MapReduce应用程序执行过程





7.4 实例分析: WordCount

- 7.4.1 WordCount程序任务
- 7.4.2 WordCount设计思路
- 7.4.3 一个WordCount执行过程的实例



7.4.1 WordCount程序任务

表7-2 WordCount程序任务

程序	WordCount
输入	一个包含大量单词的文本文件
输出	文件中每个单词及其出现次数（频数），并按照单词字母顺序排序，每个单词和其频数占一行，单词和频数之间有间隔

表7-3 一个WordCount的输入和输出实例

输入	输出
Hello World	Hadoop 1
Hello Hadoop	Hello 3
Hello MapReduce	MapReduce 1
	World 1



7.4.2 WordCount设计思路

- 首先，需要检查WordCount程序任务是否可以采用MapReduce来实现
- 其次，确定MapReduce程序的设计思路
- 最后，确定MapReduce程序的执行过程



7.4.3一个WordCount执行过程的实例

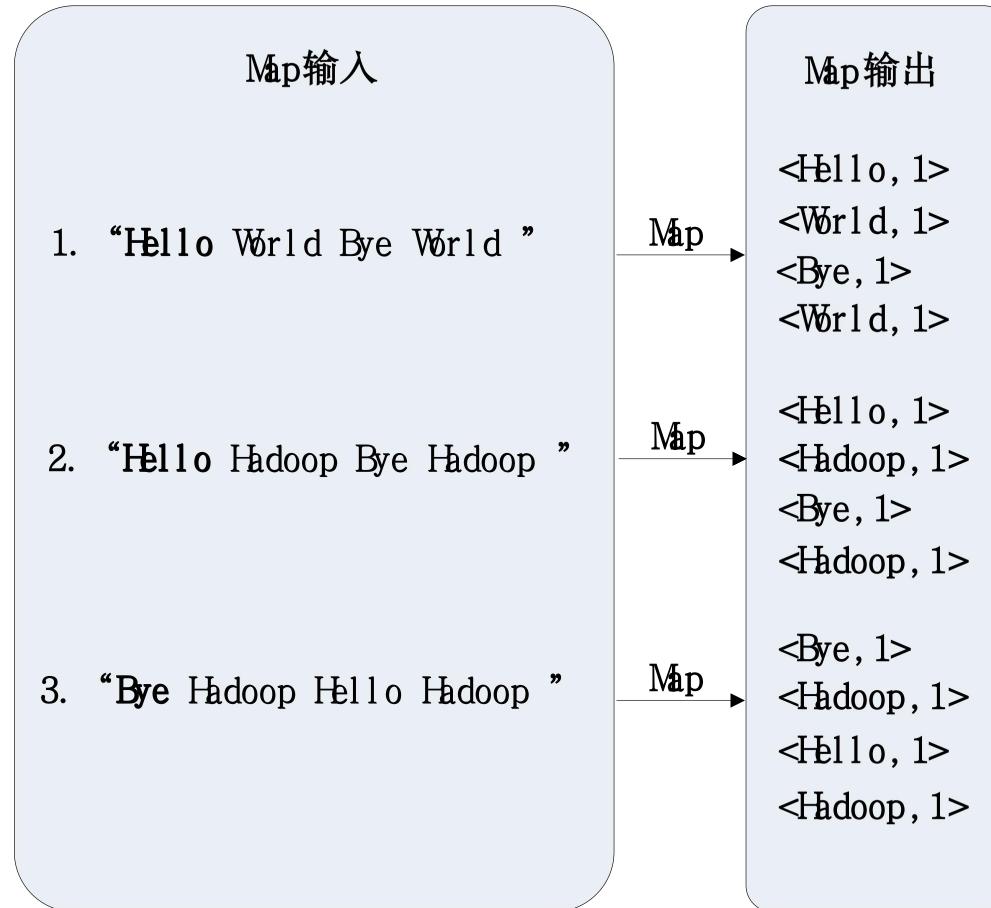


图7-7 Map过程示意图



7.4.3一个WordCount执行过程的实例

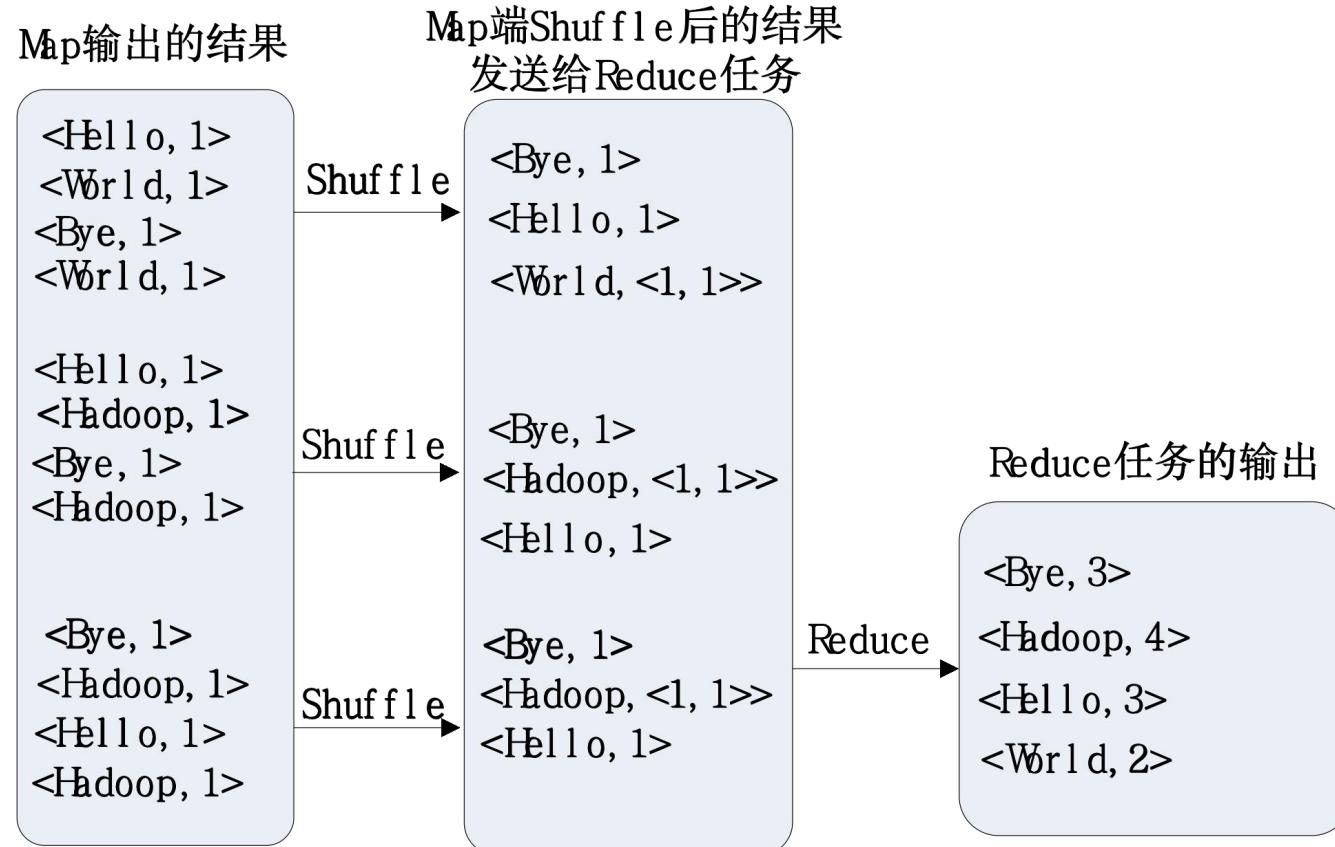


图7-8 用户没有定义Combiner时的Reduce过程示意图



7.4.3一个WordCount执行过程的实例

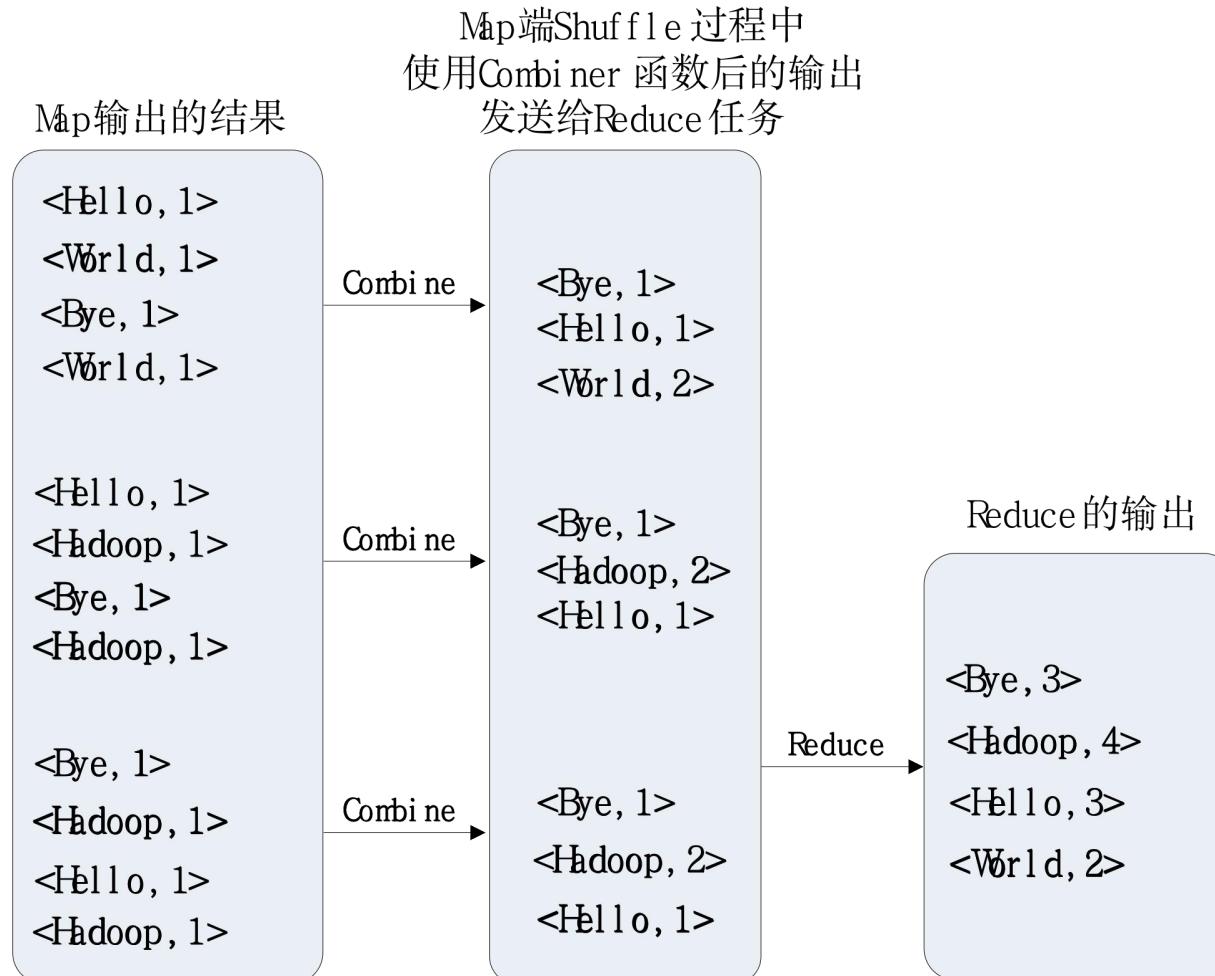


图7-9 用户有定义Combiner时的Reduce过程示意图



7.5 MapReduce的具体应用

MapReduce可以很好地应用于各种计算问题

- 关系代数运算（选择、投影、并、交、差、连接）
- 分组与聚合运算
- 矩阵-向量乘法
- 矩阵乘法



7.5 MapReduce的具体应用

用MapReduce实现关系的自然连接

雇员		
Name	Empld	DeptName
Harry	3415	财务
Sally	2241	销售
George	3401	财务
Harriet	2202	销售

部门	
DeptName	Manager
财务	George
销售	Harriet
生产	Charles

雇员 ⚫ 部门			
Name	Empld	DeptName	Manager
Harry	3415	财务	George
Sally	2241	销售	Harriet
George	3401	财务	George
Harriet	2202	销售	Harriet

- 假设有关系R(A, B)和S(B,C), 对二者进行自然连接操作
- 使用Map过程, 把来自R的每个元组[<a,b>](#)转换成一个键值对[<b, <R,a>>](#), 其中的键就是属性B的值。把关系R包含到值中, 这样做使得我们可以在Reduce阶段, 只把那些来自R的元组和来自S的元组进行匹配。类似地, 使用Map过程, 把来自S的每个元组[<b,c>](#), 转换成一个键值对[<b,<S,c>>](#)
- 所有具有相同B值的元组被发送到同一个Reduce进程中, Reduce进程的任务是, 把来自关系R和S的、具有相同属性B值的元组进行合并
- Reduce进程的输出则是连接后的元组[<a,b,c>](#), 输出被写到一个单独的输出文件中



7.5MapReduce的具体应用

用**MapReduce**实现关系的自然连接

Order

Orderid	Account	Date
1	a	d1
2	a	d2
3	b	d3

Map

Key

Value

1 “Order” ,(a,d1)

2 “Order” ,(a,d2)

3 “Order” ,(b,d3)

Item

Orderid	itemid	Num
1	10	1
1	20	3
2	10	5
2	50	100
3	20	1

Map

Key

Value

Reduce

(1,a,d1,10,1)

(1,a,d1,20,3)

(2,a,d2,10,5)

(2,a,d2,50,100)

(3,b,d3,20,1)

1 “Item” ,(10,1)

1 “Item” ,(20,3)

2 “Item” ,(10,5)

2 “Item” ,(50,100)

3 “Item” ,(20,1)



7.6 MapReduce编程实践

- 7.6.1任务要求
- 7.6.2编写Map处理逻辑
- 7.6.3编写Reduce处理逻辑
- 7.6.4 编写main方法
- 7.6.5 编译打包代码以及运行程序



7.6.1 任务要求

文件**A**的内容如下：

China is my motherland
I love China

文件**B**的内容如下：

I am from China

期望结果如右侧所示：

I	2
is	1
China	3
my	1
love	1
am	1
from	1
motherland	1



7.6.2 编写Map处理逻辑

- Map输入类型为<key,value>
- 期望的Map输出类型为<单词, 出现次数>
- Map输入类型最终确定为<Object,Text>
- Map输出类型最终确定为<Text,IntWritable>

```
public static class MyMapper extends Mapper<Object,Text,Text,IntWritable>{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(Object key, Text value, Context context) throws
IOException,InterruptedException{
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens())
        {
            word.set(itr.nextToken());
            context.write(word,one);
        }
    }
}
```



7.6.3 编写Reduce处理逻辑

- 在Reduce处理数据之前，Map的结果首先通过Shuffle阶段进行整理
- Reduce阶段的任务：对输入数字序列进行求和
- Reduce的输入数据为<key,Iterable容器>

Reduce任务的输入数据：

<"I",<1,1>>

<"is",1>

.....

<"from",1>

<"China",<1,1,1>>



7.6.3 编写Reduce处理逻辑

```
public static class MyReducer extends  
Reducer<Text,IntWritable,Text,IntWritable>{  
    private IntWritable result = new IntWritable();  
    public void reduce(Text key, Iterable<IntWritable>  
values, Context context) throws IOException,InterruptedException {  
    int sum = 0;  
    for (IntWritable val : values)  
    {  
        sum += val.get();  
    }  
    result.set(sum);  
    context.write(key,result);  
}  
}
```



7.6.4 编写main方法

```
public static void main(String[] args) throws Exception{  
    Configuration conf = new Configuration(); //程序运行时参数  
    String[] otherArgs = new GenericOptionsParser(conf,args).getRemainingArgs();  
    if (otherArgs.length != 2)  
    {  
        System.err.println("Usage: wordcount <in> <out>");  
        System.exit(2);  
    }  
    Job job = new Job(conf,"word count"); //设置环境参数  
    job.setJarByClass(WordCount.class); //设置整个程序的类名  
    job.setMapperClass(MyMapper.class); //添加MyMapper类  
    job.setReducerClass(MyReducer.class); //添加MyReducer类  
    job.setOutputKeyClass(Text.class); //设置输出类型  
    job.setOutputValueClass(IntWritable.class); //设置输出类型  
    FileInputFormat.addInputPath(job,new Path(otherArgs[0])); //设置输入文件  
    FileOutputFormat.setOutputPath(job,new Path(otherArgs[1])); //设置输出文件  
    System.exit(job.waitForCompletion(true)?0:1);  
}
```



完整代码

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class WordCount{

    //WordCount类的具体代码见下一页
}
```



```
public class WordCount{
    public static class MyMapper extends Mapper<Object,Text,Text,IntWritable>{
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(Object key, Text value, Context context) throws IOException,InterruptedException{
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()){
                word.set(itr.nextToken());
                context.write(word,one);
            }
        }
    }
    public static class MyReducer extends Reducer<Text,IntWritable,Text,IntWritable>{
        private IntWritable result = new IntWritable();
        public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException,InterruptedException{
            int sum = 0;
            for (IntWritable val : values)
            {
                sum += val.get();
            }
            result.set(sum);
            context.write(key,result);
        }
    }
    public static void main(String[] args) throws Exception{
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf,args).getRemainingArgs();
        if (otherArgs.length != 2)
        {
            System.err.println("Usage: wordcount <in> <out>");
            System.exit(2);
        }
        Job job = new Job(conf,"word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(MyMapper.class);
        job.setReducerClass(MyReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job,new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job,new Path(otherArgs[1]));
        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```



7.6.5 编译打包代码以及运行程序

实验步骤：

- 使用java编译程序，生成.class文件
- 将.class文件打包为jar包
- 运行jar包（需要启动Hadoop）
- 查看结果



7.6.5 编译打包代码以及运行程序

Hadoop 3.1.3 版本中的依赖 jar

使用 Hadoop 3.1.3 运行 WordCount 实例至少需要如下三个 jar:

- \$HADOOP_HOME/share/hadoop/common/hadoop-common-3.1.3.jar
- \$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-core-3.1.3.jar
- \$HADOOP_HOME/share/hadoop/common/lib/commons-cli-1.2.jar

通过命令 `hadoop classpath` 可以得到运行 Hadoop 程序所需的全部 classpath 信息



7.6.5 编译打包代码以及运行程序

将 Hadoop 的 classpath 信息添加到 CLASSPATH 变量中，在 `~/.bashrc` 中增加如下几行：

```
export HADOOP_HOME=/usr/local/hadoop
export
CLASSPATH=$(($HADOOP_HOME/bin/hadoop classpath)):$CLASSPATH
```

执行 `source ~/.bashrc` 使变量生效，接着就可以通过 `javac` 命令编译 `WordCount.java`

```
$ javac WordCount.java
```

接着把 `.class` 文件打包成 `jar`，才能在 Hadoop 中运行：

```
jar -cvf WordCount.jar ./WordCount*.class
```

运行程序：

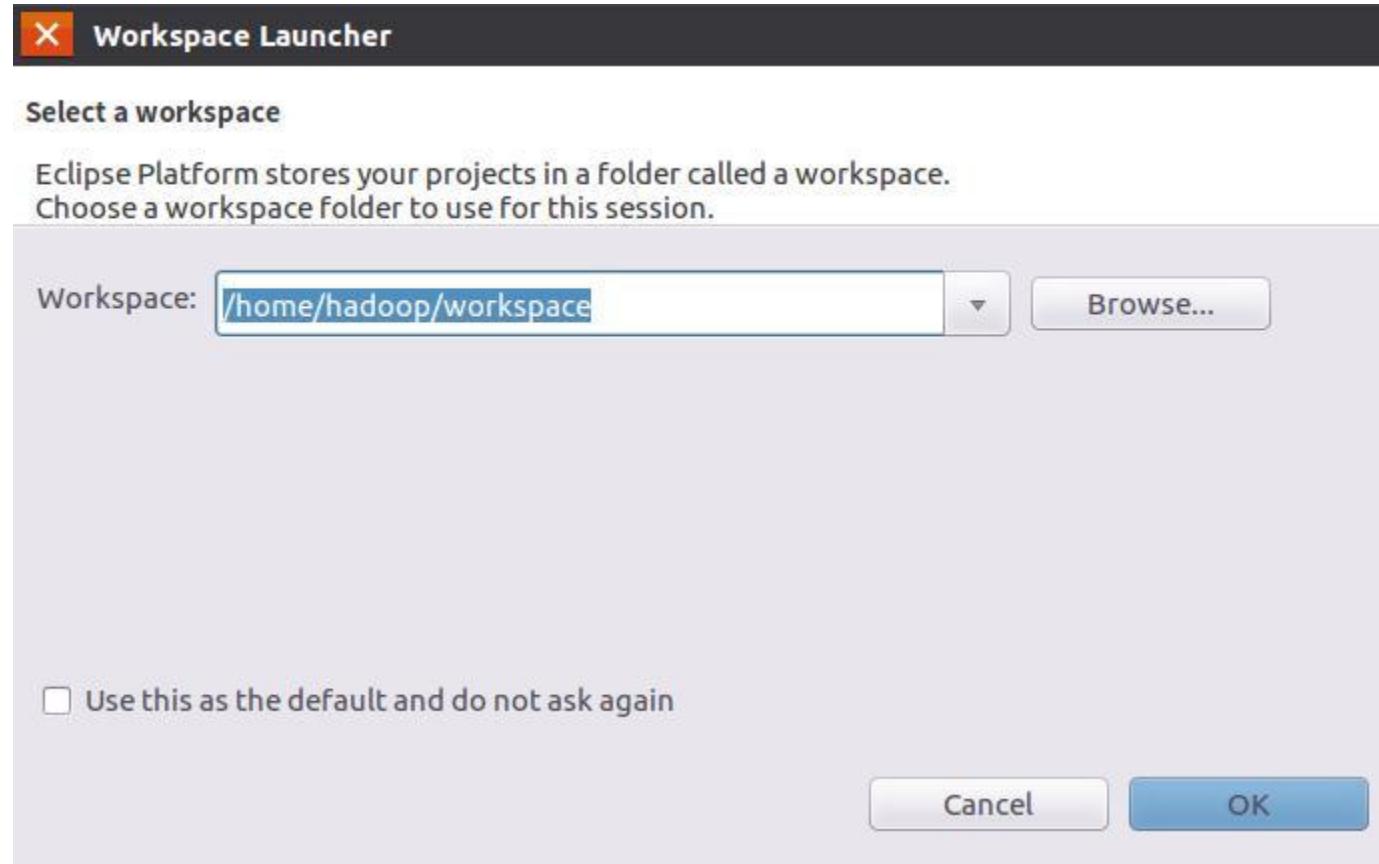
```
/usr/local/hadoop/bin/hadoop jar WordCount.jar WordCount input output
```



7.6.5 编译打包代码以及运行程序

1、在Eclipse中创建项目

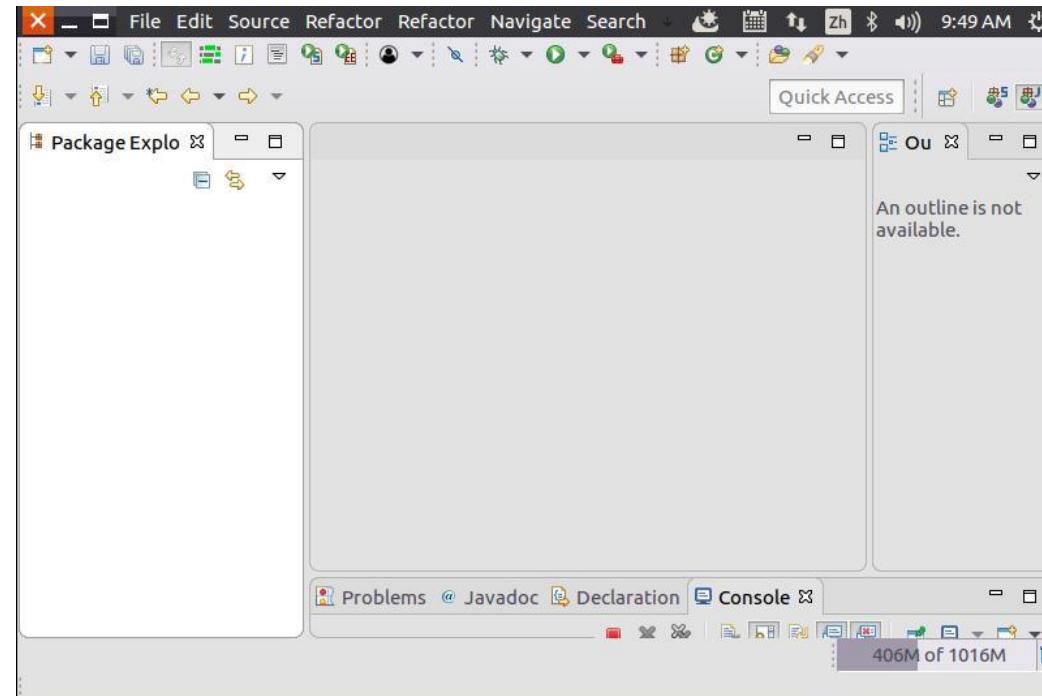
首先，启动Eclipse，启动以后会弹出如下图所示界面，提示设置工作空间（workspace）。





7.6.5 编译打包代码以及运行程序

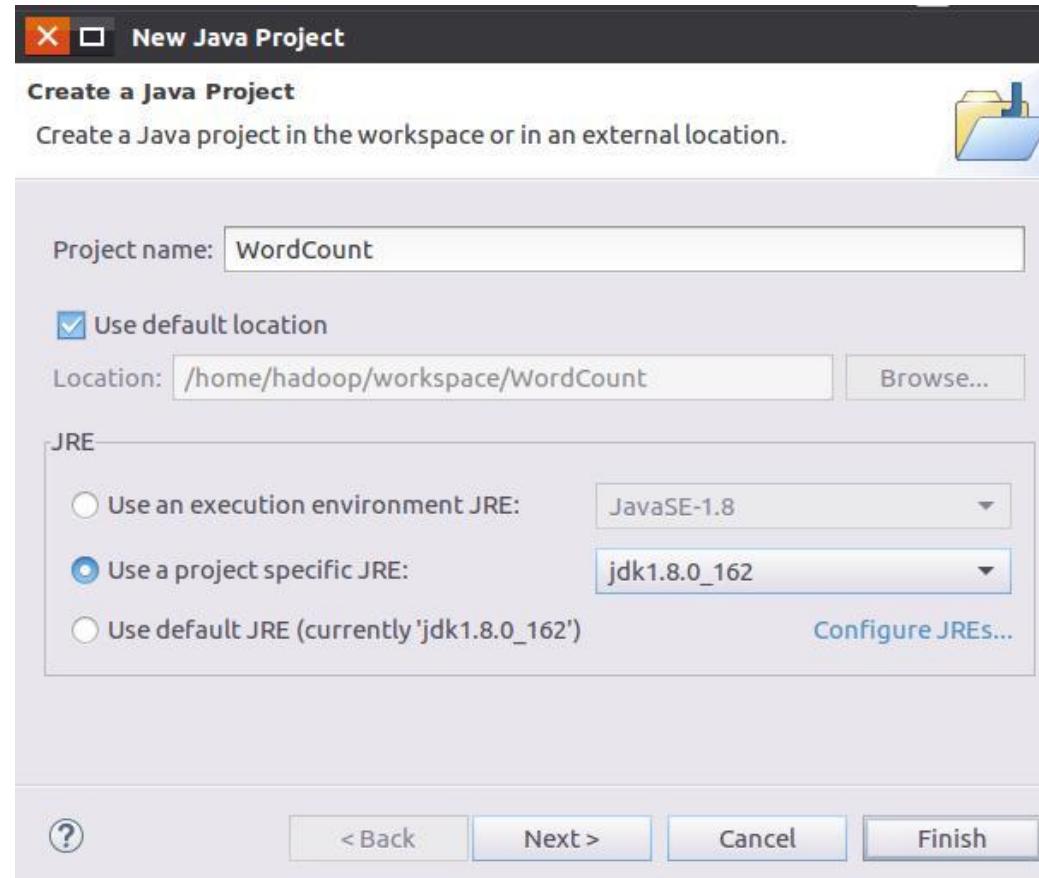
可以直接采用默认的设置“/home/hadoop/workspace”，点击“OK”按钮。可以看出，由于当前是采用hadoop用户登录了Linux系统，因此，默认的工作空间目录位于hadoop用户目录“/home/hadoop”下。
Eclipse启动以后，呈现的界面如下图所示。





7.6.5 编译打包代码以及运行程序

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





7.6.5 编译打包代码以及运行程序

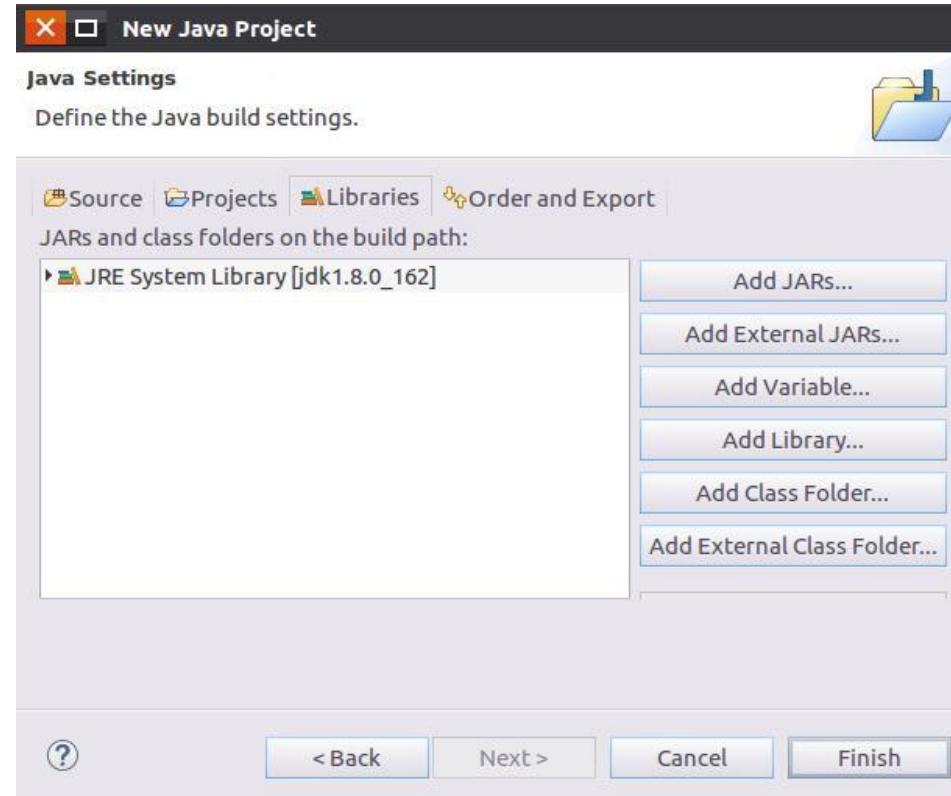
在“Project name”后面输入工程名称“WordCount”，选中“Use default location”，让这个Java工程的所有文件都保存到“/home/hadoop/workspace/WordCount”目录下。在“JRE”这个选项卡中，可以选择当前的Linux系统中已经安装好的JDK，比如jdk1.8.0_162。然后，点击界面底部的“Next>”按钮，进入下一步的设置。



7.6.5 编译打包代码以及运行程序

2. 为项目添加需要用到的JAR包

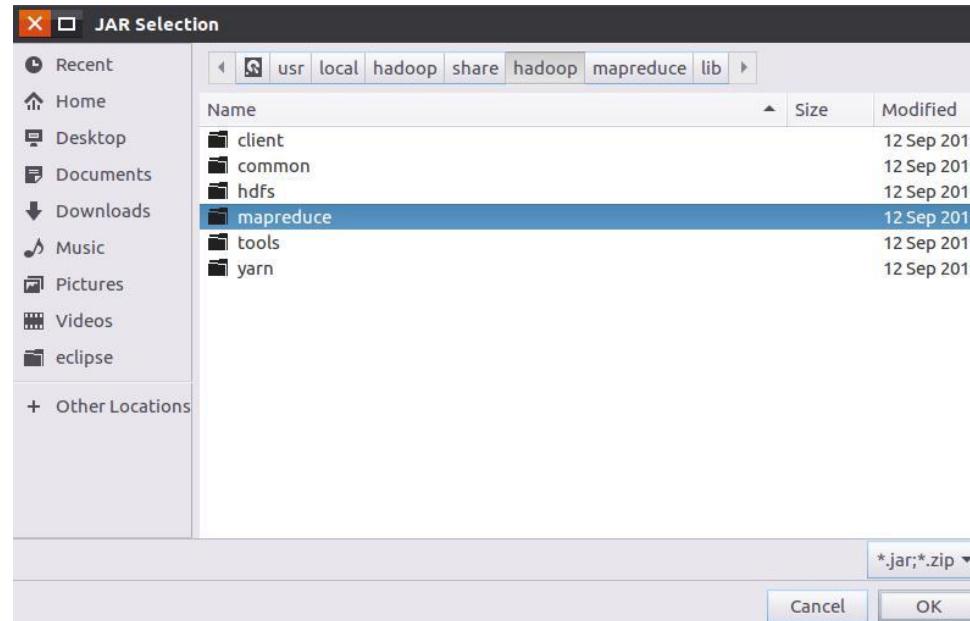
进入下一步的设置以后，会弹出如下图所示界面。





7.6.5 编译打包代码以及运行程序

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包中包含了与Hadoop相关的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，弹出如下图所示界面。



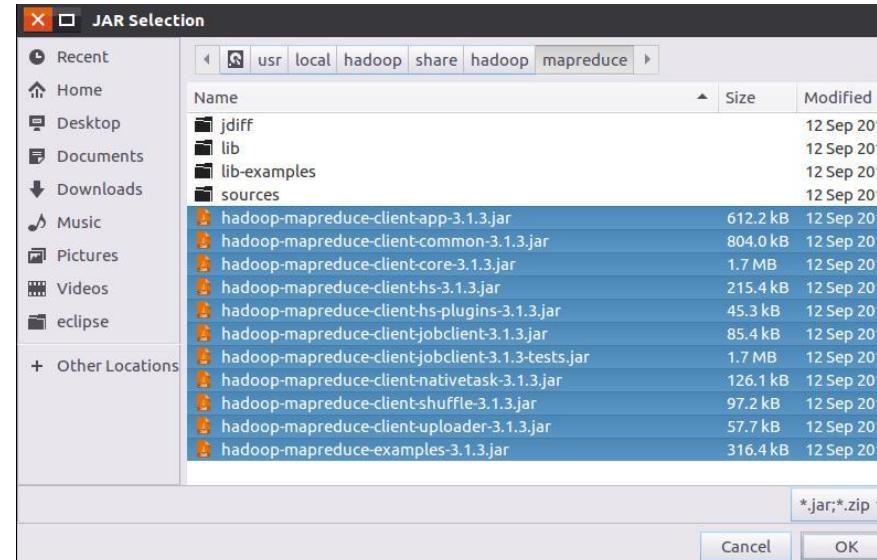


7.6.5 编译打包代码以及运行程序

在该界面中，上面有一排目录按钮（即“usr”、“local”、“hadoop”、“share”、“hadoop”、“mapreduce”和“lib”），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MapReduce程序，一般需要向Java工程中添加以下JAR包：

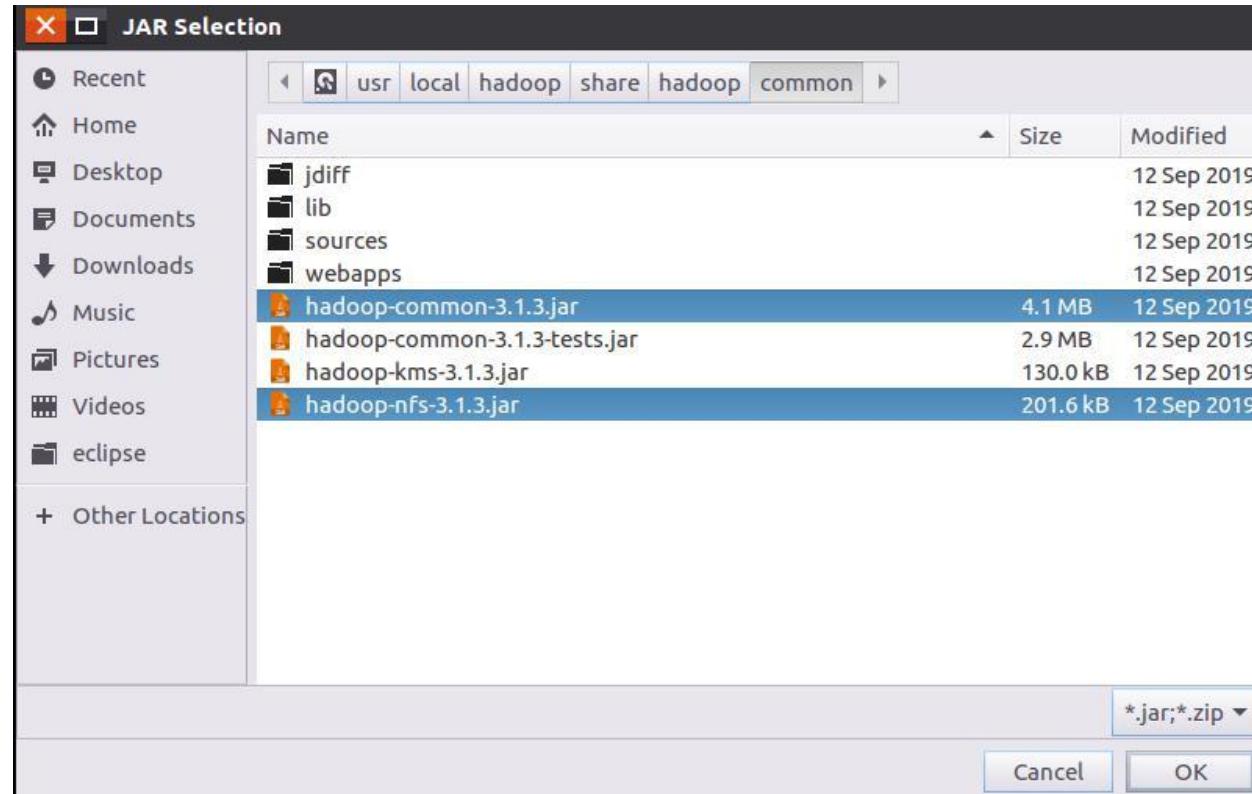
- (1) “/usr/local/hadoop/share/hadoop/common”目录下的hadoop-common-3.1.3.jar和hadoop-nfs-3.1.3.jar;
- (2) “/usr/local/hadoop/share/hadoop/common/lib”目录下的所有JAR包;
- (3) “/usr/local/hadoop/share/hadoop/mapreduce”目录下的所有JAR包，但是，不包括jdiff、lib、lib-examples和sources目录，具体如下图所示。





7.6.5 编译打包代码以及运行程序

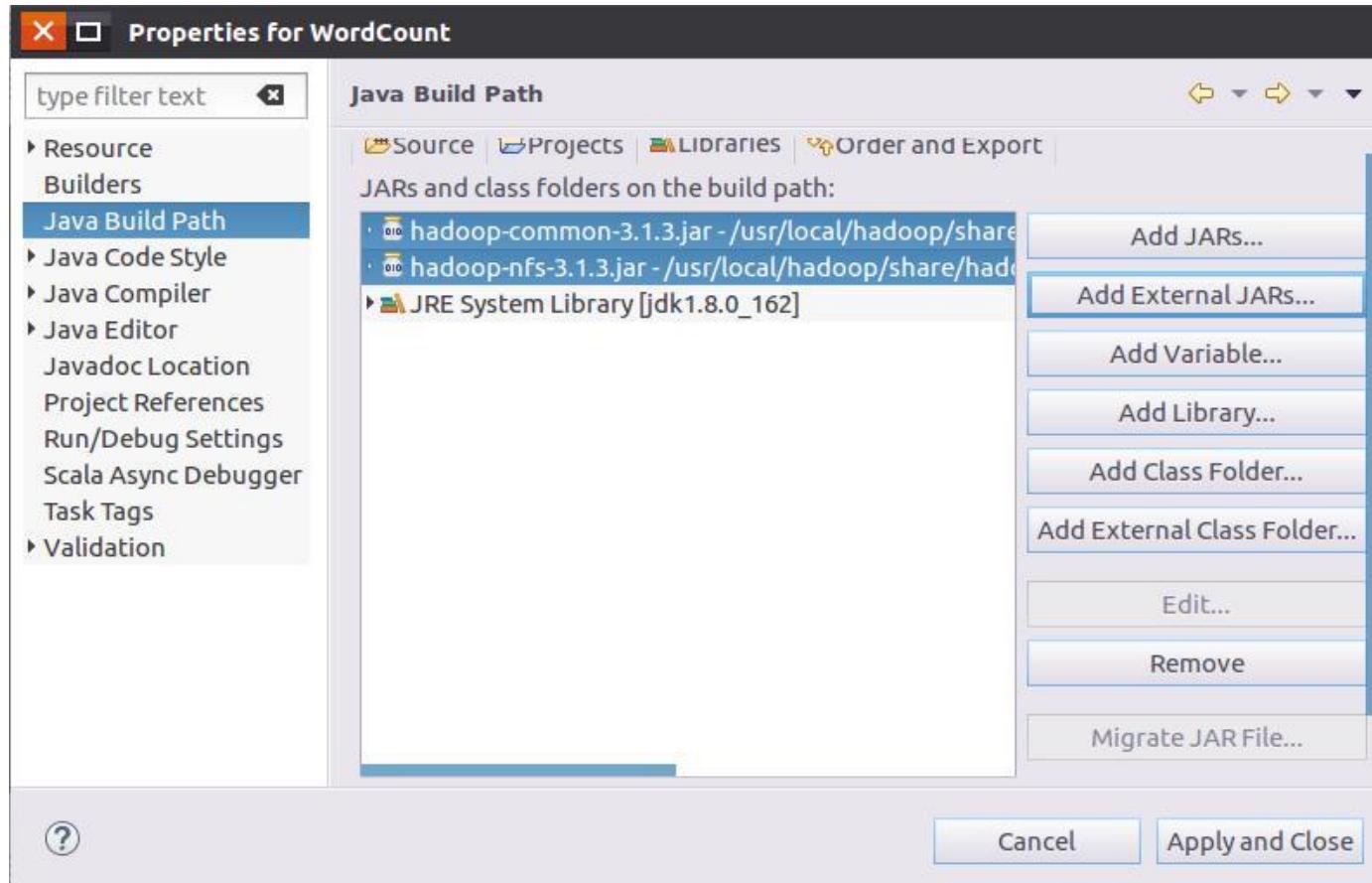
(4) “/usr/local/hadoop/share/hadoop/mapreduce/lib”目录下的所有JAR包。比如，如果要把“/usr/local/hadoop/share/hadoop/common”目录下的hadoop-common-3.1.3.jar和hadoop-nfs-3.1.3.jar添加到当前的Java工程中，可以在界面中点击相应的目录按钮，进入到common目录，然后，界面会显示出common目录下的所有内容（如下图所示）。





7.6.5 编译打包代码以及运行程序

请在界面中用鼠标点击选中hadoop-common-3.1.3.jar和haoop-nfs-3.1.3.jar, 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。





7.6.5 编译打包代码以及运行程序

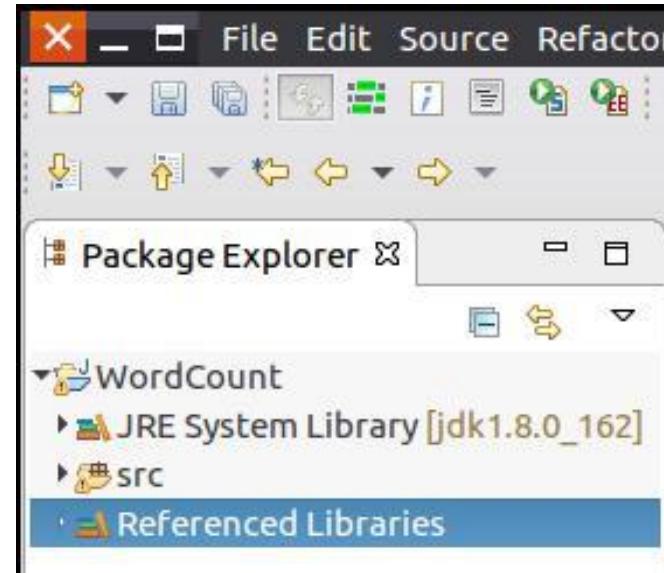
从这个界面中可以看出，`hadoop-common-3.1.3.jar`和
`hadoop-nfs-3.1.3.jar`已经被添加到当前Java工程中。然后，按照类似的操作方法，可以再次点击“**Add External JARs...**”按钮，把剩余的其他JAR包都添加进来。需要注意的是，当需要选中某个目录下的所有JAR包时，可以使用“**Ctrl+A**”组合键进行全选操作。全部添加完毕以后，就可以点击界面右下角的“**Finish**”按钮，完成Java工程WordCount的创建。



7.6.5 编译打包代码以及运行程序

3、编写Java应用程序

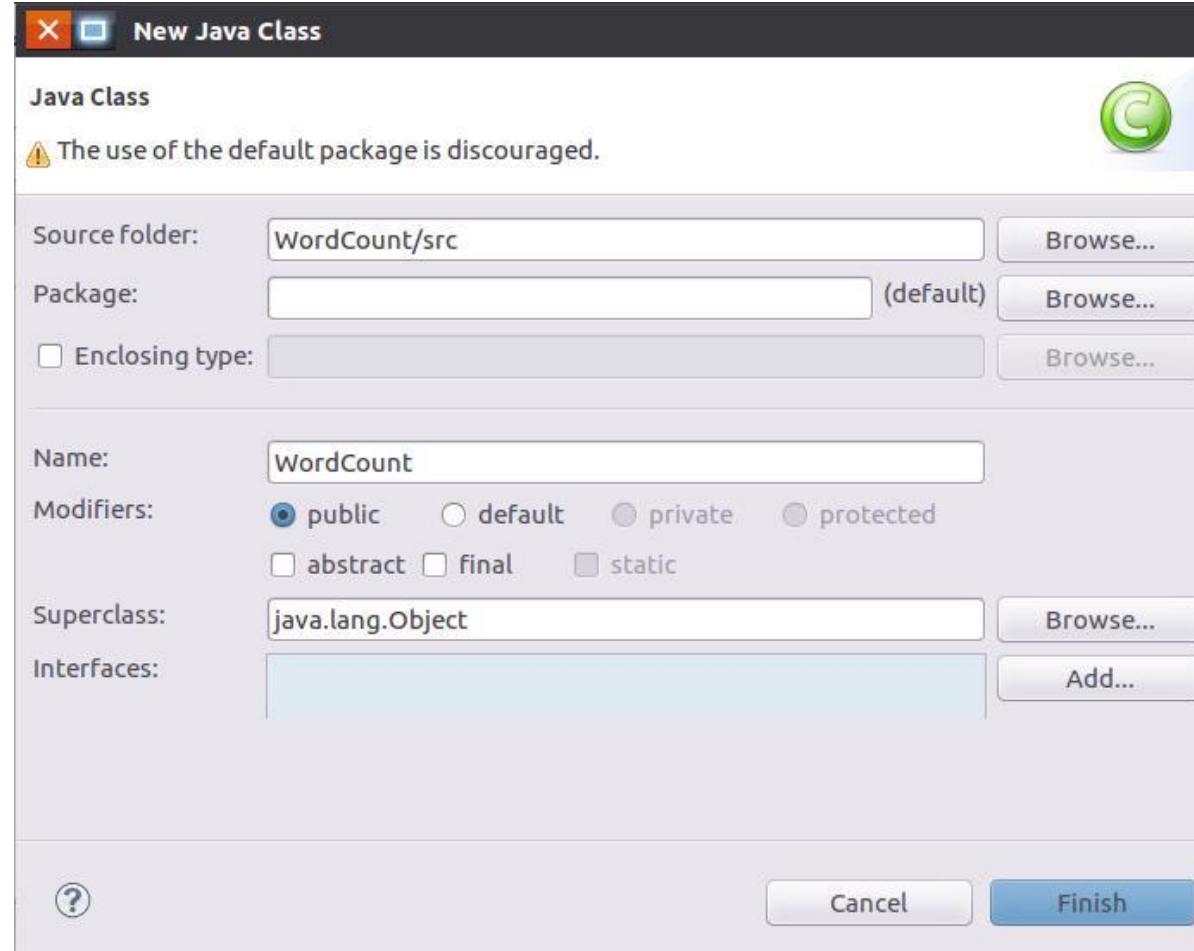
下面编写一个Java应用程序，即WordCount.java。请在 Eclipse 工作界面左侧的“Package Explorer”面板中（如下图所示），找到刚才创建好的工程名称“WordCount”，然后在该工程名称上点击鼠标右键，在弹出的菜单中选择“New->Class”菜单。





7.6.5 编译打包代码以及运行程序

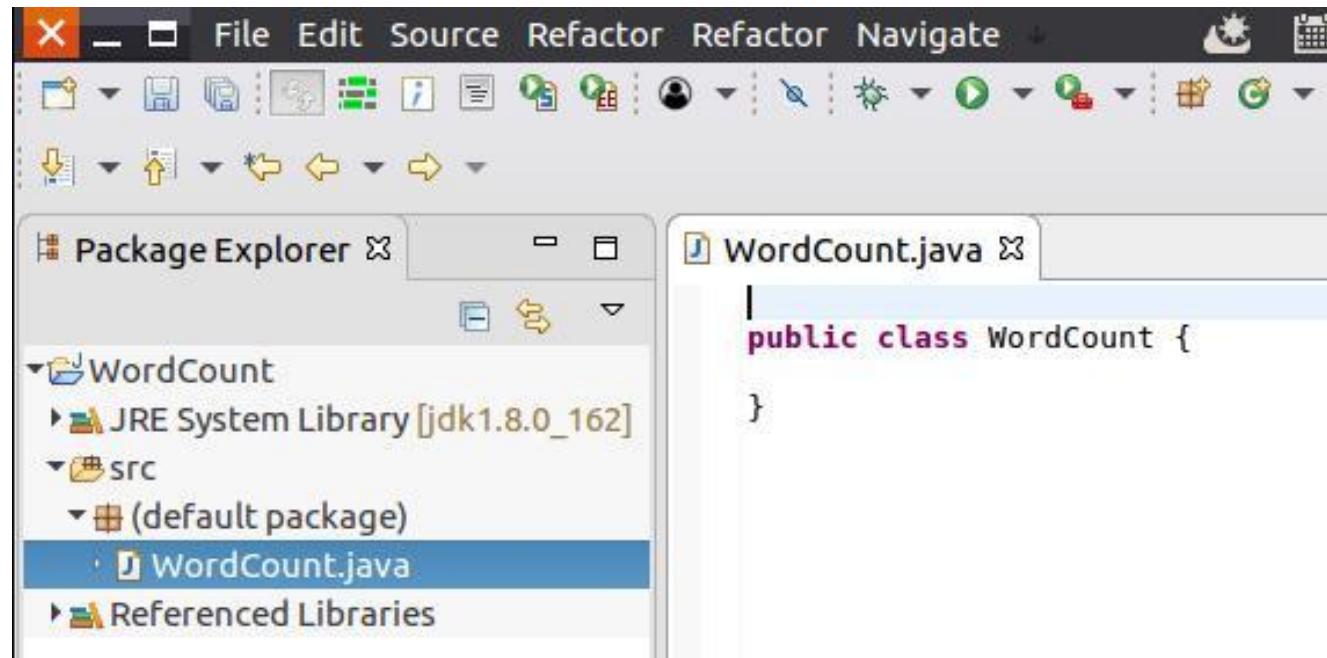
选择“New->Class”菜单以后会出现如下图所示界面。





7.6.5 编译打包代码以及运行程序

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“WordCount”，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。





7.6.5 编译打包代码以及运行程序

可以看出，Eclipse自动创建了一个名为“WordCount.java”的源代码文件，并且包含了代码“`public class WordCount{}`”，请清空该文件里面的代码，然后在该文件中输入完整的词频统计程序代码，具体如下：



7.6.5 编译打包代码以及运行程序

```
import java.io.IOException;
import java.util.Iterator;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
public class WordCount {
    public WordCount() {
    }
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        String[] otherArgs = (new GenericOptionsParser(conf, args)).getRemainingArgs();
        if(otherArgs.length < 2) {
            System.err.println("Usage: wordcount <in> [<in>...] <out>");
            System.exit(2);
        }
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(WordCount.TokenizerMapper.class);
        job.setCombinerClass(WordCount.IntSumReducer.class);
        job.setReducerClass(WordCount.IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        for(int i = 0; i < otherArgs.length - 1; ++i) {
            FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
        }
    }
}
```



7.6.5 编译打包代码以及运行程序

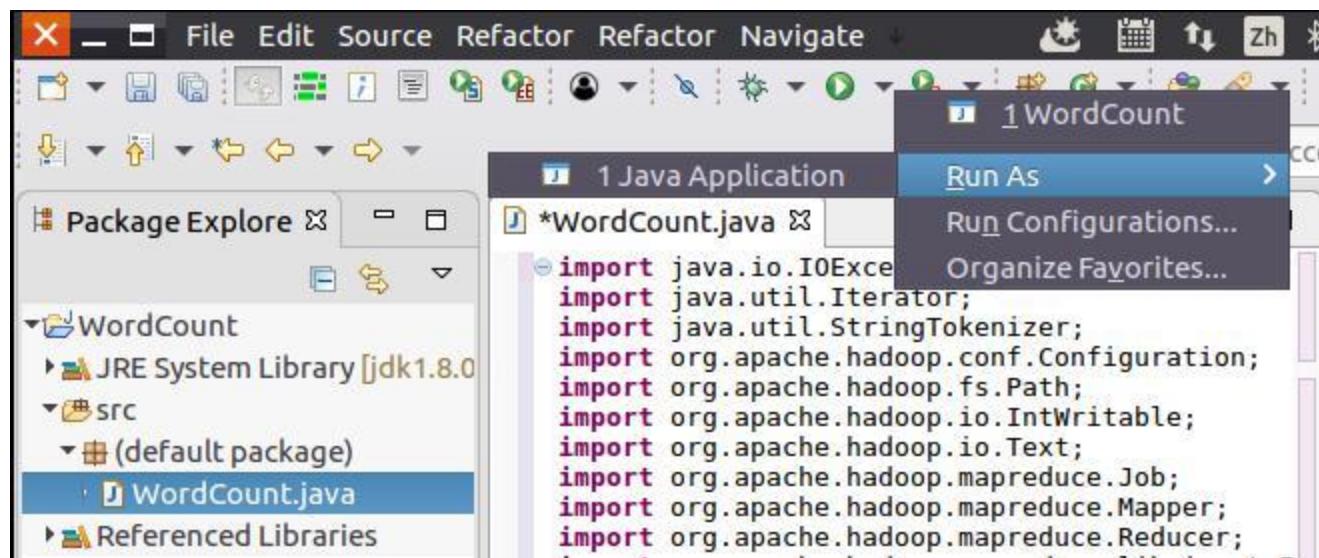
```
FileOutputFormat.setOutputPath(job, new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true)?0:1);
}
public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
    private static final IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public TokenizerMapper() {
    }
    public void map(Object key, Text value, Mapper<Object, Text, Text, IntWritable>.Context context) throws IOException,
InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while(itr.hasMoreTokens()) {
            this.word.set(itr.nextToken());
            context.write(this.word, one);
        }
    }
}
public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();
    public IntSumReducer() {
    }
    public void reduce(Text key, Iterable<IntWritable> values, Reducer<Text, IntWritable, Text, IntWritable>.Context context) throws
IOException, InterruptedException {
        int sum = 0;
        IntWritable val;
        for(Iterator i$ = values.iterator(); i$.hasNext(); sum += val.get()) {
            val = (IntWritable)i$.next();
        }
        this.result.set(sum);
        context.write(key, this.result);
    }
}
```



7.6.5 编译打包代码以及运行程序

4、编译打包程序

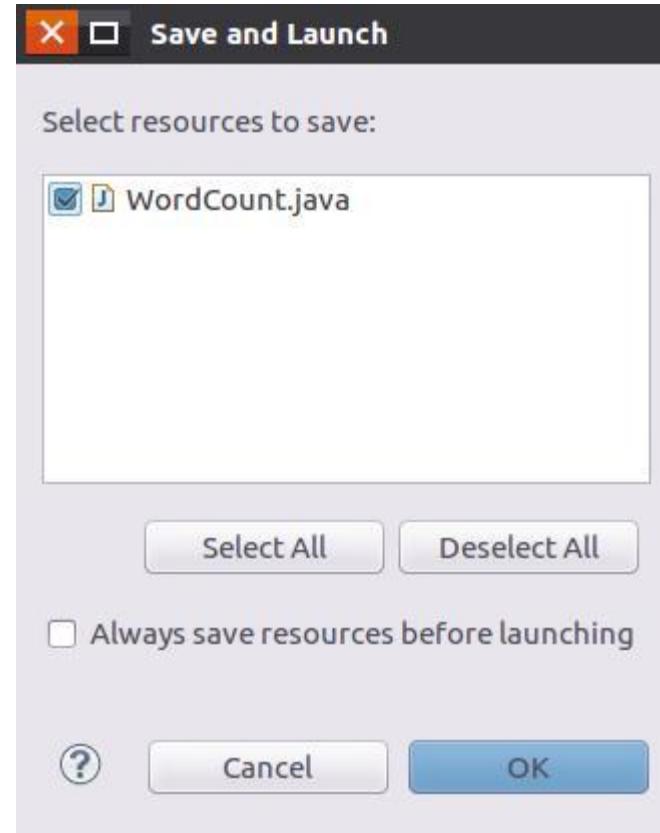
现在就可以编译上面编写的代码。可以直接点击Eclipse工作界面上部的运行程序的快捷按钮，当把鼠标移动到该按钮上时，在弹出的菜单中选择“Run as”，继续在弹出来的菜单中选择“Java Application”，如下图所示。





7.6.5 编译打包代码以及运行程序

然后，会弹出如下图所示界面。





7.6.5 编译打包代码以及运行程序

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）。

```
<terminated> WordCount [Java Application] /usr/lib/jvm/jdk1.8.0_162/bin/java (Jan 27, 2020, 10:30:40 AM)
Usage: wordcount <in> [<in>...] <out>
```



7.6.5 编译打包代码以及运行程序

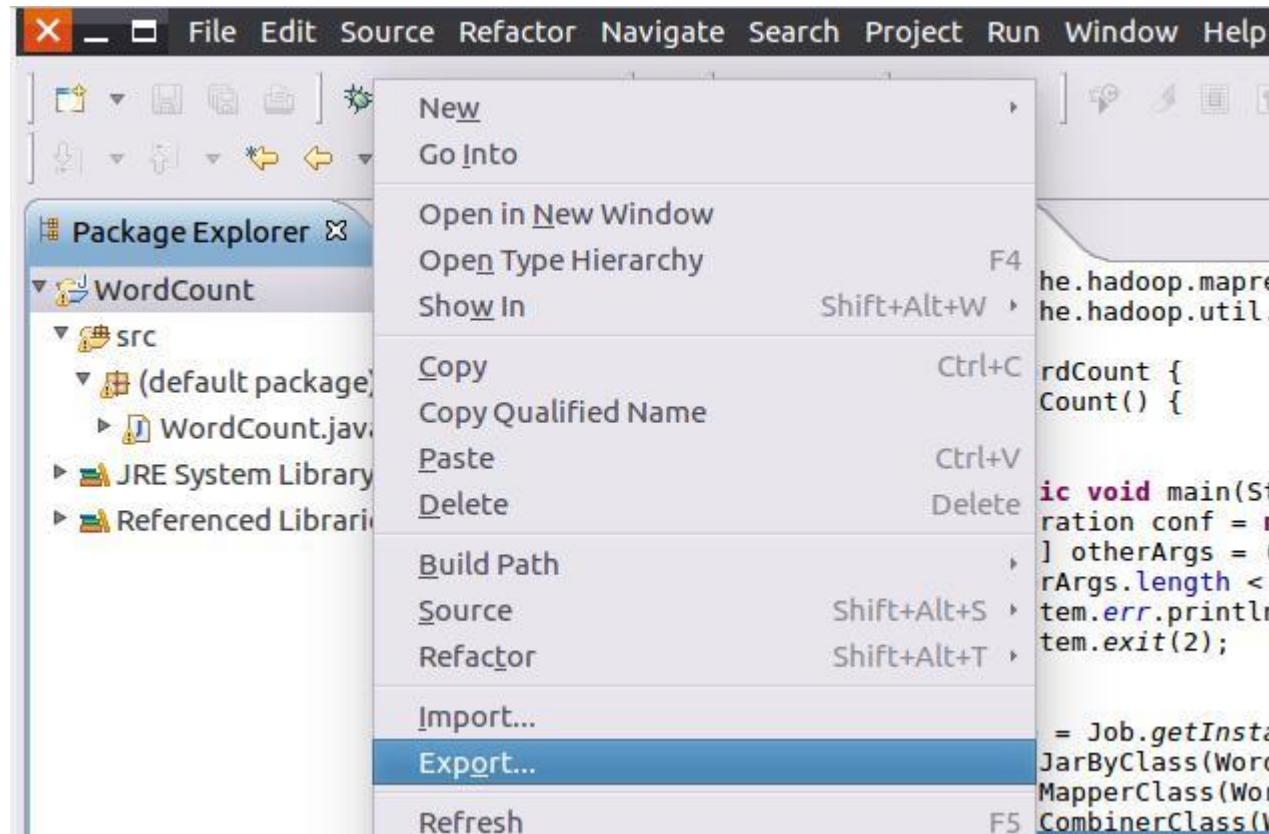
下面就可以把Java应用程序打包生成JAR包，部署到Hadoop平台上运行。现在可以把词频统计程序放在“/usr/local/hadoop/myapp”目录下。如果该目录不存在，可以使用如下命令创建：

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



7.6.5 编译打包代码以及运行程序

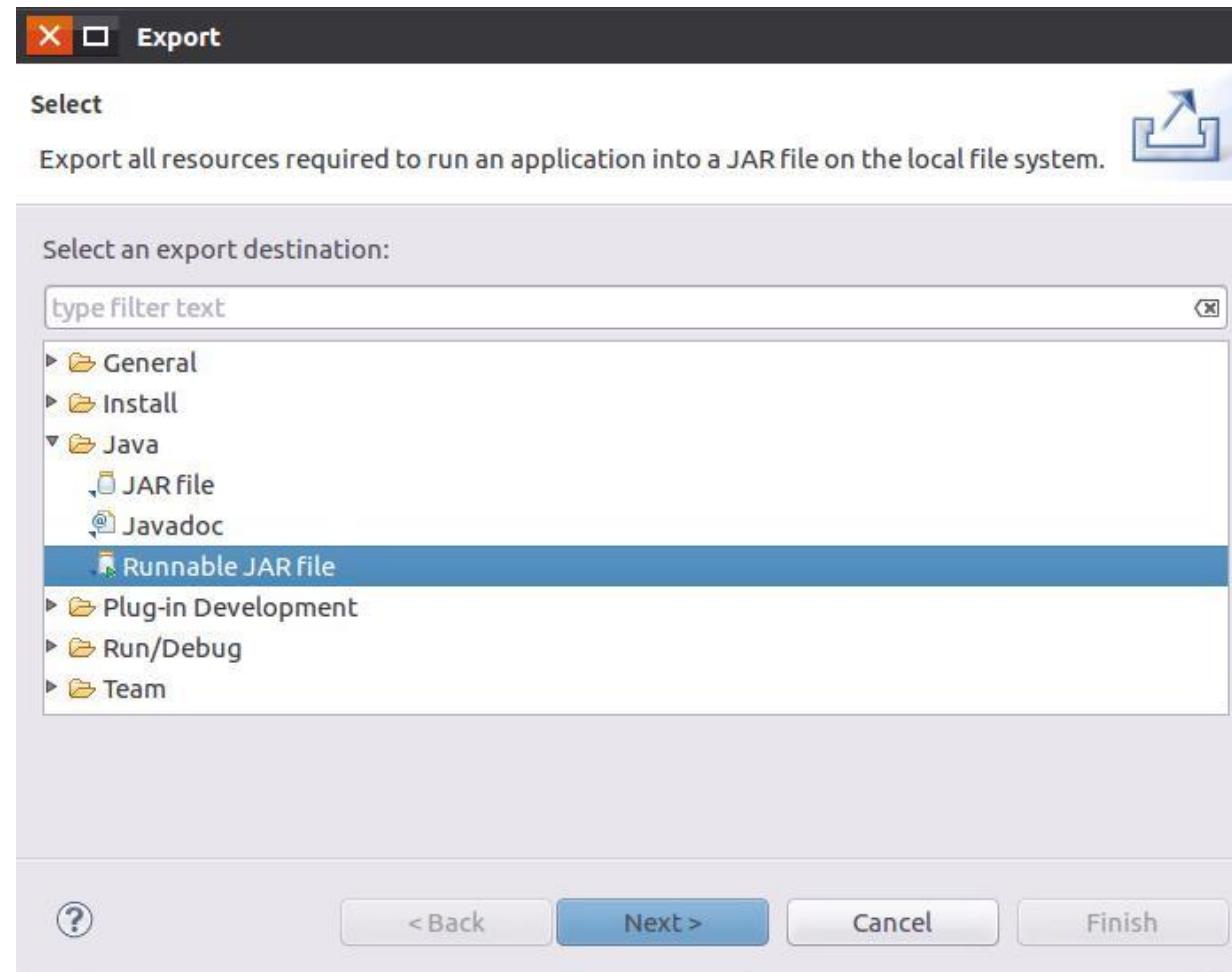
首先，请在Eclipse工作界面左侧的“Package Explorer”面板中，在工程名称“WordCount”上点击鼠标右键，在弹出的菜单中选择“Export”，如下图所示。





7.6.5 编译打包代码以及运行程序

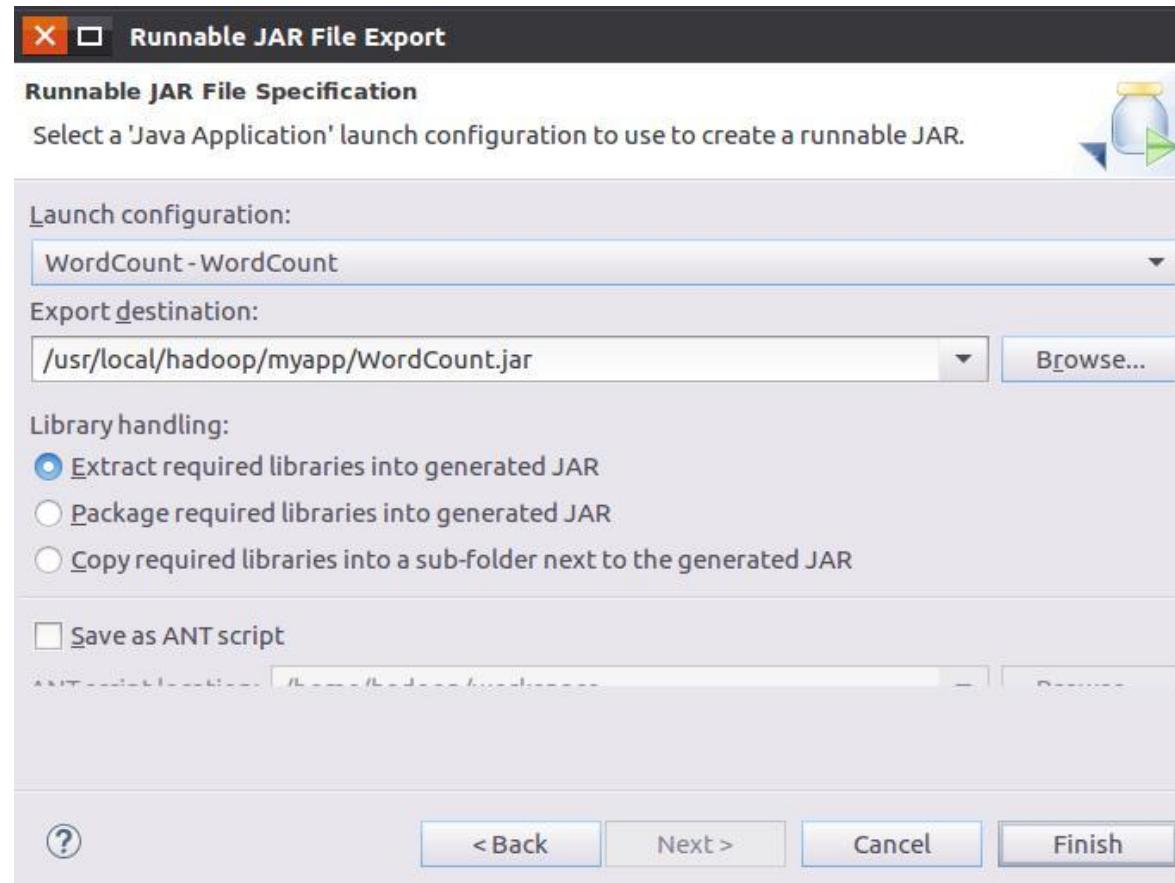
然后，会弹出如下图所示界面。





7.6.5 编译打包代码以及运行程序

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。

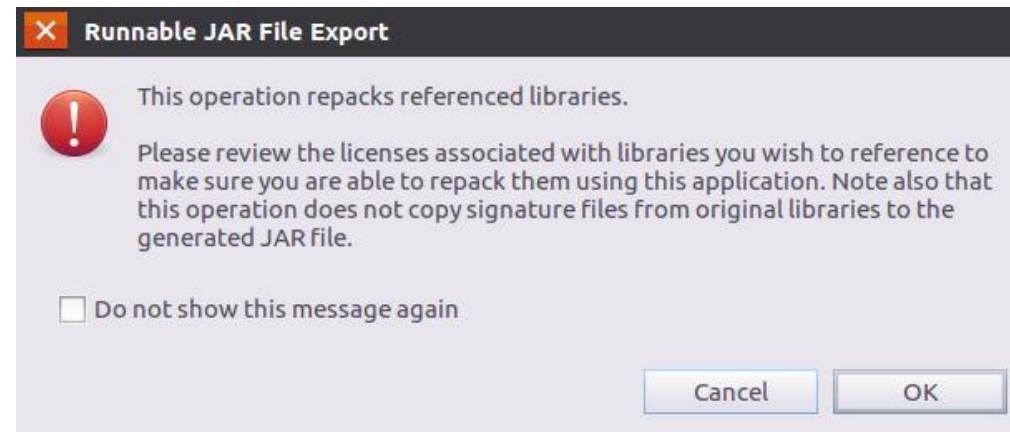




7.6.5 编译打包代码以及运行程序

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“WordCount-WordCount”。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

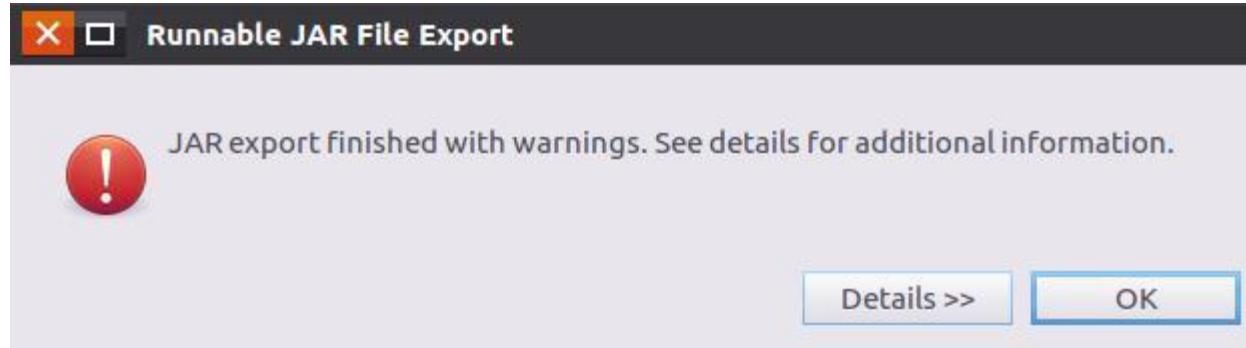
“/usr/local/hadoop/myapp/WordCount.jar”。在“Library handling”下面选择“Extract required libraries into generated JAR”。然后，点击“Finish”按钮，会出现如下图所示界面。





7.6.5 编译打包代码以及运行程序

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。





7.6.5 编译打包代码以及运行程序

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WordCount工程打包生成了WordCount.jar。可以到Linux系统中查看一下生成的WordCount.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WordCount.jar文件。



7.6.5 编译打包代码以及运行程序

5、运行程序

在运行程序之前，需要启动**Hadoop**，命令如下：

```
cd /usr/local/hadoop  
./sbin/start-dfs.sh
```

在启动**Hadoop**之后，需要首先删除**HDFS**中与当前**Linux**用户**hadoop**对应的**input**和**output**目录（即**HDFS**中的“/user/hadoop/input”和“/user/hadoop/output”目录），这样确保后面程序运行不会出现问题，具体命令如下：

```
cd /usr/local/hadoop  
./bin/dfs dfs -rm -r input  
./bin/dfs dfs -rm -r output
```



7.6.5 编译打包代码以及运行程序

然后，再在HDFS中新建与当前Linux用户hadoop对应的input目录，即“/user/hadoop/input”目录，具体命令如下：

```
cd /usr/local/hadoop  
./bin/dfs dfs -mkdir input
```

然后，把之前在Linux本地文件系统中新建的两个文件wordfile1.txt和wordfile2.txt（假设这两个文件位于“/usr/local/hadoop”目录下，并且里面包含了一些英文语句），上传到HDFS中的“/user/hadoop/input”目录下，命令如下：

```
cd /usr/local/hadoop  
./bin/dfs dfs -put ./wordfile1.txt input  
./bin/dfs dfs -put ./wordfile2.txt input
```



7.6.5 编译打包代码以及运行程序

如果HDFS中已经存在目录 “/user/hadoop/output”，
则使用如下命令删除该目录：

```
cd /usr/local/hadoop  
.bin/hdfs dfs -rm -r /user/hadoop/output
```

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/WordCount.jar input output
```



7.6.5 编译打包代码以及运行程序

上面命令执行以后，当运行顺利结束时，屏幕上会显示类似如下的信息：

.....//这里省略若干屏幕信息

2020-01-27 10:10:55,157 INFO mapreduce.Job: map 100% reduce 100%

2020-01-27 10:10:55,159 INFO mapreduce.Job: Job job_local457272252_0001 completed successfully

2020-01-27 10:10:55,174 INFO mapreduce.Job: Counters: 35
File System Counters FILE: Number of bytes read=115463648 FILE:
Number of bytes written=117867638 FILE: Number of read operations=0
FILE: Number of large read operations=0 FILE: Number of write
operations=0 HDFS: Number of bytes read=283 HDFS: Number of bytes
written=40



7.6.5 编译打包代码以及运行程序

词频统计结果已经被写入了HDFS的“/user/hadoop/output”目录中，可以执行如下命令查看词频统计结果：

```
cd /usr/local/hadoop  
./bin/hdfs dfs -cat output/*
```

上面命令执行后，会在屏幕上显示如下词频统计结果：

```
Hadoop 2  
I 2  
Spark 2  
fast 1  
good 1  
is 2  
love 2
```



本章小结

- 本章介绍了**MapReduce**编程模型的相关知识。**MapReduce**将复杂的、运行于大规模集群上的并行计算过程高度地抽象到了两个函数：**Map**和**Reduce**，并极大地方便了分布式编程工作，编程人员在不会分布式并行编程的情况下，也可以很容易将自己的程序运行在分布式系统上，完成海量数据集的计算
- **MapReduce**执行的全过程包括以下几个主要阶段：从分布式文件系统读入数据、执行**Map**任务输出中间结果、通过**Shuffle**阶段把中间结果分区排序整理后发送给**Reduce**任务、执行**Reduce**任务得到最终结果并写入分布式文件系统。在这几个阶段中，**Shuffle**阶段非常关键，必须深刻理解这个阶段的详细执行过程
- **MapReduce**具有广泛的应用，比如关系代数运算、分组与聚合运算、矩阵-向量乘法、矩阵乘法等
- 本章最后以一个单词统计程序为实例，详细演示了如何编写**MapReduce**程序代码以及如何运行程序



Thank You!