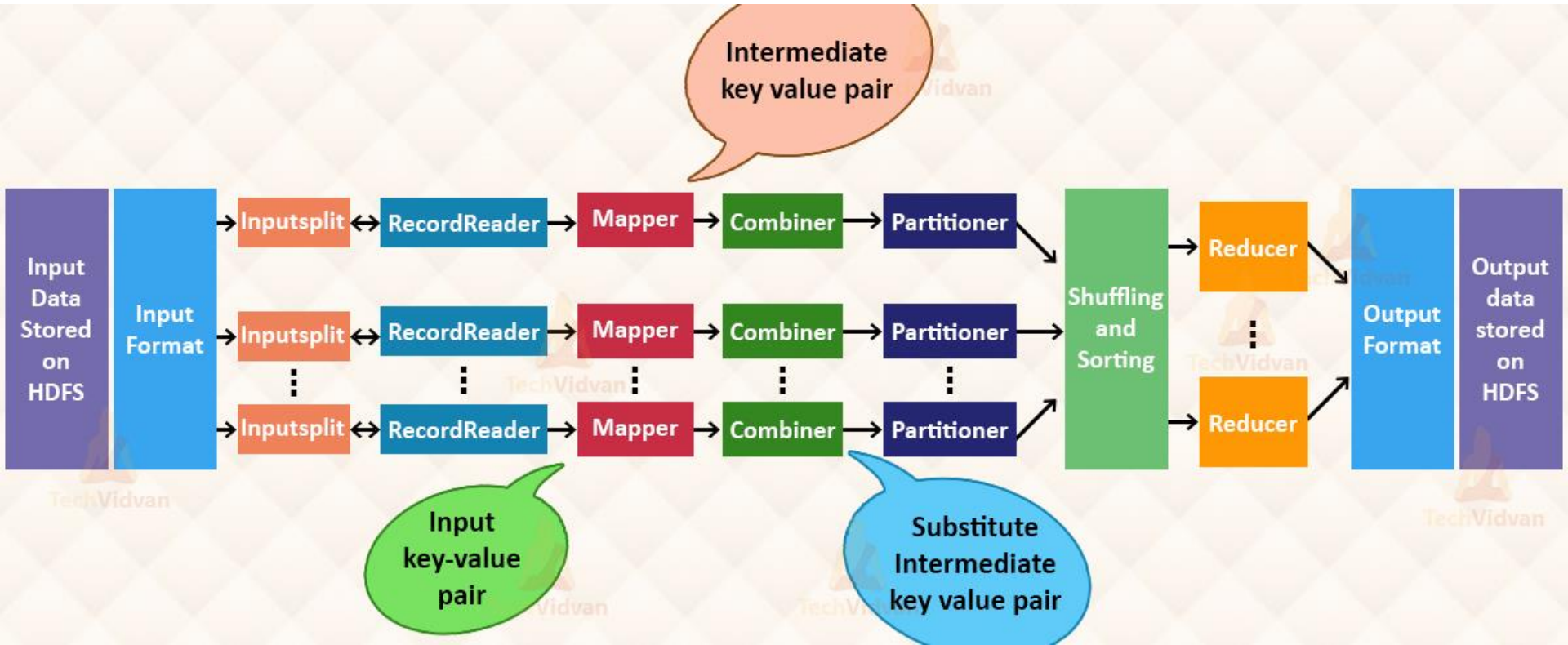
Chap. 7 MapReduce

王怡洋 副教授

大连海事大学 信息科学技术学院



内容提纲

本PPT是基于如下教材的配套讲义：

《大数据技术原理与应用——概念、存储、处理、分析与应用》（2017年2月第2版） 林子雨 编著，人民邮电出版社

7.1 概述

- 7.1.1 分布式并行编程
- 7.1.2 MapReduce模型简介
- 7.1.3 Map和Reduce函数

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

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
- 编程容易，不需要掌握分布式并行编程细节，也可以很容易把自己的程序运行在分布式系统上，完成海量数据的计算
- MapReduce采用“**分而治之**”策略，一个存储在分布式文件系统中的大规模数据集，会被切分成许多独立的分片（split），这些分片可以被多个Map任务并行处理
- MapReduce设计的一个理念就是“**计算向数据靠拢**”，而不是“数据向计算靠拢”，因为，移动数据需要大量的网络传输开销
- MapReduce框架采用了Master/Slave架构，包括一个Master和若干个Slave。Master上运行JobTracker，Slave上运行TaskTracker
- Hadoop框架是用Java实现的，但是，MapReduce应用程序则不一定要用Java来写

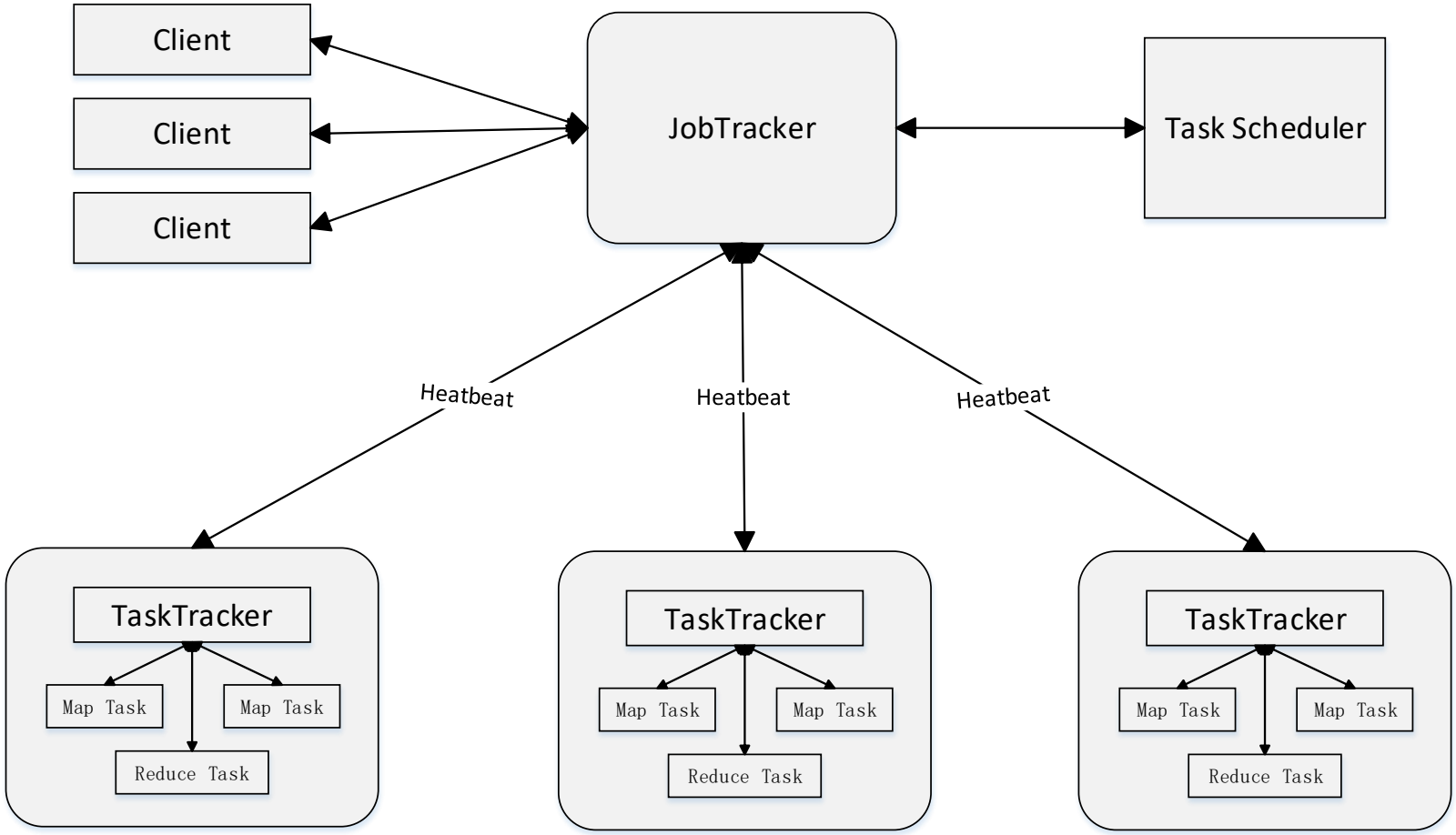
7.1.3 Map和Reduce函数

表7-1 Map和Reduce

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

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

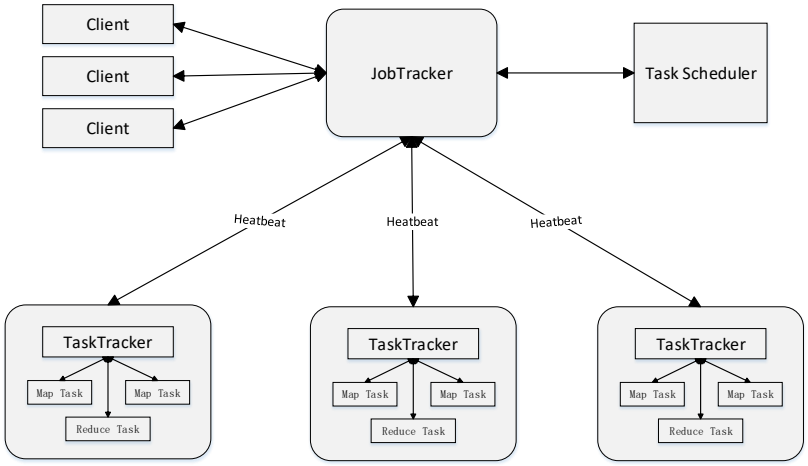
MapReduce主要有以下4个部分组成：

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

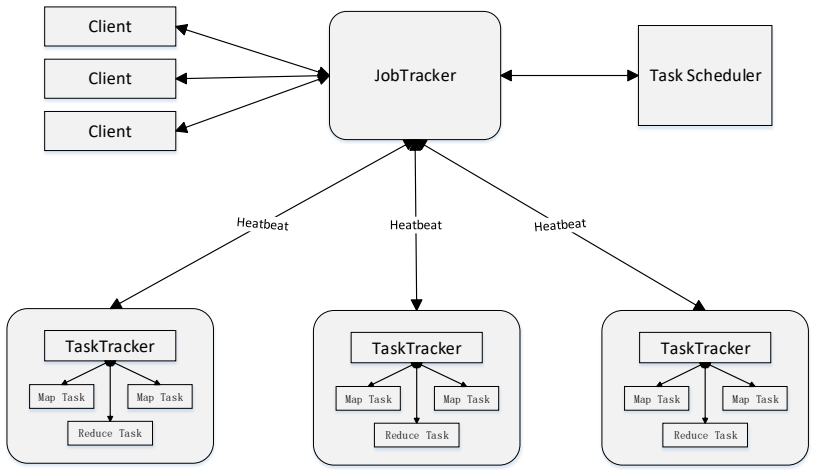
3）TaskTracker

•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 工作流程概述

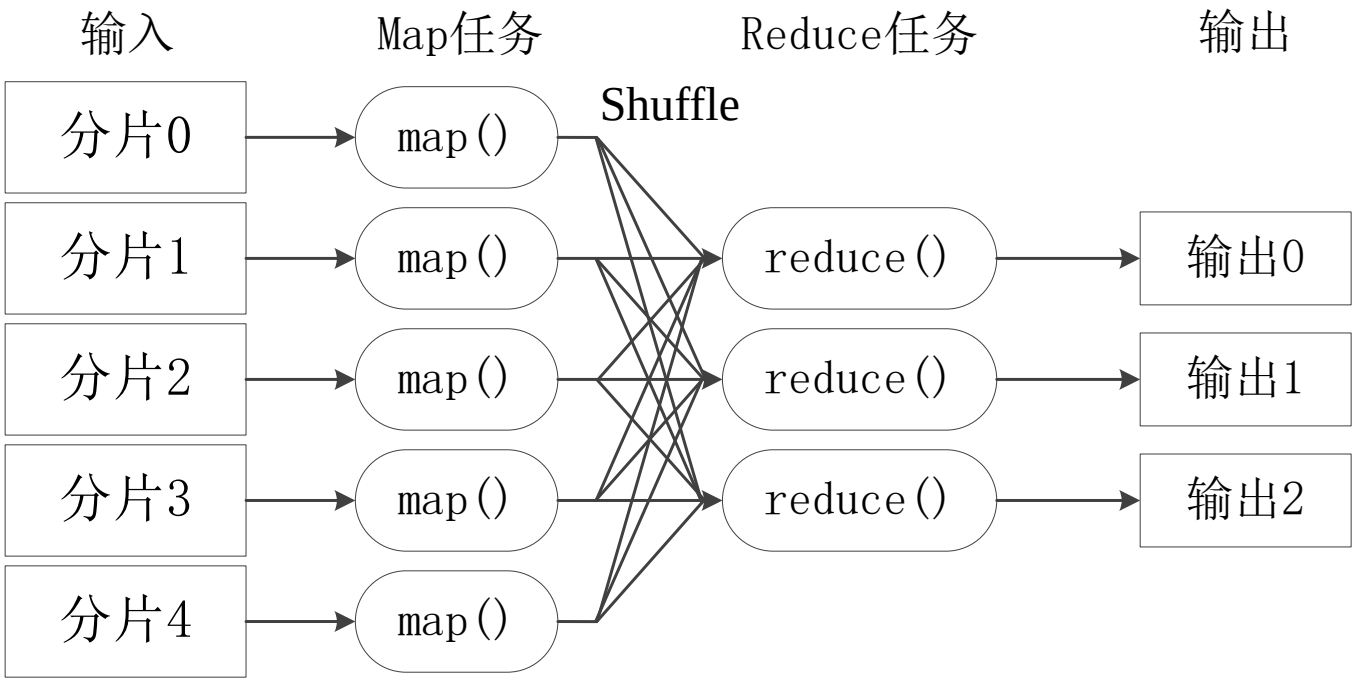
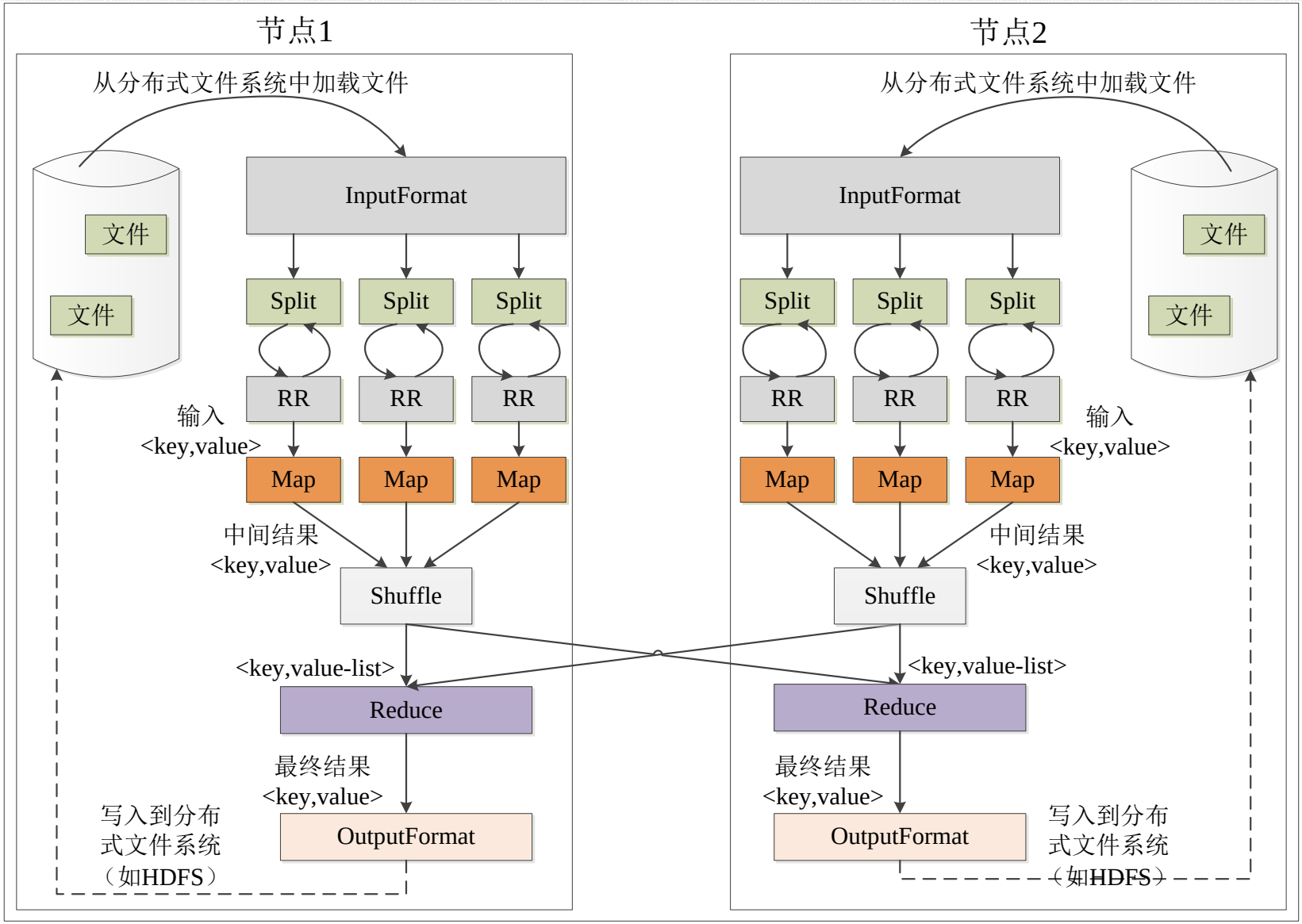


图7-1 MapReduce工作流程

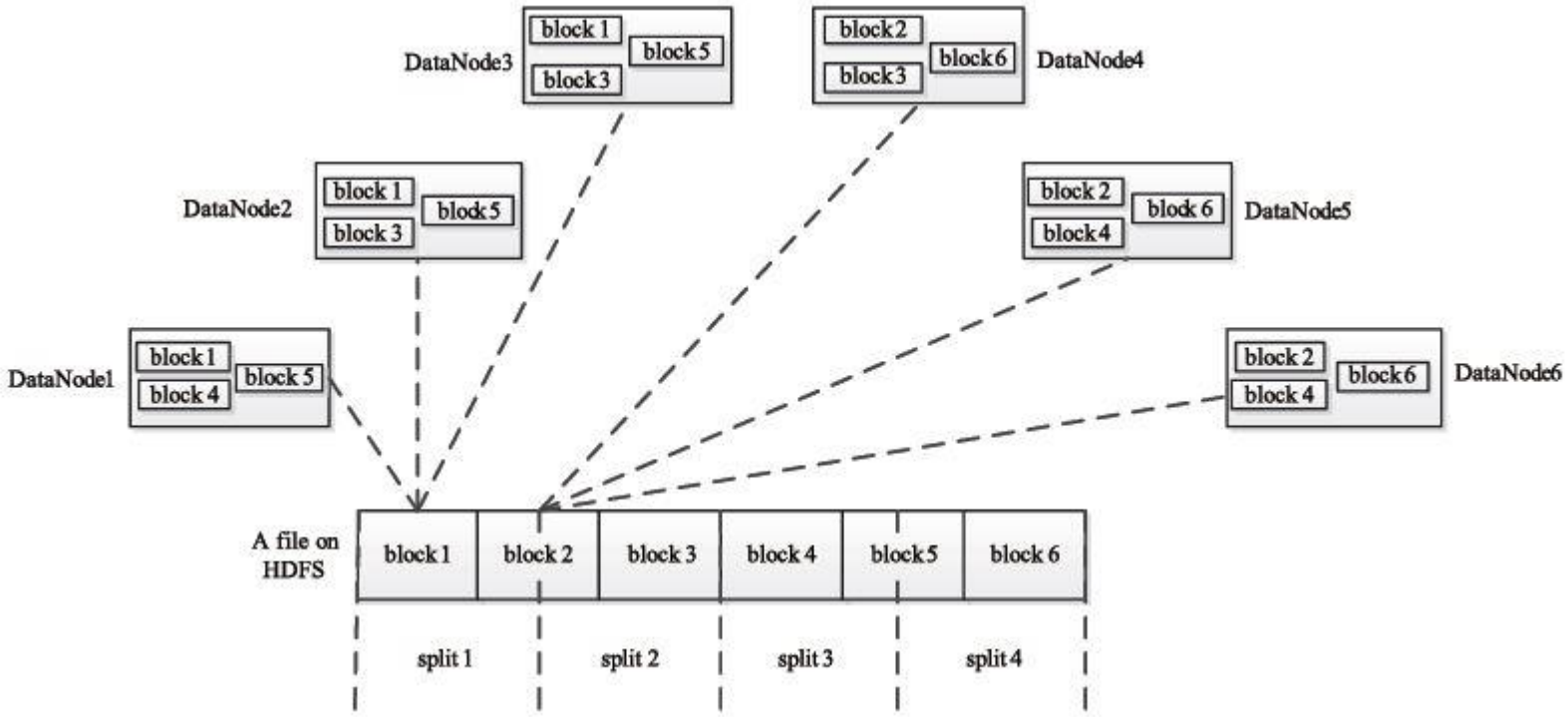
- 不同的Map任务之间不会进行通信
- 不同的Reduce任务之间也不会发生任何信息交换
- 用户不能显式地从一台机器向另一台机器发送消息
- 所有的数据交换都是通过MapReduce框架自身去实现的

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）

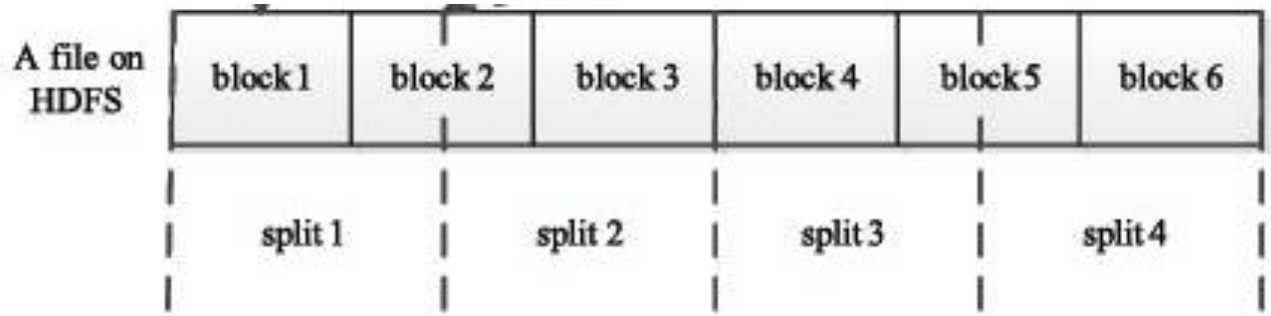


HDFS 以固定大小的block 为基本单位存储数据，而对于MapReduce 而言，其处理单位是split。split 是一个逻辑概念，它只包含一些元数据信息，比如数据起始位置、数据长度、数据所在节点等。它的划分方法完全由用户自己决定。

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

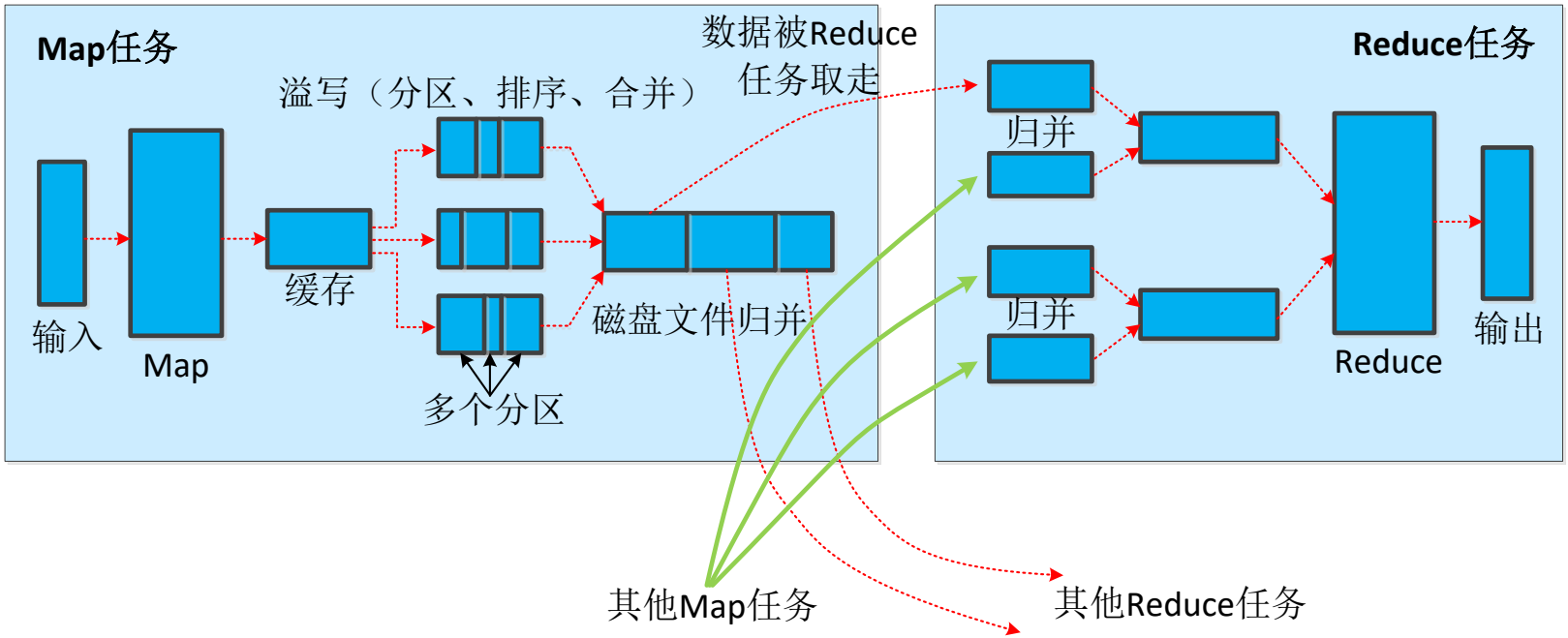
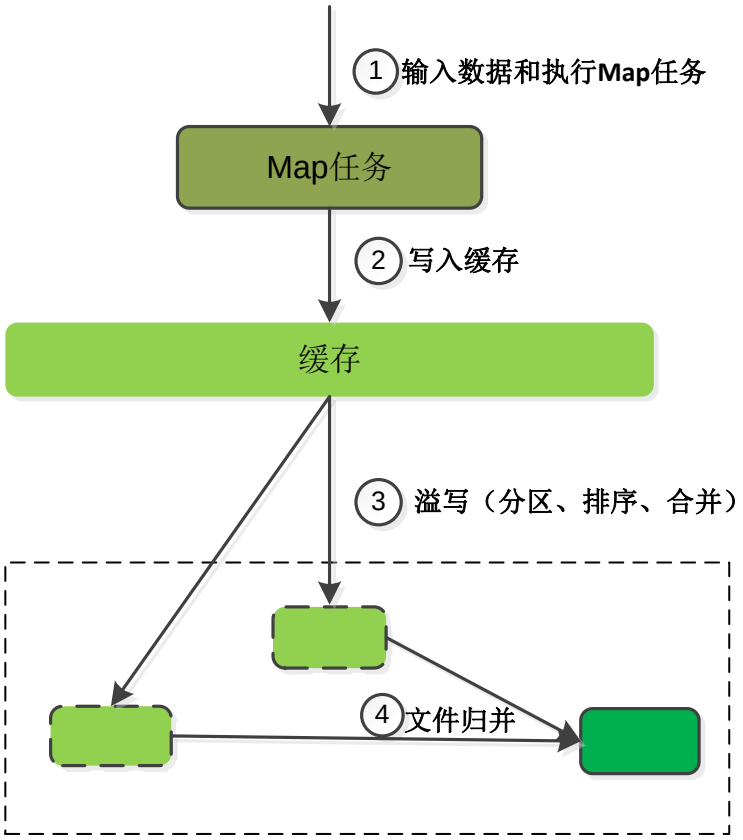


图7-3 Shuffle过程

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

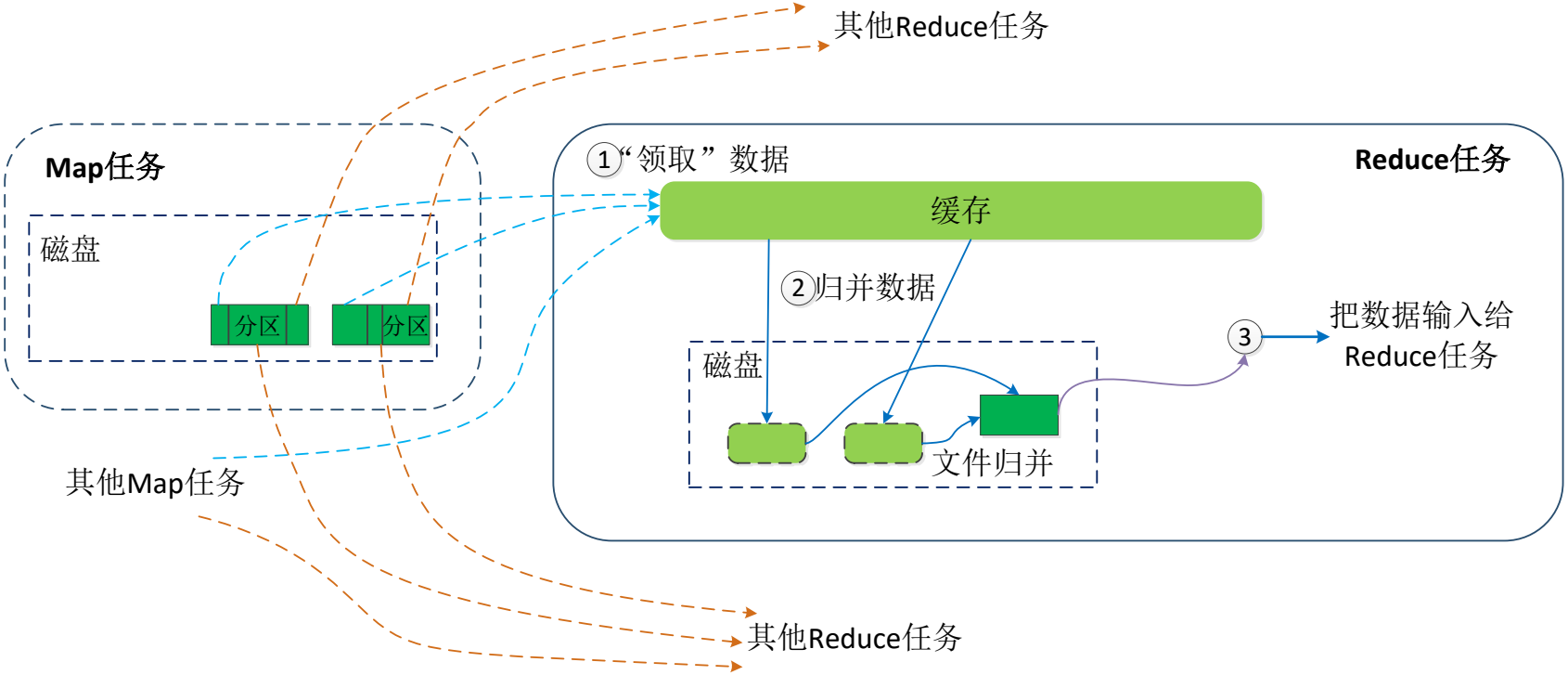
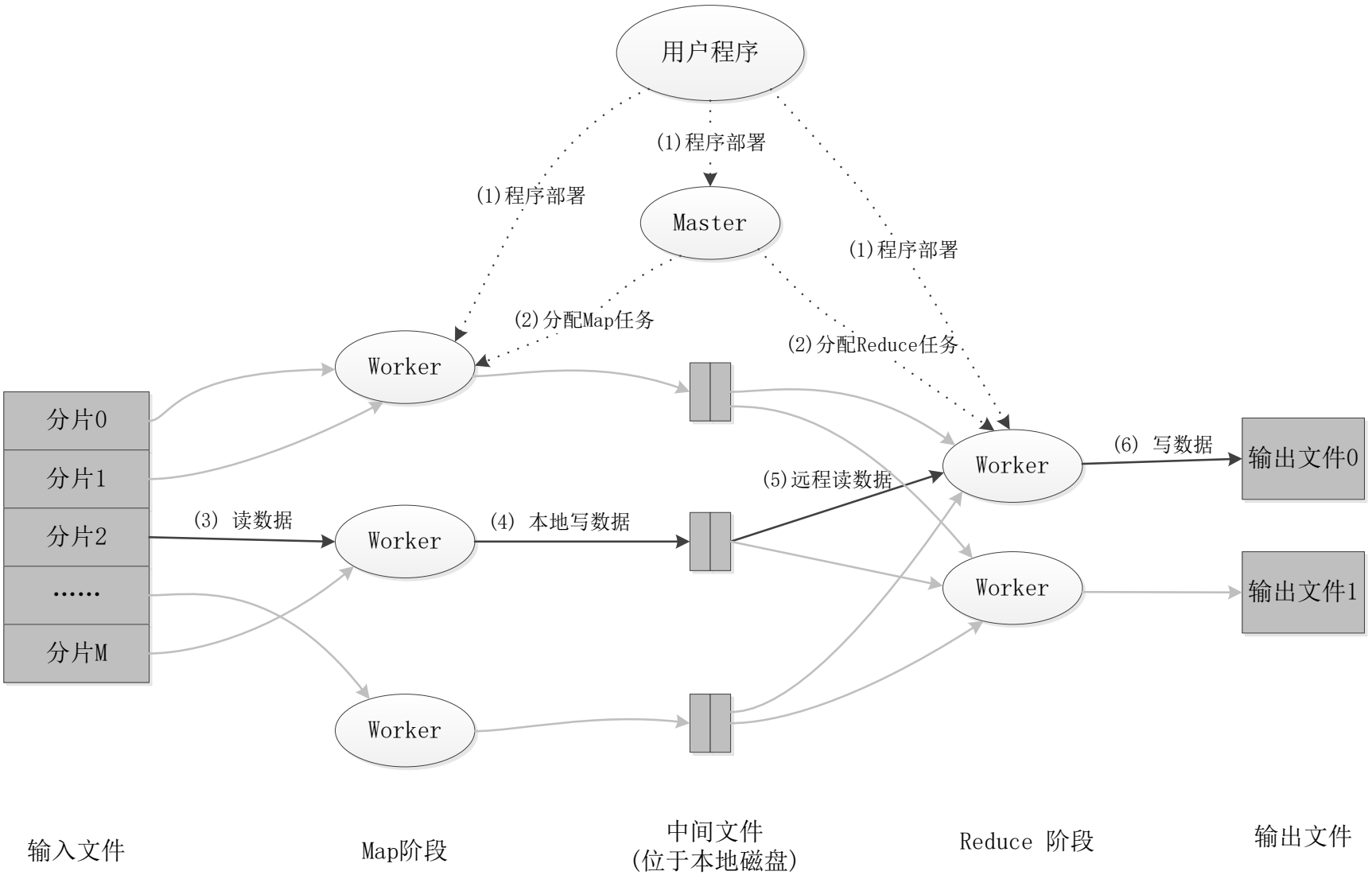


图7-5 Reduce端的Shuffle过程

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 Hello Hadoop Hello MapReduce	Hadoop 1 Hello 3 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

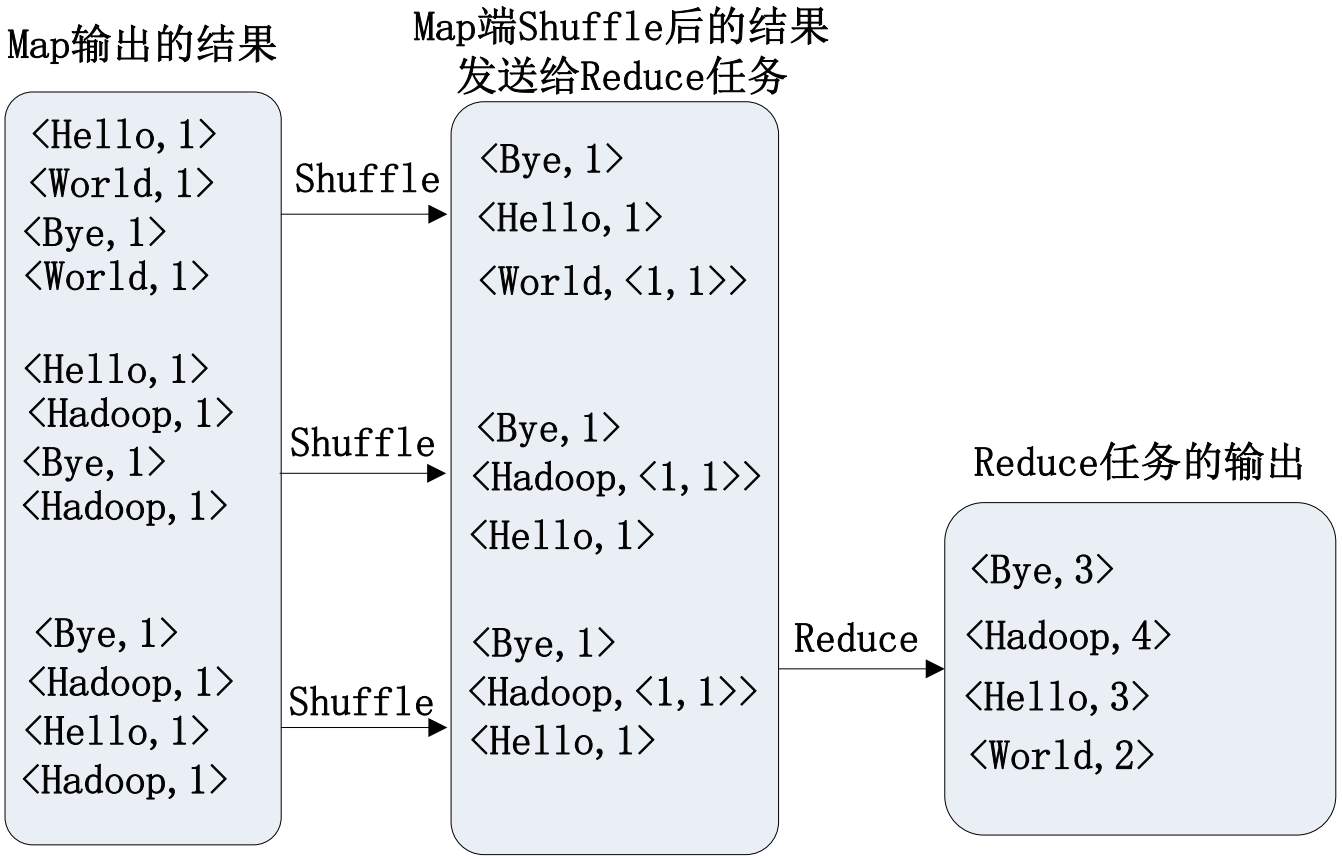


图7-8 用户没有定义Combiner时的Reduce过程示意图

7.4.3 一个WordCount执行过程的实例

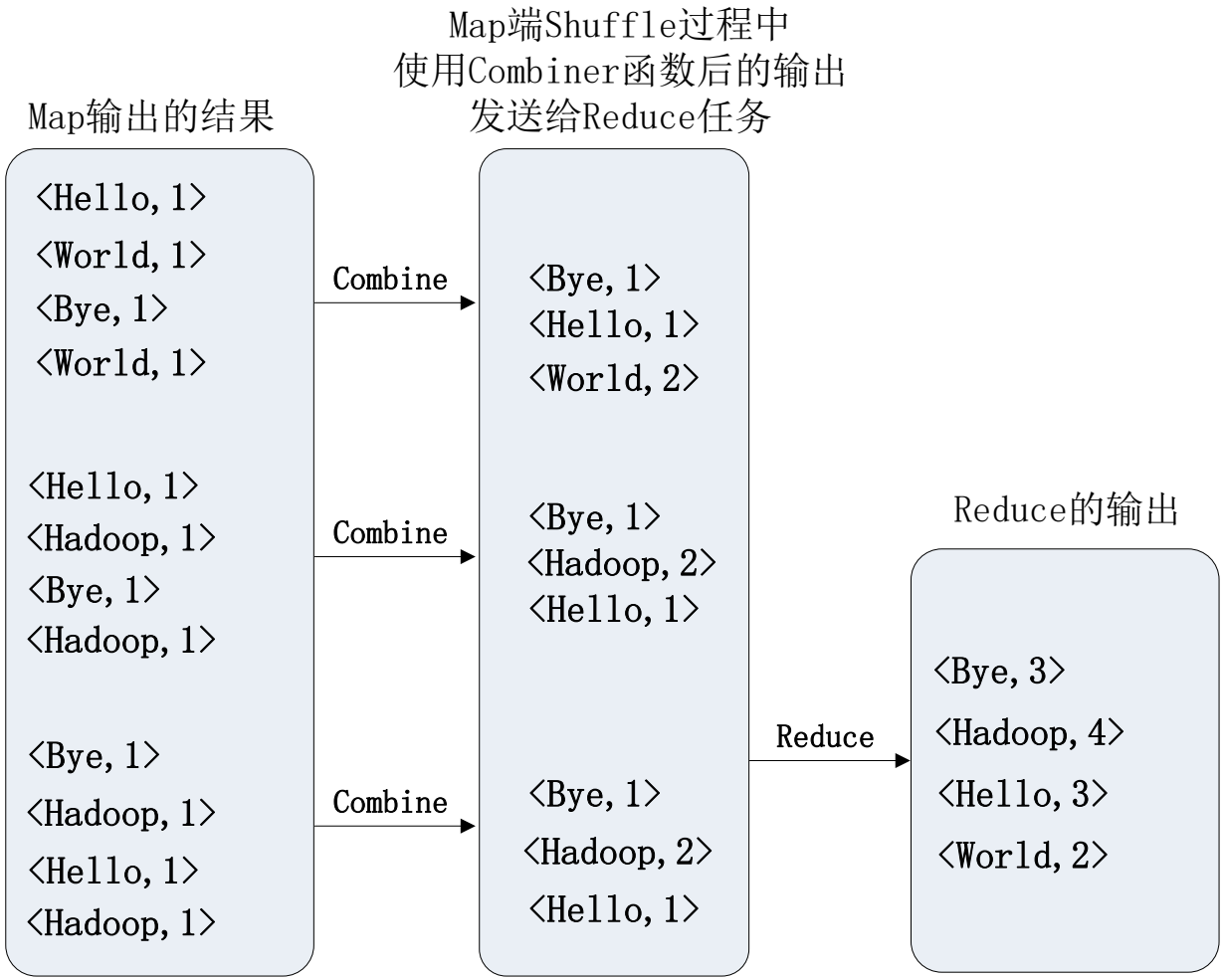


图7-9 用户有定义Combiner时的Reduce过程示意图

7.5MapReduce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

7.5MapReduce的具体应用

用MapReduce实现关系的自然连接

雇员

Name	EmpId	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	EmpId	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
1	“Item”,(10,1)
1	“Item”,(20,3)
2	“Item”,(10,5)
2	“Item”,(50,100)
3	“Item”,(20,1)

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)

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
- 7.6.6 Hadoop中执行MapReduce任务的几种方式

详细编程实践指南请参考：
《大数据原理与应用 第七章 MapReduce 学习指南》，
访问地址：http://dblab.xmu.edu.cn/blog/631-2/

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 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
- 7.6.6 Hadoop中执行MapReduce任务的几种方式

详细编程实践指南请参考：
《大数据原理与应用 第七章 MapReduce 学习指南》，
访问地址：http://dblab.xmu.edu.cn/blog/631-2/

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 2.x 版本中的依赖 jar

Hadoop 2.x 版本中 jar 不再集中在一个 hadoop-core*.jar 中，而是分成多个 jar，如使用 Hadoop 2.6.0 运行 WordCount 实例至少需要如下三个 jar:

- $HADOOP_HOME/share/hadoop/common/hadoop-common-2.6.0.jar
- $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.6.0.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
```