北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY

分布式并行编程框架 MapReduce

大数据处理技术

计算机学院

Hadoop is Google's Tech in Open Source

SOSP 2003

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

OSDI 2004

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

OSDI 2006

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

课程提纲

- MapReduce编程思想
- MapReduce体系结构
- MapReduce工作流程
- 实例分析：WordCount
- MapReduce的具体应用
- MapReduce编程实践

分布式并行编程

- CPU摩尔定律：当价格不变时，集成电路上可容纳的元器件的数目，约每隔18-24个月便会增加一倍，性能也将提升一倍
- 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

分布式并行编程

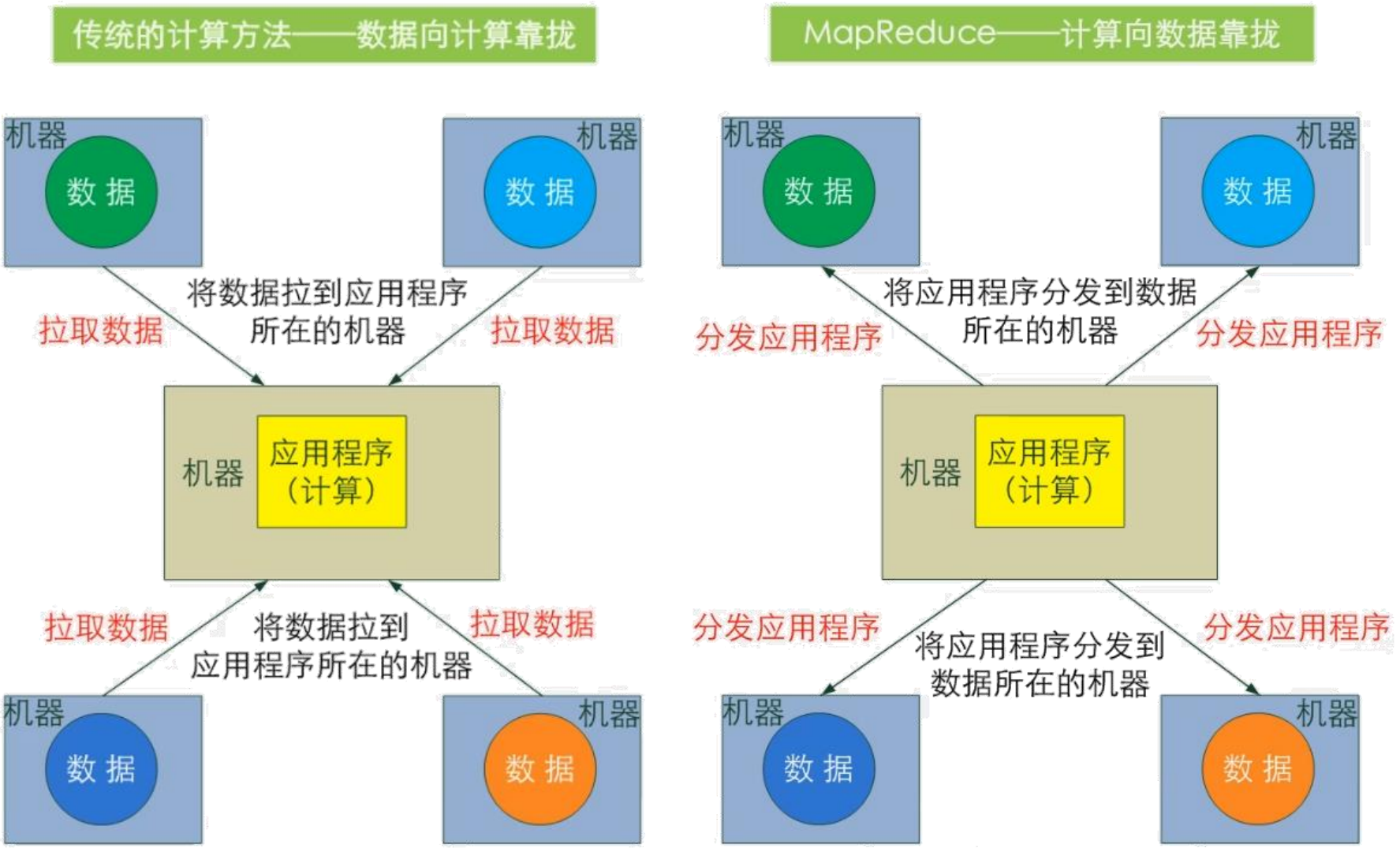
- 谷歌公司最先提出了分布式并行编程模型MapReduce，Hadoop MapReduce是它的开源实现
- 在MapReduce出现之前，已经有像MPI这样非常成熟的并行计算框架了，那么MapReduce相较于传统的并行计算框架有什么优势？

	集群的架构和容错性	硬件价格及扩展性	编程和学习难度	适用场景
传统并行编程框架	①通常采用**共享式架构**（共享内存、共享存储），底层通常采用统一的存储区域网络SAN ②**容错性差**，其中一个硬件发生故障容易导致整个集群不可工作	通常采用刀片服务器，高速网络以及共享存储区域网络SAN，**价格高**，**扩展性差**	**编程难度大**，需要解决**what-how**问题，编程原理和多线程编程逻辑类似，需要借助互斥量、信号量、锁等机制，实现不同任务之间的同步和通信	适用于实时、细粒度计算，尤其适用于**计算密集型**的应用
Map-Reduce	①采用典型的**非共享式架构** ②**容错性好**，在整个集群中每个节点都有自己的内存和存储，任何一个节点出现问题不会影响其他节点正常运行，同时系统中设计了冗余和容错机制	整个集群可以随意增加或减少相关的计算节点，不需要高端服务器，普通PC机也可以，**价格低廉**，**扩展性好**	**编程简单**，只需要告诉系统要做什么即解决**what**问题，系统自动实现分布式部署，屏蔽分布式同步、通信、、负载均衡、失败恢复等底层细节	一般适用于非实时的批处理及**数据密集型**应用

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模型简介



MapReduce模型简介

- 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

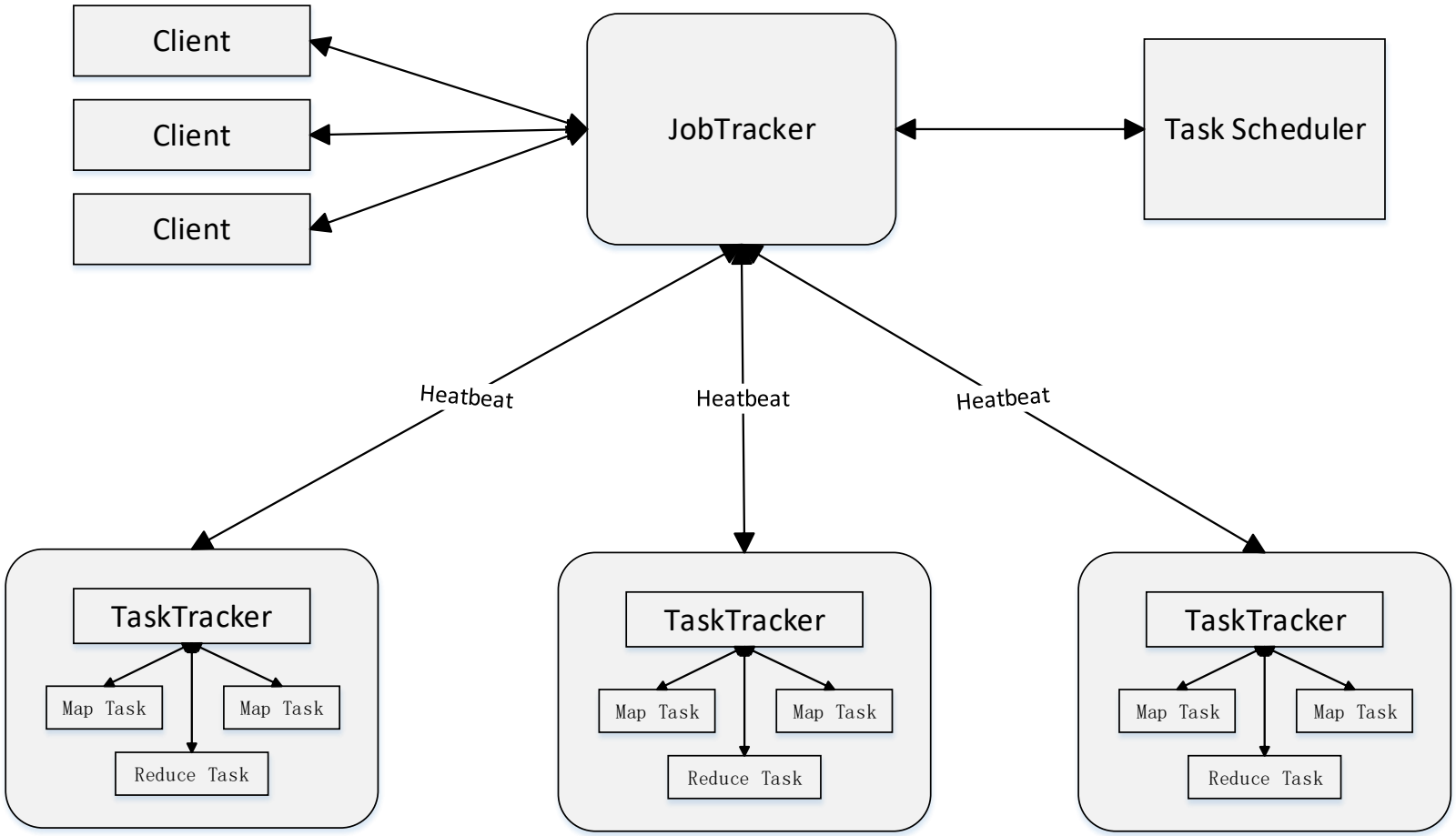
Map和Reduce函数

- Map函数和Reduce函数都是以<key, value>作为输入，按一定的映射规则转换成另一个或一批<key, value>进行输出

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（split）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> 如： <"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

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



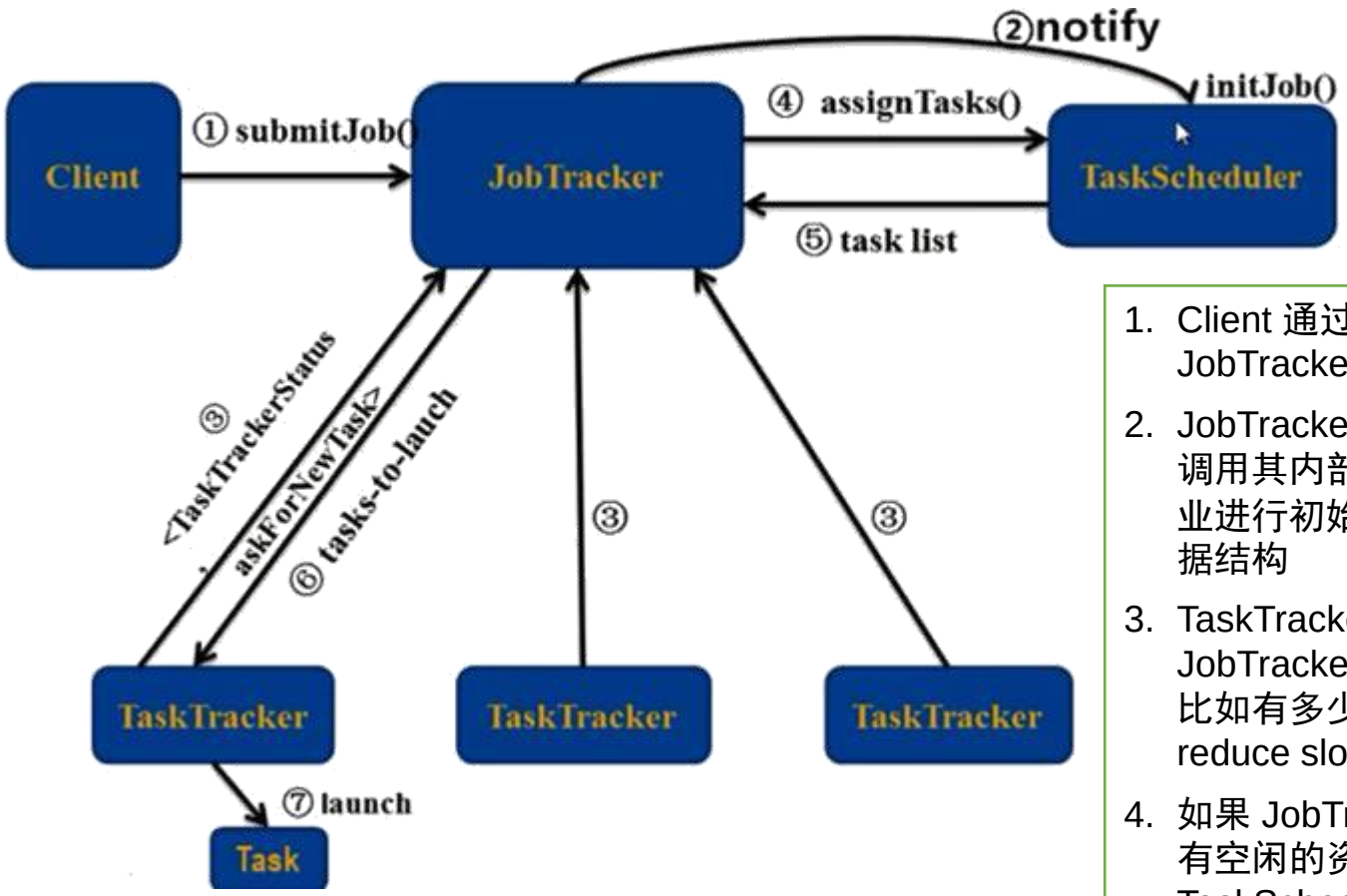
MapReduce的体系结构

- 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
- JobTracker
  - 负责资源监控和作业调度
  - 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
  - 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
  - 调度器是一个可插拔的模块，用户可以根据自己的实际应用要求设计调度器

MapReduce的体系结构

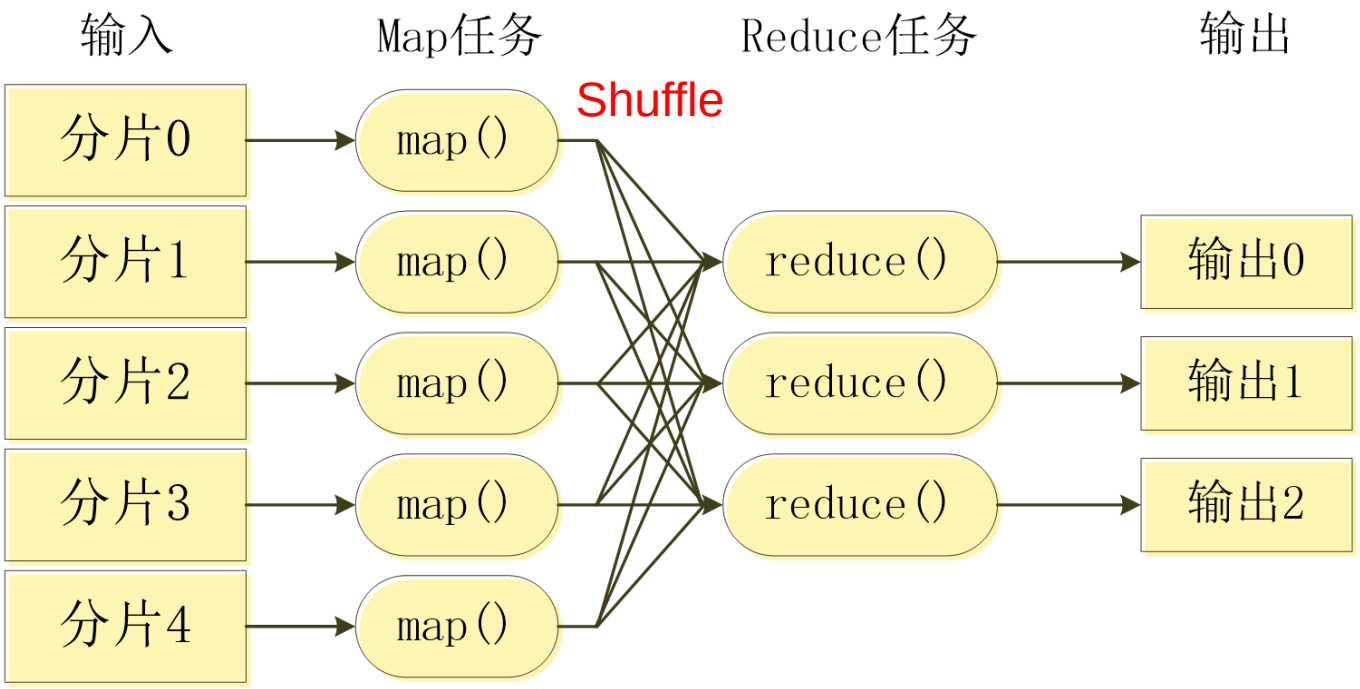
- TaskTracker
  - 接收JobTracker 发送过来的命令并执行相应的操作（如启动新任务、杀死任务等）
  - TaskTracker 会周期性地通过“心跳”将本节点上资源的使用情况和任务的运行进度汇报给JobTracker
  - TaskTracker 使用“slot”等量划分本节点上的资源量（CPU、内存等）。一个Task 获取到一个slot 后才有机会运行，而Hadoop调度器的作用就是将各个TaskTracker上的空闲slot分配给Task使用。slot 分为Map slot 和Reduce slot 两种，分别供MapTask 和Reduce Task 使用
-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

MapReduce的体系结构



1. Client 通过submitJob()函数向JobTracker提交一个作业
2. JobTracker通知 TaskScheduler，调用其内部函数initJob()对这个作业进行初始化，创建一些内部的数据结构
3. TaskTracker 通过心跳来向JobTracker 汇报它的资源情况，比如有多少个空闲的map slot和reduce slot
4. 如果 JobTracker 发现TaskTracker有空闲的资源 ，就会调用TaskScheduler 的 assignTasks()函数，返回task list给第一个TaskTracker，这时TaskTracker就会执行调度器分配的任务

MapReduce工作流程

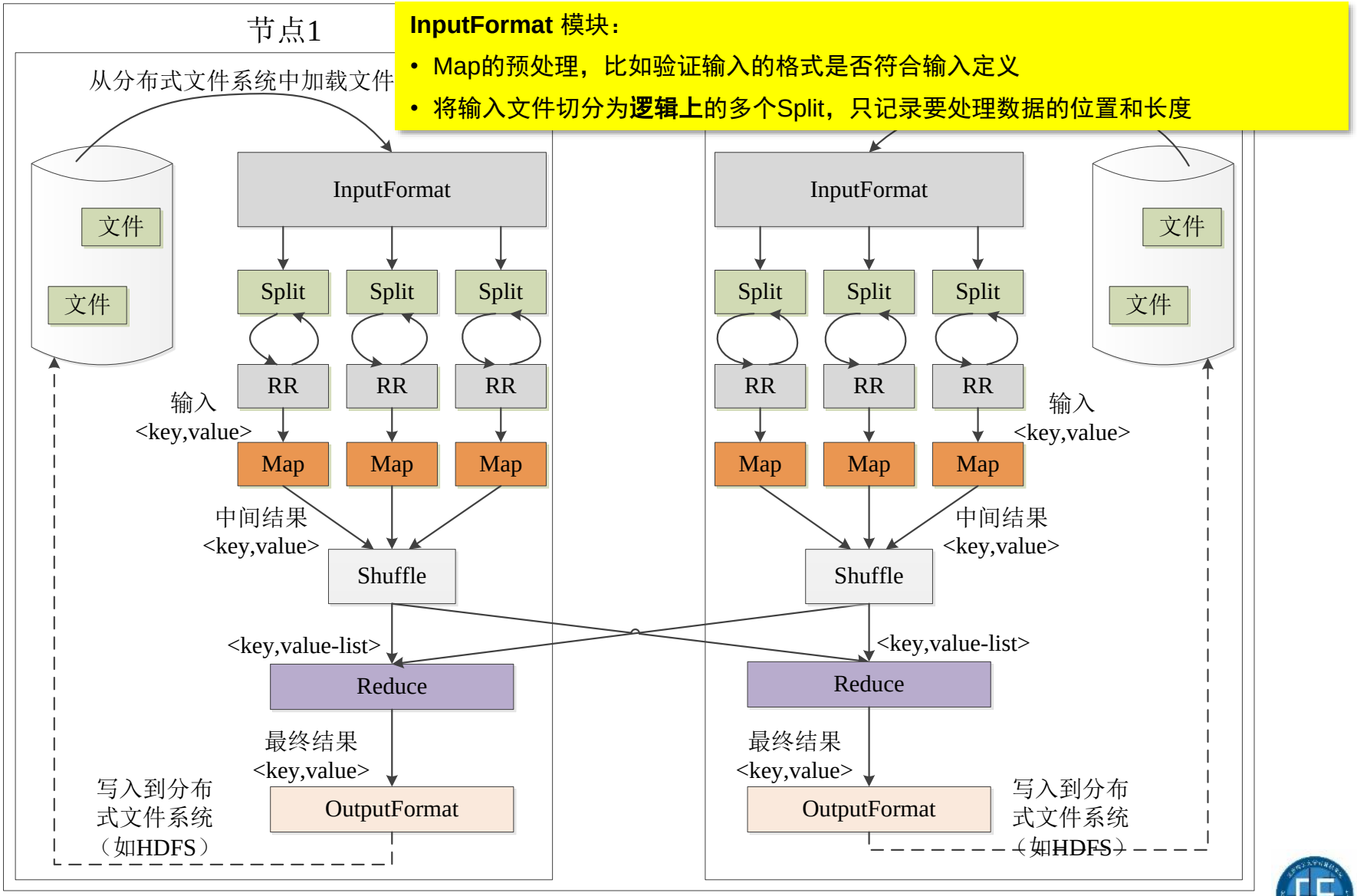


- 不同的Map任务之间不会进行通信
- 不同的Reduce任务之间也不会发生任何信息交换
- 用户不能显式地从一台机器向另一台机器发送消息
- 所有的数据交换都是通过MapReduce框架自身去实现的

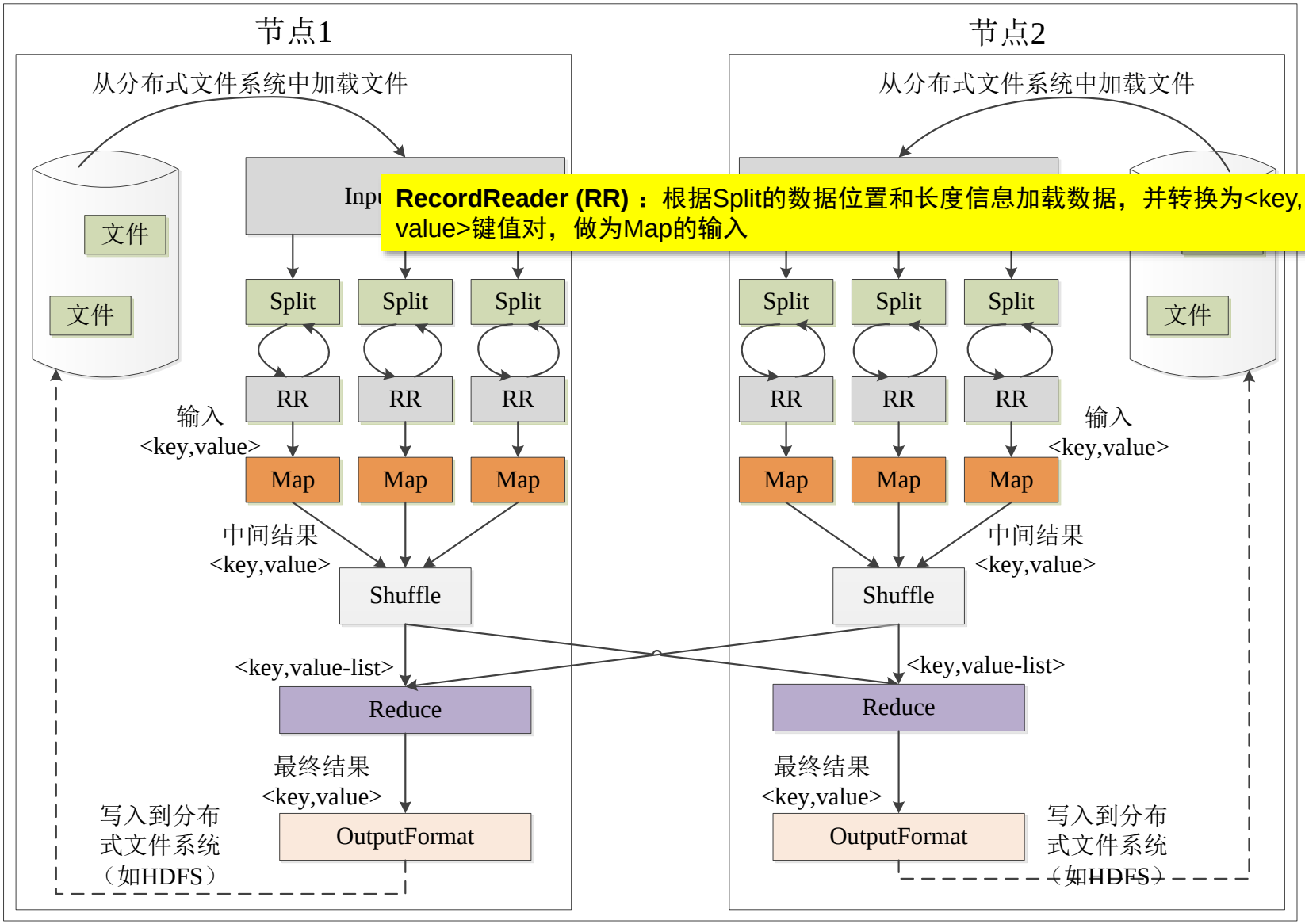
MapReduce各个执行阶段

InputFormat 模块：

- Map的预处理，比如验证输入的格式是否符合输入定义
- 将输入文件切分为**逻辑上**的多个Split，只记录要处理数据的位置和长度

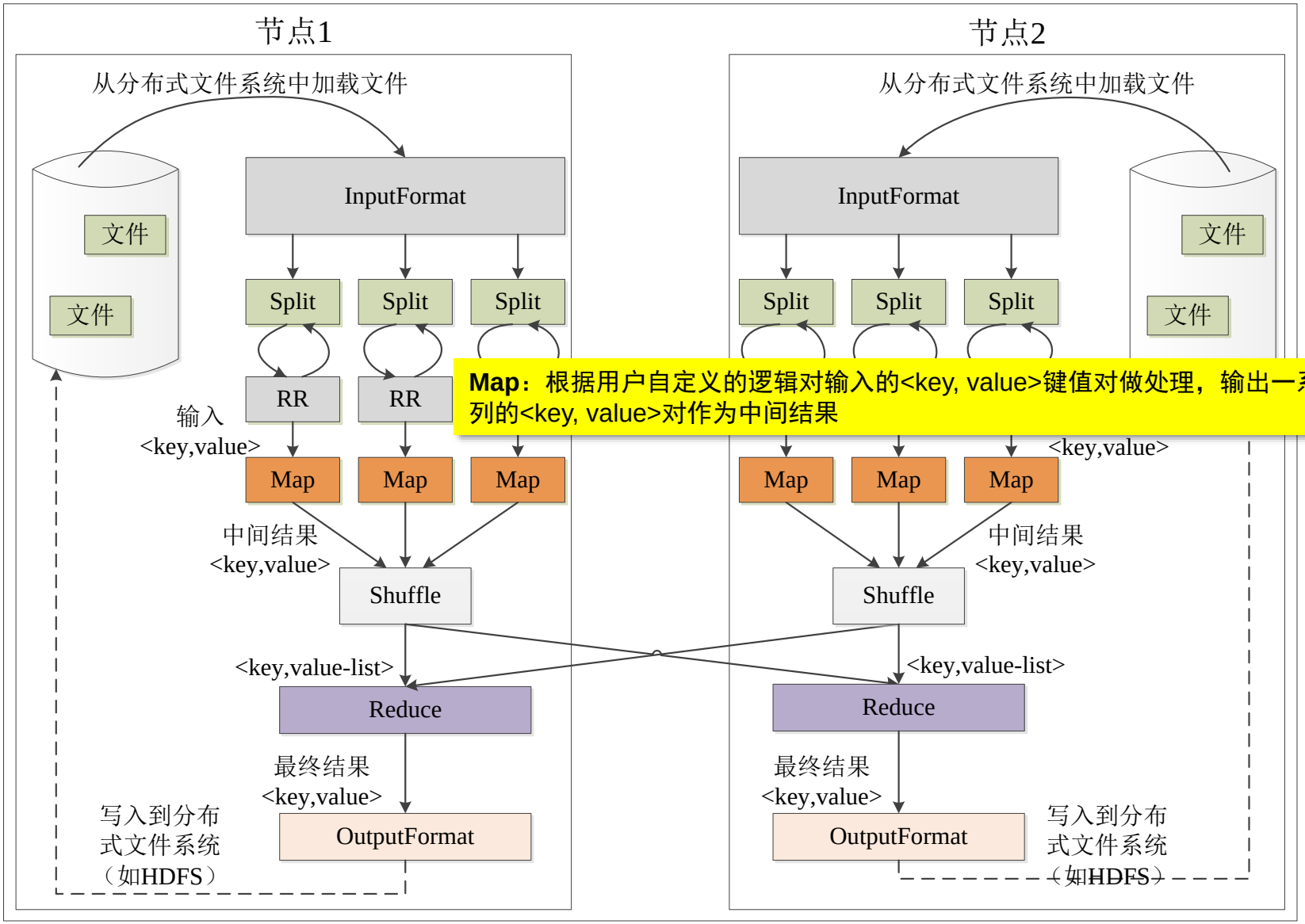


MapReduce各个执行阶段

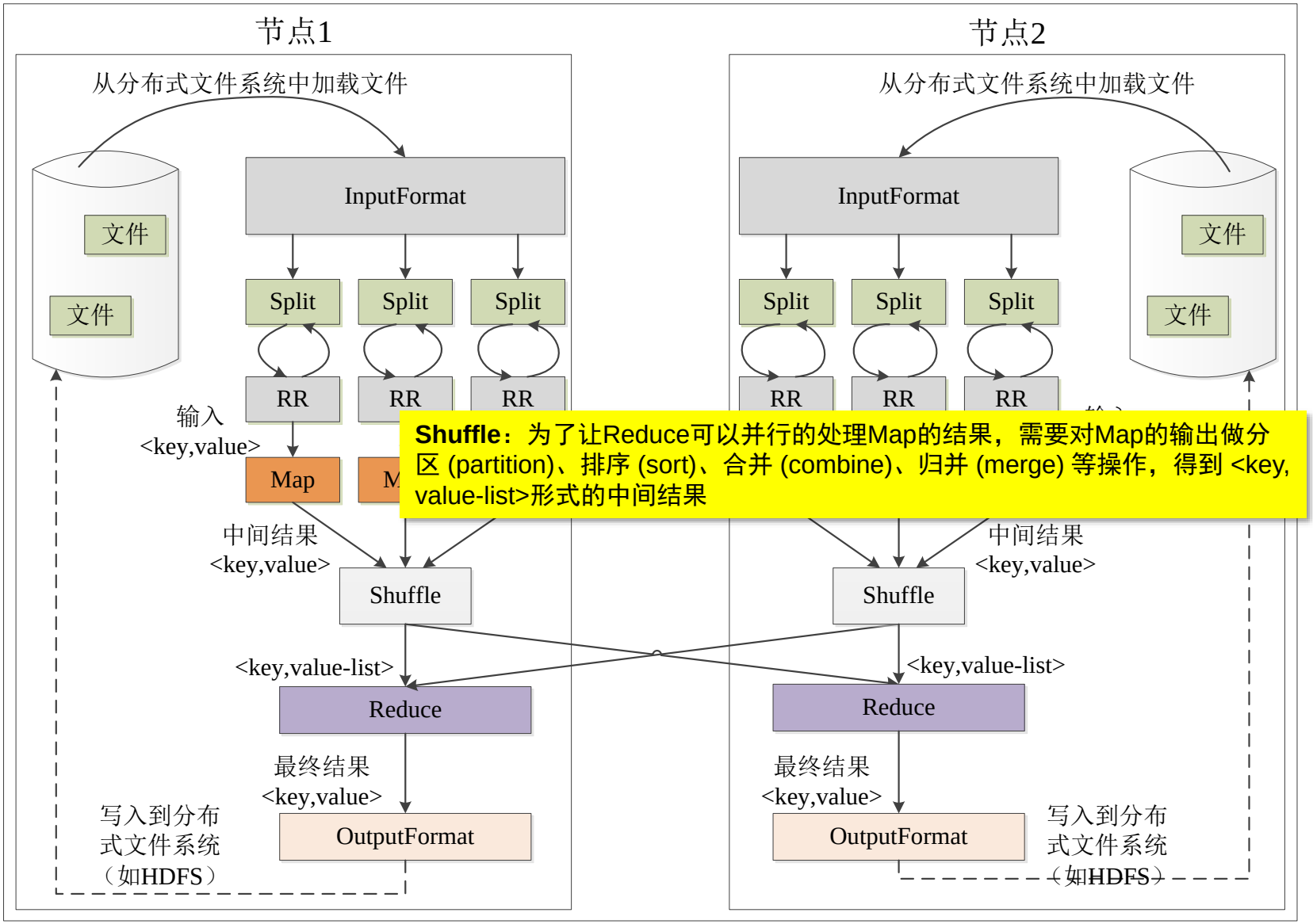


RecordReader (RR)：根据Split的数据位置和长度信息加载数据，并转换为<key, value>键值对，做为Map的输入

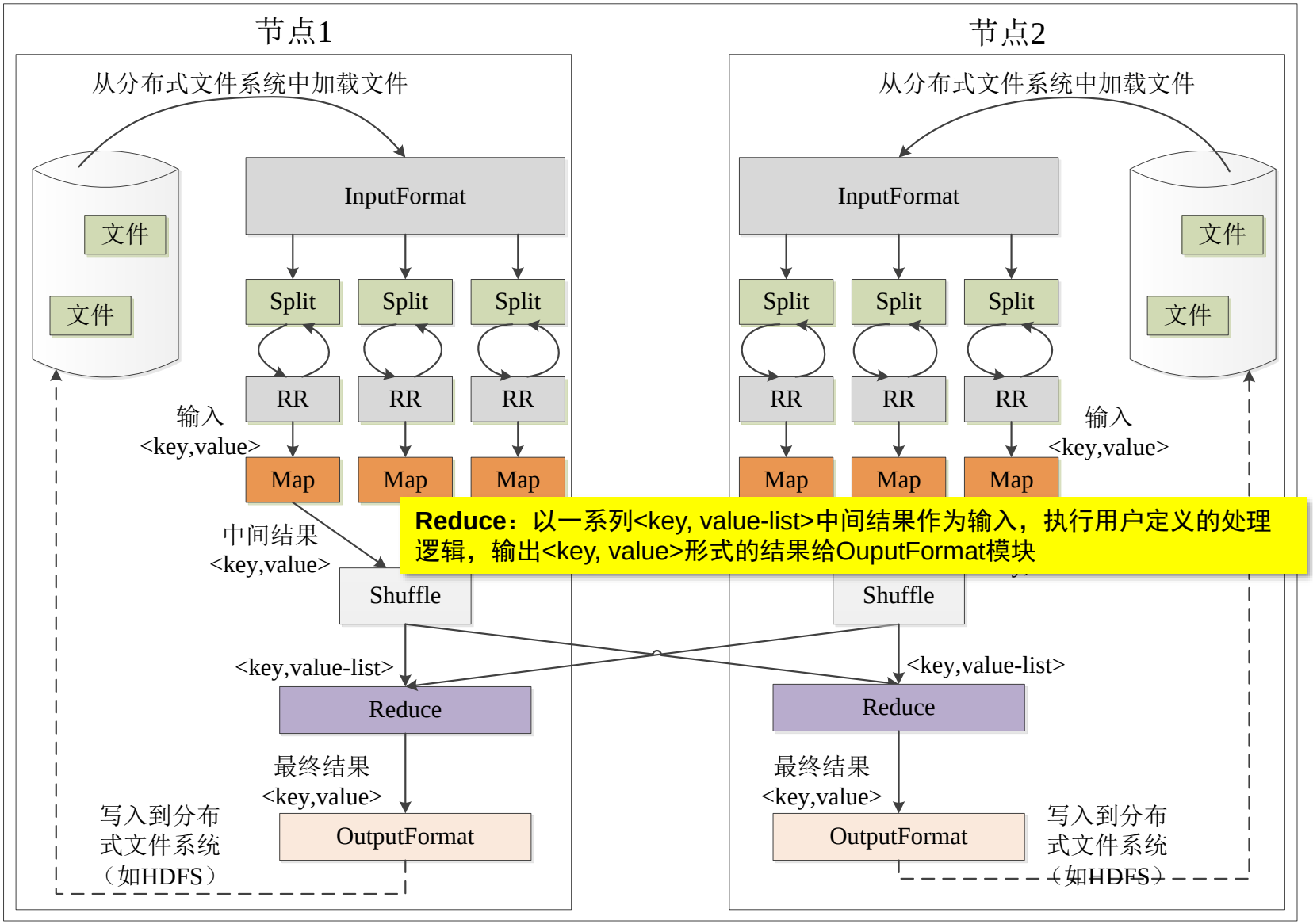
MapReduce各个执行阶段



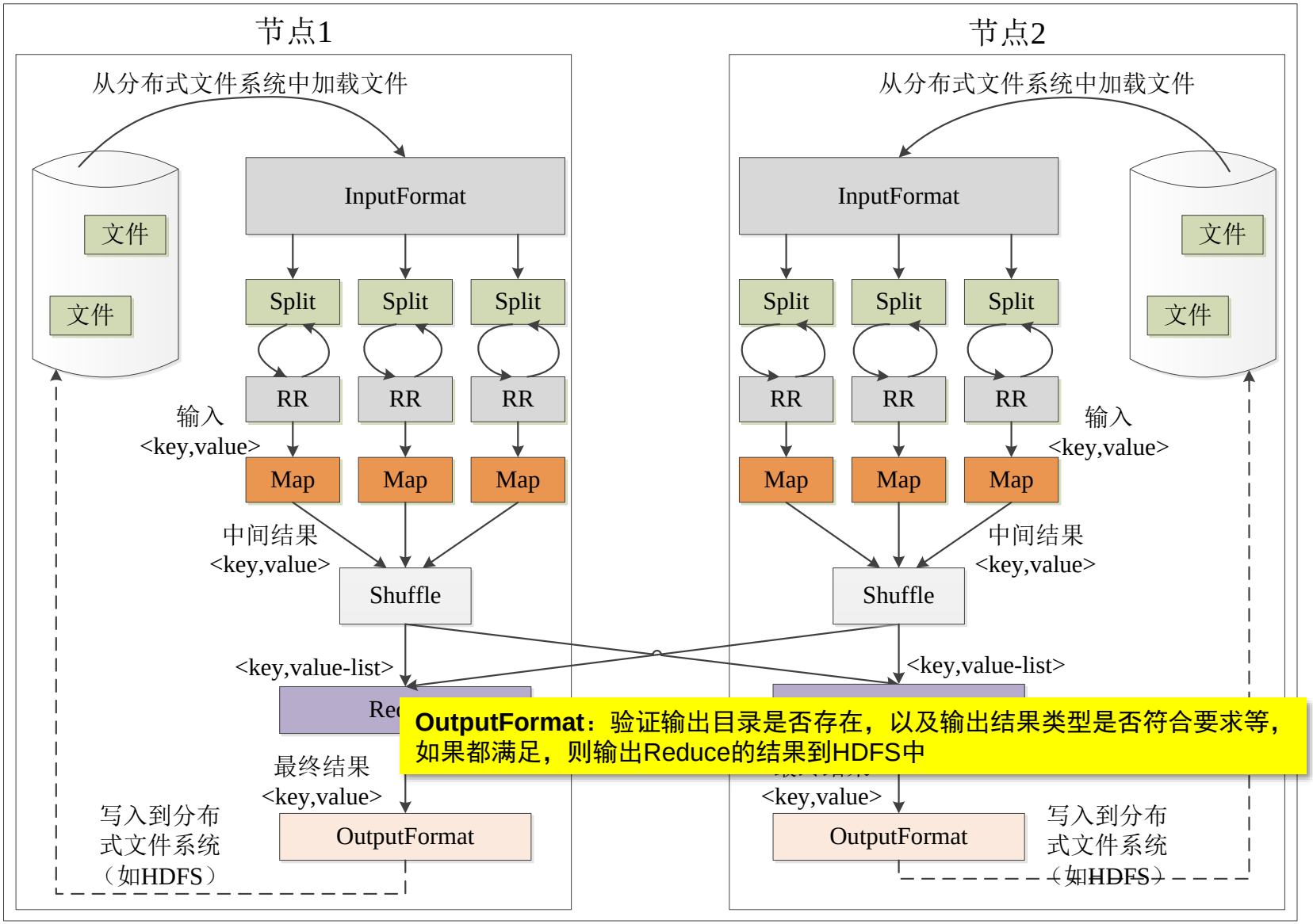
MapReduce各个执行阶段



MapReduce各个执行阶段



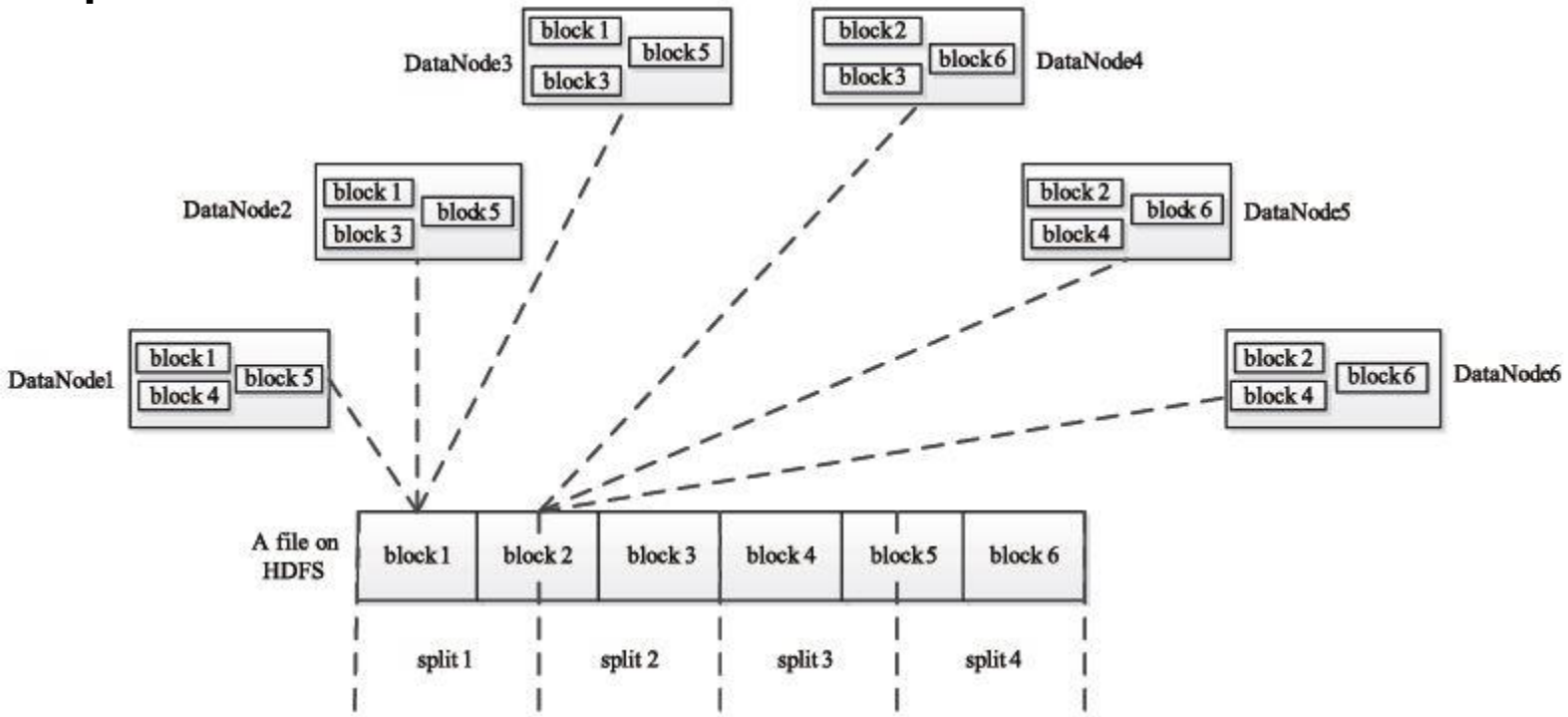
MapReduce各个执行阶段



OutputFormat：验证输出目录是否存在，以及输出结果类型是否符合要求等，如果都满足，则输出Reduce的结果到HDFS中

MapReduce各个执行阶段

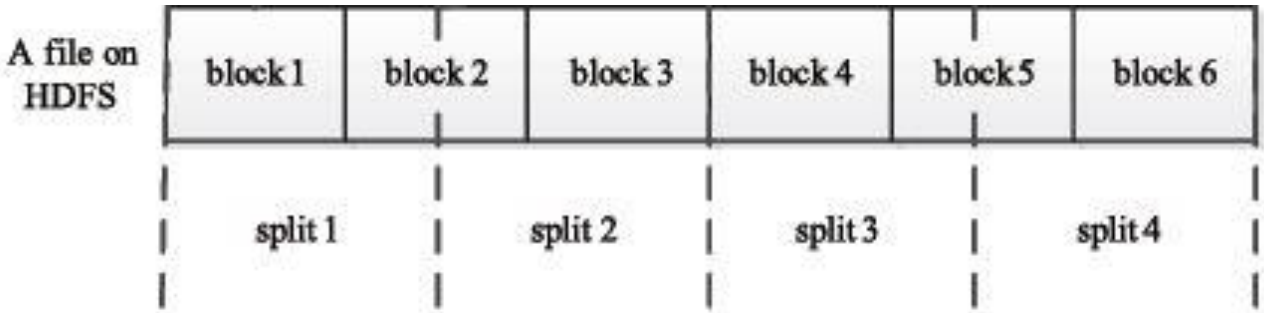
关于**Split**（分片）



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。可以通过设置“mapred.min.split.size”参数来设置split大小，默认把一个块的大小作为分片大小。

MapReduce各个执行阶段

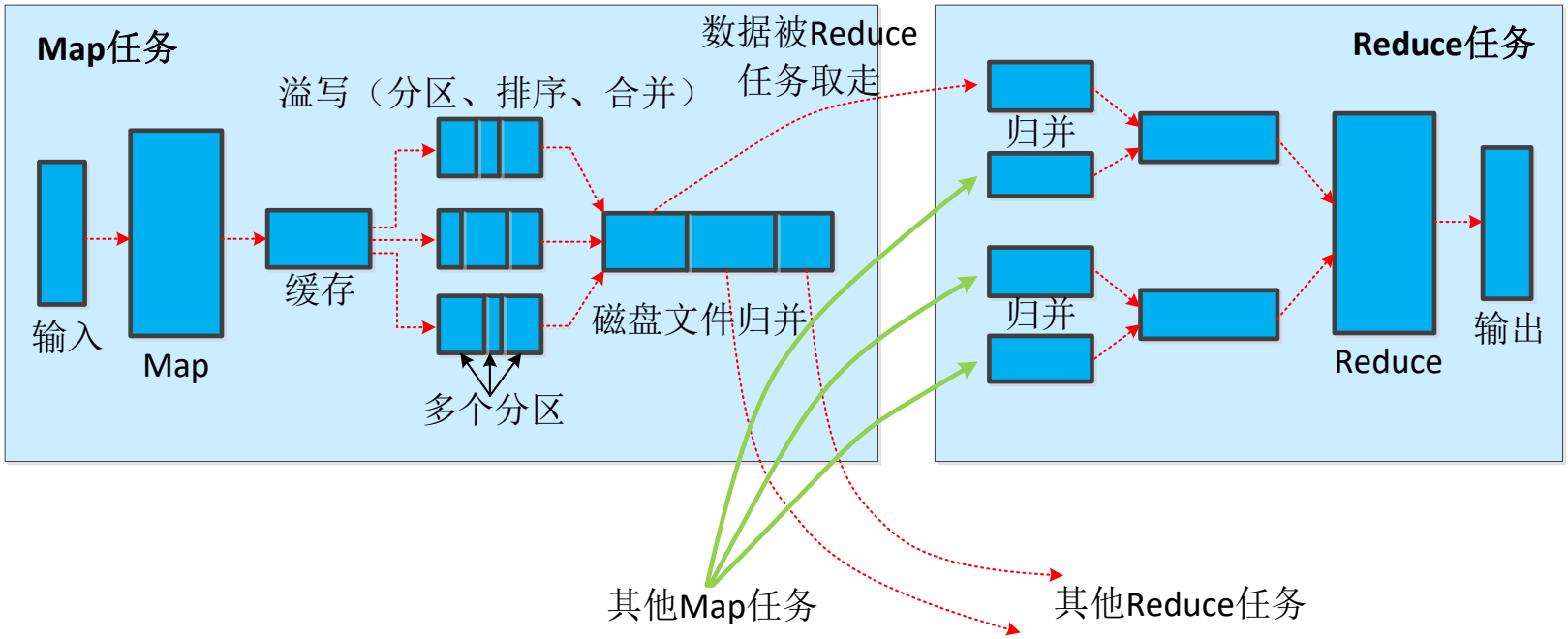
- **Map任务的数量**
  - Hadoop为每个split创建一个Map任务，split 的多少决定了Map任务的数目。大多数情况下，理想的分片大小是一个HDFS块



- **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）
  - 可以设置setNumReduceTasks(), 官方建议0.95* NUMBER_OF_NODES (集群中计算节点的个数) * mapred.tasktracker.reduce.tasks.maximum (每一个节点所分配的Reduce slot的个数)

Shuffle过程详解

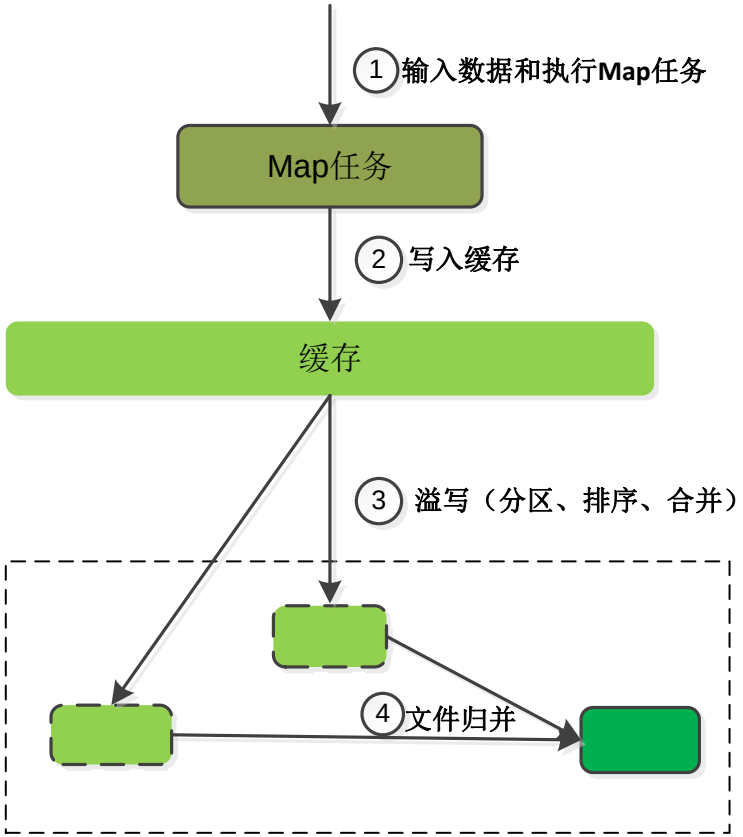
1. Shuffle过程简介



Shuffle：是指对Map输出的结果进行分区、排序、合并、归并等处理并交给Reduce的过程，分为Map端的操作和Reduce端的操作。

Shuffle过程详解

2. Map端的Shuffle过程



① 输入数据和执行Map任务

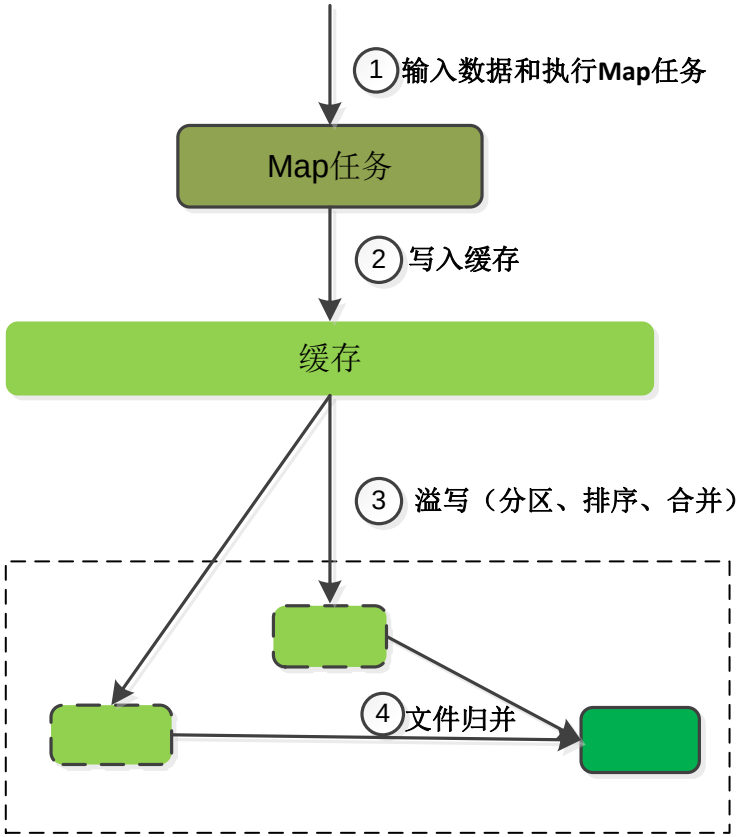
- Map任务的输入数据一般保存在分布式文件系统（如HDFS）的文件块中
- Map任务接受 被分配的分片Split中的<key, value> 作为输入后，按一定的映射规则转换成一批 <key, value> 进行输出

② 写入缓存

- 每个Map任务都会被分配一个缓存，默认大小是100MB
- Map的输出结果不是立即写入磁盘而是先写入缓存，当积累了一定数量的Map结果后，再批量的写入磁盘，以降低I/O开销

Shuffle过程详解

2. Map端的Shuffle过程



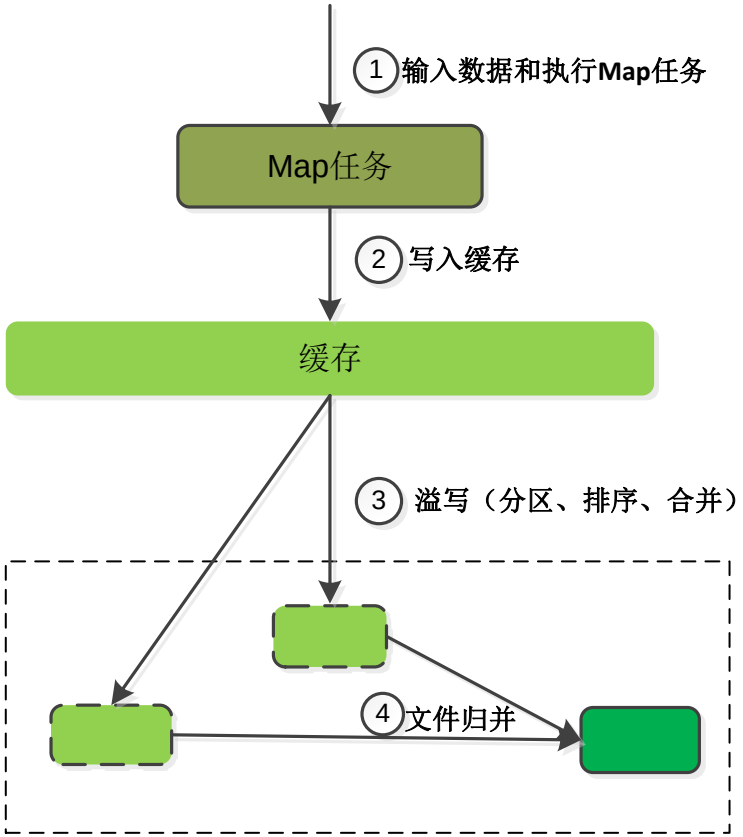
③ 溢写 (Spill)

- 提供给Map任务的缓存是有限的，默认大小是100MB
- 为保证Map结果可以不间断的写入缓存，不能等到缓存占满才启动溢写，通常设置溢写比例 0.8
- 写磁盘前先对缓存中数据做分区(partition)，默认采用Hash函数对key进行哈希再用Reduce任务数进行取模，即 hash(key) mod R，一个分区的数据交给个一Reduce处理
- 对于分区内的键值对会按key做排序(sort)
- 排序后可能包含一个可选的合并(combine)操作，由用户定义，与Reduce功能类似
- 经过分区、排序及可选的合并后，写磁盘并清缓存，每次溢写生成新文件，这个文件是分区有序的

- 合并（Combine）和归并（Merge）的区别:
- 两个键值对<“a”,1>和<“a”,1>，合并得到<“a”,2>，归并得到<“a”,<1,1>>

Shuffle过程详解

2. Map端的Shuffle过程



③ 文件归并 (merge)

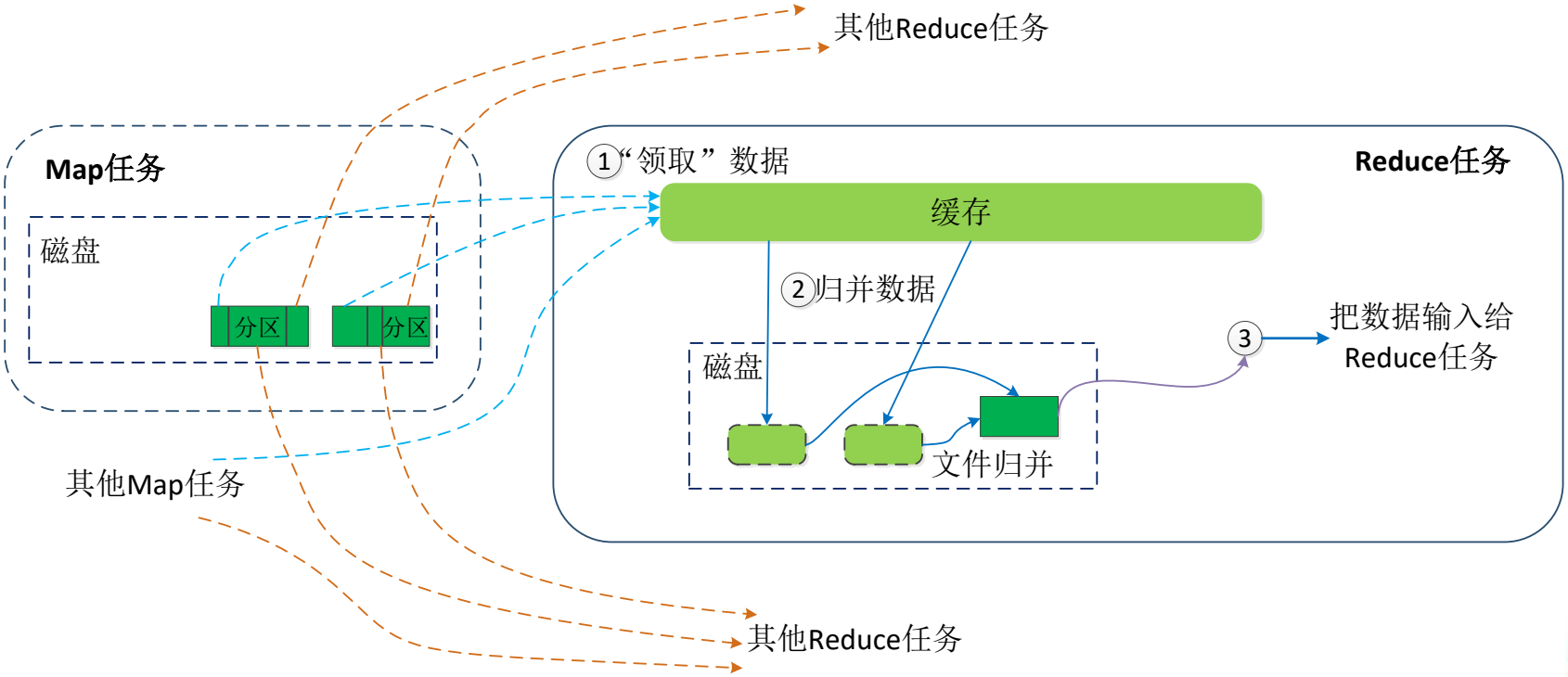
- 每次溢写都会在磁盘中产生一个新的溢写文件
- 在Map任务结束前，会对所有溢写文件中的数据进行归并，生成一个大的溢写文件，这个文件中的所有键值对也是分区有序的
- “归并”是将具有相同key的键值对归并成一个新的键值对，即将 <k1, v1>, <k1, v2>, …, <k1, vn>归并成 <k1, <v1, v2, …, vn>>
- 在做归并时，如果磁盘中的溢写文件数超过min.num.spills.for.combine值时（默认为3）会再次运行combiner对数据进行合并以减少写入磁盘的数据量
- 最终归并生成的文件存在本地磁盘上，这个文件也是分区有序的，不同的分区会被发给不同的Reduce任务
- JobTracker监测到Map任务执行结束，就会通知的Reduce任务来取数据，然后开始Reduce端的Shuffle

Shuffle过程详解

3. Reduce端的Shuffle过程

① “领取” (Fetch) 数据

- Map任务成功完成后，会通知TaskTracker，TaskTracker进而通知JobTracker
- Reduce会定期向JobTracker获取Map的输出位置，一旦拿到输出位置，Reduce任务就会从此输出对应的节点上复制输出到本地，而不会等到所有的Map任务结束

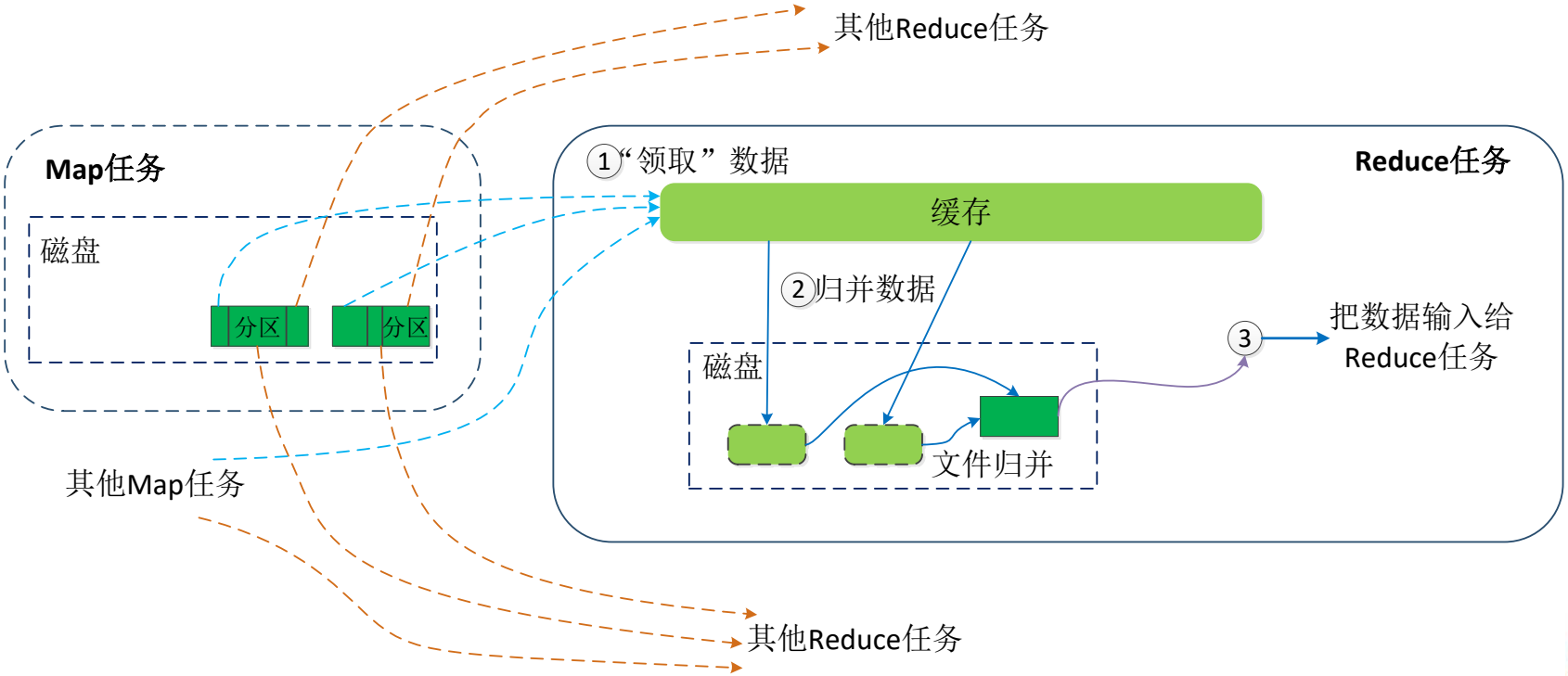


Shuffle过程详解

3. Reduce端的Shuffle过程

② 归并数据

- 从Map端取回的数据首先被放入缓存，缓存满则会启动溢写
- 溢写过程对来自不同的Map的数据中具有相同key的数据进行归并，如果用户定义了combiner，可再执行合并，再写入磁盘，以减少写入磁盘的数据量

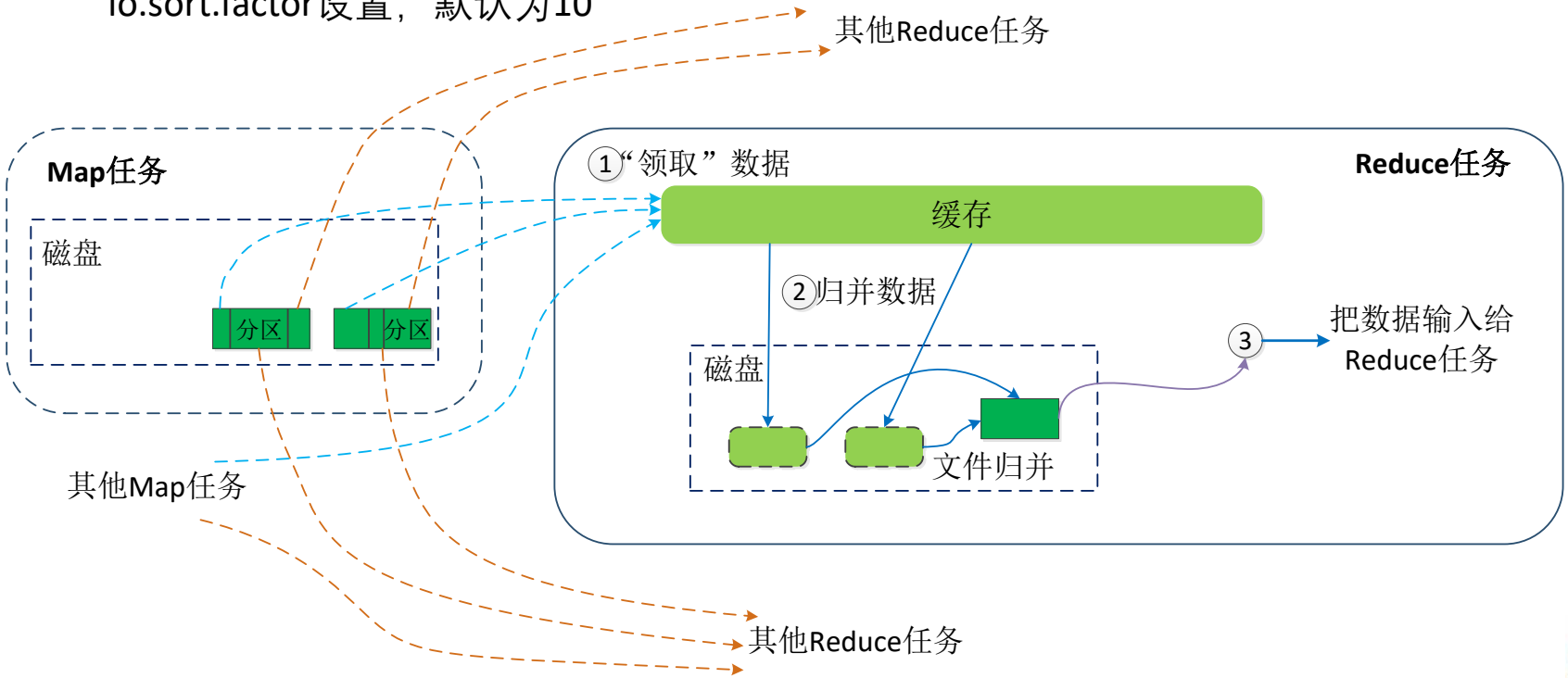


Shuffle过程详解

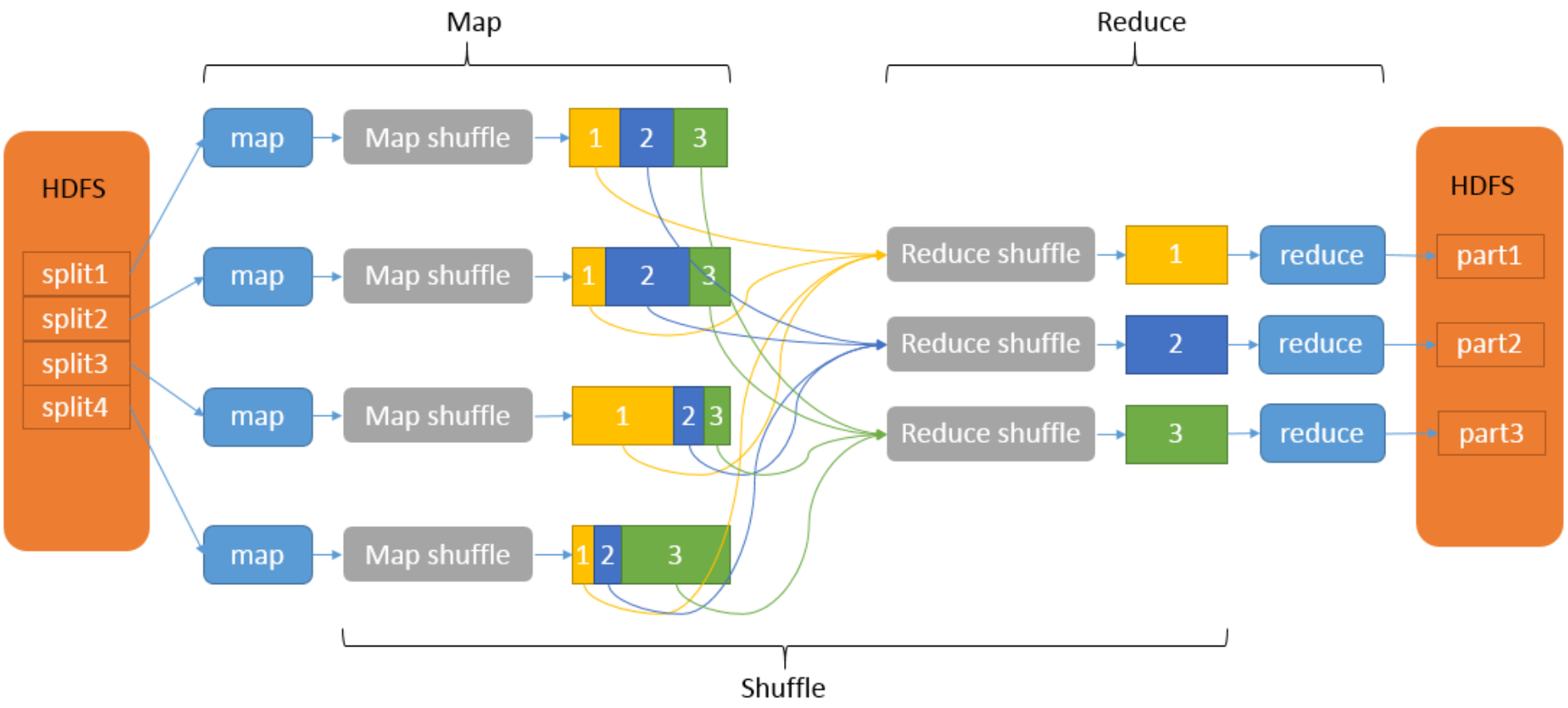
3. Reduce端的Shuffle过程

② 归并数据

- 当所有Map端的数据都被取回时，与Map端类似，多个溢写文件会被归并成一个大文件，同样是对键值排序的
- 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
- 当数据量很大溢写文件很多时，可能产生多轮归并，每轮可归并的文件数量由io.sort.factor设置，默认为10

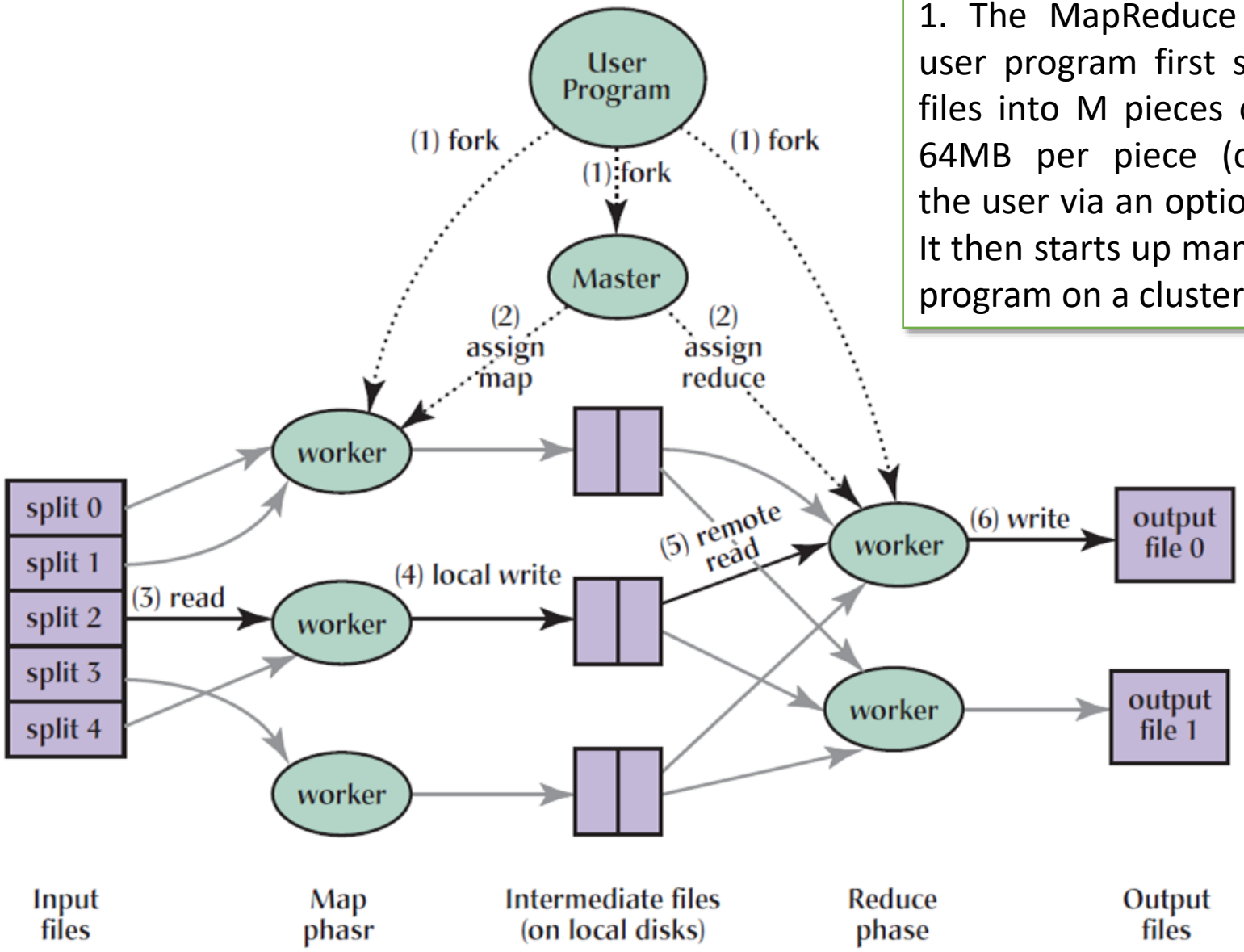


Shuffle过程详解



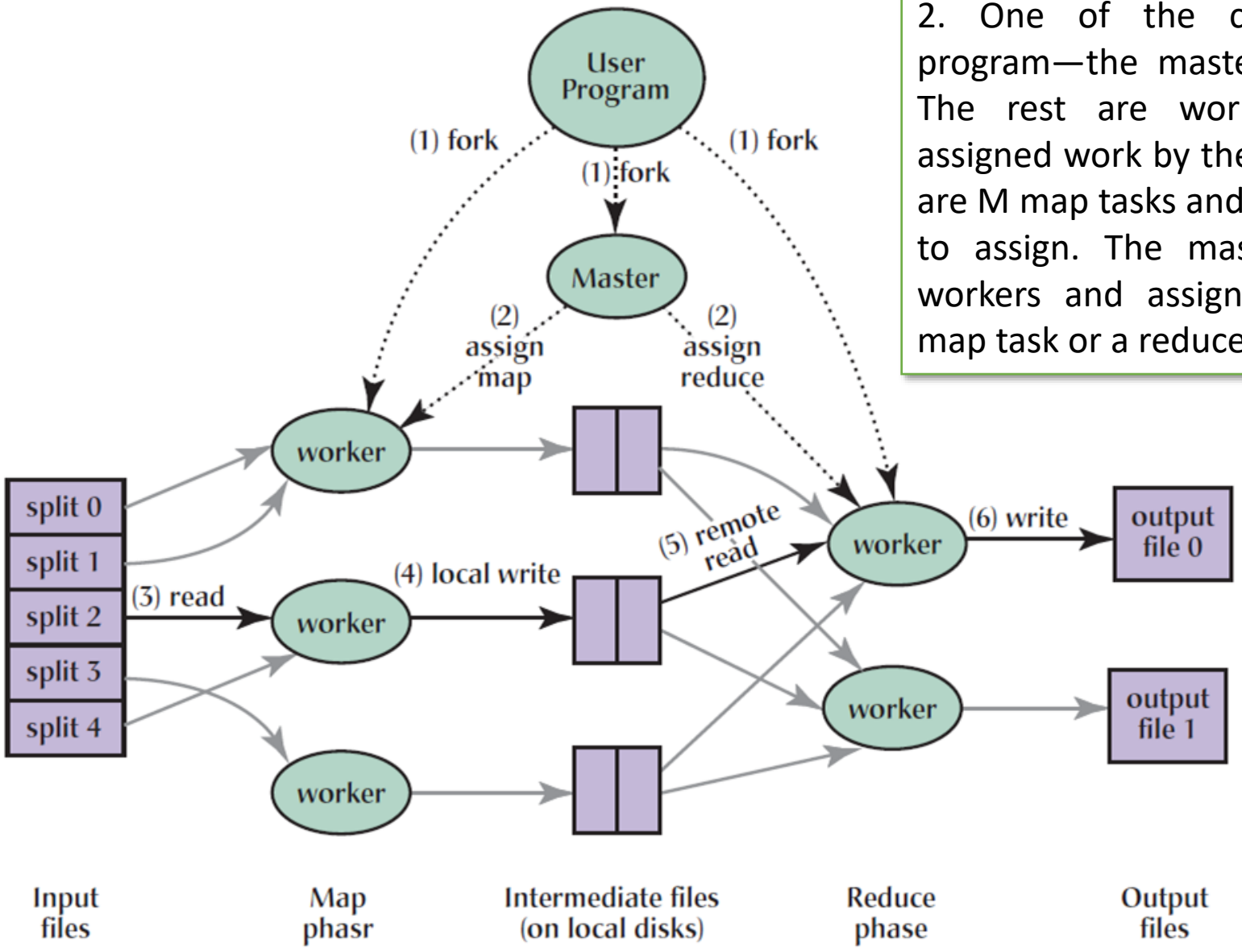
MapReduce应用程序执行过程

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16-64MB per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.



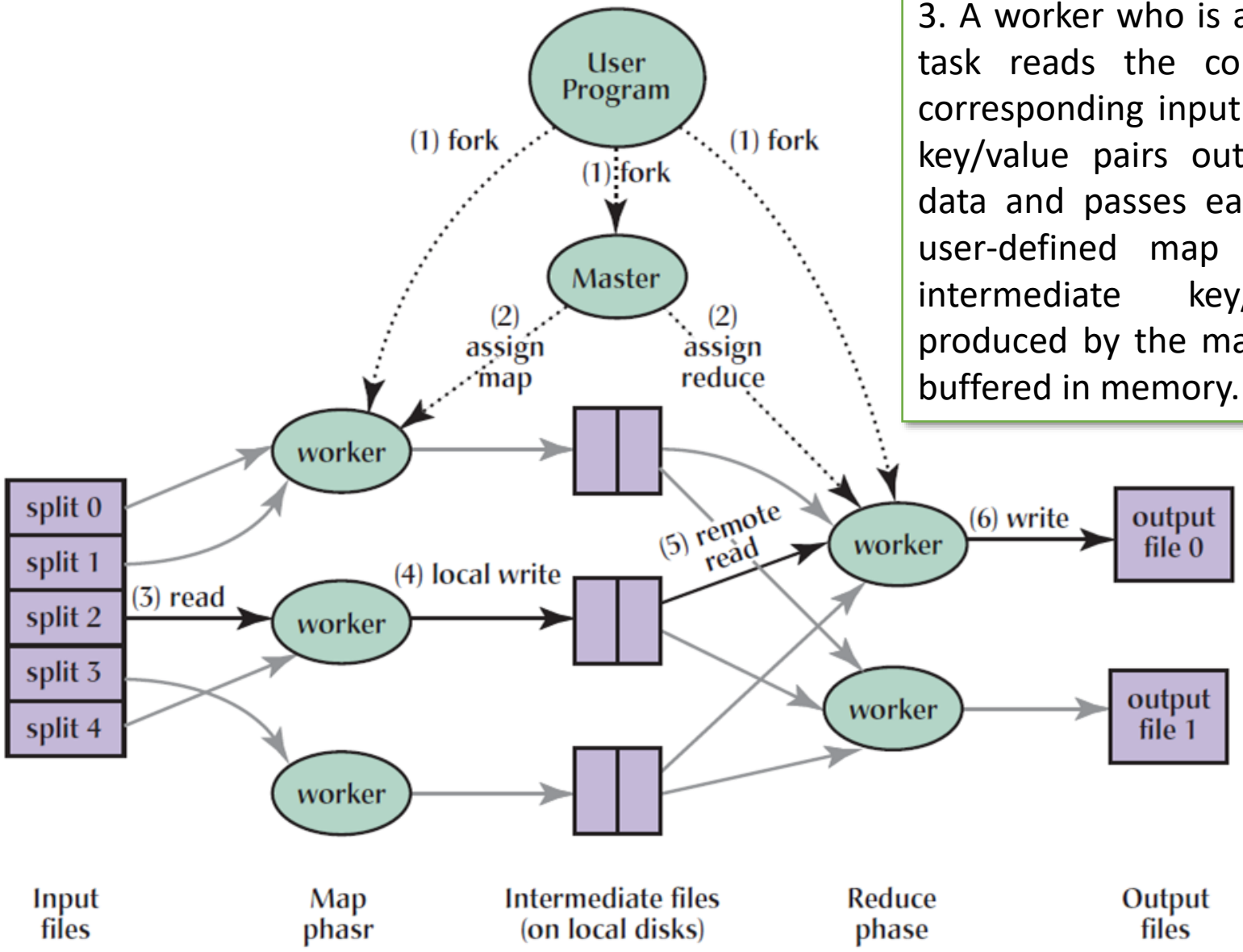
MapReduce应用程序执行过程

2. One of the copies of the program—the master— is special. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.



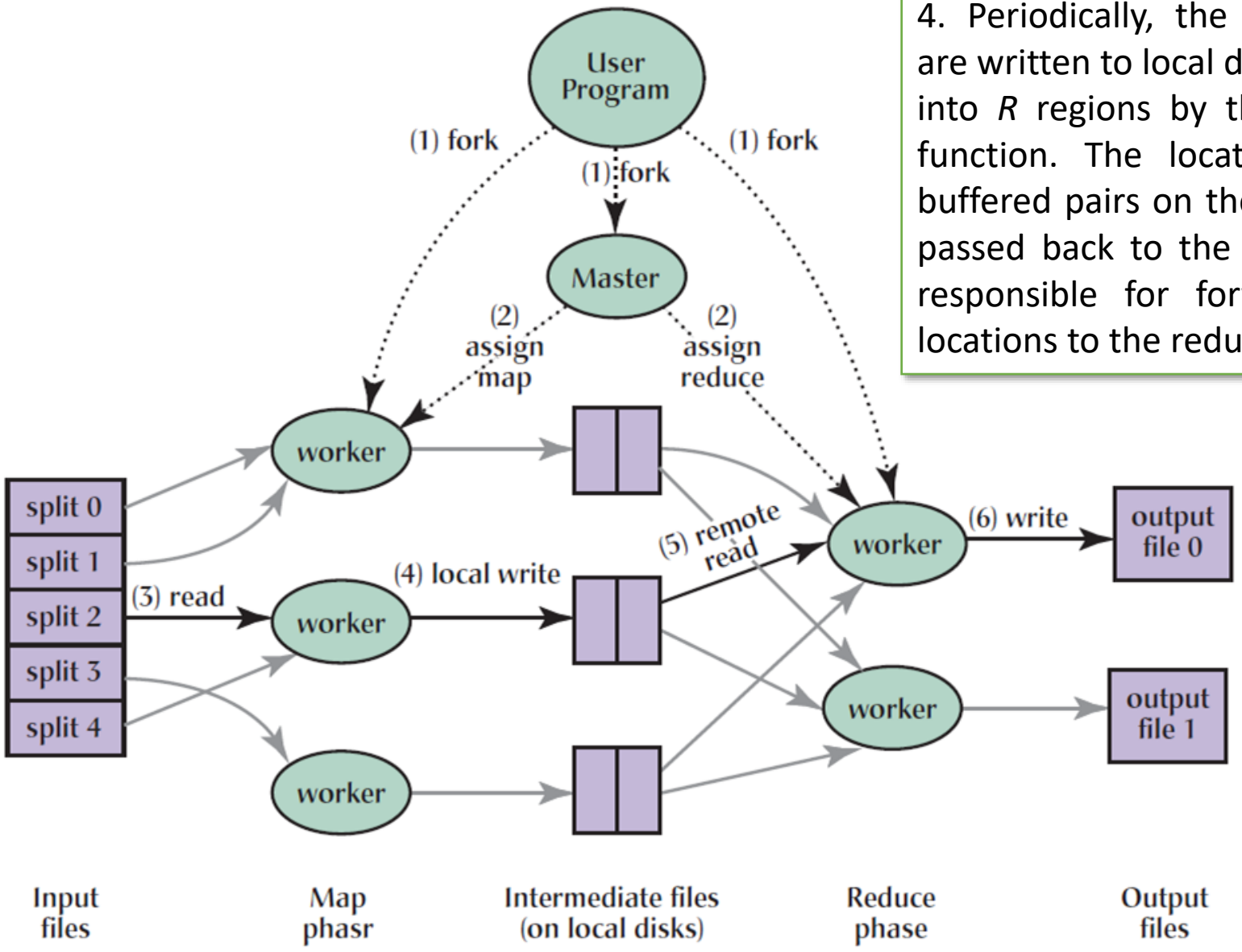
MapReduce应用程序执行过程

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined map function. The intermediate key/value pairs produced by the map function are buffered in memory.

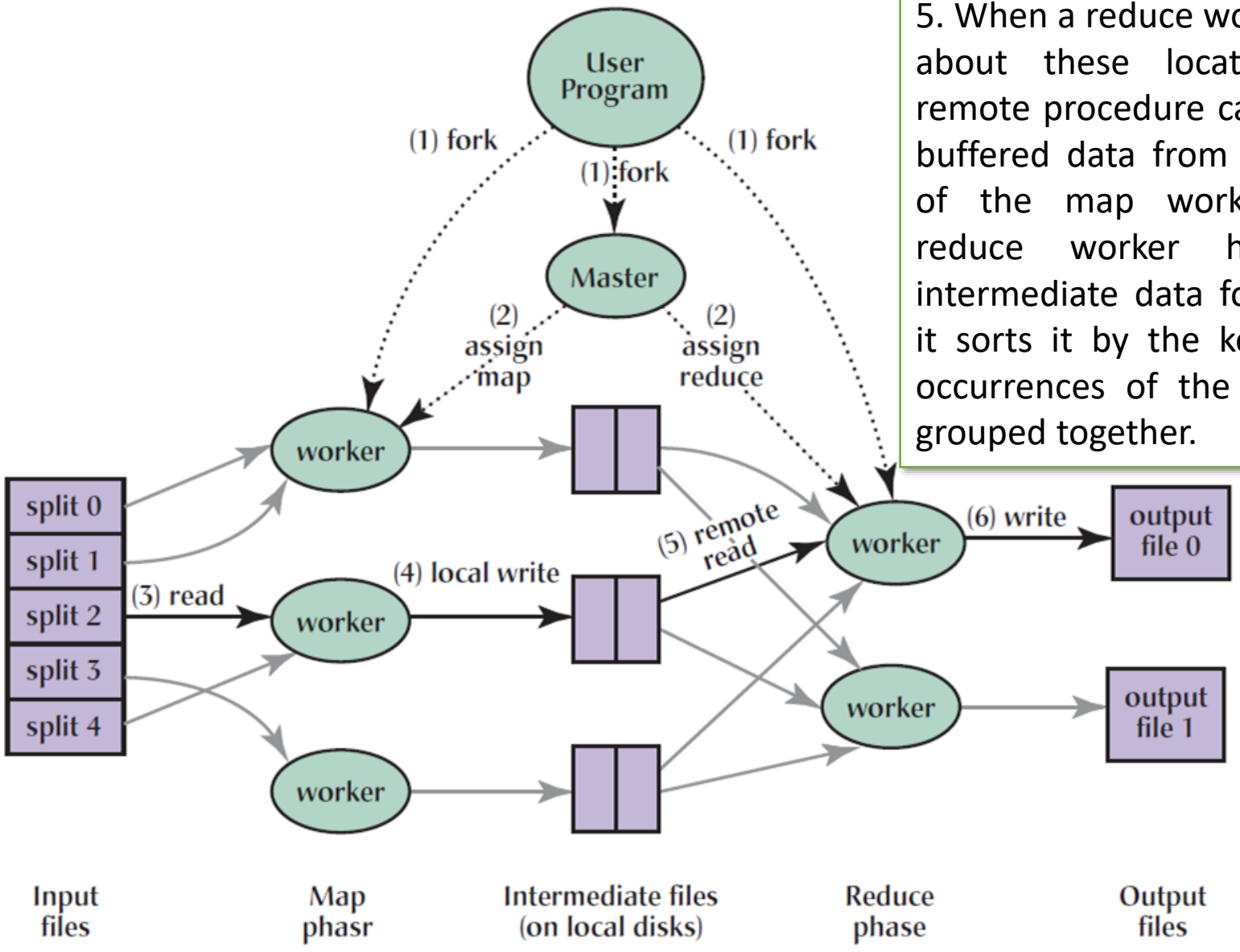


MapReduce应用程序执行过程

4. Periodically, the buffered pairs are written to local disk, partitioned into *R* regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master who is responsible for forwarding these locations to the reduce workers.



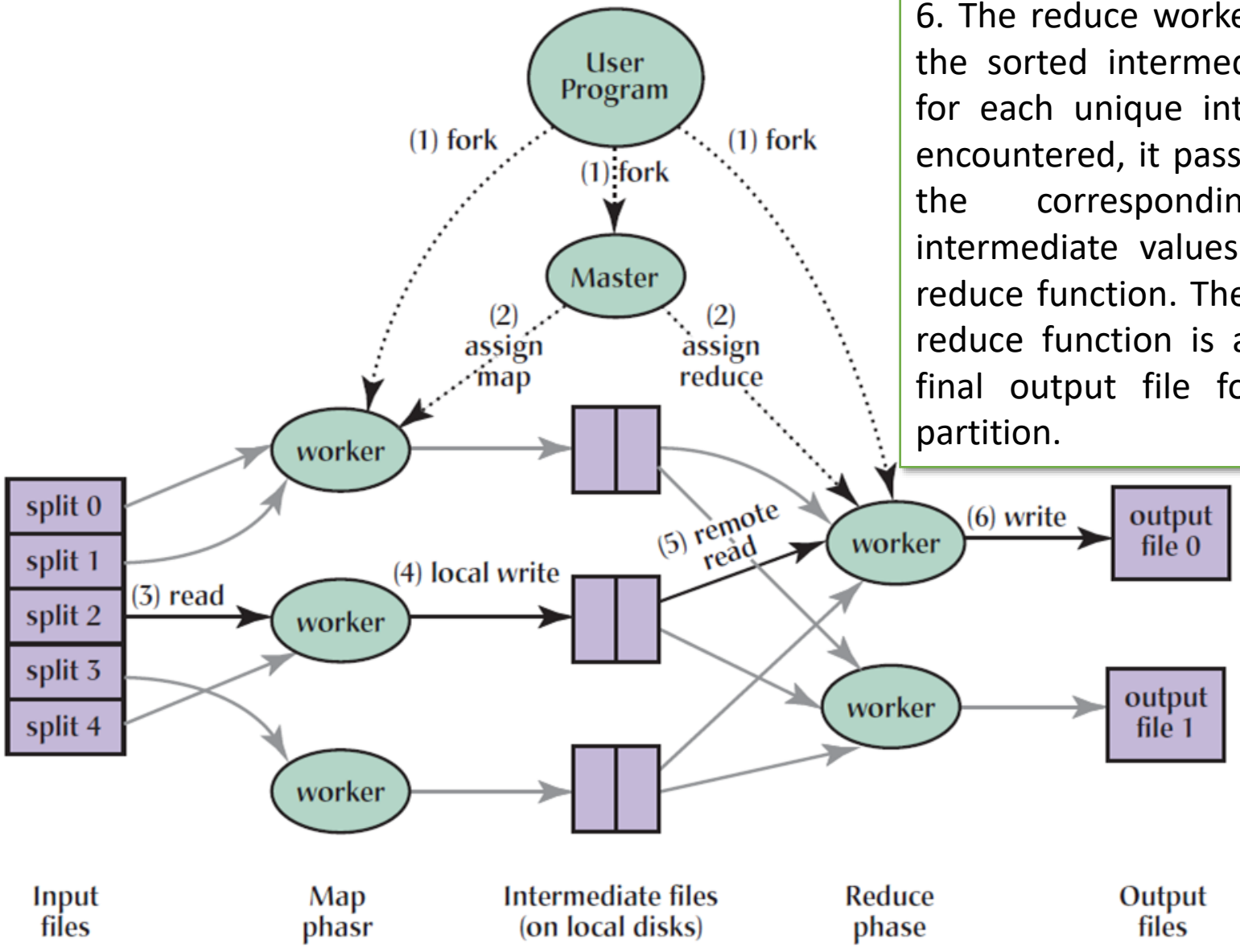
MapReduce应用程序执行过程



5. When a reduce worker is notified about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data for its partition, it sorts it by the keys so that all occurrences of the same key are grouped together.

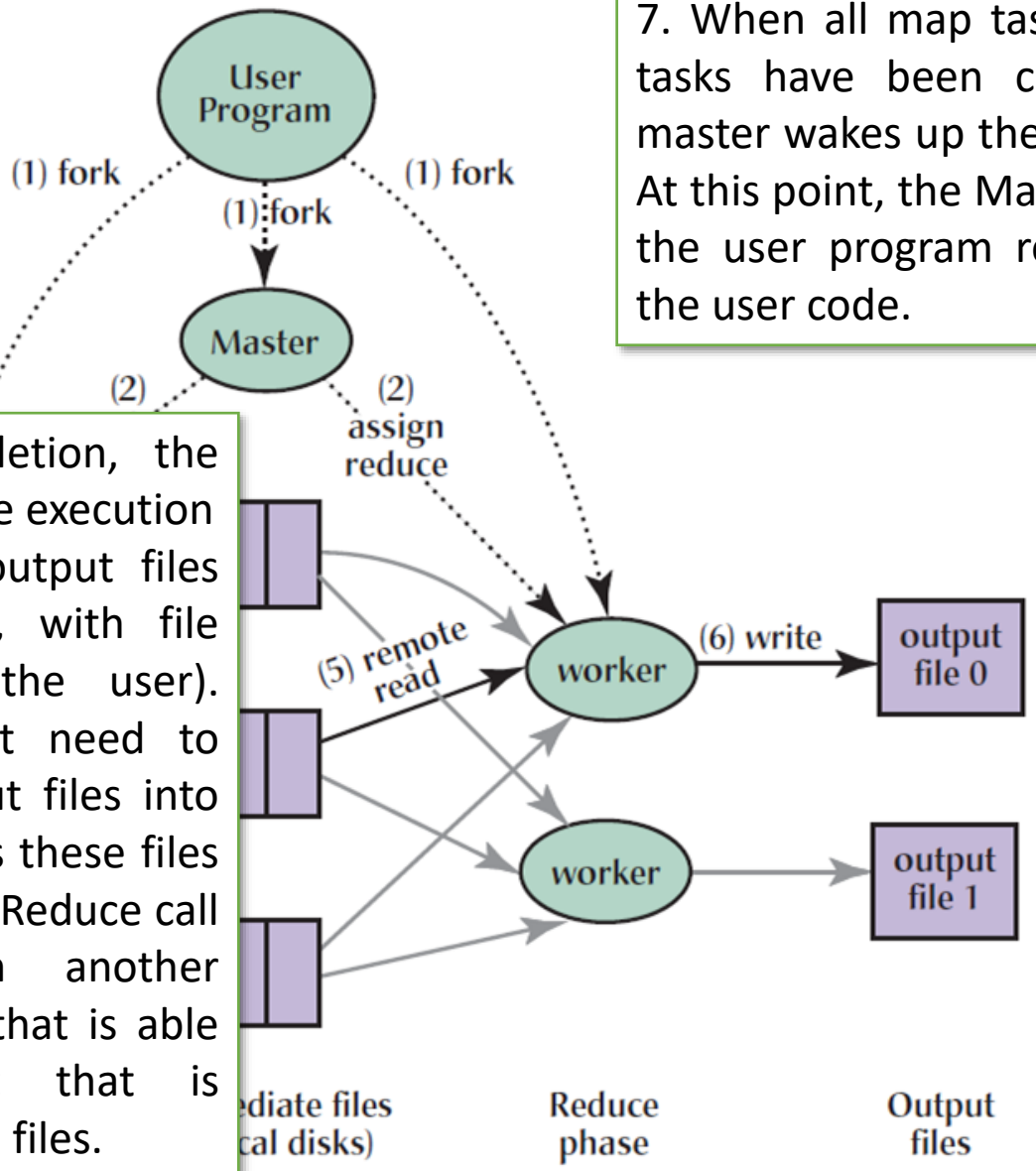
MapReduce应用程序执行过程

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's reduce function. The output of the reduce function is appended to a final output file for this reduce partition.



MapReduce应用程序执行过程

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.



After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names specified by the user). Typically, users do not need to combine these R output files into one file; they often pass these files as input to another MapReduce call or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

实例分析：WordCount

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

一个WordCount的输入和输出实例

输入	输出
Hello World Bye World Hello Hadoop Bye Hadoop Bye Hadoop Hello Hadoop	Bye 3 Hadoop 4 Hello 3 World 2

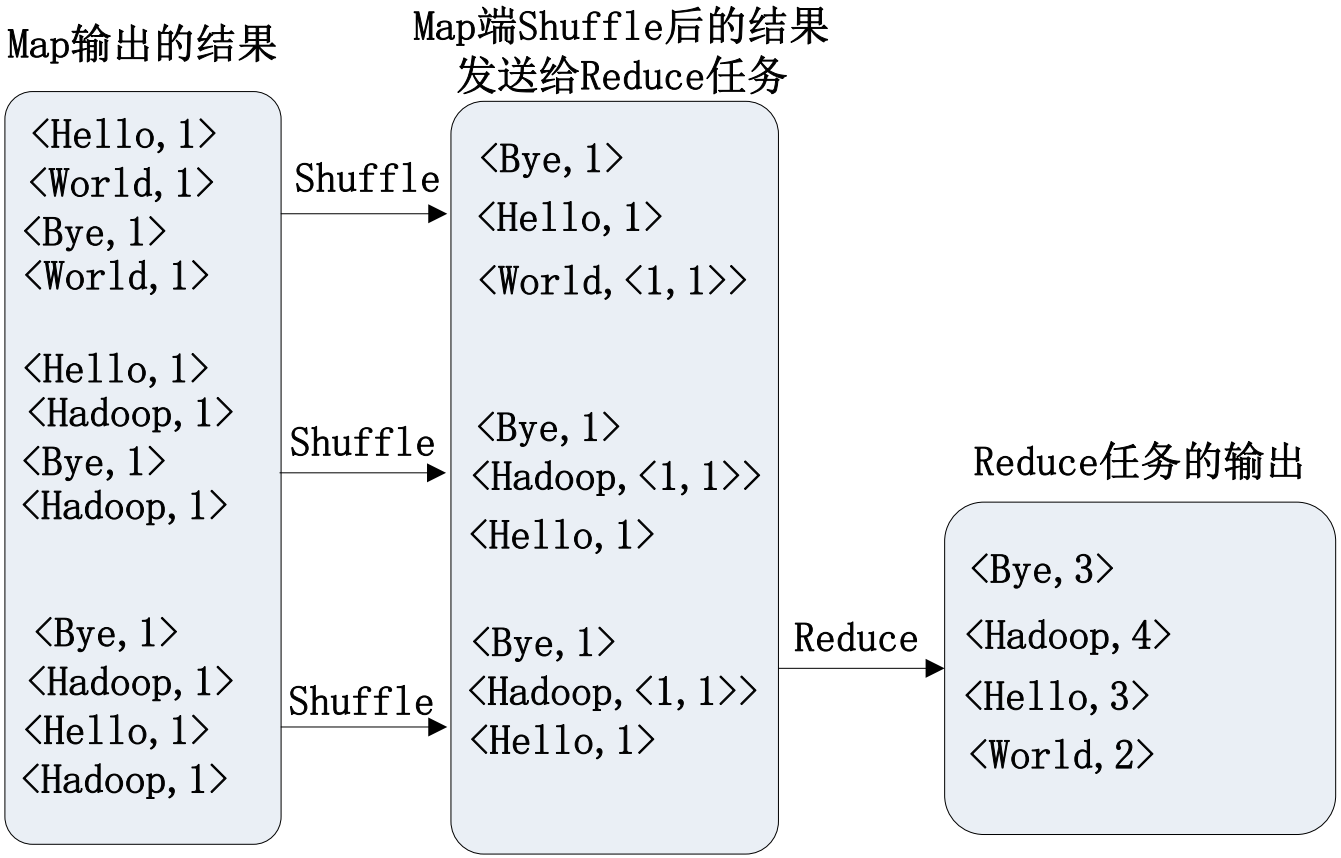
MapReduce程序设计思路

- 首先，需要检查程序任务是否可以采用MapReduce来实现，即是否可以将待处理的数据集分解成许多小数据集，而每个小数据集可以完全并行的进行处理。
- 其次，确定MapReduce程序的设计思路和执行过程

一个WordCount执行过程的实例

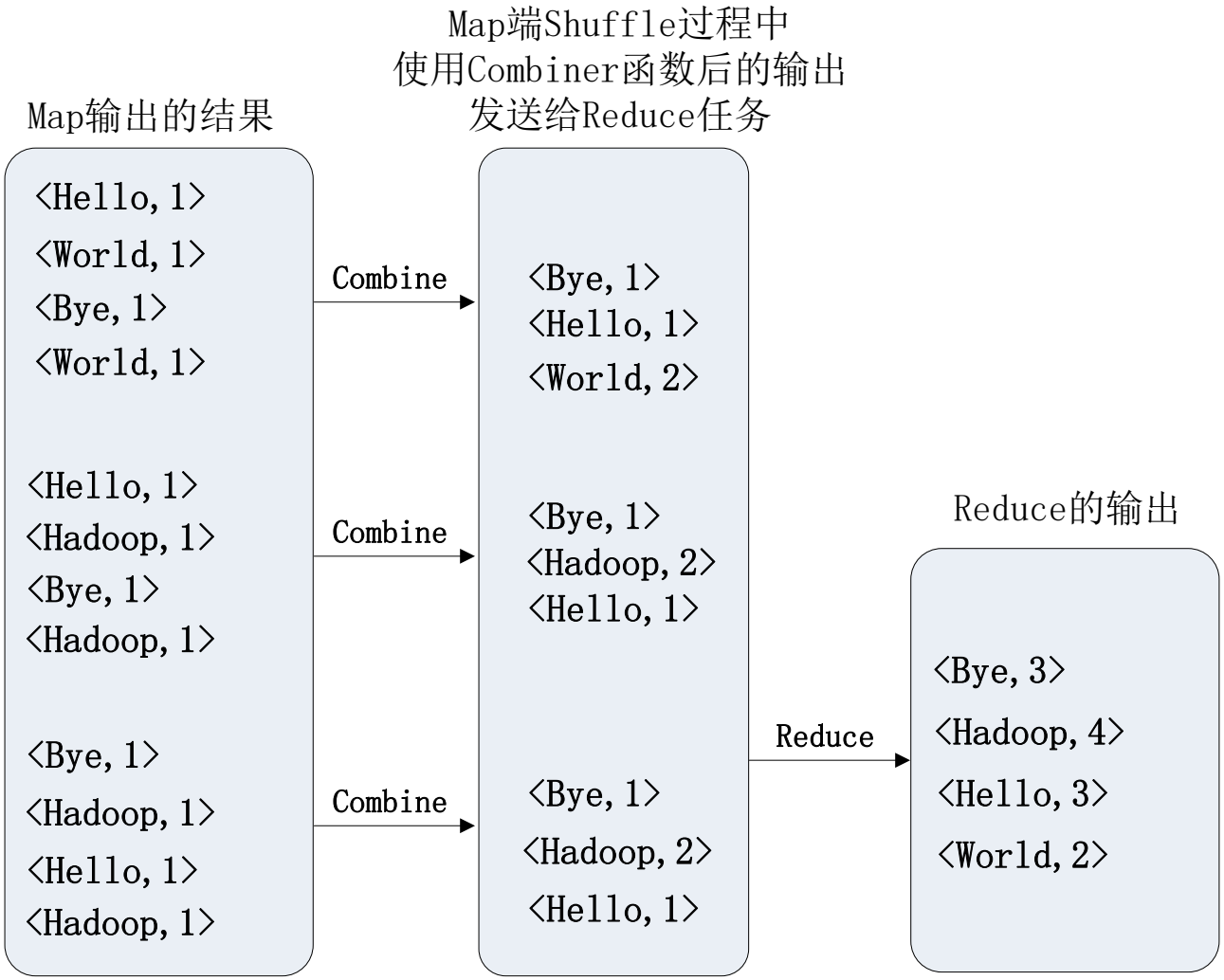
Map过程示意图

一个WordCount执行过程的实例



用户没有定义Combiner时的Reduce过程示意图

一个WordCount执行过程的实例



用户有定义Combiner时的Reduce过程示意图

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

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

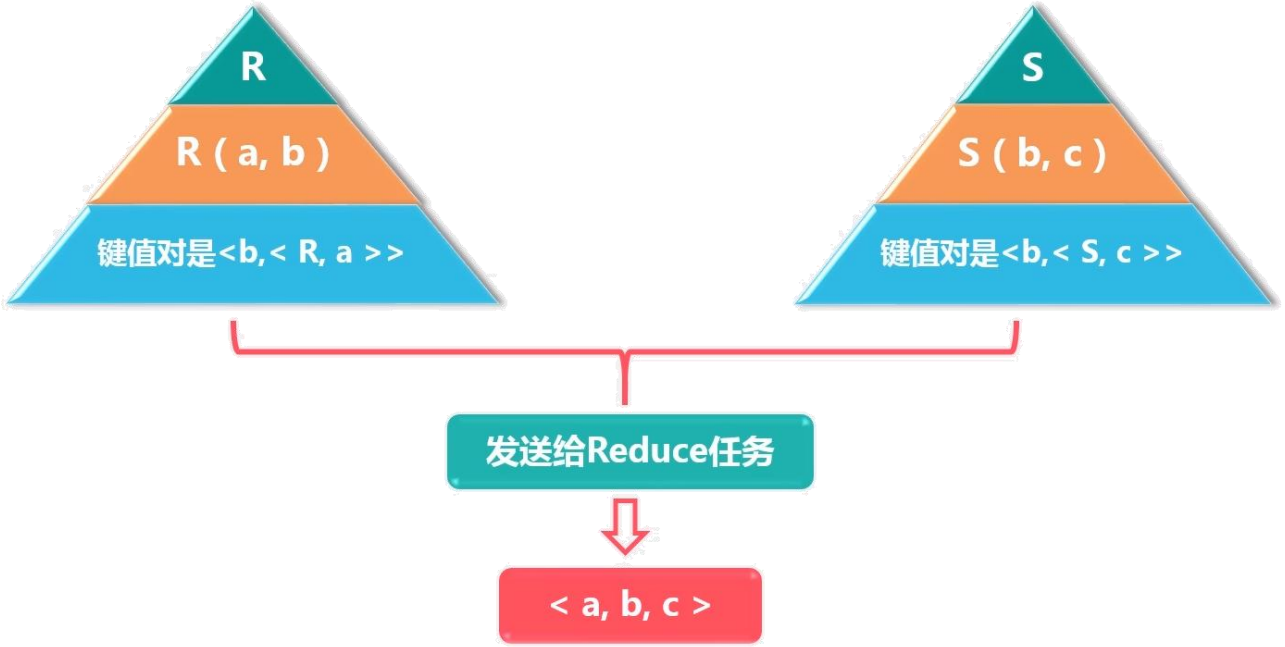
部门

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

用MapReduce实现关系的自然连接



- 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

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)

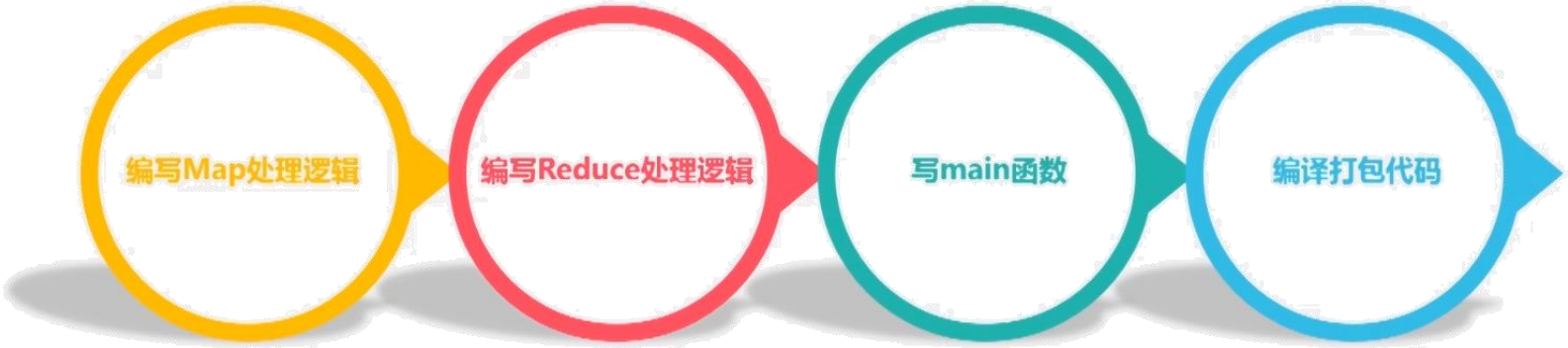
(2,a,d2,10,5)

(2,a,d2,50,100)

(3,b,d3,20,1)

MapReduce编程实践

- 任务要求：用MapReduce实现对输入文件中的单词做词频统计



https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

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>

```
public static class TokenizerMapper
     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);
    }
  }
}
```

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的输入数据为<key, Iterable容器>
- Reduce阶段的任务：对输入数字序列进行求和

```
public static class IntSumReducer
     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);
  }
}
```

编写main函数

```
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(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(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]));
    }
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[otherArgs.length - 1]));

    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.Text;
import org.apache.hadoop.io.IntWritable;
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 static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>{ ...
    }

    public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable> { ...
    }

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

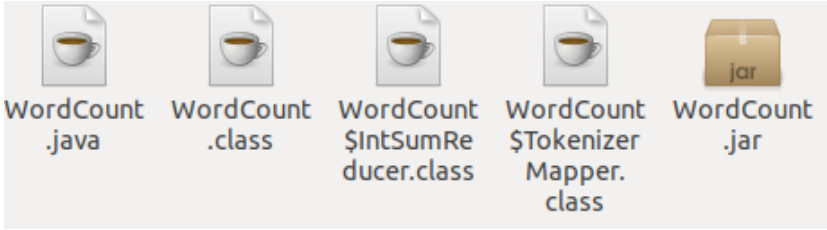
编译打包代码以及运行程序

- 使用命令行编译打包运行MapReduce程序
  - http://dblab.xmu.edu.cn/blog/hadoop-build-project-by-shell/
- 使用Eclipse编译运行MapReduce程序
  - http://dblab.xmu.edu.cn/blog/hadoop-build-project-using-eclipse/

使用命令行编译打包MapReduce程序

实验步骤：

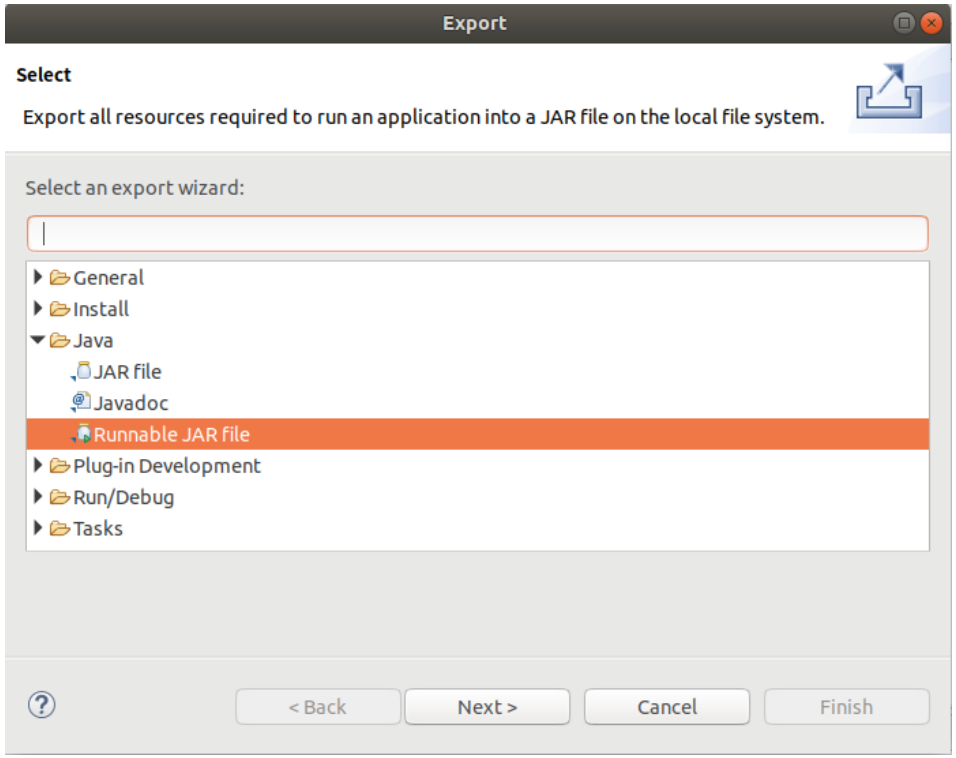
- 注意需要将Hadoop 的 classpath 信息添加到 CLASSPATH 变量中
- 通过 javac 命令编译.java文件，生成.class文件
- 把.class 文件打包成 jar
- 运行jar包（启动Hadoop）
- 查看结果

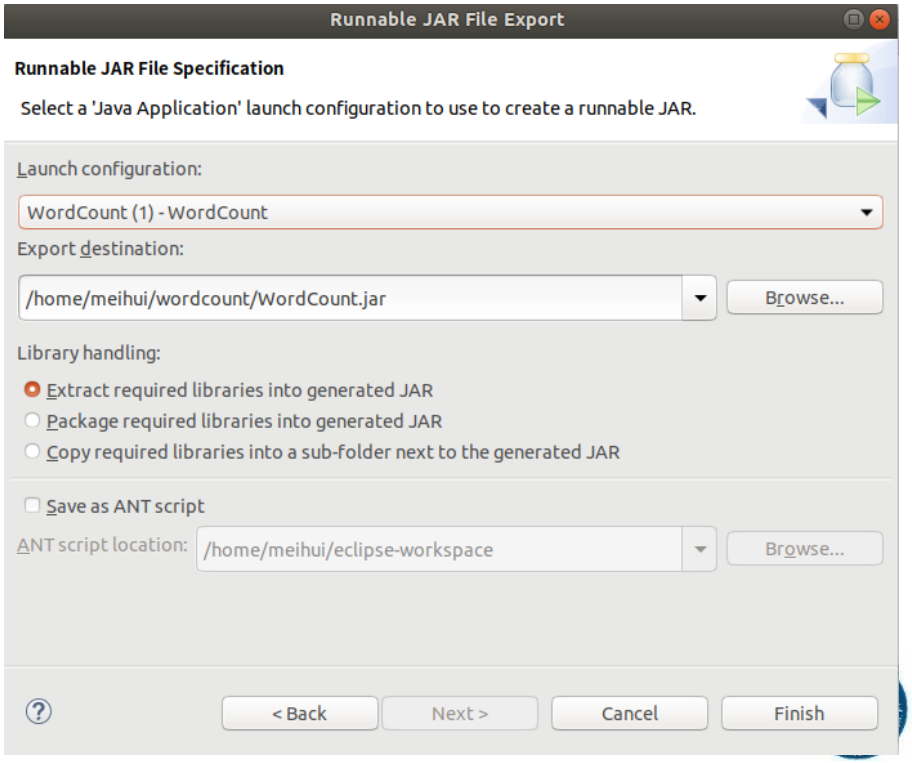



```
meihui@meihui-VirtualBox:~/wordcount$ javac WordCount.java
meihui@meihui-VirtualBox:~/wordcount$ jar -cvf WordCount.jar *.class
已添加清单
正在添加: WordCount.class(输入 = 1907) (输出 = 1041)(压缩了 45%)
正在添加: WordCount$IntSumReducer.class(输入 = 1739) (输出 = 741)(压缩了 57%)
正在添加: WordCount$TokenizerMapper.class(输入 = 1736) (输出 = 750)(压缩了 56%)
meihui@meihui-VirtualBox:~/wordcount$ hadoop jar WordCount.jar WordCount input/file1.txt input/file2.txt output
```

使用Eclipse打包MapReduce程序

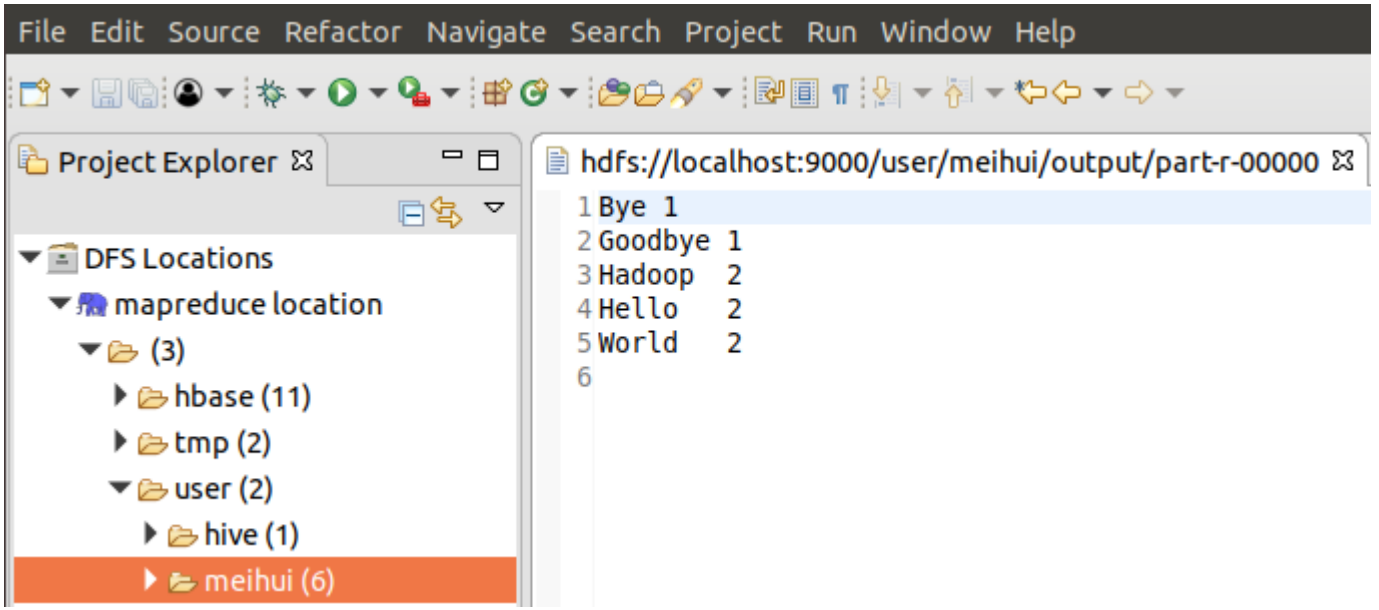
- 选择需要打包的项目，右键 Export
- 选择Runnable JAR file,然后点击 Next
- 选择jar包运行的main类，以及定义jar包的名字，保存的地方





使用Eclipse运行MapReduce程序

- 安装 Hadoop-Eclipse-Plugin
  - 将插件复制到 Eclipse 安装目录的 plugins 文件夹中，运行 eclipse -clean 重启 Eclipse
- 配置好hadoop location后，可以直接查看 HDFS 中的文件



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
- 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
- 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