02

大数据处理框架Hadoop

- Hadoop简介
- Hadoop的发展历程
- Hadoop的优点
- Mapreduce计算流程
- Yarn调度流程
- WordCount例子

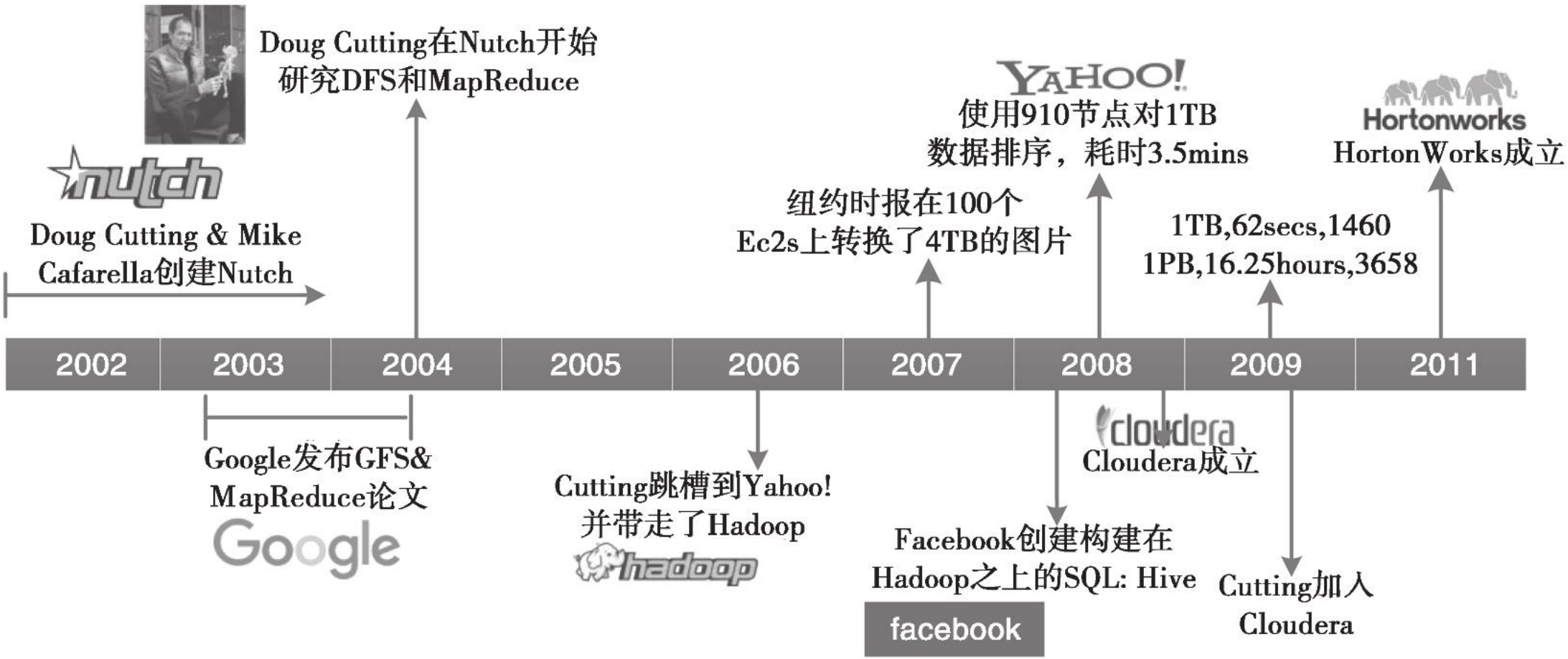
Hadoop

Hadoop 是一个由 Apache 基金会所开发的分布式系统基础架构，使用户在不了解分布式底层细节的情况下开发分布式程序，充分利用集群的威力进行高速运算和存储。

Hadoop 的两大核心：HDFS 和 MapReduce，分别解决了两大问题：大数据存储和大数据分析。

- HDFS(Hadoop Distributed File System)是可扩展、容错、高性能的分布式文件系统，异步复制，一次写入多次读取，主要负责存储。
- MapReduce 为分布式计算框架，包含map(映射)和 reduce(归约)过程，负责在 HDFS 上进行计算。

Hadoop发展历史


Doug Cutting在Nutch开始
研究DFS和MapReduce
Doug Cutting & Mike
Cafarella创建Nutch
YAHOO!
使用910节点对1TB
数据排序，耗时3.5mins
Hortonworks
HortonWorks成立
纽约时报在100个
Ec2s上转换了4TB的图片
1TB,62secs,1460
1PB,16.25hours,3658
2002
2003
2004
2005
2006
2007
2008
2009
2011
Google发布GFS&
MapReduce论文
Google
Cutting跳槽到Yahoo!
并带走了Hadoop
hadoop
cloudera
Cloudera成立
Facebook创建构建在
Hadoop之上的SQL: Hive
facebook
Cutting加入
Cloudera

Hadoop的优点

Hadoop 是一个能够让用户轻松架构和使用的分布式计算的平台。用户可以轻松地在 Hadoop 发和运行处理海量数据的应用程序。其优点主要有以下几个：

(1) 高可靠性： Hadoop 按位存储和处理数据的能力值得人们信赖。

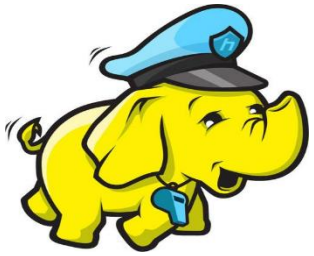
(2) 高扩展性： Hadoop 是在可用的计算机集簇间分配数据并完成计算任务的，这些集簇可以方便地扩展到数以千计的节点中。

(3) 高效性： Hadoop能够在节点之间动态地移动数据，并保证各个节点的动态平衡，因此处理速度非常快。

(4) 高容错性： Hadoop能够自动保存数据的多个副本，并且能够自动将失败的任务重新分。

(5) 低成本： 与一体机、商用数据仓库以及 QlikView、 Yonghong Z- Suites 等数据集市相比，Hadoop 是开源的，项目的软件成本因此会大大降低。

Hadoop 带有用 Java 语言编写的框架，因此运行在 linux 生产平台上是非常理想的， Hadoop 上的应用程序也可以使用其他语言编写，比如 C++，Python。



MapReduce模型

•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；

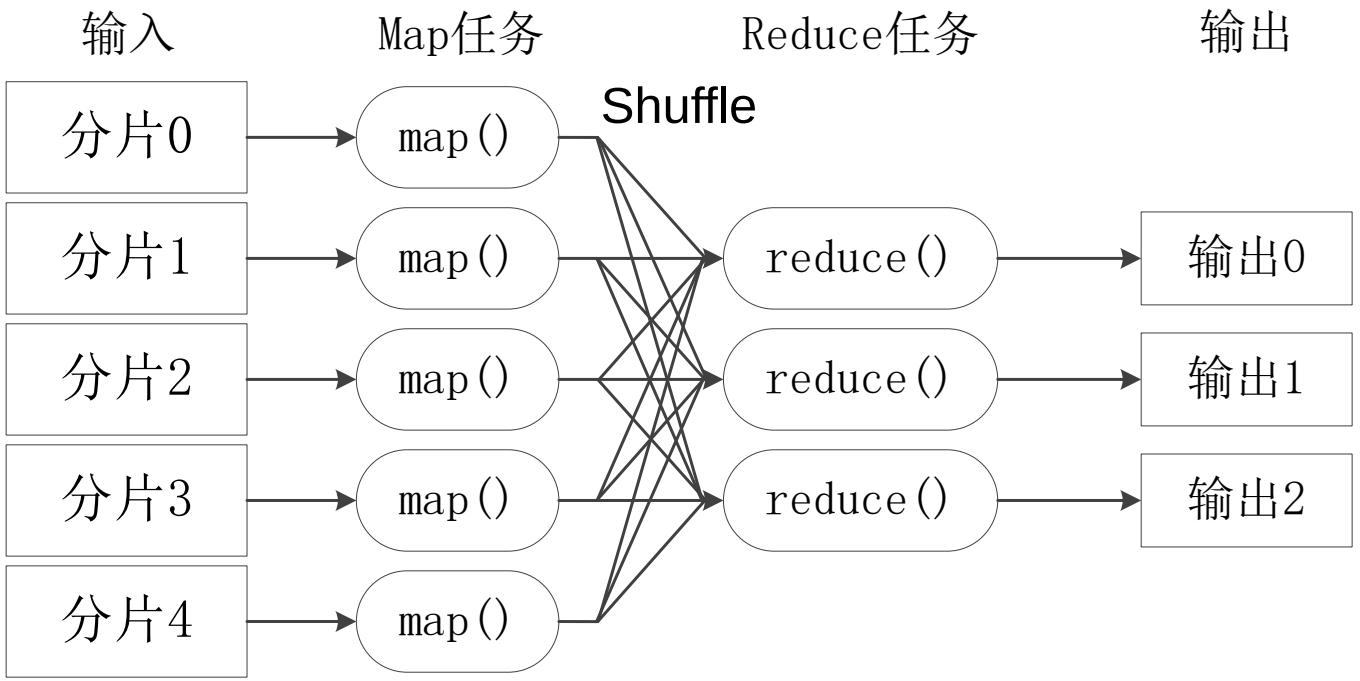
•MapReduce采用“**分而治之**”策略，一个存储在分布式文件系统中的大规模数据集，会被切分成许多独立的分片（split），这些分片可以被多个Map任务并行处理；

•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。

•Hadoop框架是用Java实现的，但是，MapReduce应用程序支持多种编程语言。

MapReduce工作流程



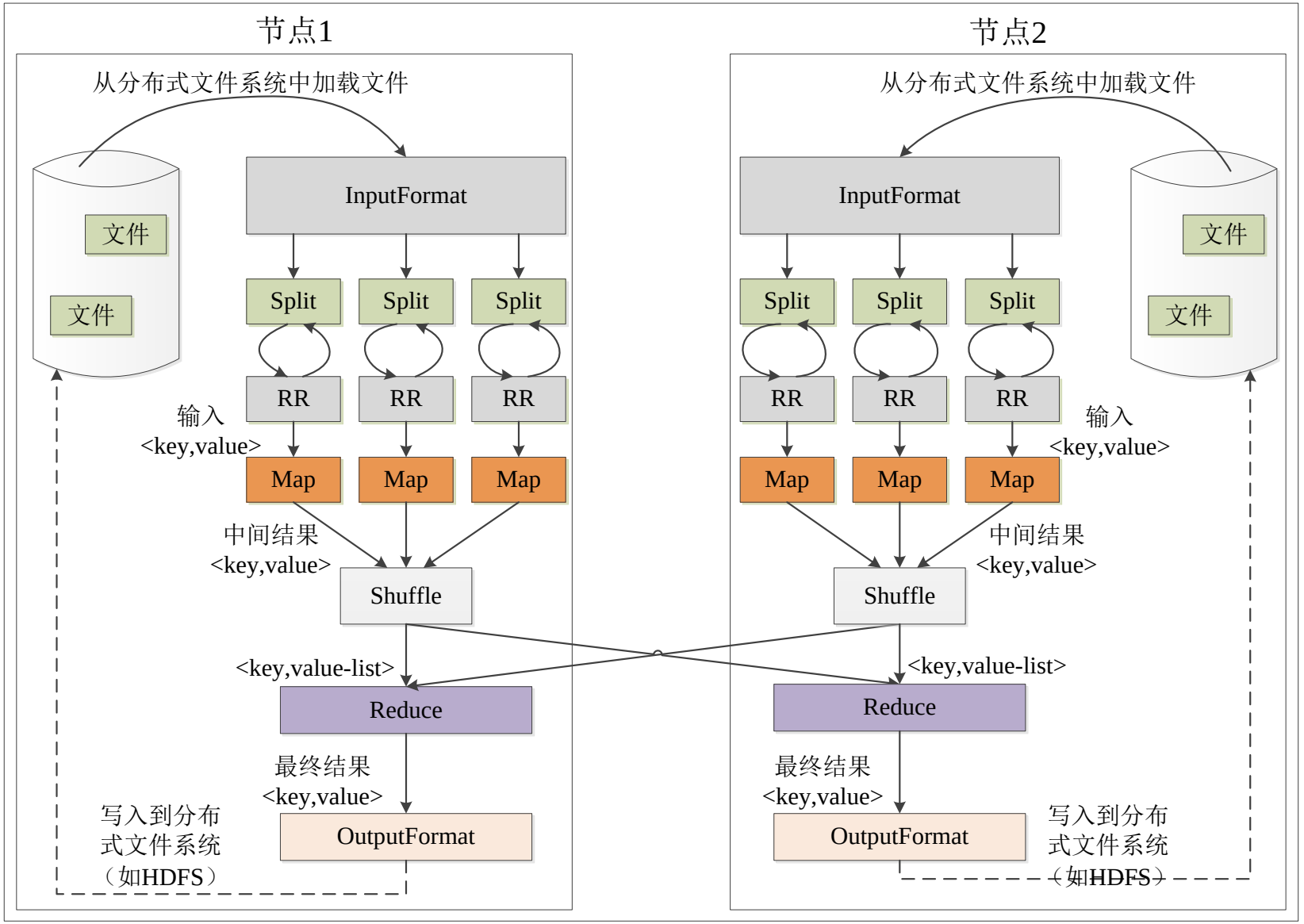
- 不同的Map任务之间不会进行通信
- 不同的Reduce任务之间也不会发生任何信息交换
- 用户不能显式地从一台机器向另一台机器发送消息
- 所有的数据交换都是通过MapReduce框架自身去实现的

Map和Reduce函数

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

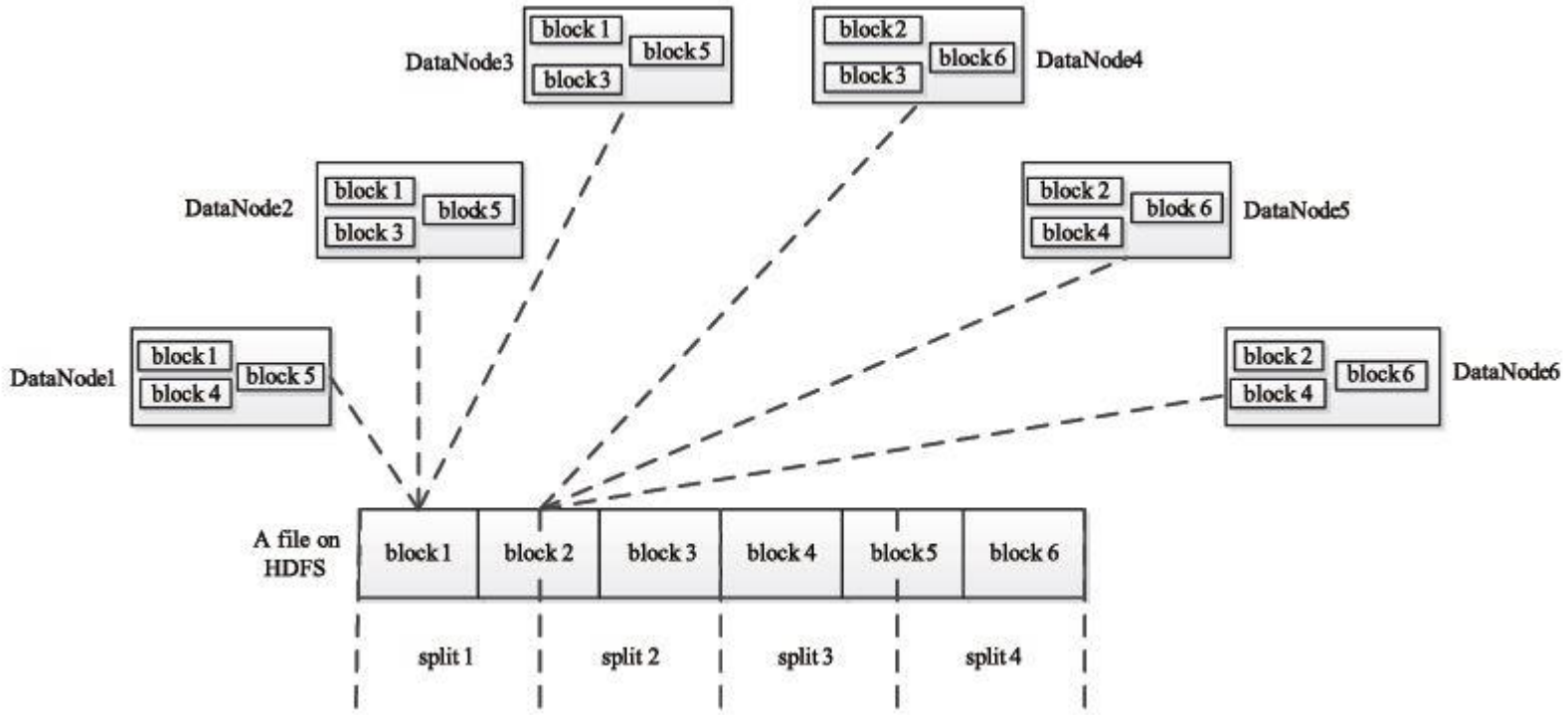
MapReduce各个执行阶段

RR: RecordReader



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。

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

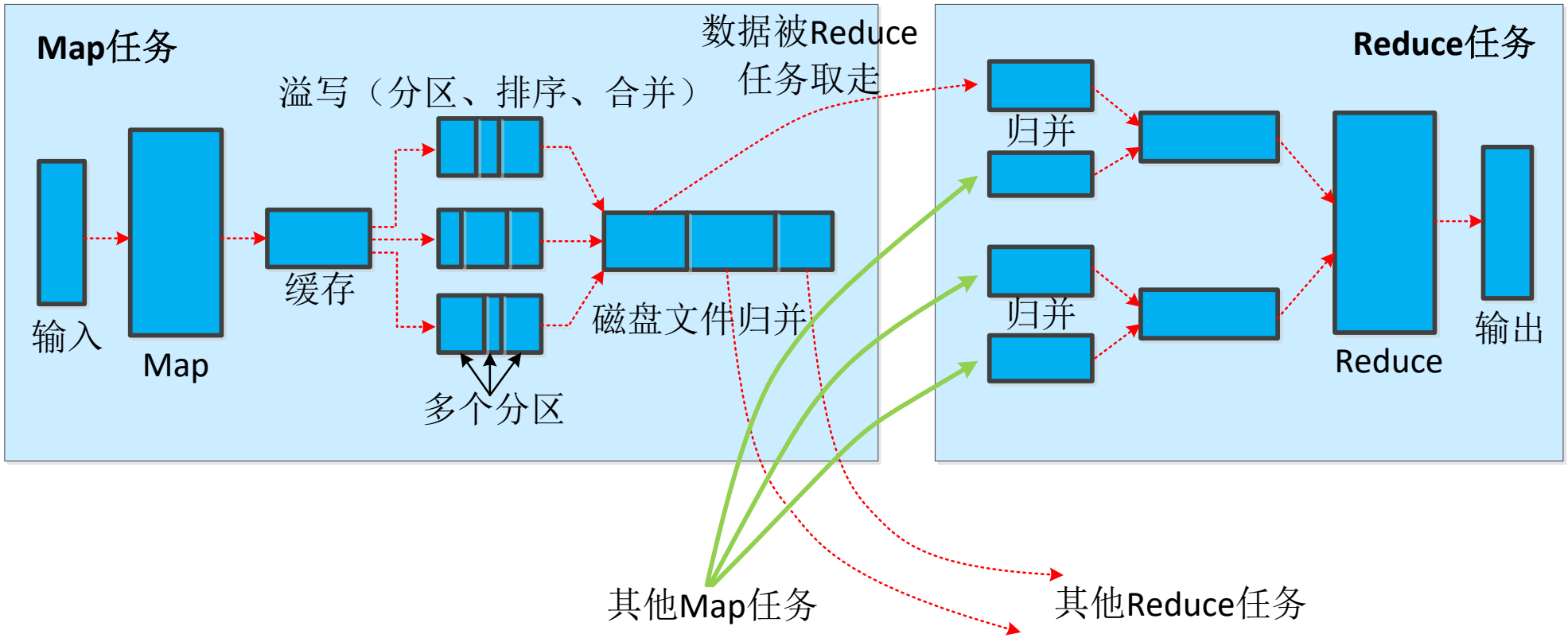
•Reduce任务是一个数据聚合的步骤，数量默认为1。而使用过多的Reduce任务则意味着复杂的shuffle，并使输出文件的数量激增。

•一个job的ReduceTasks数量是通过mapreduce.job.reduces参数设置

•也可以通过编程的方式，调用Job对象的setNumReduceTasks()方法来设置

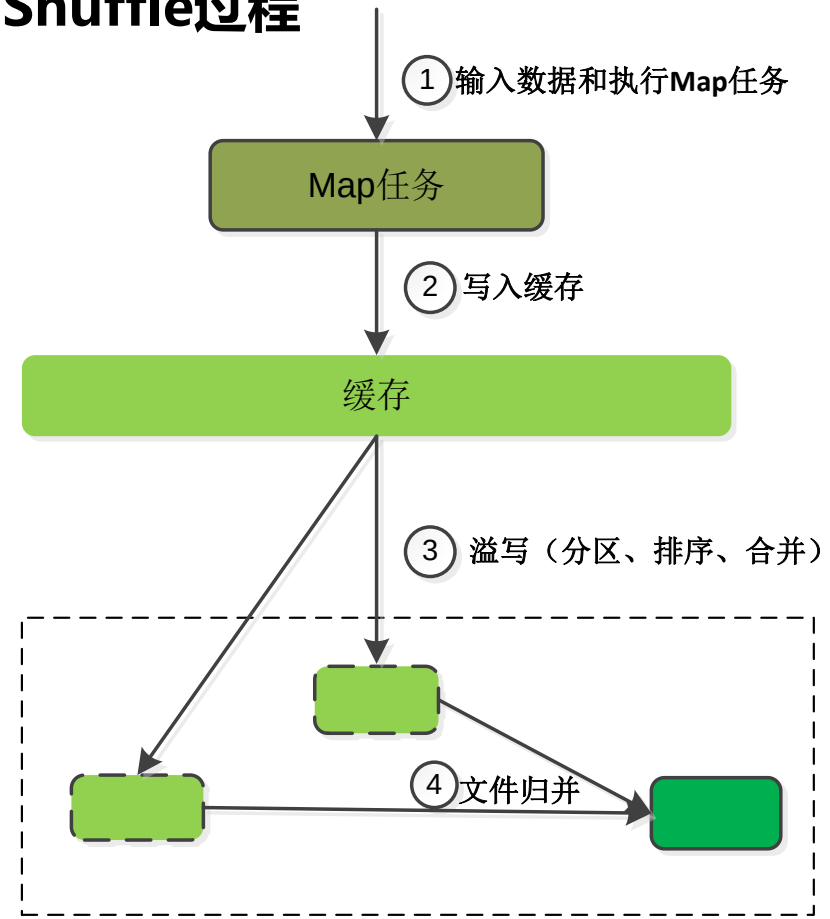
•一个节点Reduce任务数量上限由mapreduce.tasktracker.reduce.tasks.maximum设置（默认2）。

Shuffle过程


Map任务
溢写（分区、排序、合并）
数据被Reduce
任务取走
Reduce任务
归并
输入
Map
缓存
磁盘文件归并
归并
Reduce
输出
多个分区
其他Map任务
其他Reduce任务

Shuffle过程

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

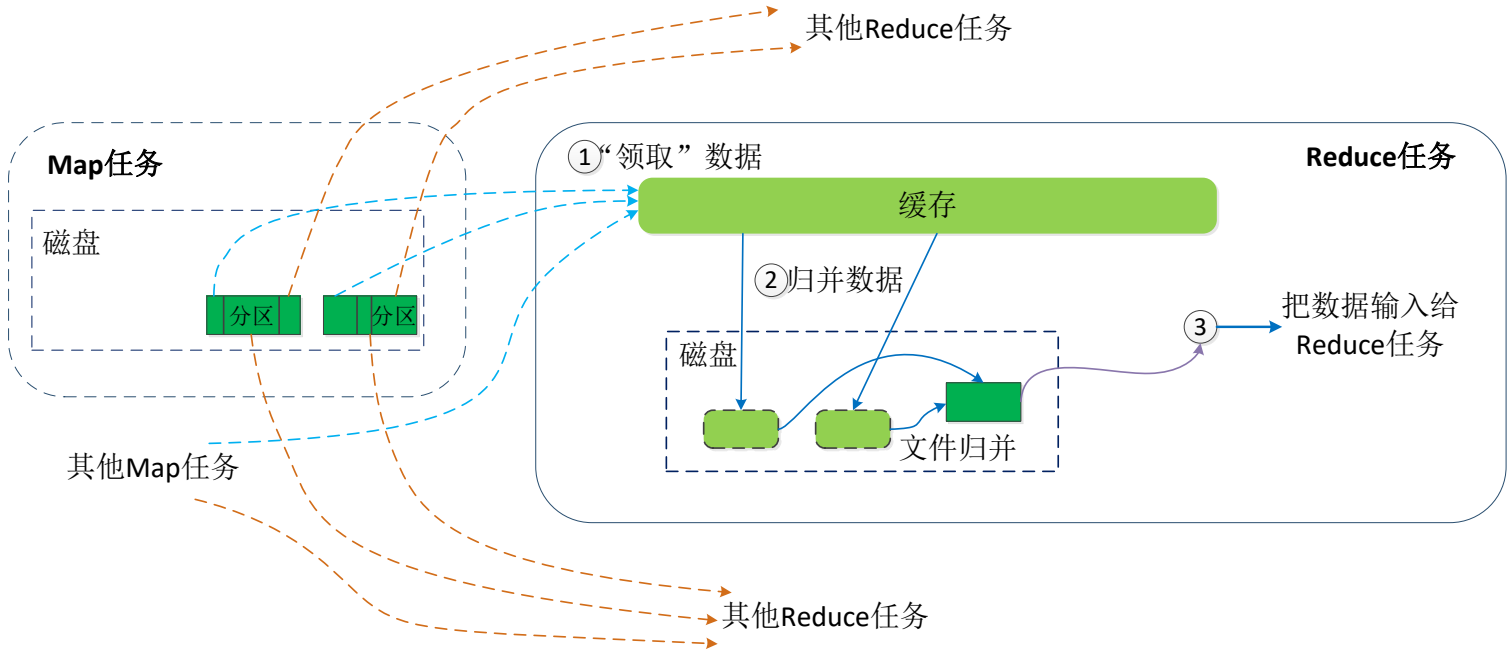
- 在Map任务全部结束之前进行归并
- 归并得到一个大的文件，放在本地磁盘
- 文件归并时，如果溢写文件数量大于预定值（默认是3）则可以再次启动Combiner，少于3不需要

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>>

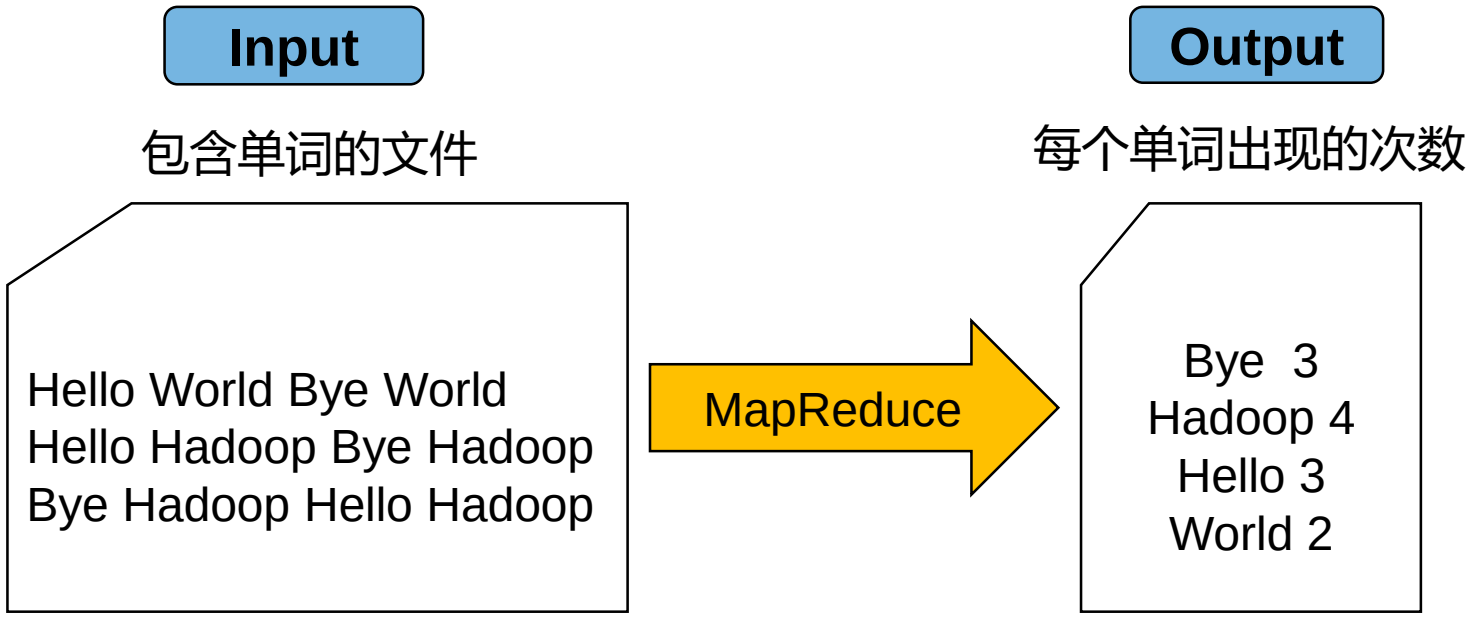
Shuffle过程

Reduce端的Shuffle过程

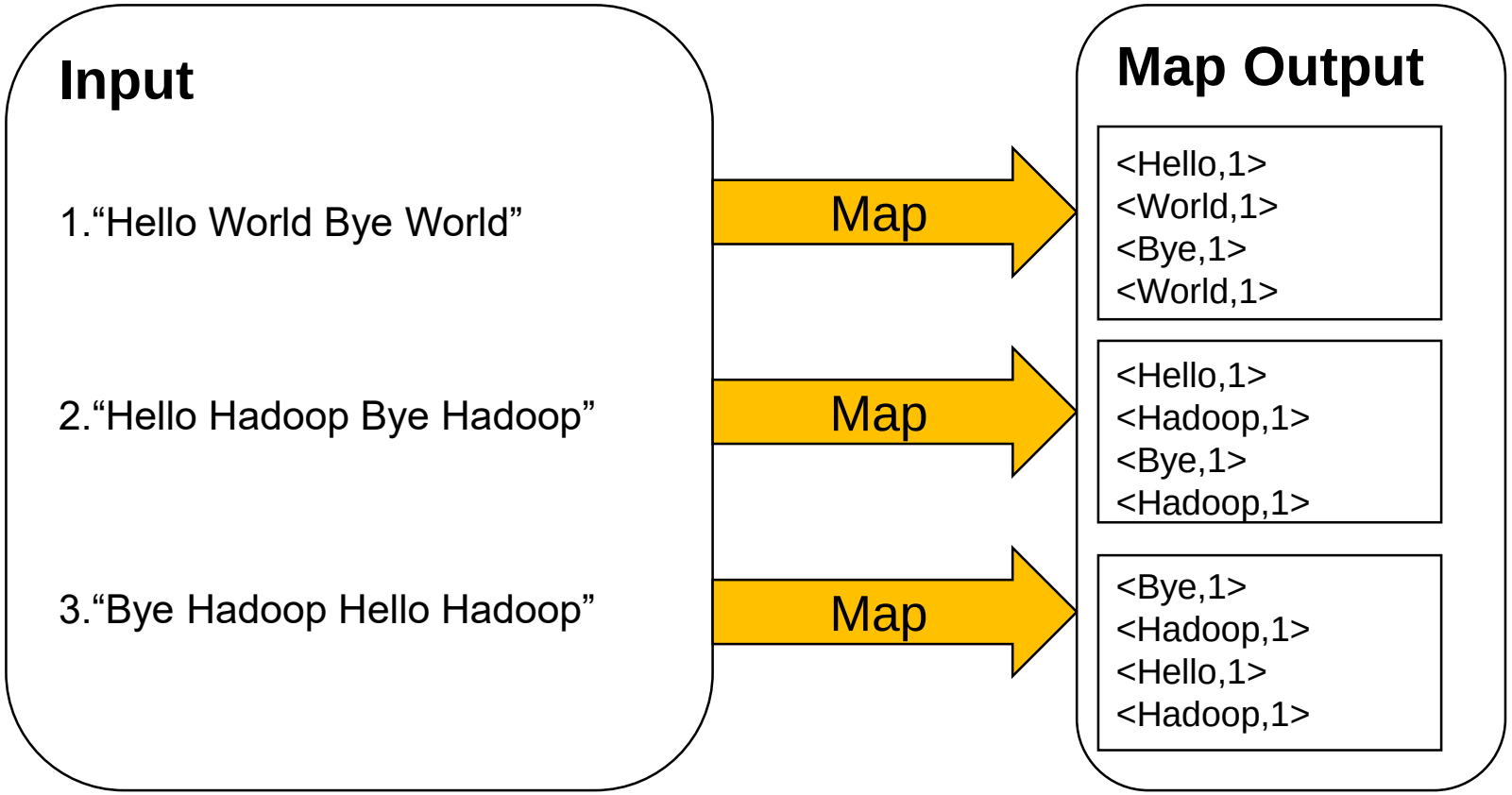
- Reduce任务通过RPC（Remote Procedure Call，远程过程调用）询问Map任务是否已经
- 完成，若完成，则领取数据
- Reduce领取数据先放入缓存，来自不同Map机器，先归并，再合并，写入磁盘
- 多个溢写文件归并成一个或多个大文件，文件中的键值对是排序的
- 当数据很少时，不需要溢写到磁盘，直接在缓存中归并，然后输出给Reduce



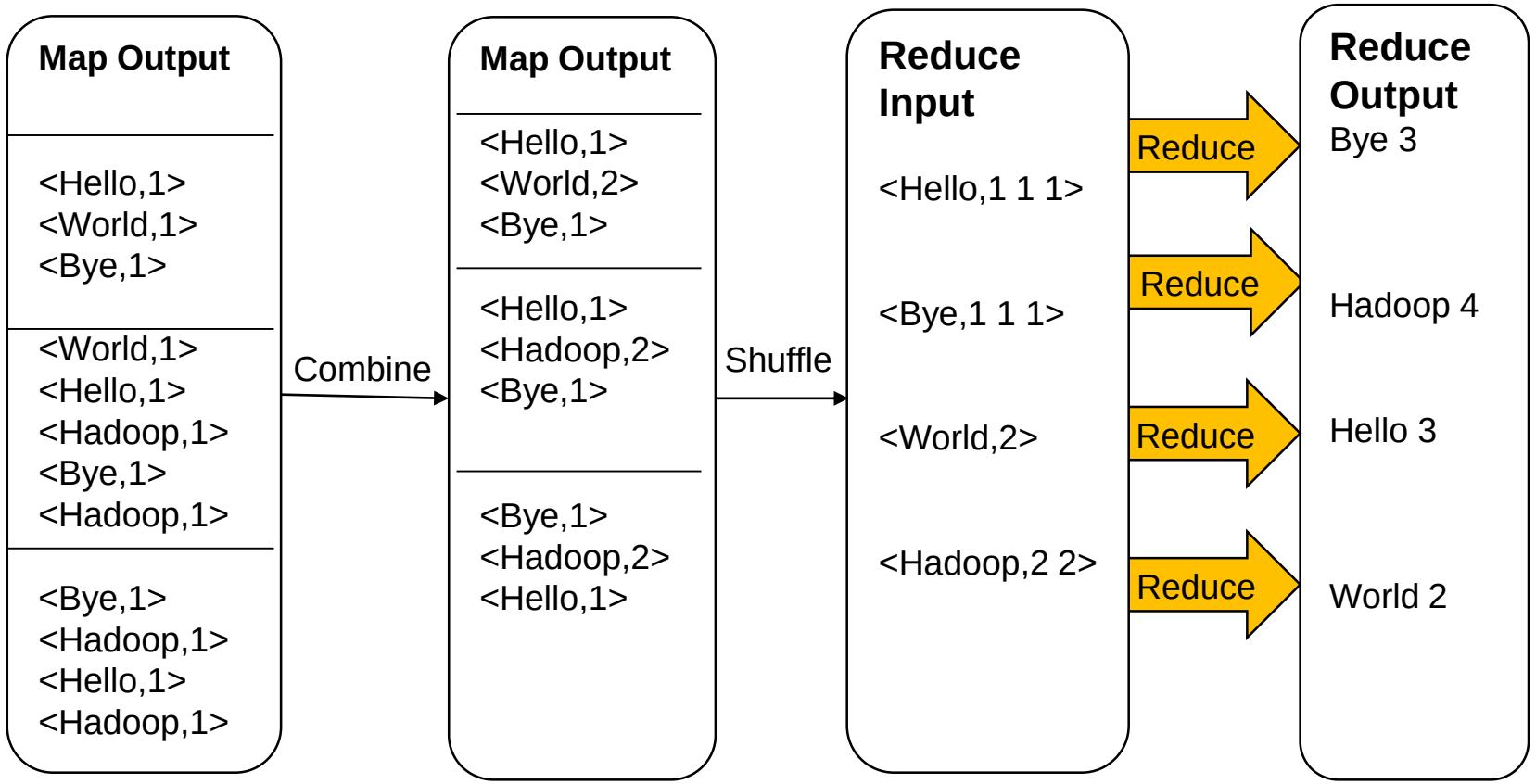
WordCount程序



WordCount的Map过程



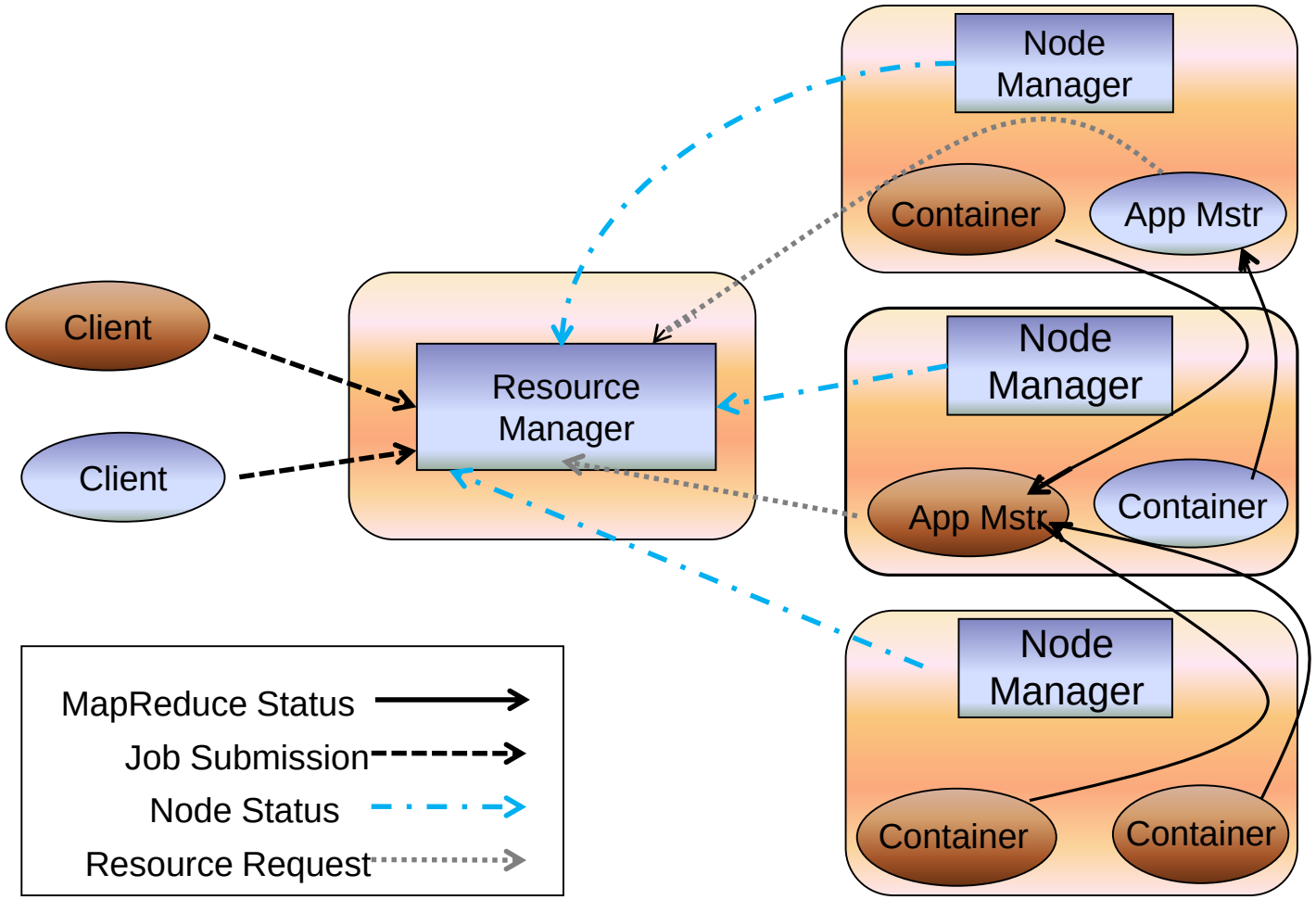
WordCount的Reduce过程



YARN

- Apache Hadoop YARN (Yet Another Resource Negotiator，另一种资源协调者) 是一种新的 Hadoop 资源管理器，它是一个通用资源管理系统，可为上层应用提供统一的资源管理和调度，它的引入为集群在利用率、资源统一管理和数据共享等方面带来了巨大好处。
- Yarn主要由ResourceManager、NodeManager、ApplicationMaster和Container等几个组件构成。

YARN的组件架构


Node Manager
Container
App Mstr
Client
Resource Manager
Client
Node Manager
App Mstr
Container
Node Manager
Container
Container
MapReduce Status
Job Submission
Node Status
Resource Request

ResourceManager

ResourceManager 内部主要有两个组件：

1、Scheduler：这个组件的唯一功能就是给提交到集群的应用程序分配资源，并且对可用的资源和运行的队列进行限制。Scheduler并不对作业进行监控；

2、ApplicationsManager（AsM):这个组件用于管理整个集群应用程序的application masters，负责接收应用程序的提交；为application master启动提供资源；监控应用程序的运行进度以及在应用程序出现故障时重启它。

ApplicationMaster

ApplicationMaster是应用程序级别的，每个ApplicationMaster管理运行在YARN上的应用程序。YARN 将 ApplicationMaster看做是第三方组件，ApplicationMaster负责和ResourceManager scheduler协商资源，并且和NodeManager通信来运行相应的task。ResourceManager 为 ApplicationMaster 分配Container，这些Container将会用来运行task。ApplicationMaster 也会追踪应用程序的状态，监控Container的运行进度。当Container运行完成， ApplicationMaster 将会向 ResourceManager 注销这个Container；如果是整个作业运行完成，其也会向 ResourceManager 注销自己，这样这些资源就可以分配给其他的应用程序使用了。

NodeManager

NodeManager是YARN中每个节点上的代理，它管理Hadoop集群中单个计算节点，根据相关的设置来启动Container。NodeManager会定期向ResourceManager发送心跳信息来更新其健康状态。同时其也会监督Container的生命周期管理，监控每个Container的资源使用（内存、CPU等）情况，追踪节点健康状况，管理日志和不同应用程序用到的附属服务（auxiliary service）。

总体来说， NodeManager有以下作用

（1）管理单个节点上的资源

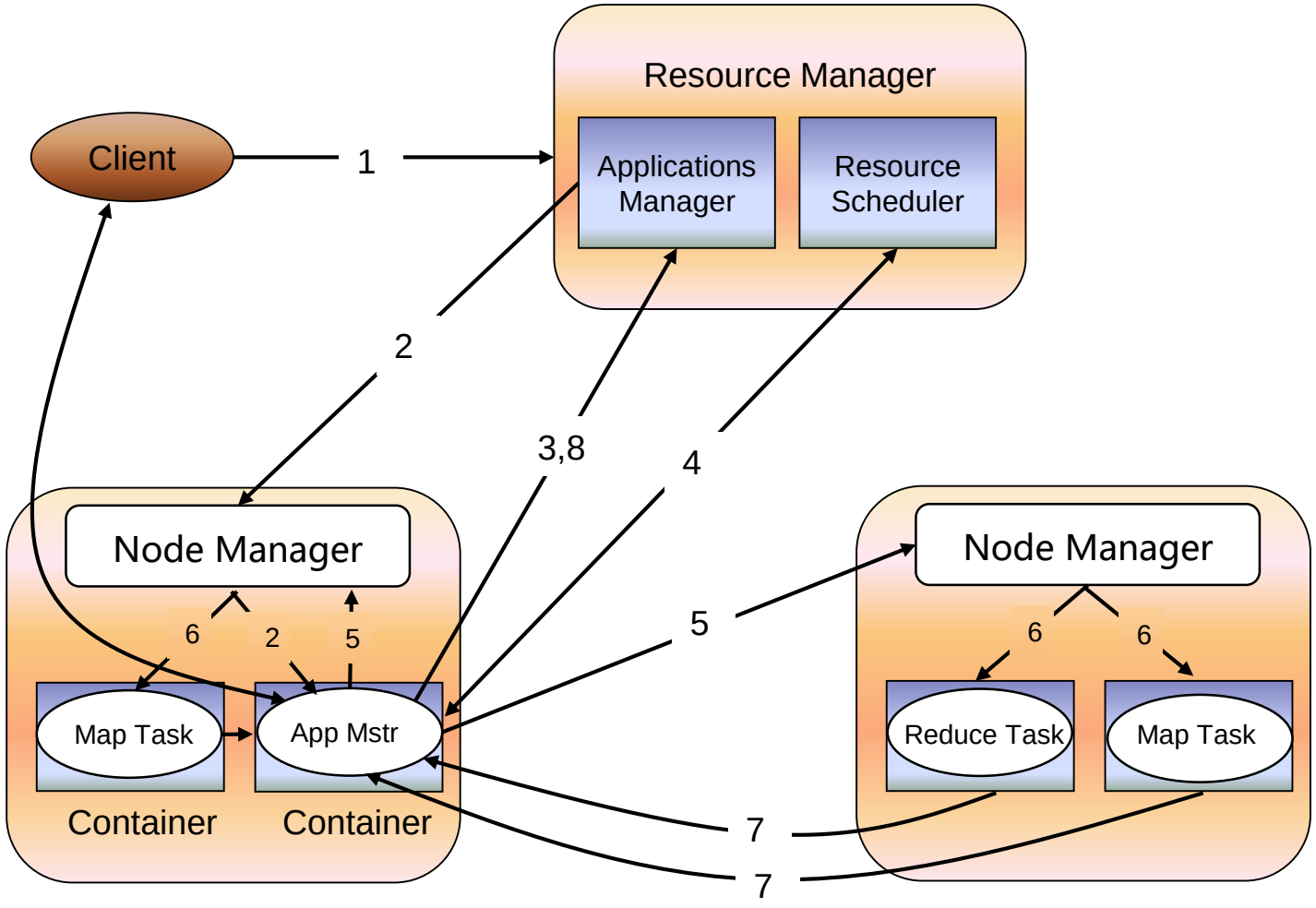
（2）处理来自ResourceManager的命令

（3）处理来自ApplicationMaster的命令

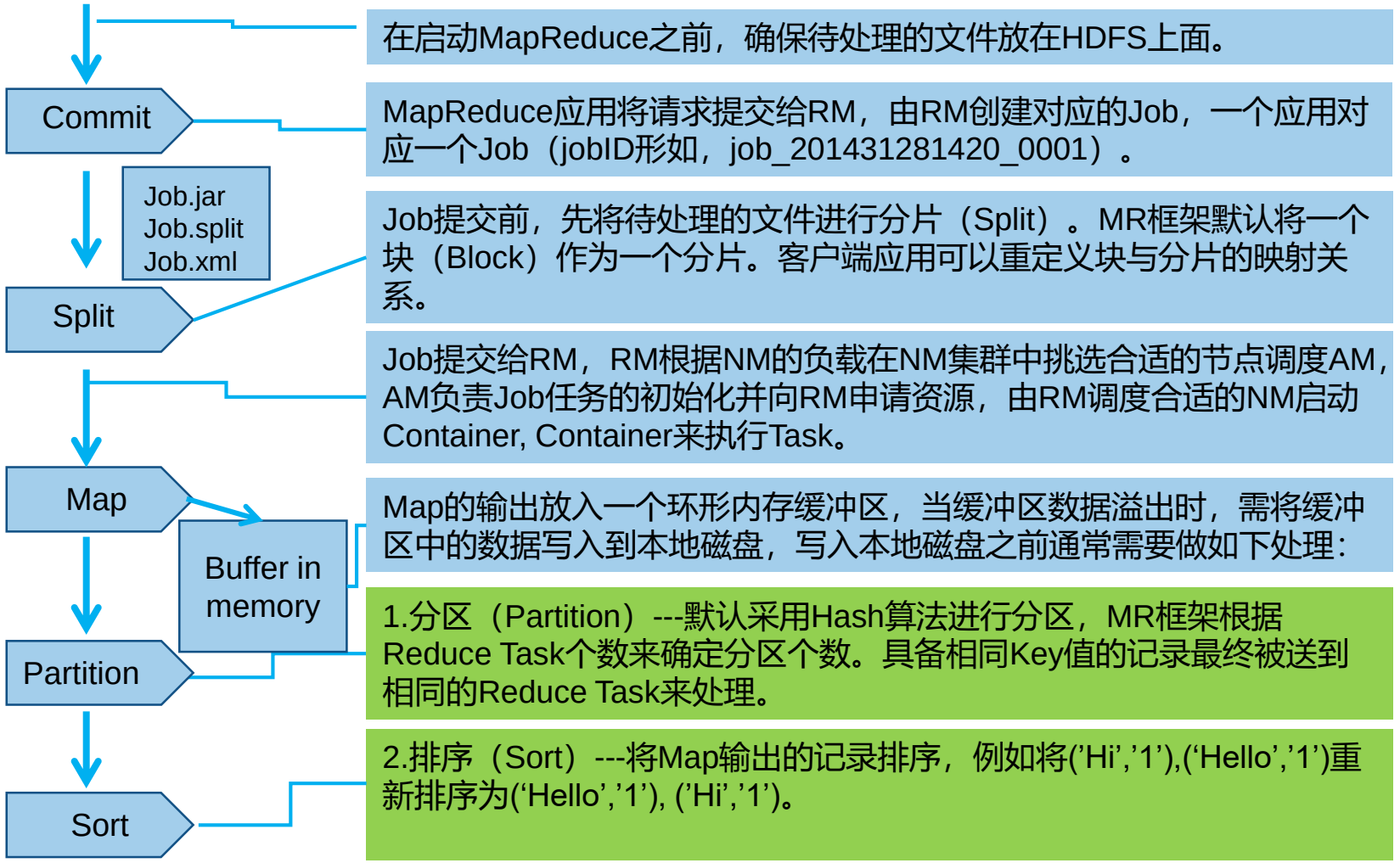
Container

Container是与特定节点绑定的，其包含了内存、CPU磁盘等逻辑资源。容器是由ResourceManager scheduler 服务动态分配的资源构成。容器授予 ApplicationMaster 使用特定主机的特定数量资源的权限。ApplicationMaster 也是在容器中运行的，其在应用程序分配的第一个容器中运行。

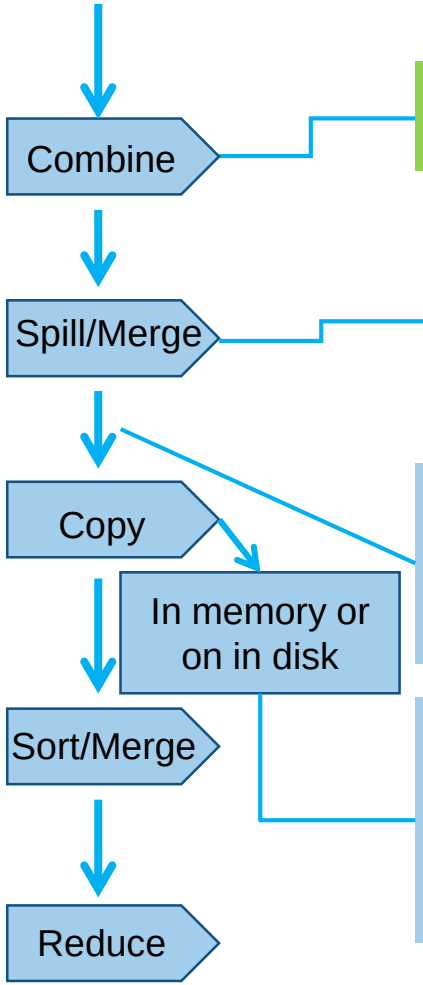
MapReduce On YARN任务调度流程



MapReduce过程



MapReduce过程



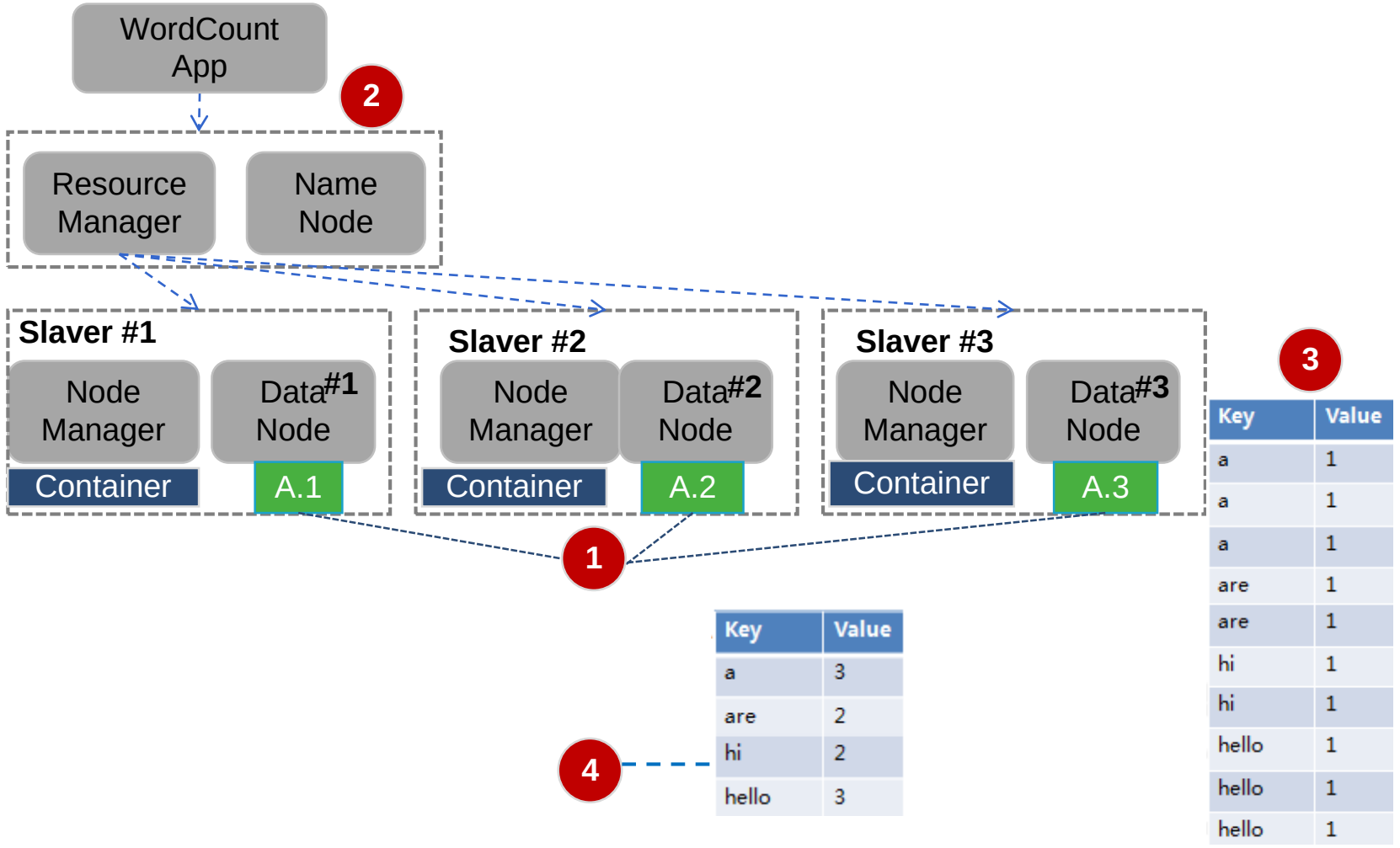
3.合并(Combine)---这个动作MR框架默认是可选的。例如将('Hi','1'), ('Hi','1'),('Hello','1'), (Hello','1')进行合并操作为('Hi','2'), ('Hello','2')。

4.归并(Merge)---Map Task在处理后会产生很多的溢写文件，这时需将多个溢出文件进行归并处理，生成一个经过分区和排序的溢写文件（MOF:MapOutFile）。为减少写入磁盘的数据量，MR支持对MOF进行压缩后再写入。

通常在Map Task任务完成MOF输出进度到3%时启动Reduce，从各个Map Task获取MOF文件。前面提到Reduce Task个数由客户端决定，Reduce Task个数决定MOF文件分区数。因此Map Task输出的MOF文件都能找到相对应的Redcue Task来处理。

前面提到的MOF文件是经过排序处理的。当Reduce Task接收的数据量不大时，则直接存放在内存缓冲区中，随着缓冲区文件的增多，MR后台线程将它们合并成一个更大的有序文件，这个动作是Reduce阶段的Merge操作，过程中会产生许多中间文件，最后一次合并的结果直接输出到用户自定义的Reduce函数。

WordCount


WordCount App
2
Resource Manager
Name Node
Slaver #1
Node Manager
Data Node #1
Container
A.1
Slaver #2
Node Manager
Data Node #2
Container
A.2
Slaver #3
Node Manager
Data Node #3
Container
A.3
1
3
4
Key
Value
a
1
a
1
a
1
are
1
are
1
hi
1
hi
1
hello
1
hello
1
hello
1
Key
Value
a
3
are
2
hi
2
hello
3

MapReduce程序开发

Hadoop数据类型

数据类型	hadoop数据类型	Java数据类型
布尔型	BooleanWritable	boolean
整型	IntWritable	int
长整型	LongWirteble	long
浮点数	FloatWritable	float
双字节数值	DoubleWritable	double
单字节数值	ByteWritable	byte
使用UTF8格式存储的文本	Text	String
当<key,value>中key或value为空时使用	NullWritable	

Map函数

通过继承类Mapper来实现Map处理逻辑。

```
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);
        }
    }
}
```

Map输入类型为<Object,Text>

Map输出类型为<Text,IntWritable>

Reduce函数

Reduce的实现需要继承自类Reducer，并实现其接口。

```
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);
    }
}
```

Main方法

```
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”); //设置环境参数
    job.setJarByClass(WordCount.class);             //设置整个程序的类名
    job.setMapperClass(TokenizerMapper.class);      //添加Mapper类
    job.setCombinerClass(IntSumReducer.class);      //添加Combiner类
    job.setReducerClass(IntSumReducer.class);       //添加Reducer类
    job.setOutputKeyClass(Text.class);              //设置输出键类型
    job.setOutputValueClass(IntWritable.class);     //设置输出值类型
    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);
}
```