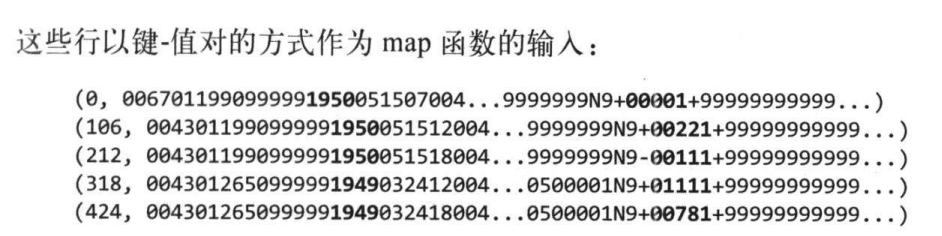
# 2.1 MapReduce 原理

## 2.1.1 Map和Reduce（重点）

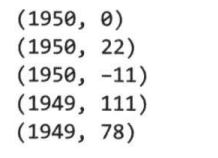
1.**MapReduce任务过程分为两个处理阶段**：**map阶段和reduce阶段**。**每阶段都以键-值对作为输入和输出**，其类型由程序员选择。程序员还需要编写两个函数**Map函数和Reduce函数**。

2.以NCDC的气象数据为例。**Map阶段的输入是NCDC原始数据**。我们选择文本格式作为输入格式，将数据集的每一行作为文本输入。**键是某一行起始位置相对于文件起始位置的偏移量**，Map函数并不需要这个信息，所以将其忽略。

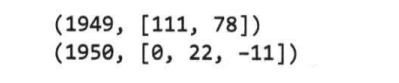


3.**Map函数的意义很简单**，由于我们只对**年份和气温感兴趣**，所以只需要取出这**两个字段的数据**。在本例中，M**ap函数只是一个数据准备阶段，通过这种方式来准备数据，使得Reduce函数能够继续对它进行处理**；即找出每年的最高气温。**Map函数还是一个比较适合去除已损记录的地方**：此处，我们筛掉缺失的、可疑的或错误的气温数据。

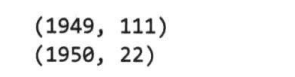
4.Map函数的功能仅限于提取**年份和气温信息**，并将他们作为**输出**（气温值已转化为整数）：



5.Map函数的输出经由MapReduce框架处理后，最后发送到Reduce函数。**这个处理过程基于键-值对进行排序和分组（Shuffle）**。因此，在这一示例中，reduce函数看到的是如下的输入：

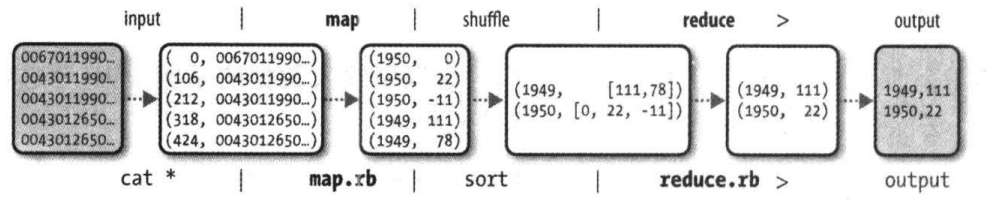


6.**每个年份后紧跟着一系列气温数据**。reduce函数现在要做的是遍历整个列表并从中找出最大的读数：



7.这就是最终的输出结果，每一年的全球最高气温纪录。

8.整个MapReduce数据流如下图所示。在图的底部是Unix管线，用于模拟整个MapReduce过程。



### Java MapReduce

1.明白MapReduce程序的工作原理之后，下步就是代码实现。需要三样东西：一个map函数，一个reduce函数和一些用来运行作业的代码。map函数由Mapper类来表示，后者声明一个抽象的map()方法。

package org.wordcount;  
  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.LongWritable;  
import org.apache.hadoop.mapreduce.Mapper;  
import org.apache.hadoop.io.Text;  
  
import java.io.IOException;  
import java.util.StringTokenizer;  
  
public class DataMapper extends Mapper<LongWritable, Text, Text, IntWritable> {  
 private static final int MISSING = 9999;  
  
 @Override  
 protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text, IntWritable>.Context context) throws IOException, InterruptedException {  
 String line = value.toString();  
 String year = line.substring(15, 19);  
 int airTemperature;  
 if (line.charAt(87) == '+') {  
 // parseInt不能前导加'+'符号  
 airTemperature = Integer.parseInt(line.substring(88, 92));  
 }  
 else {  
 airTemperature = Integer.parseInt(line.substring(87, 92));  
 }  
 String quality = line.substring(92, 93);  
 if (airTemperature != MISSING && quality.matches("[01459]")) {  
 context.write(new Text(year), new IntWritable((airTemperature)));  
 }  
 }  
}

2.这个**Mapper类是一个泛型类型**，它有四个形参类型，分别制定map函数的**输入键、输入值、输出键和输出值的类型**。在这个例子中，**输入键是一个长整数偏移量**，**输入值是一行文本**，**输出键是年份**，**输出值是气温**（整数）。

3.Hadoop本身提供了一套**可优化网络序列化传输的基本类型**，而不直接用Java内嵌的类型。这些类型都在**org.apache.hadoop.io**包中。这里使用LongWritable类型(相当于Java的Long类型)、Text类型(相当于Java中的String类型)和IntWritable类型(相当于Java中Integer类型)。

4.**map()方法的输入是一个键和一个值**。首先将包含有一行输入的Text值转换为Java中String类型，方便我们使用substring()方法提取感兴趣的列。

5.map()方法还提供**Context实例用于输出内容的写入**。在这种情况下，我们将年份数据按Text对象进行读/写（因为我们把年份当作键），将气温值封装在IntWritable类型中。只有气温数据不缺并且所对应质量代码显示为正确的气温读数时，这些数据才会被写入输出记录中。

6.以类似方法用Reducer来定义reduce函数。

package org.wordcount;  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.mapreduce.Reducer;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.io.LongWritable;  
  
import java.io.IOException;  
  
public class DataReducer extends Reducer<Text, IntWritable, Text, IntWritable> {  
 @Override  
 protected void reduce(Text key, Iterable<IntWritable> values, Reducer<Text, Iterable<IntWritable>, Text, IntWritable>.Context context) throws IOException, InterruptedException {  
 int maxValue = Integer.MIN\_VALUE;  
 for (IntWritable value : values) {  
 maxValue = Math.max(value.get(), maxValue);  
 }  
 context.write(key, new IntWritable(maxValue));   
 }  
}

7.同样，reduce函数也有四个形式参数类型用于指定输入和输出类型。**reduce函数的输入类型必须匹配map函数的输出类型**：即Text类型和IntWritable类型。在这种情况下，**reduce函数的输出类型也必须是Text和IntWritable类型**，分别输出年份及其最高气温。这个最高气温是通过循环比较每个气温与当前所知最高气温所得到的。

8.第三部分代码负责运行MapReduce作业。

package org.wordcount;  
  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.mapreduce.Job;  
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;  
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.io.LongWritable;  
import java.io.FileInputStream;  
  
public class Driver {  
 public static void main(String[] args) throws Exception {  
 if (args.length != 2) {  
 System.err.println("Usage: MaxTemperature <input path> <output path>");  
 System.exit(-1);  
 }  
 Job job = new Job();  
 job.setJarByClass(Driver.class);  
 job.setJobName("Max Temperature");  
  
 FileInputFormat.addInputPath(job, new Path(args[0]));  
 FileOutputFormat.setOutputPath(job, new Path(args[1]));  
 job.setMapperClass(DataMapper.class);  
 job.setReducerClass(DataReducer.class);  
  
 job.setOutputKeyClass(Text.class);  
 job.setOutputValueClass(IntWritable.class);  
 System.exit(job.waitForCompletion(true) ? 0 : 1);  
 }  
}

9.Job对象指定作业执行规范。我们可以用它来控制整个作业的运行。**我们在Hadoop集群上运行这个作业时，要把代码打包成一个JAR文件**(Hadoop在集群上发布这个文件)。不必明确指定JAR文件的名称，在Job对象的setJarByClass()方法中传递一个类即可，Hadoop利用这个类来查找包含它的JAR文件，进而找到相关的JAR文件。

10.构造Job对象之后，**需要指定输入和输出的路径**。调用FileInputFormat类的静态方法addInputPath()来定义输入数据的路径，这个路径可以是单个文件、一个目录(此时，将目录下所有文件当作输入)或符合特定文件模式的一系列文件。由函数名可知，**可以多次调用addInputPath()来实现多路径的输入**。

11.调用FileOutputFormat类中的静态方法setOutputPath()来指定输出路径（只能有一个输出路径）。这个方法指定的是reduce函数输出文件的写入目录。**在运行作业前该目录是不应该存在的，否则Hadoop会报错并拒绝运行作业**。这种预防措施的目的是**防止数据丢失**（防止运行的作业被意外覆盖）。

12.接着，通过setMapperClass()和setReducerClass()方法指定要用的map类型和reduce类型。**setOutputKeyClass()和setOutputValueClass()方法控制reduce函数的输出类型**，并且必须和Reduce类产生的相匹配。

13.**map函数的输出类型默认情况下和reduce函数是相同的**，因此如果mapper产生出和reducer相同的类型时(如本例所示)，不需要单独设置。但是，如果不同，则必须通过setMapOutputKeyClass()和setMapOutputValueClass()方法来设置map函数的输出类型。

14.输入的类型通过输入格式在空值，在例子中没有设置，默认使用TextInputFormat(文本输入格式)。

15.在设置定义map和reduce函数的类之后，可以开始运行作业。Job中的**waitForCompletion()方法提交作业并等待执行完成**。该方法唯一的参数是一个标识，指示是否已经生成详细输出。当表示为true（成功）时，会把作业进度写入控制台。

16.waitForCompletion()方法返回一个布尔值，表示运行的成(true)败(false)，这个布尔值被转换为程序的退出代码0或者1。

17.如果**调用Hadoop命令的第一个参数是类名**，Hadoop会启动一个JVM来运行这个类。该Hadoop命令将Hadoop库（及其依赖关系）添加到类路径中，同时也能获得Hadoop配置信息。为了将应用类添加到类路径中，我们必须定义HADOOP\_CLASSPATH环境变量，然后由Hadoop脚本来执行相关操作。

18.**输出数据写入output目录，其中每个reducer都有一个输出文件**。

# 2.2 MapReduce横向扩展

## 2.2.1 数据流（重点）

1.MapReduce作业(Job)**是客户端需要执行的一个工作单元**；它包括**输入数据**、**MapReduce程序**和**配置信息**。Hadoop将作业分成若**干个任务**(task)来执行，其中包括两类任务：**map任务和reduce任务**。这些任务运行在**集群的节点**上，**并通过Yarn进行调度**。如果一个任务失败，它将在**另一个不同的节点上自动重新调度运行**。

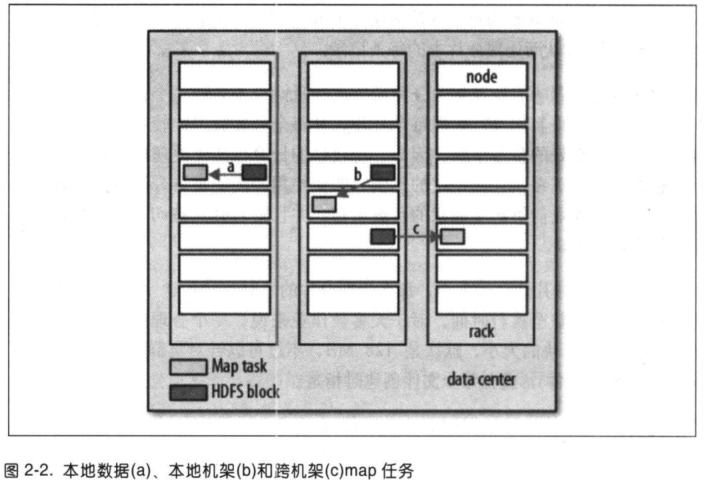
2.Hadoop将**MapReduce的输入数据划分成等长的小数据块**，称为**输入分片**(input split)或简称“分片”。**Hadoop为每个分片构建一个map任务**，并由该任务来运行用户**自定义的map函数从而处理分片中的每条记录**。

3.拥有**更多分片**，**意味着处理每个分片所需要的时间少于处理整个输入数据所花的时间**。因此，如果我们**并行处理每个分片**，且每个**分片数据比较小**，那么**整个处理的过程将获得很好的负载均衡**，并且随着**分片被切分得很细**，**负载平衡的质量会越高**。

4.另一方面，如果**分片切分得太小**，**那么管理分片的总时间和构建map任务的总时间将决定作业的整个执行时间**。对大多数作业来说，**一个合理的分片大小趋向于一个HDFS的块大小，默认是128MB**，不过可以针对集群调整这个默认值，或在每个文件创建时被指定。

5.**Hadoop在存储有输入数据**（HDFS中的数据）的节点上运行map任务，**可以获得最佳性能**，因为**它无需使用宝贵的集群带宽资源**，**即数据本地化优化**。

6.但是，**有时对于一个map任务的输入分片来说**，**存储该分片的HDFS数据块复本的所有节点可能正在运行其他map任务**，此时作业**调度需要从某一数据块所在的机架中的一个节点寻找一个空闲的map槽(slot)来运行该map任务分片**。仅仅在非常偶然的情况下（该情况基本不会发生），**会使用其他机架中的节点运行该map任务，这导致机架与机架之间的网络传输**。下图揭示了其三种可能性。



7.那么**最佳分片的大小与块大小相同的原因**：

因为**它确保可以存储在单个节点上的最大输入块的大小**。**如果分片跨越两个数据块，那么对于任何一个HDFS节点，基本上都不可能同时存储这两个数据块，因此分片中的部分数据需要通过网络传输到map任务运行的节点**。与使用本地数据运行整个map任务相比，这种方法显然效率更低。

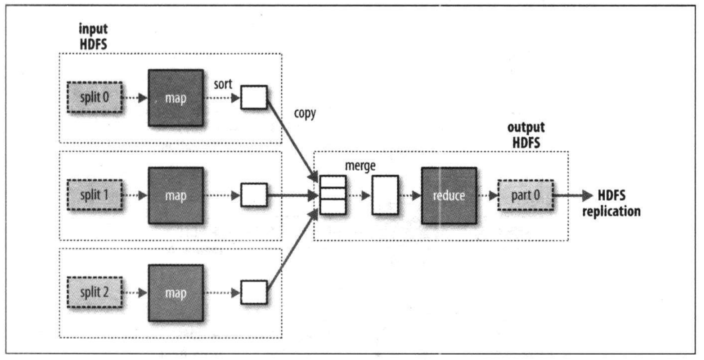
8.**map任务将其输出写入本地硬盘，而非HDFS**。

其原因：**map输出是中间结果**：该**中间结果由reduce任务处理后才能产生最终结果**，**而且一旦作业完成，map的输出结果就可以删除**。因此，如果把它存储在HDFS中并实现备份，会小题大做。如果运行**map任务的节点在将map中间结果传送给reduce任务之前失败**，**Hadoop将在另一个节点上重新运行这个map任务以再次构建map中间结果**。

9.**reduce任务并不具备数据本地化的优势**，**单个reduce任务的输入通常来自于所有mapper的输出**。在本例中，仅有一个reduce任务，**其输入是所有map任务的输出**。因此，**排过序的map输出需通过网络传输发送到运行reduce任务的节点**。**数据在reduce端合并，然后由用户定义的reduce函数处理**。**reduce的输出通常存储在HDFS中以实现可靠存储**。

10.对于**reduce输出的每个HDFS块**，**第一个复本存储在本地节点上**，其**他复本出于可靠性考虑存储在其他机架的节点中**。因此，**将reduce的输出写入HDFS确实需要占用网络带宽**，**但这与正常的HDFS管线写入的消耗一样**。

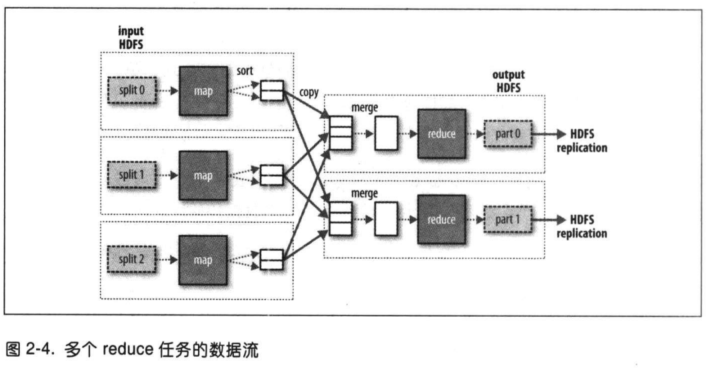
11.一个**reduce任务的完整数据流如下所示**。虚线框的代表节点，虚线箭头表示节点内部的数据传输，而实线箭头表示不同节点的数据传输。



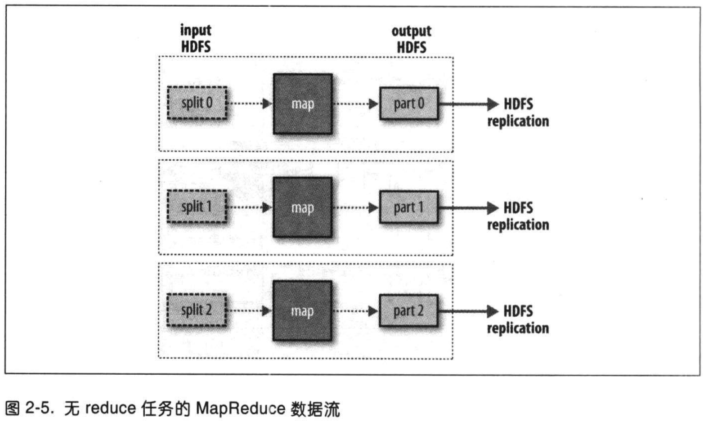
12.**reduce任务的数量并非由输入数据的大小决定，相反是独立指定的**。

13.如果**有好多个reduce任务**，每个**map任务就会针对输出进行分区**(partition)，即为每个**reduce任务建立一个分区**。**每个分区有许多键（及其所对应的值）**，**但每个键对应的键值对记录都在同一分区中**。分区可由用户定义的分区函数控制，**但通常用默认的partitioner通过哈希函数来分区**，很高效。

14.一般情况下，**多个reduce任务的数据流如下图所示**。该图清楚地表明了为什么map任务和reduce任务之间的数据流称为**shuffle**(混洗)，因为每个**reduce任务的输入都来自许多map任务**。shuffle一般比下图所示更复杂，而且调整混洗参数对作业总执行时间影响非常大。



15.最后，当**数据处理可以完全并行**（无需混洗时），**可能会出现无reduce任务的情况**。在这种情况下，唯一的非本地节点数据传输时map任务将结果写入HDFS（如下图所示）。



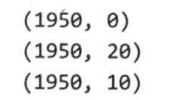
16.所以**发生排序的部分在map函数**，**合并的位置在reduce函数**。

## 2.2.2 combiner函数（重点）

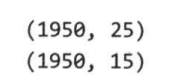
1.**集群上的可用带宽限制了MapReduce作业的数量**，因此尽量**避免map和reduce任务之间的数据传输是有利的**。

2.**Hadoop允许用户针对map任务的输出指定一个combiner**(就像mapper和reducer一样)，**combiner函数（预聚合函数）的输出作为reduce函数的输入**。由于combiner属于优化方案，**所以Hadoop无法确定要对一个指定的map任务输出记录调用多少次combiner**（如果需要）。换而言之，不管调用combiner多少次，0次、1次或多次，reducer的输出结果都是一样的。

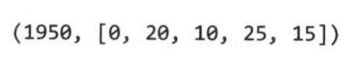
3.**combiner的规则制约着可用的函数类型**。假设以前计算最高气温的例子，1950年的读数由两个map任务处理（因为他们存储在不同的分片中）。假设第一个map的输出如下：



第二个map输出如下：



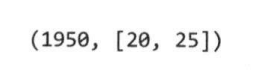
reduce函数被调用时，输入如下：



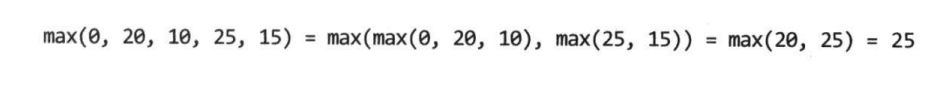
因为25为该列数据中最大的，所以它的输出如下：

image.png

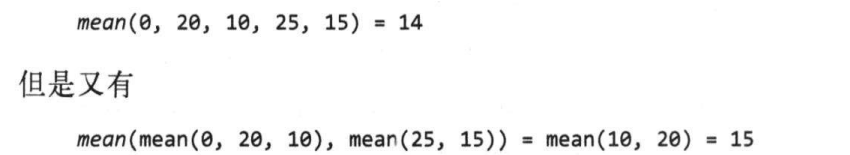
我们可以像使用reduce函数那样，使用combiner找出每个map任务输出结果中最高气温。如此一来，reduce函数调用时将被传入以下数据：



reduce输出的结果和以前一样。更简单地说，可以通过下面的表达式来说明气温数值的函数调用：



并非所有函数都具有该属性。例如，计算平均气温时，就不能用求平均函数mean作为combiner函数。因为：



4.combiner函数不能取代reduce函数的原因：

**最终我们仍然需要reduce函数来处理map输出中具有相同键的记录**。但**combiner函数能帮助减少mapper和reducer之间的数据传输量**，因此，单纯就这点而言，在MapReduce作业中是否使用combiner函数还是值得斟酌的。

5.运行分布式的MapReduce作业：

我们编写的Java MapReduce程序不用修改便可以在一个完整的数据集上直接运行。这是MapReduce的优势：**可以根据数据量的大小和硬件规模进行扩展**。

### Java MapReduce中指定combiner

1.在Java MapReduce程序，combiner是通过Reducer类来定义的，并且在这个例子中，**它的实现与MaxTemperatureReducer中的reduce函数相同**。唯一的改动实在Job中设置combiner类。

package org.wordcount;  
  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.mapreduce.Job;  
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;  
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.io.LongWritable;  
import java.io.FileInputStream;  
  
public class Driver {  
 public static void main(String[] args) throws Exception {  
 if (args.length != 2) {  
 System.err.println("Usage: MaxTemperature <input path> <output path>");  
 System.exit(-1);  
 }  
 Job job = new Job();  
 job.setJarByClass(Driver.class);  
 job.setJobName("Max Temperature");  
  
 FileInputFormat.addInputPath(job, new Path(args[0]));  
 FileOutputFormat.setOutputPath(job, new Path(args[1]));  
 job.setMapperClass(DataMapper.class);  
 job.setCombinerClass(DataReducer.class);  
 job.setReducerClass(DataReducer.class);  
  
 job.setOutputKeyClass(Text.class);  
 job.setOutputValueClass(IntWritable.class);  
 job.waitForCompletion(true);  
 }  
}

## 2.2.3 Hadoop Streaming（了解补充）

1.Hadoop提供了MapReduce的API，允许使用非Java的其他语言来编写map和reduce函数。**Hadoop Streaming使用Unix标准流作为Hadoop和应用程序之间的接口，所以我们可以使用任何编程语言通过标准输入/输出来写MapReduce程序**。

2.Streaming天生适合用于文本处理。map的输入数据通过标准输入流传递给map函数，并且一行一行地传输，最后将结果写到标准输出。map输出的键值对是以一个制表符分隔的行，reduce函数的输入格式与之相同（通过制表符来分割键值对）并通过标准输入流进行传输。reduce函数从标准输入流中读取输入行，该输入已由Hadoop框架根据键排序，最后将结果写入标准输出。