第4讲 MapReduce类型与输入/输出格式

辜希武
IDC实验室
1694551702@qq.com

主要内容

- MapReduce数据处理模型非常简单，map和reduce函数的输入输出都是键/值对(key/value pair)，这一讲介绍各种类型的数据如何在MapReduce中使用
  - MapReduce的类型
  - MapReduce输入格式
  - MapReduce输出格式

MapReduce的类型

- MapReduce的map、reduce函数的通用形式如下：

map: (K1,V1)->list<K2,V2>

reduce: (K2,list(V2))->list<K3,V3>

- 其中K1，V1，K2，V2，K3，V3都是类型
- 通常map函数的输入类型（K1，V1）和map的输出类型（K2，V2）不一样
- 但是reduce的输入类型必须和map的输出类型一样
- reduce的输入类型（K2，V2）与输出类型（K3，V3）可以不一样

MapReduce的类型

- JAVA API可以看出以上特点

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
   public class Context extends MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
      // ...
   }
   protected void map(KEYIN key, VALUEIN value,
            Context context) throws IOException, InterruptedException {
      // ...
   }
}
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
   public class Context extends ReducerContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
      // ...
   }
   protected void reduce(KEYIN key, Iterable<VALUEIN> values,
            Context context) throws IOException,InterruptedException {
      // ...
   }
}
```

用户定义的Mapper子类需要继承这个类

Mapper的Context类作为Mapper的内部类实现，map函数使用的Context对象是MapContext类型

用户定义的Mapper子类需要重新实现这个方法

用户定义的Reducer子类需要继承这个类

Reducer的Context类作为Reducer的内部类实现，reduce函数使用的Context对象是ReducerContext类型

用户定义的Reducer子类需要重新实现这个方法

MapReduce的类型

- 二种Context对象的作用是输出key-value对，因此它们被类型参数化，Context的write方法原型是

```
public void write(KEYOUT key, VALUEOUT value)
                  throws IOException, InterruptedException
```

- 既然Mapper和Reducer是不同的类，所以它们的类型参数作用域不同，例如Mapper的类型参数KEYIN的实际类型可以和Reducer的类型参数KEYIN的实际类型不同
- 同样地，虽然map的输出类型必须和reduce的输入类型一样，但Java编译器是无法保证的，即这个约束条件只能靠程序员自己保证

combine方法

- map方法的输出会通过网络传送到Reducer上作为reduce方法的输入
- 为了减少网络I/O的消耗，在某些应用场景下在Mapper端本地进行数据合并，将合并后的数据通过网络传送给Reducer
- 例如需要统计从1950年到2014年每年的最高气温，假设每天会产生大量的监测数据（例如温度传感器每秒采集一次温度），每条记录的格式为（时间，温度值），因此MapRduce的输入为如下格式的记录集合

```
1950-01-01-00-00-01    20
1950-01-01-00-00-02    20
1950-01-01-00-00-03    21
1950-01-01-00-00-04    21
1950-01-01-00-00-05    21
1950-01-01-00-00-06    21
1950-01-01-00-00-07    21
1950-01-01-00-00-08    21
1950-01-01-00-00-09    21
1950-01-01-00-00-10    21
1950-01-01-00-00-11    19
…
```

这些记录以文本形式保存在很多文本文件中，一行一条记录

combine方法

- 这些数据文件被分割成Splits，每个Mapper处理一个Split，map方法输出的key-value对的形式为(年份,温度)
- 假设第一个Mapper的输出为

```
(1950, 20)
(1950, 20)
(1950, 10)
```

第二个Mapper的输出为

```
(1950, 25)
(1950, 15)
```

- reduce方法的输入则为

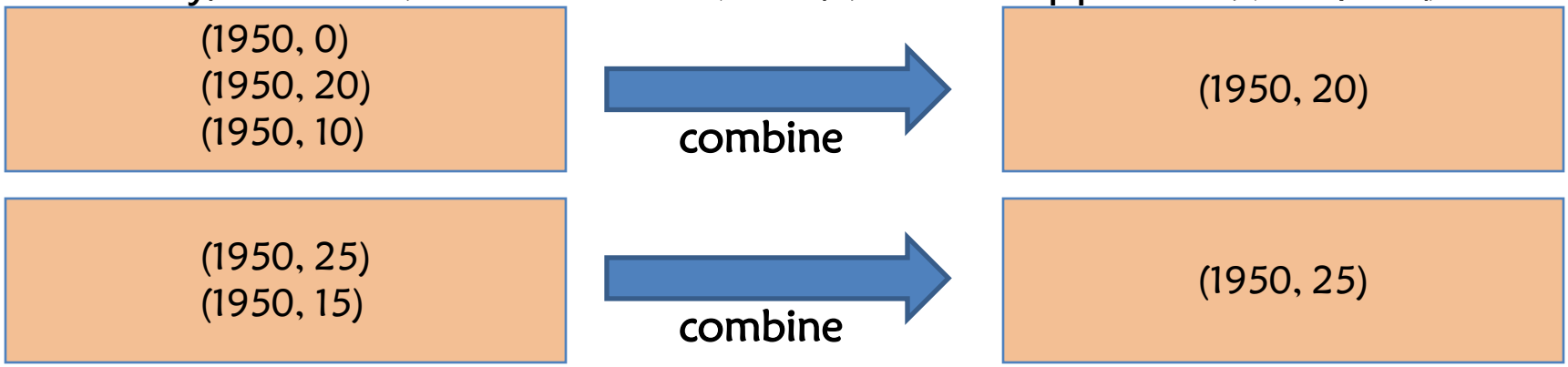
```
(1950, [20, 20, 10, 25, 15])
```

- 最后reduce的输出为

```
(1950,  25)
```

combine方法

- 由于这个应用场景是要求list<value>中的最大值，因此我们可以在map输出的list<key,value>上先运行combine方法，先求出Mapper本地数据中的最大值



- 再将combine的输出作为Reducer的输入，reduce方法的输入则为

(1950, [20，25])

- 最后reduce的输出还是为

(1950，25)

- 但通过网络发送给Reducer的数据量大大减少了

combine方法

- 由于combine方法与reduce方法一样，因此这个应用场景的计算过程可以表达为如下形式

$$max(0, 20, 10, 25, 15) = max(max(0, 20, 10), max(25, 15)) = max(20, 25) = 25$$

- 但不是所有计算过程都满足这个性质，例如我们要计算温度的均值

$$mean(0, 20, 10, 25, 15) = 14$$

但是

$$mean(mean(0, 20, 10), mean(25, 15)) = mean(10, 20) = 15$$

指定combine方法

- 在程序中可以利用Job类的setCombinerClass来指定实现combine方法的类型

```
public class MaxTemperatureWithCombiner {
   public static void main(String[] args) throws Exception {
      if (args.length != 2) {
         System.err.println("Usage: MaxTemperatureWithCombiner <input path> " +
                            "<output path>");
         System.exit(-1);
      }
      Job job = new Job();
      job.setJarByClass(MaxTemperatureWithCombiner.class);
      job.setJobName("Max temperature");
      FileInputFormat.addInputPath(job, new Path(args[0]));
      FileOutputFormat.setOutputPath(job, new Path(args[1]));
      job.setMapperClass(MaxTemperatureMapper.class);
      job.setCombinerClass(MaxTemperatureReducer.class);
      job.setReducerClass(MaxTemperatureReducer.class);
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(IntWritable.class);
      System.exit(job.waitForCompletion(true) ? 0 : 1);
   }
}
```

Combiner的实现类和Reducer一样

MapReduce的类型

- 如果使用combine方法，则它和reduce方法的形式一样（是Reducer的实现），除了它的输出不同：

map: (K1,V1)->list<K2,V2>

combine: (K2, list(V2)) → list(K2, V2)

reduce: (K2,list(V2))->list<K3,V3>

partition方法

- partition方法的作用是根据map方法输出的中间结果（K2，V2）产生一个partition索引（别忘了不同的partition由不同的Reducer处理）

```
partition: (K2, V2) → integer
```

实际上partition索引的计算只依赖于key，value被忽略

- 在JAVA API中，partition方法是由抽象类Partitioner定义

```
public abstract class Partitioner<KEY, VALUE> {
        public abstract int getPartition(KEY key, VALUE value, int numPartitions);
}
```

- 最后全部汇总在一起

```
map: (K1,V1)->list<K2,V2>
combine: (K2, list(V2)) → list(K2, V2)
reduce: (K2,list(V2))->list<K3,V3>
partition: (K2, V2) → integer
```

在Job中配置MapReduce的类型

- 下表给出了配置MapReduce类型的属性与API函数

Table 7-1. Configuration of MapReduce types in the new API

Property	Job setter method	Input types		Intermediate types		Output types	
		K1	V1	K2	V2	K3	V3
Properties for configuring types:							
mapreduce.job.inputformat.class	setInputFormatClass()	•	•				
mapreduce.map.output.key.class	setMapOutputKeyClass()			•			
mapreduce.map.output.value.class	setMapOutputValueClass()				•		
mapreduce.job.output.key.class	setOutputKeyClass()					•	
mapreduce.job.output.value.class	setOutputValueClass()						•
Properties that must be consistent with the types:							
mapreduce.job.map.class	setMapperClass()	•	•	•	•		
mapreduce.job.combine.class	setCombinerClass()			•	•		
mapreduce.job.partitioner.class	setPartitionerClass()			•	•		
mapreduce.job.output.key.comparator.class	setSortComparatorClass()			•			
mapreduce.job.output.group.comparator.class	setGroupingComparatorClass()			•			
mapreduce.job.reduce.class	setReducerClass()			•	•	•	•
mapreduce.job.outputformat.class	setOutputFormatClass()					•	•

- 这些配置属性分成二类
  - 对类型进行配置的属性
  - 必须和配置的类型兼容的属性

在Job中配置MapReduce的类型

- 下表给出了配置MapReduce类型的属性与API函数

Table 7-1. Configuration of MapReduce types in the new API

Property	Job setter method	Input types		Intermediate types		Output types	
		K1	V1	K2	V2	K3	V3
Properties for configuring types:							
mapreduce.job.inputformat.class	setInputFormatClass()	•	•				
mapreduce.map.output.key.class	setMapOutputKeyClass()			•			
mapreduce.map.output.value.class	setMapOutputValueClass()				•		
mapreduce.job.output.key.class	setOutputKeyClass()					•	
mapreduce.job.output.value.class	setOutputValueClass()						•
Properties that must be consistent with the types:							
mapreduce.job.map.class	setMapperClass()	•	•	•	•		
mapreduce.job.combine.class	setCombinerClass()			•	•		
mapreduce.job.partitioner.class	setPartitionerClass()			•	•		
mapreduce.job.output.key.comparator.class	setSortComparatorClass()			•			
mapreduce.job.output.group.comparator.class	setGroupingComparatorClass()			•			
mapreduce.job.reduce.class	setReducerClass()			•	•	•	•
mapreduce.job.outputformat.class	setOutputFormatClass()					•	•

- map的输入类型由input format决定
  - 例如，当mapreduce.job.inputformat.class属性设置为TextInputFormat时，map的输入类型K1为LongWritable，V1为Text.
- 其它类型通过调用Job类的setter方法决定

在Job中配置MapReduce的类型

- 下表给出了配置MapReduce类型的属性与API函数

Table 7-1. Configuration of MapReduce types in the new API

Property	Job setter method	Input types		Intermediate types		Output types	
		K1	V1	K2	V2	K3	V3
Properties for configuring types:							
mapreduce.job.inputformat.class	setInputFormatClass()	•	•				
mapreduce.map.output.key.class	setMapOutputKeyClass()			•			
mapreduce.map.output.value.class	setMapOutputValueClass()				•		
mapreduce.job.output.key.class	setOutputKeyClass()					•	
mapreduce.job.output.value.class	setOutputValueClass()						•
Properties that must be consistent with the types:							
mapreduce.job.map.class	setMapperClass()	•	•	•	•		
mapreduce.job.combine.class	setCombinerClass()			•	•		
mapreduce.job.partitioner.class	setPartitionerClass()			•	•		
mapreduce.job.output.key.comparator.class	setSortComparatorClass()			•			
mapreduce.job.output.group.comparator.class	setGroupingComparatorClass()			•			
mapreduce.job.reduce.class	setReducerClass()			•	•	•	•
mapreduce.job.outputformat.class	setOutputFormatClass()					•	•

- 如果没有显式地设置中间类型，中间类型缺省地和最终的输出类型一样，即K2=K3，V2=V3
  - K3的缺省类型是LongWritable，V3的缺省类型是Text
  - 如果中间类型和最终输出类型一样，没有必要调用setMapOutputKeyClass()和setMapOutputValueClass()，只需要调用setOutputKeyClass()和setOutputValueClass()

MapReduce的输入格式

- MapReduce能处理很多不同类型的数据格式，从flat text文件到数据库
- Mapreduce的输入首先被分为很多InputSplit，每个InputSplit是输入的其中一块数据（chunck of input），每个Mapper处理一个InputSplit
- 每个InputSplit被分成多条记录（record），每一个record就会产生一个key-value对
  - InputSplit和record都是逻辑上的概念，不一定必须基于文件，也可以基于数据库
  - 如果输入数据来自数据库，InputSplit就对应数据库表中若干行，而record则对应其中一条数据库记录，事实上DBInputFormat就是这么处理数据库数据的

MapReduce的输入API

- MapReduce输入输出API位于包org.apache.hadoop.mapreduce中
- InputSplit由类InputSplit表示，这是一个抽象类

```
public abstract class InputSplit {
    public abstract long getLength() throws IOException, InterruptedException;
    public abstract String[] getLocations() throws IOException,InterruptedException;
}
```

  - getLength方法返回InputSplit的大小（以字节为单位）
  - getLocations返回存储这个InputSplit的位置列表
  - 注意InputSplit不包含数据，只是数据的引用（即数据的位置）
  - MapReduce框架会利用数据的位置信息来就近调度Mapper来处理这个InputSplit

MapReduce的输入API

- MapReduce应用开发者不需要直接处理InputSplit，它是由InputFormat产生
- InputFormat负责产生input split，并将input split分成一条条的record

```
public abstract class InputFormat<K, V> {

   public abstract List<InputSplit> getSplits(JobContext context)
                  throws IOException, InterruptedException;

   public abstract RecordReader<K, V> createRecordReader(InputSplit split,
                  TaskAttemptContext context) throws IOException, InterruptedException;
}
```

这是抽象类，指定了类型参数K，V，类型参数指定了map的输入类型

根据Job上下文将输入分成InputSplit

给定一个InputSplit，创建RecordReader对象读取InputSplit中的记录

MapReduce的输入API

- RecordReader的定义为

```
public abstract class RecordReader<KEYIN,VALUEIN> extends Object implements Closeable{
  //初始化split为要处理的分片，context为任务上下文
  public abstract void initialize(InputSplit  split,   TaskAttemptContext  context )
                                        throws IOException, InterruptedException;
  //读取下一个key-value对，如果返回false表示所有记录读取完毕
  //读取的key和value保存在具体实现类的实例变量里
  public abstract   boolean nextKeyValue() throws IOException, InterruptedException;
  //获取当前已读取记录的key
  public abstract  KEYIN getCurrentKey() throws IOException, InterruptedException;
  //获取当前已读取记录的value
  public abstract   VALUEIN getCurrentValue() throws IOException, InterruptedException;
  //获取当前处理的进度，返回百分比
  public abstract float getProgress() throws IOException, InterruptedException;
  //关闭RecordReader
  public abstract void close() throws IOException;
}
```

MapReduce的输入API

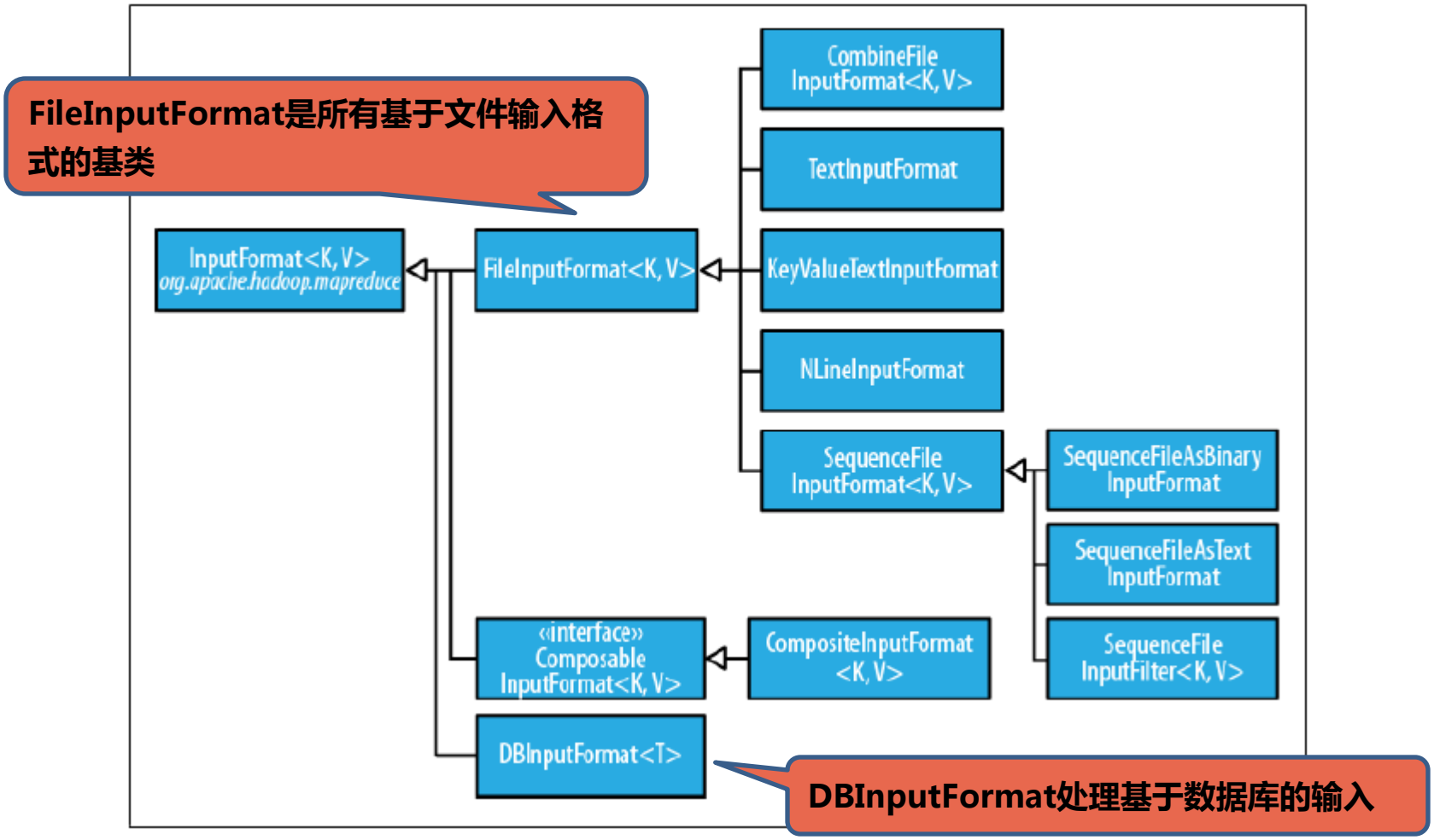
- 当客户端运行一个Job时，客户端会自动调用getSplits方法，然后将得到的InputSplit列表发送给jobtracker
  - jobtracker利用InputSplit中的位置信息来调度map tasks， map task运行在tasktracker上
  - map task会将split传递给createRecordReader()得到这个split的RecordReader
  - RecordReader会一次读取一条记录，并由map task产生一个key-value对传递给map方法
  - map task是Mapper对象，从它的 run方法就可以看出这个过程

```
//Mapper类的run方法
public void run(Context context) throws IOException, InterruptedException {
   setup(context); //在任务开始运行调用一次setup完成任务的设置
   while (context.nextKeyValue()) {
      map(context.getCurrentKey(), context.getCurrentValue(), context);
   }
   cleanup(context); //在任务结束运行调用一次cleanup完成任务的清理
}
```

  - 通过上下文对象contex反复调用nextKeyValue方法时，调用请求会委托给RecordReader的同名方法
  - RecordReader会根据record产生key-value对
  - 当RecordReader读完所有记录， nextKeyValue方法返回false
  - map task清理任务
  - context.getCurrentKey()和context.getCurrentValue()调用都会委托委托给RecordReader的同名方法

MapReduce的输入API

- InputFormat类层次结构



FileInputFormat类

- FileInputFormat提供了二个功能
  - 定义job的输入包含哪些文件（通过定义输入文件路径）
  - 实现了将输入文件分成Splits的功能
  - 进一步将Splits分成record的功能由其子类实现
- FileInputFormat 输入路径
  - 一个job的输入可以包括多个输入路径， FileInputFormat 提供了四个静态方法设置输入路径

```
public static void addInputPath(Job job, Path path)

public static void addInputPaths(Job job, String commaSeparatedPaths)

public static void setInputPaths(Job job, Path... inputPaths)

public static void setInputPaths(Job job, String commaSeparatedPaths)
```

  - addInputPaths是加入新的路径
  - setInputPaths一次设置完整的路径列表，会替换掉前面调用所设置的所有路径
  - 路径可以是文件，可以是目录
  - 目录不会被递归处理

FileInputFormat类的输入分片

- FileInputFormat只分割大文件，这里的“大”指大于HDFS的块大小
- Split的大小通常等于HDFS块的大小，对大多数应用来说是合理的。但是也可以通过设置不同的Hadoop属性来改变

属性名称	类型	默认值	描述
mapred.min.split.size	int	1	文件分片最小的有效字节数
mapred.max.split.size	long	Long.MAX_VALUE,即,9223372036854775807	文件分片最大的有效字节数
dfs.block.size	long	64MB，即67108864	HDFS块的大小

- 分片的大小由以下公式计算（参见FileInputFormat的computeSplitSize方法）

```
max(minimumSize, min(maximumSize, blockSize))
```

- 默认情况下， minimumSize < blockSize < maximumSize，因此分片大小就是blockSize

FileInputFormat类的输入分片设置举例

最小分片大小	最大分片大小	块的大小	分片大小	说明
1（缺省）	Long.MAX_VALUE（缺省）	64MB（缺省）	64MB	默认情况下，分片大小与块大小相同
1(缺省）	Long.MAX_VALUE（缺省）	128M	128M	增加分片大小最自然的办法是提供更大的HDFS块大小
128MB	Long.MAX_VALUE（缺省）	64MB（缺省）	128MB	通过使最小分片值大于块大小的方法来增大分片大小，但会增加对map任务来说不是本地的块数
1（缺省）	32MB	64MB（缺省）	32M	通过使最大分片大小小于块大小的方法来减少分片大小

小文件与CombineFileInputFormat

- Hadoop更适合处理少量的大文件而不是大量的小文件
  - 一个原因是FileInputFormat生成的InputSplit是一个文件或该文件的一部分。如果文件很小（比HDFS block小很多），并且这种文件数量很多，那么每个map任务只能处理很小的输入数据（一个小文件），而且这样的map任务会很多
  - 比较分割成16个64MB块的1GB的文件和100KB大小的10000个文件：10000个文件每个都需要一个map操作，作业时间比前一个文件上的16个map操作慢几十倍甚至几百倍
- CombineFileInputFormat 可以缓解这个问题：FileInputFormat为每个小文件产生一个Split，而CombineFileInputFormat 把多个文件打包到一个Split中
  - 决定哪些文件放入同一个Split时， CombineFileInputFormat 会考虑节点和机架的因素
- 当然，如果可能，，应该尽量避免许多小文件的情况，因为会浪费namenode的内存
  - 一个减少大量小文件的方法是使用SequenceFile将这些小文件合并成一个或多个大文件
  - 可以用文件名作为key，文件的内容作为value放入SequenceFile
- CombineFileInputFormat只是一个抽象类，因此使用时需要自己实现具体子类

避免切分

- 有些应用程序可能不希望文件被切分，而是用一个mapper完整地处理每一个输入文件
  - 例如，检查一个文件中的记录是否有序，一个简单的方法是顺序扫描每一条记录并比较后一个记录是否比前一个记录小（或大）。如果要将它实现为一个map任务，那么只有当map操作扫描整个文件该算法才可行
- 有二种方法可以保证输入文件不被切分
  - 第一种是增加最小分片大小，将它设置成要处理的最大文件 的大小或设置成LONG.MAX_VALUE
  - 第二种方法就是使用FileInputFormat的具体子类（如TextInputFormat），并且重新实现isSplitable（）方法把返回值设为false

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.JobContext;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

public class NonSplittableTextInputFormat extends TextInputFormat {
    @Override
    protected boolean isSplitable(JobContext context, Path file) {
        return false;
    }
}
```

Mapper中的文件信息

- 处理文件输入分片的Mapper可以从作业配置对象的某些特定属性中读取输入分片的有关信息
  - 通过调用Mapper的Context对象的getInputSplit()返回InputSplit对象
  - 如果输入格式对象是FileInputFormat的子类（如TextInputFormat）时，将返回的InputSplit对象强制转换成FileSplit对象（InputSplit的子类）
  - 接着可以利用获取下表所示的信息

FileSplit method	Property name	Type	Description
getPath()	map.input.file	Path/String	The path of the input file being processed
getStart()	map.input.start	long	The byte offset of the start of the split from the beginning of the file
getLength()	map.input.length	long	The length of the split in bytes

把整个文件作为一个记录处理

- 有时即使不分割文件，仍然需要一个RecordReader来读取文件文件内容作为一个record的值

```
public class WholeFileInputFormat extends FileInputFormat<NullWritable, BytesWritable> {

@Override
   protected boolean isSplitable(JobContext context, Path file) {
      return false;
   }

   @Override
   public RecordReader<NullWritable, BytesWritable> createRecordReader(InputSplit split,
                     TaskAttemptContext context) throws IOException,InterruptedException
   {
      WholeFileRecordReader reader = new WholeFileRecordReader();
      reader.initialize(split, context);
      return reader;
   }
}
```

继承FileInputFormat，key类型为NullWritable，value类型为BytesWritable
键值为空，文件内容表示成BytesWritable

重新实现isSplitable方法，返回false。不对输入文件分片

重新实现createRecordReader方法

实例化定制的RecordReader并返回

把整个文件作为一个记录处理

- 定制的RecordReader实现

```
class WholeFileRecordReader extends RecordReader<NullWritable, BytesWritable> {
    private FileSplit fileSplit;        //保存输入的分片，它将被转换成一条（key，value）记录
    private Configuration conf;         //配置对象
    private BytesWritable value = new BytesWritable(); //value对象，内容为空
    private boolean processed = false;  //布尔变量记录记录是否被处理过

    @Override
    public void initialize(InputSplit split, TaskAttemptContext context)
                                                throws IOException, InterruptedException {
        this.fileSplit = (FileSplit) split;             //将输入分片强制转换成FileSplit
        this.conf = context.getConfiguration(); //从context获取配置信息
    }
    @Override
    public NullWritable getCurrentKey() throws IOException, InterruptedException {
        return NullWritable.get();
    }
    @Override
    public BytesWritable getCurrentValue() throws IOException, InterruptedException {
        return value;
    }
}
```

重新实现RecordReader的初始化方法。split为输入分片，context为任务的上下文

重新实现getCurrentKey方法

重新实现getCurrentValue方法

把整个文件作为一个记录处理

- 定制的RecordReader实现

```
class WholeFileRecordReader extends RecordReader<NullWritable, BytesWritable> {
    @Override
    public boolean nextKeyValue() throws IOException, InterruptedException {
        if (!processed) { //如果记录没有被处理过
            //从fileSplit对象获取split的字节数，创建byte数组contents
            byte[] contents = new byte[(int) fileSplit.getLength()];
            Path file = fileSplit.getPath(); //从fileSplit对象获取输入文件路径
            FileSystem fs = file.getFileSystem(conf); //获取文件系统对象
            FSDataInputStream in = null; //定义文件输入流对象
            try {
                in = fs.open(file); //打开文件，返回文件输入流对象
                IOUtils.readFully(in, contents, 0, contents.length); //从输入流读取所有字节到contents
                value.set(contents, 0, contents.length); //将contens内容设置到value对象中
            } finally {
                IOUtils.closeStream(in); //关闭输入流
            }
            processed = true; //将是否处理标志设为true，下次调用该方法会返回false
            return true;
        }
        return false; //如果记录处理过，返回false，表示split处理完毕
    }
}
```

重新实现nextKeyValue方法

把整个文件作为一个记录处理

- 定制的RecordReader实现

```
class WholeFileRecordReader extends RecordReader<NullWritable, BytesWritable> {


    @Override
    public float getProgress() throws IOException {
        return processed ? 1.0f : 0.0f;
    }

    @Override
    public void close() throws IOException {
        // do nothing
    }
}
```

重新实现getProgress方法
该方法返回一个浮点数，表示已经处理的数据占所有要处理数据的百分比
由于是将整个文件作为一个记录读取，因此返回要么是0.0，要么是1.0

重新实现close方法
该方法关闭RecordReader
由于前面已经关闭文件输入流，所以这里不要做任何事

将小文件打包成SequenceFile的MapReduce程序

- 利用WholeFileInputFormat将多个小文件打包成SequenceFile的MapReduce实现

```
public class SmallFilesToSequenceFileConverter extends Configured implements Tool {
```

实现Tool接口，利用ToolRunner来运行这个MapReduce程序

利用嵌套类实现SequenceFileMapper，继承Mapper类

```
  static class SequenceFileMapper
          extends Mapper<NullWritable, BytesWritable, Text, BytesWritable> {
    private Text filenameKey;  //被打包的小文件名作为key，表示为Text对象
    @Override //重新实现setup方法，进行map任务的初始化设置
    protected void setup(Context context) throws IOException, InterruptedException {
      InputSplit split = context.getInputSplit(); //从context获取split
      Path path = ((FileSplit) split).getPath();  //从split获取文件路径
      filenameKey = new Text(path.toString()); //将文件路径实例化为key对象
    }
    @Override //实现map方法
    protected void map(NullWritable key, BytesWritable value, Context context)
                    throws IOException, InterruptedException {
      context.write(filenameKey, value);
    }
  }
}
```

map的输入是由WholeFileInputFormat将一个小文件作为一个记录处理生成的key-value对，其中key为NullWritable对象，value类型是BytesWritable，值为小文件内容。Map只是简单地将key-value对写到context中，由于没有reduce，所以context内容被直接写入SequenceFile

将小文件打包成SequenceFile的MapReduce程序

- 利用WholeFileInputFormat将多个小文件打包成SequenceFile的MapReduce实现

```
public class SmallFilesToSequenceFileConverter extends Configured implements Tool {

    @Override //实现Tool接口的run方法
    public int run(String[] args) throws Exception {
        Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
        if (job == null) {
            return -1;
        }
        job.setInputFormatClass(WholeFileInputFormat.class);
        job.setOutputFormatClass(SequenceFileOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(BytesWritable.class);
        job.setMapperClass(SequenceFileMapper.class);
        return job.waitForCompletion(true) ? 0 : 1;
    }
    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new SmallFilesToSequenceFileConverter(), args);
        System.exit(exitCode);
    }
}
```

构造Job对象，JobBuilder会帮助解析命令行参数。getConf方法获取配置信息，是调用父类实现

利用ToolRunner启动程序

将小文件打包成SequenceFile的MapReduce程序

- 注意这里没有设置Reducer class，等价于以下代码

```
public class SmallFilesToSequenceFileConverter extends Configured implements Tool {
   @Override //实现Tool接口的run方法
   public int run(String[] args) throws Exception {
      Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
      if (job == null) {
         return -1;
      }
      job.setInputFormatClass(WholeFileInputFormat.class);
      job.setOutputFormatClass(SequenceFileOutputFormat.class);
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(BytesWritable.class);
      job.setMapperClass(SequenceFileMapper.class);
      job.setReducerClasss(IdentifyReducer.class);
      return job.waitForCompletion(true) ? 0 : 1;
   }
   public static void main(String[] args) throws Exception {
      int exitCode = ToolRunner.run(new SmallFilesToSequenceFileConverter(), args);
      System.exit(exitCode);
   }
}
```

IdentifyMapper和IdentifyReducer将输入的key，value原封不动地写到输出，输入的key-value类型与输出的key-value类型一致

将小文件打包成SequenceFile的MapReduce程序

- 运行程序

```
% hadoop jar hadoop-examples.jar SmallFilesToSequenceFileConverter \
-conf  conf/hadoop-localhost.xml  -D  mapred.reduce.tasks=2  input/smallfiles  output
```

  - 由于指定了2个reduce任务，因此会产生二个SequenceFile
  - 可以用hadoop fs –text命令查看这二个文件内容

```
% hadoop fs -conf conf/hadoop-localhost.xml -text output/part-r-00000
hdfs://localhost/user/tom/input/smallfiles/a        61 61 61 61 61 61 61 61 61 61
hdfs://localhost/user/tom/input/smallfiles/c        63 63 63 63 63 63 63 63 63 63
hdfs://localhost/user/tom/input/smallfiles/e        65 65 65 65 65 65 65 65 65 65
% hadoop fs -conf conf/hadoop-localhost.xml -text output/part-r-00001
hdfs://localhost/user/tom/input/smallfiles/b        62 62 62 62 62 62 62 62 62 62
hdfs://localhost/user/tom/input/smallfiles/d        64 64 64 64 64 64 64 64 64 64
hdfs://localhost/user/tom/input/smallfiles/f        66 66 66 66 66 66 66 66 66 66
```

注意文件内容是二进制（这里是字符的ASCII码）
例如文件a的内容是10个字符’a’

TextInput-TextInputFormat

- TextInputFormat是默认的InputFormat
- 文本文件的每一行作为一个record读入
  - key是LongWritable类型，其值为该行的起始字符在整个文件中的字节偏移量
  - value是Text类型，其值为该行内容（不包括任何行终止符，如回车换行符）
  - 因此如下的文件被切分成包含4条记录的split

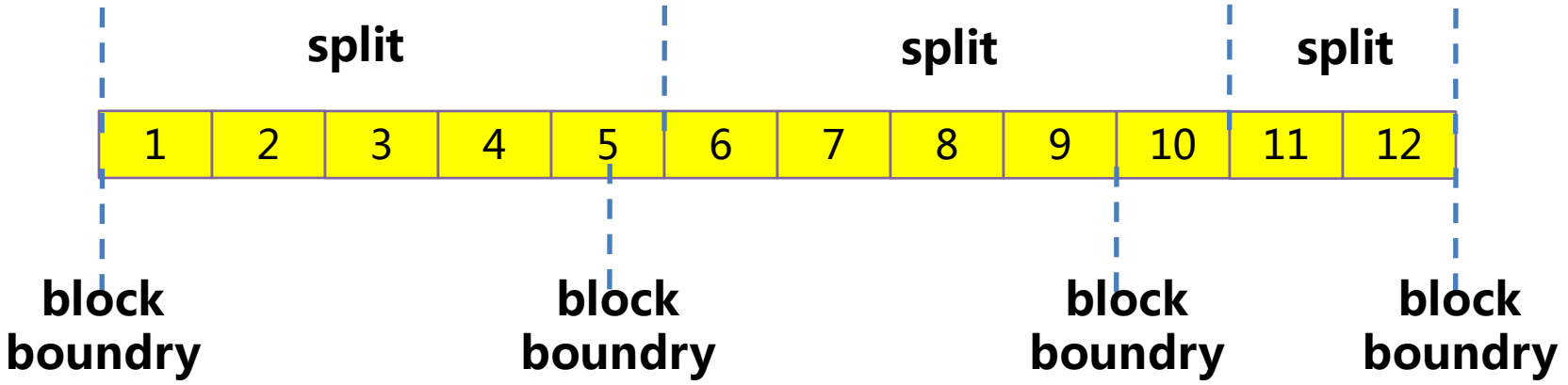
```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

```
(0,     On the top of the Crumpetty Tree)
(33,    The Quangle Wangle sat,)
(57,    But his face you could not see,)
(89,    On account of his Beaver Hat.)
```

  - 为什么key不使用行号？因为文件是按字节而不是行号来分片，用行号不方便计算分片的大小
  - 但TextInputFormat不会把一行分到不同分片里

输入分片与HDFS块的关系

- FileInputFormat定义的逻辑记录并不能很好地匹配HDFS的文件块
- 例如，TextInputFormat的逻辑记录是以行为单位，但很有可能一行是跨HDFS文件块存放，这意味着map会执行一些远程读操作，对性能可能会有一些影响



TextInput-KeyValueTextInputFormat

- TextInputFormat的键是每一行在文件中的偏移量，并不十分有用
- 通常情况下，文件中的每一行是key-value对，使用某个分界符进行分隔
- 这类文件用KeyValueTextInputFormat比较合适
- 可以通过mapreduce.input.keyvaluelinerecordreader.key.value.separator属性设置分隔符，它默认是制表符('\t' ,下例中用→表示)

line1→On the top of the Crumpetty Tree
line2→The Quangle Wangle sat,
line3→But his face you could not see,
line4→On account of his Beaver Hat.

(line1, On the top of the Crumpetty Tree)
(line2, The Quangle Wangle sat,)
(line3, But his face you could not see,)
(line4, On account of his Beaver Hat.)

TextInput- NLineInputFormat

- 利用TextInputFormat和KeyValueTextInputFormat作为输入格式的mapper任务，处理的行数（record数）是不确定的，因为行数取决于split的字节数和每行的字节数
- 如果希望mapper处理固定数量的输入记录，就可以使用NLineInputFormat
  - 和TextInputFormat一样，key是每行起始位置在文件中的字节偏移量，value是行内容
  - mapreduce.input.lineinputformat.linespermap属性指定每个map处理的行数，在下面例子中， mapreduce.input.lineinputformat.linespermap =2，每个split包括2行，分别由2个mapper处理

On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.

(0, On the top of the Crumpetty Tree)
(33, The Quangle Wangle sat,)

(57, But his face you could not see,)
(89, On account of his Beaver Hat.)

BinaryInput- SequenceFileInputFormat

- Hadoop也支持二进制格式的输入
- Hadoop的SequenceFileInputFormat存储二进制key-value对序列
- SequenceFile是可分割的
  - 内部有同步点，所以RecordReader可以从文件中任意一点（例如分片的起点）找到一条记录的起点
- 若MapReduce中输入是SequenceFile，就必须使SequenceFileInputFormat
  - Mapper的输入key和value的类型由SequenceFile决定
  - 例如如果输入SequenceFile的key类型是IntWritable，value类型是Text，那么mapper的格式应该是Mapper<IntWritable，Text,K,V>

BinaryInput- SequenceFileAsTextInputFormat

- Hadoop也支持二进制格式的输入
- Hadoop的SequenceFileInputFormat存储二进制key-value对序列
- SequenceFile是可分割的
  - 内部有同步点，所以RecordReader可以从文件中任意一点（例如分片的起点）找到一条记录的起点
- 若MapReduce中输入是SequenceFile，就必须使SequenceFileInputFormat
  - Mapper的输入key和value的类型由SequenceFile决定
  - 例如如果输入SequenceFile的key类型是IntWritable，value类型是Text，那么mapper的格式应该是Mapper<IntWritable，Text,K,V>
- SequenceFileAsTextInputFormat是SequenceFileInputFormat的变体
  - 它将key和value转换为Text对象。转换的时候会调用键和值的toString()方法
- SequenceFileAsBinaryInputFormat也是SequenceFileInputFormat的变体
  - 它将SequenceFile的键和值作为二进制对象。它们被封装为BytesWritebale对象，因而应用程序可以任意地将这些字节数组解释为它们想要的类型

MultipleInputs

- 虽然一个MapReduce程序可能有多个输入文件（输入路径），但所有文件都是由同一个InputFormat和同一个Mapper类解释
- 但通常情况下数据格式会随时间演变，必须使用当前的Mapper能够适应和处理遗留的数据格式
- 或者有些数据源会提供相同的数据，但数据的格式不同
  - 例如有些数据可能是使用Tab分隔的文本文件，另外一些可能是二进制顺序文件，这就需要分别进行解析
- 这些问题可以用MultipleInputs来解决，它可以在每一个输入路径上规定InputFormat和Mapper的类型

```
MultipleInputs.addInputPath(job, ncdcInputPath,
        TextInputFormat.class, MaxTemperatureMapper.class);
MultipleInputs.addInputPath(job, metOfficeInputPath,
        TextInputFormat.class, MetOfficeMaxTemperatureMapper.class);
```

  - 这段代码取代了通常的对FileInputFormat.addInputPath()和对job.setMapperClass()的调用
  - 二个不同路径下的文件都是文本格式，因此都可以用TextInputFormat。但这二个文件中每一行的表示不同，所以需要二个不一样的Mapper
  - 重要的一点是两个Mapper的输出类型要一致，这样可以使用同一个Reducer来操作

DBInputFormat

- DBInputFomat是一种使用JDBC并且从关系数据库中读取数据的一种输入格式
- 使用它的时候必须非常小心，太多的Mapper可能会使数据库无法承受。由于这个原因，DBInputFormat最好在加载小量的数据集时使用
- 与之对应的输出格式是DBOutputFormat，在将结果写入数据库时非常有用
- HBase中的TableInputFormat可以让MapReduce程序访问HBase表里的数据
- 同样，TableOutputFormat可以将MapReduce的输出写入HBase

MapReduce的输出API

- 对前面介绍的每一种输入格式类，MapReduce都有对应的输出格式类
- 这些类都从抽象类OutputFormat派生，其职责有
  - 验证输出是否合规范，例如输出目录是否已经存在（若存在则不符合规范）
  - 返回RecordWriter，用于将输出写入文件系统中

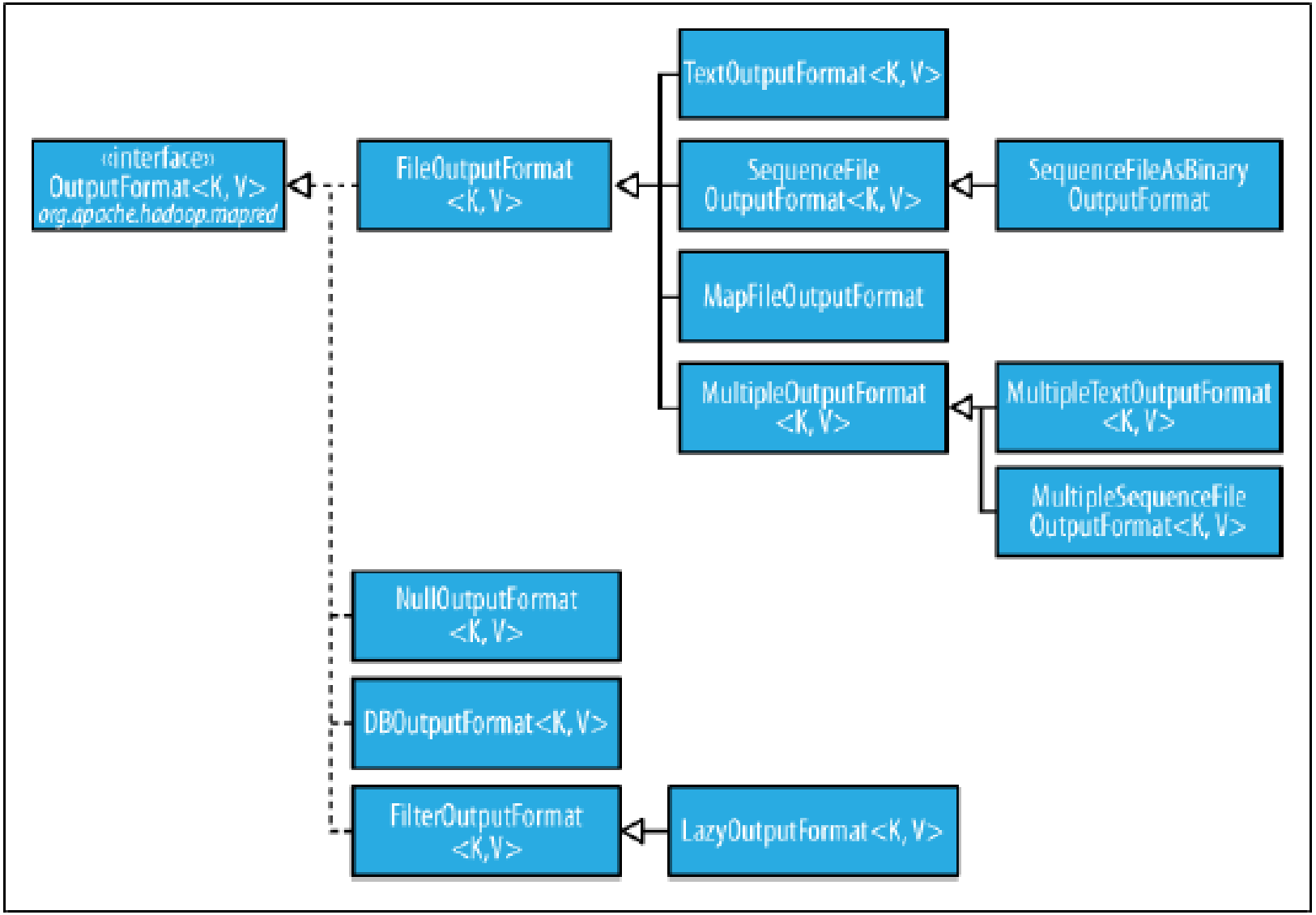
```
public abstract class OutputFormat<K,V>extends Object{
   //返回RecordWriter，由RecordWriter负责将记录写入文件系统
   public abstract RecordWriter<K,V> getRecordWriter(TaskAttemptContext context)
                        throws IOException, InterruptedException);
   //检查输出是否合规范
   public abstract void checkOutputSpecs(JobContext context)
                        throws IOException, InterruptedException
   //返回OutputCommitter，它保证写写操作被正确提交
   public abstract OutputCommitter getOutputCommitter(TaskAttemptContext context)
                        throws IOException, InterruptedException
}
```

- RecordWriter

```
public abstract class RecordWriter<K,V>extends Object{
   public abstract void write(K key, V value) throws IOException, InterruptedException;
   public abstract void close(TaskAttemptContext context)
                        throws IOException, InterruptedException
}
```

MapReduce的输出API

- OutputFormat类层次结构



TextOutputFormat

- MapReduce默认的输出格式是TextOutputFormat
- 将每条记录以一行的形式存入文本文件
- 它的键和值可以是任意类型，因为通过toString()方法将它们转换成字符串再输出。
  - 每个key-value对都默认用Tab符分隔
- 可以在mapreduce.output.textoutputformat.separator属性中设置分隔符，这时对应的输入格式是KeyValueTextInputFormat。因为它使用指定的分隔符将key和value分开
- 如果不想输出key或value，将其数据类型设置成NullWritable。这时也不会输出分隔符，使得输出更适合用TextInputFormat进行下一步处理
- 如果key和value的类型都是NullWritable，则什么都不会输出，这时等价于输出格式NullOutputFormat

Binary Output

- SequenceFileOutputFormat
  - SequenceFileOutputFormat将输出写入一个顺序文件
  - 如果输出需要作为后续MapReduce任务的输入，那么这是一个比较好的选择，因为它的格式非常紧凑，而且还可以被压缩
  - 对它的压缩可以由SequenceFileOutputFormat的静态方法setOutputCompressionType实现
- SequenceFileAsBinaryOutputFormat
  - SequenceFileAsBinaryOutputFormat与SequenceFileAsBinaryInputFormat相对应，它将key-value对当做raw二进制数据（字节数组）写入一个顺序文件
- MapFileOutputFormat
  - MapFileOutputFormat将结果写入一个MapFile中。MapFile中的键必须是排序的，所以在reducer中必须保证输出的的键有序

Multiple Outputs

- FileOutputFormat及其子类会在某个目录下生成若干文件
  - 每个reducer会输出一个文件并且文件名与分区相对应：part-00000，part-00001等
- 有时需要对文件名进行控制，或者让每个Reducer输出多个文件，这时需要用到MultipleOutputs类
- 实例分析：数据分区
- 假设需要分析来自不同气象站的数据，如果想将所有的气象数据按照气象站分开，就需要写一个程序，它以每个气象站输出一个文件，并且此文件包含了所有该气象站的数据
- 一种方法是让每个reducer处理一个气象站的数据。这样必须要做二件事：
  - 必须实现一个partioner将同一个气象站的数据分配到同一个分区
  - 我们需要将reducer的个数设置成气象站的总数

MultipleOutputs

- partioner的实现如下

```
public class StationPartitioner extends Partitioner<LongWritable, Text> {

    private NcdcRecordParser parser = new NcdcRecordParser(); //value的解析器

    @Override
    public int getPartition(LongWritable key, Text value, int numPartitions) {
        parser.parse(value); //解析每条记录的value
        return getPartition(parser.getStationId()); //以value中解析出来的气象站id作为partion index
    }
    private int getPartition(String stationId) {
        …
    }
}
```

继承Partioner类，key和value类型分别是LongWritable和Text

实现Partioner类的getPartion方法

根据stationId计算partition

- 这样做的弊端
  - 必须事先知道气象站和分区的数目
  - 如果元数据中有气象站但没有这个气象站的数据就会浪费一个reducer slot
  - 如果元数据中没有这个气象站但是出现了它的数据，那么它将不会被Reducer处理

MultipleOutputs

- 最好能根据集群的容量来决定分区的个数，这就是HashPartioner表现出色的原因：它可以适应任意数量的partion，并能够较好地保证各partion之间是比较均匀的
- 但如果使用HashPartioner，那么每个partion上会包含多个气象站的数据。如果需要实现每一个气象站一个输出文件，就需要让每个Reducer使用MultipleOutputs类
- MultipleOutputs能在将数据写入文件时根据输出的key和value决定文件名。这就使得一个Reducer能将数据写到多个文件中
  - 如果没有Reducer，则每个Mapper直接将输出写到多个文件里
- Mapper直接输出的文件名的格式为name-m-nnnn，Reducer直接输出的文件名格式为name-r-nnnn
  - 其中：name是程序指定的任意名字，nnnn则为partion index

MultipleOutputs

- 下面的代码演示了如何使用MultipleOutputs来按气象站来将数据输出到一个文件里

```
public class PartitionByStationUsingMultipleOutputs extends Configured implements Tool {

    static class StationMapper  extends Mapper<LongWritable, Text, Text, Text> {

        private NcdcRecordParser parser = new NcdcRecordParser(); //value的解析器

        @Override
        protected void map(LongWritable key, Text value, Context context)
                            throws IOException, InterruptedException {
            parser.parse(value); //解析每条记录的value
            context.write(new Text(parser.getStationId()), value);
        }
    }
}
```

实现自定义Mapper类，继承Mapper类

实现map方法

以stationId为key，将key-value写入context

注意同一个stattionId的数据不一定会写入同一个partion

MultipleOutputs

- 下面的代码演示了如何使用MultipleOutputs来按气象站来将数据输出到一个文件里

```
public class PartitionByStationUsingMultipleOutputs extends Configured implements Tool {
  //实现自定义的Reducer，继承Reducer
  static class MultipleOutputsReducer  extends Reducer<Text, Text, NullWritable, Text> {
    private MultipleOutputs<NullWritable, Text> multipleOutputs; //声明MultipleOutputs对象
    @Override  //实现setup方法
    protected void setup(Context context) throws IOException, InterruptedException {
      //实例化MultipleOutputs对象
      multipleOutputs = new MultipleOutputs<NullWritable, Text>(context);
    }
    @Override //实现reduce方法，注意key是stationId
    protected void reduce(Text key, Iterable<Text> values, Context context)
                          throws IOException, InterruptedException {
      for (Text value : values) {
        //将相同key的value写到以key命名的文件中去
        multipleOutputs.write(NullWritable.get(), value, key.toString());
      }
    }
    @Override //实现cleanup方法
    protected void cleanup(Context context) throws IOException, InterruptedException {
      multipleOutputs.close(); //关闭MultipleOutputs对象对象
    }
  }
}
```

MultipleOutputs

- 下面的代码演示了如何使用MultipleOutputs来按气象站来将数据输出到一个文件里

```
public class PartitionByStationUsingMultipleOutputs extends Configured implements Tool {
    @Override //实现run方法
   public int run(String[] args) throws Exception {
      Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
      if (job == null) {
         return -1;
      }
      job.setMapperClass(StationMapper.class); //设置Mapper类
      job.setMapOutputKeyClass(Text.class);     //设置map的输出key类型
      job.setReducerClass(MultipleOutputsReducer.class); //设置Reducer类
      job.setOutputKeyClass(NullWritable.class); //设置最后输出key的类型
      return job.waitForCompletion(true) ? 0 : 1; //启动job
   }
   public static void main(String[] args) throws Exception {
      int exitCode = ToolRunner.run(new PartitionByStationUsingMultipleOutputs(), args);
      System.exit(exitCode);
   }
}
```

MultipleOutputs

- 部分输出结果，文件名格式为station_identifier-r-nnnnn
- 每个文件包含的数据来自同一个stationId，即同一个partion

```
output/010010-99999-r-00027
output/010050-99999-r-00013
output/010100-99999-r-00015
output/010280-99999-r-00014
output/010550-99999-r-00000
output/010980-99999-r-00011
output/011060-99999-r-00025
output/012030-99999-r-00029
output/012350-99999-r-00018
output/012620-99999-r-00004
```

LazyOutputFormat

- FileOutputFormat的子类会产生形如"part-nnnnn"的输出文件，哪怕它们是空的
- 有些应用程序要求不创建空文件，这时可以使用LazyOutputFormat
- 它是一个Wrapper，可以保证在第一条记录输出的时候才真正创建文件
- 要使用它，调用JobConf的setOutputFormatClass()方法