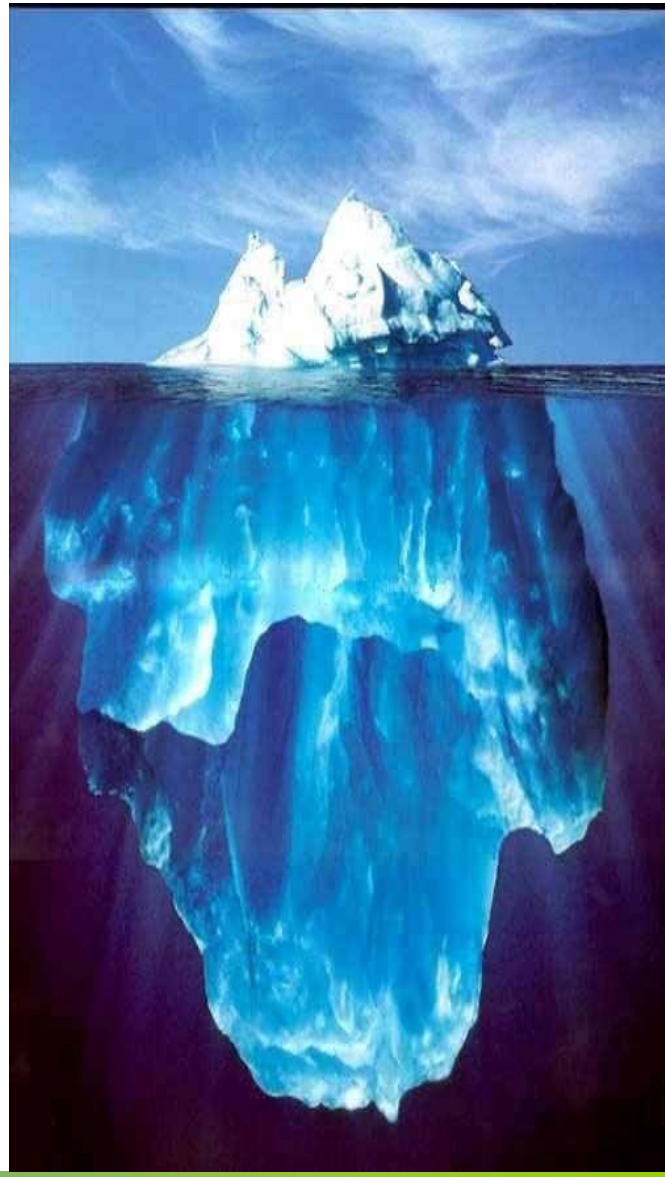
《大数据技术应用》

第七章 MapReduce API

提纲

1 MapReduce API概述
2 MapReduce的数据类型
3 MapReduce的输入
4 MapReduce的输出
5 MapReduce的任务
6 MapReduce应用举例——倒排索引



7.1 MapReduce API概述

1. REST API

MapReduce的REST API包含两部分内容：ApplicationMaster REST API（应用程序管家）和HistoryServer REST API（历史服务器）。

2. Java API

Java API主要位于org.apache.hadoop.mapreduce和org.apache.hadoop.io包中，这些API能够支持的操作包括：启动作业、设置作业信息、设置Map、设置Reduce等。

MapReduce API编程思路

1. 实例化Configuration
2. 实例化Job
3. 设置作业任务
4. 设置输入输出路径
5. 作业的启动与执行
6. 作业的监控与管理

例如wordcont.java代码。

WordCount.java

```
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.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;


public class WordCount {

    public static void main(String[] args) throws Exception {
        // 实例化configuraiton
        Configuration conf = new Configuration();
        String[] ortherArgs = new GenericOptionsParser(conf,args).getRemainingArgs();
        if(ortherArgs.length!=2)
        {
            System.err.println("Usage:wordcount <in> <out>");
            System.exit(2);
        }
        //实例化job
        Job job =Job.getInstance(conf,"Word Count");
        //设置作业任务
        job.setJarByClass(WordCount.class);
        job.setMapperClass(MyMapper.class);
        job.setCombinerClass(MyReducer.class);
        job.setReducerClass(MyReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setOutputFormatClass(CustomOutputFormat.class);
        //设置输入输出路径
        FileInputFormat.addInputPath(job,new Path(ortherArgs[0]));
        FileOutputFormat.setOutputPath(job,new Path(ortherArgs[1]));
        //启动作业执行
        int isSuccess = job.waitForCompletion(true)?0:1;
        if(isSuccess==0)
        System.out.println("鎵ц  鎴愬姛");
        System.exit(isSuccess);

    }

}
```

7.2 MapReduce的数据类型

7.2.1 序列化

- 序列化（**serialization**）是指将结构化对象转化为字节流以便在网络上传输或写到磁盘进行永久存储的过程。反序列化（**deserialization**）是指将字节流转换回结构化对象的逆过程。
- **Hadoop**使用自身的序列化存储格式，实现**Writable**接口。
- **hadoop**系统多个节点间通信通过**RPC**实现。**RPC**将消息序列化成二进制流后发到远程节点，远程节点接受后将二进制流反序列化为原始数据。

7.2.2 Writable接口

1. Writable接口

Writable接口位于org.apahce.hadoop.io包中，定义了两个方法：一个将数据写到DataOutput二进制流，另一个从DataInput二进制流读取数据。

2. WritableComparable接口和comparator

WritableComparator类是实现RawComparator接口的具体子类，它提供两个主要功能：

1) 提供对原始compare()方法的一个默认实现，该方法能够反序列化字节流中要比较的对象，并调用对象的compare()方法。

2) 充当的是RawComparator实例的工厂（已注册Writable的实现），即Writable类中都含有相应的Comparator内部类。

7.2.3 Writable类

1. Java基本类型封装

Writable类的内部都包含一个对应基本类型的成员变量value，用户可以通过get()和set()方法获取或存储value的值。

2. Text类型

Text类使用UTF-8编码存储文本，可以看作Java中String类型的Writable封装。它提供了序列化、反序列化和在字节级别上比较文本的方法。

7.2.3 Writable类

3. BytesWritable

BytesWritable是对二进制数组的封装。

4. NullWritable

NullWritable是一个非常特殊的Writable类型，序列化不包含任何字符，不会从数据流中读数据也不会写入数据。

5. ObjectWritable和GenericWritable

ObjectWritable提供一个封装，适用于字段需要使用多种类型的情况。

6. Writable集合类

在Hadoop包（org.apache.hadoop.io）中一共有6个Writable集合类： aarray,arrayprimitive,twoarray,map,sortedmap,enummap

7.3 MapReduce的输入

7.3.1 输入分片

- 一个输入分片（**InputSplit**）就是一个**Map**任务的输入数据。每个**Map**任务只处理一个输入分片，也就是说有多少个分片就有多少个**Map**任务。**split**可以看作是**MapReduce**的处理单位。
- 输入分片是一个逻辑概念，没有对应到文件，它被表示为**org.apache.hadoop.mapreduce**包中的**InputSplit**抽象类。

7.3.2 文件输入

1. FileInputFormat类

- **FileInputFormat类位于org.apache.hadoop.mapreduce.lib.input包，是所有使用文件作为其数据源的InputFormat实现的基类，它不能直接使用，但提供了两个功能：指出作业的输入文件位置；输入文件生成分片的代码实现。**
- **MapReduce作业的输入被设定为一组路径，这对指定作业输入提供了很强的灵活性。**

7.3.2 文件输入

2. FileInputFormat的输入分片

- **FileInputFormat只切分大文件，大文件是指文件超过HDFS块的大小，因为通常分片大小是和HDFS块大小相同的。**

3. 小文件和CombineFileInputFormat

- **如果一个文件的大小比Block小，它将不会被划分。当Hadoop处理很多小文件的时候，由于FileInputFormat不会对小文件进行划分，所以每一个小文件都会被当做一个split并分配一个Map任务，导致效率低下。CombineFileInputFormat可以将多个文件打包到一个输入单元中，这样每次Map操作就会有更多的数据来处理。**

7.3.3 文本输入

1. TextInputFormat

- **TextInputFormat是默认的InputFormat，它把输入文件的每一行作为单独的一个记录（默认以换行符或回车符作为一行记录）。键是LongWritable类型，存储该行在整个文件中的字节偏移量；值是这行的内容，不包括行终止符（换行符、回车符），它被打包成一个Text对象。**

2. KeyValueTextInputFormat

- **KeyValueTextInputFormat会用分隔符将一行的数据分割为一个key-value键值对，分隔符前面的字符为key，后面的字符为value。它的默认值是一个制表符（\t）。**

7.3.3 文本输入

3. NLineInputFormat

- 如果用户希望**Mapper**收到固定行数的输入，需要将**NLineInputFormat**作为**InputFormat**，即**NLineInputFormat**可以设置每个**Mapper**处理的行数。它与**TextInputFormat**一样，键**key**是文件中行的字节偏移量，值**value**是行本身。

7.3.4 二进制输入

1. SequenceFileInputFormat

SequenceFileInputFormat用于读取sequence file，键和值由用户定义。序列文件是Hadoop专用的、压缩的、高性能的二进制文件格式。

2. SequenceFileAsTextInputFormat

是SequenceFileInputFormat的变体，它将键和值转换为Text对象。

3. SequenceFileAsBinaryInputFormat

是SequenceFileInputFormat的另一种变体，它将顺序文件的键和值作为二进制对象，它们被封装为BytesWritable对象。

7.3.5 多个输入

- 虽然一个**MapReduce**作业的输入可以包含多个输入文件，但所有文件都由同一个**InputFormat**和同一个**Mapper**来处理。然而，随着时间的推移数据格式可能会有所不同，那么就必须重写**Mapper**来处理应用中的数据格式问题。或者，有些数据源会提供相同的数据，但是格式却有了改变。
- 这些问题可以使用**MultipleInputs**类来妥善处理。它允许为每条输入路径指定**InputFormat**和**Mapper**。

7.3.6 数据库输入

- **DBInputFormat是一个使用JDBC并且从关系数据库中读取数据到HDFS上的一种输入格式。由于它没有任何共享能力，所以在访问数据库的时候必须非常小心，太多的Mapper读数据可能会使数据库承受不了。因此DBInputFormat最好在加载小量数据集的时候用。**

7.4 MapReduce的输出

- 处理输出格式的类都继承自**OutputFormat**类。
- **OutputFormat**位于**org.apache.hadoop.mapreduce**包，它的功能跟前面描述的**InputFormat**类相似，**OutputFormat**的实例会将输出的键值对写到本地磁盘或**HDFS**上。一般来说，**Mapper**类和**Reducer**类都会用到**OutputFormat**类。**Mapper**类用它存储中间结果，**Reducer**类用它存储最终结果。

7.4.1 文本输出

- **TextOutputFormat**是默认的输出格式，它将每条记录写为文本行。它的键和值可以是任意的类型，因为**TextOutputFormat**会调用**toString()**方法将它们转换为字符串。每个**key-value**键值对由制表符进行分割。

7.4.2 二进制输出

1. SequenceFileOutputFormat

它将key-value键值对写到一个序列文件。由于它的格式紧凑，很容易被压缩，所以易于作为MapReduce的输入。

2. SequenceFileAsBinaryOutputFormat

SequenceFileAsBinaryOutputFormat与SequenceFileAsBinaryInputFormat相对应，它把key-value键值对作为二进制格式写到一个序列文件中。

3.MapFileOutputFormat

MapFileOutputFormat把MapFile作为输出。MapFile中的key必须顺序添加，所以必须确保Reducer输出的key已经排好序。

7.4.3 多个输出

- **FileOutputFormat**及其子类产生的文件放在输出目录下。默认情况下，每个**Reducer**会产生一个文件并且文件由分区号命名：**part-r-00000**，**part-r-00001**等。
- **MultipleOutputFormat**类可以将数据写到多个文件中，这些文件名称源于输出的键和值。

7.4.4 延迟输出

- **FileOutputFormat**的子类在即使没有数据的情况下也会产生一个空文件。但有时候我们并不想创建空文件，这时候就可以使用**LazyOutputFormat**。**LazyOutputFormat**类只有在第一条数据真正输出的时候才会创建文件。

7.4.5 数据库输出

- **DBOutputFormat与DBInputFormat相对应，它将作业输出数据（中等规模的数据）存储到数据库中。**

7.5 MapReduce的任务

- MapReduce任务的执行是MapReduce程序的主体，每个任务之间不是独立的而是相互联系的。一个job的完成，依赖于所有任务的正确执行。
- 结合一个“求平均成绩”的例子来学习MapReduce的各个任务、每个任务所完成的功能以及它们之间的联系。
- 任务描述：对输入文件中数据进行计算学生平均成绩。输入文件中的每行内容均为一个学生的姓名和他相应的成绩，如果有多门学科，则每门学科为一个文件。要求在输出中每行有两个间隔的数据，其中，第一个代表学生的姓名，第二个代表其平均成绩。

7.5.1 Map任务

- 每个Map任务都是一个Java进程。从广义上来讲，Map任务会读取指定的HDFS中的文件，解析成很多的键值对，调用map方法处理后，转换为相应的键值对再输出。
- map函数会根据输入的key-vlaue键值对生成中间结果。
- map函数完成将类型为<LongWritable,Text>的键值对映射为类型为<Text,IntWritable>的键值对，获得<学生姓名，成绩>的键值对。

7.5.2 Combine任务

- 每一个Map都可能会产生大量的本地输出，Combine任务（也称为Combiner类）的作用就是对Map端的输出先做一次合并，以减少在Map和Reduce节点之间的数据传输量，以提高网络IO性能，是MapReduce的一种优化手段之一。

7.5.3 Partition任务

一般对Partitioner有两个要求：

1. 均衡负载，尽量的将工作均匀的分配给不同的Reduce。

2. 效率，分配速度一定要快。

MapReduce用户通常会指定Reduce任务和Reduce任务输出文件的数量（R）。用户在中间key上使用分区函数来对数据进行分区，之后在再输入到后续任务执行进程。

7.5.4 Reduce任务

Reduce任务有三个主要的阶段：

1. Shuffle。把来自Mapper的已经排序的输出数据通过网络经HTTP拷贝到本地。
2. Sort。MapReduce框架按key对Reducer输入进行合并、排序。因为不同的Mapper可能会输出同样的key给某个Reducer。
3. Reduce。此时，会为已排序的Reduce输入中的每个key-values对调用reduce函数。

reduce函数完成了将输入的学生成绩进行算术平均计算，最后得到每个学生的平均成绩。

AverageMapper.java | AverageScore.java

代码 | 拆分 | 设计 | 标题: | 检查页面

```java
import java.io.IOException;

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.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class AverageScore {

    public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {
        // TODO Auto-generated method stub
        Configuration conf = new Configuration(); //获得配置信息
        String[] ortherArgs = new GenericOptionsParser(conf,args).getRemainingArgs();
        //获取程序的输入参数
        Job job =Job.getInstance(conf,"Average Score");
        job.setJarByClass(AverageScore.class);
        job.setMapperClass(AverageMapper.class);       //设置Mapper类，执行map函数
        job.setReducerClass(AverageReducer.class);      //设置Reducer类，执行reduce函数
        job.setOutputKeyClass(Text.class);           //设置输出key格式
        job.setOutputValueClass(IntWritable.class);  //设置输出value格式
        FileInputFormat.addInputPath(job,new Path(ortherArgs[0]));     //文件输入路径
        FileOutputFormat.setOutputPath(job,new Path(ortherArgs[1]));      //结果输出路径
        System.exit(job.waitForCompletion(true)?0:1);    //提交job并等待结束

    }

}
```

AverageMapper.java

代码 拆分 设计 标题: 检查页面

```java
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class AverageMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
    private Text name = new Text();                    //定义一个Text对象，用来存储学生姓名
    private IntWritable score = new IntWritable();    //存储学生的成绩
//实现map函数
public void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException
{
StringTokenizer stk = new StringTokenizer(value.toString());
//将value转换为String后，创建一个StringTokenizer对象进行解析
        while(stk.hasMoreTokens())            //判断是否还有分隔符（有的话代表还有内容）
        {
          name.set(stk.nextToken()); //返回从当前位置到下一个分隔符的字符串，并将值赋给name
          score.set(Integer.parseInt(stk.nextToken())); //将得到的字符串转换为Int后赋给score
          context.write(name,score);          //将获得的key-value对写入map上下文
```

AverageMapper.java | AverageScore.java | AverageReducer.java

代码 | 拆分 | 设计 标题: 检查页面

```java
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class AverageReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
//实现reduce函数
public void reduce(Text key, Iterable<IntWritable> values,Context context)
 throws IOException, InterruptedException
{
int sum = 0;      //定义变量存储学生的总成绩
 int count = 0;   //定义变量存储学生的总科目
for(IntWritable val:values)           //遍历values中的每一个值
            {
                sum+=val.get();          //得到values中的值并进行累加，得到该学生的总成绩
                count++;                 //统计总科目
            }
int average = (int) sum / count;      //计算平均成绩
context.write(key, new IntWritable(average));   //将获得的key-value对写入reduce上下文
```

1 K / 1 秒

7.5.5 任务的配置与执行

- 完成上述任务代码的编写后，必须在job对象中进行相应设置才能正常运行。

7.6 MapReduce应用举例——倒排索引

- “倒排索引”是文档检索系统中最常用的数据结构，被广泛地应用于全文搜索引擎。它主要是用来存储某个单词（或词组）在一个文档或一组文档中的存储位置的映射，即提供了一种根据内容来查找文档的方式。由于不是根据文档来确定文档所包含的内容，而是进行相反的操作，称为倒排索引（**Inverted Index**）。
- 通常情况下，倒排索引由一个单词（或词组）以及相关的文档列表组成，文档列表中的文档可以是标识文档的**ID**号，也可以是指文档所在位置的**URL**。

7.6.2 准备数据

- 为验证输出的结果是否正确，准备了三个小的测试文件：file1.txt，file2.txt，file3.txt。

样本输入：

file1.txt

MapReduce is simple

file2.txt

MapReduce is powerful is simple

file3.txt

Hello MapReduce bye MapReduce

样本输出：

MapReduce	file1.txt:1;file2.txt:1;file3.txt:2;
is	file1.txt:1;file2.txt:2;
simple	file1.txt:1;file2.txt:1;
powerful	file2.txt:1;
Hello	file3.txt:1;
bye	file3.txt:1;

7.6.3 分析与设计

实现“倒排索引”需要关注的信息为：单词、文档URL及词频

1. Map过程

首先使用默认的TextInputFormat类对输入文件进行处理，得到文本中每行的偏移量及其内容。显然，Map过程首先必须分析输入的<key,value>键值对，得到倒排索引中需要的三个信息：单词、文档URL和词频。

2. Combine过程

Combine过程将key值相同的value值累加，得到一个单词在文档中的词频。

7.6.4 MapReduce编码实现

1. 编写InvertedIndexMap类实现map函数。
2. 编写InvertedIndexCombiner类实现Combine过程（本地reduce函数）。
3. 编写InvertedIndexReduce类实现reduce函数。
4. 编写InvertedIndex类实现main函数。

I:\...\src\MyMapper.java

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;


public class MyMapper extends Mapper<Object,Text,Text,Text> {
    private Text keyInfo = new Text();//存储单词和URI的组合 private Text valueInfo = new Text();//存储词频
    private FileSplit split;



    public void map(Object key,Text value,Context context)
       throws IOException,InterruptedException
       {
            split = (FileSplit)context.getInputSplit();

            StringTokenizer stk = new StringTokenizer(value.toString());

            while(stk.hasMoreTokens())
            {
                //由单词和URI组成key
                keyInfo.set(stk.nextToken()+":"+split.getPath().getName().toString());
                //设置词频初始值为1
                valueInfo.set("1");
                context.write(keyInfo,valueInfo);
            }
       }
}
```

I:\...\src\MyCombiner.java *

```
import java.io.IOException;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class MyCombiner extends Reducer<Text,Text,Text,Text> {
    private Text info = new Text();

    public void reduce(Text key, Iterable<Text> values,Context context)
    throws IOException,InterruptedException
    {
        int sum =0;
        for(Text value:values)
        {
            sum += Integer.parseInt(value.toString());
        }

        int splitIndex = key.toString().indexOf(":");
        //重新设置value值由URI和词频组成
        info.set(key.toString().substring(splitIndex+1)+":"+sum);
        //重新设置key值为单词
        key.set(key.toString().substring(0,splitIndex));
        context.write(key,info);
    }

}
```

I:\...\src\MyReducer.java

```
import java.io.IOException;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;


public class MyReducer extends Reducer<Text,Text,Text,Text>{
    private Text result = new Text();

    public void reduce(Text key, Iterable<Text> values,Context context)
       throws IOException,InterruptedException
       {
           String fileList = new String();
           for(Text value:values)
           {
               fileList += value.toString()+";";
           }
           result.set(fileList);
           context.write(key,result);
       }
}
```

I:\...\src\InvertedIndex.java

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;


public class InvertedIndex {

    public static void main(String[] args) throws Exception {
        // TODO Auto-generated method stub
        Configuration conf = new Configuration();
        String[] files=new GenericOptionsParser(conf,args).getRemainingArgs();
        if(files.length!=2)
        {
            System.err.println("Usage: invertedIndex <in> <out>");
            System.exit(2);
        }
        @SuppressWarnings("deprecation")
        Job job = new Job(conf,"InvertedIndex");
        job.setJarByClass(InvertedIndex.class);
        job.setMapperClass(MyMapper.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);

        job.setCombinerClass(MyCombiner.class);
        job.setReducerClass(MyReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        FileInputFormat.addInputPath(job,new Path(files[0]));
        FileOutputFormat.setOutputPath(job,new Path(files[1]));

        System.exit(job.waitForCompletion(true)?0:1);
    }

}
```

7.6.5 测试结果

1. 上传数据。
2. 配置运行参数。
3. 运行程序。