第7章

细则手册

7

本章内容

- 向任务传递定制参数
- 获取任务特定的信息
- 生成多个输出
- 与关系数据库交互
- 让输出做全局排序

本书到此已经涵盖了编写MapReduce程序的核心技术。Hadoop是一个巨大的框架，它所支持的功能要比那些核心技术多得多。在这个必应和谷歌的时代，你可以很容易地搜索到特定的MapReduce技术，我们并不试图成为百科全书式的参考书。通过自身的使用经验以及与其他Hadoop用户的讨论，我们发现了一些通常很有用的技巧，例如可以用一个标准的关系数据库作为MapReduce作业的输入或输出的技巧。本章收集了我们最喜欢的一些“配方”。

7.1 向任务传递作业定制的参数

在编写Mapper和Reducer时，通常会想让一些地方可以配置。例如，第5章的联结程序被固定地写为取第一个数据列作为联结键。如果用户可以在运行时指定某个列作为联结键，就会让程序更具普适性。Hadoop自身使用一个配置对象来存储所有作业的配置属性。你也可以使用这个对象将参数传递到Mapper和Reducer。

我们已经知道MapReduce的driver是如何用属性来配置JobConf对象的，这些属性包括输入格式、输出格式、Mapper类等。若要引入自己的属性，需要在这个配置对象中，给属性一个唯一的名称并设置它的值。这个配置对象会被传递给所有的TaskTracker，然后作业中的所有任务就能够看到配置对象中的属性。Mapper和Reducer也就可以读取该配置对象并获得它的属性值。

Configuration类（JobConf的父类）有许多通用的setter方法。属性采用键/值对的形式，键必须是一个String，而值可以是常用类型的任意一个。常用setter方法的签名为

```
public void set(String name, String value)
public void setBoolean(String name, boolean value)
public void setInt(String name, int value)
```

```
public void setLong(String name, long value)
public void setStrings(String name, String... values)
```

请注意在Hadoop内部，所有的属性都存为字符串。在set(String,String)方法之外的所有其他方法都是它的便捷方法。例如，setStrings(String, String...)方法取一个String数组，把它变成一个单一的以逗号分隔的String，并把这个String设置为属性值。同样，GetStrings()方法将这个连结好的字符串拆分后放入一个数组中。考虑到这一点，在原始数组的字符串中不要出现逗号。如果需要有逗号，则应使用自定义的字符串编码函数。

Driver会首先设置配置对象中的属性，让它们在所有任务中可见。Mapper和Reducer可以访问configure()方法中的配置对象。任务初始化时会调用configure()，它已经被覆写为可以提取和存储你设置的属性。之后，map()和reduce()方法会访问这些属性的副本。在下面的示例中，我们调用新的属性myjob.myproperty，它用一个由用户指定的整数值。

```
public int run(String[] args) throws Exception {

    Configuration conf = getConf();
    JobConf job = new JobConf(conf, MyJob.class);

    ...

    job.setInt("myjob.myproperty", Integer.parseInt(args[2]));   ❶ 设置特定属性

    JobClient.runJob(job);
    return 0;
}
```

在MapClass中，configure()方法取出属性值，并将它存储在对象的范围中。Configuration类的getter方法需要指定默认的值，如果所请求的属性未在配置对象中设置，就会返回默认值。在这个例子中，我们取默认值为0：

```
public static class MapClass extends MapReduceBase
    implements Mapper<Text, Text, Text, Text> {

    int myproperty;

    public void configure(JobConf job) {
        myproperty = job.getInt("myjob.myproperty", 0);   ❶ 得到特定属性
    }
    ...
}
```

如果你希望在Reducer中使用该属性，Reducer也必须检索这个属性。

```
public static class Reduce extends MapReduceBase
    implements Reducer<Text, Text, Text, Text> {

    int myproperty;

    public void configure(JobConf job) {
        myproperty = job.getInt("myjob.myproperty", 0);
    }

    ...
```

Configuration类中getter方法的列表比setter方法更长，但是它们大多一目了然。几乎所有的getter方法都需要将参数设置为默认值。唯一例外的是get(String)，如果没有设置特定的名称，它就返回null。

```
public String get(String name)
public String get(String name, String defaultValue)
public boolean getBoolean(String name, boolean defaultValue)
public float getFloat(String name, float defaultValue)
public int getInt(String name, int defaultValue)
public long getLong(String name, long defaultValue)
public String[] getStrings(String name, String... defaultValue)
```

既然我们的job类实现了Tool接口并使用了ToolRunner，我们还可以让用户直接使用通用的选项来配置定制化的属性，方法与用户设置Hadoop的配置属性相同。

```
bin/hadoop jar MyJob.jar MyJob -D myjob.myproperty=1 input output
```

我们可以将driver中总是需要用户通过参数来设定属性值的那行代码删掉。如果在大多数时间里默认值是可用的，这样做会让用户感觉更加方便。

```
public int run(String[] args) throws Exception {

    Configuration conf = getConf();
    JobConf job = new JobConf(conf, MyJob.class);

    ...

    Int myproperty = job.getInt("myjob.myproperty", 0);
    if (myproperty< 0) {
        System.err.println("Invalid myjob.myproperty: " + myproperty);
        System.exit(0);
    }

    JobClient.runJob(job);
    return 0;
}
```

当你允许用户设定特定属性时，在driver中最好对用户的输入进行验证。上面的例子可以确保用户设定的myjob.myproperty不是负数。

7.2 探查任务特定信息

除了获取自定义属性和全局配置之外，我们还可以使用配置对象上的getter方法获得当前任务和作业状态的一些信息。例如，在Mapper中，你可以通过map.input.file属性来得到当前map任务的文件路径。在datajoin软件包的DataJoinMapperBase中，这正是configure()方法中所做的，即用一个标签来表示数据源。

```
this.inputFile = job.get("map.input.file");
this.inputTag = generateInputTag(this.inputFile);
```

表7-1列出了一些其他的任务特定状态信息。

表7-1　在配置对象中可获得的任务特定状态信息

属　性	类　型	描　述
mapred.job.id	String	作业ID
mapred.jar	String	作业目录中jar的位置
job.local.dir	String	作业的本地空间
mapred.tip.id	String	任务ID
mapred.task.id	String	任务重试ID
mapred.task.is.map	boolean	标志量，表示是否为一个map任务
mapred.task.partition	int	作业内部的任务ID
map.input.file	String	Mapper读取的文件路径
map.input.start	long	当前Mapper输入分片的文件偏移量
map.input.length	long	当前Mapper输入分片中的字节数
mapred.work.output.dir	String	任务的工作（即临时）输出目录

配置属性还可以通过环境变量为Streaming程序所用。在执行脚本之前，Streaming的API会把所有的配置属性添加到运行环境中。属性名被重新格式化，使得非字母数字编码的字符都被下划线替代。例如，一个Streaming脚本会在环境变量map_input_file中查找当前mapper所读取文件的完整路径。

```
import os
filename = os.environ["map_input_file"]
localdir = os.environ["job_local_dir"]
```

上面的代码显示了在Python中如何访问配置属性。

7.3　划分为多个输出文件

直到现在我们看到的所有MapReduce作业都输出一组文件。然而，经常在有些场景下，输出多组文件或把一个数据集分为多个数据集更为方便。一个常见的案例将一个大的日志文件按天划分到不同的日志文件中。

MultipleOutputFormat提供了一个简单的方法，将相似的记录结组为不同的数据集。在写每条输出记录之前，这个OutputFormat类调用一个内部方法来确定要写入的文件名。更具体地说，你将扩展MultipleOutputFormat的某个特定子类，并实现generateFileNameForKeyValue()方法。你扩展的子类将决定输出的格式。例如，MultipleTextOutputFormat将输出文本文件，而MultipleSequenceFileOutputFormat将输出序列文件。无论哪种情况，你会覆写下面的方法以返回每个输出键/值对的文件名：

```
protected String generateFileNameForKeyValue(K key, V value, String name)
```

默认实现返回参数name，即叶文件名。你可以让该方法根据记录的内容返回文件名。

在此示例中，我们取专利的元数据并按国家对它进行分区。来自美国发明者的所有专利将都进入一组文件，而来自日本的所有专利进入另一个，以此类推。该示例程序的框架是一个map-only的作业，取得输入后立即将其输出。我们做的主要变化是创建自己的MultipleTextOutputFormat

子类，名为PartitionbyCountryMTOF。（请注意MTOF是MultipleTextOutputFormat的缩写。）我们的子类将基于该记录列出的发明专利所属国家，将每条记录存储到相应的位置。因为我们将generateFileNameForKeyValue()的返回值作为文件路径，通过返回的country + "/" + filename，我们可以为每个国家创建一个子目录。请参阅代码清单7-1。

代码清单7-1 根据国家将专利元数据分割到多个目录中

```
public class MultiFile extends Configured implements Tool {

public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, NullWritable, Text> {

    public void map(LongWritable key, Text value,
                    OutputCollector<NullWritable, Text> output,
                    Reporter reporter) throws IOException {

        output.collect(NullWritable.get(), value);
    }
}

public static class PartitionByCountryMTOF
    extends MultipleTextOutputFormat<NullWritable,Text>
{
    protected String generateFileNameForKeyValue(NullWritable key,
                                                 Text value,
                                                 String filename)
    {
        String[] arr = value.toString().split(",", -1);
        String country = arr[4].substring(1,3);
        return country + "/" + filename;
    }
}

public int run(String[] args) throws Exception {
    Configuration conf = getConf();

    JobConf job = new JobConf(conf, MultiFile.class);

    Path in = new Path(args[0]);
    Path out = new Path(args[1]);
    FileInputFormat.setInputPaths(job, in);
    FileOutputFormat.setOutputPath(job, out);

    job.setJobName("MultiFile");
    job.setMapperClass(MapClass.class);

    job.setInputFormat(TextInputFormat.class);
    job.setOutputFormat(PartitionByCountryMTOF.class);
    job.setOutputKeyClass(NullWritable.class);
    job.setOutputValueClass(Text.class);

    job.setNumReduceTasks(0);

    JobClient.runJob(job);

    return 0;
```

```
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(),
                                 new MultiFile(),
                                 args);

        System.exit(res);
    }
}
```

在上述程序执行以后，我们会在输出目录中看到，现在每个国家都有一个单独的目录。

```
ls output/
```

AD	BN	CS	GE	IN	LC	MT	PH	SV	VE
AE	BO	CU	GF	IQ	LI	MU	PK	SY	VG
AG	BR	CY	GH	IR	LK	MW	PL	SZ	VN
AI	BS	CZ	GL	IS	LR	MX	PT	TC	VU
AL	BY	DE	GN	IT	LT	MY	PY	TD	YE
AM	BZ	DK	GP	JM	LU	NC	RO	TH	YU
AN	CA	DO	GR	JO	LV	NF	RU	TN	ZA
AR	CC	DZ	GT	JP	LY	NG	SA	TR	ZM
AT	CD	EC	GY	KE	MA	NI	SD	TT	ZW
AU	CH	EE	HK	KG	MC	NL	SE	TW	
AW	CI	EG	HN	KN	MG	NO	SG	TZ	
AZ	CK	ES	HR	KP	MH	NZ	SI	UA	
BB	CL	ET	HT	KR	ML	OM	SK	UG	
BE	CM	FI	HU	KW	MM	PA	SM	US	
BG	CN	FO	ID	KY	MO	PE	SN	UY	
BH	CO	FR	IE	KZ	MQ	PF	SR	UZ	
BM	CR	GB	IL	LB	MR	PG	SU	VC	

而且在每个国家的目录中的文件都是仅由这些国家所创建的记录（专利）。

```
ls output/AD
part-00003      part-00005      part-00006

head output/AD/part-00006
5765303,1998,14046,1996,"AD","",,1,12,42,5,59,11,1,0.4545,0,0,1,67.3636,,,,
5785566,1998,14088,1996,"AD","",,1,9,441,6,69,3,0,1,,0.6667,,4.3333,,,,
5894770,1999,14354,1997,"AD","",,1,,82,5,51,4,0,1,,0.625,,7.5,,,,
```

我们编写的这个简单的分区练习是一个map-only程序。你也可以对reducer的输出应用相同的技术。请注意不要与MapReduce框架中的partitioner混淆。partitioner查看中间记录的键，并决定哪些Reducer将处理它们。这里我们要做的分割是查看输出的键/值对，并决定存储到哪个文件中。

MultipleOutputFormat很简单，但也有局限。例如，我们可以按行拆分输入数据，但如果我们想按列拆分会该怎么做？假设我们要从专利的元数据中创建两个数据集：一个包含每个专利的时间相关信息（例如发布日期），另一种包含地理信息（例如发明的国家）。这两个数据集可能有不同的输出格式以及不同数据类型的键和值。我们可以用在Hadoop 0.19版本中引入的MultipleOutputs，以获得更强的能力。

MultipleOutputs所采用的方法不同于MultipleOutputFormat。它不是要求给每条记录请求文件名，而是创建多个OutputCollector。每个OutputCollector可以有自己的OutputFormat和键/值对的类型。MapReduce程序将决定如何向每个OutputCollector输出数据。代码清单7-2

显示了一个程序，取出专利的元数据，并输出两个数据集。一个包含时间顺序的信息，例如发布日期。另一个数据集包含与每个专利相关的地理信息。它也是个map-only程序，但你可以直接在reducer上使用多个输出收集器。

代码清单7-2 将输入数据的不同列提取为不同文件的程序

```
public class MultiFile extends Configured implements Tool {

    public static class MapClass extends MapReduceBase
        implements Mapper<LongWritable, Text, NullWritable, Text> {

    private MultipleOutputs mos;
    private OutputCollector<NullWritable, Text> collector;

    public void configure(JobConf conf) {
        mos = new MultipleOutputs(conf);
    }

    public void map(LongWritable key, Text value,
                    OutputCollector<NullWritable, Text> output,
                    Reporter reporter) throws IOException {

        String[] arr   = value.toString().split(",", -1);
        String chrono = arr[0] + "," + arr[1] + "," + arr[2];
        String geo    = arr[0] + "," + arr[4] + "," + arr[5];

        collector = mos.getCollector("chrono", reporter);
        collector.collect(NullWritable.get(), new Text(chrono));
        collector = mos.getCollector("geo", reporter);
        collector.collect(NullWritable.get(), new Text(geo));
    }

    public void close() throws IOException {
        mos.close();
    }
}

public int run(String[] args) throws Exception {
    Configuration conf = getConf();

    JobConf job = new JobConf(conf, MultiFile.class);

    Path in = new Path(args[0]);
    Path out = new Path(args[1]);
    FileInputFormat.setInputPaths(job, in);
    FileOutputFormat.setOutputPath(job, out);

    job.setJobName("MultiFile");
    job.setMapperClass(MapClass.class);

    job.setInputFormat(TextInputFormat.class);
    job.setOutputKeyClass(NullWritable.class);
    job.setOutputValueClass(Text.class);
    job.setNumReduceTasks(0);

    MultipleOutputs.addNamedOutput(job,
                                   "chrono",
                                   TextOutputFormat.class,
```

```
                                        NullWritable.class,
                                        Text.class);
        MultipleOutputs.addNamedOutput(job,
                                        "geo",
                                        TextOutputFormat.class,
                                        NullWritable.class,
                                        Text.class);

        JobClient.runJob(job);

        return 0;
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(),
                                 new MultiFile(),
                                 args);

        System.exit(res);
    }
}
```

若要使用MultipleOutputs，MapReduce程序的driver必须设置它想要使用的输出收集器。创建收集器涉及调用MultipleOutputs的静态方法addNamedOutput()。我们已经创建了两个输出收集器，一个称为chrono和另一个称为geo。它们都使用TextOutputFormat，并具有相同的键/值类型，但我们也可以选择使用不同的输出格式或数据类型。

设置完driver中的输出收集器后，我们需要获取MultipleOutputs对象，当configure()方法中的mapper被初始化时，它会跟踪这些输出收集器。该对象必须在整个map任务的生命期中可用。在map()函数自身，我们可以在MultipleOutputs对象中调用getCollector()方法取回chrono和geo的OutputCollectors。我们将不同的数据输出到每个合适的输出收集器中。

我们已经为MultipleOutputs中的每个输出收集器赋予了一个名字，而MultipleOutputs会自动生成输出文件名。我们可以通过脚本来查看这些文件，看看MultipleOutputs是如何生成输出文件名的：

```
ls -l output/
total 101896
-rwxrwxrwx 1 Administrator None 9672703 Jul 31 06:28 chrono-m-00000
-rwxrwxrwx 1 Administrator None 7752888 Jul 31 06:29 chrono-m-00001
-rwxrwxrwx 1 Administrator None 6884496 Jul 31 06:29 chrono-m-00002
-rwxrwxrwx 1 Administrator None 6933561 Jul 31 06:29 chrono-m-00003
-rwxrwxrwx 1 Administrator None 7164558 Jul 31 06:29 chrono-m-00004
-rwxrwxrwx 1 Administrator None 7273561 Jul 31 06:29 chrono-m-00005
-rwxrwxrwx 1 Administrator None 8281663 Jul 31 06:29 chrono-m-00006
-rwxrwxrwx 1 Administrator None 9428951 Jul 31 06:28 geo-m-00000
-rwxrwxrwx 1 Administrator None 7464690 Jul 31 06:29 geo-m-00001
-rwxrwxrwx 1 Administrator None 6580482 Jul 31 06:29 geo-m-00002
-rwxrwxrwx 1 Administrator None 6448648 Jul 31 06:29 geo-m-00003
-rwxrwxrwx 1 Administrator None 6432392 Jul 31 06:29 geo-m-00004
-rwxrwxrwx 1 Administrator None 6546828 Jul 31 06:29 geo-m-00005
-rwxrwxrwx 1 Administrator None 7450768 Jul 31 06:29 geo-m-00006
-rwxrwxrwx 1 Administrator None       0 Jul 31 06:28 part-00000
-rwxrwxrwx 1 Administrator None       0 Jul 31 06:28 part-00001
```

```
-rwxrwxrwx 1 Administrator None      0 Jul 31 06:29 part-00002
-rwxrwxrwx 1 Administrator None      0 Jul 31 06:29 part-00003
-rwxrwxrwx 1 Administrator None      0 Jul 31 06:29 part-00004
-rwxrwxrwx 1 Administrator None      0 Jul 31 06:29 part-00005
-rwxrwxrwx 1 Administrator None      0 Jul 31 06:29 part-00006
```

我们有一组以chrono为前缀的文件，以及另一组以geo为前缀的文件。请注意该程序创建了默认的输出文件part-*，即使它没有显示地写任何东西。这些文件是完全有可能通过map()方法中的原始OutputCollector来写的。事实上，如果这不是一个map-only的程序，只有那些被写到原始OutputCollector中的记录才会被传递到reducer上加工。

使用MultipleOutputs需要权衡的一个地方是它有一个比MultipleOutputFormat更为严格的命名结构。输出收集器的名称不能为part，因为它已经被用作默认值。输出文件名还严格定义为在输出收集器的名称之后，要跟m或r，这取决于是在Mapper还是在Reducer上收集输出。最后，还要跟一个分区号。

```
head output/chrono-m-00000
"PATENT","GYEAR","GDATE"
3070801,1963,1096
3070802,1963,1096
3070803,1963,1096
3070804,1963,1096
3070805,1963,1096
3070806,1963,1096
3070807,1963,1096
3070808,1963,1096
3070809,1963,1096

head output/geo-m-00000
"PATENT","COUNTRY","POSTATE"
3070801,"BE",""
3070802,"US","TX"
3070803,"US","IL"
3070804,"US","OH"
3070805,"US","CA"
3070806,"US","PA"
3070807,"US","OH"
3070808,"US","IA"
3070809,"US","AZ"
```

查看输出文件，可以看到我们已经成功地将专利数据集中的列抽取到不同的文件中。

7.4 以数据库作为输入输出

虽然Hadoop善于处理较大数据，但在许多数据处理应用中，关系数据库是仍然是主力。很多时候Hadoop会需要与数据库进行接口。

虽然有可能建立一个MapReduce程序通过直接查询数据库来取得输入数据，而不是从HDFS中读取文件，但其性能不甚理想。更多时候，你需要将数据集从数据库复制到HDFS中。你可以很容易地通过标准的数据库工具dump，来取得一个flat文件。然后使用HDFS的shell命令put将它上传到HDFS中。

但是有时更合理的做法是让MapReduce程序直接写入数据库。许多MapReduce程序获取大型数据集，并把它们处理到数据库可管理的大小。例如，我们经常在ETL之类的过程中用MapReduce获取堆积如山的日志文件，并计算出一个更小、更易于管理的统计数据集供分析者查看。

DBOutputFormat是用于访问数据库的关键类。在driver中，你可以将输出格式设置为这个类。你需要指定配置，以便能够联结到该数据库。你可以通过在DBConfiguration中的静态方法configureDB()做到这一点：

```
public static void configureDB(JobConf job, String driverClass,
    ➥ String dbUrl, String userName, String passwd)
```

之后，你要指定将要写入的表，以及那里有哪些字段。这是通过在DBOutputFormat中的静态setOutput()方法做到的。

```
public static void setOutput(JobConf job, String tableName,
    ➥ String... fieldNames)
```

你的driver应该包含如下样式的几行代码：

```
conf.setOutputFormat(DBOutputFormat.class);
DBConfiguration.configureDB(job,
                            "com.mysql.jdbc.Driver",
                            "jdbc:mysql://db.host.com/mydb",
                            "username",
                            "password")
DBOutputFormat.setOutput(job, "Events", "event_id", "time");
```

使用DBOutputFormat将强制你输出的键实现DBWritable接口。只有这个键会被写入到数据库中。通常，键必须实现Writable接口。Writable和DBWritable的签名类似；只是它们的参数类型是不同的。在Writable中的write()方法用DataOutput，而DBWritable中的write()用PreparedStatement。类似地，用在Writable中的readFields()方法采用DataInput，而DBWritable中的readFields()采用ResultSet。除非你打算使用DBInputFormat直接从数据库中读取输入的数据，在DBWritable中的readFields()将永远不会被调用。

```
public class EventsDBWritable implements Writable, DBWritable {
    private int id;
    private long timestamp;

    public void write(DataOutput out) throws IOException {
    out.writeInt(id);
    out.writeLong(timestamp);
    }

    public void readFields(DataInput in) throws IOException {
        id = in.readInt();
        timestamp = in.readLong();
    }

    public void write(PreparedStatement statement) throws SQLException {
        statement.setInt(1, id);
        statement.setLong(2, timestamp);
    }
```

```
    public void readFields(ResultSet resultSet) throws SQLException {
        id = resultSet.getInt(1);
        timestamp = resultSet.getLong(2);
    }
}
```

我们想再次强调的是，从Hadoop内部读写数据库仅适用于凭借Hadoop的标准来说相对较小的数据集。除非你的数据库是与Hadoop并行的（如果你Hadoop集群相对较小而你的数据库系统中有许多分片，就可以这样设置），否则，你的数据库将成为性能瓶颈，而无法利用Hadoop集群的可扩展性优势。很多时候，最好批量地将数据装载到数据库，而不是从Hadoop上直接写入。对于非常大的数据库，你需要采用定制化的解决方案。[①]

7.5 保持输出的顺序

MapReduce框架保证每个reducer的输入都按键来排序。在许多情况下，reducer只是对键/值对中值的部分做简单的计算。输出仍保持顺序排序。请记住MapReduce框架并不能保证reducer输出的顺序。它只是已经排序好的输入以及reducer所执行的典型操作类型的一种副产品。

对于某些应用，这种排序是没有必要的，有时就会有疑问，是否可以关闭排序操作来消除reducer中不必要的步骤。事实上，排序操作的目的并不是强制对reducer的输入进行排序。相反，排序是将相同键的所有记录进行组合的一种有效途径。如果分组功能是不必要的，那么我们就可以直接从单个输入记录中生成输出记录。在这种情况下，你就可以消除整个reduce阶段来提高性能。你可以将reducer的个数设置为0，从而让应用程序成为一个map-only作业。

另外，对于某些应用程序，很好将所有的输出做整体排序。（由一个reducer生成的）每个输出文件都已经排好序了；很好的结果是所有`part-00000`的记录都小于`part-00001`，而`part-00001`又都小于`part-00002`，以此类推。要做到这点，关键还在于框架中的partitioner操作。

Partitioner的任务是确定地为每个键分配一个reducer。相同键的所有记录都结成组并在reduce阶段被集中处理。Partitioner的一个重要设计需求是在reducer之间达到负载平衡；没有一个reducer会比其他的reducer被分配更多的键。由于没有以前对键的分配信息，partitioner默认使用散列函数来均匀地将键分配给reducer。这通常可以很好地在reducer之间均匀地分配工作，但分配是完全随意的，而不存在任何顺序。如果事先知道键是大致均匀分布的，我们就可以使用一个partitioner给每个reducer分配一个键的范围，仍然可以确保reducer的负载是相对均衡的。

注意 如果某些键的处理比其他键要占用更多的时间，散列分区可能也无法均匀分布作业。例如，在高度倾斜的数据集中，大量的记录可能有相同的键。如果可能的话，你应该使用combiner在map阶段尽可能多地做预处理，来减轻reduce阶段的负载。此外，你还可以选择写一个特殊的partitioner来非均匀地分配键，使其可以平衡这个倾斜的数据及其处理。

① LinkedIn有一个有趣的博客，讨论了将离线处理（即Hadoop）的海量数据结果移动到在线系统所面临的挑战：http://project-voldemort.com/blog/2009/06/building-a-self-healing-key-value-store-with-hadoop-and-project-voldemort/。

TotalOrderPartitioner是一个可以保证在输入分区之间，而不仅仅是分区内部排序的partitioner。大规模数据的排序（即TeraSort基准测试）最初使用的就是这个类的一个类似版本。这个类利用一个排好序的分区键组读取一个序列文件，并进一步将不同区域的键分配到reducer上。

7.6　小结

本章讨论了许多工具和技术来让你的Hadoop作业对用户更友好，或使它可以更好地与数据处理平台中的其他组件接口。一个Hadoop作业可获得的全部支持在Hadoop的API中有详细的描述：http://hadoop.apache.org/common/docs/current/api/index.html。你还可以使用更多的抽象概念（如pig和Hive）以简化你的编程。我们会在第10章和第11章介绍这些工具。

如果你的角色涉及管理一个Hadoop集群，你会在下一章发现管理Hadoop集群的那些有用的技巧。