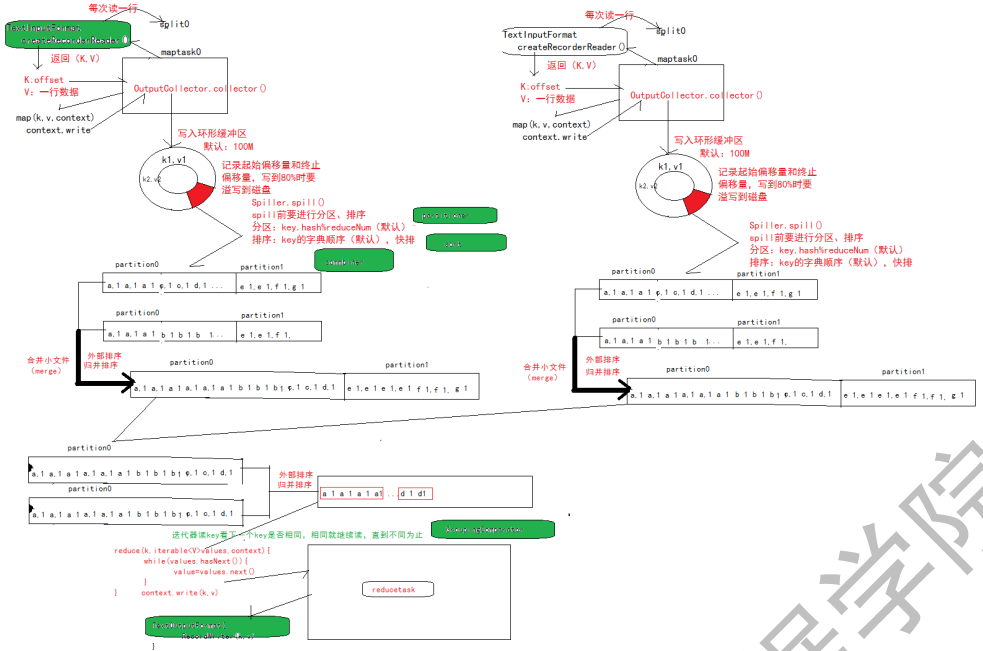
2.4.8、maptask执行流程详解（源码）



maptask调用FileInputFormat的createRecordReader读取分片数据

每行数据读取一次，返回一个(K,V)对，K是offset,V是一行数据

将k-v对交给maptask处理

每对k-v调用一次map(K,V，context)方法，然后context.write(k,v)

写出的数据交给收集器OutputCollector.collector()处理

将数据写入环形缓冲区，并记录写入的起始偏移量，终止偏移量，环形缓冲区默认大小100M

默认写到80%的时候要溢写磁盘，溢写磁盘的过程中数据继续写入剩余20%

溢写磁盘之前要先进行分区然后分区内进行排序

默认的分区规则是hashpatitioner，即key的hash%reduceNum

默认的排序规则是key的字典顺序，使用的是快速排序

溢写会形成多个文件，在maptask读取完一个分片数据后，先将环形缓冲区数据刷写到磁盘

将数据多个溢写文件进行合并，分区内排序（外部排序===》归并排序）

2.4.9、reduceTask执行详解

数据按照分区规则发送到reducetask

reducetask将来自多个maptask的数据进行合并，排序（外部排序===》归并排序）

按照key相同分组（ ）

一组数据调用一次reduce(k,iterablevalues,context)

处理后的数据交由reducetask

reducetask调用FileOutputFormat组件

FileOutputFormat组件中的write方法将数据写出

2.4.10、MapReduce的shuffle过程

shuffle过程从map写数据到环形缓冲区到reduce读取数据合并(见maptask和reducetask执行过程)

2.4.11、MR的本地运行

mr的本地执行需要对应的模拟器（winutils.exe）和动态类库（hadoop.dll），将这两个文件放入hadoop的安装目录下

如果运行报错加入org包放入java目录下

```
     public static void main(String[] args) throws InterruptedException,
IOException, ClassNotFoundException {
        //1、配置连接hadoop集群的参数
        Configuration conf = new Configuration();
         //设置本地
        conf.set("mapreduce.framework.name","local");
         //提交给本地localRunner模拟集群运行
        conf.set("fs.defaultFS","file:///");


         //可以设置从集群读取数据在本地运行
        conf.set("fs.defaultFS","hdfs://qianfeng");
        //解决用户操作权限问题的方法之二，临时设置环境变量HADOOP_USER_NAME=root
        //System.setProperty("HADOOP_USER_NAME", "root");

        //2、获取job对象实例
        Job job = Job.getInstance(conf,"FLOWCOUNT");
        //3、指定本业务job的路径
        job.setJarByClass(FlowCount.class);

        //4、指定本业务job要使用的Mapper类
        job.setMapperClass(MyMapper.class);

        //5、指定mapper类的输出数据的kv的类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(FlownBean.class);

        //6、指定本业务job要使用的Reducer类
        job.setReducerClass(MyReducer.class);

        //7、设置程序的最终输出结果的kv的类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FlownBean.class);

        //8、设置job要处理的数据的输入源
        FileInputFormat.setInputPaths(job,new Path("d:/data1/flow"));

        //判断输出目录是否存在，如果存在，则删除之
        //9、设置job的输出目录
```

```
        FileOutputFormat.setOutputPath(job,new Path("d:/data1/flow-output"));
        //10、提交job
        boolean b = job.waitForCompletion(true);
        System.exit(b?0:1);
    }
}
```

2.4.12、自定义组件combiner

1.Combiner是MR程序中Mapper和Reduce之外的一种组件

2.Combiner组件的父类就是Reducer

3.Combiner和Reducer之间的区别在于运行的位置

4.Reducer是每一个接收全局的Map Task 所输出的结果

5.Combiner是在MapTask的节点中运行

6.每一个map都会产生大量的本地输出，Combiner的作用就是对map输出的结果先做一次合并，以较少的map和reduce节点中的数据传输量

7.Combiner的存在就是提高当前网络IO传输的性能，也是MapReduce的一种优化手段。

8.Combiner的具体实现：

wordcount案例中直接调用写好的reduce，即在job的设置中加入

job.setCombinerClass(WordCountReduce.class);

```
public class WordCountRunner {
    public static void main(String[] args) {
        try {
            Configuration conf = new Configuration();
            //获取job并携带参数
            Job job = Job.getInstance(conf,"wordCount");
            //可以用job对象封装一些信息
            //首先程序是一个jar包，就要指定jar包的位置
            //将jar包放在root目录下
            //可以将这个程序打包为pv.jar,上传到linux机器上
            //使用命令运行
            //hadoop jar /root/pv.jar pvcount.PvCountRunner /data/pvcount
/out/pvcount
            //job.setJar("d:/word.jar");
            /**
             * 一个jar包中可能有多个job,所以我们要指明job使用的是哪儿一个map类
             * 哪儿一个reduce类
             */
            job.setMapperClass(WordCountMapper.class);
            job.setReducerClass(WordCountReduce.class);

            /**
             * map端输出的数据要进行序列化，所以我们要告诉框架map端输出的数据类型
             */
            job.setMapOutputKeyClass(Text.class);
            job.setMapOutputValueClass(IntWritable.class);
            /**
             * reduce端要输出，所有也要指定数据类型
             */
```

```
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);
            /**
             * 告诉框架用什么组件去读数据，普通的文本文件，就用TextInputFormat
             * 导入长包
             */
            job.setInputFormatClass(TextInputFormat.class);
            /**
             * 告诉这个组件去哪儿读数据
             * TextInputFormat有个父类FileInputFormat
             * 用父类去指定到哪儿去读数据
             * 输入路径是一个目录，该目录下如果有子目录需要进行设置递归遍历，否则会报错
             */
            FileInputFormat.addInputPath(job,new Path(args[0]));
            //设置写出的组件
            job.setOutputFormatClass(TextOutputFormat.class);
            //设置写出的路径
            FileOutputFormat.setOutputPath(job,new Path(args[1]));
            job.setCombinerClass(WordCountReduce.class);
            /**
             * 信息设置完成，可以调用方法向yarn去提交job
             * waitForCompletion方法就会将jar包提交给RM
             * 然后rm可以将jar包分发，其他机器就执行
             */
            //传入一个boolean类型的参数，如果是true,程序执行会返回true/flase
            //如果参数传入true,集群在运行时会有进度，这个进度会在客户端打印
            boolean res = job.waitForCompletion(true);
            /**
             * 客户端退出后会返回一个状态码，这个状态码我们可以写shell脚本的时候使用
             * 根据返回的状态码的不同区执行不同的逻辑
             */
            System.exit(res? 0:1);
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```


2.4.13 Hadoop序列化类型，以及自定义类型

MapReduce序列化，序列化是指将结构化对象转为字节流以便于通过网络进行传输或写入持久存储的过程。反序列化指的是将字节流转为结构化对象的过程。在Hadoop MapReduce中，序列化的主要作用有两个：永久存储和进程间通信。

为了能够读取或者存储Java对象，MapReduce编程模型要求用户输入和输出数据中的key和value必须是可序列化的。在Hadoop MapReduce中，使一个Java对象可序列化的方法是让其对应的类实现Writable接口。但对于key而言，由于它是数据排序的关键字，因此还需要提供比较两个key对象的方法。为此，key对应类需实现WritableComparable接口

2.4.14 案例：手机流量的汇总(自定义数据类型)

- **需求:**对于记录用户手机信息的文件，得出统计每一个用户（手机号）所耗费的总上行流量、下行流量，总流量结果
- **实现思路**：实现自定义的 bean 来封装流量信息，使用手机号码作为Key,Bean作为value。这个Bean的传输需要实现可序列化，Java类中提供的序列化方法中会由很多冗余信息（继承关系，类

信息）是我们不需要的，而这些信息在传输中占据大量资源，会导致有效信息传输效率减低。因此我们需要实现MapReduce的序列化接口Writable,自定义方法实现。

计算上行流量、下行流量、计费流量

取一行TextInputFormat类去读取，offset做key,一行数据做value，拆分，取出倒数第二倒数第三段

map端读取数据然后输出

offset:phoneNum,upflow,downflow

phoneNum:upfolw,downflow

phoneNum:{upfolw_downflow;upfolw1_downflow2}

phoneNum:totalupflow,totaldownflow,totalflow

考虑把字符串拼接的方式存储数据改写成类去存储数据，这个类三个属性，upflow,downflow,totalflow，这些数据在hadoop框架要进行序列化和反序列化

所以要实现writable接口，重写序列化和反序列方法

```
public class FlowBean implements Writable{
    long upflow;
    long downflow;
    long sumflow;

    //如果空参构造函数被覆盖，一定要显示定义一下，否则在反序列化时会抛出异常
    public flowBean() {
    }

    public flowBean(long upflow, long downflow) {
        this.upflow = upflow;
        this.downflow = downflow;
        this.sumflow=upflow+downflow;
    }

    public long getUpflow() {
        return upflow;
    }

    public void setUpflow(long upflow) {
        this.upflow = upflow;
    }

    public long getDownflow() {
        return downflow;
    }

    public void setDownflow(long downflow) {
        this.downflow = downflow;
    }

    public long getSumflow() {
        return sumflow;
    }

    public void setSumflow(long sumflow) {
        this.sumflow = sumflow;
    }
```

```
    @Override
    public String toString() {
        return upflow + "\t" + downflow + "\t" + sumflow;
    }
    //序列化，将对象的字段信息写入输出流
    @Override
    public void write(DataOutput out) throws IOException {
        out.writeLong(upflow);
        out.writeLong(downflow);
        out.writeLong(sumflow);
    }
    //反序列化，从输入流读取各字段的信息
    @Override
    public void readFields(DataInput in) throws IOException {
        upflow=in.readLong();
        downflow=in.readLong();
        sumflow=in.readLong();
    }
}
```

FlowCount的实现

```
public class flowCount {
    static class MyMapper extends Mapper<LongWritable,Text,Text,flowBean>{
        @Override
        protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
            String line = value.toString();

            String[] fields = line.split("\t");

            String phoneNum = fields[1];
            long upFlow = Long.parseLong(fields[fields.length - 3]);
            long downFlow = Long.parseLong(fields[fields.length - 2]);

            flowBean bean = new flowBean(upFlow,downFlow);

            context.write(new Text(phoneNum),bean);
        }
    }

    static class MyReducer extends Reducer<Text,flowBean,Text,flowBean>{
        @Override
        protected void reduce(Text key, Iterable<flowBean> values, Context
context) throws IOException, InterruptedException {
            Iterator<flowBean> it = values.iterator();

            long upflow=0;
            long downflow=0;

            while (it.hasNext()){
                flowBean bean = it.next();
                upflow+=bean.getUpflow();
                downflow+=bean.getDownflow();
            }

            flowBean total = new flowBean(upflow, downflow);
```

```
            context.write(key,total);
        }
    }

    public static void main(String[] args) throws InterruptedException,
IOException, ClassNotFoundException {
        //1、配置连接hadoop集群的参数
        Configuration conf = new Configuration();
//      conf.set("fs.defaultFS","hdfs://qianfeng");
        conf.set("fs.defaultFS","file:///");
        conf.set("mapreduce.framework.name","local");
        //2、获取job对象实例
        Job job = Job.getInstance(conf,"FLOWCOUNT");
        //3、指定本业务job的路径
        job.setJarByClass(flowCount.class);

        //4、指定本业务job要使用的Mapper类
        job.setMapperClass(MyMapper.class);

        //5、指定mapper类的输出数据的kv的类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(flowBean.class);

        //6、指定本业务job要使用的Reducer类
        job.setReducerClass(MyReducer.class);

        //7、设置程序的最终输出结果的kv的类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(flowBean.class);

        //8、设置job要处理的数据的输入源
        FileInputFormat.setInputPaths(job,new Path("/data/flowinput"));

        //判断输出目录是否存在，如果存在，则删除之
        //9、设置job的输出目录
        FileOutputFormat.setOutputPath(job,new Path("/out/flowoutput"));
        //10、提交job
        boolean b = job.waitForCompletion(true);
        System.exit(b?0:1);
    }
}
```