MapRduce第一天

课前提示

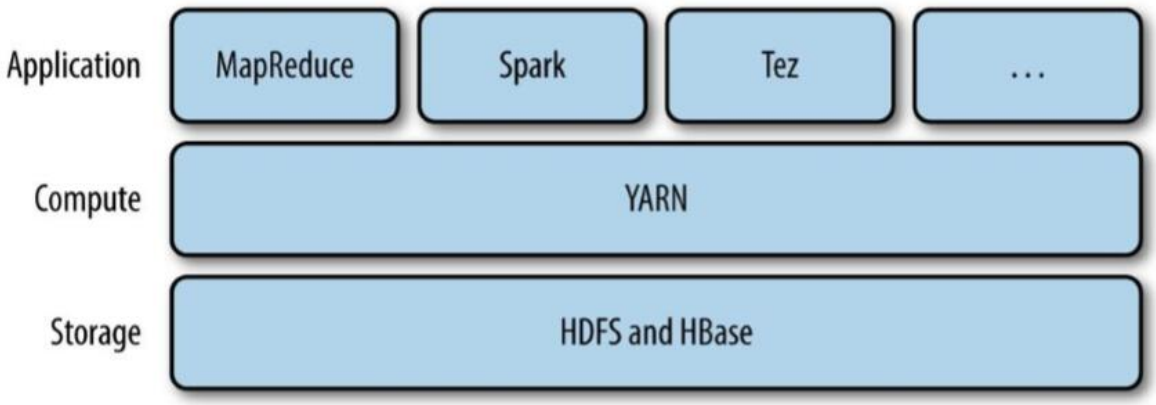
> ## 今日任务
>
> 1. yarn的简介和运行机制
> 2. yarn的配置和启动
> 3. MR的概念
> 4. MR原型解析
> 5. MR的核心思想
> 6. MR的入门案例
>
> ## 今日目标
>
> 1. 了解部分
>   - yarn的简介
>   - MR的概念
> 2. **重点部分**
>
> - yarn的运行机制
> - yarn的属性配置
> - MR的核心思想
>   - wordcount案例的编写

正课内容

第一节：YARN的相关内容

1. yarn简介

1. 概述
   Apache YARN（Yet another Resource Negotiator的缩写）是Hadoop的集群资源管理系统。yarn被引入Hadoop 2,最初是为了改善MapReduce的实现，但是因为具有足够的通用性，同样可以支持其他的分布式计算模式



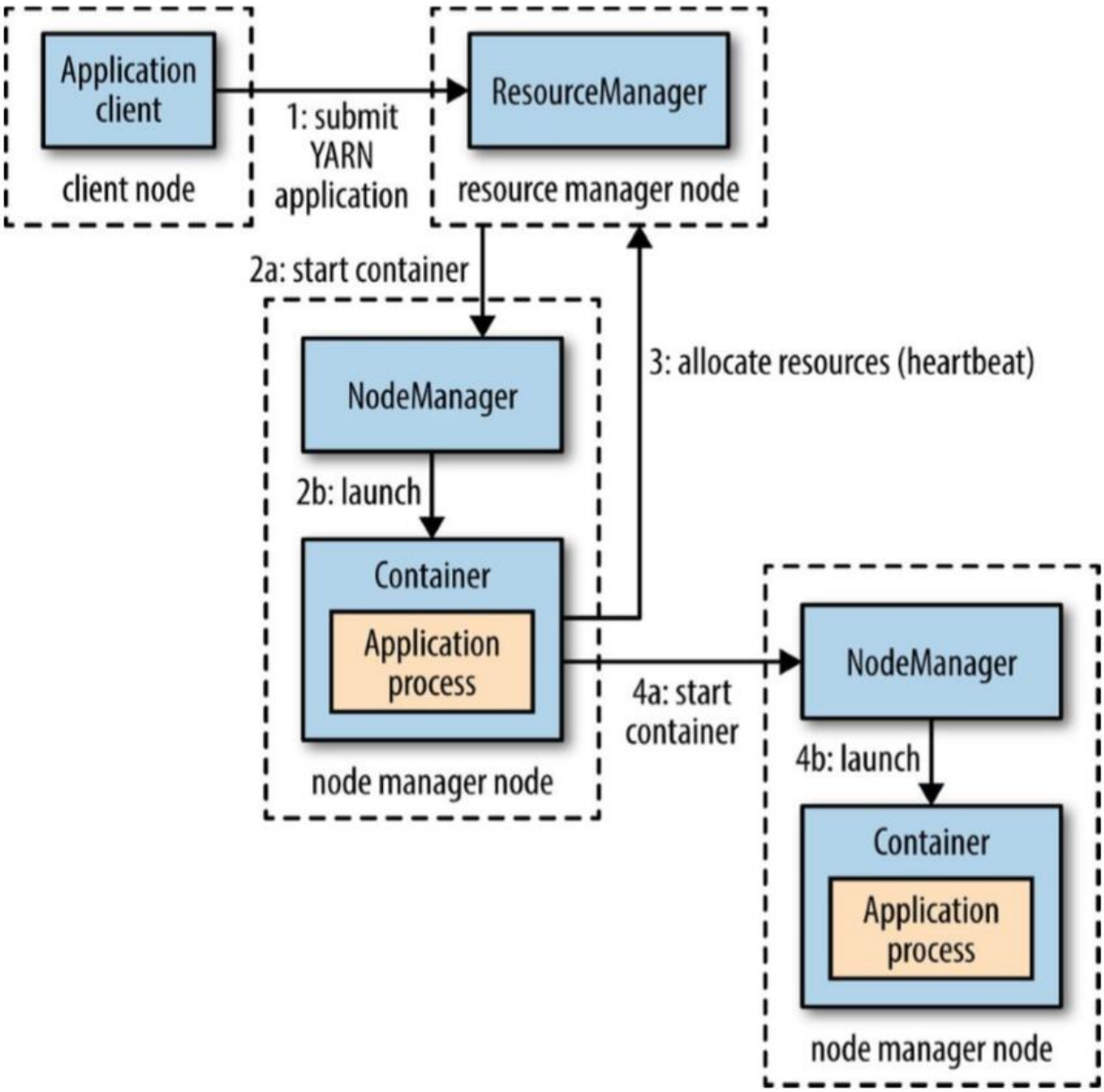
2. 思想

yarn的基本思想是将资源管理和作业调度/监视功能划分为单独的守护进程。其思想是拥有一个全局ResourceManager (RM)和每个应用程序的ApplicationMaster (AM)。应用程序可以是单个作业，也可以是一组作业

ResourceManager和NodeManager构成数据计算框架。ResourceManager是在系统中的所有应用程序之间仲裁资源的最终权威。NodeManager是每台机器的框架代理，负责监视容器的资源使用情况(cpu、内存、磁盘、网络)，并向ResourceManager/Scheduler报告相同的情况

每个应用程序ApplicationMaster实际上是一个特定于框架的库，它的任务是与ResourceManager协商资源，并与NodeManager一起执行和监视任务

2. yarn的运行机制



1. 角色
    1. resource manager 资源管理器：YARN的守护进程，管理集群资源使用
    2. node manager 节点管理器：YARN的守护进程，启动和监控容器，运行在集群中所有节点上
    3. container 容器：执行job进程，容器有资源限制(内存，CPU等)，具体容器取决于YARN配置
2. 运行机制

    yarn通过两类长期运行的守护进程提供自己的核心服务：管理集群上资源使用的资源管理器（ResourceManager）,运行再集群中所有节点上且能够启动和监控容器（container）的节点管理器（node manager）。容器用于执行特定应用程序的进程，每个容器都有资源限制（内存，cpu等）。一个容器可以适应unix进程，也可以是一个linux cgroup,取决于yarn的配置。

    上图描述了yarn是如何运行一个应用的

    1. 首先，客户端联系resoucemanager,要求他运行一个application master进程。
    2. 然后，resourcemanager找到一个能够在容器中启动application master的节点管理器（步骤2a和2b）.
    3. application master通过心跳机制向resourcemanager请求更多的容器资源
    4. application master运行起来之后需要做什么依赖于客户端传递的应用

- 简单地运算后直接返回结果给客户端
- 请求更多容器进行分布式计算

3. yarn配置与启动

- yarn属于hadoop的一个组件，不需要再单独安装程序，hadoop中已经存在配置文件的设置
- yarn也是一个集群，有主节点和从节点。
- 在mapred-site.xml中的配置如下

  mapreduce.framework.name yarn
- 在yarn-site.xml中的配置如下 yarn.resourcemanager.hostname master

```
<!--NodeManager上运行的附属服务-->
<property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
</property>

<!--配置resourcemanager的scheduler的内部通讯地址-->
<property>
<name>yarn.resourcemanager.scheduler.address</name>
<value>master:8030</value>
</property>

<!--配置resoucemanager的资源调度的内部通讯地址-->
<property>
<name>yarn.resourcemanager.resource-tracker.address</name>
<value>master:8031</value>
</property>

<!--配置resourcemanager的内部通讯地址-->
<property>
<name>yarn.resourcemanager.address</name>
<value>master:8032</value>
</property>

<!--配置resourcemanager的管理员的内部通讯地址-->
<property>
<name>yarn.resourcemanager.admin.address</name>
<value>master:8033</value>
</property>

<!--配置resourcemanager的web ui 的监控页面-->
<property>
<name>yarn.resourcemanager.webapp.address</name>
<value>master:8088</value>
</property>
```

2. **启动**：

yarn-daemon.sh start ResourceManager yarn-daemon.sh start NodeManager start-yarn.sh stop-yarn.sh

第二节: MR的概念

1. MR简介

Hadoop MapReduce是对google提出的分布式并行编程模型MapReduce论文的开源实现，以可靠，容错的方式运行在分布式文件系统HDFS上的并行处理数据的编程模型。MapReduce的优势在于处理大规模数据集（1TB以上）。

> - 在过去的很长一段时间里，CPU的性能都会遵循”摩尔定律“，在性能上每隔18个月左右就是提高一倍。那个时候，不需要对程序做任何改变，仅仅通过使用更优秀的CPU，就可以进行性能提升。但是现在，在CPU性能提升的道路上，人类已经到达了制作工艺的瓶颈，因此，我们不能再把希望寄托在性能更高的CPU身上了。
> - 现在这个时候，大规模数据存储在分布式文件系统上，人们也开始采用分布式并行编程来提高程序的性能。分布式程序运行在大规模计算机集群上，集群是大量的廉价服务器，可以并行执行大规模数据处理任务，这样就获得了海量的计算能力
> - 分布式并行编程比传统的程序有明显的区别，它运行在大量计算机构成的集群上，可以充分利用集群的并行处理能力；同时，通过向集群中增加新的计算节点，就可以很容易的实现集群计算能力的扩展。

MapReduce 作业通常将输入数据集拆分为独立的块，这些块由map任务以完全并行的方式处理。框架对map的输出进行排序，然后输入到reduce任务。通常，作业的输入和输出都存储在文件系统中。该框架负责调度任务，监视任务并重新执行失败的任务。

通常，计算节点和存储节点是相同的，即MapReduce框架和Hadoop分布式文件系统在同一组节点上运行。此配置允许框架有效地在已存在数据的节点上调度任务，从而在集群中产生非常高的聚合带宽。

MapReduce框架由单个主ResourceManager，每个集群节点一个从NodeManager和每个应用程序的MRAppMaster组成（参见YARN简介）。

最低限度，应用程序通过适当的接口和/或抽象类的实现来指定输入/输出位置并提供映射和减少功能。这些和其他作业参数包括作业配置。

然后，Hadoop 作业客户端将作业（jar /可执行文件等）和配置提交给ResourceManager，然后ResourceManager负责将软件/配置分发给从站，调度任务并监视它们，为作业提供状态和诊断信息 -客户。

虽然Hadoop框架是用Java™实现的，但MapReduce应用程序不需要用Java编写。

2. 原型解析

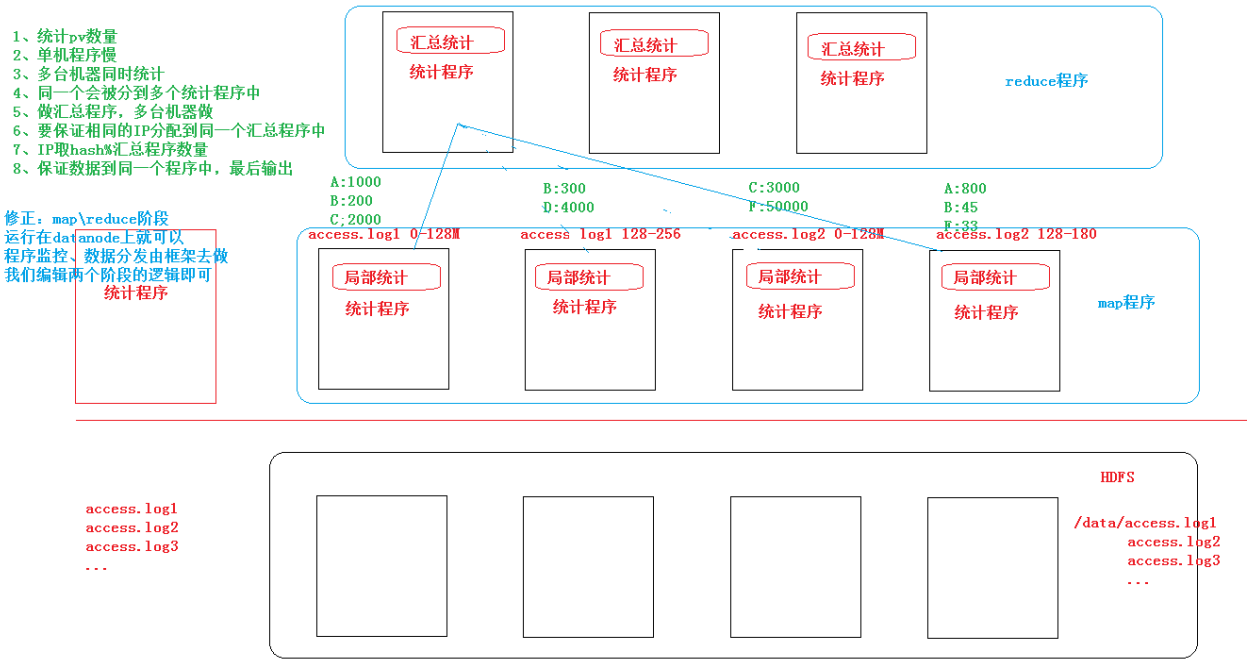
Hadoop的MapReduce核心技术起源于谷歌在2004年发表的关于MapReduce系统的论文介绍。论文中有这么一句话：Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages。这句话提到了MapReduce思想来源，大致意思是，MapReduce的灵感来源于函数式语言（比如Lisp）中的内置函数map（映射）和reduce（规约）。

简单来说，在函数式语言里，map表示对一个列表（List）中的每个元素做计算，reduce表示对一个列表中的每个元素做迭代计算。它们具体的计算是通过传入的函数来实现的，map和reduce提供的是计算的框架。我们想一下，reduce既然能做迭代计算，那就表示列表中的元素是相关的（比如我想对列表中的所有元素做相加求和，那么列表中至少都应该是数值吧）。而map是对列表中每个元素做单独处理的，这表示列表中可以是杂乱无章的数据。这样看来，就有点联系了。在MapReduce里，Map处理的是原始数据，自然是杂乱无章的，每条数据之间互相没有关系；到了Reduce阶段，数据是以key后面跟着若干个value来组织的，这些value有相关性，至少它们都在一个key下面，于是就符合函数式语言里map和reduce的基本思想了。

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，其类型由程序员来选择。程序员需要为每个阶段写一个函数，分别是map函数和reduce函数。

在执行MapReduce任务时，一个大规模的数据集会被划分成许多独立的等长的小数据块，称为输入分片（input split）或简称"分片"。Hadoop为每个输入分片分别构建一个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。map任务处理后的结果会继续作为reduce任务的输入，最终由reduce任务输出最后结果，并写入分布式文件系统。

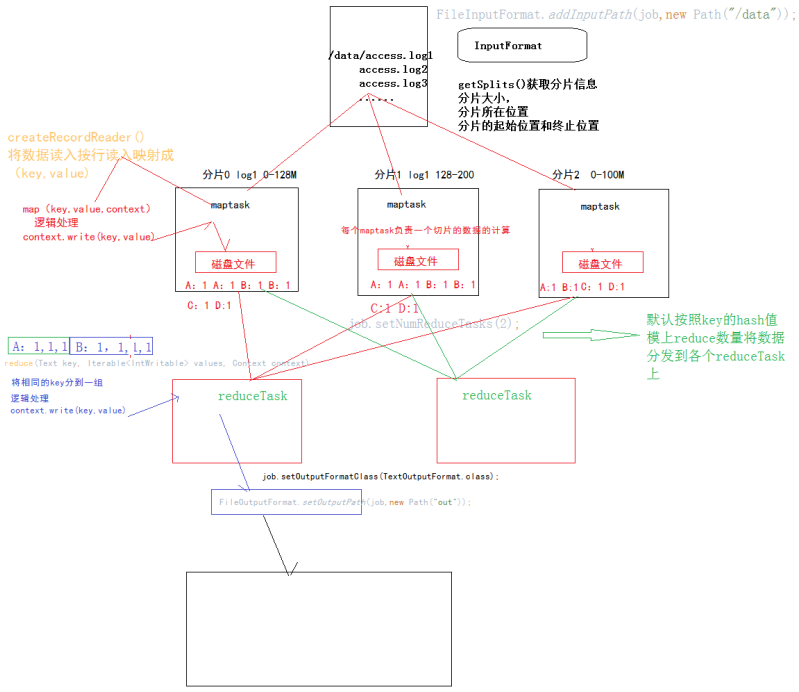
1. 案例分析



3. 核心思想

分而治之、移动计算不移动数据 MapReduce设计的一个理念是“计算向数据靠拢”（移动计算），而不是“数据向计算靠拢”（移动数据）。因为移动数据需要大量的网络传输开销，尤其是在大规模数据环境下，这种开销尤为惊人，所以移动计算要比移动数据更加经济。所以，在一个集群中，只要有可能，MapReduce框架就会将Map程序就近的在HDFS数据所在的节点上运行，即将计算节点和存储节点放在一起运行，从而减少节点间的数据移动开销。正因为这样，MapReduce可以并行的进行处理，解决计算效率问题。

第三节：wordcount案例



HDFS文件系统，多个文件/data/log1 log2 log3

程序提交时会先分片，分片信息包括文件路劲，分片起始位置，分片大小，分片数据所在块的信息，块的主机信息

maptask多个，每个maptask处理一个分片的数据

一行数据调用createRecorderReader方法返回键值对（由换行符决定）

一个（K,V）对调用一次map(k,v，context)方法，然后将数据交由maptask写出到磁盘

数据分发，默认的是key的hash值%redueceNum

reduce数量设置job.setNumReducetasks(2)

reduce保存数据到本地磁盘

分组读取数据（key相同为一组）

一组数据调用一次reduce（k,values迭代器，context）方法

1. MyMapper类型的编写

`public class MyMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

```
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    /**
     *    针对每一行数据，都会执行一次下面的map方法
     * @param key
     * @param value
     * @param context
     * @throws IOException
     * @throws InterruptedException
     */
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
//          String row = value.toString();
//          String[] words = row.split(" ");
//          for(String word:words) {
//              //上下文的write(k2 ,v2)
//              Text t = new Text();
//              t.set(word);
//              context.write(t, new IntWritable(1));
//          }
        // 参数key 和value,是<k1,v1> 作为map方法的输入数据
        StringTokenizer itr = new StringTokenizer(value.toString(), " ");
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, one);// 以<k2, v2>的形式发送出去：输出数据
        }
    }

```

```
28 }`
```

2. MyReducer类型的编写

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

```
1         /**
2          * reduce函数：输入数据 是map端的输出 <k2,v2>
3          */
4         @Override
5         protected void reduce(Text key, Iterable<IntWritable> values, Context
  context)
6                 throws IOException, InterruptedException {
7             // v2接受的是map端的输出的排序与合并的一个列表(迭代器)
8             // 将v2里的元素遍历出来，进行累加，并以k3,v3的形式写出去
9             int sum = 0;
10            for (IntWritable val : values) {
11                sum += val.get();
12            }
13            result.set(sum);
14            context.write(key, result);// 将最終統計的結果輸出到文件系統上
15        }
16    }`
```

3. MyDriver类型的编写

   `public class MyDriver { public static void main(String[] args) throws Exception, IOException { // 获取配置对象 Configuration conf = new Configuration(); //创建job实例，同时设置job名称 Job job = Job.getInstance(conf, "word count"); //用于指定驱动类型 job.setJarByClass(MyDriver.class); //用于指定Map阶段的类型 job.setMapperClass(MyMapper.class); //用于指定Reduce端的类型 job.setReducerClass(MyReducer.class); //设置k3的输出类型 job.setOutputKeyClass(Text.class); //设置v3的输出类型 job.setOutputValueClass(IntWritable.class); //设置要统计的文件的输入路径, FileInputFormat.addInputPath(job, new Path(args[0])); //设置结果的输出路径 FileOutputFormat.setOutputPath(job, new Path(args[1])); //等待作业成功执行，并退出程序 System.exit(job.waitForCompletion(true) ? 0 : 1); } }`

4. 本地运行(略)

   注意：本地运行时，不使用分布式文件系统。输入输出路径要明确

5. 集群运行

   1. 注意：集群运行时，需要修改驱动类中的两个路径 分别为：args[0]，args[1]
   2. 打包：右键-->export-->java-->jar file-->选中要导出的项目，定义存储位置和包名。 -->next-->next-->选择main class--ok
   3. 将xxxx.jar包上传到linux虚拟机中
   4. 确定要统计的文件是否存储HDFS中。 比如/word.txt
   5. 运行

方法1：不指定main Class的方式：hadoop jar xxxxx.jar /word.txt /out10000 方法2：如果导出jar包时，没有指定mainclass hadoop jar wordcount111.jar com.qianfeng.wordcount.MyDriver /word.txt /out10000

MapReduce:second day

课前提示

> ## 今日任务
>
> 1. MR的分片机制
> 2. mapTask执行流程详解
> 3. reduceTask执行流程详解
> 4. shuffle流程解析
> 5. mr的本地运行
> 6. 自定义组件combiner
> 7. 序列化类型以及自定义类型
> 8. 案例：手机流量的汇总
>
> ## 今日目标
>
> 1. 了解部分
>   - MR的本地运行
>   - 熟悉ReduceTask的执行流程
>   - 熟悉combiner的自定义
>   - 熟悉Hadoop的序列化机制和自定义类型
> 2. **重点部分**
>
> - 分片机制
> - MapTask的执行流程
> - Shuffle的执行流程
> - 手机流量的分析需求

正课内容

第一节：MR的分片机制

1. 分片简介

   Hadoop将MapReduce的输入数据划分成等长的小数据块，称之为输入分片（inputSplit）或者简称“分片”，Hadoop为每一个分片构建一个单独的map任务，并由该任务来运行用户自定义的map方法，从而处理分片中的每一条记录

2. 分片大小的选择

1. 拥有许多分片，意味着处理每个分片所需要的时间要小于处理整个输入数据所花的时间(分而治之的优势)。
2. 并行处理分片，且每个分片比较小。负载平衡，好的计算机处理的更快，可以腾出时间，做别的任务
3. 如果分片太小，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。
4. 如果分片跨越两个数据块，那么分片的部分数据需要通过网络传输到map任务运行的节点，占用网络带宽，效率更低
5. 因此最佳分片大小应该和HDFS上的块大小一致。hadoop2.x默认128M.

3. 分片类型源码：FileSplit类型 `public class FileSplit extends InputSplit implements Writable { private Path file; //要处理的文件名 private long start; //当前逻辑分片的偏移量 private long length; //当前逻辑分片的字节长度 private String[] hosts; //当前逻辑分片对应的块数据所在的主机名 private SplitLocationInfo[] hostInfos;

```
  public FileSplit() {}
  public FileSplit(Path file, long start, long length, String[] hosts) {
    this.file = file;        //创建逻辑分片对象时调用的构造器
    this.start = start;
    this.length = length;
    this.hosts = hosts;
  }
  .....
}`
```

4. 文件输入格式类型的源码：FileInputFormat

`public abstract class FileInputFormat<K, V> implements InputFormat<K, V> { public static final String NUM_INPUT_FILES; public static final String INPUT_DIR_RECURSIVE; private static final double SPLIT_SLOP = 1.1; private long minSplitSize = 1; ..... protected FileSplit makeSplit(Path file, long start, long length, String[] hosts) { return new FileSplit(file, start, length, hosts); } ..... public InputSplit[] getSplits(JobConf job, int numSplits)throws IOException {

```
    //  获取文件的状态信息
    FileStatus[] files = listStatus(job);

    // Save the number of input files for metrics/loadgen
    job.setLong(NUM_INPUT_FILES, files.length);
    long totalSize = 0;                           // compute total size
    for (FileStatus file: files) {                // check we have valid files
     .....
      totalSize += file.getLen();
    }

    long goalSize = totalSize / (numSplits == 0 ? 1 : numSplits);
    long minSize = Math.max(job.getLong(org.apache.hadoop.mapreduce.lib.input.
      FileInputFormat.SPLIT_MINSIZE, 1), minSplitSize);

    // generate splits
    ArrayList<FileSplit> splits = new ArrayList<FileSplit>(numSplits);
    for (FileStatus file: files) {
      Path path = file.getPath();
      long length = file.getLen();
      if (length != 0) {
```

```
        FileSystem fs = path.getFileSystem(job);
        BlockLocation[] blkLocations;
        .......
        if (isSplitable(fs, path)) {
          long blockSize = file.getBlockSize();
          long splitSize = computeSplitSize(goalSize, minSize, blockSize);
          long bytesRemaining = length;
          while (((double) bytesRemaining)/splitSize > SPLIT_SLOP) {
            ......
            splits.add(makeSplit(path, length-bytesRemaining, splitSize,
                splitHosts[0], splitHosts[1]));
            bytesRemaining -= splitSize;
          }

          if (bytesRemaining != 0) {
            String[][] splitHosts = getSplitHostsAndCachedHosts(blkLocations,
length
                - bytesRemaining, bytesRemaining, clusterMap);
            splits.add(makeSplit(path, length - bytesRemaining, bytesRemaining,
                splitHosts[0], splitHosts[1]));
          }
        } else {
          ......
        }
      } else {
        ......
      }
    }
    ......
    return splits.toArray(new FileSplit[splits.size()]);
  }
```

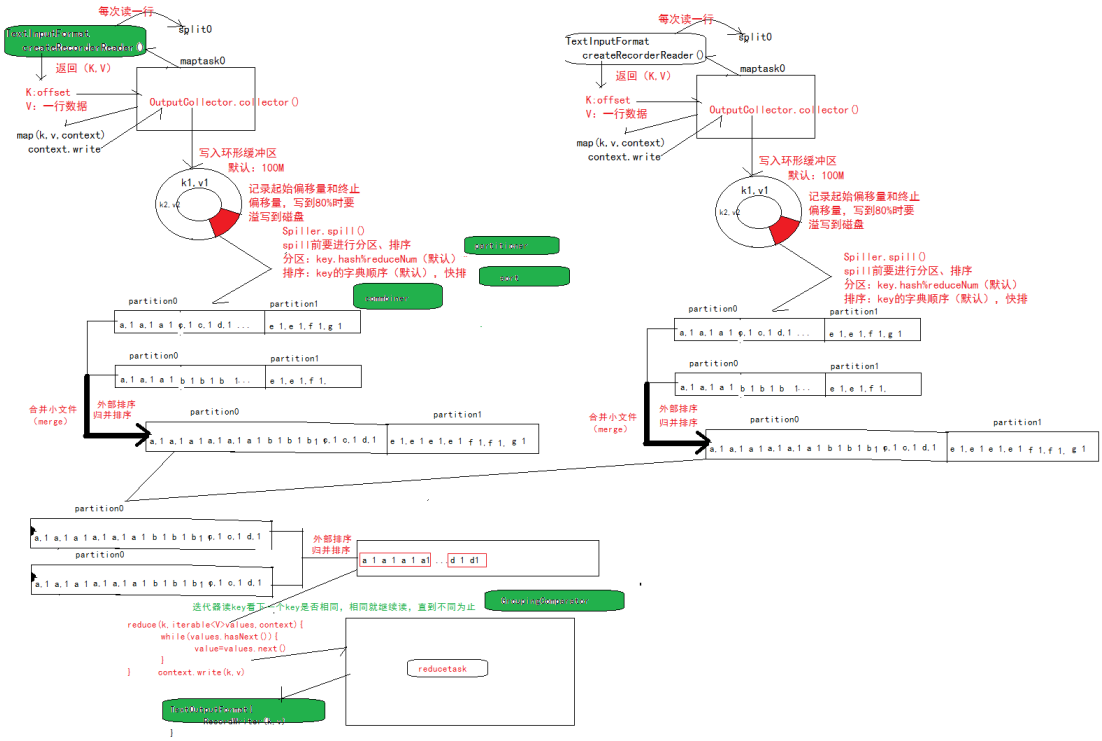
5. 创建分片的过程
    1. 获取文件大小及位置
    2. 判断文件是否可以分片（压缩格式有的可以进行分片，有的不可以）
    3. 获取分片的大小
    4. 剩余文件的大小/分片大小>1.1时，循环执行封装分片信息的方法，具体如下
        1. 封装一个分片信息(包含文件的路径，分片的起始偏移量，要处理的大小，分片包含的块的信息，分片中包含的块存在哪儿些机器上)
        2. 剩余文件的大小/分片大小<=1.1且 不等于0时，封装一个分片信息(包含文件的路径，分片的起始偏移量，要处理的大小，分片包含的块的信息，分片中包含的块存在哪儿些机器上)

    分片的注意事项：1.1倍的冗余。 问题：260M文件分几个片?

```
考虑Hadoop应用处理的数据集比较大，因此需要借助压缩。按照效率从高到低排列的
（1）使用容器格式文件，例如：顺序文件、RCFile、Avro数据格式支持压缩和切分文件。另外在配合使用一些快速压缩工具，例如：LZO、LZ4或者Snappy.
（2）使用支持切分压缩格式，例如bzip2
（3）在应用中将文件切分成块，对每块进行任意格式压缩。这种情况确保压缩后的数据库接近HDFS块大小。
（4）存储未压缩文件，以原始文件存储。
```

6. 读取分片的细节：如果有多个分片
    - 第一个分片读到末尾再多读一行
    - 既不是第一个分片也不是最后一个分片第一行数据舍弃，末尾多读一行
    - 最后一个分片舍弃第一行，末尾多读一行

第一节：MapTask执行流程详解



1. maptask调用FileInputFormat的getRecordReader读取分片数据
2. 每行数据读取一次，返回一个(K,V)对，K是offset,V是一行数据
3. 将k-v对交给MapTask处理
4. 每对k-v调用一次map(K,V，context)方法，然后context.write(k,v)
5. 写出的数据交给收集器OutputCollector.collector()处理
6. 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
7. 默认写到80%的时候要溢写到磁盘，溢写磁盘的过程中数据继续写入剩余20%

8. 溢写磁盘之前要先进行分区然后分区内进行排序
9. 默认的分区规则是hashpatitioner，即key的hash%reduceNum
10. 默认的排序规则是key的字典顺序，使用的是快速排序
11. 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
12. 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）

第二节：ReduceTask执行流程详解

1. 数据按照分区规则发送到reducetask
2. reducetask将来自多个maptask的数据进行合并，排序（外部排序===》归并排序）
3. 按照key相同分组（）
4. 一组数据调用一次reduce(k,iterablevalues,context)
5. 处理后的数据交由reducetask
6. reducetask调用FileOutputFormat组件
7. FileOutputFormat组件中的write方法将数据写出

第三节：shuffle流程

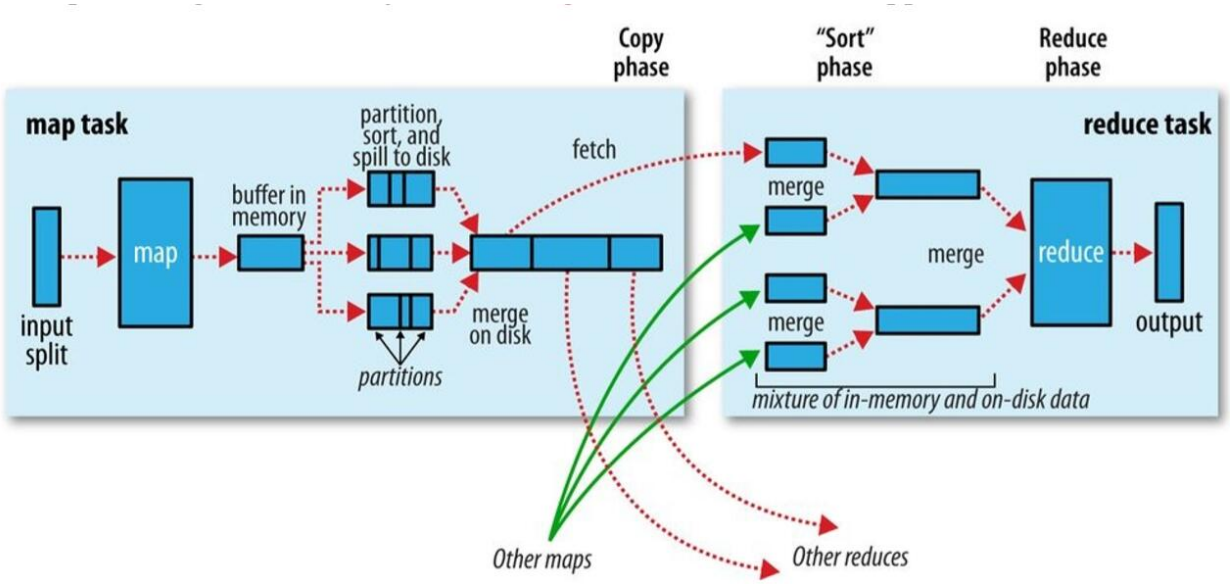


Figure 7-4. Shuffle and sort in MapReduce

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)

1. 从map函数输出到reduce函数接受输入数据，这个过程称之为shuffle.
2. map函数的输出，存储环形缓冲区（默认大小100M,阈值80M）

   环形缓冲区：其实是一个字节数组kvbuffer. 有一个sequator标记，kv原始数据从左向右填充(顺时针)，kvmeta是对kvbuffer的一个封装，封装成了int数组，用于存储kv原始数据的对应的元数据（valstart,keystart,partition,vallen）,从右向左(逆时针)。
3. 当达到阈值时，准备溢写到本地磁盘(因为是中间数据，因此没有必要存储在HDFS上)。在溢写前要进行对元数据分区(partition)整理，然后进行排序(quick sort,通过元数据找到出key，同一分区的所有key进行排序，排序完，元数据就已经有序了，在溢写时，按照元数据的顺序寻找原始数据进行溢写)
4. 如果有必要，可以在排序后，溢写前调用combiner函数进行运算，来达到减少数据的目的
5. 溢写文件有可能产生多个，然后对这多个溢写文件进行再次合并(也要进行分区和排序)。当溢写个数>=3时，可以再次调用combiner函数来减少数据。如果溢写个数<3时，默认不会调用combiner函数。

6. 合并的最终溢写文件可以使用压缩技术来达到节省磁盘空间和减少向reduce阶段传输数据的目的。（存储在本地磁盘中）
7. Reduce阶段通过HTTP写抓取属于自己的分区的所有map的输出数据(默认线程数是5，因此可以并发抓取)。
8. 抓取到的数据存在内存中，如果数据量大，当达到本地内存的阈值时会进行溢写操作，在溢写前会进行合并和排序(排序阶段)，然后写到磁盘中，
9. 溢写文件可能会产生多个，因此在进入reduce之前会再次合并(合并因子是10),最后一次合并要满足10这个因子，同时输入给reduce函数，而不是产生合并文件。reduce函数输出数据会直接存储在HDFS上。

第四节：mr的本地运行

1. 编写mr的步骤
    1. 创建一个项目
    2. 自定义Mapper类型，继承Mapper<K1,V1,K2,V2>，重写map(K1,V1,Context)
    3. 自定义Reducer类型，继承Reducer<K2,V2,K3,V3>,重写reduce(K2,V2,Context)
    4. 自定义驱动类型，编写Job作业，设置其工作内容并提交。

第五节：自定义combiner函数

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：

   job.setCombinerClass(WordCountReduce.class);

第六节：Hadoop的序列化机制

1. 序列化简介
    - 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。
    - 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

MapReduce:third day

课前提示

> ## 今日任务
>
> 1. mr的本地运行
> 2. 自定义组件combiner
> 3. 序列化类型以及自定义类型
> 4. 案例：手机流量的汇总
>
> ## 今日目标
>
> 1. 了解部分
>   - MR的本地运行
>   - combiner函数
>   - 序列化简介
>   - hadoop常用的类型
> 2. **重点部分**
>   - 自定义类型
>   - 手机流量的统计

正课内容

第一节：mr的本地运行

1. 编写MapReduce程序的步骤

- 创建一个项目
- 自定义Mapper类型，继承Mapper<K1,V1,K2,V2>，重写map(K1,V1,Context)
- 自定义Reducer类型，继承Reducer<K2,V2,K3,V3>,重写reduce(K2,V2,Context)
- 自定义驱动类型，编写Job作业，设置其工作内容并提交。

2. 本地运行

小贴士：mr的本地运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bin目录下。重启机器或者是IDE。如果还报错，将org包放入java目录下

情况1：统计本地文件，结果存储到本地

```
```java
public static void main(String[] args) throws Exception{
        //加载default参数
        Configuration conf = new Configuration();
        //创建作业对象
        Job job = Job.getInstance(conf,"max air");
        //设置驱动类
```
```

```
        job.setJarByClass(MyDriver.class);
        //设置Mapper类型
        job.setMapperClass(MyMapper.class);
        job.setReducerClass(MyReducer.class);
        //设置输出数据的类型
        job.setOutputKeyClass(LongWritable.class);
        job.setOutputValueClass(LongWritable.class);
        //设置FileInputFormat要切分的文件
        FileInputFormat.setInputPaths(job, new Path("D:/academia/classes/XSQ-1903\\MR-DO1\\数据及文件\\air"));
        //设置输出路径
        FileOutputFormat.setOutputPath(job, new Path("D:/out445"));
        //提交作业，等待完成
        System.exit(job.waitForCompletion(true)?0:1);

}
```

情况2：读取集群上的文件到本地执行，存储到集群上。如果是普通集群，只需要讲core-site.xml放入src下。如果是HA集群，需要将core-site.xml和hdfs-site.xml方法src下。

```
```java
    public class MyDriver {
        public static void main(String[] args) throws Exception{
            //加载default参数
            Configuration conf = new Configuration();
            //创建作业对象
            Job job = Job.getInstance(conf,"max air");
            //设置驱动类
            job.setJarByClass(MyDriver.class);
            //设置Mapper类型
            job.setMapperClass(MyMapper.class);
            job.setReducerClass(MyReducer.class);
            //设置输出数据的类型
            job.setOutputKeyClass(LongWritable.class);
            job.setOutputValueClass(LongWritable.class);
            //设置FileInputFormat要切分的文件
            FileInputFormat.setInputPaths(job, new Path("/input/air"));
            //设置输出路径
            FileOutputFormat.setOutputPath(job, new Path("/out"));
            //提交作业，等待完成
            System.exit(job.waitForCompletion(true)?0:1);
        }
    }
```
```

第二节：自定义的combiner函数

集群的可用带宽本来就很稀缺，因此在不影响结果数据的前提下，尽可能的减少磁盘IO和网络传输，是非常合适的。Hadoop允许用户针对map任务的输出指定一个combiner函数（其实是一个运行在map端的reduce函数），用于优化MR的执行效率。

特点总结：

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. Reduce阶段的Reducer是每一个接收全局的Map Task 所输出的结果
5. Combiner是在合并排序后运行的。因此map端和reduce端都可以调用此函数。
6. Combiner的存在就是提高当前网络IO传输的性能，是MapReduce的一种优化手段。
7. Combiner的具体实现：

   job.setCombinerClass(MyCombiner.class);
8. combiner不適合做求平均值这类需求，很可能就影响了结果。

第三节：Hadoop的序列化机制

1. 序列化简介

- 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。
- 序列化的两个领域：永久存储和进程间通信。

2. 不选择使用java的序列化机制的原因

hadoop涉及到大量的数据的传输（IO），并且网络带宽稀缺，因此使用序列化机制迫不及待，而对数据序列化的要求有四个(紧凑，快速，可扩展，支持互操作)，而java语言自带的序列化机制并不符合这些理想要求，Java类中提供的序列化机制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，因此专为Hadoop单独设计了一套序列化机制：Writable。

MapReduce的key和value,都必须是可序列化的。而针对于key而言，是数据排序的关键字，因此还需要提供比较接口：WritableComparable

3. 常用的序列化类型

Java primitive	Writable implementation	Serialized size (bytes)
boolean	BooleanWritable	1
byte	ByteWritable	1
short	ShortWritable	2
int	IntWritable	4
	VIntWritable	1–5
float	FloatWritable	4
long	LongWritable	8
	VLongWritable	1–9
double	DoubleWritable	8

4. 自定义类型

定义一个表示一对字符串的类型

```java
```java
    public class TextPair implements WritableComparable<TextPair> {
        private Text first;
        private Text second;
        public TextPair() {
            set(new Text(), new Text());
        }
        public TextPair(String first, String second) {
            set(new Text(first), new Text(second));
        }
        public TextPair(Text first, Text second) {
            set(first, second);
        }
        public void set(Text first, Text second) {
            this.first = first;
            this.second = second;
        }
        public Text getFirst() {
            return first;
        }
        public void setFirst(Text first) {
            this.first = first;
        }
        public Text getSecond() {
```
```

```
            return second;
        }
        public void setSecond(Text second) {
            this.second = second;
        }
        @Override
        public void write(DataOutput out) throws IOException {
            first.write(out);
            second.write(out);
        }
        @Override
        public void readFields(DataInput in) throws IOException {
            first.readFields(in);
            second.readFields(in);
        }
        @Override
        public int compareTo(TextPair o) {
            int cmp = first.compareTo(o.first);
            if (cmp != 0) {
                return cmp;
            }
            return second.compareTo(o.second);
        }
    }
```

5. 案例：手机流量的统计

1. **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

2. **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,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

   - <k1,v1>的分析：取一行TextInputFormat类去读取，offset做key,一行数据做value，

     offset:phoneNum,upflow,downflow

   - 拆分，取出倒数第二倒数第三段，map端读取数据然后输出

   ```java
   public class FlowBean implements Writable{
    private int upFlow;
    private int downFlow;
    private int totalFlow;

    //提供构造器
```

```
    public FlowBean() {}
    public FlowBean(int upFlow, int downFlow, int totalFlow) {
        super();
        this.upFlow = upFlow;
        this.downFlow = downFlow;
        this.totalFlow = totalFlow;
    }


    public int getUpFlow() {
        return upFlow;
    }
    public void setUpFlow(int upFlow) {
        this.upFlow = upFlow;
    }
    public int getDownFlow() {
        return downFlow;
    }
    public void setDownFlow(int downFlow) {
        this.downFlow = downFlow;
    }
    public int getTotalFlow() {
        return totalFlow;
    }
    public void setTotalFlow(int totalFlow) {
        this.totalFlow = totalFlow;
    }




    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + downFlow;
        result = prime * result + totalFlow;
        result = prime * result + upFlow;
        return result;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        FlowBean other = (FlowBean) obj;
        if (downFlow != other.downFlow)
            return false;
        if (totalFlow != other.totalFlow)
            return false;
```

```
        if (upFlow != other.upFlow)
            return false;
        return true;
    }
    @Override
    public String toString() {
        return upFlow+"," + downFlow + "," + totalFlow;
    }

    /**
     * 重写序列化:
     * 把相应的属性值按照顺序序列化
     */
    @Override
    public void write(DataOutput out) throws IOException {
        //调用输出流的写出方法，即可序列化
        out.writeInt(upFlow);
        out.writeInt(downFlow);
        out.writeInt(totalFlow);
    }
    /**
     * 重写反序列化：按照序列化的顺序反序列化读取
     */
    @Override
    public void readFields(DataInput in) throws IOException {
        upFlow = in.readInt();
        downFlow = in.readInt();
        totalFlow = in.readInt();

    }
}
```

```
   ```public class FlowMapper extends Mapper<LongWritable, Text, Text, FlowBean> {
    // 提供一个FlowBean的实例。用于map函数中，进行属性赋值
    private FlowBean bean = new FlowBean();
    private Text k2 = new Text();

    @Override
    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
        // 解析行记录的v1,封装成FlowBean对象，作为v2
        String[] arr = value.toString().split("\t");
        bean.setUpFlow(Integer.parseInt(arr[1]));
        bean.setDownFlow(Integer.parseInt(arr[2]));
        bean.setTotalFlow(Integer.parseInt(arr[1]) + Integer.parseInt(arr[2]));
        // 将手机号和FlowBean对象作为k2,v2写出去
        k2.set(arr[0]);
        context.write(k2, bean);
    }

}
   ```

```

```java
      public class FlowReducer extends Reducer<Text, FlowBean, Text, FlowBean> {
    private FlowBean bean;

    @Override
    protected void reduce(Text key, Iterable<FlowBean> values, Context context)
            throws IOException, InterruptedException {
        // 遍历同一个key的所有FlowBean,将其属性对应累加，再次封装
        int totalUp = 0;
        int totalDown = 0;
        int total = 0;
        for (FlowBean b : values) {
            totalUp += b.getUpFlow();
            totalDown += b.getDownFlow();
            total += b.getTotalFlow();
        }
        bean = new FlowBean(totalUp, totalDown, total);
        // 写出去
        context.write(key, bean);
    }
}
```

```java
  public class FlowDriver {
    public static void main(String[] args) throws Exception{
        //加载default参数
        Configuration conf = new Configuration();
        //创建作业对象
        Job job = Job.getInstance(conf,"flow2");
        //设置驱动类
        job.setJarByClass(FlowDriver.class);
        //设置Mapper类型
        job.setMapperClass(FlowMapper.class);
        job.setReducerClass(FlowReducer.class);
        //设置输出数据的类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FlowBean.class);
        //设置FileInputFormat要切分的文件
        FileInputFormat.setInputPaths(job, new Path("D:\\academia\\classes\\XSQ-1903\\MR-DO3\\data\\flow"));
        //设置输出路径
        FileOutputFormat.setOutputPath(job, new Path("D:/flow2"));
        //提交作业，等待完成
        System.exit(job.waitForCompletion(true)?0:1);

    }
}
```

第四节：自定义partitioner

1. partitioner解析：

- partitioner的作用是将mapper 输出的key/value划分成不同的partition,每个reducer对应一个partition。
- 默认情况下，partitioner先计算key的散列值（hash值）。然后通过reducer个数执行取模运算：

  key.hashCode%(reducer个数)。这样能够随机地将整个key空间平均分发给每个reducer,同时也能确保不同mapper产生的相同key能被分发到同一个reducer。
- 目的： 可以使用自定义Partitioner来达到reducer的负载均衡， 提高效率。
- 适用范围： 需要非常注意的是：必须提前知道有多少个分区。比如自定义Partitioner会返回4个不同int值，而reducer number设置了小于4，那就会报错。所以我们可以通过运行分析任务来确定分区数。例如，有一堆包含时间戳的数据，但是不知道它能追朔到的时间范围，此时可以运行一个作业来计算出时间范围。
- 注意： 在自定义partitioner时一定要注意防止数据倾斜。

2. 案例：手机归属地的统计

分析:

- 手机号码属于各个省，如何将各个省的上网数据写到一个文件中呢?
- 手机号码属于哪儿个省市可查的，会有数据字典。
- 那么有了这种数据字典如何实现呢?
- 我们要解决的问题就是如何把相同省份的手机号码发送到同一个reduce中去处理
- 这种就是分区组件的实现，默认的分区规则是key的hash值%reduce的数量，那这种规则肯定实现不了我们的需求，那这种组件提供了一个抽象类，我们可以通过继承这个类重写分区方法来改变数据的分发规则，即省份相同的数据发送到同一个reduce中。这个类就是Partitioner

代码：

```java
/**
     *
     * @author Michael
     *
     * 定义分区器时，因为分区逻辑在map输出后，进入环形缓冲区之前,因此泛型是k2,v2的类型
     */
    public class ProvincePartitioner extends Partitioner<Text,FlowBean>{
        //如何调用? 每一个键值对就调用一次，每次都查询去数据库中查询数据字典太耗费资源
        //所以我们模拟数据字典加载到内存
        //我们可以把数据字典的数据加载到内存，便于数据的分发
        //模拟数据字典（查询手机号属于哪儿个省份），定义一个map用于数据字典数据的存储
        private static HashMap<String,Integer> pmap = new HashMap<>();
        //静态初始化
        static{
            //假设136是0号省份，被分发到0号reduce处理
            pmap.put("136",0);
            pmap.put("137",1);
            pmap.put("138",2);
            pmap.put("139",3);
        }
        /**
         * 分区的个数由reduceNum决定。默认使用key.hashCode&Integer.MAX_VALUE%numPartitions
         * 需要重写getParition方法。
```

```
         * 注意：分区号必须从0开始。是连续的自然数
         */
         @Override
        public int getPartition(Text key, FlowBean bean, int numPartitions) {
            String prefix = key.toString().substring(0,3);
            Integer partNum = pmap.get(prefix);
            //有可能有不属于这些分区的的数据，那就给一个默认分区
            return (partNum==null ? 4:partNum);
        }
    }
```

修改Driver类

设置job中job.setPartitionerClass(ProvincePatitioner.class)；设置reduce个数

第五节：TopN案例

1. 需求

求每个人进行评分的所有电影中的分数最高的10部电影

2. 代码

1. RateBean

   `public class RateBean implements WritableComparable { private String movie; private String rate; private String timeStamp; private String uid; public String getMovie() { return movie; }

```
    public void setMovie(String movie) {
        this.movie = movie;
    }

    public String getRate() {
        return rate;
    }

    public void setRate(String rate) {
        this.rate = rate;
    }

    public String getTimeStamp() {
        return timeStamp;
    }

    public void setTimeStamp(String timeStamp) {
        this.timeStamp = timeStamp;
    }

    public String getUid() {
        return uid;
    }

```

```
    public void setUid(String uid) {
        this.uid = uid;
    }

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(movie);
        out.writeUTF(rate);
        out.writeUTF(timeStamp);
        out.writeUTF(uid);
    }
    @Override
    public void readFields(DataInput in) throws IOException {
        this.movie = in.readUTF();
        this.rate = in.readUTF();
        this.timeStamp = in.readUTF();
        this.uid = in.readUTF();
    }
    @Override
    public String toString() {
        return movie + "\t" + rate;
    }
    @Override
    public int compareTo(RateBean o) {
        return -this.rate.compareTo(o.rate);
    }

}`
```

2. RateMapper

`public class RateMapper extends Mapper<LongWritable, Text,Text,RateBean> { Text k = new Text();//先不创建 ObjectMapper objectMapper;//先不创建，创建 @Override protected void setup(Context context) throws IOException, InterruptedException { objectMapper = new ObjectMapper();//赋值 }

```
    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        //ObjectMapper objectMapper = new ObjectMapper();//拿到外面，仅创建一次就可以
        RateBean rateBean = objectMapper.readValue(value.toString(),RateBean.class);
        System.out.println(k.toString());
        k.set(rateBean.getUid());
        context.write(k,rateBean);
    }
}`
```

3. RateReducer

`public class RateReduce extends Reducer<Text,RateBean,Text,RateBean> { @Override protected void reduce(Text key, Iterable<RateBean> values, Context context) throws IOException, InterruptedException { //定义一个list,对list排序 List<RateBean> rateBeanList = new ArrayList<RateBean>(); Configuration conf = context.getConfiguration();`

```
System.out.println("--------:"+conf.get("topN")); int topN = conf.getInt("topN", 5); System.out.println("toN default:"+topN); for (RateBean rateBean:values){ RateBean newBean = new RateBean(); newBean.setMovie(rateBean.getMovie()); newBean.setRate(rateBean.getRate()); newBean.setTimeStamp(rateBean.getTimeStamp()); newBean.setUid(rateBean.getUid()); rateBeanList.add(newBean); } Collections.sort(rateBeanList); for (int i=0;i<topN;i++){ context.write(key,rateBeanList.get(i)); } } }
```

4. RateDriver

```
public class RateDriver { public static void main(String[] args) { try { Configuration conf = new Configuration(); conf.set("topN",args[0]); Job job = Job.getInstance(conf,"rate"); job.setMapperClass(RateMapper.class); job.setReducerClass(RateReduce.class); System.out.println("topN:"+conf.get("topN")); job.setMapOutputKeyClass(Text.class); job.setMapOutputValueClass(RateBean.class); job.setOutputKeyClass(Text.class); job.setOutputValueClass(RateBean.class); FileInputFormat.addInputPath(job,new Path("D:\\academia\\classes\\XSQ-1903\\MR-DO1\\data\\topn")); Path out = new Path("D:/OO123"); FileSystem fs = FileSystem.get(conf); if (fs.exists(out)){ fs.delete(out,true); } FileOutputFormat.setOutputPath(job,out); int res = job.waitForCompletion(true)?0:1; System.exit(res); } catch (Exception e) { e.printStackTrace(); } } }
```

MapReduce:third day

课前提示

今日任务

1. TopN案例
2. 二次排序
3. 分组器
4. job提交流程
5. 优化参数

今日目标

1. 了解部分
    - TopN案例的概念
    - 二次排序的概念
    - 优化参数
2. **重点部分**

- Job提交流程

正课内容

第一节：TopN案例

1. 需求

求每个人进行评分的所有电影中的分数最高的10部电影

2. 代码

1. RateBean

   `public class RateBean implements WritableComparable { private String movie; private String rate; private String timeStamp; private String uid; public String getMovie() { return movie; }

```
    public void setMovie(String movie) {
        this.movie = movie;
    }

    public String getRate() {
        return rate;
    }

    public void setRate(String rate) {
        this.rate = rate;
    }

    public String getTimeStamp() {
        return timeStamp;
    }

    public void setTimeStamp(String timeStamp) {
        this.timeStamp = timeStamp;
    }

    public String getUid() {
        return uid;
    }

    public void setUid(String uid) {
        this.uid = uid;
    }

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(movie);
        out.writeUTF(rate);
        out.writeUTF(timeStamp);
        out.writeUTF(uid);
    }
    @Override
    public void readFields(DataInput in) throws IOException {
        this.movie = in.readUTF();
        this.rate = in.readUTF();
```

```
        this.timeStamp = in.readUTF();
        this.uid = in.readUTF();
    }
    @Override
    public String toString() {
        return movie + "\t" + rate;
    }
    @Override
    public int compareTo(RateBean o) {
        return -this.rate.compareTo(o.rate);
    }

}`
```

2. RateMapper

   `public class RateMapper extends Mapper<LongWritable, Text,Text,RateBean> { Text k = new Text();//先不创建 ObjectMapper objectMapper;//先不创建，创建 @Override protected void setup(Context context) throws IOException, InterruptedException { objectMapper = new ObjectMapper();//赋值 }

```
    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        //ObjectMapper objectMapper = new ObjectMapper();//拿到外面，仅创建一次就可以
        RateBean rateBean = objectMapper.readValue(value.toString(),RateBean.class);
        System.out.println(k.toString());
        k.set(rateBean.getUid());
        context.write(k,rateBean);
    }
}`
```

3. RateReducer

   `public class RateReduce extends Reducer<Text,RateBean,Text,RateBean> { @Override protected void reduce(Text key, Iterable<RateBean> values, Context context) throws IOException, InterruptedException { //定义一个list,对list排序 List<RateBean> rateBeanList = new ArrayList<RateBean>(); Configuration conf = context.getConfiguration(); System.out.println("--------:"+conf.get("topN")); int topN = conf.getInt("topN", 5); System.out.println("toN default:"+topN); for (RateBean rateBean:values){ RateBean newBean = new RateBean(); newBean.setMovie(rateBean.getMovie()); newBean.setRate(rateBean.getRate()); newBean.setTimeStamp(rateBean.getTimeStamp()); newBean.setUid(rateBean.getUid()); rateBeanList.add(newBean); } Collections.sort(rateBeanList); for (int i=0;i<topN;i++){ context.write(key,rateBeanList.get(i)); } } }`

4. RateDriver

   `public class RateDriver { public static void main(String[] args) { try { Configuration conf = new Configuration(); conf.set("topN",args[0]); Job job = Job.getInstance(conf,"rate"); job.setMapperClass(RateMapper.class); job.setReducerClass(RateReduce.class); System.out.println("topN:"+conf.get("topN"));`

```
job.setMapOutputKeyClass(Text.class); job.setMapOutputValueClass(RateBean.class); job.setOutputKeyClass(Text.class); job.setOutputValueClass(RateBean.class); FileInputFormat.addInputPath(job,new Path("D:\\academia\\classes\\XSQ-1903\\MR-DO1\\data\\topn")); Path out = new Path("D:/OO123"); FileSystem fs = FileSystem.get(conf); if (fs.exists(out)){ fs.delete(out,true); } FileOutputFormat.setOutputPath(job,out); int res = job.waitForCompletion(true)?0:1; System.exit(res); } catch (Exception e) { e.printStackTrace(); } } }
```

第二节：二次排序

1. 分析

MapReduce框架在记录到达reducer之前都是通过key(键)对记录进行排序，但是key所对应的value并没有排序。甚至在不同的执行轮次中，这些值的排序也不固定，应为他们来自不同的mapTask且这些MapTask在不同轮次中的完成时间各不相同。

而有的时候，我们的需求可能是这样的：数据按照第一列排序显示，如果第一列的数据相同，再按照第二列的数据来排序，这就是二次排序。

2. 实现思路

我们可以构建一个同时包含第一列和第二列数据的组合key类型，然后重写此类型的排序规则，就可以实现我们的需求。

3. 优化

1. 在使用组合键时，有可能会导致应该属于一个分区的数据被送到了不同的分区，这个时候我们可以重写partitioner

   比如每年的温度数据： 1900,35 1900,34

2. 而partitioner只能保证每一个reduce接收到同一年的所有记录，在一个分区内，reducer仍然会通过键进行分组。

   重写分区器后的1900，35 和 1900，34 能保证是一个分区，但是确是在两个组内，因此会调用两次reduce函数。

   此时，我们可以通过重写分组比较器

```
`public class RateGroupingComparator extends WritableComparator {
    public RateGroupingComparator(){
        super(RateBean.class,true);
    }
    @Override
    public int compare(WritableComparable a, WritableComparable b) {
        RateBean bean1 = (RateBean)a;
        RateBean bean2 = (RateBean)b;
        return bean1.getUid().compareTo(bean2.getUid());
    }
}`
```

第三节：job提交流程

整个过程如下图描述。在MR程序运行时，有五个独立的进程：

- YarnRunner:用于提交作业的客户端程序
- ResourceManager:yarn资源管理器，负责协调集群上计算机资源的分配
- NodeManager:yarn节点管理器，负责启动和监视集群中机器上的计算容器（container）
- Application Master:负责协调运行MapReduce作业的任务，他和任务都在容器中运行，这些容器由资源管理器分配并由节点管理器进行管理。
- HDFS:用于共享作业所需文件。

1. 调用waitForCompletion方法每秒轮询作业的进度，内部封装了submit()方法，用于创建JobCommiter实例，并且调用其的submitJobInternal方法。提交成功后，如果有状态改变，就会把进度报告到控制台。错误也会报告到控制台
2. JobCommiter实例会向ResourceManager申请一个新应用ID，用于MapReduce作业ID。这期间JobCommiter也会进行检查输出路径的情况，以及计算输入分片。
3. 如果成功申请到ID,就会将运行作业所需要的资源（包括作业jar文件，配置文件和计算所得的输入分片元数据文件）上传到一个用ID命名的目录下的HDFS上。此时副本个数默认是10.
4. 准备工作已经做好，再通知ResourceManager调用submitApplication方法提交作业。
5. ResourceManager调用submitApplication方法后，会通知Yarn调度器（Scheduler），调度器分配一个容器，在节点管理器的管理下在容器中启动 application master进程。
6. application master的主类是MRAppMaster，其主要作用是初始化任务，并接受来自任务的进度和完成报告。
7. 然后从HDFS上接受资源，主要是split。然后为每一个split创建MapTask以及参数指定的ReduceTask，任务ID在此时分配
8. 然后Application Master会向资源管理器请求容器，首先为MapTask申请容器，然后再为ReduceTask申请容器。（5%）
9. 一旦ResourceManager中的调度器（Scheduler），为Task分配了一个特定节点上的容器，Application Master就会与NodeManager进行通信来启动容器。
10. 运行任务是由YarnChild来执行的，运行任务前，先将资源本地化（jar文件，配置文件，缓存文件）
11. 然后开始运行MapTask或ReduceTask。
12. 当收到最后一个任务已经完成的通知后，application master会把作业状态设置为success。然后Job轮询时，知道成功完成，就会通知客户端，并把统计信息输出到控制台

第四节：调优参数

1. 资源相关参数

   //以下参数是在用户自己的mr应用程序中配置在mapred-site.xml就可以生效

```
(1) mapreduce.map.memory.mb: 一个Map Task可使用的资源上限（单位:MB），默认为1024。如果Map Task实际使用的资源量超过该值，则会被强制杀死。

(2) mapreduce.reduce.memory.mb: 一个Reduce Task可使用的资源上限（单位:MB），默认为1024。如果Reduce Task实际使用的资源量超过该值，则会被强制杀死。

(3) mapreduce.map.cpu.vcores: 每个Map task可使用的最多cpu core数目，默认值: 1

(4) mapreduce.reduce.cpu.vcores: 每个Reduce task可使用的最多cpu core数目，默认值: 1

(5) mapreduce.map.java.opts: Map Task的JVM参数，你可以在此配置默认的java heap size等参数，e.g.

```

```
“-Xmx1024m -verbose:gc -Xloggc:/tmp/@taskid@.gc” （@taskid@会被Hadoop框架自动换为相应的taskid），默认值：“”

(6) mapreduce.reduce.java.opts: Reduce Task的JVM参数，你可以在此配置默认的java heap size等参数，e.g.

“-Xmx1024m -verbose:gc -Xloggc:/tmp/@taskid@.gc”，默认值：“”

//shuffle性能优化的关键参数，应在yarn启动之前就配置好

(13) mapreduce.task.io.sort.mb   100         //shuffle的环形缓冲区大小，默认100m

(14) mapreduce.map.sort.spill.percent   0.8    //环形缓冲区溢出的阈值，默认80%

//应该在yarn启动之前就配置在服务器的yarn-site.xml配置文件中才能生效

(7) yarn.scheduler.minimum-allocation-mb      1024   给应用程序container分配的最小内存

(8) yarn.scheduler.maximum-allocation-mb      8192  给应用程序container分配的最大内存

(9) yarn.scheduler.minimum-allocation-vcores    1

(10)yarn.scheduler.maximum-allocation-vcores    32

(11)yarn.nodemanager.resource.memory-mb   8192   每台NodeManager最大可用内存

(12)yarn.nodemanager.resource.cpu-vcores    8    每台NodeManager最大可用cpu核数
```

2. 容错相关参数

(1) mapreduce.map.maxattempts: 每个Map Task最大重试次数，一旦重试参数超过该值，则认为Map Task运行失败，默认值：4。

```
(2) mapreduce.reduce.maxattempts: 每个Reduce Task最大重试次数，一旦重试参数超过该值，则认为Map Task运行失败，默认值：4。

(3) mapreduce.map.failures.maxpercent: 当失败的Map Task失败比例超过该值为，整个作业则失败，默认值为0. 如果你
    的应用程序允许丢弃部分输入数据，则该该值设为一个大于0的值，比如5，表示如果有低于5%的Map Task失败（如果一个Map
    Task重试次数超过mapreduce.map.maxattempts，则认为这个Map Task失败，其对应的输入数据将不会产生任何结果），整个作业扔认为
    成功。

(4) mapreduce.reduce.failures.maxpercent: 当失败的Reduce Task失败比例超过该值为，整个作业则失败，默认值为0.

(5) mapreduce.task.timeout: Task超时时间，经常需要设置的一个参数，该参数表达的意思为：如果一个task在一定时间内没有
    任何进入，即不会读取新的数据，也没有输出数据，则认为该task处于block状态，可能是卡住了，也许永远会卡主，为了防
    止因为用户程序永远block住不退出，则强制设置了一个该超时时间（单位毫秒），默认是300000。如果你的程序对每条输入
```

```
    数据的处理时间过长（比如会访问数据库，通过网络拉取数据等），建议将该参数调大，该参数过小常出现的错误提示
    是“AttemptID:attempt_14267829456721_123456_m_000224_0 Timed out after 300 sec

    sContainer killed by the ApplicationMaster.”。
```