# MapReduce详解02

|  |  |
| --- | --- |
| MapReduce快速入门 | 如何理解map、reduce计算模型 |
| MapReduce程序运行演示 |
| MapReduce编程规范及示例编写 |
| MapReduce程序运行模式及debug方法 |
| MapReduce高级特性 | MapReduce程序的核心机制 |
| MapReduce的序列化框架 |
| MapReduce的排序实现 |
| MapReduce的分区机制及自定义 |
| MapReduce的数据压缩 |
| MapReduce与yarn的结合 |
| MapReduce编程案例 |
|  | MapReduce 参数优化 |

目标：

* 掌握MapReduce分布式运算框架的编程思想
* 掌握MapReduce常用算法的编程套路
* 掌握MapReduce分布式运算框架的运行机制，具备一定自定义开发的能力

# 一、流量统计相关需求

1. 对流量日志中的用户统计总上、下行流量

技术点： 自定义javaBean用来在MapReduce中充当value

注意： javaBean要实现Writable接口，实现两个方法

|  |
| --- |
| //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，从输入流中读取各个字段信息  @Override  public void readFields(DataInput in) throws IOException {  upflow = in.readLong();  downflow = in.readLong();  sumflow = in.readLong();    } |

1. 统计流量且按照流量大小倒序排序

技术点：这种需求，用一个mapreduce -job 不好实现，需要两个mapreduce -job

第一个job负责流量统计，跟上题相同

第二个job读入第一个job的输出，然后做排序

要将flowBean作为map的key输出，这样MapReduce就会自动排序

此时，flowBean要实现接口WritableComparable

要实现其中的compareTo()方法，方法中，我们可以定义倒序比较的逻辑

1. 统计流量且按照手机号的归属地，将结果数据输出到不同的省份文件中

技术点：自定义Partitioner

|  |
| --- |
| *@Override*  *public int getPartition(Text key, FlowBean value, int numPartitions) {*    *String prefix = key.toString().substring(0,3);*  *Integer partNum = pmap.get(prefix);*    *return (partNum==null?4:partNum);*  *}* |

自定义partition后，要根据自定义partitioner的逻辑设置相应数量的reduce task

|  |
| --- |
| job.setNumReduceTasks(5); |

注意：如果reduceTask的数量>= *getPartition的结果数 ，则会多产生几个空的输出文件part-r-000xx*

*如果 1<reduceTask的数量<getPartition的结果数 ，则有一部分分区数据无处安放，会Exception！！！*

*如果* reduceTask的数量=1，则不管mapTask端输出多少个分区文件，最终结果都交给这一个reduceTask，最终也就只会产生一个结果文件 part-r-00000

# 二、社交粉丝数据分析

以下是qq的好友列表数据，冒号前是一个用，冒号后是该用户的所有好友（数据中的好友关系是单向的）

A:B,C,D,F,E,O

B:A,C,E,K

C:F,A,D,I

D:A,E,F,L

E:B,C,D,M,L

F:A,B,C,D,E,O,M

G:A,C,D,E,F

H:A,C,D,E,O

I:A,O

J:B,O

K:A,C,D

L:D,E,F

M:E,F,G

O:A,H,I,J

求出哪些人两两之间有共同好友，及他俩的共同好友都有谁？

解题思路：

|  |
| --- |
| 第一步  map  读一行 A:B,C,D,F,E,O  输出 <B,A><C,A><D,A><F,A><E,A><O,A>  在读一行 B:A,C,E,K  输出 <A,B><C,B><E,B><K,B>  REDUCE  拿到的数据比如<C,A><C,B><C,E><C,F><C,G>......  输出：  <A-B,C>  <A-E,C>  <A-F,C>  <A-G,C>  <B-E,C>  <B-F,C>.....  第二步  map  读入一行<A-B,C>  直接输出<A-B,C>  reduce  读入数据 <A-B,C><A-B,F><A-B,G>.......  输出： A-B C,F,G,..... |

扩展：求互粉的人！！！！

# 三、倒排索引建立

需求：有大量的文本（文档、网页），需要建立搜索索引

# 自定义inputFormat

## 1.1 需求

无论hdfs还是MapReduce，对于小文件都有损效率，实践中，又难免面临处理大量小文件的场景，此时，就需要有相应解决方案

## 1.2 分析

小文件的优化无非以下几种方式：

1. 在数据采集的时候，就将小文件或小批数据合成大文件再上传HDFS
2. 在业务处理之前，在HDFS上使用MapReduce程序对小文件进行合并
3. 在MapReduce处理时，可采用combineInputFormat提高效率

## 1.3 实现

本节实现的是上述第二种方式

程序的核心机制：

自定义一个InputFormat

改写RecordReader，实现一次读取一个完整文件封装为KV

在输出时使用SequenceFileOutPutFormat输出合并文件

代码如下：

自定义InputFromat

|  |
| --- |
| public class WholeFileInputFormat extends FileInputFormat<NullWritable, BytesWritable> {  //设置每个小文件不可分片,保证一个小文件生成一个key-value键值对  @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;  }  } |

自定义RecordReader

|  |
| --- |
| class WholeFileRecordReader extends RecordReader<NullWritable, BytesWritable> {  private FileSplit fileSplit;  private Configuration conf;  private BytesWritable value = new BytesWritable();  private boolean processed = false;  @Override  public void initialize(InputSplit split, TaskAttemptContext context)  throws IOException, InterruptedException {  this.fileSplit = (FileSplit) split;  this.conf = context.getConfiguration();  }  @Override  public boolean nextKeyValue() throws IOException, InterruptedException {  if (!processed) {  byte[] contents = new byte[(int) fileSplit.getLength()];  Path file = fileSplit.getPath();  FileSystem fs = file.getFileSystem(conf);  FSDataInputStream in = null;  try {  in = fs.open(file);  IOUtils.readFully(in, contents, 0, contents.length);  value.set(contents, 0, contents.length);  } finally {  IOUtils.closeStream(in);  }  processed = true;  return true;  }  return false;  }  @Override  public NullWritable getCurrentKey() throws IOException,  InterruptedException {  return NullWritable.get();  }  @Override  public BytesWritable getCurrentValue() throws IOException,  InterruptedException {  return value;  }  @Override  public float getProgress() throws IOException {  return processed ? 1.0f : 0.0f;  }  @Override  public void close() throws IOException {  // do nothing  }  } |

定义MapReduce处理流程

|  |
| --- |
| public class SmallFilesToSequenceFileConverter extends Configured implements Tool {  static class SequenceFileMapper extends Mapper<NullWritable, BytesWritable, Text, BytesWritable> {  private Text filenameKey;  @Override  protected void setup(Context context) throws IOException,  InterruptedException {  InputSplit split = context.getInputSplit();  Path path = ((FileSplit) split).getPath();  filenameKey = new Text(path.toString());  }  @Override  protected void map(NullWritable key, BytesWritable value,  Context context) throws IOException, InterruptedException {  context.write(filenameKey, value);  }  }  @Override  public int run(String[] args) throws Exception {  Configuration conf = new Configuration();  System.setProperty("HADOOP\_USER\_NAME", "hdfs");  String[] otherArgs = new GenericOptionsParser(conf, args)  .getRemainingArgs();  if (otherArgs.length != 2) {  System.err.println("Usage: combinefiles <in> <out>");  System.exit(2);  }    Job job = Job.getInstance(conf,"combine small files to sequencefile");  // 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);    }  } |

# 自定义outputFormat

## 2.1 需求

现有一些原始日志需要做增强解析处理，流程：

1. 从原始日志文件中读取数据
2. 根据日志中的一个URL字段到外部知识库中获取信息增强到原始日志
3. 如果成功增强，则输出到增强结果目录；如果增强失败，则抽取原始数据中URL字段输出到待爬清单目录

## 2.2 分析

程序的关键点是要在一个MapReduce程序中根据数据的不同输出两类结果到不同目录，这类灵活的输出需求可以通过自定义outputformat来实现

## 2.3 实现

实现要点：

1. 在MapReduce中访问外部资源
2. 自定义outputformat，改写其中的recordwriter，改写具体输出数据的方法write()

代码实现如下：

数据库获取数据的工具

|  |
| --- |
| public class DBLoader {  public static void dbLoader(HashMap<String, String> ruleMap) {  Connection conn = null;  Statement st = null;  ResultSet res = null;    try {  Class.forName("com.mysql.jdbc.Driver");  conn = DriverManager.getConnection("jdbc:mysql://hdp-node01:3306/urlknowledge", "root", "root");  st = conn.createStatement();  res = st.executeQuery("select url,content from urlcontent");  while (res.next()) {  ruleMap.put(res.getString(1), res.getString(2));  }  } catch (Exception e) {  e.printStackTrace();    } finally {  try{  if(res!=null){  res.close();  }  if(st!=null){  st.close();  }  if(conn!=null){  conn.close();  }  }catch(Exception e){  e.printStackTrace();  }  }  }      public static void main(String[] args) {  DBLoader db = new DBLoader();  HashMap<String, String> map = new HashMap<String,String>();  db.dbLoader(map);  System.out.println(map.size());  }  } |

自定义一个outputformat

|  |
| --- |
| public class LogEnhancerOutputFormat extends FileOutputFormat<Text, NullWritable>{    @Override  public RecordWriter<Text, NullWritable> getRecordWriter(TaskAttemptContext context) throws IOException, InterruptedException {  FileSystem fs = FileSystem.get(context.getConfiguration());  Path enhancePath = new Path("hdfs://hdp-node01:9000/flow/enhancelog/enhanced.log");  Path toCrawlPath = new Path("hdfs://hdp-node01:9000/flow/tocrawl/tocrawl.log");    FSDataOutputStream enhanceOut = fs.create(enhancePath);  FSDataOutputStream toCrawlOut = fs.create(toCrawlPath);      return new MyRecordWriter(enhanceOut,toCrawlOut);  }        static class MyRecordWriter extends RecordWriter<Text, NullWritable>{    FSDataOutputStream enhanceOut = null;  FSDataOutputStream toCrawlOut = null;    public MyRecordWriter(FSDataOutputStream enhanceOut, FSDataOutputStream toCrawlOut) {  this.enhanceOut = enhanceOut;  this.toCrawlOut = toCrawlOut;  }  @Override  public void write(Text key, NullWritable value) throws IOException, InterruptedException {    //有了数据，你来负责写到目的地 —— hdfs  //判断，进来内容如果是带tocrawl的，就往待爬清单输出流中写 toCrawlOut  if(key.toString().contains("tocrawl")){  toCrawlOut.write(key.toString().getBytes());  }else{  enhanceOut.write(key.toString().getBytes());  }    }  @Override  public void close(TaskAttemptContext context) throws IOException, InterruptedException {    if(toCrawlOut!=null){  toCrawlOut.close();  }  if(enhanceOut!=null){  enhanceOut.close();  }  }  }  } |

开发MapReduce处理流程

|  |
| --- |
| /\*\*  \* 这个程序是对每个小时不断产生的用户上网记录日志进行增强(将日志中的url所指向的网页内容分析结果信息追加到每一行原始日志后面)  \*  \* @author  \*  \*/  public class LogEnhancer {  static class LogEnhancerMapper extends Mapper<LongWritable, Text, Text, NullWritable> {  HashMap<String, String> knowledgeMap = new HashMap<String, String>();  /\*\*  \* maptask在初始化时会先调用setup方法一次 利用这个机制，将外部的知识库加载到maptask执行的机器内存中  \*/  @Override  protected void setup(org.apache.hadoop.mapreduce.Mapper.Context context) throws IOException, InterruptedException {  DBLoader.dbLoader(knowledgeMap);  }  @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  String line = value.toString();  String[] fields = StringUtils.split(line, "\t");  try {  String url = fields[26];  // 对这一行日志中的url去知识库中查找内容分析信息  String content = knowledgeMap.get(url);  // 根据内容信息匹配的结果，来构造两种输出结果  String result = "";  if (null == content) {  // 输往待爬清单的内容  result = url + "\t" + "tocrawl\n";  } else {  // 输往增强日志的内容  result = line + "\t" + content + "\n";  }  context.write(new Text(result), NullWritable.get());  } catch (Exception e) {  }  }  }  public static void main(String[] args) throws Exception {  Configuration conf = new Configuration();  Job job = Job.getInstance(conf);  job.setJarByClass(LogEnhancer.class);  job.setMapperClass(LogEnhancerMapper.class);  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(NullWritable.class);  // 要将自定义的输出格式组件设置到job中  job.setOutputFormatClass(LogEnhancerOutputFormat.class);  FileInputFormat.setInputPaths(job, new Path(args[0]));  // 虽然我们自定义了outputformat，但是因为我们的outputformat继承自fileoutputformat  // 而fileoutputformat要输出一个\_SUCCESS文件，所以，在这还得指定一个输出目录  FileOutputFormat.setOutputPath(job, new Path(args[1]));  job.waitForCompletion(true);  System.exit(0);  }  } |

# 自定义GroupingComparator

## 3.1 需求

有如下订单数据

|  |  |  |
| --- | --- | --- |
| 订单id | 商品id | 成交金额 |
| Order\_0000001 | Pdt\_01 | 222.8 |
| Order\_0000001 | Pdt\_05 | 25.8 |
| Order\_0000002 | Pdt\_03 | 522.8 |
| Order\_0000002 | Pdt\_04 | 122.4 |
| Order\_0000003 | Pdt\_01 | 222.8 |

现在需要求出每一个订单中成交金额最大的一笔交易

## 3.2 分析

1、利用"订单id和成交金额"作为key，可以将map阶段读取到的所有订单数据按照id分区，按照金额排序，发送到reduce

2、在reduce端利用groupingcomparator将订单id相同的kv聚合成组，然后取第一个即是最大值

## 3.3 实现

自定义groupingcomparator

|  |
| --- |
| /\*\*  \* 用于控制shuffle过程中reduce端对kv对的聚合逻辑  \* @author duanhaitao@itcast.cn  \*  \*/  public class ItemidGroupingComparator extends WritableComparator {  protected ItemidGroupingComparator() {  super(OrderBean.class, true);  }  @Override  public int compare(WritableComparable a, WritableComparable b) {  OrderBean abean = (OrderBean) a;  OrderBean bbean = (OrderBean) b;    //将item\_id相同的bean都视为相同，从而聚合为一组  return abean.getItemid().compareTo(bbean.getItemid());  }  } |

定义订单信息bean

|  |
| --- |
| /\*\*  \* 订单信息bean，实现hadoop的序列化机制  \* @author duanhaitao@itcast.cn  \*  \*/  public class OrderBean implements WritableComparable<OrderBean>{  private Text itemid;  private DoubleWritable amount;  public OrderBean() {  }  public OrderBean(Text itemid, DoubleWritable amount) {  set(itemid, amount);  }  public void set(Text itemid, DoubleWritable amount) {  this.itemid = itemid;  this.amount = amount;  }  public Text getItemid() {  return itemid;  }  public DoubleWritable getAmount() {  return amount;  }  @Override  public int compareTo(OrderBean o) {  int cmp = this.itemid.compareTo(o.getItemid());  if (cmp == 0) {  cmp = -this.amount.compareTo(o.getAmount());  }  return cmp;  }  @Override  public void write(DataOutput out) throws IOException {  out.writeUTF(itemid.toString());  out.writeDouble(amount.get());    }  @Override  public void readFields(DataInput in) throws IOException {  String readUTF = in.readUTF();  double readDouble = in.readDouble();    this.itemid = new Text(readUTF);  this.amount= new DoubleWritable(readDouble);  }  @Override  public String toString() {  return itemid.toString() + "\t" + amount.get();  }  } |

编写MapReduce处理流程

|  |
| --- |
| /\*\*  \* 利用secondarysort机制输出每种item订单金额最大的记录  \* @author duanhaitao@itcast.cn  \*  \*/  public class SecondarySort {    static class SecondarySortMapper extends Mapper<LongWritable, Text, OrderBean, NullWritable>{    OrderBean bean = new OrderBean();    @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  String line = value.toString();  String[] fields = StringUtils.split(line, "\t");    bean.set(new Text(fields[0]), new DoubleWritable(Double.parseDouble(fields[1])));    context.write(bean, NullWritable.get());    }    }    static class SecondarySortReducer extends Reducer<OrderBean, NullWritable, OrderBean, NullWritable>{      //在设置了groupingcomparator以后，这里收到的kv数据 就是： <1001 87.6>,null <1001 76.5>,null ....  //此时，reduce方法中的参数key就是上述kv组中的第一个kv的key：<1001 87.6>  //要输出同一个item的所有订单中最大金额的那一个，就只要输出这个key  @Override  protected void reduce(OrderBean key, Iterable<NullWritable> values, Context context) throws IOException, InterruptedException {  context.write(key, NullWritable.get());  }  }      public static void main(String[] args) throws Exception {    Configuration conf = new Configuration();  Job job = Job.getInstance(conf);    job.setJarByClass(SecondarySort.class);    job.setMapperClass(SecondarySortMapper.class);  job.setReducerClass(SecondarySortReducer.class);      job.setOutputKeyClass(OrderBean.class);  job.setOutputValueClass(NullWritable.class);    FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  //指定shuffle所使用的GroupingComparator类  job.setGroupingComparatorClass(ItemidGroupingComparator.class);  //指定shuffle所使用的partitioner类  job.setPartitionerClass(ItemIdPartitioner.class);    job.setNumReduceTasks(3);    job.waitForCompletion(true);    }  } |

# MapReduce中的DistributedCache应用

## 4.1 Map端join案例

### 4.1.1 需求

实现两个"表"的join操作，其中一个表数据量小，一个表很大，这种场景在实际中非常常见，比如"订单日志" join "产品信息"

### 4.1.2 分析

--原理阐述

适用于关联表中有小表的情形；

可以将小表分发到所有的map节点，这样，map节点就可以在本地对自己所读到的大表数据进行join并输出最终结果

可以大大提高join操作的并发度，加快处理速度

--示例：先在mapper类中预先定义好小表，进行join

--并用distributedcache机制将小表的数据分发到每一个maptask执行节点，从而每一个maptask节点可以从本地加载到小表的数据，进而在本地即可实现join

### 4.1.3 实现

|  |
| --- |
| public class TestDistributedCache {  static class TestDistributedCacheMapper extends Mapper<LongWritable, Text, Text, Text>{  FileReader in = null;  BufferedReader reader = null;  HashMap<String,String> b\_tab = new HashMap<String, String>();  String localpath =null;  String uirpath = null;    //是在map任务初始化的时候调用一次  @Override  protected void setup(Context context) throws IOException, InterruptedException {  //通过这几句代码可以获取到cache file的本地绝对路径，测试验证用  Path[] files = context.getLocalCacheFiles();  localpath = files[0].toString();  URI[] cacheFiles = context.getCacheFiles();    //缓存文件的用法——直接用本地IO来读取  //这里读的数据是map task所在机器本地工作目录中的一个小文件  in = new FileReader("b.txt");  reader =new BufferedReader(in);  String line =null;  while(null!=(line=reader.readLine())){  String[] fields = line.split(",");  b\_tab.put(fields[0],fields[1]);  }  IOUtils.closeStream(reader);  IOUtils.closeStream(in);  }    @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  //这里读的是这个map task所负责的那一个切片数据（在hdfs上）  String[] fields = value.toString().split("\t");  String a\_itemid = fields[0];  String a\_amount = fields[1];  String b\_name = b\_tab.get(a\_itemid);  // 输出结果 1001 98.9 banan  context.write(new Text(a\_itemid), new Text(a\_amount + "\t" + ":" + localpath + "\t" +b\_name ));    }  }  public static void main(String[] args) throws Exception {    Configuration conf = new Configuration();  Job job = Job.getInstance(conf);    job.setJarByClass(TestDistributedCache.class);    job.setMapperClass(TestDistributedCacheMapper.class);    job.setOutputKeyClass(Text.class);  job.setOutputValueClass(LongWritable.class);    //这里是我们正常的需要处理的数据所在路径  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));    //不需要reducer  job.setNumReduceTasks(0);  //分发一个文件到task进程的工作目录  job.addCacheFile(new URI("hdfs://hadoop-server01:9000/cachefile/b.txt"));    //分发一个归档文件到task进程的工作目录  // job.addArchiveToClassPath(archive);  //分发jar包到task节点的classpath下  // job.addFileToClassPath(jarfile);    job.waitForCompletion(true);  }  } |

# MapReduce的其他补充

## 5.1 计数器应用

在实际生产代码中，常常需要将数据处理过程中遇到的不合规数据行进行全局计数，类似这种需求可以借助MapReduce框架中提供的全局计数器来实现

示例代码如下：

|  |
| --- |
| public class MultiOutputs {  //通过枚举形式定义自定义计数器  enum MyCounter{MALFORORMED,NORMAL}  static class CommaMapper extends Mapper<LongWritable, Text, Text, LongWritable> {  @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  String[] words = value.toString().split(",");  for (String word : words) {  context.write(new Text(word), new LongWritable(1));  }  //对枚举定义的自定义计数器加1  context.getCounter(MyCounter.MALFORORMED).increment(1);  //通过动态设置自定义计数器加1  context.getCounter("counterGroupa", "countera").increment(1);  }  } |

## 5.2 多job串联

一个稍复杂点的处理逻辑往往需要多个MapReduce程序串联处理，多job的串联可以借助MapReduce框架的JobControl实现

示例代码：

|  |
| --- |
| ControlledJob cJob1 = new ControlledJob(job1.getConfiguration());  ControlledJob cJob2 = new ControlledJob(job2.getConfiguration());  ControlledJob cJob3 = new ControlledJob(job3.getConfiguration());    // 设置作业依赖关系  cJob2.addDependingJob(cJob1);  cJob3.addDependingJob(cJob2);    JobControl jobControl = new JobControl("RecommendationJob");  jobControl.addJob(cJob1);  jobControl.addJob(cJob2);  jobControl.addJob(cJob3);    cJob1.setJob(job1);  cJob2.setJob(job2);  cJob3.setJob(job3);    // 新建一个线程来运行已加入JobControl中的作业，开始进程并等待结束  Thread jobControlThread = new Thread(jobControl);  jobControlThread.start();  while (!jobControl.allFinished()) {  Thread.sleep(500);  }  jobControl.stop();    return 0; |

5.3 Configuration对象高级应用

# MapReduce参数优化

MapReduce重要配置参数

## 11.1 资源相关参数

(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.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: ""

(4) 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: ""

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

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

//应该在yarn启动之前就配置在服务器的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

//shuffle性能优化的关键参数，应在yarn启动之前就配置好

1. mapreduce.task.io.sort.mb 100 //shuffle的环形缓冲区大小，默认100m
2. mapreduce.map.sort.spill.percent 0.8 //环形缓冲区溢出的阈值，默认80%

## 11.2 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 secsContainer killed by the ApplicationMaster."。

## 11.3 本地运行MapReduce 作业

设置以下几个参数:

mapreduce.framework.name=local

mapreduce.jobtracker.address=local

fs.defaultFS=local

## 11.4 效率和稳定性相关参数

(1) mapreduce.map.speculative: 是否为Map Task打开推测执行机制，默认为false

(2) mapreduce.reduce.speculative: 是否为Reduce Task打开推测执行机制，默认为false

(3) mapreduce.job.user.classpath.first & mapreduce.task.classpath.user.precedence：当同一个class同时出现在用户jar包和hadoop jar中时，优先使用哪个jar包中的class，默认为false，表示优先使用hadoop jar中的class。

(4) mapreduce.input.fileinputformat.split.minsize: 每个Map Task处理的数据量（仅针对基于文件的Inputformat有效，比如TextInputFormat，SequenceFileInputFormat），默认为一个block大小，即 134217728。