Mapreduce中文处理策略

hadoop源代码中涉及编码问题时都是写死的utf-8，但是不少情况下，也会遇到输入文件和输出文件需要GBK编码的情况。  
  
GBK编码文件的输入：  
  
（1）输入文件为GBK，则只需在mapper或reducer程序中读取Text时，进行一下转码，以确保都是以UTF-8的编码方式在运行。  
  
// 转码  
  
 Text newText = transformTextToUTF8(value, "GBK");  
 String line = newText.toString();  
  
//自定义以下方法  
  
public static Text transformTextToUTF8(Text text, String encoding) {  
    String value = null;  
    try {  
      value = new String(text.getBytes(), 0, text.getLength(), encoding);  
    } catch (UnsupportedEncodingException e) {  
      e.printStackTrace();  
    }  
    return new Text(value);  
  }  
  
（2）输出GBK文件，则重写TextOutputFormat类。  
  
public class GBKFileOutputFormat<K, V> extends FileOutputFormat<K, V>，把TextOutputFormat的源码拷过来，然后把里面写死的utf-8编码改成GBK编码。运行程序时设置：job.setOutputFormatClass(GBKFileOutputFormat.class);  
  
 private static final String utf8 = “UTF-8″;//这里被写死成了utf-8  
改为：  
 =>private static final String gbk = “gbk”;  
private void writeObject(Object o) throws IOException {  
      if (o instanceof Text) {  
        Text to = (Text) o;  
       out.write(to.getBytes(), 0, to.getLength());//这里也需要修改  
      } else {  
        out.write(o.toString().getBytes(utf8));  
      }  
    }  
=>把writeObject按如下进行修改。  
private void writeObject(Object o) throws IOException {  
      if (o instanceof Text) {  
        out.write(o.toString().getBytes(gbk));  
      }  
    }

package main;

import java.io.DataOutputStream;

import java.io.IOException;

import java.io.UnsupportedEncodingException;

import org.apache.hadoop.classification.InterfaceAudience;

import org.apache.hadoop.classification.InterfaceStability;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.fs.FSDataOutputStream;

import org.apache.hadoop.io.NullWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.io.compress.CompressionCodec;

import org.apache.hadoop.io.compress.GzipCodec;

import org.apache.hadoop.mapreduce.OutputFormat;

import org.apache.hadoop.mapreduce.RecordWriter;

import org.apache.hadoop.mapreduce.TaskAttemptContext;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.util.\*;

@InterfaceAudience.Public

@InterfaceStability.Stable

public class GbkOutputFormat<K, V> extends FileOutputFormat<K, V> {

public static String SEPERATOR = "mapreduce.output.textoutputformat.separator";

protected static class LineRecordWriter<K, V>

extends RecordWriter<K, V> {

private static final String utf8 = "GBK";

private static final byte[] newline;

static {

try {

newline = "\n".getBytes(utf8);

} catch (UnsupportedEncodingException uee) {

throw new IllegalArgumentException("can't find " + utf8 + " encoding");

}

}

protected DataOutputStream out;

private final byte[] keyValueSeparator;

public LineRecordWriter(DataOutputStream out, String keyValueSeparator) {

this.out = out;

try {

this.keyValueSeparator = keyValueSeparator.getBytes(utf8);

} catch (UnsupportedEncodingException uee) {

throw new IllegalArgumentException("can't find " + utf8 + " encoding");

}

}

public LineRecordWriter(DataOutputStream out) {

this(out, "\t");

}

/\*\*

\* Write the object to the byte stream, handling Text as a special

\* case.

\* @param o the object to print

\* @throws IOException if the write throws, we pass it on

\*/

private void writeObject(Object o) throws IOException {

if (o instanceof Text) {

// Text to = (Text) o;

// out.write(to.getBytes(), 0, to.getLength());

// } else {

out.write(o.toString().getBytes(utf8));

}

}

public synchronized void write(K key, V value)

throws IOException {

boolean nullKey = key == null || key instanceof NullWritable;

boolean nullValue = value == null || value instanceof NullWritable;

if (nullKey && nullValue) {

return;

}

if (!nullKey) {

writeObject(key);

}

if (!(nullKey || nullValue)) {

out.write(keyValueSeparator);

}

if (!nullValue) {

writeObject(value);

}

out.write(newline);

}

public synchronized

void close(TaskAttemptContext context) throws IOException {

out.close();

}

}

public RecordWriter<K, V>

getRecordWriter(TaskAttemptContext job

) throws IOException, InterruptedException {

Configuration conf = job.getConfiguration();

boolean isCompressed = getCompressOutput(job);

String keyValueSeparator= conf.get(SEPERATOR, "\t");

CompressionCodec codec = null;

String extension = "";

if (isCompressed) {

Class<? extends CompressionCodec> codecClass =

getOutputCompressorClass(job, GzipCodec.class);

codec = (CompressionCodec) ReflectionUtils.newInstance(codecClass, conf);

extension = codec.getDefaultExtension();

}

Path file = getDefaultWorkFile(job, extension);

FileSystem fs = file.getFileSystem(conf);

if (!isCompressed) {

FSDataOutputStream fileOut = fs.create(file, false);

return new LineRecordWriter<K, V>(fileOut, keyValueSeparator);

} else {

FSDataOutputStream fileOut = fs.create(file, false);

return new LineRecordWriter<K, V>(new DataOutputStream

(codec.createOutputStream(fileOut)),

keyValueSeparator);

}

}

}

job.setOutputKeyClass和job.setOutputValueClas在默认情况下是同时设置map阶段和reduce阶段的输出，也就是说只有map和reduce输出是一样的时候才不会出问题。

当map和reduce输出是不一样的时候就需要通过job.setMapOutputKeyClass和job.setMapOutputValueClas来设置map阶段的输出。

[Mapreduce中Combiner的使用及误区](http://blog.csdn.net/chen517611641/article/details/24429403)

问题提出：

众所周知，[**Hadoop**](http://lib.csdn.net/base/hadoop)框架使用Mapper将数据处理成一个<key,value>键值对，再网络节点间对其进行整理(shuffle)，然后使用Reducer处理数据并进行最终输出。    在上述过程中，我们看到至少两个性能瓶颈：（引用）

1. 如果我们有10亿个数据，Mapper会生成10亿个键值对在网络间进行传输，但如果我们只是对数据求最大值，那么很明显的Mapper只需要输出它所知道的最大值即可。这样做不仅可以减轻网络压力，同样也可以大幅度提高程序效率。
2. 使用专利中的国家一项来阐述数据倾斜这个定义。这样的数据远远不是一致性的或者说平衡分布的，由于大多数专利的国家都属于美国，这样不仅Mapper中的键值对、中间阶段(shuffle)的键值对等，大多数的键值对最终会聚集于一个单一的Reducer之上，压倒这个Reducer，从而大大降低程序的性能。

目标：

Mapreduce中的Combiner就是为了避免map任务和reduce任务之间的数据传输而设置的，Hadoop允许用户针对map task的输出指定一个合并函数。即为了减少传输到Reduce中的数据量。它主要是为了削减Mapper的输出从而减少网络带宽和Reducer之上的负载。

数据格式转换：

map: (K1, V1) → list(K2,V2)   
combine: (K2, list(V2)) → list(K3, V3)   
reduce: (K3, list(V3)) → list(K4, V4)

注意：combine的输入和reduce的完全一致，输出和map的完全一致

使用注意：

对于Combiner有几点需要说明的是：

1）有很多人认为这个combiner和map输出的数据合并是一个过程，其实不然，map输出的数据合并只会产生在有数据spill出的时候，即进行merge操作。

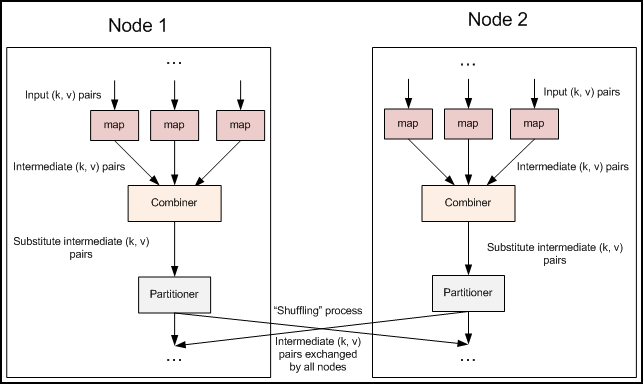
2）与mapper与reducer不同的是，combiner没有默认的实现，需要显式的设置在conf中才有作用。

3）并不是所有的job都适用combiner，只有操作满足结合律的才可设置combiner。combine操作类似于：opt(opt(1, 2, 3), opt(4, 5, 6))。如果opt为求和、求最大值的话，可以使用，但是如果是求中值的话，不适用。

4）一般来说，combiner和reducer它们俩进行同样的操作。

**但是：特别值得注意的一点，一个combiner只是处理一个结点中的的输出，而不能享受像reduce一样的输入（经过了shuffle阶段的数据），这点非常关键。具体原因查看下面的数据流解释：**

融合combiner的数据流



                          插入了Combiner的MapReduce数据流  
　　Combiner：前面展示的流水线忽略了一个可以优化MapReduce作业所使用带宽的步骤，这个过程叫Combiner，它在Mapper之后Reducer之前运行。Combiner是可选的，如果这个过程适合于你的作业，Combiner实例会在每一个运行map任务的节点上运行。Combiner会接收特定节点上的Mapper实例的输出作为输入，接着Combiner的输出会被发送到Reducer那里，而不是发送Mapper的输出。Combiner是一个“迷你reduce”过程，**它只处理单台机器生成的数据（特别重要，作者在做一个矩阵乘法的时候，没有领会到这点，把它当成一个完全的reduce的输入数据来处理，结果出错。）。**  
　　词频统计是一个可以展示Combiner的用处的基础例子，上面的词频统计程序为每一个它看到的词生成了一个（word，1）键值对。所以如果在同一个文档内“cat”出现了3次，（”cat”，1）键值对会被生成3次，这些键值对会被送到Reducer那里。通过使用Combiner，这些键值对可以被压缩为一个送往Reducer的键值对（”cat”，3）。现在每一个节点针对每一个词只会发送一个值到reducer，大大减少了shuffle过程所需要的带宽并加速了作业的执行。这里面最爽的就是我们不用写任何额外的代码就可以享用此功能！如果你的reduce是可交换及可组合的，那么它也就可以作为一个Combiner。你只要在driver中添加下面这行代码就可以在词频统计程序中启用Combiner。

HashMap遍历整个集合方法

for (java.util.Map.Entry<String, Long> entry : empMap.entrySet()) {

if (entry.getValue() > mgrSalary) {

context.write(new Text(entry.getKey()), new Text("" + entry.getValue()));

}

}

主函数不断调用Reduce中的reduce

**public** **static** **class** Reduce **extends** Reducer<IntWritable, Text, Text, IntWritable> {

**float** average = 0;

**int** count=0,sum=0;

@Override

**public** **void** reduce(IntWritable key, Iterable<Text> values, Context context) **throws** IOException, InterruptedException {

mapper<K1, V1, K2, V2>  
reducer<K2, V2, K3, V3>

if(K2, V2!= K3, V3){

job.setMapOutputKeyClass(K2.**class**);

job.setMapOutputValueClass(V2.**class**);

job.setOutputKeyClass(K3.**class**);

job.setOutputValueClass(V3.**class**);

}else{

job.setOutputKeyClass(K3.**class**);

job.setOutputValueClass(V3.**class**);

}

[hadoop的几个排序](http://blog.csdn.net/zhubing10061210/article/details/52511088)

1，map阶段的排序

map阶段的排序时对key进行排序，最简单的方式就是将要排序的字段封装成对象，然后这个对象实现WritableComparator接口重写compare这个比较方法,在shuffle阶段就会按照这个定义排序；

2，reduce阶段排序

其实在reduce阶段还有一个排序，因为reduce的数据来源于多个map，来源于map的数据是有序的，所以我们要对多个map的数据归并排序，怎么归并排序我们可以实WritableComparator；job.setSortComparatorClass(XXXXX.**class**);

3,分组job.setGroupingComparatorClass(GroupComparator.class);

在partition分区，每个map对数据按照key分到不同的本地分区数据文件中，每个key整合成一个记录，每次reduce方法调用处理一个记录，但是group的目的是让一次reduce调用处理多条记录;reduce方法每次是读一条记录，读到相应的key，但是处理value集合时，处理完当前记录的values后，还会判断下一条记录是不是和当前的key是不是同一个组，如果是的话，会继续读取这些记录的值，而这个记录也会被认为已经处理了，直到记录不是当前组，这次reduce调用才结束，这样一次reduce调用就会处理掉一个组中的所有记录，而不仅仅是一条了。

这个有什么用呢？如果不用分组，那么同一组的记录就要在多次reduce方法中独立处理，那么有些状态数据就要传递了，就会增加复杂度，在一次调用中处理的话，这些状态只要用方法内的变量就可以的。比如查找最大值，只要读第一个值就可以了。

**[java]** [view plain](http://blog.csdn.net/zhubing10061210/article/details/52511088) [copy](http://blog.csdn.net/zhubing10061210/article/details/52511088)

[在CODE上查看代码片](https://code.csdn.net/snippets/1877007)

1. **package** cn.zb.nginx;
3. **import** java.io.DataInput;
4. **import** java.io.DataOutput;
5. **import** java.io.IOException;
6. **import** java.util.Iterator;
7. **import** java.util.StringTokenizer;
9. **import** org.apache.hadoop.conf.Configuration;
10. **import** org.apache.hadoop.fs.Path;
11. **import** org.apache.hadoop.io.LongWritable;
12. **import** org.apache.hadoop.io.NullWritable;
13. **import** org.apache.hadoop.io.RawComparator;
14. **import** org.apache.hadoop.io.Text;
15. **import** org.apache.hadoop.io.WritableComparable;
16. **import** org.apache.hadoop.io.WritableComparator;
17. **import** org.apache.hadoop.mapreduce.Job;
18. **import** org.apache.hadoop.mapreduce.Mapper;
19. **import** org.apache.hadoop.mapreduce.Partitioner;
20. **import** org.apache.hadoop.mapreduce.Reducer;
21. **import** org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
22. **import** org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
24. **public** **class** SecondSort {

27. **public** **static** **class** SMapper **extends** Mapper<LongWritable, Text, Slog, Text>{
29. Slog log = **new** Slog();
30. Text active = **new** Text();
31. @Override
32. **protected** **void** map(LongWritable key, Text value,Context context)
33. **throws** IOException, InterruptedException {
35. StringTokenizer values = **new** StringTokenizer(value.toString());
36. **int** index = 0;
37. **while**(values.hasMoreTokens()){
39. **if**(index == 0){
40. log.setUserId(values.nextToken());
41. }**else** **if**(index == 1){
42. log.setVisitTime(Integer.parseInt(values.nextToken()));
43. }**else**{
44. active.set(values.nextToken());
45. }
46. index++;
47. }
48. context.write(log, active);
49. }
50. }
52. **public** **static** **class** SReducer **extends** Reducer<Slog, Text, Text, NullWritable>{
53. Text out = **new** Text();
54. @Override
55. **protected** **void** reduce(Slog log, Iterable<Text> values,Context context)
56. **throws** IOException, InterruptedException {
57. **int** i = 0;
59. Iterator<Text> iterator = values.iterator();
60. //只取出最大或者最小值
61. **while**(iterator.hasNext() && i == 0){
62. i++;
63. out.set(log.toString()+iterator.next());
64. context.write(out, NullWritable.get());
65. }
66. }
67. }

70. **public** **static** **void** main(String[] args) {
71. Configuration conf = **new** Configuration();
72. **try** {
73. Job job = Job.getInstance(conf);
74. job.setJarByClass(SecondSort.**class**);
75. job.setPartitionerClass(UserPatitioner.**class**);
76. job.setGroupingComparatorClass(GroupComparator.**class**);
78. job.setMapperClass(SMapper.**class**);
79. job.setReducerClass(SReducer.**class**);
81. job.setMapOutputKeyClass(Slog.**class**);
82. job.setMapOutputValueClass(Text.**class**);

85. job.setOutputKeyClass(Text.**class**);
86. job.setOutputValueClass(NullWritable.**class**);
87. //job.setNumReduceTasks(1);
88. String devSrc = "c:/wc/src3";
89. String devDest = "c:/wc/output3";
90. FileInputFormat.addInputPath(job, **new** Path(devSrc));
91. FileOutputFormat.setOutputPath(job, **new** Path(devDest));
93. **boolean** waitForCompletion = job.waitForCompletion(**true**);
94. System.out.println("#####################"+waitForCompletion+"########################");
95. } **catch** (Exception e) {
96. e.printStackTrace();
97. }
98. }
99. }
101. **class** GroupComparator **extends** WritableComparator{



106. **public** GroupComparator() {
107. **super**(Slog.**class**, **true**);
108. }
110. @Override
111. **public** **int** compare(WritableComparable a, WritableComparable b) {
113. Slog log1 = (Slog)a;
114. Slog log2 = (Slog)b;
116. **return** log1.getUserId().equals(log2.getUserId())?0 :(log1.getUserId().hashCode()>log2.getUserId().hashCode()?1:-1);
117. }
119. }
120. **class** UserPatitioner **extends** Partitioner<Slog, Text>{
122. @Override
123. **public** **int** getPartition(Slog key, Text value, **int** partitionNum) {
125. **return** key.getUserId().hashCode()%partitionNum;
126. }
128. }
130. **class** Slog **implements** WritableComparable<Slog>{
131. **private** String userId;
132. **private** Integer visitTime;
133. **public** String getUserId() {
134. **return** userId;
135. }
136. **public** **void** setUserId(String userId) {
137. **this**.userId = userId;
138. }
139. **public** Integer getVisitTime() {
140. **return** visitTime;
141. }
142. **public** **void** setVisitTime(Integer visitTime) {
143. **this**.visitTime = visitTime;
144. }

147. @Override
148. **public** **void** readFields(DataInput in) **throws** IOException {
150. userId = in.readUTF();
151. visitTime = in.readInt();
152. }
153. @Override
154. **public** **void** write(DataOutput out) **throws** IOException {
156. out.writeUTF(userId);
157. out.writeInt(visitTime);
158. }
159. @Override
160. **public** **int** compareTo(Slog o) {
162. **if**(!userId.equals(o.getUserId())){
163. **return** userId.hashCode() > o.getUserId().hashCode()? -1:1;
164. }**else** **if**(visitTime != o.getVisitTime()){
165. **if**(visitTime > o.visitTime){
166. **return** -1;
167. }**else** **if**(visitTime < o.visitTime){
168. **return** 1;
169. }
170. }
172. **return** 0;
173. }
174. @Override
175. **public** String toString() {
176. **return** userId + "\t" + visitTime + "\t";
177. }
178. @Override
179. **public** **int** hashCode() {
180. **final** **int** prime = 31;
181. **int** result = 1;
182. result = prime \* result + ((userId == **null**) ? 0 : userId.hashCode());
183. result = prime \* result
184. + ((visitTime == **null**) ? 0 : visitTime.hashCode());
185. **return** result;
186. }
187. @Override
188. **public** **boolean** equals(Object obj) {
189. **if** (**this** == obj)
190. **return** **true**;
191. **if** (obj == **null**)
192. **return** **false**;
193. **if** (getClass() != obj.getClass())
194. **return** **false**;
195. Slog other = (Slog) obj;
196. **if** (userId == **null**) {
197. **if** (other.userId != **null**)
198. **return** **false**;
199. } **else** **if** (!userId.equals(other.userId))
200. **return** **false**;
201. **if** (visitTime == **null**) {
202. **if** (other.visitTime != **null**)
203. **return** **false**;
204. } **else** **if** (!visitTime.equals(other.visitTime))
205. **return** **false**;
206. **return** **true**;
207. }



212. }