# Hadoop

## MapReduce

### 1 MapReduce基本原理

#### (★)什么是mapreduce

MapReduce是一种可用于数据处理的并行运算框架。基于该框架能够容易地编写应用程序，这些应用程序能够运行在由上千个商用机器组成的大集群上，并以一种可靠的，具有容错能力的方式并行地处理上TB级别的海量数据集。

#### (★)mr启动过程

client客户端向yarn集群(resourcemanager)提交任务

resourcemanager选择一个node创建appmaster

appmaster根据任务向rm(resourcemanager)申请资源

rm返回资源申请的结果

appmaster去对应的node上创建任务需要的资源（container形式，包括内存和CPU）

appmaster负责与nodemanager进行沟通，监控任务运行

最后任务运行成功，汇总结果

#### (★)怎样决定一个job的map和reduce的数量

具体会影响到map task 数量的因素(参数)有：

blockSize: HDFS block大小，默认为64MB

totalSize: 所有输入文件的总大小

splitSize: 每个Split的大小

numSplits: org.apache.hadoop.mapred.JobConf.setNumMapTasks(int n)设置

goalSize: 输入总大小与numSplits的比值，即期望每个Mapper处理的数据量

mapred.max.split.size参数，默认为long型最大值

mapred.min.split.size参数，默认为0(byte)

以下为计算方式：

minSize = max{min{totalSize/numSplits, blockSize}, mapred.min.split.size}

splitSize = max{minSize, min{maxSize, blockSize}}

mapNum ≈ totalSize/splitSize

reduce的数量job.setNumReduceTasks(x); //x 为reduce的数量.

如果不设置，一般reduce的数量由partition决定。

#### (★)Shuffle过程中排序用的什么算法

快速排序和归并排序

https://blog.csdn.net/nysyxxg/article/details/78272141

#### (★)说一下Combiner有什么作用？

MapReduce中的Combiner：

①是为了避免MapTask和ReduceTask之间的数据传输而设置的。

②Hadoop允许针对Map Task的输出指定一个合并函数。

③Combiner为了减少传输到Reduce阶段的数据量

减少了Mapper阶段的输出，从而减少网络带宽和Reduce阶段的负载

#### (★)HADOOP的一个job提交到resourcemanager之后，会生成什么容器来放这个job

resourceManager会在对应任务节点上生成资源容器container来存放job。

#### (★★★)job 的运行流程(提交一个 job 的流程)？

任务提交：在main方法中调用job.waitForCompletion(true)[这里的参数true是确定打印进度信息给用户]，waitForCompletion中又调用了job.submit(),这个时候提交的过程就开始了。

提交作业到集群：

当提交开始后，作业就进入到了定义状态（DEFINE）。之后会默认地使用新的api；

获取ClientProtocol代理对象：job提交到yarn上实际上就是客户端与RM进行RPC通信，这里使用的是ClientProtocol的通讯协议，所以就需要在客户端获取到ClientProtocol的代理对象。这个过程就是由job.connect()完成的。它返回的是一个集群信息对象（cluster）。Cluster内部构建了这个代理对象。

获取提交器：获取到集群信息对象后就会根据该对象封装的信息创建一个job的提交器对象（jobSubmitter）。【getJobSubmitter(cluster.getFileSystem(), cluster.getClient())】.

通讯

客户端与RM的通信操作：客户端拿到了通信的代理对象后就会与RM进行通信，并完成一系列的操作：

检查作业输出路径是否存在，若存在就报异常。【checkSpecs(job)】；

然后将job的配置对象放入到RM的框架中(JobSubmitter.addMRFrameworkToDistributedCache())；

获取登录目录路径【JobSubmissionFiles.getStagingDir(cluster, conf)】：获取这个路径后会与后面形成的jobID拼接形成一个submitJobDir（job资源提交路径：hdfs://……./. Staging/jobid ）的目录路径。这个目录用于存放客户端提交的作业文件。

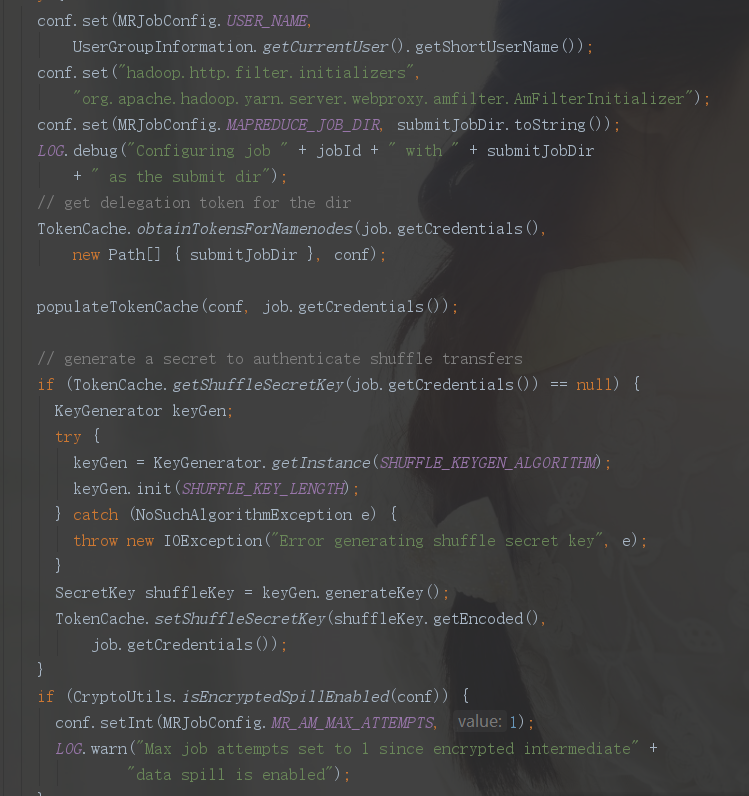
submitJobDir

配置提交任务的主机ID与主机名；

创建一个新的jobID并将这个ID设置到job上；

创建id

配置相关权限；



CopyAndConfigureFiles(job, submitJobDir)：将job所需的全部文件上传到对应的目录下；（-libjars, -files, -archives）

计算作业的输入分片（JobSubmitter.writeSplits()）：

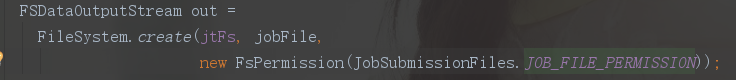
这里面会获取到一个job的配置对象，如果配置对象里采用的是新的mapper就会根据这个新的作业计算新的分分片数，如果不是新的mapper就会沿用以前的分片数。（自己理解的）新计算分片数的话：会根据FileInputFormat类的规则（文件总剩余大小大于块大小的110%时分片大小就等于块大小，否则就切割成一片）进行分片。

分片完成后会将inputSpilt放入到一个List集合中然后会将这个集合转化成一个泛型数组，再将数组里的元素按照文件大小进行排序，排好序后会取出数组的长度（即分片个数）；

泛型数组

之后会将分片个数配置到集群中的job配置里。conf.setInt(MRJobConfig.NUM\_MAPS, maps);

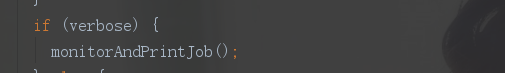
Job.writeConf()这个现将job的配置转化成一个job.xml文件，再将这个文件上传到文件系统中（JOB\_FILE\_PERMISSION：rw-r--r--）；



JobSubmitter.submitClient.submitJob()：到这一步就是真正的job的提交，注意的是这里的提交者是cluster对象中构建的ClientProtocol的代理对象，返回一个状态信息。

真正提交

提交完成后持续监控程序并输出监控信息；



返回当前MR应用程序最终成功与否的状态。（return isSuccessful()）

#### (★)1000个maptask任务,同时在运行,reduce所拉取的文件是如何找到相应的map端

Hadoop 2.x版本前

map任务一直和其父TaskTracker保持联系，而TaskTracker又一直和JobTracker保持心跳。所以JobTracker中保存了整个集群中的宏观信息。只要reduce任务向JobTracker获取对应的map输出位置就可以了。

2.x版本后

ApplicationMaster取代了JobTracker、TaskTracker的作用，reduce任务直接向ApplicationMaster获取输出位置就行。

#### (★) “MapReduce 2.0”与“YARN”是否等同，尝试解释说明

MapReduce 2.0是一个计算框架，在运行时环境上引入全新的资源管理框架：YARN，将JobTracker中的资源管理和作业控制功能分开，分别由两个不同进程ResourceManager和ApplicationMaster实现，ResourceManager负责所有应用程序的资源分配，Application负责管理一个应用程序。

YARN：是资源管理框架

#### (★★)MapReduce 进阶知识：Hadoop 的几种文件格式、常见输入输出格式化类、多输入多输出机制。

Hadoop的文件格式：

Sequence File

RCFile

Avro

文本格式

外部格式

常见的输入输出格式化类：

DBInputFormat 、 FileInputFormat 、 KeyValueTextInputFormat 、NlineInputFormat、 TextInputFormat

多输入多输出机制：

多路径输出使用MultipleInputs类的addInputPath（）方法，代码如下：

private void setInputPathMothed3(Job job) throws IOException {

MultipleInputs.addInputPath(job, new Path(inputPath\_1), TextInputFormat.class, CoreComputer.CoreMapper.class);

MultipleInputs.addInputPath(job, new Path(inputPath\_2), TextInputFormat.class, CoreComputer.CoreMapper.class);

MultipleInputs.addInputPath(job, new Path(inputPath\_3), TextInputFormat.class, CoreComputer.CoreMapper.class);

}

多路径输出：

将 MultipleOutputs 的初始化放在 setup() 中，因为在 setup() 只会被调用一次，如果放在 reduce() 中，则 MultipleOutputs 可能被 reduce 方法初始化 N 次，而你全然不知； 你需要在 cleanup() 方法中关闭 MultipleOutputs。通过源码我们了解到，关闭 MultipleOutputs，也就是关闭 RecordWriter，并且是一堆 RecordWriter，因为这里会有很多 reduce 被调用。

还有一个是你需要重点关注的，那就是 reduce() 方法里的 multipleOutputs.write(…)。你需要把以前的 context.write(…) 替换成现在的这个。代码如下：

public static class CoreReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

private MultipleOutputs<Text, IntWritable> multipleOutputs = null;

@Override

protected void setup(Reducer<Text, IntWritable, Text, IntWritable>.Context context)

throws IOException, InterruptedException {

multipleOutputs = new MultipleOutputs<Text, IntWritable>(context);

}

@Override

protected void reduce(Text key, Iterable<IntWritable> values,

Reducer<Text, IntWritable, Text, IntWritable>.Context context)

throws IOException, InterruptedException {

( ... 省略无关的 N 行 ... )

multipleOutputs.write(splitKeys[1], new Text(splitKeys[0]), count);

}

@Override

protected void cleanup(Reducer<Text, IntWritable, Text, IntWritable>.Context context)

throws IOException, InterruptedException {

multipleOutputs.close();

}

}

#### (★★★)MapReduce 的 shuffle 工作原理、resourceManager如何创建 map 任务和 reduce任务是面试的重点。

Shuffle工作原理：

**maptask阶段**

在将数据给到MapTask进行处理时会使用InputFormat接口（常见的是TextinputFormat实现类），创建一个RecordReader抽象类（其实现类是LineRecordReader）使用readLine()方法读取Split内的数据，并按照<key,value>（key是行偏移量（行号），value为行内容）的形式传给Mapper进行处理。

**逻辑运算：**

然后进入到Mapper 映射器，进行具体的逻辑运算，运算结束后就输出key/value，key/value 会被序列化为字节流，由OutputCollector 接口收集这些输出的数据，然后通过这个它的collect（）方法写入到环形缓冲区（环形缓冲区实质是在内存上的字节数组，默认大小100M，这个值可以通过改变mapreduce.task.io.sort.mb属性来调整）；

运算结果输出到环形缓冲区：

当缓冲区中的数据达到阈值（默认80%）的时候，就调用spill单线程对临时缓冲区的数据进行分区，排序和可选的合并操作之后临时写入磁盘（具体内容在下一步进行解释），然后重新利用这块缓冲区。这个spill是单线程，不影响往缓冲区写map结果的线程。spill线程也不应该阻止map方法的结果的输出，所以整个缓冲区有个溢写的比例，这个比例默认为0.8，也就是当缓冲区的数据已经达到阈值，spill线程启动，锁定这80%的内存，执行溢写过程。Maptask 的输出结果还可以往剩下的20%的内存中写，互不影响。

**Spill线程对环形缓冲区的数据进行分区、排序、合并等操作：**

分区：我们会根据默认的HashPartitioner或者是自定义的Partitioner对key进行分区。每个分区都会分配指定的ReduceTask。

排序：分区之后我们会对每个分区的key进行单独的字典升序排序。这步操作就让之后的Merge操作节省了重复读取的消耗。

合并：他的使用在步骤11 进行了解释说明。

**溢写文件：**

当数据进行完分区，排序时，会从环形缓冲区中溢写到磁盘文件中，注意这个文件是分区的，并且区内是有序的。

如果环形缓冲区内的数据小于其最大输出阈值（默认80%），又大于最小输出阈值（可以自定义），仍然会溢写到磁盘中；但是如果环形缓冲区内的数据足够小（小于最小输出的阈值），就不会溢写，直接保留在内存中，ReduceTask直接从内存中拉取数据。

**归并排序：**

情况一：如果是第一个MapTask 的128 M 的文件，将产生两个spill 文件，会执行Merge 步骤：先将上一步溢出到磁盘的文件全部读取到内存中，然后进行归并排序，最后再写入到磁盘中。（文件也是分区且有序的）；

情况二：如果我们上一步环形缓冲区的文件小于或等于阈值，只产生一个spill 文件，例如第二个MapTask 的 72 M 的文件，那个就不会有Merge 这一步。

**使用了combiner的情况：**

combiner相当于是一个Reducer规约器，作用是将数据进行预聚合, 将相同分区的相同key的键值对的value相加,以减少溢写到磁盘的数据量,（注意：combiner决不能改变最终的计算结果，所以要谨慎使用）。作用范围在Mapper阶段。

如果客户端设置了combiner，在spill线程写到磁盘中的文件数量大于等于3的时候（combiner默认是大于等于3个文件启动，可以通过mapreduce.map.combine.minspills属性设置），等到分片的所有数据都处理完毕以后，开始对这些溢写到磁盘中的文件进行 Merge （归并排序）完成后，在内存中进行combine 操作，combine操作完成后，会将数据落地到磁盘文件中，此时merge阶段结束。注意，Merge的默认最大处理的文件数是10（mapreduce.task.io.sort.factor（default：10）），如果spill文件数大于这个最大处理数，就会有多次Merge操作。

**开始启动reduceTask：**

当MapTask完成任务数超过总数的5%后（默认），启动相应数量的ReduceTask，并告知ReduceTask处理的数据范围（数据分区）。

**ReduceTask 拉取文件：**

ReduceTask默认启动5个copy线程(可以修改mapreduce.reduce.shuffle.parallelcopies属性的默认值)到完成的MapTask任务节点上分别copy一份属于自己的数据（使用Http的方式）。MapTask完成后，它会使用心跳机制通知它们的application master。applicationmaster记录了Mapper输出和主机的位置之间的映射关系，Reducer中的一个线程定期询问master以便获得Mapper输出主机的位置，直到获得所有输出位置；

Copy过来的数据会先放入内存缓冲区中，如果内存缓冲区中能放得下这次数据的话就直接把数据写到内存中，即内存到内存merge。Reduce要向每个Map去拖取数据，在内存中每个Map对应一块数据，当内存缓存区中存储的Map数据占用空间达到一定程度的时候，开始启动内存中merge，把内存中的数据merge输出到磁盘上一个文件中，即内存到磁盘merge。Reduce的内存缓冲区可通过mapred.job.shuffle.input.buffer.percent配置，默认是JVM的heap size的70%。内存到磁盘merge的启动门限可以通过mapred.job.shuffle.merge.percent配置，默认是66%。

当属于该reducer的map输出全部拷贝完成，则会在reducer上生成多个文件（如果拖取的所有map数据总量都没有内存缓冲区，则数据就只存在于内存中），这时开始执行合并操作，即磁盘到磁盘merge，Map的输出数据已经是有序的，Merge进行一次合并排序，所谓Reduce端的sort过程就是这个合并的过程，采取的排序方法跟map阶段不同，因为每个map端传过来的数据是排好序的，因此众多排好序的map输出文件在reduce端进行合并时采用的是归并排序，针对键进行归并排序。一般Reduce是一边copy一边sort，即copy和sort两个阶段是重叠而不是完全分开的。最终Reduce shuffle过程会输出一个整体有序的数据块。

注意：由于reducer有可能失败，因此主机并没有在第一个reducer检测到map输出时就立即从磁盘上删除它们。相反，主机会等待，直到application master告诉它删除map输出，这是Job完成后执行的。

**进行Reducer中的逻辑运算：**

一次读取一组数据到Reducer中；一组指的是相同Key的键值对,然后通过reduce方法进行逻辑运算；

**分组：**

要知道同一个Reducetask中放入的数据的key并不是就一种，当有多个key进入时，就需要将相同key的key/value对进行分组。当然，如果我们的key是一个自定义的bean对象时，相同key的定义就是bean对象的所有成员变量都相等。而有的时候并不是都相等。这时候我们就要用到GroupingComparator组件来自定义分组的规则（可以理解为是欺骗reduce进行分组）。

**Reducer 规约器输出：**

用到的是 OutPutFormat 输出格式化器组件，默认是TextOutputFormat，调用RecordWriter抽象类的LineRecordWriter类的write()方法将运算好的数据写入到磁盘中。有几个ReduceTask就会生产几个文件，文件名从Part-r-00000->Part-r-00001…

#### (★)每个 map 槽就是一个线程。（ ） 错误

分析：首先我们要知道什么是map 槽, map 槽 -> map slot

mapslot只是一个逻辑值 ( org.apache.hadoop.mapred.TaskTracker.TaskLauncher.numFreeSlots )，而不是对应着一个线程或者进程

#### (★)例举 hadoop 中定义的最常用的 InputFormats，哪个是默认的？

提 示 ： DBInputFormat 、 FileInputFormat （ KeyValueTextInputFormat 、NlineInputFormat、 TextInputFormat）

默认：TextInputFormat

#### (★)TextInputFormat 和 KeyValueInputFormat 类之间的不同之处在于哪里？

提示：TextInputFormat 中的 key 表示行的偏移量，value 是行文本内容

KeyValueInputFormat 的 key value 是通过第一个制表符进行划分的

#### (★★)Mapreduce 的 input split 就是一个 block。（ ） 错误

大部分情况下一个inputsplit不会是一个block，

因为可能存在跨Block的一行数据，所以这方面LineRecordReader使用了一个简单而巧妙的方法:既然无法断定每一个split开始的一行是独立的一行还是被切断的一行的一部分,那就跳过每个split的开始一行(当然要除第一个split之外),从第二行开始读取,然后在到达split的结尾端时总是再多读一行,这样数据既能接续起来又避开了断行带来的麻烦。

还有一种情况就是剩下的block的大小小于默认大小（128M可自己设置）的10%，就会被直接分配给上一个block的inputsplit。

#### (★★)hadoop 中 job 和 task 之间是什么关系？

一个mapreduce为一个job，而一个job包含一个或多个task，task又分为maptask与reducetask

#### (★★★)MapReduce 中排序发生在哪几个阶段？这些排序是否可以避免？为什么？

一个MapReduce作业由Map阶段和Reduce阶段两部分组成，这两个阶段会对数据排序，从这个意义上说，MapReduce框架本质上就是一个DistributedSort。在Map阶段，MapTask会在本地磁盘输出一个按照key排序（采用的是快速排序）的文件（中间可能产生多个文件，但最终会合并成一个），在Reduce阶段，每个ReduceTask会对收到的数据排序，这样，数据便按照key分成了若干组，之后以组为单位交给reduce()处理。很多人的误解在Map阶段，以为如果不使用Combiner便不会排序，这是错误的，不管你用不用Combiner，MapTask均会对产生的数据进行排序（如果没有ReduceTask，则不会排序，实际上Map阶段的排序就是为了减轻Reduce端排序负载）。由于这些排序是MapReduce系统自动完成的，用户无法控制，因此，在hadoop1.x中无法避免，也不可以关闭，但hadoop2.x是可以关闭的。

#### (★★)hadoop mapreduce 创建类 DataWritable 的作用是什么？

序列化，数据需要在网络上传输到别的节点，需要序列化，对对象进行流化，在网络上才能传输，才可以传输。

#### (★★)为什么创建类 DataWritable？

现有的Hadoop Writable应用已得到很好的优化，但为了对付更复杂的结构，最好创建一个新的Writable类型，而不是已有的类型。

#### (★★)flush 的过程？

一般主要用在IO中，即清空缓冲区数据，就是说你用读写流的时候，其实数据是先被读到了内存中，然后用数据写道文件中，当你数据读完的时候不代表你的数据已经写完了，因为还有一部分有可能会留在内存这个缓冲区中。这时候你如果调用了close方法关闭了读写流，那么这部分数据就会丢失，所以应该在关闭读写流之前先flush（），先清空数据。

### 2 MapReduce组件

#### (★★★)mapreduce中常见的组件有哪些？

**Combiner**

Combiner是MapReduce程序中常见三大组件之一，他的作用在MapTask之后给MapTask的结果进行局部汇总，以减轻reducetask的计算负载，减少网络传输。

**Partitioner**

Mapreduce常见的三大组件之一，它的作用是根据将 map 输出的 kv 对，按照相同 key 分组，然后分发给不同的reducetask。

**Sort**

Mapreduce常见的三大组件之一，它可以对map输出的kv对根据key来进行比较进行排序，然后在传入Reducer。

**Mapper**

Mappreduce主要组件之一，作用是处理原始数据、为数据打标签、对数据进行分发。

**Reducer**

Mapreduce主要组件之一，作用对Mapper处理后的分组后的数据进行聚合。

#### (★★★)MapReduce中Combiner和Partition的作用

Combiner是Reduce的实现，在Map端运行计算任务，减少Map端的输出数据。

作用就是优化。

但是Combiner的使用场景是MapReduce的Map输出结果必须和Reduce输入输出一样。

Partition的默认实现是HashPartition，是Map端将数据按照Reduce个数取余，进行分区，不同的Reduce来Copy自己的数据。

Partiton的作用是将数据分到不同的Reduce进行计算，加快计算效果

#### (★★)combiner出现在那个过程

前提是参数设置combiner已开启

1）map输出数据根据分区排序完成后，在写入文件之前会执行一次combine操作;

2）如果map输出比较大，溢出文件个数大于3(此值可以通过属性min.num.spills.for.combine配置)时，在merge的过程(多个spill文件合并为一个大文件)中前还会执行combiner操作;

#### (★★)MapReduce 2.0 中，MRAppMaster 主要作用是什么，MRAppMaster 如何实现任务容错的？

作用：

 MRAppMaster是MapReduce的ApplicationMaster实现，它使得MapReduce应用程序可以直接运行于YARN之上。在YARN中，MRAppMaster负责管理MapReduce作业的生命周期，包括作业管理、资源申请与再分配、Container启动与释放、作业恢复等。

容错：

MRAppMaster采用了任务级别的恢复机制，即以任务为基本单位进行恢复，这种机制是基于事务型日志完成作业恢复的，它只关注两种任务：运行完成的任务和未完成的任务。作业执行过程中，MRAppMaster会以日志的形式将作业以及状态记录下来，一旦MRAppMaster重启，则可从日志中恢复作业的运行状态。

      当前MRAppMaster的作业恢复机制仅能做到恢复上一次已经运行完成的任务，对于正在运行的任务，则在前一次MRAppMaster运行实例退出时由ResourceManager强制将其杀死并回收资源。

#### (★★★)MapReduce 执行流程：“天龙八步”，计数器、自定义分区、自定义排序、自定义分组、如何对 value 进行排序：次排序+自定义分组、归约？

**“天龙八步”：**

partition过程：每一组<key, value>在被收集的时候，就已经确定了分区（partition），即在这个时候就已经确定了要交给哪个reduce任务处理。分区会给<key, value>加上一个索引标识。假设分区后（分区算法可以设定，默认是hash值模运算）；

spill过程：缓冲区默认是100M，每当里面的数据达到80M（比例80%，这个比例也可以人为设置），就会另起一个线程SpillThread往磁盘溢写，每次溢写都会产生一个数据文件和对应的索引文件。

sort过程：在溢写的过程中一直在排序，比较算法可以定制，默认排序算法是快速排序（可以人为设定），排序的过程就是一些位置的索引在不断的变化。

combine过程：这个过程默认是没有的，需要明确指定combiner。combiner其实就是一个reducer，可以让数据交给reduce任务之前，进行一些计算、合并。它的意义在于，使数据进一步减少，减轻了reduce任务通过网络获取数据的压力和reduce处理数据的压力。combiner也可以自己定制

merge过程：一个map所有的溢写文件都会进行合并，产生一个最终的溢写文件和一个索引文件。合并是针对于不同的溢写文件中相同分区的数据。在这个合并的过程中，也会进行combine操作（如果设置了的话），此处的combine过程同上

copy数据过程：每个reduce任务会远程copy属于自己的多个map输出数据文件，通过http传输，在本地会合并。另外，这个过程也会进行combine

sort过程：会对上不copy来的数据进行排序。

GroupComparator分组过程：通过一个GroupComparator对key进行比较，相同的key（比较结果为0，比较算法可以定制），会被放到一个集合的迭代器中，然后迭代进行一次reduce运算，产生一个输出。

**计数器：**

MapReduce 计数器（Counter）为我们提供一个窗口，用于观察 MapReduce Job 运行期的各种细节数据。对MapReduce性能调优很有帮助，MapReduce性能优化的评估大部分都是基于这些 Counter 的数值表现出来的。内置的计数器一般有任务计数器、文件系统计数器、FileInputFormat计数器、FileOutputFormat计数器。

**自定义分区：**

自定义分区很简单，我们只需要继承抽象类Partitioner，重写getPartition方法即可，另外还要给任务设置分区：job.setPartitionerClass()，就可以了。   
注意： 自定义分区的数量需要和reduce task的数量保持一致。

**自定义排序：**

默认mapreduce是对key字符串按照字母进行排序的，而我们想任意排序，只需要把key设成一个类，再对该类写一个compareTo（大于要比较对象返回1，等于返回0，小于返回-1）方法就可以了。

注：这里要实现 WritableComparable。

**规约：**

用到的是 OutPutFormat 输出格式化器组件，默认是TextOutputFormat，调用RecordWriter抽象类的LineRecordWriter类的write()方法将运算好的数据写入到磁盘中。有几个ReduceTask就会生产几个文件。

#### (★★)如果 hadoop 中没有定义定制分区，那么如何在输出到 reducer 前执行数据分区？

默认的分区器为各个键计算一个哈希值，并分配给基于这个结果的分区

#### (★★)如何设置 compression

map的压缩输出

conf.setBoolean("mapred.compress.map.out", true);//设置map输出压缩

conf.setClass(Job.MAP\_OUTPUT\_COMPRESS\_CODEC, GzipCodec.class, CompressionCodec.class);

reduce的压缩输出

conf.setBoolean("mapred.output.compress", true);//设置输出压缩

### 3 MapReduce代码及实现思路

#### (★★)两个单词文件求交集MR代码

public int[] intersect(int[] nums1, int[] nums2) {

int[] news = new int[0];

if (nums1 == null || nums2 == null) {

return null;

}

if (nums1.length == 0 || nums2.length == 0) {

return news;

}

news = new int[nums1.length];

int[] count = new int[nums1.length];

int index = 0;

for (int i = 0; i < nums2.length; i++) {

for (int j = 0; j < nums1.length; j++) {

if (nums1[j] == nums2[i] && count[j] == 0) {

news[index] = nums1[j];

index++;

count[j] = 1;

break;

}

}

}

int[] ints = new int[index];

for (int i = 0; i < ints.length; i++) {

ints[i] = news[i];

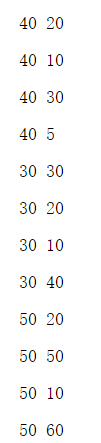
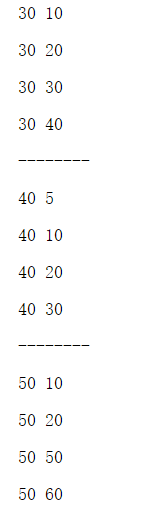
}

return ints;

}

#### (★★★)二次排序

sort.txt内容: 输出:

public class SecondarySort {

public static class Map extends Mapper<LongWritable, Text, IntPair, IntWritable> {

public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

String line = value.toString();

StringTokenizer tokenizer = new StringTokenizer(line);

int left = 0;

int right = 0;

if (tokenizer.hasMoreTokens()) {

left = Integer.parseInt(tokenizer.nextToken());

if (tokenizer.hasMoreTokens())

right = Integer.parseInt(tokenizer.nextToken());

context.write(new IntPair(left, right), new IntWritable(right));

}

}

}

/\*

\* 自定义分区函数类FirstPartitioner，根据 IntPair中的first实现分区

\*/

public static class FirstPartitioner extends Partitioner<IntPair, IntWritable>{

@Override

public int getPartition(IntPair key, IntWritable value,int numPartitions){

return Math.abs(key.getFirst() \* 127) % numPartitions;

}

}

/\*

\* 自定义GroupingComparator类，实现分区内的数据分组

\*/

@SuppressWarnings("rawtypes")

public static class GroupingComparator extends WritableComparator{

protected GroupingComparator(){

super(IntPair.class, true);

}

@Override

public int compare(WritableComparable w1, WritableComparable w2){

IntPair ip1 = (IntPair) w1;

IntPair ip2 = (IntPair) w2;

int l = ip1.getFirst();

int r = ip2.getFirst();

return l == r ? 0 : (l < r ? -1 : 1);

}

}

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

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

for (IntWritable val : values) {

context.write(new Text(Integer.toString(key.getFirst())), val);

}

}

}

public static void main(String[] args) throws IOException, InterruptedException, ClassNotFoundException {

// 读取配置文件

Configuration conf = new Configuration();

// 判断路径是否存在，如果存在，则删除

Path mypath = new Path(args[1]);

FileSystem hdfs = mypath.getFileSystem(conf);

if (hdfs.isDirectory(mypath)) {

hdfs.delete(mypath, true);

}

Job job = new Job(conf, "secondarysort");

// 设置主类

job.setJarByClass(SecondarySort.class);

// 输入路径

FileInputFormat.setInputPaths(job, new Path(args[0]));

// 输出路径

FileOutputFormat.setOutputPath(job, new Path(args[1]));

// Mapper

job.setMapperClass(Map.class);

// Reducer

job.setReducerClass(Reduce.class);

// 分区函数

job.setPartitionerClass(FirstPartitioner.class);

// 本示例并没有自定义SortComparator，而是使用IntPair中compareTo方法进行排序 job.setSortComparatorClass();

// 分组函数

job.setGroupingComparatorClass(GroupingComparator.class);

// map输出key类型

job.setMapOutputKeyClass(IntPair.class);

// map输出value类型

job.setMapOutputValueClass(IntWritable.class);

// reduce输出key类型

job.setOutputKeyClass(Text.class);

// reduce输出value类型

job.setOutputValueClass(IntWritable.class);

// 输入格式

job.setInputFormatClass(TextInputFormat.class);

// 输出格式

job.setOutputFormatClass(TextOutputFormat.class);

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

#### (★★★)给定两张表，手写MR进行join操作

表一：NAME，SEX 表二：NAME，AGE

public class MapJoin {

public static class MyMapper extends Mapper<LongWritable, Text, Text, Text>{

Map<String,String> tableA = new ConcurrentHashMap<>();

Map<String,String> tableB = new ConcurrentHashMap<>();

@Override

protected void setup(Context context)

throws IOException, InterruptedException {

URI[] uris = context.getCacheFiles();

for(URI u : uris) {

String uriPath = u.toString();

if(uriPath.endsWith("tableA")) {

BufferedReader bs = new BufferedReader(new FileReader(new File(u.toString())));

String str = null;

while((str = bs.readLine())!=null) {

String[] strs = str.split("\t");

sexMap.put(strs[0], strs[1]);

}

}else if(uriPath.endsWith("tableB")){

BufferedReader bs = new BufferedReader(new FileReader(new File(u.toString())));

String str = null;

while((str = bs.readLine())!=null) {

String[] strs = str.split("\t");

userMap.put(strs[0], strs[1]);

}

}

}

}

@Override

protected void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException {

if(tableA.containsKey(name) && tableB.containsKey(name)) {

//输出

context.write(new Text(sex+"\t"+tableA.get(name)+"\t"+"age"+tableB.get(name)+"\t"), new Text(""));

}

}

}

}

#### (★★★)有三个map,一个reduce来做top10.哪种方法最优。数据量特别大。

package jtlyuan.csdn;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.conf.Configured;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

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

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

import org.apache.hadoop.util.Tool;

import org.apache.hadoop.util.ToolRunner;

//利用MapReduce求最大值海量数据中的K个数

public class TopKNum extends Configured implements Tool {

public static class MapClass extends Mapper<LongWritable, Text, IntWritable, IntWritable> {

public static final int K = 100;

private int[] top = new int[K];

public void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException {

String[] str = value.toString().split(",", -2);

try {// 对于非数字字符我们忽略掉

int temp = Integer.parseInt(str[8]);

add(temp);

} catch (NumberFormatException e) {

}

}

private void add(int temp) {//实现插入

if(temp>top[0]){

top[0]=temp;

int i=0;

for(;i<99&&temp>top[i+1];i++){

top[i]=top[i+1];

}

top[i]=temp;

}

}

@Override

protected void cleanup(Context context) throws IOException,

InterruptedException {

for(int i=0;i<100;i++){

context.write(new IntWritable(top[i]), new IntWritable(top[i]));

}

}

}

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

public static final int K = 100;

private int[] top = new int[K];

public void reduce(IntWritable key, Iterable<IntWritable> values, Context context)

throws IOException, InterruptedException {

for (IntWritable val : values) {

add(val.get());

}

}

private void add(int temp) {//实现插入if(temp>top[0]){

top[0]=temp;

int i=0;

for(;i<99&&temp>top[i+1];i++){

top[i]=top[i+1];

}

top[i]=temp;

}

}

@Override

protected void cleanup(Context context) throws IOException,

InterruptedException {

for(int i=0;i<100;i++){

context.write(new IntWritable(top[i]), new IntWritable(top[i]));

}

}

}

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

Configuration conf = getConf();

Job job = new Job(conf, "TopKNum");

job.setJarByClass(TopKNum.class);

FileInputFormat.setInputPaths(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1]));

job.setMapperClass(MapClass.class);

job.setCombinerClass(Reduce.class);

job.setReducerClass(Reduce.class);

job.setInputFormatClass(TextInputFormat.class);

job.setOutputFormatClass(TextOutputFormat.class);

job.setOutputKeyClass(IntWritable.class);

job.setOutputValueClass(IntWritable.class);

System.exit(job.waitForCompletion(true) ? 0 : 1);

return 0;

}

public static void main(String[] args) throws Exception {

int res = ToolRunner.run(new Configuration(), new TopKNum(), args);

System.exit(res);

}

}

/\*

\* 列举一部分出来：

\* 306 306

307 307

309 309

313 313

320 320

346 346

348 348

393 393

394 394

472 472

642 642

706 706

868 868

\*/

#### (★★)用MapReduce来实现下面需求？

现在有10个文件夹,每个文件夹都有1000000个url.现在让你找出top1000000url。

解答：见上题

#### (★★)如果要存储海量的小文件（大小都是几百K~几M），请简述自己的设计方案

将小文件打成har文件存储

将小文件序列化到HDFS中

#### (★★)有两个文本文件，文件中数据按行存放，请编写MapReduce程序，找到两个文件中彼此不相同的行（写出思路即可）

写个MapReduce链 用依赖关系，一共三个MapReduce，第一个处理第一个文件，第二个处理第二个文件，第三个处理前两个的输出结果，第一个MapReduce将文件去重，第二个MapReduce也将文件去重，第三个做wordcount，wordcount为1的结果就是不同的

#### (★)我们开发job时，是否可以去掉reduce阶段。

可以。设置reduce数为0 即可。

#### (★★★)找共同朋友：

思路：例如A，他的朋友是B\C\D\E\F\，那么BC的共同朋友就是A。所以将BC作为key，将A作为value，在map端输出即可！其他的朋友循环处理。

|  |
| --- |
| import java.io.IOException;  2.  import java.util.Set;  3.  import java.util.StringTokenizer;  4.  import java.util.TreeSet;  5.    6.  import org.apache.Hadoop.conf.Configuration;  7.  import org.apache.Hadoop.fs.Path;  8.  import org.apache.Hadoop.io.Text;  9.  import org.apache.Hadoop.MapReduce.Job;  10. import org.apache.Hadoop.MapReduce.Mapper;  11. import org.apache.Hadoop.MapReduce.Reducer;  12. import org.apache.Hadoop.MapReduce.Mapper.Context;  13. import org.apache.Hadoop.MapReduce.lib.input.FileInputFormat;  14. import org.apache.Hadoop.MapReduce.lib.output.FileOutputFormat;  15. import org.apache.Hadoop.util.GenericOptionsParser;  16.  17. public class FindFriend {  www.aboutyun.com  18.  19. public static class ChangeMapper extends Mapper<Object, Text, Text,  Text>{  20. @Override  21. public void map(Object key, Text value, Context context) throws  IOException, InterruptedException {  22. StringTokenizer itr = new StringTokenizer(value.toString());  23. Text owner = new Text();  24. Set<String> set = new TreeSet<String>();  25. owner.set(itr.nextToken());  26. while (itr.hasMoreTokens()) {  27. set.add(itr.nextToken());  28. }  29. String[] friends = new String[set.size()];  30. friends = set.toArray(friends);  31.  32. for(int i=0;i<friends.length;i++){  33. for(int j=i+1;j<friends.length;j++){  34. String outputkey = friends[i]+friends[j];  35. context.write(new Text(outputkey),owner);  36. }  37. }  38. }  39. }  40.  41. public static class FindReducer extends Reducer<Text,Text,Text,Text>  {  42. public void reduce(Text key, Iterable<Text> values,  43. Context context) throws IOException,  InterruptedException {  44. String commonfriends ="";  www.aboutyun.com  45. for (Text val : values) {  46. if(commonfriends == ""){  47. commonfriends = val.toString();  48. }else{  49. commonfriends =  commonfriends+":"+val.toString();  50. }  51. }  52. context.write(key, new  Text(commonfriends));  53. }  54. }  55.  56.  57. public static void main(String[] args) throws IOException,  58. InterruptedException, ClassNotFoundException {  59.  60. Configuration conf = new Configuration();  61. String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();  62. if (otherArgs.length < 2) {  63. System.err.println("args error");  64. System.exit(2);  65. }  66. Job job = new Job(conf, "word count");  67. job.setJarByClass(FindFriend.class);  68. job.setMapperClass(ChangeMapper.class);  69. job.setCombinerClass(FindReducer.class);  70. job.setReducerClass(FindReducer.class);  71. job.setOutputKeyClass(Text.class);  72. job.setOutputValueClass(Text.class);  73. for (int i = 0; i < otherArgs.length - 1; ++i) {  www.aboutyun.com  74. FileInputFormat.addInputPath(job, new Path(otherArgs[i]));  75. }  76. FileOutputFormat.setOutputPath(job,  77. new Path(otherArgs[otherArgs.length - 1]));  78. System.exit(job.waitForCompletion(truE、 ? 0 : 1);  79.  80. }  81.  82. } |

结果：

|  |
| --- |
| 1. AB E:C:D  2. AC E:B  3. AD B:E  4. AE C:B:D  5. BC A:E  6. BD A:E  7. BE C:D:A  8. BF A  9. CD E:A:B  10. CE A:B  11. CF A  12. DE B:A  13. DF A  14. EF A |

#### (★★)hive 的两张表关联，使用 mapreduce 是怎么写的？

方案1：Reduce端join算法实现

实现思路：

①将关联的条件作为map输出的key

②将两表满足join条件的数据并携带数据所来源的文件信息，发往同一个Reduce Task

③在Reduce中进行数据的串联

缺点：

①Join的操作是在Reduce阶段完成

②Reduce阶段处理压力太大

③Map节点的运算负载很低，资源利用率不高

④在Reduce阶段极易产生数据倾斜

方案2：Map端join算法实现

实现思路：

①将小表分发到所有Map。

②Map节点就可以在本地对自己读到的大表的数据进行join。

③输出最终的结果。

适用场景：适用于关联表中有小表的情形。

优点：可以大大提高join操作的并发度，加快处理速度。

#### (★★)假如：Flume 收集到的数据很多个小文件,我需要写 MR 处理时将这些文件合并(是在 MR 中进行优化,不让一个小文件一个 MapReduce)

（1） 默认情况下，TextInputFormat对任务的切片机制是按照文件规划切片，不管有多少个小文件，都会是单独的切片，都会交给一个maptask，这样，如果有大量的小文件

就会产生大量的maptask，处理效率极端底下

（2）优化策略

最好的方法：在数据处理的最前端（预处理、采集），就将小文件合并成大文件，在上传到HDFS做后续的分析

补救措施：如果已经是大量的小文件在HDFS中了，可以使用另一种inputformat来做切片（CombineFileInputformat），它的切片逻辑跟FIleinputformat不同：

它可以将多个小文件从逻辑上规划到一个切片中，这样，多个小文件就可以交给一个maptask了

//如果不设置InputFormat，它默认的用的是TextInputFormat.class

/\*CombineTextInputFormat为系统自带的组件类

\* setMinInputSplitSize 中的2048是表示n个小文件之和不能大于2048

\* setMaxInputSplitSize 中的4096是

当满足setMinInputSplitSize中的2048情况下 在满足n+1个小文件之和不能大于4096

\*/

job.setInputFormatClass(CombineTextInputFormat.class);

CombineTextInputFormat.setMinInputSplitSize(job, 2048);

CombineTextInputFormat.setMaxInputSplitSize(job, 4096);

补充：当n+1大于最大值时候，大文件会被切开，放到n个文件去

#### (★)假设 hadoop 一个 job 产生 100 个 task，并且其中的一个 task 失败了，hadoop会怎样处理？

hadoop本身的一个设计理念就是在普通的pc硬件上构建高可靠性的系统，任何failedtask都不会引起整个job的失败，因为所有失败的任务都会被重新执行（rescheduleexecution），只有当重新执行的次数超过4次，才会把这任务标记为失败，导致整个job的失败。

#### (★)mapreduce 全排序原理。

直接可以想到的方法是“多个maptask”+“一个reducetask”，其中各个maptask对自己负责的数据进行排序，而唯一的reducetask则实现全局排序。

#### (★)编写 MapReduce 作业时，如何做到在 Reduce 阶段，先对 Key 排序，再对 Value排序？

该问题通常称为“二次排序”，最常用的方法是将value放到key中，实现一个组合Key，然后自定义key排序规则（为key实现一个WritableComparable）。

#### (★)MapReduce 怎么实现 Top10？

如：求音乐数据的点击率top10

Mapper端：完成数据抽取后，自定义一个比较器，可以按照点击率降序排序

reducer端输出前10组数据

#### (★★)写代码实现 1G 大小的文本文件，行分隔符为\x01\x02,统计一下该文件中的总行数, 要求注意边界情况的处理

package org.aboutyun;

import java.io.BufferedReader;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.IOException;

public class LineCounter {

public static void main(String[] args) {

try {

BufferedReader reader = new BufferedReader(new FileReader(args[0]));

char[] buffer = new char[4096];

int count;

char last = 0;

long line\_count = 0;

while((count = reader.read(buffer)) >= 0) {

if (count > 0 && line\_count == 0) {

// has something in file, so at least 1 line.

line\_count = 1;

}

for (int i = 0; i < count ; ++i) {

char c = buffer[i];

if (c == 0x02) {

if (i == 0 && last == 0x01) {

// buffer split the 0x01,0x02

++line\_count;

} else if (buffer[i-1] == 0x01) {

// normal one

++line\_count;

}

}

}

// keep the last one

last = buffer[count-1];

}

System.out.println(line\_count);

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

}

}

#### (★)用 mapreduce 实现 sql 语句 select count(x) from a group by b？

map阶段，b列的值作为key输出，1作为value输出，

reduce阶段进行统计

### 4 问答部分（面试官问）

#### (★)Hadoop用的什么版本？

主要有这三大商业发行版: DKhadoop发行版、cloudera发行版、hortonworks发行版。

我们公司用的是CDH，Cloudera的hadoop发行版，完全开源，比Apache hadoop在兼容性，安全性，稳定性上有增强。

#### (★★)你的项目提交到 job 的时候数据量有多大？(面试了三家，都问这个问题)

答：1）回答出数据是什么格式，有没有采用什么压缩，采用了压缩的话，压缩比大概是多少；2）文件大概多大：大概起了多少个map，起了多少个reduce，map阶段读取了多少数据，reduce阶段读取了多少数据，程序大约执行了多久，3）集群什么规模，集群有多少节点，多少内存，多少CPU核数等。把这些点回答进去，而不是给个数字了事。

#### (★)reduce 后输出的数据量有多大？

并不是想知道确切的数据量有多大这个，而是想问你，MR的执行机制，开发完程序，有没有认真评估程序运行效率

1）用于处理redcue任务的资源情况，如果是MRV1的话，分了多少资源给map，多少个reduce

如果是MRV2的话，可以提一下，集群有分了多少内存、CPU给yarn做计算 。

2）结合实际应用场景回答，输入数据有多大，大约多少条记录，做了哪些逻辑操作，输出的时候有多少条记录，执行了多久，reduce执行时候的数据有没有倾斜等

3）再提一下，针对mapReduce做了哪几点优化，速度提升了多久，列举1,2个优化点就可以