# MapReduce相关

# 概念

1. **Job(作业)** : 一个MR程序称为一个Job
2. **MRAppMaster（MR任务的主节点）**: 一个Job在运行时，会先启动一个进程， 这个进程为 MRAppMaster。负责Job中执行状态的监控，容错，和RM申请资 源，提交Task等！
3. **Task**是一个进程！负责某项计算！
4. **Map**是MapReduce程序运行的第一个阶段！

|  |
| --- |
| Map阶段的目的是将输入的数据，进行切分。将一个大数据，切分为若干小部分！切分后，每个部分称为1片(split)，每片数据会交给一个Task（进程）进行计算！  Task负责是Map阶段程序的计算，称为MapTask!  在一个MR程序的Map阶段，会启动N（取决于切片数）个MapTask。每个MapTask是并行运行！ |

1. **Reduce(Reduce阶段)**： Reduce是MapReduce程序运行的第二个阶段(最后一 个阶段)！

|  |
| --- |
| Reduce阶段的目的是将Map阶段，每个MapTask计算后的结果进行合并汇总！得到最终结果！Reduce阶段是可选的！  Task负责是Reduce阶段程序的计算，称为ReduceTask!  一个Job可以通过设置，启动N个ReduceTask，这些ReduceTask也是并行运行！  每个ReduceTask最终都会产生一个结果！ |

# MapReduce中常用的组件

1. Mapper: map阶段核心的处理逻辑
2. Reducer: reduce阶段核心的处理逻辑
3. InputFormat: 输入格式
4. RecordReader: 记录读取器
5. OutPutFormat: 输出格式
6. RecordWriter: 记录写出器
7. Partitioner: 分区器

# MapReduce的运行流程概述

|  |
| --- |
| 需求： 统计/hello目录中每个文件的单词数量，  a-p开头的单词放入到一个结果文件中，  q-z开头的单词放入到一个结果文件中。    例如： /hello/a.txt 200M  hello,hi,hadoop  hive,hadoop,hive,  zoo,spark,wow  zoo,spark,wow  ...  /hello/b.txt 100m  hello,hi,hadoop  zoo,spark,wow  ... |

1. Map阶段(运行MapTask，将一个大的任务切分为若干小任务，处理输出阶段性的结果)

①切片(切分数据)

/hello/a.txt 200M

/hello/b.txt 100m

默认的切分策略是以文件为单位，以文件的块大小(128M)为片大小进行切片！

split0:/hello/a.txt,0-128M

split1: /hello/a.txt,128M-200M

split2: /hello/b.txt,0M-100M

②运行MapTask（进程），每个MapTask负责一片数据

split0:/hello/a.txt,0-128M--------MapTask1

split1: /hello/a.txt,128M-200M--------MapTask2

split2: /hello/b.txt,0M-100M--------MapTask3

③读取数据阶段

在MR中，所有的数据必须封装为key-value

MapTask1,2,3都会初始化一个InputFormat（默认TextInputFormat），每个 InputFormat对象负责创建一个RecordReader(LineRecordReader)对象，

RecordReader负责从每个切片的数据中读取数据，封装为key-value.

LineRecordReader: 将文件中的每一行封装为一个key（offset）-value(当前 行的内容)

④进入Mapper的map()阶段

map()是Map阶段的核心处理逻辑！ 单词统计! map()会循环调用，对输入的每个Key-value都进行处理！

输入：(0,hello,hi,hadoop)

输出：(hello,1),(hi,1),(hadoop,1)

输入：(20,hive,hadoop,hive)

输出：(hive,1),(hadoop,1),(hive,1)

输入：(30,zoo,spark,wow)

输出：(zoo,1),(spark,1),(wow,1)

输入：(40,zoo,spark,wow)

输出：(zoo,1),(spark,1),(wow,1)

⑤目前，我们需要启动两个ReduceTask,生成两个结果文件，需要将MapTask输出的记录进行分区(分组，分类)

在Mapper输出后，调用Partitioner，对Mapper输出的key-value进行分区，分区后也会排序（默认字典顺序排序）

分区规则： a-p开头的单词放入到一个区

q-z开头的单词放入到另一个区

MapTask1:

0号区： (hadoop,1)，(hadoop,1)，(hello,1),(hi,1),(hive,1),(hive,1)

1号区： (spark,1),(spark,1),(wow,1) ，(wow,1),(zoo,1)(zoo,1)

MapTask2:

0号区： 。。。

1号区： ...

MapTask3:

0号区： (hadoop,1),(hello,1),(hi,1),

1号区： (spark,1),(wow,1),(zoo,1)

1. Reduce阶段

①copy

ReduceTask启动后，会启动shuffle线程，从MapTask中拷贝相应分区的数 据！

ReduceTask1: 只负责0号区

将三个MapTask，生成的0号区数据全部拷贝到ReduceTask所在的机器！

(hadoop,1)，(hadoop,1)，(hello,1),(hi,1),(hive,1),(hive,1)

(hadoop,1),(hello,1),(hi,1),

ReduceTask2: 只负责1号区

将三个MapTask，生成的1号区数据全部拷贝到ReduceTask所在的机器！

(spark,1),(spark,1),(wow,1) ，(wow,1),(zoo,1)(zoo,1)

(spark,1),(wow,1),(zoo,1)

②sort

ReduceTask1: 只负责0号区进行排序：

(hadoop,1),(hadoop,1),(hadoop,1),(hello,1),(hello,1),(hi,1),(hi,1),(hive,1),(hive, 1)

ReduceTask2: 只负责1号区进行排序：

(spark,1),(spark,1),(spark,1),(wow,1) ，(wow,1),(wow,1),(zoo,1),(zoo,1)(zoo,1)

③reduce

ReduceTask1---->Reducer----->reduce(一次读入一组数据)

何为一组数据： key相同的为一组数据

输入： (hadoop,1)，(hadoop,1)，(hadoop,1)

输出： (hadoop,3)

输入： (hello,1),(hello,1)

输出： (hello,2)

输入： (hi,1),(hi,1)

输出： (hi,2)

输入：(hive,1),(hive,1)

输出： （hive,2）

ReduceTask2---->Reducer----->reduce(一次读入一组数据)

输入： (spark,1),(spark,1),(spark,1)

输出： (spark,3)

输入： (wow,1) ，(wow,1),(wow,1)

输出： (wow,3)

输入：(zoo,1),(zoo,1)(zoo,1)

输出： (zoo,3)

④调用OutPutFormat中的RecordWriter将Reducer输出的记录写出

ReduceTask1---->OutPutFormat（默认TextOutPutFormat） ------>RecordWriter（LineRecoreWriter）

LineRecoreWriter将一个key-value以一行写出，key和alue之间使用\t分割

在输出目录中，生成文件part-r-0000

hadoop 3

hello 2

hi 2

hive 2

ReduceTask2---->OutPutFormat（默认TextOutPutFormat） ------>RecordWriter（LineRecoreWriter）

LineRecoreWriter将一个key-value以一行写出，key和alue之间使用\t分割

在输出目录中，生成文件part-r-0001

spark 3

wow 3

zoo 3

# MR总结

1. Map阶段(MapTask)： 切片(Split)-----读取数据(Read)-------交给Mapper处理(Map)------分区和排序(sort)
2. Reduce阶段(ReduceTask): 拷贝数据(copy)------排序(sort)-----合并(reduce)-----写出(write)

# 调节块大小

默认的片大小就是文件的块大小。

文件的块大小是128M，默认没片128M

调节块大小＞片大小：mapreduce.input.fileinputformat.split.minsize ＞128M

调节块大小＜片大小：mapreduce.input.fileinputformat.split.maxsize＜128M

SPLIT\_MAXSIZE = "mapreduce.input.fileinputformat.split.maxsize";

SPLIT\_MINSIZE = "mapreduce.input.fileinputformat.split.minsize";

理论上：文件的数据量一定，块越大，切片的数量就越小。

# 切片

## 1. FileInputFormat的切片策略(默认)

|  |
| --- |
| **public** List<InputSplit> getSplits(JobContext job) **throws** IOException {  StopWatch sw = **new** StopWatch().start();  // minSize从mapreduce.input.fileinputformat.split.minsize和1之间对比，取最大值  **long** minSize = Math.*max*(getFormatMinSplitSize(), *getMinSplitSize*(job));  // 读取mapreduce.input.fileinputformat.split.maxsize，如果没有设置使用Long.MaxValue作为默认值  **long** maxSize = *getMaxSplitSize*(job);  // generate splits  List<InputSplit> splits = **new** ArrayList<InputSplit>();  // 获取当前Job输入目录中所有文件的状态(元数据)  List<FileStatus> files = listStatus(job);  // 以文件为单位进行切片  **for** (FileStatus file: files) {  Path path = file.getPath();  **long** length = file.getLen();  **if** (length != 0) {  BlockLocation[] blkLocations;  **if** (file **instanceof** LocatedFileStatus) {  blkLocations = ((LocatedFileStatus) file).getBlockLocations();  } **else** {  FileSystem fs = path.getFileSystem(job.getConfiguration());  blkLocations = fs.getFileBlockLocations(file, 0, length);  }  // 判断当前文件是否可切，如果可切，切片  **if** (isSplitable(job, path)) {  **long** blockSize = file.getBlockSize();  **long** splitSize = computeSplitSize(blockSize, minSize, maxSize);  // 声明待切部分数据的余量  **long** bytesRemaining = length;  // 如果 待切部分 / 片大小 > 1.1，先切去一片，再判断  **while** (((**double**) bytesRemaining)/splitSize > ***SPLIT\_SLOP***) {  **int** blkIndex = getBlockIndex(blkLocations, length-bytesRemaining);  splits.add(makeSplit(path, length-bytesRemaining, splitSize,  blkLocations[blkIndex].getHosts(),  blkLocations[blkIndex].getCachedHosts()));  bytesRemaining -= splitSize;  }  // 否则，将剩余部分整个作为1片。 最后一片有可能超过片大小，但是不超过其1.1倍  **if** (bytesRemaining != 0) {  **int** blkIndex = getBlockIndex(blkLocations, length-bytesRemaining);  splits.add(makeSplit(path, length-bytesRemaining, bytesRemaining,  blkLocations[blkIndex].getHosts(),  blkLocations[blkIndex].getCachedHosts()));  }  } **else** { // not splitable  // 如果不可切，整个文件作为1片！  splits.add(makeSplit(path, 0, length, blkLocations[0].getHosts(),  blkLocations[0].getCachedHosts()));  }  } **else** {  //Create empty hosts array for zero length files  // 如果文件是个空文件，创建一个切片对象，这个切片从当前文件的0offset起，向后读取0个字节  splits.add(makeSplit(path, 0, length, **new** String[0]));  }  }  // Save the number of input files for metrics/loadgen  job.getConfiguration().setLong(***NUM\_INPUT\_FILES***, files.size());  sw.stop();  **if** (***LOG***.isDebugEnabled()) {  ***LOG***.debug("Total # of splits generated by getSplits: " + splits.size()  + ", TimeTaken: " + sw.now(TimeUnit.***MILLISECONDS***));  }  **return** splits;  } |

总结： ①获取当前输入目录中所有的文件

②以文件为单位切片，如果文件为空文件，默认创建一个空的切片

③如果文件不为空，尝试判断文件是否可切(不是压缩文件，都可切)

④如果文件不可切，整个文件作为1片

⑤如果文件可切，先获取片大小(默认等于块大小)，

循环判断 待切部分/ 片大小 > 1.1倍，如果大于先切去一片，再判断…

⑥剩余部分整个作为1片

## 2. 从Job的配置中获取参数

job.getConfiguration().getLong(SPLIT\_MINSIZE, 1L)

指从配置文件中获取名称为SPLIT\_MINSIZE的参数，如果获取到，将参数的值返回，否则就使用默认值1L！

设置Job的参数：

|  |
| --- |
| conf.setClass(***INPUT\_FORMAT\_CLASS\_ATTR***, cls,  InputFormat.**class**); |

要为INPUT\_FORMAT\_CLASS\_ATTR设置参数为cls，cls必须是InputFormat.class的子类！

## 3. TextInputFormat判断文件是否可切

|  |
| --- |
| @Override  **protected** **boolean** isSplitable(JobContext context, Path file) {  // 根据文件的后缀名获取文件使用的相关的压缩格式  **final** CompressionCodec codec =  **new** CompressionCodecFactory(context.getConfiguration()).getCodec(file);  // 如果文件不是一个压缩类型的文件，默认都可以切片  **if** (**null** == codec) {  **return** **true**;  }  //否则判断是否是一个可以切片的压缩格式，默认只有Bzip2压缩格式可切片  **return** codec **instanceof** SplittableCompressionCodec;  } |

## 4.片大小的计算

long splitSize = computeSplitSize(blockSize, minSize, maxSize);

|  |
| --- |
| **protected** **long** computeSplitSize(**long** blockSize, **long** minSize,  **long** maxSize) {  **return** Math.*max*(minSize, Math.*min*(maxSize, blockSize));  } |

blockSize： 块大小

minSize: minSize从mapreduce.input.fileinputformat.split.minsize和1之间对比，取最大值

maxSize: 读取mapreduce.input.fileinputformat.split.maxsize，如果没有设置使用Long.MaxValue作为默认值

默认的片大小就是文件的块大小！

文件的块大小默认为128M，默认每片就是128M！

调节片大小 > 块大小：配置 mapreduce.input.fileinputformat.split.minsize > 128M

调节片大小 < 块大小：配置 mapreduce.input.fileinputformat.split.maxsize < 128M

理论上来说：如果文件的数据量是一定的话，片越大，切片数量少，启动的MapTask少，Map阶段运算慢！

片越小，切片数量多，启动的MapTask多，Map阶段运算快！

## 5.片和块的关系

片(InputSplit)：在计算MR程序时，才会切片。片在运行程序时，临时将文件从逻辑上划分为若干部分！

使用的输入格式不同，切片的方式不同，切片的数量也不同！

每片的数据最终也是以块的形式存储在HDFS！

块(Block)： 在向HDFS写文件时，文件中的内容以块为单位存储！块是实际的物理存在！

建议： 片大小最好等于块大小！

将片大小设置和块大小一致，可以最大限度减少因为切片带来的磁盘IO和网络IO!

原因： MR计算框架速度慢的原因在于在执行MR时，会发生频繁的磁盘IO和网络IO!

优化MR ： 减少磁盘IO和网络IO！

# 二、常见的输入格式

## 1. TextInputFormat

TextInputFormat常用于输入目录中全部是文本文件！

切片： 默认的切片策略

RecordReader: LineRecordReader,一次处理一行，将一行内容的偏移量作为key，一行内容作为value!

LongWritable key

Text value

## 2. NlineInputFormat

切片： 读取配置中mapreduce.input.lineinputformat.linespermap，默认为1，以文件为单位，切片每此参数行作为1片！

RecordReader: LineRecordReader,一次处理一行，将一行内容的偏移量作为key，一行内容作为value!

LongWritable key

Text value

## 3. KeyValueTextInputFormat

作用： 针对文本文件！使用分割字符，将每一行分割为key和value!

如果没有找到分隔符，当前行的内容作为key，value为空串!

默认分隔符为\t，可以通过参数mapreduce.input.keyvaluelinerecordreader.key.value.separator指定！

切片：默认的切片策略

RR ： KeyValueLineRecordReader

Text key:

Text value

## 4. ConbineTextInputFormat

作用： 改变了传统的切片方式！将多个小文件，划分到一个切片中！

适合小文件过多的场景！

RecordReader: LineRecordReader,一次处理一行，将一行内容的偏移量作为key，一行内容作为value!

LongWritable key

Text value

切片： 先确定片的最大值maxSize，maxSize通过参数mapreduce.input.fileinputformat.split.maxsize设置！

流程： a. 以文件为单位，将每个文件划分为若干part

①判断文件的待切部分的大小 <= maxSize,整个待切部分作为1part

②maxsize < 文件的待切部分的大小 <= 2\* maxSize,将整个待切部分均分为2part

③文件的待切部分的大小 > 2\* maxSize,先切去maxSize大小，作为1部分，剩余待切部分继续判断！

举例： maxSize=2048

a.txt 4.38KB

part1(a.txt,0,2048)

part2(a.txt,2048,1219)

part3(a.txt.3xxx,1219)

b.txt 4.18KB

part4(b.txt,0,2048)

part5(b.txt,2048,1116)

part6(b.txt,3xxx,1116)

c.txt 2.71kb

part7(c.txt,0 ,13xxx)

part8(c.txt,13 ,27xx)

d.txt 5.04kb

part9(d.txt,0,2048)

part10(d.txt,2048,1519)

part11(a.txt.3xxx,1519)

b. 将之前切分的若干part进行累加，累加后一旦累加的大小超过 maxSize，这些作为1片！

# 三、关键设置

## 1.如何设置MapTask的数量

MapTask的数量，人为设置是无效的！只能通过切片方式来设置！

MapTask只取决于切片数！

# 四、Job提交流程阶段总结

## 1.准备阶段

运行Job.waitForCompletion(),先使用JobSubmitter提交Job，在提交之前，会在Job的作业目录中生成以下信息：

job.split: 当前Job的切片信息，有几个切片对象

job.splitmetainfo: 切片对象的属性信息

job.xml: job所有的属性配置

## 2. 提交阶段

本地模式： LocalJobRunner进行提交！

创建一个LocalJobRunner.Job()

Job.start() //作业启动

Map阶段： 采用线程池提交多个MapTaskRunable线程！

每个MapTaskRunable线程上，实例化一个MapTask对象！

每个MapTask对象，实例化一个Mapper!

Mapper.run()

线程运行结束，会在线程的作业目录中生成 file.out文件，保存MapTask输出的所有的key-value!

阶段定义： 如果有reduceTask，MapTask运行期间，分为 map(67%)---sort(33%)

没有ReduceTask，MapTask运行期间，分为 map(100%)

map: 使用RR将切片中的数据读入到Mapper.map() -------context.write(key,value)

Reduce阶段： 采用线程池提交多个ReduceTaskRunable线程！

每个ReduceTaskRunable线程上，实例化一个ReduceTask对象！

每个ReduceTask对象，实例化一个Reduce!

reducer.run()

线程运行结束，会在输出目录中生成part-r-000x文件，保存ReduceTask输出的所有的key-value!

阶段定义： copy： 使用shuffle线程拷贝MapTask指定分区的数据！

sort: 将拷贝的所有的分区的数据汇总后，排序

reduce : 对排好序的数据，进行合并！

YARN上运行： 参考课件第六章！