# 一、切片

## 1. FileInputFormat的切片策略(默认)【TextInputFormat中没有】

|  |
| --- |
| **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  // 配置文件中默认配置的是0所以结果是1  **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，配置文件默认没有，所以是MaxValue  **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(); // win local 32M  **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;  } |

默认的切片策略总结：

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)， // computeSplitSize

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

Key是偏移量 value是每一行的记录 ,每一行后面多一个\n字符 偏移量从0开始

keyin:0----keyout:zhege shi

keyin:10----keyout:ceshi hello world

keyin:28----keyout:ban ban ban

keyin:40----keyout:ben zhe he shi xiao cang

keyin:65----keyout:xiao cang ni hao ma

keyin:85----keyout:zhou si

## 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！ 两者相等

# 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：

【eclipse工作空间磁盘/tmp….】

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Reducer

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！

# 五、MapTask的shuffle细节

## 1.记录输出收集器的赋值

|  |
| --- |
| **if** (job.getNumReduceTasks() == 0) {  output =  **new** NewDirectOutputCollector(taskContext, job, umbilical, reporter);  } **else** {  output = **new** NewOutputCollector(taskContext, job, umbilical, reporter);  } |

如果没有Reduce阶段，使用直接的记录收集器，它不会读数据进行排序！按照Mapper输出的顺序输出！

如果有Reduce阶段，使用NewOutputCollector来收集记录！

## 2. MapTask记录输出收集器的初始化

|  |
| --- |
| NewOutputCollector(org.apache.hadoop.mapreduce.JobContext jobContext,  JobConf job,  TaskUmbilicalProtocol umbilical,  TaskReporter reporter  ) **throws** IOException, ClassNotFoundException {  //真正干活的收集器，缓存区对象，这个缓冲区对其会对收集的记录进行排序  collector = createSortingCollector(job, reporter);  // 获取当前job所有的reduceTask数量，以此数量作为总的分区数，默认JobReduceTask的数量为1  partitions = jobContext.getNumReduceTasks();  // 为MapTask使用的分区器进行赋值  **if** (partitions > 1) {  partitioner = (org.apache.hadoop.mapreduce.Partitioner<K,V>)  ReflectionUtils.*newInstance*(jobContext.getPartitionerClass(), job);  } **else** {  // 默认使用此分区器进行分区，这个分区器将所有的key-value都分到0号区  partitioner = **new** org.apache.hadoop.mapreduce.Partitioner<K,V>() {  @Override  **public** **int** getPartition(K key, V value, **int** numPartitions) {  **return** partitions - 1;  }  };  }  } |

## 3.获取Partitioner

|  |
| --- |
| @SuppressWarnings("unchecked")  **public** Class<? **extends** Partitioner<?,?>> getPartitionerClass()  **throws** ClassNotFoundException {  **return** (Class<? **extends** Partitioner<?,?>>)  conf.getClass(***PARTITIONER\_CLASS\_ATTR***, HashPartitioner.**class**);  } |

从配置中获取mapreduce.job.partitioner.class参数，如果没有设置，就使用HashPartitioner作为分区器！

|  |
| --- |
| **public** **class** HashPartitioner<K, V> **extends** Partitioner<K, V> {  /\*\* Use {@link Object#hashCode()} to partition. \*/  **public** **int** getPartition(K key, V value,  **int** numReduceTasks) {  **return** (key.hashCode() & Integer.***MAX\_VALUE***) % numReduceTasks;  }  } |

## 4.缓冲区对象的初始化

如果有Reduce阶段，在MapTask中使用MapOutPutBuffer作为缓冲区的实现类！

|  |
| --- |
| **public** **void** init(MapOutputCollector.Context context  ) **throws** IOException, ClassNotFoundException {    //sanity checks  // 获取当前缓冲区溢写的阀值，默认读取配置中mapreduce.map.sort.spill.percent  没有配置，默认为0.8  **final** **float** spillper = job.getFloat(JobContext.***MAP\_SORT\_SPILL\_PERCENT***, (**float**)0.8);  //获取缓冲区的初始大小，读取mapreduce.task.io.sort.mb，如果没有配置，默认为100  **final** **int** sortmb = job.getInt(JobContext.***IO\_SORT\_MB***, 100);  indexCacheMemoryLimit = job.getInt(JobContext.***INDEX\_CACHE\_MEMORY\_LIMIT***,  ***INDEX\_CACHE\_MEMORY\_LIMIT\_DEFAULT***);  **if** (spillper > (**float**)1.0 || spillper <= (**float**)0.0) {  **throw** **new** IOException("Invalid \"" + JobContext.***MAP\_SORT\_SPILL\_PERCENT*** +  "\": " + spillper);  }  **if** ((sortmb & 0x7FF) != sortmb) {  **throw** **new** IOException(  "Invalid \"" + JobContext.***IO\_SORT\_MB*** + "\": " + sortmb);  }  // 实例化排序器，默认使用快排，只排索引  sorter = ReflectionUtils.*newInstance*(job.getClass("map.sort.class",  QuickSort.**class**, IndexedSorter.**class**), job);    // k/v serialization  comparator = job.getOutputKeyComparator();  // 根据Mapper输出的Key-value类型，获取序列化器  // 如果Mapper的输出Key-value实现了Wriable接口，Hadoop自动提供序列化器  // 如果Mapper输出的key-value没有实现Wriable接口，需要自定提供序列化器，设置到Job中  keyClass = (Class<K>)job.getMapOutputKeyClass();  valClass = (Class<V>)job.getMapOutputValueClass();  serializationFactory = **new** SerializationFactory(job);  keySerializer = serializationFactory.getSerializer(keyClass);  keySerializer.open(bb);  valSerializer = serializationFactory.getSerializer(valClass);  valSerializer.open(bb);  //MapTask输出的记录可以使用压缩格式，到ReduceTask时，再解压缩  // 压缩可以节省磁盘IO和网络IO，提供MR的运行效率  // compression  **if** (job.getCompressMapOutput()) {  Class<? **extends** CompressionCodec> codecClass =  job.getMapOutputCompressorClass(DefaultCodec.**class**);  codec = ReflectionUtils.*newInstance*(codecClass, job);  } **else** {  codec = **null**;  }  // combiner  **final** Counters.Counter combineInputCounter =  reporter.getCounter(TaskCounter.***COMBINE\_INPUT\_RECORDS***);  combinerRunner = CombinerRunner.*create*(job, getTaskID(),  combineInputCounter,  reporter, **null**);  **if** (combinerRunner != **null**) {  **final** Counters.Counter combineOutputCounter =  reporter.getCounter(TaskCounter.***COMBINE\_OUTPUT\_RECORDS***);  combineCollector= **new** CombineOutputCollector<K,V>(combineOutputCounter, reporter, job);  } **else** {  combineCollector = **null**;  }  spillInProgress = **false**;  minSpillsForCombine = job.getInt(JobContext.***MAP\_COMBINE\_MIN\_SPILLS***, 3);  spillThread.setDaemon(**true**);  spillThread.setName("SpillThread");  spillLock.lock();  **try** {  spillThread.start();  **while** (!spillThreadRunning) {  spillDone.await();  }  } **catch** (InterruptedException e) {  **throw** **new** IOException("Spill thread failed to initialize", e);  } **finally** {  spillLock.unlock();  }  **if** (sortSpillException != **null**) {  **throw** **new** IOException("Spill thread failed to initialize",  sortSpillException);  }  } |

## 5. 获取Mapper输出的key的比较器

|  |
| --- |
| **public** RawComparator getOutputKeyComparator() {  // 从配置中获取mapreduce.job.output.key.comparator.class的值，必须是RawComparator类型，  如果没有配置，默认为null  Class<? **extends** RawComparator> theClass = getClass(  JobContext.***KEY\_COMPARATOR***, **null**, RawComparator.**class**);  // 一旦用户配置了此参数，实例化一个用户自定义的比较器实例  **if** (theClass != **null**)  **return** ReflectionUtils.*newInstance*(theClass, **this**);  //用户没有配置，判断Mapper输出的key的类型是否是WritableComparable的子类，如果不是，就抛异常，如果是，系统会自动为我们提供一个key的比较器  **return** WritableComparator.*get*(getMapOutputKeyClass().asSubclass(WritableComparable.**class**), **this**);  } |

如何自定义比较器：

1. 自定义类，这个类必须是RawComparator类型，通过设置
2. mapreduce.job.output.key.comparator.class=自定义的类的类型。

自定义类时，可以继承WriableComparator，也可以实现RawCompartor!

调用方法时，先调用RawCompartor. compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2),再调用RawCompartor.compare()

或

②定义Mapper输出的key,让key实现WritableComparable,实现CompareTo()

## 6.Combiner

Combiner实际上本质是一个Reducer类！

Conbiner只有在设置了之后，才会运行！

Combiner和Reducer的区别：

map----sort---copy---sort---reduce

Reducer是在reduce阶段调用！

Combiner是在shuffle阶段调用！

本质都是Reducer类，作用都是对有相同key的key-value进行合并！

意义： 在shuffle阶段对相同key的key-value进行提前合并，可以减少磁盘IO和网络IO！

使用条件： Combiner用在+，-操作的场景，不能用在\*,/操作的场景！

使用Combiner必须保证不能影响处理逻辑和结果！

Combiner可以优化MR程序，提高运行效率！

Combiner既有可能在MapTask端调用：

①每次溢写前会调用Combiner对溢写的数据进行局部合并

②在merge时，如果溢写的片段数>=3，如果设置了Combiner，Combiner会再次对

数据进行Combine！

Combiner既有可能在ReduceTask端调用：

③shuffle线程拷贝多个MapTask同一分区的数据，拷贝后执行merge和sort,

如果数据量过大，需要将部分数据先合并排序后，溢写到磁盘！

如果设置了Combiner，Combiner会再次运行！

## 7.总结

①分区

a)总的分区数取决于reduceTask的数量！

一个Job要启动几个reduceTask，取决于期望产生几个分区，每个分区最后都会生成一个结果文件！

b)Partionner的确定

reduceTask>1，尝试获取用户设置的Partionner，如果没有设置使用HashPartitoner

reduceTask<=1,系统默认提供一个Partionner，它会将所有记录都分到0号区

②排序

每次溢写前，使用快速排序

最后merge时，使用归并排序

③比较器

排序时，根据比较器比较的结果进行排序！

a)如果用户自定义了比较器，MR就使用用户自定义的比较器(RawComparator类型)

b)如果用户没有自定义，那么Mapper输出的Key需要实现WriableComparable接口

系统会自动提供比较器

总结： 不管是自己提供比较器还是实现WriableComparable接口，最后在比较时，都是调用

自己实现的CompareTo()

④Combiner

Combiner在shuffle阶段运行！

a)每次溢写前会调用Combiner对溢写的数据进行局部合并

b)在merge时，如果溢写的片段数>=3，如果设置了Combiner，Combiner会再次对

数据进行Combine！

⑤执行流程

a)Partitioner计算分区

b)满足溢写条件，所有数据进行排序，排序时用比较器对比key

每次溢写前的排序，默认使用快排

如果设置Combiner，在溢写前，排好序的结果会先被Combiner进行combine

再溢写

c) b过程会发生N次

d) 所有的溢写片段需要merge为一个总的文件

合并时，使用归并排序，对key进行排序！

如果溢写片段数量超过3，在溢写成一个最终的文件时，Combiner再次调用，

执行Combine，combine后再溢写！

# 六、ReduceTask的shuffle细节

## 1.获取分组比较器

|  |
| --- |
| **public** RawComparator getOutputValueGroupingComparator() {  // 从配置中读取mapreduce.job.output.group.comparator.class，  Class<? **extends** RawComparator> theClass = getClass(  JobContext.***GROUP\_COMPARATOR\_CLASS***, **null**, RawComparator.**class**);  //如果没有设置，默认使用MapTask对key排序时，key的比较器  **if** (theClass == **null**) {  **return** getOutputKeyComparator();  }  // 否则用户设置了，就使用用户自定义的比较器  **return** ReflectionUtils.*newInstance*(theClass, **this**);  } |