# Task

抽象方法：

1. **public** **abstract** **void** run(JobConf job, TaskUmbilicalProtocol umbilical)

负责调用用户定义的map，reduce等任务

1. **public** **abstract** TaskRunner createRunner(TaskTracker tracker, TaskTracker.TaskInProgress tip, TaskTracker.RunningJob rjob)

属性：

1. **private** String jobFile; // job configuration file
2. **private** String user; // user running the job
3. **private** TaskAttemptID taskId; // unique, includes job id
4. **private** **int** partition; // id within job
5. TaskStatus taskStatus; // current status of the task
6. **protected** JobStatus.State jobRunStateForCleanup;
7. **protected** **boolean** jobCleanup = **false**;

用于判断是否为job的cleanup任务

1. **protected** **boolean** jobSetup = **false**;

用于判断是否为job的setup任务

1. **protected** **boolean** taskCleanup = **false**;
2. **private** SortedRanges skipRanges = **new** SortedRanges();
3. **private** **boolean** skipping = **false**;
4. **private** **boolean** writeSkipRecs = **true**;
5. **private** **volatile** **long** currentRecStartIndex;
6. **private** Iterator<Long> currentRecIndexIterator = skipRanges.skipRangeIterator();
7. **private** ResourceCalculatorPlugin resourceCalculator = **null**;
8. **private** **long** initCpuCumulativeTime = 0;
9. **protected** JobConf conf;
10. **protected** MapOutputFile mapOutputFile = **new** MapOutputFile();
11. **protected** LocalDirAllocator lDirAlloc;
12. **private** **final** **static** **int** *MAX\_RETRIES* = 10;
13. **protected** JobContext jobContext;
14. **protected** TaskAttemptContext taskContext;
15. **protected** org.apache.hadoop.mapreduce.OutputFormat<?,?> outputFormat;
16. **protected** org.apache.hadoop.mapreduce.OutputCommitter committer;
17. **protected** **final** Counters.Counter spilledRecordsCounter;
18. **private** **int** numSlotsRequired;
19. **private** String pidFile = "";
20. **protected** TaskUmbilicalProtocol umbilical;
21. **protected** SecretKey tokenSecret;
22. **protected** JvmContext jvmContext;

用于执行task的属性：

1. **private** **transient** Progress taskProgress = **new** Progress();
2. **private** **transient** Counters counters = **new** Counters();
3. **private** AtomicBoolean taskDone = **new** AtomicBoolean(**false**);

构造函数中初始化taskStatus时会设置为TaskStatus.State.*UNASSIGNED*

方法：

1. **protected** **void** reportFatalError(TaskAttemptID id, Throwable throwable, String logMsg)
   1. 先在自己的log中打印错误
   2. 然后提交到TT：umbilical.fatalError(id, cause, jvmContext);（TT接到错误报告后，会记录到tip的diagnosticInfo信息中，心跳时会传递给JT，并最终返回给用户，然后清理这个tip：purgeTask(tip, **true**)）
2. **boolean** isJobAbortTask()

如果jobCleanup为true，且jobRunStateForCleanup为*KILLED*或*FAILED*，则返回true（即，如果一个task被标记为cleanup，且最终状态为killed或failed，则为abort task）

1. **boolean** isMapOrReduce()

jobSetup，jobCleanup和taskCleanup都为false，则返回true

1. **public** **void** initialize(JobConf job, JobID id, Reporter reporter, **boolean** useNewApi)
   1. 初始化jobContext和taskContext
   2. 修改状态为*RUNNING*
   3. 初始化outputFormat和committer
   4. 初始化resourceCalculator
2. **public** **void** done(TaskUmbilicalProtocol umbilical, TaskReporter reporter)
   1. 更新updateCounters();
   2. 循环调用umbilical.commitPending(taskId, taskStatus, jvmContext);直到正确的提交了任务进度
   3. 调用commit(umbilical, reporter, committer);（见6，此处会阻塞至提交成功）
   4. 设置taskDone.set(**true**);
   5. 停止线程reporter.stopCommunicationThread();
   6. 最后一次更新状态sendLastUpdate(umbilical);
   7. 通知TT任务执行完毕sendDone(umbilical);
3. **private** **void** commit(TaskUmbilicalProtocol umbilical, TaskReporter reporter, org.apache.hadoop.mapreduce.OutputCommitter committer)
   1. 循环调用umbilical.canCommit(taskId, jvmContext)直到其返回true
   2. 提交任务committer.commitTask(taskContext);
4. **protected** **void** runJobCleanupTask(TaskUmbilicalProtocol umbilical, TaskReporter reporter)
   1. 设置setPhase(TaskStatus.Phase.*CLEANUP*);并发送状态到啊TT：statusUpdate(umbilical);
   2. 如果状态为*FAILED*和*KILLED*，则调用committer.abortJob(jobContext, jobRunStateForCleanup);
   3. 如果状态为*SUCCEEDED*，则调用committer.commitJob(jobContext);
   4. 调用done(umbilical, reporter);
5. **protected** **void** runJobSetupTask(TaskUmbilicalProtocol umbilical, TaskReporter reporter )
   1. 调用committer.setupJob(jobContext);
   2. 结束：done(umbilical, reporter);

抽象方法：

1. **public** **abstract** **void** run(JobConf job, TaskUmbilicalProtocol umbilical)
2. **public** **abstract** TaskRunner createRunner(TaskTracker tracker, TaskTracker.TaskInProgress tip, TaskTracker.RunningJob rjob )

## TaskReporter

实现了Runnable，负责和主进程通信，报告task的执行状态，和TT通信

属性：

1. **private** TaskUmbilicalProtocol umbilical;
2. **private** InputSplit split = **null**;
3. **private** Progress taskProgress;
4. **private** JvmContext jvmContext;
5. **private** Thread pingThread = **null**;
6. **private** **static** **final** **int** *PROGRESS\_STATUS\_LEN\_LIMIT* = 512;
7. **private** **boolean** done = **true**;
8. **private** Object lock = **new** Object();

**public** **void** run()

和TT通信的主循环：

1. 如果sendProgress为true：
   1. 调用updateCounters();
   2. 更新taskStatus.statusUpdate(taskProgress.get(),taskProgress.toString(),counters);
   3. 发送taskFound = umbilical.statusUpdate(taskId, taskStatus, jvmContext);
   4. 清空taskStatus.clearStatus();
2. 否则，只需要发送ping：taskFound = umbilical.ping(taskId, jvmContext);
3. 如果taskFound为false，即TT没有保存本task的信息，则调用resetDoneFlag();然后退出System.*exit*(66);

## CombinerRunner<K,V>

属性：

1. **protected** **final** Counters.Counter inputCounter;
2. **protected** **final** JobConf job;
3. **protected** **final** TaskReporter reporter;

抽象方法：

**abstract** **void** combine(RawKeyValueIterator iterator, OutputCollector<K,V> collector)

### OldCombinerRunner<K,V>

### NewCombinerRunner<K,V>

上面两个是新旧两种api，其实最后都是调用Reducer的reduce方法

# MapTask

继承自Task，处理map任务

实例块中会执行：

setPhase(TaskStatus.Phase.*MAP*);

属性：

**private** TaskSplitIndex splitMetaInfo = **new** TaskSplitIndex();

方法：

1. **public** **void** run(**final** JobConf job, **final** TaskUmbilicalProtocol umbilical)
   1. 初始化TaskReporter reporter = **new** TaskReporter(getProgress(), umbilical, jvmContext);并启动reporter.startCommunicationThread();
   2. 初始化initialize(job, getJobID(), reporter, useNewApi);
   3. 如果不是正式的map任务：
      1. 如果jobCleanup，则调用runJobCleanupTask(umbilical, reporter);
      2. 如果jobSetup则调用runJobSetupTask(umbilical, reporter);
      3. 如果taskCleanup，则调用runTaskCleanupTask(umbilical, reporter);
      4. 上面三个都是调用父类Task的方法，调用后直接return
   4. 调用runNewMapper(job, splitMetaInfo, umbilical, reporter);
   5. 全部运行结束后，调用done(umbilical, reporter);
2. **void** runNewMapper(**final** JobConf job, **final** TaskSplitIndex splitIndex, **final** TaskUmbilicalProtocol umbilical, TaskReporter reporter)
   1. 初始化TaskAttemptContext taskContext
   2. 从配置中读取Mapper<INKEY,INVALUE,OUTKEY,OUTVALUE> mapper
   3. 从配置中读取InputFormat<INKEY,INVALUE> inputFormat
   4. 构建InputSplit split = getSplitDetails(**new** Path(splitIndex.getSplitLocation()),splitIndex.getStartOffset());
   5. 新建RecordReader<INKEY,INVALUE> input = **new** NewTrackingRecordReader<INKEY,INVALUE>
   6. 如果reduce数目为0，则初始化output为NewDirectOutputCollector，否则初始化output为NewOutputCollector
   7. 使用反射初始化Mapper.Context mapperContext
   8. 初始化input：input.initialize(split, mapperContext);
   9. 开始执行map任务：mapper.run(mapperContext);
   10. 最后关闭input和output

## TrackedRecordReader<K, V>

实现了RecordReader<K,V>

/\*\*

\* This class wraps the user's record reader to update the counters and

\* progress as records are read.

\*/

## SkippingRecordReader<K, V>

继承自TrackedRecordReader<K,V>

/\*\*

\* This class skips the records based on the failed ranges from previous

\* attempts.

\*/

属性：

1. **private** SkipRangeIterator skipIt;
2. **private** SequenceFile.Writer skipWriter;
3. **private** **boolean** toWriteSkipRecs;
4. **private** TaskUmbilicalProtocol umbilical;
5. **private** Counters.Counter skipRecCounter;
6. **private** **long** recIndex = -1;

方法：

1. **public** **synchronized** **boolean** next(K key, V value)
   1. 调用**boolean** ret = moveToNext(key, value);（会调用父类的同名方法）
   2. 获取**long** nextRecIndex = skipIt.next();（即要跳过的index）
   3. 循环**while**(recIndex<nextRecIndex && ret)（直到跳过了nextRecIndex才停止）：
      1. 如果toWriteSkipRecs为true，则调用writeSkippedRec(key, value);
      2. 调用ret = moveToNext(key, value);移动到下一个位置
      3. skip++;
   4. 如果skip>0 && skipIt.skippedAllRanges() && skipWriter!=**null**，则表示已经跳过了所有的坏点，可以关闭skipWriter.close();
   5. 报告：reportNextRecordRange(umbilical, recIndex);

## NewTrackingRecordReader<K,V>

和TrackedRecordReader对应，新的api，继承自org.apache.hadoop.mapreduce.RecordReader<K,V>

构造函数中需要注意：

初始化**this**.real = inputFormat.createRecordReader(split, taskContext);这个real会在后面调用initialize时初始化（eg：如果input是TextInputFormat，则会返回LineRecordReader）

其他的操作都托管给real，即RecordReader

## NewDirectOutputCollector

继承自org.apache.hadoop.mapreduce.RecordWriter<K,V>，简单的包装了RecordWriter

## NewOutputCollector<K,V>

继承自org.apache.hadoop.mapreduce.RecordWriter<K,V>

属性：

1. **private** **final** MapOutputCollector<K,V> collector;
2. **private** **final** org.apache.hadoop.mapreduce.Partitioner<K,V> partitioner;

如果reduce数目大于1，则会初始化为定义的类，默认为HashPartitioner

1. **private** **final** **int** partitions;

partitions = jobContext.getNumReduceTasks();（即reduce的数目）

其他操作都是简单的包装MapOutputCollector（实际使用MapOutputBuffer）

## MapOutputBuffer

实现了IndexedSortable和MapOutputCollector<K, V>

**实现原理解析：**

1. 开始时把map处理的结果都放在内存中，当需要spill到磁盘时，通知SpillThread进行操作
2. SpillThread实际调用的是sortAndSpill方法，这个方法会把当前内存中的数据都存储到一个临时文件中，如果需要，也会把索引写到临时文件中
3. 写临时文件的方式是，从第一个partition开始，把所有属于这个partition的数据都append文件中，然后依次执行接下来的partition
4. map结束后，会把所有临时文件合并到一个最终文件中（通过mergeParts方法），合并时的处理方式和写临时文件时类似，每个partition的数据写在一起。
5. map时有可能出现单个输出数据过大，内存中存放不下的情况，这时要调用spillSingleRecord单独产生一个临时文件。

属性:

1. **private** **final** **int** partitions;
2. **private** **final** JobConf job;
3. **private** **final** TaskReporter reporter;
4. **private** **final** CombinerRunner<K,V> combinerRunner;
5. **private** **final** CombineOutputCollector<K, V> combineCollector;
6. **private** CompressionCodec codec = **null**;
7. **private** ArrayList<SpillRecord> indexCacheList;
8. **private** **final** SpillThread spillThread = **new** SpillThread();
9. **private** **byte**[] kvbuffer; // main output buffer
10. **private** **final** **int**[] kvindices; // partition, k/v offsets into kvbuffer

存储的是kvbuffer中的偏移信息

1. **private** **final** **int**[] kvoffsets; // indices into kvindices

存储的是kvindices中的偏移信息

1. **private** **final** BlockingBuffer bb = **new** BlockingBuffer();

缓存了key和value，满了之后会复制到kvbuffer中

构造函数：

**public** MapOutputBuffer(TaskUmbilicalProtocol umbilical, JobConf job, TaskReporter reporter)

1. 从配置中读取spillper，recper，sortmb，然后检查合法性
2. 从配置中读取sorter的类并初始化，默认使用QuickSort
3. 设置**int** maxMemUsage = sortmb << 20;（即100 \* 1024 \*10240,100M）
4. 初始化kvbuffer，
5. 初始化comparator = job.getOutputKeyComparator();
6. 如果job.getCompressMapOutput()为true，就需要压缩输出，则初始化codec
7. 初始化combinerRunner和combineCollector
8. 读取minSpillsForCombine = job.getInt("min.num.spills.for.combine", 3);
9. 启动spillThread

方法：

1. **public** **synchronized** **void** collect(K key, V value, **int** partition )
   1. 获取**final** **int** kvnext = (kvindex + 1) % kvoffsets.length;
   2. 持有锁spillLock.lock();（这里和SpillThread互斥）
   3. 如果数据没有就位，则等待
   4. 序列化key到bb：keySerializer.serialize(key);
   5. 序列化value到bb：valSerializer.serialize(value);
   6. 设置index：
2. **int** ind = kvindex \* *ACCTSIZE*;
3. kvoffsets[kvindex] = ind;
4. kvindices[ind + *PARTITION*] = partition;
5. kvindices[ind + *KEYSTART*] = keystart;
6. kvindices[ind + *VALSTART*] = valstart;
7. 滚动到下一个kvindex = kvnext;
8. **public** **int** compare(**int** i, **int** j)

先比较分区，再比较key，即线比较kvindices中二者*PARTITION*的大小，如果相同，再比较kvindices中二者key的大小

1. **private** **void** sortAndSpill()
   1. 计算预估长度**long** size = (bufend >= bufstart ? bufend - bufstart : (bufvoid - bufend) + bufstart) + partitions \* *APPROX\_HEADER\_LENGTH*;
   2. 新建**final** SpillRecord spillRec = **new** SpillRecord(partitions);
   3. 获取文件名**final** Path filename = mapOutputFile.getSpillFileForWrite(numSpills, size);并初始化out = rfs.create(filename);
   4. 执行排序：sorter.sort(MapOutputBuffer.**this**, kvstart, endPosition, reporter);
   5. 循环**for** (**int** i = 0; i < partitions; ++i)：
      1. 新建IFile.Writer<K, V> writer = **new** Writer<K, V>(job, out, keyClass, valClass, codec, spilledRecordsCounter);
      2. 如果combinerRunner为null：
         1. 从缓存中读取key和value
         2. 遍历kvindices，对满足当前分区i的，把key和value写到writer.append(key, value);
      3. 如果combinerRunner不为null：
         1. 设置combineCollector.setWriter(writer);
         2. 调用combinerRunner.combine(kvIter, combineCollector);进行combine操作
      4. 关闭writer.close();
      5. 可以看出，对于每个partition，都是写在一起的，而所有的partition都是写在一个文件中。每次循环都重新打开writer？
   6. 如果totalIndexCacheMemory >= *INDEX\_CACHE\_MEMORY\_LIMIT*，则需要创建index文件：
      1. 获取路径Path indexFilename = mapOutputFile.getSpillIndexFileForWrite
      2. 写：spillRec.writeToFile(indexFilename, job);
   7. 否则，添加到indexCacheList.add(spillRec);并增加totalIndexCacheMemory +=spillRec.size() \* *MAP\_OUTPUT\_INDEX\_RECORD\_LENGTH*;
   8. 最后numSpills++
   9. 因为每次执行spill，获取文件时都会加上numSpills参数，而每次spill之后都会执行增加numSpills，所以每次spill都会产生一个新的文件，而每个文件中都包含了所有partition的数据，最后读取的时候直接合并即可
2. **private** **void** spillSingleRecord(**final** K key, **final** V value, **int** partition)

当内存不能承载一个record时，需要直接spill到文件中：

找到对应的partition，然后写到writer中，最后增加++numSpills;（其他都和sortAndSpill一样，除了不执行combine操作，这里只写了一条数据，不需要combine）

1. **private** **void** mergeParts()
   1. 如果numSpills为1，则不需要合并，只要重命名即可
   2. 否则，把所有文件名填充到Path[] filename，并把所有索引文件读取出来后放入indexCacheList
   3. 新建最终的输出文件finalOutputFile：file.out和最终的索引文件finalIndexFile：file.out.index
   4. 如果numSpills为0，则写两个空文件
   5. 否则，循环**for** (**int** parts = 0; parts < partitions; parts++)
      1. 内存循环**for**(**int** i = 0; i < numSpills; i++)读取当前partition的所有数据
      2. 使用Merger.*merge*合并
      3. 写到最终文件
   6. 最后删除所有的临时文件

### SpillThread

实现了Thread

**public** **void** run()：

1. 锁住spillLock.lock();并设置spillThreadRunning = **true**;
2. 循环**while** (**true**)：
   1. spillDone.signal();
   2. 当kvstart == kvend，则一直等待spillReady.await();
   3. **try**
      1. 释放锁：spillLock.unlock();
      2. 调用sortAndSpill();
   4. **finally**
      1. 再次持有锁spillLock.lock();
      2. 如果bufend < bufindex && bufindex < bufstart，则设置bufvoid = kvbuffer.length;
      3. 设置kvstart = kvend;和bufstart = bufend;
3. 如果循环结束，则释放锁spillLock.unlock();并设置spillThreadRunning = **false**;

### BlockingBuffer

继承自DataOutputStream

/\*\*

\* Inner class managing the spill of serialized records to disk.

\*/

### Buffer

继承自OutputStream

属性：

**private** **final** **byte**[] scratch = **new** **byte**[1];

# ReduceTask

继承自Task，处理reduce任务

实例块中会执行：

setPhase(TaskStatus.Phase.*SHUFFLE*);

属性：

1. **private** **int** numMaps;
2. **private** ReduceCopier reduceCopier;
3. **private** Progress copyPhase;
4. **private** Progress sortPhase;
5. **private** Progress reducePhase;

reduce的进度分为三个部分，参考web界面

1. **private** **final** SortedSet<FileStatus> mapOutputFilesOnDisk

存储的本地的map输出，父类Task中的mapOutputFile和这个对应，是非本地的

1. **private** CompressionCodec codec;

方法：

1. **public** **void** run(JobConf job, **final** TaskUmbilicalProtocol umbilical)
   1. 初始化三个子进度阶段：copyPhase，sortPhase和reducePhase
   2. 初始化TaskReporter reporter并启动之
   3. 和map一样，针对三个特殊任务进行处理：jobCleanup，jobSetup和taskCleanup
   4. 如果配置中mapred.job.tracker不为local（即不是本地的测试情况）：
      1. 初始化reduceCopier = **new** ReduceCopier(umbilical, job, reporter);
      2. 调用reduceCopier.fetchOutputs()获取数据
   5. 到此处数据复制阶段应该已经完成：
      1. 调用copyPhase.complete();设置进度
      2. 设置阶段setPhase(TaskStatus.Phase.*SORT*);
      3. 通知TT：statusUpdate(umbilical);
   6. 根据是否为local的mapred，初始化RawKeyValueIterator rIter：
      1. 如果是local，调用Merger.*merge*获取
      2. 否则，使用reduceCopier.createKVIterator(job, rfs, reporter);
   7. 清空mapOutputFilesOnDisk.clear();
   8. 到此处，sort阶段应该已经完成：
      1. sortPhase.complete();
      2. setPhase(TaskStatus.Phase.*REDUCE*);
      3. statusUpdate(umbilical);
   9. 调用runNewReducer(job, umbilical, reporter, rIter, comparator, keyClass, valueClass);
   10. 最后调用done(umbilical, reporter);
2. **private** <INKEY,INVALUE,OUTKEY,OUTVALUE> **void** runNewReducer(JobConf job, **final** TaskUmbilicalProtocol umbilical, **final** TaskReporter reporter, RawKeyValueIterator rIter, RawComparator<INKEY> comparator, Class<INKEY> keyClass, Class<INVALUE> valueClass)
   1. 把输入数据的迭代器rIter包装成可以报告process的
   2. 新建用task的配置：taskContext = **new** TaskAttemptContext(job, getTaskID());
   3. 从配置中读取Reducer<INKEY,INVALUE,OUTKEY,OUTVALUE> reducer
   4. 初始化结果输出类RecordWriter<OUTKEY,OUTVALUE> trackedRW = **new** NewTrackingRecordWriter
   5. 新建用于reduce的配置：Reducer.Context reducerContext = *createReduceContext*
   6. 运行reducer.run(reducerContext);
   7. 然后关闭输出：trackedRW.close(reducerContext);

## NewTrackingRecordWriter

继承自RecordWriter

构造函数中会初始化

RecordWriter<K,V> real = outputFormat.getRecordWriter(taskContext);

outputFormat即为最后reduce输出的格式，一般为文件。

所有的操作都是对real的封装

## ReduceCopier<K, V>

属性：

1. **private** List<MapOutputLocation> scheduledCopies;

the list of map outputs currently being copied

1. **private** List<CopyResult> copyResults;

the results of dispatched copy attempts

1. **private** **int** numCopiers;

the number of outputs to copy in parallel

1. **private** **int** maxInFlight;

可参与竞争的数目

1. **private** Map<String, Long> penaltyBox;

busy hosts from which copies are being backed off （host -> next contact time）

1. **private** Set<String> uniqueHosts;

the set of unique hosts from which we are copying

1. **private** ShuffleRamManager ramManager;

A reference to the RamManager for writing the map outputs to.

1. **private** List<MapOutputCopier> copiers = **null**;

The threads for fetching the files

1. **private** List<MapOutputLocation> retryFetches

a list of map output locations for fetch retrials

1. **private** Set <TaskID> copiedMapOutputs

The set of required map outputs？不是已经复制的吗？

1. **private** Set <TaskAttemptID> obsoleteMapIds

The set of obsolete map taskids.

1. **private** CombinerRunner combinerRunner;
2. **private** CombineOutputCollector combineCollector = **null**;
3. Set<TaskID> fetchFailedMaps = **new** TreeSet<TaskID>();

尝试了maxFetchRetriesPerMap次后依然失败

1. Map<TaskAttemptID, Integer> mapTaskToFailedFetchesMap

A map of taskId -> no. of failed fetches

1. **private** **final** List<MapOutput> mapOutputsFilesInMemory

List of in-memory map-outputs.

1. **private** **final** Map<String, List<MapOutputLocation>> mapLocations

主机到主机上map输出列表的映射

构造函数：

**public** ReduceCopier(TaskUmbilicalProtocol umbilical, JobConf conf, TaskReporter reporter)

1. 设置classloader：configureClasspath(conf);
2. 初始化combinerRunner和combineCollector
3. 初始化ramManager = **new** ShuffleRamManager(conf);
4. 初始化其他数据结构和属性

方法：

1. **private** **void** configureClasspath(JobConf conf)
   1. 把jar包同级目录下的lib目录中的所有文件添加到classloader
   2. 把同级的classes目录也添加到classloader
   3. 把jobCacheDir也添加到classloader
   4. 把存放job配置文件的目录添加到classloader（**new** File(task.getJobFile()).getParentFile()）
   5. 最后新建URLClassLoader loader = **new** URLClassLoader(urls, parent);
   6. 设置conf.setClassLoader(loader);
2. **public** **boolean** fetchOutputs()
   1. 给reduceTask的copyPhase添加numMaps个子progress
   2. 初始化copiers = **new** ArrayList<MapOutputCopier>(numCopiers);并新建numCopiers个MapOutputCopier填充到其中，然后启动所有的MapOutputCopier
   3. 初始化localFSMergerThread = **new** LocalFSMerger((LocalFileSystem)localFileSys);和inMemFSMergeThread = **new** InMemFSMergeThread();，然后启动这两个线程
   4. 初始化getMapEventsThread = **new** GetMapEventsThread();并启动之
   5. 循环**while** (copiedMapOutputs.size() < numMaps && mergeThrowable == **null**)，直到收集了所有map的输出
      1. 遍历retryFetches，把所有失败重试的location都添加到mapLocations，然后清空retryFetches.clear();
      2. 持有scheduledCopies的锁，开始调度：
         1. 获取所有主机hostList.addAll(mapLocations.keySet());，然后混排Collections.*shuffle*(hostList, **this**.random);（为了防止出现热点机器）
         2. 遍历**while** (hostsItr.hasNext())，对每个host：
            1. 查询List<MapOutputLocation> knownOutputsByLoc = mapLocations.get(host);（每个host上可能有多个TT）
            2. 如果uniqueHosts.contains(host)即已经包含了host，则设置numDups += knownOutputsByLoc.size();后**continue**;
            3. 查询Long penaltyEnd = penaltyBox.get(host);（host是否被添加到暂时禁用列表），如果penaltyEnd不为null：

如果currentTime < penaltyEnd.longValue()，即还没有度过禁用期，**continue**;

否则，移除penaltyBox.remove(host);

* + - * 1. 遍历该节点上所有的location：对每个MapOutputLocation loc：

如果obsoleteMapIds中包含了loc，则删除之locItr.remove();，然后**continue**;

添加uniqueHosts.add(host);（这个步骤和上面的numDups对应）

添加scheduledCopies.add(loc);

删除locItr.remove();

设置numInFlight++; numScheduled++;

**break**;（每个host每次只取一个map location）

* + - 1. 唤醒scheduledCopies.notifyAll();
    1. 如果numInFlight == 0 && numScheduled == 0，既没有发现可以复制的location：
       1. 通知TT reporter.progress();，以防止被杀掉
       2. 睡眠等待Thread.*sleep*(5000);
    2. 循环**while** (numInFlight > 0 && mergeThrowable == **null**)
       1. 调用CopyResult cr = getCopyResult(numInFlight);
       2. 如果cr为null，则直接**break**;
       3. 如果cr.getSuccess()，即复制成功：
          1. numCopied++;
          2. 统计复制的平均速度并更新到copyPhase
          3. 从fetchFailedMaps中移除cr.getLocation().getTaskId()（即这个task的输出已近复制成功，可以移除）
       4. 如果cr.isObsolete()，即这个复制任务被废弃，可直接忽略这种情况
       5. 其他情况（即复制失败的情况）：
          1. 添加到重试列表retryFetches.add(cr.getLocation());
          2. 获取TaskAttemptID mapTaskId和TaskID mapId
          3. totalFailures++;
          4. 查询之前总的失败次数：Integer noFailedFetches = mapTaskToFailedFetchesMap.get(mapTaskId);
          5. 设置noFailedFetches为noFailedFetches+1并更新到mapTaskToFailedFetchesMap
          6. 如果noFailedFetches >= abortFailureLimit，即超过了允许的失败次数：打印fatal，然后报告给TT：umbilical.shuffleError
          7. 调用checkAndInformJobTracker(noFailedFetches, mapTaskId, cr.getError().equals(CopyOutputErrorType.*READ\_ERROR*));
          8. 如果noFailedFetches == maxFetchFailuresBeforeReporting：

添加到fetchFailedMaps.add(mapId);

检查reducer是否healthy（reducerHealthy）：totalFailures / (totalFailures + numCopied)是否小于*MAX\_ALLOWED\_FAILED\_FETCH\_ATTEMPT\_PERCENT*（即失败率低于50%）

检查reducer是否progressed enough（reducerProgressedEnough）：numCopied / numMaps是否大于等于*MIN\_REQUIRED\_PROGRESS\_PERCENT*（即已处理的数目是否超过了50%）

检查reducer是否stall（reducerStalled）：stallDuration / minShuffleRunDuration是否大于*MAX\_ALLOWED\_STALL\_TIME\_PERCENT*

如果

fetchFailedMaps的大小超过了maxFailedUniqueFetches（即多次尝试后依然失败的map数超过了限额）或者fetchFailedMaps的数目等于numMaps - copiedMapOutputs.size()（即剩余没完成的复制任务，就是那些一直失败的任务）

且reducerHealthy为false

且reducerProgressedEnough为false或者reducerStalled

满足上面三点，则打印fatal，并报告错误umbilical.shuffleError

* + - * 1. 设置禁用时间：penaltyBox.put(cr.getHost(), currentTime + currentBackOff);
      1. 移除uniqueHosts.remove(cr.getHost());
      2. numInFlight--;
  1. 到此处已经执行完毕所有的复制任务
  2. 设置exitGetMapEvents= **true**;
  3. 等待getMapEventsThread.join();
  4. 清空copiers
  5. 持有mapOutputFilesOnDisk的锁：
     1. 设置exitLocalFSMerge = **true**;
     2. 唤醒mapOutputFilesOnDisk.notify();
  6. 关闭ramManager.close();
  7. 如果mergeThrowable == **null**（即没有出现错误）：
     1. 等待磁盘merge结束localFSMergerThread.join();
     2. 等待内存merge结束inMemFSMergeThread.join();
  8. 返回mergeThrowable == **null** && copiedMapOutputs.size() == numMaps;

1. **private** RawKeyValueIterator createKVIterator(JobConf job, FileSystem fs, Reporter reporter)
   1. 如果mapOutputsFilesInMemory.size() > 0
      1. 获取第一个TaskID mapId = mapOutputsFilesInMemory.get(0).mapId;
      2. 构建内层中的segment数据结构：inMemToDiskBytes = createInMemorySegments(memDiskSegments, maxInMemReduce);
      3. 查询内存中segment的数目**final** **int** numMemDiskSegments = memDiskSegments.size();（memDiskSegments即为上一步中计算的结果）
      4. 如果numMemDiskSegments > 0且ioSortFactor > mapOutputFilesOnDisk.size()（即factor大于磁盘上文件的数目，这种情况要把内存中的数据先写到磁盘，可能是因为后面merge的时候需要考虑优化，所以在此处先处理这种情况，参考MergeQueue的factor计算方法）：
         1. 新获取一个map中间文件的输出路径：**final** Path outputPath = mapOutputFile.getInputFileForWrite(mapId, inMemToDiskBytes);
         2. 执行merge：
            1. **final** RawKeyValueIterator rIter = Merger.*merge*(job, fs, keyClass, valueClass, memDiskSegments, numMemDiskSegments, tmpDir, comparator, reporter, spilledRecordsCounter, **null**);
            2. **final** Writer writer = **new** Writer(job, fs, outputPath, keyClass, valueClass, codec, **null**);
            3. Merger.*writeFile*(rIter, writer, reporter, job);
         3. 最后生成的文件加入到磁盘segment列表中addToMapOutputFilesOnDisk(fs.getFileStatus(outputPath));
         4. 清空内存中的segment列表：memDiskSegments.clear();（注意，这个步骤之后mapOutputsFilesInMemory中可能还包含内容）
      5. 否则，保留内存中的segment，之后会和磁盘上的segment一起merge
   2. 获取所有磁盘上的map输出文件（已经过了copy的阶段，所有的文件应该都在本地磁盘）Path[] onDisk = getMapFiles(fs, **false**);
   3. 然后遍历所有onDisk，全部添加到diskSegments.add(**new** Segment<K, V>(job, fs, file, codec, keepInputs));
   4. 按照文件长度排序diskSegments，小的在前
   5. 再次调用**long** inMemBytes = createInMemorySegments(finalSegments, 0);（和上面不同，这次传入的是0，所以会把所有的内存中的mapout都填充到finalSegments）先把内存中的mapout填充到finalSegments（由此可见，内存中的segment是排在最前面的，如果不做特殊配置，这里获得的内存中的segment列表应该为空！因为第一次调用时，maxInMemReduce默认为0）
   6. 对于磁盘上的segment，和该方法最开始，第一次调用createInMemorySegments产生的memDiskSegments：
      1. 把memDiskSegments添加到diskSegments的最前部
      2. 执行合并RawKeyValueIterator diskMerge = Merger.*merge*(job, fs, keyClass, valueClass, codec, diskSegments, ioSortFactor, numInMemSegments, tmpDir, comparator, reporter, **false**, spilledRecordsCounter, **null**);
      3. 清空diskSegments.clear();
      4. 如果finalSegments为空（默认情况），则直接返回上面的合并结果diskMerge
      5. 否则，添加到finalSegments.add(**new** Segment<K,V>( **new** RawKVIteratorReader(diskMerge, onDiskBytes), **true**));
   7. 如果上面没有返回，则再次执行最后一次merge，然后返回：**return** Merger.*merge*(job, fs, keyClass, valueClass, finalSegments, finalSegments.size(), tmpDir, comparator, reporter, spilledRecordsCounter, **null**);

注意！！：有一个参数需要注意：maxInMemReduce，这个参数实际值是配置值mapred.job.reduce.input.buffer.percent乘以最大内存数。含义是：保存在内存中的map输出数据占总内存数的比例，默认值为0，即要求map输出shuffle阶段全部dump到硬盘，不保存在内存中。而如果修改这个值，就可以保存一部分map输出在内存中，这就对应了最后一个步骤g。综合考虑步骤a，f，g：

* + 1. a中，如果factor大于磁盘segment数目，就会导致合并时只执行一轮，没有把内存中的segment写到磁盘，所以在a步骤中会特别对待这种情况，提前把超出maxInMemReduce的内存中的segment写到磁盘
    2. 而f步骤中，经过了a中的过滤，可以保证：
       1. 要么内存中超限的segment已经被写到磁盘了
       2. 要么会在第一轮merge时被写到磁盘

最后，如果maxInMemReduce为0，在此处就会直接返回，否则才会执行g

* + 1. 最后，在g步骤中，除了仍保存在内存中的segment，就只有一个f步骤的输出最为磁盘上的segment，所以最后会直接返回merge的结果，内存中的还在内存中！

见图：



1. **private** **long** createInMemorySegments(List<Segment<K, V>> inMemorySegments, **long** leaveBytes)

持有mapOutputsFilesInMemory的锁：

* 1. 遍历mapOutputsFilesInMemory，计算总的数据大小：fullSize
  2. 循环**while**(fullSize > leaveBytes)（leaveBytes默认为0）：
     1. 获取第一个MapOutput mo = mapOutputsFilesInMemory.remove(0);
     2. fullSize -= mo.data.length; totalSize += mo.data.length;
     3. 新建Reader<K, V> reader = **new** InMemoryReader<K, V>(ramManager, mo.mapAttemptId, mo.data, 0, mo.data.length);
     4. 构造Segment<K, V> segment = **new** Segment<K, V>(reader, **true**);
     5. 添加到inMemorySegments.add(segment);
  3. 最后返回已添加的总大小：totalSize

### CopyResult

/\*\* Represents the result of an attempt to copy a map output \*/

属性：

1. **private** **final** MapOutputLocation loc;
2. **private** **final** **long** size;

### MapOutputLocation

/\*\*

\* Abstraction to track a map-output.

\*/

属性：

1. TaskAttemptID taskAttemptId;
2. TaskID taskId;
3. String ttHost;
4. URL taskOutput;

### MapOutput

/\*\* Describes the output of a map; could either be on disk or in-memory. \*/

属性：

1. **final** TaskID mapId;
2. **final** TaskAttemptID mapAttemptId;
3. **final** Path file;
4. **final** Configuration conf;
5. **byte**[] data;
6. **final** **boolean** inMemory;
7. **long** compressedSize;

### ShuffleRamManager

实现了RamManager接口，负责管理shuffle使用的内存。

保证reduce使用的内存不超过配置的数目，即已经使用的内存和size不大于maxSize

方法：

1. **public** **synchronized** **boolean** reserve(**int** requestedSize, InputStream in)

申请内存，如果不能满足则等待

1. **public** **synchronized** **void** unreserve(**int** requestedSize)

归还内存，会唤醒1中的等待

1. **public** **boolean** waitForDataToMerge()

待看

### MapOutputCopier

继承自Thread，负责在map的输出可用的时候复制到本地

属性：

1. **private** MapOutputLocation currentLocation = **null**;
2. **private** CompressionCodec codec = **null**;
3. **private** Decompressor decompressor = **null**;

方法：

1. **public** **void** run()

一直循环直到map任务产出数据可以抓取：**while** (**true**)：

* 1. 等待scheduledCopies被填充，如果不为空，则取第一个：loc = scheduledCopies.remove(0);
  2. 调用start(loc);设置currentLocation
  3. 开始执行size = copyOutput(loc);
  4. 最后调用finish(size, error);

1. **private** **long** copyOutput(MapOutputLocation loc)

通过http从远程复制数据：

* 1. 如果copiedMapOutputs或obsoleteMapIds中包含了loc，则直接返回CopyResult.*OBSOLETE*（即loc已经复制成功或者已经废弃）
  2. 新建一个临时文件用于存放复制的数据，然后执行获取数据：MapOutput mapOutput = getMapOutput(loc, tmpMapOutput, reduceId.getTaskID().getId());
  3. 持有锁ReduceTask.**this**：
     1. 再次查询copiedMapOutputs，如果已经包含了loc，则删除复制的文件mapOutput.discard();然后返回*OBSOLETE*
     2. 如果bytes = mapOutput.compressedSize;为0，则删除这个空文件，并通知完成noteCopiedMapOutput(loc.getTaskId());，然后返回bytes
     3. 如果mapOutput.inMemory，即复制的数据存放在内存中，则添加到mapOutputsFilesInMemory.add(mapOutput);
     4. 否则，即复制到了本地文件：
        1. 把tmpMapOutput重命名为正式文件
        2. 添加到mapOutputFilesOnDisk中
     5. 最后调用noteCopiedMapOutput(loc.getTaskId());并返回bytes

1. **private** MapOutput getMapOutput(MapOutputLocation mapOutputLoc, Path filename, **int** reduce)

从远程复制数据到本地，可能在内存也肯能在本地文件：

* 1. 获取到mapOutputLoc的连接URLConnection connection = url.openConnection();
  2. 建立连接：InputStream input = setupSecureConnection(mapOutputLoc, connection);
  3. 读取连接的header信息，并验证
  4. 如果满足下面两点则会把远程数据复制到内存：
     1. 未压缩的文件大小小于inmem fs的25%
     2. inmem fs有space（ramManager先检查单个文件大小，再检查所有文件加起来是否超限）
     3. 检查是否可以写入内存：**boolean** shuffleInMemory = ramManager.canFitInMemory(decompressedLength);
  5. 如果shuffleInMemory：
     1. 调用mapOutput = shuffleInMemory(mapOutputLoc, connection, input, (**int**)decompressedLength, (**int**)compressedLength);
  6. 否则：
     1. 调用mapOutput = shuffleToDisk(mapOutputLoc, input, filename, compressedLength);

1. **private** InputStream setupSecureConnection(MapOutputLocation mapOutputLoc, URLConnection connection)
   1. 获取输入流：InputStream input = getInputStream(connection, shuffleConnectionTimeout, shuffleReadTimeout);
   2. 获取对方的回复：String replyHash = connection.getHeaderField(SecureShuffleUtils.*HTTP\_HEADER\_REPLY\_URL\_HASH*);
   3. 验证：SecureShuffleUtils.*verifyReply*(replyHash, encHash, jobTokenSecret);
2. **private** MapOutput shuffleInMemory(MapOutputLocation mapOutputLoc, URLConnection connection, InputStream input, **int** mapOutputLength, **int** compressedLength)
   1. 申请内存：**boolean** createdNow = ramManager.reserve(mapOutputLength, input);
   2. 如果createdNow为false，可能是因为连接失败，重新连接，如果还是没有连接成功，则归还内存ramManager.unreserve(mapOutputLength);
   3. 如果codec不为null，则需要使用压缩input = codec.createInputStream(input, decompressor);
   4. 新建字节数组：**byte**[] shuffleData = **new** **byte**[mapOutputLength];，然后用其初始化MapOutput mapOutput
   5. 循环读取**int** n = input.read(shuffleData, 0, shuffleData.length);直到读完后关闭连接
   6. 调用ramManager.closeInMemoryFile(mapOutputLength);
   7. 如果读取的数据头信息中的不同：bytesRead != mapOutputLength，则进行清空处理后抛出IOException
   8. 返回mapOutput
3. **private** MapOutput shuffleToDisk(MapOutputLocation mapOutputLoc, InputStream input, Path filename, **long** mapOutputLength)
   1. 在本地查找一个位置：Path localFilename = lDirAlloc.getLocalPathForWrite(filename.toUri().getPath(),mapOutputLength, conf);然后用之新建MapOutput mapOutput
   2. 构建输出流output = rfs.create(localFilename);，然后循环读取和写出
   3. 如果读取的数据头信息中的不同：bytesRead != mapOutputLength，则进行清空处理后抛出IOException

### LocalFSMerger

继承自Thread，负责执行磁盘上mapout的merge，当文件较多时才会触发，每次合并ioSortFactor个文件到一个文件中。

当复制阶段全部结束后就会停止这个线程（设置exitLocalFSMerge为true），并等待localFSMergerThread结束！

属性：

**private** LocalFileSystem localFileSys;

**public** **void** run()：

循环**while**(!exitLocalFSMerge)：

1. 持有mapOutputFilesOnDisk的锁：
   1. 如果mapOutputFilesOnDisk.size() < (2 \* ioSortFactor - 1)，则一直等待：mapOutputFilesOnDisk.wait();
2. 再次检查exitLocalFSMerge，如果为false，则直接break
3. 持有mapOutputFilesOnDisk的锁：
   1. 取mapOutputFilesOnDisk中的前ioSortFactor个，移除后，添加到mapFiles中
4. 新建一个输出文件Path outputPath = lDirAlloc.getLocalPathForWrite(mapFiles.get(0).toString(),approxOutputSize, conf) .suffix(".merged");
5. 新建Writer writer = **new** Writer(conf,rfs, outputPath, conf.getMapOutputKeyClass(),conf.getMapOutputValueClass(),codec, **null**);
6. 执行merge：
   1. iter = Merger.*merge*
   2. Merger.*writeFile*(iter, writer, reporter, conf);
   3. writer.close();
7. 最后添加回addToMapOutputFilesOnDisk(localFileSys.getFileStatus(outputPath));

### InMemFSMergeThread

继承自Thread，负责执行内存中mapout的merge

当复制阶段全部结束后就会停止这个线程，并等待localFSMergerThread结束！

注意，如果配置了combine函数，在此处会执行combine！！

方法：

1. **public** **void** run()

查询exit = ramManager.waitForDataToMerge();，只要exit不为true，就一直循环（只有ramManager为close状态才会执行merge，在fetchOutputs方法中，数据全部复制结束后才会执行ramManager.close();）：

调用doInMemMerge();

1. **private** **void** doInMemMerge()
   1. 查询TaskID mapId = mapOutputsFilesInMemory.get(0).mapId;（使用第一个作为输出的文件名，和merge阶段的内存合并时一样的）
   2. 把内存中的segment全部取出：**long** mergeOutputSize = createInMemorySegments(inMemorySegments, 0);（第二个参数为0，表示全部取出）
   3. 和磁盘合并一样，新建Path outputPath和Writer writer
   4. 执行合并rIter = Merger.*merge*
   5. 如果combinerRunner为null，则写道文件Merger.*writeFile*(rIter, writer, reporter, conf);
   6. 否则，如果combineCollector不为null，则需要执行合并优化，其实和map端的combine类似，在reduce正式开始之前，对部分数据执行reduce操作：
      1. 设置combineCollector.setWriter(writer);
      2. 执行合并combinerRunner.combine(rIter, combineCollector);
   7. 关闭writer.close();
   8. 最后添加到磁盘segment列表：addToMapOutputFilesOnDisk(status);

### GetMapEventsThread

继承自Thread，

方法：

1. **public** **void** run()

如果exitGetMapEvents为false则一直循环：

* 1. 调用**int** numNewMaps = getMapCompletionEvents();
  2. 睡眠Thread.*sleep*(*SLEEP\_TIME*);

1. **private** **int** getMapCompletionEvents()
   1. 查询MapTaskCompletionEventsUpdate update = umbilical.getMapCompletionEvents
   2. TaskCompletionEvent events[] = update.getMapTaskCompletionEvents();
   3. 遍历**for** (TaskCompletionEvent event : events)，根据event.getTaskStatus()判断：
      1. **case** *SUCCEEDED*:
         1. 新建URI u = URI.*create*(event.getTaskTrackerHttp());
         2. 新建URL mapOutputLocation = **new** URL(event.getTaskTrackerHttp() + "/mapOutput?job=" + taskId.getJobID() + "&map=" + taskId + "&reduce=" + getPartition());
         3. 把host和mapOutputLocation 的映射关系添加到mapLocations
         4. **break**;
      2. **case** *FAILED*，*KILLED*和*OBSOLETE*:
         1. 添加到obsoleteMapIds.add(event.getTaskAttemptId());
         2. **break**;
      3. **case** *TIPFAILED*:
         1. 添加到copiedMapOutputs.add(event.getTaskAttemptId().getTaskID());
         2. 这里不知道为什么添加到copiedMapOutputs？这样可以保证之后就忽略这个TaskAttemptId？不再需要这个task的输出？因为复制完成的标志就是copiedMapOutputs的数目已经达到了numMaps

# SpillRecord

属性：

1. **private** **final** ByteBuffer buf; /\*\* Backing store \*/
2. **private** **final** LongBuffer entries; /\*\* View of backing storage as longs \*/

构造函数：

1. **public** SpillRecord(**int** numPartitions)
   1. 初始化buf = ByteBuffer.*allocate*(numPartitions \* MapTask.*MAP\_OUTPUT\_INDEX\_RECORD\_LENGTH*);（每个index战友的字节数默认为24）
   2. 初始化entries = buf.asLongBuffer();
2. **public** SpillRecord(Path indexFileName, JobConf job, Checksum crc, String expectedIndexOwner)
   1. 从文件读取index
   2. 然后和上面类似初始化buf和entries

## IndexRecord

属性：

1. **long** startOffset;
2. **long** rawLength;
3. **long** partLength;

# Progress

用于描述一个任务的进度，可以构建层级式的进度，父级的进度是把子级进度个数平均分配。

属性：

1. **private** String status = "";
2. **private** **float** progress;
3. **private** **int** currentPhase;
4. **private** ArrayList<Progress> phases = **new** ArrayList<Progress>();
5. **private** Progress parent;
6. **private** **float** progressPerPhase;

# Merger

方法：

1. **public** **static** <K **extends** Object, V **extends** Object>RawKeyValueIterator merge
   1. 调用**new** MergeQueue<K, V>的merge方法并返回

## Segment

属性：

1. Reader<K, V> reader = **null**;
2. DataInputBuffer key = **new** DataInputBuffer();
3. DataInputBuffer value = **new** DataInputBuffer();
4. **boolean** preserve = **false**;
5. CompressionCodec codec = **null**;
6. **long** segmentOffset = 0;
7. **long** segmentLength = -1;

主要封装了Reader<K, V>

## MergeQueue

继承自PriorityQueue<Segment<K, V>>（一个部分排序的队列，可以在线性时间内找到最小元素，小根堆？），实现了RawKeyValueIterator接口

MergeQueue中维护了Segment的优先级队列

属性：

List<Segment<K, V>> segments = **new** ArrayList<Segment<K,V>>();

RawComparator<K> comparator;

**private** **long** totalBytesProcessed;

**private** **float** progPerByte;

**private** Progress mergeProgress = **new** Progress();

Segment<K, V> minSegment;

方法：

1. RawKeyValueIterator merge(Class<K> keyClass, Class<V> valueClass, **int** factor, **int** inMem, Path tmpDir, Counters.Counter readsCounter, Counters.Counter writesCounter)

对segments做merge，循环**do** {…} **while**(**true**)：

* 1. 计算factor = getPassFactor(factor, passNo, numSegments - inMem);，如果passNo为1，即第一轮，则设置factor += inMem; （内存中的segment是排在队列最前面的，第一轮把内存中的segment一起merge）
  2. 把所有本轮要进行merge的segment从segments中取出，内部循环**while** (**true**)（每次指定了要factor个segment，在这个内层循环中就必须收集齐，除非已经没有可用的segment）：
     1. 获取List<Segment<K, V>> mStream =getSegmentDescriptors(numSegmentsToConsider);（numSegmentsToConsider即为需要的segment个数，第一次循环为factor，getSegmentDescriptors从segments中取出numSegmentsToConsider个segment，不足则全取）
     2. 遍历mStream，对每个Segment<K, V> segment：
        1. 初始化segment.init(readsCounter);（即打开文件的流，在最后一刻才初始化）
        2. 如果segment.next()为true，即有数据，则添加到segmentsToMerge.add(segment);并设置segmentsConsidered++;
        3. 否则，关闭segment.close();，并设置numSegments--;
     3. 如果segmentsConsidered == factor（即已经搜集齐了factor个segment）或者segments.size() == 0（即已经抽空了所有的segment），则退出循环
     4. 如果这轮循环没收集齐，则设置numSegmentsToConsider = factor - segmentsConsidered;后继续循环
  3. 把上面取出的segmentsToMerge放到优先级队列中：
     1. 调用initialize(segmentsToMerge.size());
     2. 调用clear();
     3. 遍历segmentsToMerge，对每个segment调用put(segment);
     4. 上面三个都是PriorityQueue中的方法，实际是一个排序的过程，把要merge的segment放入到最小堆中，保证了每次取出的都是一条最小的记录
  4. 如果numSegments <= factor（即总的segment数目小于等于factor，一般在最后一次merge时出现）：
     1. 设置totalBytesProcessed = startBytes;（这就是最终的字节总数）
     2. 遍历segmentsToMerge，累加segment的长度到totalBytes
     3. 设置mergeProgress为1，即完成了merge
     4. 返回**this**（这是最后一轮merge，所以不需要执行其他操作，直接返回最终的iterator）
  5. 如果numSegments > factor，即需要再merge：
     1. 遍历segmentsToMerge，估算需要的磁盘空间approxOutputSize（叠加数据和checksum的长度）
     2. 新建一个存储位置：Path tmpFilename =**new** Path(tmpDir, "intermediate").suffix("." + passNo);，在磁盘上找到一个可用的文件位置Path outputFile = *lDirAlloc*.getLocalPathForWrite
     3. 新建Writer<K, V> writer =**new** Writer<K, V>(conf, fs, outputFile, keyClass, valueClass, codec, writesCounter);
     4. 写到文件*writeFile*(**this**, writer, reporter, conf);（因为之前把segmentsToMerge都添加到最小堆中了，所以这里写的时候，直接从最小堆中取record，就是排好序的），写完后关闭writer.close();
     5. 清空最小堆**this**.close();
     6. 把新产出的文件新建为segment：Segment<K, V> tempSegment = **new** Segment<K, V>(conf, fs, outputFile, codec, **false**);
     7. 添加到segments.add(tempSegment);
     8. 重新设置numSegments = segments.size();
     9. 重新排序Collections.*sort*(segments, segmentComparator);（按文件长度排序，小的在前）
     10. passNo++;
  6. 重置factor = origFactor;（因为只有第一轮merger需要考虑特殊情况，之后的直接使用配置的值）

1. **private** **int** getPassFactor(**int** factor, **int** passNo, **int** numSegments) {
   1. 计算一轮需要执行merge的segment个数，参数含义：
      1. factor，用户定义的factor，reduce阶段默认为10，即每次merge的个数
      2. passNo，第几轮merge，初始时为1
      3. numSegments，总的segment个数（除去了原先已经在内存中的）
   2. 如果passNo大于1（不是第一轮），或者numSegments <= factor，或者factor == 1，则直接返回factor
   3. 计算**int** mod = (numSegments - 1) % (factor - 1);
   4. 如果mod为0，则返回factor
   5. 如果mod不为0，则返回mod + 1
   6. 这样可以最小化merge的次数，保证除了第一次外，剩下的每一轮都是合并factor个segment（因为每一轮都新产出一个segment，所以最后一轮也是factor个），这么做的原因可能是考虑到，第一轮会和之前已经在内存中的segment合并，所以尽量减少第一轮要merge的个数？其实并没有减少合并要执行的轮数！（同时可以减少写到磁盘的数据量，摘自hadoop权威指南）
   7. 参考ReduceCopier的createKVIterator方法，如果第一轮时，numSegments <= factor，即不能采取优化方法，会直接把内存中的segment先merge到临时文件

# IFile

IFile表示一个map任务的输出，格式为<key-len, value-len, key, value>

Writer负责写这个文件，Reader负责读

## Writer

属性：

1. FSDataOutputStream out;
2. **boolean** ownOutputStream = **false**;
3. **long** start = 0;
4. FSDataOutputStream rawOut;
5. CompressionOutputStream compressedOut;
6. Compressor compressor;
7. **boolean** compressOutput = **false**;
8. **long** decompressedBytesWritten = 0;
9. **long** compressedBytesWritten = 0;
10. **private** **long** numRecordsWritten = 0;
11. IFileOutputStream checksumOut;

## Reader

属性：

1. **private** **long** numRecordsRead = 0;
2. **private** **long** numRecordsRead = 0;
3. **private** **final** Counters.Counter readRecordsCounter;
4. **final** InputStream in; // Possibly decompressed stream that we read
5. Decompressor decompressor;
6. **long** bytesRead = 0;
7. **final** **long** fileLength;
8. **boolean** eof = **false**;
9. **final** IFileInputStream checksumIn;
10. **byte**[] buffer = **null**;
11. **int** bufferSize = *DEFAULT\_BUFFER\_SIZE*;
12. DataInputBuffer dataIn = **new** DataInputBuffer();
13. **int** recNo = 1;

## InMemoryReader

继承自Reader

属性：

1. RamManager ramManager;
2. TaskAttemptID taskAttemptId;

从内存中直接读取

# IndexCache

# Task总结

## MapTask

1. 每个map任务对应一个输入split
2. 正常的map任务，一般情况下使用新的api，调用runNewMapper，遍历split中所有的记录，使用用户的定义的mapper执行
3. map的输出使用NewOutputCollector进行处理，其实是包装了MapOutputBuffer
4. MapOutputBuffer会根据定义的reduce数目（partitions）处理输出数据：
   1. 数据分为两部分，一个是索引，一个是实际数据，索引记录了数据的kvindex（索引编号）*，PARTITION*（即属于哪个reduce），*KEYSTART*，和*VALSTART*
   2. 主要接收输出数据的方法是collect
   3. 开始时把map处理的结果都放在内存中，当需要spill到磁盘时，通知SpillThread进行操作
   4. SpillThread实际调用的是sortAndSpill方法，这个方法会把当前内存中的数据都存储到一个临时文件中，如果需要，也会把索引写到临时文件中
   5. spill的同时，如果combinerRunner不为null，即需要执行combine操作，则会把写数据到磁盘的任务托管到combinerRunner
   6. 写临时文件的方式是，从第一个partition开始，把所有属于这个partition的数据都append文件中，然后依次执行接下来的partition
   7. map结束后，会把所有临时文件（每次spill都生成一个新的文件）合并到一个最终文件中（通过mergeParts方法），合并时的处理方式和写临时文件时类似，每个partition的数据写在一起。
   8. map时有可能出现单个输出数据过大，内存中存放不下的情况，这时要调用spillSingleRecord单独产生一个临时文件。

## ReduceTask

1. reduce从每个map那里复制数据到本地，然后进行处理
2. 共分为三个阶段，分别是copyPhase，sortPhase，reducePhase
3. copyPhase负责从每个map任务的主机上复制数据
   1. 使用ReduceCopier的fetchOutputs方法
   2. fetchOutputs方法为每个map输出文件分配一个MapOutputCopier
   3. MapOutputCopier的getMapOutput方法从远端复制数据到本地，分情况输出到文件或保留在内存（分别对应shuffleToDisk和shuffleInMemory两个方法）
   4. ReduceCopier会启动三个线程：
      1. LocalFSMerger，负责在复制的同时，对已经复制到本地的临时文件进行merge
      2. InMemFSMergeThread，类似上面，对内存中的进行merge
      3. GetMapEventsThread，接受map任务的完成时间，触发复制数据
4. sortPhase负责对所有数据进行排序和和并：
   1. 使用Merger.*merge*进行合并
   2. 其实这连个是在一起进行的，因为使用了最小堆（PriorityQueue），保证每次取的时候都是取出所有record中最小的一个
   3. 最终生成一个所有临时文件的迭代器，RawKeyValueIterator rIter，这个迭代器能保证所有的数据，在取出的时候是有序的（其实实体文件，在此时并没有被合并成一个文件，只是构造了一个统一的已排序视图）
5. reducePhase，实际执行用户定义的reducer
   1. reduce的输出应该是用户定义的