# 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

静态块中，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. 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];

# 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;

# IFile