1、Map-Reduce 的逻辑过程

假设我们需要处理一批有关天气的数据，其格式如下：

•按照 ASCII 码存储，每行一条记录

•每一行字符从 0 开始计数，第 15 个到第 18 个字符为年

•第 25 个到第 29 个字符为温度，其中第 25 位是符号+/-

```
0067011990999991950051507+0000+
0043011990999991950051512+0022+
0043011990999991950051518-0011+
0043012650999991949032412+0111+
0043012650999991949032418+0078+
0067011990999991937051507+0001+
0043011990999991937051512-0002+
0043011990999991945051518+0001+
0043012650999991945032412+0002+
0043012650999991945032418+0078+
```

我们现在需要统计出每年的最高温度。

Map-Reduce 主要包括两个步骤：Map 和 Reduce

每一步都有 key-value 对作为输入和输出：

•map 阶段的 key-value 对的格式是由输入的格式所决定的，如果是默认的 TextInputFormat，则每行作为一个记录进程处理，其中 key 为此行的开头相对于文件的起始位置，value 就是此行的字符文本

•map 阶段的输出的 key-value 对的格式必须同 reduce 阶段的输入 key-value 对的格式相对应

对于上面的例子，在 map 过程，输入的 key-value 对如下：

(0, 0067011990999991950051507+0000+)
(33, 0043011990999991950051512+0022+)
(66, 0043011990999991950051518-0011+)
(99, 0043012650999991949032412+0111+)
(132, 0043012650999991949032418+0078+)
(165, 0067011990999991937051507+0001+)
(198, 0043011990999991937051512-0002+)
(231, 0043011990999991945051518+0001+)
(264, 0043012650999991945032412+0002+)
(297, 0043012650999991945032418+0078+)

在 map 过程中，通过对每一行字符串的解析，得到年-温度的 key-value 对作为输出：

(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
(1949, 78)
(1937, 1)
(1937, -2)
(1945, 1)
(1945, 2)
(1945, 78)

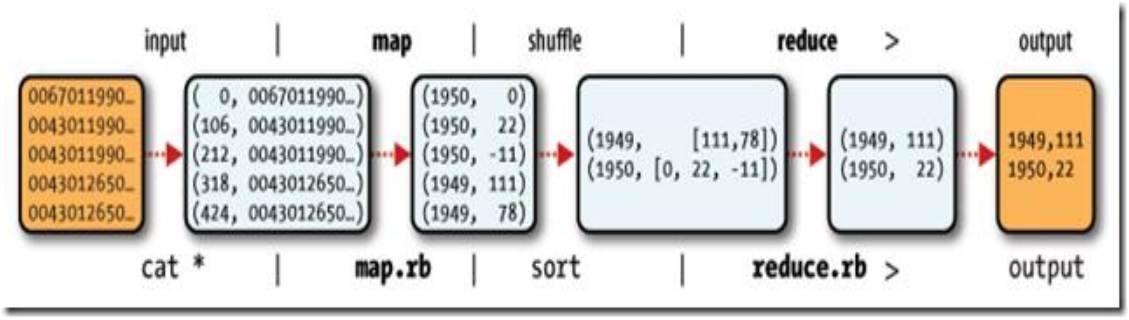
在 reduce 过程，将 map 过程中的输出，按照相同的 key 将 value 放到同一个列表中作为 reduce 的输入

(1950, [0, 22, –11])
(1949, [111, 78])
(1937, [1, -2])
(1945, [1, 2, 78])

在 reduce 过程中，在列表中选择出最大的温度，将年-最大温度的 key-value 作为输出：

(1950, 22)
(1949, 111)
(1937, 1)
(1945, 78)

其逻辑过程可用如下图表示：



2、编写 Map-Reduce 程序

编写 Map-Reduce 程序，一般需要实现两个函数：mapper 中的 map 函数和 reducer 中的 reduce 函数。

一般遵循以下格式：

- map: (K1, V1) -> list(K2, V2)

```
public interface Mapper<K1, V1, K2, V2> extends JobConfigurable, Closeable {
 void map(K1 key, V1 value, OutputCollector<K2, V2> output, Reporter reporter)
 throws IOException;
}
```

- reduce: (K2, list(V)) -> list(K3, V3)

```
public interface Reducer<K2, V2, K3, V3> extends JobConfigurable, Closeable {
 void reduce(K2 key, Iterator<V2> values,
        OutputCollector<K3, V3> output, Reporter reporter)
   throws IOException;
}
```

注：Reporter：

Reporter 是用于 Map/Reduce 应用程序报告进度（心跳），设定应用级别的状态消息， 更新 Counters（计数器）的机制。

OutputCollector：

OutputCollector 是一个 Map/Reduce 框架提供的用于收集 Mapper 或 Reducer 输出数据的通用机制 （包括中间输出结果和作业的输出结果）。

对于上面的例子，则实现的 mapper 如下：

```
public class MaxTemperatureMapper extends MapReduceBase implements Mapper<LongWritable, Text, Text,
IntWritable> {
  @Override
  public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws
IOException {
    String line = value.toString();
    String year = line.substring(15, 19);
    int airTemperature;
    if (line.charAt(25) == '+') {
      airTemperature = Integer.parseInt(line.substring(26, 30));
    } else {
      airTemperature = Integer.parseInt(line.substring(25, 30));
    }
    output.collect(new Text(year), new IntWritable(airTemperature));
  }
}
```

实现的 reducer 如下：

```
public class MaxTemperatureReducer extends MapReduceBase implements Reducer<Text, IntWritable, Text,
IntWritable> {
  public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter
reporter) throws IOException {
    int maxValue = Integer.MIN_VALUE;
    while (values.hasNext()) {
      maxValue = Math.max(maxValue, values.next().get());
    }
    output.collect(key, new IntWritable(maxValue));
  }
}
```

注：代码中 LongWritable, IntWritable, Text 均是 Hadoop 中实现的用于封装 Java 数据类型的类,这些类都能够被串行化从而便于在分布式环境中进行数据交换，你可以将它们分别视为 long, int, String 的替代品

欲运行上面实现的 Mapper 和 Reduce，则需要生成一个 Map-Reduce 得任务(Job)，即初始化 Job 的过程，其基本包括以下三部分：

•输入的数据，也即需要处理的数据

•Map-Reduce 程序，也即上面实现的 Mapper 和 Reducer

•此任务的配置项 JobConf

欲配置 JobConf，需要大致了解 Hadoop 运行 job 的基本原理：

@..Hadoop 将 Job 分成 task 进行处理，共两种 task：map task 和 reduce task

@..Hadoop 有两类的节点控制 job 的运行：JobTracker 和 TaskTracker

JobTracker 协调整个 job 的运行,将 task 分配到不同的 TaskTracker 上 。TaskTracker 负责运行 task，并将结果返回给 JobTracker

@..Hadoop 将输入数据分成固定大小的块，我们称之 input split

@..Hadoop 为每一个 input split 创建一个 task，在此 task 中依次处理此 split 中的一个个记录(record)

@..Hadoop 会尽量让输入数据块所在的 DataNode 和 task 所执行的 DataNode(每个 DataNode 上都有一个 TaskTracker)为同一个，可以提高运行效率，所以 input split 的大小也一般是 HDFS 的 block 的

大小。

@..Reduce task 的输入一般为 Map Task 的输出，Reduce Task 的输出为整个 job 的输出，保存在 HDFS 上。

@..在 reduce 中，相同 key 的所有的记录一定会到同一个 TaskTracker 上面运行，然而不同的 key 可以在不同的 TaskTracker 上面运行，我们称之为 partition（分区）

Partitioner 用于划分键值空间（key space）。

例如：map1 的 output 是：<hello,1> <word,1> <is,1>

map2 的 output 是：<hello,2> <is,2> <the,2>

Partitioner 保证了<hello,1> <hello,2> 放在一个分区了

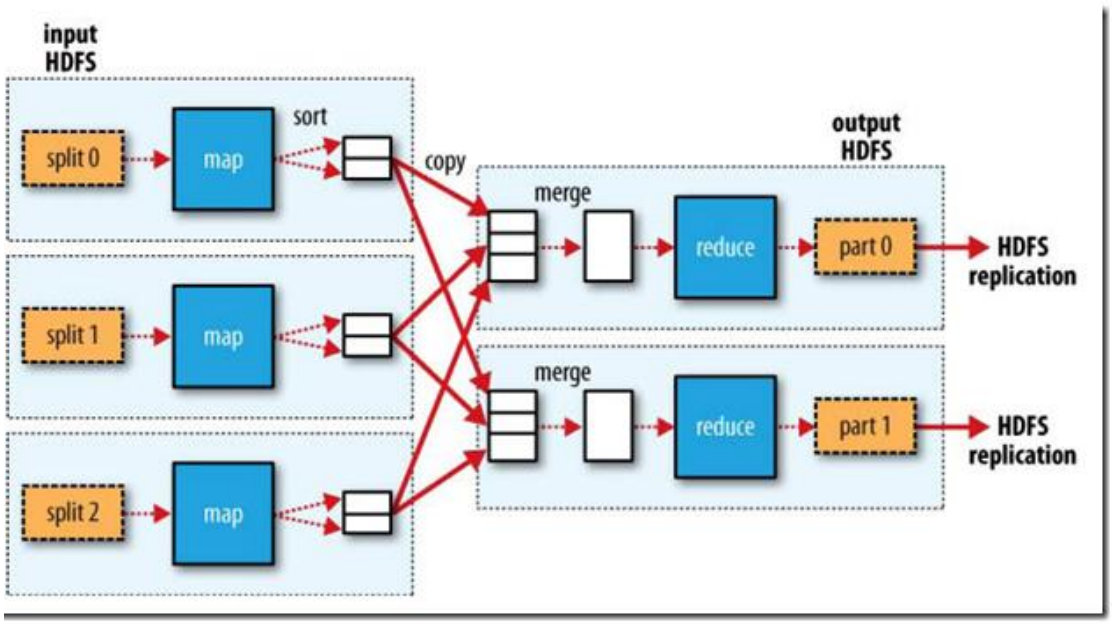
即一个 reduce 里执行

HashPartitioner 是默认的 Partitioner。

partition 的规则为：(K2, V2) -> Integer， 也即根据 K2，生成一个 partition 的 id，具有相同 id 的 K2 则进入同一个 partition，被同一个 TaskTracker 上被同一个 Reducer 进行处理。

```
public interface Partitioner<K2, V2> extends JobConfigurable
{
  int getPartition(K2 key, V2 value, int numPartitions);
}
```

下图大概描述了 Map-Reduce 的 Job 运行的基本原理：



下面我们讨论 JobConf，其有很多的项可以进行配置：

•setInputFormat：设置 map 的输入格式，默认为 TextInputFormat，key 为 LongWritable, value 为 Text

•setNumMapTasks：设置 map 任务的个数，此设置通常不起作用，map 任务的个数取决于输入的数据所能分成的 input split 的个数

•setMapperClass：设置 Mapper，默认为 IdentityMapper

•setMapRunnerClass：设置 MapRunner, map task 是由 MapRunner 运行的，默认为 MapRunnable，其功能为读取 input split 的一个个 record，依次调用 Mapper 的 map 函数

•setMapOutputKeyClass 和 setMapOutputValueClass：设置 Mapper 的输出的 key-value 对的格式

•setOutputKeyClass 和 setOutputValueClass：设置 Reducer 的输出的 key-value 对的格式

•setPartitionerClass 和 setNumReduceTasks：设置 Partitioner，默认为 HashPartitioner，其根据 key 的 hash 值来决定进入哪个 partition,每个 partition 被一个 reduce task 处理,所以 partition 的个数等于 reduce task 的个数

•setReducerClass：设置 Reducer，默认为 IdentityReducer

•setOutputFormat：设置任务的输出格式，默认为 TextOutputFormat

•FileInputFormat.addInputPath：设置输入文件的路径，可以使一个文件，一个路径，一个通配符。可以被调用多次添加多个路径

•FileOutputFormat.setOutputPath：设置输出文件的路径，在 job 运行前此路径不应该存在

当然不用所有的都设置，由上面的例子，可以编写 Map-Reduce 程序如下：

```
public class MaxTemperature {
  public static void main(String[] args) throws IOException {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperature <input path> <output path>");
      System.exit(-1);
    }
    JobConf conf = new JobConf(MaxTemperature.class);
    conf.setJobName("Max temperature");
    FileInputFormat.addInputPath(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    conf.setMapperClass(MaxTemperatureMapper.class);
    conf.setReducerClass(MaxTemperatureReducer.class);
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    JobClient.runJob(conf);
  }
}
```

3、Map-Reduce 数据流(data flow)

Map-Reduce 的处理过程主要涉及以下四个部分：

•客户端 Client：用于提交 MapReduce job

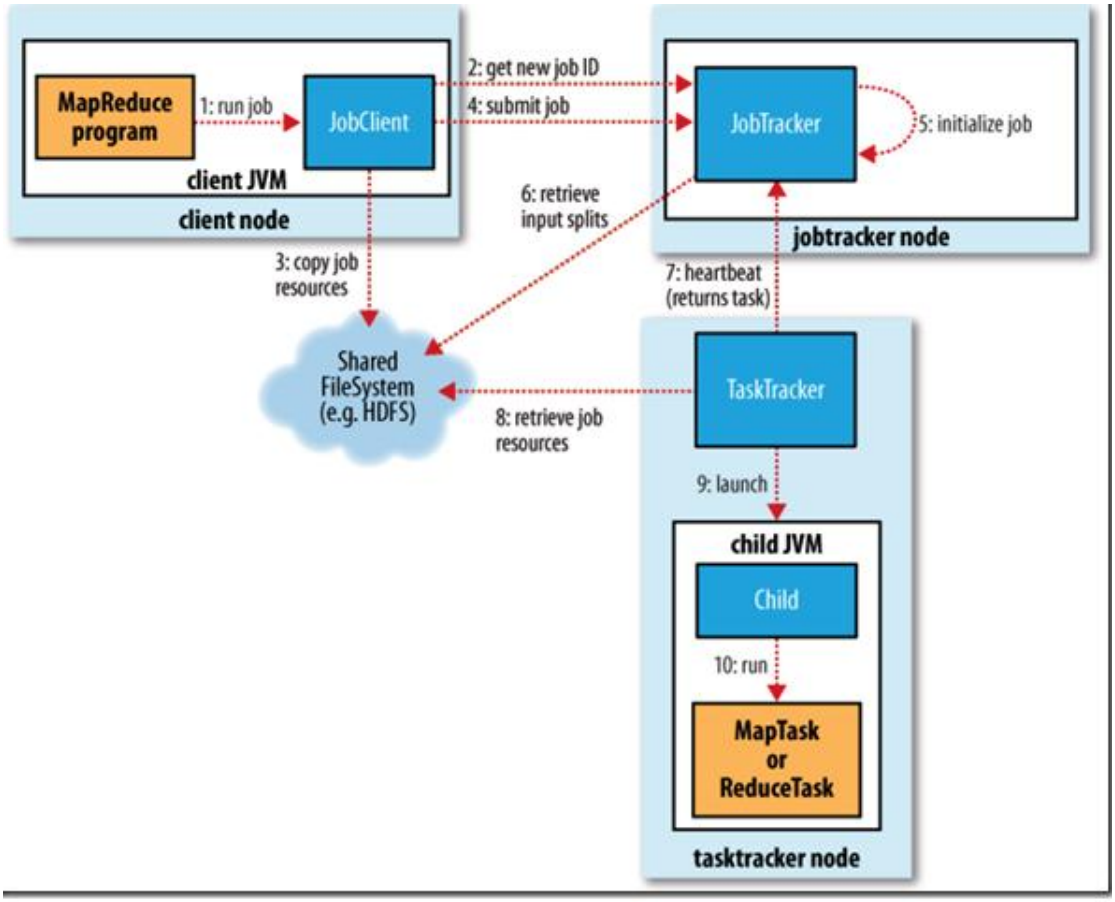
•JobTracker：用户提交作业的服务器，同时，它还负责各个作业任务的分配，管理所有的任务服务器。

•TaskTracker：任劳任怨的工蜂，负责执行具体的任务。

•HDFS：hadoop 分布式文件系统，用于在各个进程间共享 Job 相关的文件

3.1、提交作业

JobClient 的 runJob()方法是用于创建 JobClient 实例和调用其 submitJob()方法的简便方法（参考下图步骤）。



JobClient 的 submitJob()方法所实现作业提交的过程：

1、向 JobTracker 请求一个新的 job ID（通过调用 JobTracker 的 getNewJobId()）

2、检测此 job 的 output 配置 。比如，如果没有指定输出目录或者它已经存在，作业就不会被提交，并有错误返回给 MapReduce 程序

3、计算此 job 的 input splits （即计算作业的输入划分，将作业分成多少块，每块大小是多少等等）。如果输入路径不存在，则划分无法计算，作业就不会被提交，并有错误返回给 MapReduce 程序。

4、复制运行Job所需要的资源到HDFS目录中以job ID命名的jobtracker’s filesystem中。这些资源包括作业JAR文件、配置文件和计算所得的输入划分。作业JAR的副本较多(由

mapred.submit.replication属性控制，默认为10)，这样一来，在taskTracker运行作业任务时，集群能为他们提供许多副本进行访问（步骤3）

5、通知 JobTracker 此 Job 已经可以运行了 (步骤 4)

补充：一个 MapReduce Job 的执行过程

Table 2-1. *Parts of a MapReduce Job*

Part	Handled By
Configuration of the job	User
Input splitting and distribution	Hadoop framework
Start of the individual map tasks with their input split	Hadoop framework
Map function, called once for each input key/value pair	User
Shuffle, which partitions and sorts the per-map output	Hadoop framework
Sort, which merge sorts the shuffle output for each partition of all map outputs	Hadoop framework
Start of the individual reduce tasks, with their input partition	Hadoop framework
Reduce function, which is called once for each unique input key, with all of the input values that share that key	User
Collection of the output and storage in the configured job output directory, in *N* parts, where *N* is the number of reduce tasks	Hadoop framework

前面是 Job 执行的步骤，后面是说这部分归谁负责

1、用户自己配置工作

2、Hadoop 进行分割数据并且分发任务

3、Hadoop 启动 map

4、调用用户自己写 map

5、Hadoop 对 map 的输出结果进行排序，创建分区

6、Hadoop 对 map 排序后的结果进行分组，即使得 key 相同的放在一个分区中，在同一个 reduce 执行

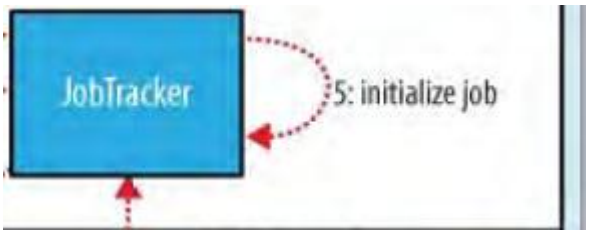
7、启动 map task 其输入数据就是 map 经过处理后的输出，在每个 map 输出分区中

8、执行用户自己写的 reduce，key 是相同的

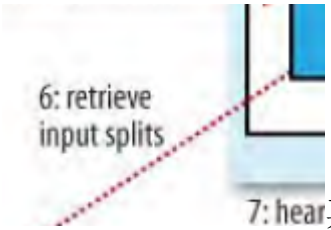
9、收集输出，并存放在配置好的 job 输出目录中（目录中不应该有该文件），n 个 reduce task 放在 n 个文件，都是在一个目录里面的

3.2、作业初始化

当 JobTracker 收到 submitJob 调用的时候，将此作业放到一个队列中，交由 Job 调度器进行调度，并对其初始化。初始化首先创建一个对象来封装 job 运行的 tasks, status 以及 progress（步骤 5）.



在创建 task 之前，job 调度器首先从共享文件系统中获得 JobClient 计算出的 input splits 信息（步骤 6）。

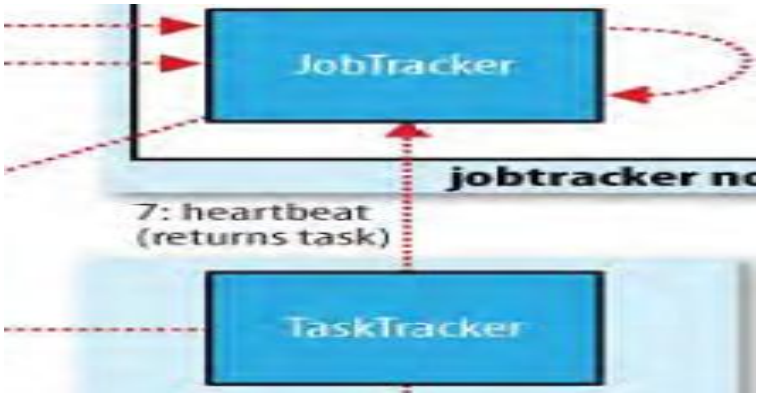


其为每个 input split 创建一个 map task。每个 task 被分配一个 ID。

3.3、任务分配

TaskTracker 周期性的向 JobTracker 发送心跳。告知 JobTracker 它是否存活，同时也充当两者之间的消息通道。可以通过心跳告诉

JobTracker，它是否已经准备运行新的任务，如果是，JobTracker 会为它分配一个任务，并使用心跳方法的返回值于 taskTracker 进行通信（步骤 7）。



在 JobTracker 为 TaskTracker 选择一个 task 之前，JobTracker 必须先选择一个 Job，有很多调度算法（详见 Hadoop 权威指南的第二版第七章），默认的方法是按照 job 的优先级。

3.4、任务的执行

现在 TaskTracker 已经被分配了一个 task，下一步就是要运行此 task。

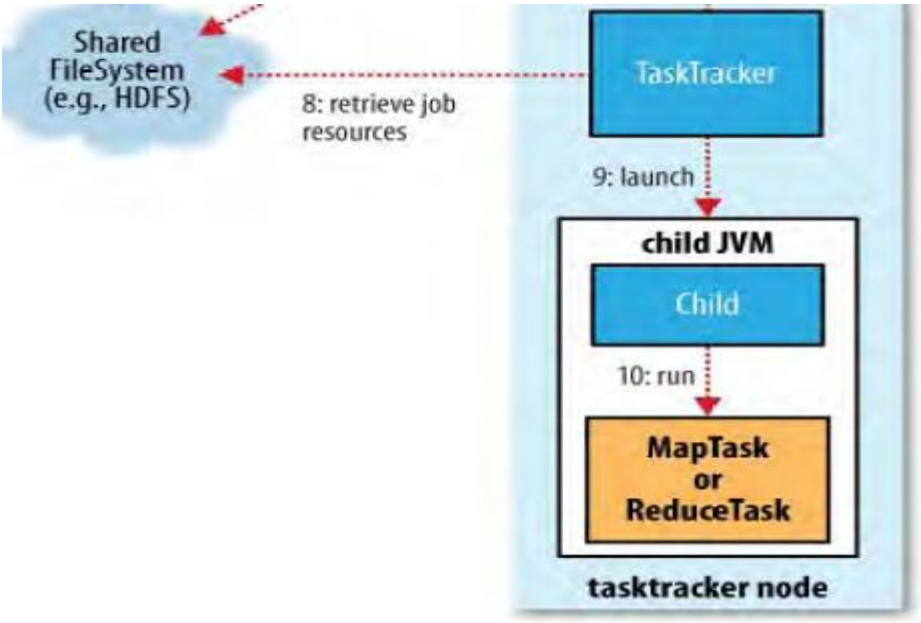
@..首先，将它要执行的本地化作业的 JAR 文件从 HDFS 复制到 TaskTracker 所在的文件系统。同时，将应用程序所需要的全部文件从分布式缓存复制到本地磁盘（详见 Hadoop 权威指南第二版中第 8 章的“分布式缓存”）（步骤 8）。

@..然后，任务新建一个本地工作目录，并把 JAR 文件中的内容解压到这个文件夹下。

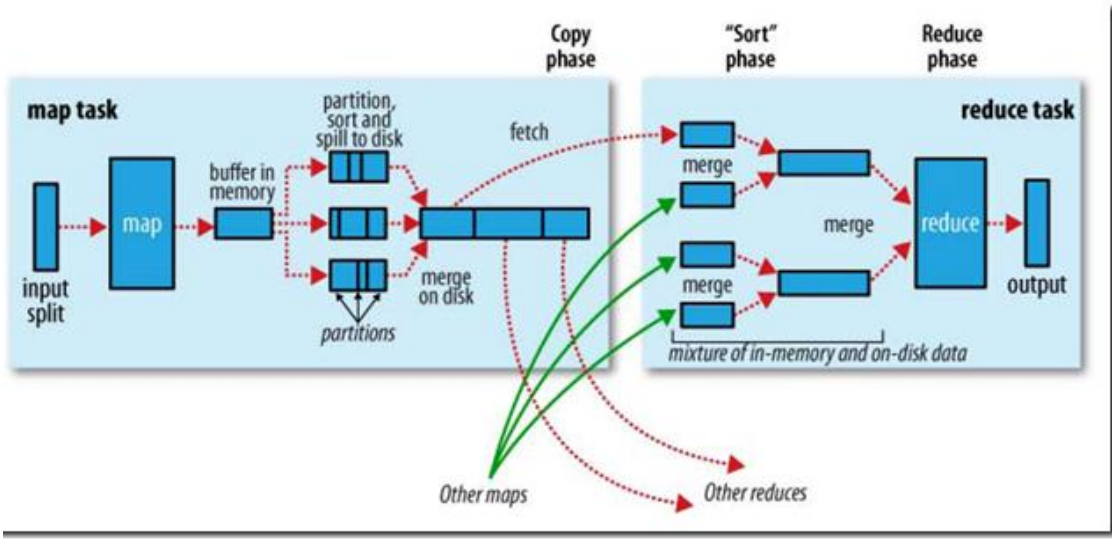
@..第三步，新建一个 TaskRunner 实例来运行任务。

TaskRunner 创建一个新的 JVM（步骤 9）来运行 task（步骤 10）。

被创建的 child JVM 和 TaskTracker 通信来报告运行进度。



3.4.1、Map 的过程



map 从 input split 中读取一个个的 record，然后依次调用 Mapper 的 map 函数，将结果输出。

map 的输出并不是简单的写入硬盘，而是利用缓冲的方式写到内存，并处于效率原因预先进行排序。每个 map 任务都有一个环形内存缓冲区，任务会把输出写到此。默认情况下缓冲区大小为 100MB。当 buffer

中数据的到达一定的大小（默认为 0.80，80%），一个后台线程将数据开始写入硬盘（此时 map 输出继续被写到缓冲区，但如果在此期间缓冲区被填满，map 会阻塞直到写入过程结束）。

在写入硬盘之前，内存中的数据通过 partitioner 分成多个 partition(分区)。

在同一个 partition 中，后台线程会将数据按照 key 在内存进行内中排序。

每次从内存向硬盘 flush 数据，都生成一个新的 spill 文件。

当此 task 结束之前，所有的 spill 文件被合并为一个整的被 partition 的而且排好序的文件。

reducer 可以通过 http 协议请求 map 的输出文件，tracker.http.threads 可以设置 http 服务线程数，默认为 40。

补充：

Map 的输出文件位于运行 map 任务的 TaskTracker 的本地磁盘，而不是 HDFS。

map 的输出作为中间输出，而中间输出则是用来作为 reduce 任务的输入，经 reduce 处理后产生最终的输出，一旦作业完成，map 的输出就可以删除了。（你可能会问 reduce 如何知道从哪个 TaskTracker 取得 map 输出呢？不要心急，后面会给你答案的哦~）

3.4.2、Reduce 的过程

现在，TaskTracker 需要为它分区文件运行 reduce 任务，而 reduce 任务需要其对应的 partition 的所有的 map 输出，这些 map 输出可能是若干个 map 执行的结果。

补充：reduce 如何知道要从那个 TaskTracker 取得 map 输出呢？map 任务成功完成后，它们会通知其父 TaskTracker 状态已更新，然后 TaskTracker 进而通知 JobTracker。这些通知在前面介绍的心跳交流机制中传输。因此，对于指定作业，JobTracker 知道 map 输出和 TaskTracker 之间的映射关系。Reduce 线程定期向 JobTracker 获取 map 输出的位置，直到得到所有输出位置。

Reduce 任务中的 copy 过程即当每个 map 任务结束的时候就开始拷贝输出，因为不同的 map task 完成时间不同。Reduce 任务中有多个 copy 线程，可以并行拷贝 map 输出。当很多 map 输出拷贝到 reduce 任务后，一个背景线程将其合并为一个大的排好序的文件。当所有的 map 输出都拷贝到 reduce task 后，进入 sort 过程，将所有的 map 输出合并为大的排好序的文件。

最后进入 reduce 过程，调用 reducer 的 reduce 函数，处理排好序的输出的每个 key，最后的结果写入 HDFS。

补充：Reducer 有 3 个主要阶段：copy（shuffle）、sort 和 reduce。Shuffle（Shuffle, which partitions and sorts the per-map output）

Reducer 的输入就是 Mapper 已经排好序的输出。在这个阶段，框架通过 HTTP 为每个 Reducer 获得所有 Mapper 输出中与之相关的分块。

Sort（Sort, which merge sorts the shuffle output for each partition of all map outputs）

这个阶段，框架将按照 key 的值对 Reducer 的输入进行分组 （因为不同 mapper 的输出中可能会有相同的 key）。

Shuffle 和 Sort 两个阶段是同时进行的;map 的输出也是一边被取回一边被合并的。

3.5、任务结束

当 JobTracker 获得最后一个 task 的运行成功的报告后，将 job 得状态改为成功。

当 JobClient 从 JobTracker 轮询的时候，发现此 job 已经成功结束，则向用户打印消息，从 runJob 函数中返回。