一、 课程计划

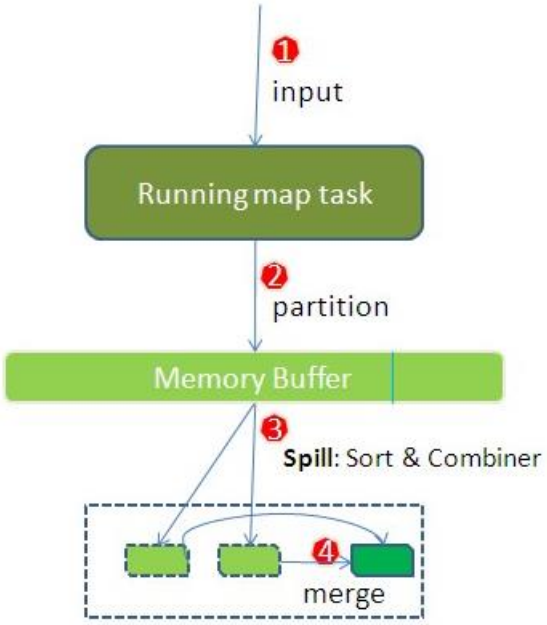
目录

一、 课程计划.....1
二、 MapReduce 工作机制详解.....3
1. MapTask 工作机制.....3
2. ReduceTask 工作机制.....6
3. Shuffle 机制.....8
三、 MapReduce 并行度机制.....10
1. MapTask 并行度机制.....10
2. Reducetask 并行度机制.....12
3. Task 并行度经验之谈.....12
四、 MapReduce 优化参数.....13
1. 资源相关参数.....13
2. 容错相关参数.....14
3. 效率跟稳定性参数.....14
五、 MapReduce 其他功能.....15
1. 计数器应用.....15
2. 多 job 串联.....16
六、 MapReduce 案例.....17
1. 倒排索引.....17
1.1. 实例描述.....17
1.2. 设计思路.....18
1.3. 程序代码.....20
2. 数据去重.....23
2.1. 实例描述.....23
2.2. 设计思路.....24
2.3. 程序代码.....24
3. Top N.....26
3.1. 实例描述.....26
3.2. 设计思路.....26
3.3. 程序代码.....27
七、 Apache Hadoop YARN.....30
1. Yarn 通俗介绍.....30
2. Yarn 基本架构.....31
3. Yarn 三大组件介绍.....31

3.1. ResourceManager 31
3.2. NodeManager 32
3.3. ApplicationMaster 32
4. Yarn 运行流程 33
5. Yarn 调度器 Scheduler 34
5.1. FIFO Scheduler 34
5.2. Capacity Scheduler 35
5.3. Fair Scheduler 36
5.4. 示例：Capacity 调度器配置使用 37

二、 MapReduce 工作机制详解

1. MapTask 工作机制



整个 Map 阶段流程大体如上图所示。简单概述：input File 通过 split 被逻辑切分为多个 split 文件，通过 Record 按行读取内容给 map（用户自己实现的）进行处理，数据被 map 处理结束之后交给 OutputCollector 收集器，对其结果 key 进行分区（默认使用 hash 分区），然后写入 buffer，每个 map task 都有一个内存缓冲区，存储着 map 的输出结果，当缓冲区快满的时候需要将缓冲区的数据以一个临时文件的方式存放到磁盘，当整个 map task 结束后再对磁盘中这个 map task 产生的所有临时文件做合并，生成最终的正式输出文件，然后等待 reduce task 来拉数据。

详细步骤：

- 首先，读取数据组件 InputFormat（默认 TextInputFormat）会通过 getSplits 方法对输入目录中文件进行逻辑切片规划得到 splits，有多少个 split 就对应启动多少个 MapTask。split 与 block 的对应关系默认是一对一。
- 将输入文件切分为 splits 之后，由 RecordReader 对象（默认 LineRecordReader）进行读取，以\n 作为分隔符，读取一行数据，返回<key，value>。Key 表示每行首字符偏移值，value 表示这一行文本内容。

- 读取 split 返回<key,value>，进入用户自己继承的 Mapper 类中，执行用户重写的 map 函数。RecordReader 读取一行这里调用一次。
- map 逻辑完之后，将 map 的每条结果通过 context.write 进行 collect 数据收集。在 collect 中，会先对其进行分区处理，默认使用 HashPartitioner。

MapReduce 提供 Partitioner 接口，它的作用就是根据 key 或 value 及 reduce 的数量来决定当前的这对输出数据最终应该交由哪个 reduce task 处理。默认对 key hash 后再以 reduce task 数量取模。默认的取模方式只是为了平均 reduce 的处理能力，如果用户自己对 Partitioner 有需求，可以订制并设置到 job 上。

- 接下来，会将数据写入内存，内存中这片区域叫做环形缓冲区，缓冲区的作用是批量收集 map 结果，减少磁盘 IO 的影响。我们的 key/value 对以及 Partition 的结果都会被写入缓冲区。当然写入之前，key 与 value 值都会被序列化成字节数组。

环形缓冲区其实是一个数组，数组中存放着 key、value 的序列化数据和 key、value 的元数据信息，包括 partition、key 的起始位置、value 的起始位置以及 value 的长度。环形结构是一个抽象概念。

*缓冲区是有大小限制，默认是 100MB。当 map task 的输出结果很多时，就可能会撑爆内存，所以需要在一定条件下将缓冲区中的数据临时写入磁盘，然后重新利用这块缓冲区。这个从内存往磁盘写数据的过程被称为 Spill，中文可译为溢写。这个溢写是由单独线程来完成，不影响往缓冲区写 map 结果的线程。溢写线程启动时不应该阻止 map 的结果输出，所以整个缓冲区有个溢写的比例 spill.percent。这个比例默认是 0.8，也就是当缓冲区的数据已经达到阈值（buffer size * spill percent = 100MB * 0.8 = 80MB），溢写线程启动，锁定这 80MB 的内存，执行溢写过程。Map task 的输出结果还可以往剩下的 20MB 内存中写，互不影响。*

- 当溢写线程启动后，需要对这 80MB 空间内的 key 做排序(Sort)。排序是 MapReduce 模型默认的行为，这里的排序也是对序列化的字节做的排序。

如果 job 设置过 Combiner，那么现在就是使用 Combiner 的时候了。将有相同 key 的 key/value 对的 value 加起来，减少溢写到磁盘的数据量。Combiner 会优化 MapReduce 的中间结果，所以它在整个模型中会多次使用。

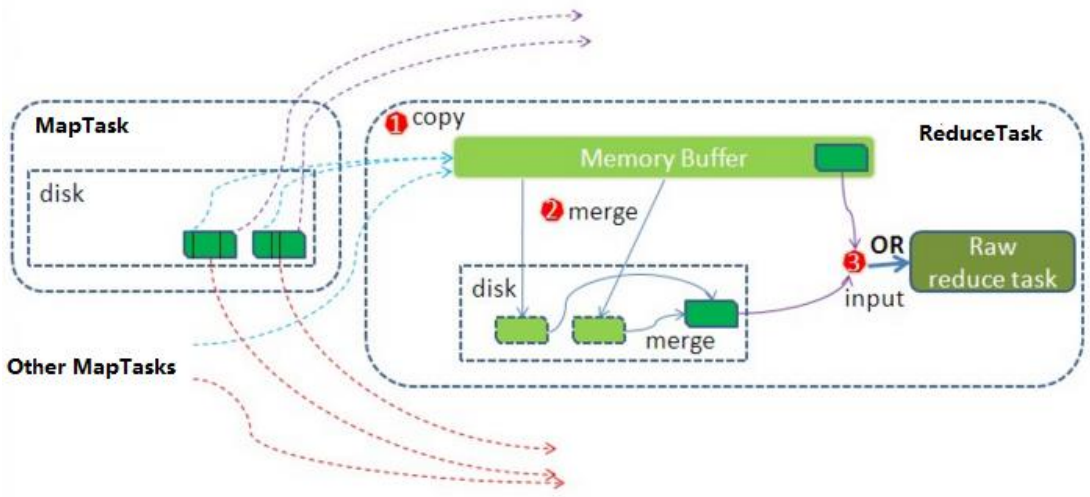
那哪些场景才能使用 Combiner 呢？从这里分析，Combiner 的输出是 Reducer 的输入，Combiner 绝不能改变最终的计算结果。Combiner 只应该用于那种 Reduce 的输入 key/value

与输出 key/value 类型完全一致，且不影响最终结果的场景。比如累加，最大值等。Combiner 的使用一定得慎重，如果用好，它对 job 执行效率有帮助，反之会影响 reduce 的最终结果。

- 每次溢写会在磁盘上生成一个临时文件（写之前判断是否有 combiner），如果 map 的输出结果真的很大，有多次这样的溢写发生，磁盘上相应的就会有多个临时文件存在。当整个数据处理结束之后开始对磁盘中的临时文件进行 merge 合并，因为最终的文件只有一个，写入磁盘，并且为这个文件提供了一个索引文件，以记录每个 reduce 对应数据的偏移量。

至此 map 整个阶段结束。

2. ReduceTask 工作机制



Reduce 大致分为 copy、sort、reduce 三个阶段，重点在前两个阶段。copy 阶段包含一个 eventFetcher 来获取已完成的 map 列表，由 Fetcher 线程去 copy 数据，在此过程中会启动两个 merge 线程，分别为 inMemoryMerger 和 onDiskMerger,分别将内存中的数据 merge 到磁盘和将磁盘中的数据进行 merge。待数据 copy 完成之后，copy 阶段就完成了，开始进行 sort 阶段，sort 阶段主要是执行 finalMerge 操作，纯粹的 sort 阶段，完成之后就是 reduce 阶段，调用用户定义的 reduce 函数进行处理。

详细步骤：

- Copy 阶段,简单地拉取数据。Reduce 进程启动一些数据 copy 线程(Fetcher)，通过 HTTP 方式请求 maptask 获取属于自己的文件。
- Merge 阶段。这里的 merge 如 map 端的 merge 动作，只是数组中存放的是不同 map 端 copy 来的数值。Copy 过来的数据会先放入内存缓冲区中，这里的缓冲区大小要比 map 端的更为灵活。merge 有三种形式：内存到内存；内存到磁盘；磁盘到磁盘。默认情况下第一种形式不启用。当内存中的数据量到达一定阈值，就启动内存到磁盘的 merge。与 map 端类似，这也是溢写的过程，这个过程中如果你设置有 Combiner，也是会启用的，然后在磁盘中生成了众多的溢写文件。第二种 merge 方式一直在运行，直到没有 map 端的数据

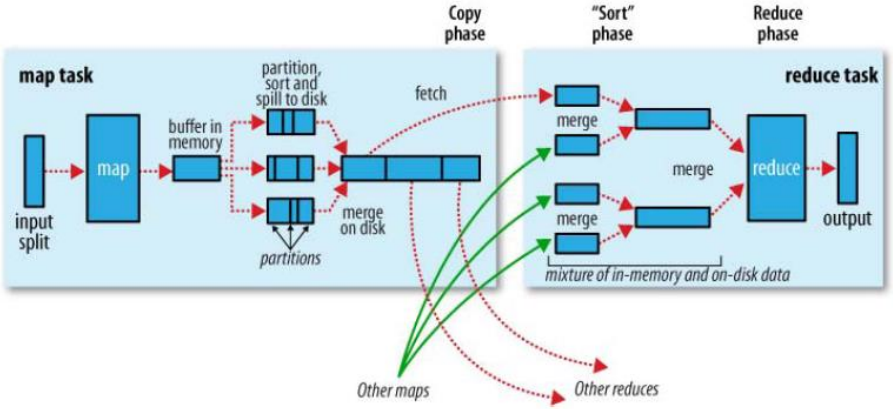
时才结束，然后启动第三种磁盘到磁盘的 merge 方式生成最终的文件。

- 把分散的数据合并成一个大的数据后，还会再对合并后的数据排序。
- 对排序后的键值对调用 reduce 方法，键相等的键值对调用一次 reduce 方法，每次调用会产生零个或者多个键值对，最后把这些输出的键值对写入到 HDFS 文件中。

3. Shuffle 机制

map 阶段处理的数据如何传递给 reduce 阶段，是 MapReduce 框架中最关键的一个流程，这个流程就叫 shuffle。

shuffle：洗牌、发牌——（核心机制：数据分区，排序，合并）。



shuffle 是 Mapreduce 的核心，它分布在 Mapreduce 的 map 阶段和 reduce 阶段。一般把从 Map 产生输出开始到 Reduce 取得数据作为输入之前的过程称作 shuffle。

1).Collect 阶段：将 MapTask 的结果输出到默认大小为 100M 的环形缓冲区，保存的是 key/value，Partition 分区信息等。

2).Spill 阶段：当内存中的数据量达到一定的阀值的时候，就会将数据写入本地磁盘，在将数据写入磁盘之前需要对数据进行一次排序的操作，如果配置了 combiner，还会将有相同分区号和 key 的数据进行排序。

3).Merge 阶段：把所有溢出的临时文件进行一次合并操作，以确保一个 MapTask 最终只产生一个中间数据文件。

4).Copy 阶段： ReduceTask 启动 Fetcher 线程到已经完成 MapTask 的节点上复制一份属于自己的数据，这些数据默认会保存在内存的缓冲区中，当内存的缓冲区达到一定的阀值的时候，就会将数据写到磁盘之上。

5).Merge 阶段：在 ReduceTask 远程复制数据的同时，会在后台开启两个线程对内存到本地的数据文件进行合并操作。

6).Sort 阶段：在对数据进行合并的同时，会进行排序操作，由于 MapTask 阶段已经对数据进行了局部的排序，ReduceTask 只需保证 Copy 的数据的最终整

体有效性即可。

Shuffle 中的缓冲区大小会影响到 mapreduce 程序的执行效率，原则上说，缓冲区越大，磁盘 io 的次数越少，执行速度就越快

缓冲区的大小可以通过参数调整， 参数：io.sort.mb 默认 100M

三、 MapReduce 并行度机制

1. MapTask 并行度机制

MapTask 的并行度指的是 map 阶段有多少个并行的 task 共同处理任务。map 阶段的任务处理并行度，势必影响到整个 job 的处理速度。那么，MapTask 并行实例是否越多越好呢？其并行度又是如何决定呢？

一个 MapReduce job 的 map 阶段并行度由客户端在提交 job 时决定，即客户端提交 job 之前会对待处理数据进行逻辑切片。切片完成会形成切片规划文件（job.split），每个逻辑切片最终对应启动一个 maptask。

逻辑切片机制由 FileInputFormat 实现类的 getSplits()方法完成。

FileInputFormat 切片机制

FileInputFormat 中默认的切片机制：

A. 简单地按照文件的内容长度进行切片

B. 切片大小，默认等于 block 大小

C. 切片时不考虑数据集整体，而是逐个针对每一个文件单独切片

比如待处理数据有两个文件：

file1.txt 320M

file2.txt 10M

经过 FileInputFormat 的切片机制运算后，形成的切片信息如下：

file1.txt.split1—0M~128M

file1.txt.split2—128M~256M

file1.txt.split3—256M~320M

file2.txt.split1—0M~10M

FileInputFormat 中切片的大小的参数配置

在 FileInputFormat 中，计算切片大小的逻辑：

Math.max(minSize, Math.min(maxSize, blockSize));

切片主要由这几个值来运算决定：

minsize：默认值：1

配置参数： mapreduce.input.fileinputformat.split.minsize

maxsize：默认值：Long.MAXValue

配置参数：mapreduce.input.fileinputformat.split.maxsize

blocksize

因此，**默认情况下，split size=block size**,在 hadoop 2.x 中为 128M。

maxsize（切片最大值）：参数如果调得比 blocksize 小，则会让切片变小，而且就等于配置的这个参数的。

minsize （切片最小值）：参数调的比 blockSize 大，则可以让切片变得比 blocksize 还大。

但是，不论怎么调参数，都不能让多个小文件“划入”一个 split。

还有个细节就是：

当 bytesRemaining/splitSize > 1.1 不满足的话，那么最后所有剩余的会作为一个切片。从而不会形成例如 129M 文件规划成两个切片的局面。

2. Reducetask 并行度机制

reducetask 并行度同样影响整个 job 的执行并发度和执行效率，与 maptask 的并发数由切片数决定不同，Reducetask 数量的决定是可以直接手动设置：

```
job.setNumReduceTasks(4);
```

如果数据分布不均匀，就有可能在 reduce 阶段产生数据倾斜。

注意： reducetask 数量并不是任意设置，还要考虑业务逻辑需求，有些情况下，需要计算全局汇总结果，就只能有 1 个 reducetask。

3. Task 并行度经验之谈

最好每个 task 的执行时间至少一分钟。

如果 job 的每个 map 或者 reduce task 的运行时间都只有 30-40 秒钟，那么就减少该 job 的 map 或者 reduce 数，每一个 task(map|reduce)的 setup 和加入到调度器中进行调度，这个中间的过程可能都要花费几秒钟，所以如果每个 task 都非常快就跑完了，就会在 task 的开始和结束的时候浪费太多的时间。

此外，默认情况下，每一个 task 都是一个新的 JVM 实例，都需要开启和销毁的开销。在一些情况下，JVM 开启和销毁的时间可能会比实际处理数据的时间要消耗的长，配置 task 的 JVM 重用可以改善该问题：

（mapred.job.reuse.jvm.num.tasks，默认是 1，表示一个 JVM 上最多可以顺序执行的 task 数目（属于同一个 Job）是 1。也就是说一个 task 启一个 JVM）

如果 input 的文件非常的大，比如 1TB，可以考虑将 hdfs 上的每个 block size 设大，比如设成 256MB 或者 512MB

四、 MapReduce 优化参数

1. 资源相关参数

//以下参数是在用户自己的 MapReduce 应用程序中配置就可以生效

(1) mapreduce.map.memory.mb: 一个 Map Task 可使用的内存上限(单位:MB),默认为 1024。如果 Map Task 实际使用的资源量超过该值，则会被强制杀死。

(2) mapreduce.reduce.memory.mb: 一个 Reduce Task 可使用的资源上限（单位:MB），默认为 1024。如果 Reduce Task 实际使用的资源量超过该值，则会被强制杀死。

(3) mapreduce.map.cpu.vcores: 每个 Maptask 可用的最多 cpu core 数目，默认值: 1

(4) mapreduce.reduce.cpu.vcores: 每个 Reducetask 可用最多 cpu core 数目默认值: 1

(5) mapreduce.map.java.opts: Map Task 的 JVM 参数，你可以在此配置默认的 java heap size 等参数，例如:“-Xmx1024m -verbose:gc -Xloggc:/tmp/@taskid@.gc”（@taskid@会被 Hadoop 框架自动换为相应的 taskid），默认值: “”

(6) mapreduce.reduce.java.opts: Reduce Task 的 JVM 参数，你可以在此配置默认的 java heap size 等参数，例如:“-Xmx1024m -verbose:gc -Xloggc:/tmp/@taskid@.gc”，默认值: “”

//应该在 yarn 启动之前就配置在服务器的配置文件中才能生效

(1) yarn.scheduler.minimum-allocation-mb RM 中每个容器请求的最小配置，以 MB 为单位，默认 1024。

(2) yarn.scheduler.maximum-allocation-mb RM 中每个容器请求的最大分配，以 MB 为单位，默认 8192。

(3) yarn.scheduler.minimum-allocation-vcores 1

(4)yarn.scheduler.maximum-allocation-vcores 32

(5) yarn.nodemanager.resource.memory-mb 表示该节点上 YARN 可使用的物理内存总量，默认是 8192（MB），注意，如果你的节点内存资源不够 8GB，则需要调减小这个值，而 YARN 不会智能的探测节点的物理内存总量。

//shuffle 性能优化的关键参数，应在 yarn 启动之前就配置好

(1) mapreduce.task.io.sort.mb 100 shuffle 的环形缓冲区大小，默认 100m

(2) mapreduce.map.sort.spill.percent 0.8 环形缓冲区溢出的阈值，默认 80%

2. 容错相关参数

(1) mapreduce.map.maxattempts: 每个 Map Task 最大重试次数，一旦重试参数超过该值，则认为 Map Task 运行失败，默认值：4。

(2) mapreduce.reduce.maxattempts: 每个 Reduce Task 最大重试次数，一旦重试参数超过该值，则认为 Map Task 运行失败，默认值：4。

(3) mapreduce.map.failures.maxpercent: 当失败的 Map Task 失败比例超过该值，整个作业则失败，默认值为 0. 如果你的应用程序允许丢弃部分输入数据，则该该值设为一个大于 0 的值，比如 5，表示如果有低于 5%的 Map Task 失败（如果一个 Map Task 重试次数超过 mapreduce.map.maxattempts，则认为这个 Map Task 失败，其对应的输入数据将不会产生任何结果），整个作业扔认为成功。

(4) mapreduce.reduce.failures.maxpercent: 当失败的 Reduce Task 失败比例超过该值为，整个作业则失败，默认值为 0.

(5) mapreduce.task.timeout:如果一个 task 在一定时间内没有任何进入，即不会读取新的数据，也没有输出数据，则认为该 task 处于 block 状态，可能是临时卡住，也许永远会卡住。为了防止因为用户程序永远 block 不退出，则强制设置了一个超时时间（单位毫秒），默认是 600000，值为 0 将禁用超时。。

3. 效率跟稳定性参数

(1) mapreduce.map.speculative: 是否为 Map Task 打开推测执行机制，默认为 true, 如果为 true，则可以并行执行一些 Map 任务的多个实例。

(2) mapreduce.reduce.speculative: 是否为 Reduce Task 打开推测执行机制，默认为 true

(3) mapreduce.input.fileinputformat.split.minsize: FileInputFormat 做切片时最小切片大小，默认 1。

(5) mapreduce.input.fileinputformat.split.maxsize: FileInputFormat 做切片时最大切片大小

推测执行机制（Speculative Execution）：它根据一定的法则推测出“拖后腿”的任务，并为这样的任务启动一个备份任务，让该任务与原始任务同时处理同一份数据，并最终选用最先成功运行完成任务的计算结果作为最终结果。

五、 MapReduce 其他功能

1. 计数器应用

计数器是用来记录 job 的执行进度和状态的。MapReduce 计数器（Counter）为我们提供一个窗口，用于观察 MapReduce Job 运行期的各种细节数据。对 MapReduce 性能调优很有帮助，MapReduce 性能优化的评估大部分都是基于这些 Counter 的数值表现出来的。

MapReduce 自带了许多默认 Counter。在执行 mr 程序的日志上，大家也许注意到了类似以下这样的信息：

```
Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        WRONG_REDUCE=0
File Input Format Counters
        Bytes Read=89
File Output Format Counters
        Bytes Written=86
```

内置计数器包括：

文件系统计数器（File System Counters）

作业计数器（Job Counters）

MapReduce 框架计数器（Map-Reduce Framework）

Shuffle 错误计数器（Shuffle Errors）

文件输入格式计数器（File Output Format Counters）

文件输出格式计数器（File Input Format Counters）

当然， Hadoop 也支持自定义计数器。在实际生产代码中，常常需要将数据处理过程中遇到的不合规数据行进行全局计数，类似这种需求可以借助 mapreduce 框架中提供的全局计数器来实现。

示例代码如下：

```
public class WordCount{
```

```
    static class WordCount Mapper extends Mapper<LongWritable, Text, Text, LongWritable> {
        @Override
        protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
            Counter counter =context.getCounter("SelfCounters","myCounters");
            String[] words = value.toString().split(",");
            for (String word : words) {
                if("hello".equals(word)){counter.increment(1)};
                context.write(new Text(word), new LongWritable(1));
            }
        }
    }
```

2. 多 job 串联

一个稍复杂点的处理逻辑往往需要多个 mapreduce 程序串联处理，多 job 的串联可以借助 mapreduce 框架的 JobControl 实现

示例代码：

```
        ControlledJob controlledJob1 = new ControlledJob(job1.getConfiguration());
        controlledJob1.setJob(job1);
        ControlledJob controlledJob2 = new ControlledJob(job2.getConfiguration());
        controlledJob2.setJob(job2);
        controlledJob2.addDependingJob(controlledJob1);  // job2 依赖于 job1
        JobControl jc = new JobControl(chainName);
        jc.addJob(controlledJob1);
        jc.addJob(controlledJob2);
        Thread jcThread = new Thread(jc);
        jcThread.start();
        while(true){
            if(jc.allFinished()){
                System.out.println(jc.getSuccessfulJobList());
                jc.stop();
                return 0;
            }
            if(jc.getFailedJobList().size() > 0){
                System.out.println(jc.getFailedJobList());
                jc.stop();
                return 1;
            }
        }
```

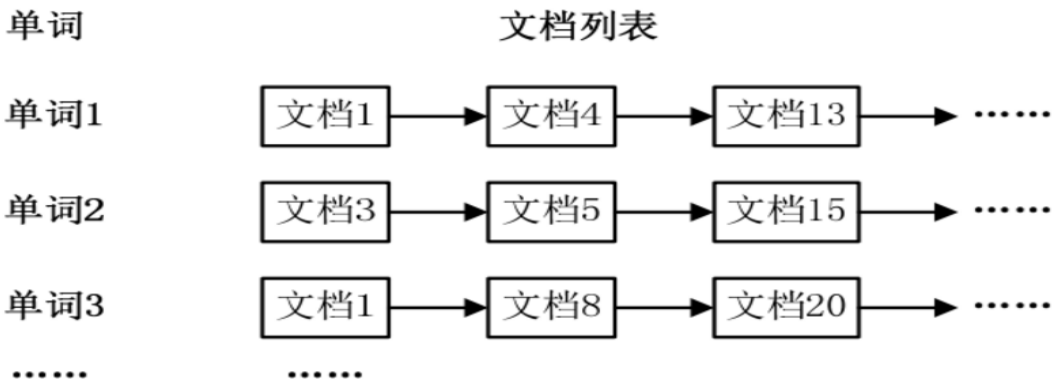
六、 MapReduce 案例

1. 倒排索引

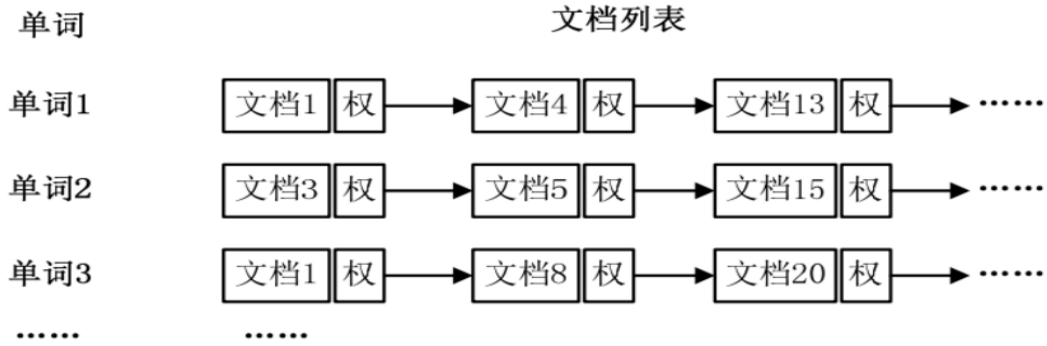
倒排索引是文档检索系统中最常用的数据结构，被广泛地应用于全文搜索引擎。 它主要是用来存储某个单词（或词组） 在一个文档或一组文档中的存储位置的映射，即提供了一种根据内容来查找文档的方式。由于不是根据文档来确定文档所包含的内容，而是进行相反的操作，因而称为倒排索引（ Inverted Index）。

1.1. 实例描述

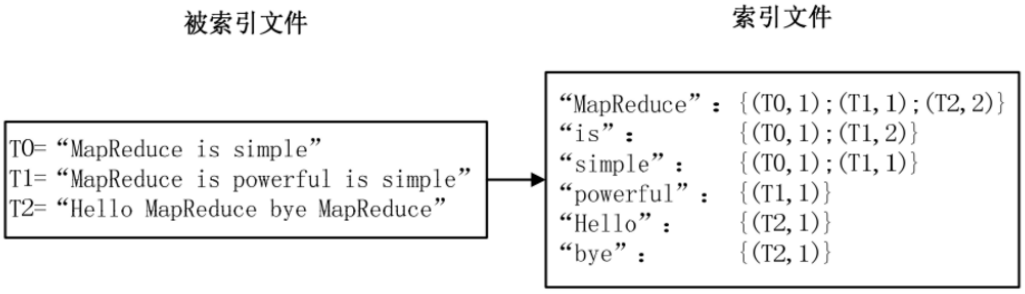
通常情况下，倒排索引由一个单词（或词组）以及相关的文档列表组成，文档列表中的文档或者是标识文档的 ID 号，或者是指文档所在位置的 URL。如下图所示：



从上图可以看出，单词 1 出现在{文档 1，文档 4，文档 13， ……}中，单词 2 出现在{文档 3，文档 5，文档 15， ……}中，而单词 3 出现在{文档 1，文档 8，文档 20， ……}中。在实际应用中，还需要给每个文档添加一个权值，用来指出每个文档与搜索内容的相关度，如下图所示：



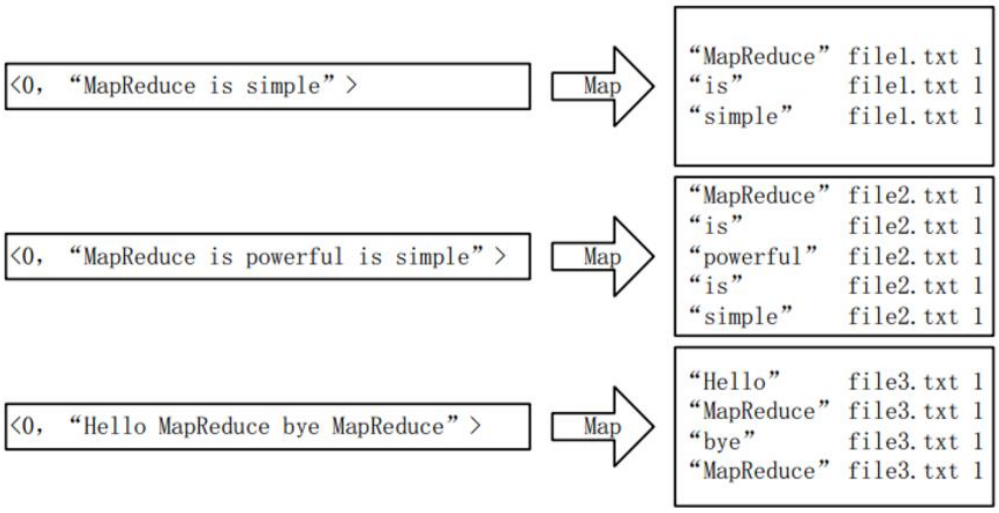
最常用的是使用词频作为权重，即记录单词在文档中出现的次数。以英文为例，如下图所示，索引文件中的“ MapReduce”一行表示：“ MapReduce”这个单词在文本 T0 中 出现过 1 次,T1 中出现过 1 次,T2 中出现过 2 次。当搜索条件为“ MapReduce”、“ is”、“ Simple” 时，对应的集合为： {T0， T1， T2}∩{T0， T1}∩{T0， T1}={T0， T1}，即文档 T0 和 T1 包 含了所要索引的单词，而且只有 T0 是连续的。



1.2. 设计思路

1）Map 过程

首先使用默认的 TextInputFormat 类对输入文件进行处理，得到文本中每行的偏移量及其内容。显然， Map 过程首先必须分析输入的<key,value>对，得到倒排索引中需要的三个信息：单词、文档 URL 和词频，如下图所示。



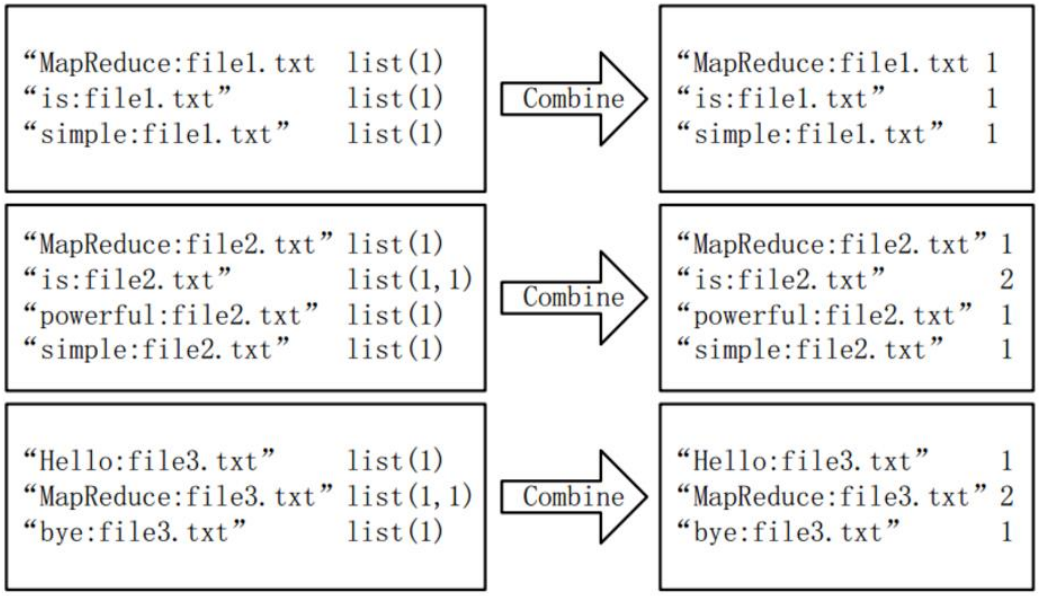
这里存在两个问题：第一， <key,value>对只能有两个值，在不使用 Hadoop 自定义数据类型的情况下，需要根据情况将其中两个值合并成一个值，作为 key 或 value 值；

第二，通过一个 Reduce 过程无法同时完成词频统计和生成文档列表，所以必须增加一个 Combine 过程完成词频统计。

这里将单词和 URL 组成 key 值（如“ MapReduce： file1.txt”），将词频作为 value，这样做的好处是可以利用 MapReduce 框架自带的 Map 端排序，将同一文档的相同单词的词频组成列表，传递给 Combine 过程，实现类似于 WordCount 的功能。

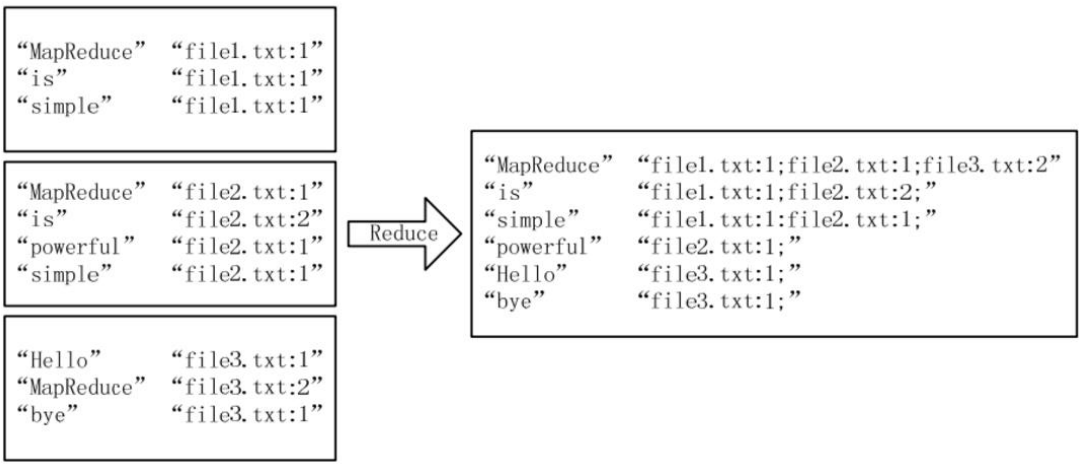
2）Combine 过程

经过 map 方法处理后， Combine 过程将 key 值相同 value 值累加，得到一个单词在文档中的词频。 如果直接将图所示的输出作为 Reduce 过程的输入，在 Shuffle 过程时将面临一个问题：所有具有相同单词的记录（由单词、 URL 和词频组成）应该交由同一个 Reducer 处理，但当前的 key 值无法保证这一点，所以必须修改 key 值和 value 值。这次将单词作为 key 值， URL 和词频组成 value 值（如“ file1.txt： 1”）。这样做的好处是可以利用 MapReduce 框架默认的 HashPartitioner 类完成 Shuffle 过程，将相同单词的所有记录发送给同一个 Reducer 进行处理。



3）Reduce 过程

经过上述两个过程后， Reduce 过程只需将相同 key 值的 value 值组合成倒排索引文件所需的格式即可，剩下的事情就可以直接交给 MapReduce 框架进行处理了。



1.3. 程序代码

InvertedIndexMapper：

```
public class InvertedIndexMapper extends Mapper<LongWritable, Text, Text, Text> {

    private static Text keyInfo = new Text();// 存储单词和 URL 组合
    private static final Text valueInfo = new Text("1");// 存储词频,初始化为 1

    @Override
    protected void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {

        String line = value.toString();
        String[] fields = StringUtils.split(line, " ");// 得到字段数组

        FileSplit fileSplit = (FileSplit) context.getInputSplit();// 得到这行数据所在的文件切
片
        String fileName = fileSplit.getPath().getName();// 根据文件切片得到文件名

        for (String field : fields) {
            // key 值由单词和 URL 组成，如“MapReduce:file1”
            keyInfo.set(field + ":" + fileName);
            context.write(keyInfo, valueInfo);
        }
    }
}
```

InvertedIndexCombiner：

```
public class InvertedIndexCombiner extends Reducer<Text, Text, Text, Text> {

	private static Text info = new Text();

	// 输入：  <MapReduce:file3 {1,1,...}>
	// 输出：<MapReduce file3:2>
	@Override
	protected void reduce(Text key, Iterable<Text> values, Context context)
			throws IOException, InterruptedException {
		int sum = 0;// 统计词频
		for (Text value : values) {
			sum += Integer.parseInt(value.toString());
		}

		int splitIndex = key.toString().indexOf(":");
		// 重新设置 value 值由 URL 和词频组成
		info.set(key.toString().substring(splitIndex + 1) + ":" + sum);
		// 重新设置 key 值为单词
		key.set(key.toString().substring(0, splitIndex));

		context.write(key, info);
	}
}
```

InvertedIndexReducer：

```
public class InvertedIndexReducer extends Reducer<Text, Text, Text, Text> {

	private static Text result = new Text();

	// 输入：<MapReduce file3:2>
	// 输出：<MapReduce file1:1;file2:1;file3:2;>
	@Override
	protected void reduce(Text key, Iterable<Text> values, Context context)
			throws IOException, InterruptedException {
		// 生成文档列表
		String fileList = new String();
		for (Text value : values) {
			fileList += value.toString() + ";";
```

```
            }

            result.set(fileList);
            context.write(key, result);
        }
}
```

InvertedIndexRunner：

```
public class InvertedIndexRunner {
        public static void main(String[] args) throws IOException,
                ClassNotFoundException, InterruptedException {
            Configuration conf = new Configuration();
            Job job = Job.getInstance(conf);

            job.setJarByClass(InvertedIndexRunner.class);

            job.setMapperClass(InvertedIndexMapper.class);
            job.setCombinerClass(InvertedIndexCombiner.class);
            job.setReducerClass(InvertedIndexReducer.class);

            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(Text.class);

            FileInputFormat.setInputPaths(job, new Path(args[0]));
            // 检查参数所指定的输出路径是否存在，若存在，先删除
            Path output = new Path(args[1]);
            FileSystem fs = FileSystem.get(conf);
            if (fs.exists(output)) {
                fs.delete(output, true);
            }
            FileOutputFormat.setOutputPath(job, output);

            System.exit(job.waitForCompletion(true) ? 0 : 1);
        }
}
```

2. 数据去重

数据去重主要是为了掌握和利用并行化思想来对数据进行有意义的筛选。统计大数据集上的数据种类个数、从网站日志中计算访问地等这些看似庞杂的任务都会涉及数据去重。

2.1. 实例描述

对数据文件中的数据进行去重。数据文件中的每行都是一个数据。比如原始输入数据为：

File1:

2017-3-1 a

2017-3-2 b

2017-3-3 c

2017-3-4 d

2017-3-5 a

2017-3-6 b

2017-3-7 c

2017-3-3 c

File2：

2017-3-1 b

2017-3-2 a

2017-3-3 b

2017-3-4 d

2017-3-5 a

2017-3-6 c

2017-3-7 d

2017-3-3 c

输出结果为:

2017-3-1 a

2017-3-1 b

2017-3-2 a

2017-3-2 b

2017-3-3 b

2017-3-3 c

2017-3-4 d

2017-3-5 a

2017-3-6 b

2017-3-6 c

2017-3-7 c

2017-3-7 d

2.2. 设计思路

数据去重的最终目标是让原始数据中出现次数超过一次的数据在输出文件中只出现一次。我们自然而然会想到将同一个数据的所有记录都交给一台 reduce 机器，无论这个数据出现多少次，只要在最终结果中输出一次就可以了。具体就是 reduce 的输入应该以数据作为 key，而对 value-list 则没有要求。当 reduce 接收到一个<key， value-list>时就直接将 key 复制到输出的 key 中，并将 value 设置成空值。

在 MapReduce 流程中， map 的输出<key，value>经过 shuffle 过程聚集成<key，value-list>后会交给 reduce。所以从设计好的 reduce 输入可以反推出 map 的输出 key 应为数据， value 任意。继续反推， map 输出数据的 key 为数据，而在这个实例中每个数据代表输入文件中的一行内容，所以 map 阶段要完成的任务就是在采用 Hadoop 默认的作业输入方式之后，将 value 设置为 key，并直接输出（输出中的 value 任意）。 map 中的结果经过 shuffle 过程之后交给 reduce。 reduce 阶段不会管每个 key 有多少个 value，它直接将输入的 key 复制为输出的 key，并输出就可以了（输出中的 value 被设置成空了）。

2.3. 程序代码

Mapper：

```
public class DedupMapper extends Mapper<LongWritable, Text, Text, NullWritable> {
    private static Text field = new Text();
```

```
    @Override
    protected void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        field = value;
        context.write(field, NullWritable.get());
    }
}
```

Reducer：

```
public class DedupReducer extends
        Reducer<Text, NullWritable, Text, NullWritable> {
    @Override
    protected void reduce(Text key, Iterable<NullWritable> values,
            Context context) throws IOException, InterruptedException {
        context.write(key, NullWritable.get());
    }
}
```

Runner：

```
public class DedupRunner {
    public static void main(String[] args) throws IOException,
            ClassNotFoundException, InterruptedException {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);
        job.setJarByClass(DedupRunner.class);

        job.setMapperClass(DedupMapper.class);
        job.setReducerClass(DedupReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(NullWritable.class);

        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.waitForCompletion(true);
    }
}
```

3． Top N

Top-N 分析法是指从研究对象中得到所需的 N 个数据，并对这 N 个数据进行重点分析的方法。那么该如何利用 MapReduce 来解决在海量数据中求 Top N 个数。

3.1． 实例描述

对数据文件中的数据取最大 top-n。数据文件中的每个都是一个数据。

原始输入数据为：

10 3 8 7 6 5 1 2 9 4

11 12 17 14 15 20

19 18 13 16

输出结果为（最大的前 5 个）：

20

19

18

17

16

3.2． 设计思路

要找出 top N，核心是能够想到 reduce Task 个数一定只有一个。

因为一个 map task 就是一个进程,有几个 map task 就有几个中间文件，有几个 reduce task 就有几个最终输出文件。我们要找的 top N 是指的全局的前 N 条数据，那么不管中间有几个 map，reduce 最终只能有一个 reduce 来汇总数据，输出 top N。

Mapper 过程

使用默认的 mapper 数据，一个 input split（输入分片）由一个 mapper 来处理。

在每一个 map task 中，我们找到这个 input split 的前 n 个记录。这里我们用 TreeMap 这个数据结构来保存 top n 的数据，TreeMap 默认按键的自然顺序升序进行排序。下一步，我们来加入新记录到 TreeMap 中去。在 map 中，我们对每一条记录都尝试去更新 TreeMap，

最后我们得到的就是这个分片中的 local top n 的 n 个值。

以往的 mapper 中，我们都是处理一条数据之后就 context.write 一次。而在这里是把所有这个 input split 的数据处理完之后再进行写入。所以，我们可以把这个 context.write 放在 cleanup 里执行。cleanup 就是整个 mapper task 执行完之后会执行的一个函数。

TreeMap 是一个有序的 key-value 集合，默认会根据其键的自然顺序进行排序，也可根据创建映射时提供的 Comparator 进行排序。其 firstKey()方法用于返回当前这个集合第一个(最低)键。

Reducer 过程

只有一个 reducer，就是对 mapper 输出的数据进行再一次汇总，选出其中的 top n，即可达到我们的目的。注意的是，Treemap 默认是正序排列数据，要想满足求取 top n 倒序最大的 n 个，需要实现自己的 Comparator（）方法。

3.3. 程序代码

TopNMapper：

```
private TreeMap<Integer, String> repToRecordMap = new TreeMap<Integer, String>();

    @Override
    public void map(LongWritable key, Text value, Context context) {

        String line = value.toString();
        String[] nums = line.split(" ");
        for (String num : nums) {
            repToRecordMap.put(Integer.parseInt(num), " ");
            if (repToRecordMap.size() > 5) {
                repToRecordMap.remove(repToRecordMap.firstKey());
            }
        }

    @Override
    protected void cleanup(Context context) {
        for (Integer i : repToRecordMap.keySet()) {
            try {
                context.write(NullWritable.get(), new IntWritable(i));
            } catch (Exception e) {
```

```
                    e.printStackTrace();
                }
            }
        }
```

TopNReducer：

```
private TreeMap<Integer, String> repToRecordMap = new TreeMap<Integer, String>(new Comparator<Integer>() {
        /*
        * int compare(Object o1, Object o2) 返回一个基本类型的整型，
        * 返回负数表示：o1 小于 o2，
        * 返回 0 表示：o1 和 o2 相等，
        * 返回正数表示：o1 大于 o2。
        * 谁大谁排后面
        */
        public int compare(Integer a, Integer b) {
            return b - a;
        }
    });

    public void reduce(NullWritable key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
        for (IntWritable value : values) {
            repToRecordMap.put(value.get(), " ");
            if (repToRecordMap.size() > 5) {
                repToRecordMap.remove(repToRecordMap.firstKey());
            }
        }
        for (Integer i : repToRecordMap.keySet()) {
            context.write(NullWritable.get(), new IntWritable(i));
        }
    }
```

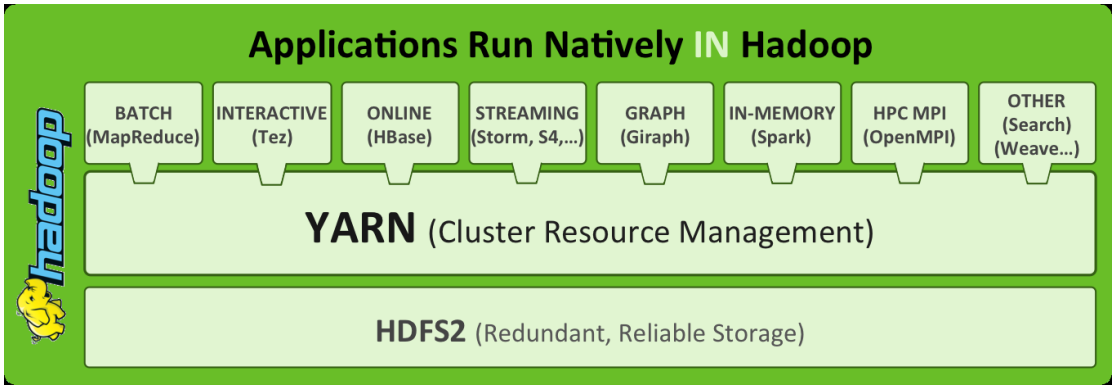
TopNRunner：

```
Configuration conf = new Configuration();

        Job job = Job.getInstance(conf);

        job.setJarByClass(TopNRunner.class);
        job.setMapperClass(TopNMapper.class);
```

```
job.setReducerClass(TopNReducer.class);

job.setNumReduceTasks(1);

job.setMapOutputKeyClass(NullWritable.class);// map 阶段的输出的 key
job.setMapOutputValueClass(IntWritable.class);// map 阶段的输出的 value

job.setOutputKeyClass(NullWritable.class);// reduce 阶段的输出的 key
job.setOutputValueClass(IntWritable.class);// reduce 阶段的输出的 value

FileInputFormat.setInputPaths(job, new Path("D:\\topN\\input"));
FileOutputFormat.setOutputPath(job, new Path("D:\\topN\\output"));

boolean res = job.waitForCompletion(true);
System.exit(res ? 0 : 1);
```

七、 Apache Hadoop YARN

1. Yarn 通俗介绍

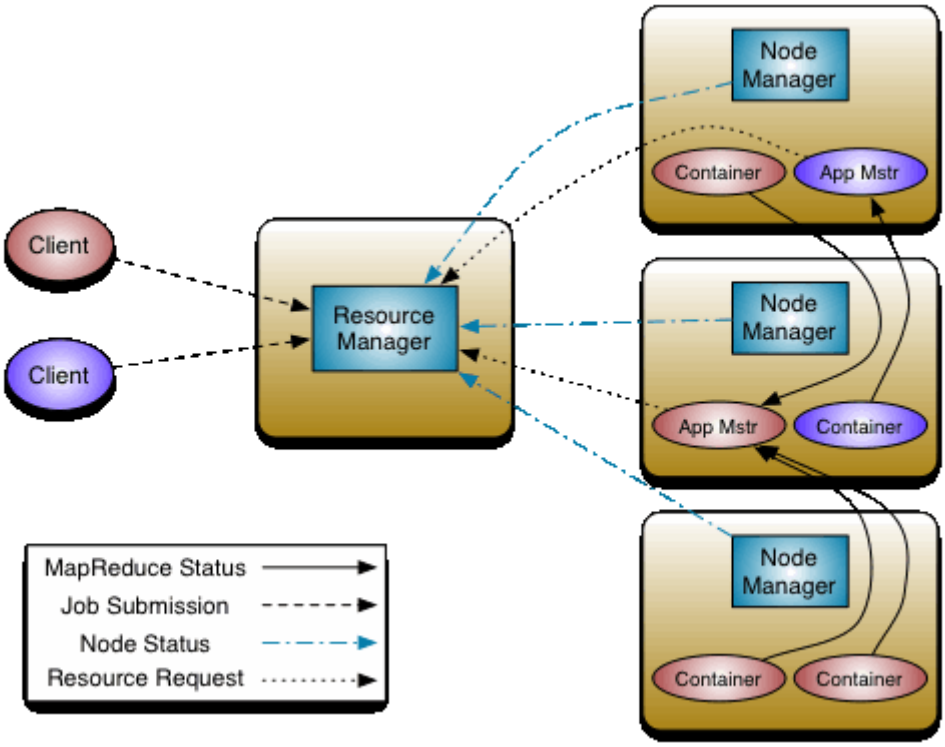


Apache Hadoop YARN （Yet Another Resource Negotiator，另一种资源协调者）是一种新的 Hadoop 资源管理器，它是一个通用资源管理系统和调度平台，可为上层应用提供统一的资源管理和调度，它的引入为集群在利用率、资源统一管理和数据共享等方面带来了巨大好处。

可以把 yarn 理解为相当于一个分布式的操作系统平台，而 mapreduce 等运算程序则相当于运行于操作系统之上的应用程序，Yarn 为这些程序提供运算所需的资源（内存、cpu）。

- yarn 并不清楚用户提交的程序的运行机制
- yarn 只提供运算资源的调度（用户程序向 yarn 申请资源，yarn 就负责分配资源）
- yarn 中的主管角色叫 ResourceManager
- yarn 中具体提供运算资源的角色叫 NodeManager
- yarn 与运行的用户程序完全解耦，意味着 yarn 上可以运行各种类型的分布式运算程序，比如 mapreduce、storm，spark，tez ……
- spark、storm 等运算框架都可以整合在 yarn 上运行，只要他们各自的框架中有符合 yarn 规范的资源请求机制即可
- yarn 成为一个通用的资源调度平台.企业中以前存在的各种运算集群都可以整合在一个物理集群上，提高资源利用率，方便数据共享

2. Yarn 基本架构



YARN 是一个资源管理、任务调度的框架，主要包含三大模块：ResourceManager（RM）、NodeManager（NM）、ApplicationMaster（AM）。

ResourceManager 负责所有资源的监控、分配和管理；

ApplicationMaster 负责每一个具体应用程序的调度和协调；

NodeManager 负责每一个节点的维护。

对于所有的 applications，RM 拥有绝对的控制权和对资源的分配权。而每个 AM 则会和 RM 协商资源，同时和 NodeManager 通信来执行和监控 task。

3. Yarn 三大组件介绍

3.1. ResourceManager

- ResourceManager 负责整个集群的资源管理和分配，是一个全局的资源管理系统。
- NodeManager 以心跳的方式向 ResourceManager 汇报资源使用情况（目前主要是 CPU 和内存的使用情况）。RM 只接受 NM 的资源回报信息，对于具体的资源处理则交给 NM 自己处理。

- YARN Scheduler 根据 application 的请求为其分配资源，不负责 application job 的监控、追踪、运行状态反馈、启动等工作。

3.2. NodeManager

- NodeManager 是每个节点上的资源和任务管理器，它是管理这台机器的代理，负责该节点程序的运行，以及该节点资源的管理和监控。YARN 集群每个节点都运行一个 NodeManager。
- NodeManager 定时向 ResourceManager 汇报本节点资源（CPU、内存）的使用情况和 Container 的运行状态。当 ResourceManager 宕机时 NodeManager 自动连接 RM 备用节点。
- NodeManager 接收并处理来自 ApplicationMaster 的 Container 启动、停止等各种请求。

3.3. ApplicationMaster

- 用户提交的每个应用程序均包含一个 ApplicationMaster，它可以运行在 ResourceManager 以外的机器上。
- 负责与 RM 调度器协商以获取资源（用 Container 表示）。
- 将得到的任务进一步分配给内部的任务(资源的二次分配)。
- 与 NM 通信以启动/停止任务。
- 监控所有任务运行状态，并在任务运行失败时重新为任务申请资源以重启任务。
- 当前 YARN 自带了两个 ApplicationMaster 实现，一个是用于演示 AM 编写方法的实例程序 DistributedShell，它可以申请一定数目的 Container 以并行运行一个 Shell 命令或者 Shell 脚本；另一个是运行 MapReduce 应用程序的 AM—MRAppMaster。

注：RM 只负责监控 AM，并在 AM 运行失败时候启动它。RM 不负责 AM 内部任务的容错，任务的容错由 AM 完成。

4. Yarn 运行流程

- client 向 RM 提交应用程序，其中包括启动该应用的 ApplicationMaster 的必须信息，例如 ApplicationMaster 程序、启动 ApplicationMaster 的命令、用户程序等。
- ResourceManager 启动一个 container 用于运行 ApplicationMaster。
- 启动中的 ApplicationMaster 向 ResourceManager 注册自己，启动成功后与 RM 保持心跳。
- ApplicationMaster 向 ResourceManager 发送请求，申请相应数目的 container。
- ResourceManager 返回 ApplicationMaster 的申请的 containers 信息。申请成功的 container，由 ApplicationMaster 进行初始化。container 的启动信息初始化后，AM 与对应的 NodeManager 通信，要求 NM 启动 container。AM 与 NM 保持心跳，从而对 NM 上运行的任务进行监控和管理。
- container 运行期间，ApplicationMaster 对 container 进行监控。container 通过 RPC 协议向对应的 AM 汇报自己的进度和状态等信息。
- 应用运行期间，client 直接与 AM 通信获取应用的状态、进度更新等信息。
- 应用运行结束后，ApplicationMaster 向 ResourceManager 注销自己，并允许属于它的 container 被收回。

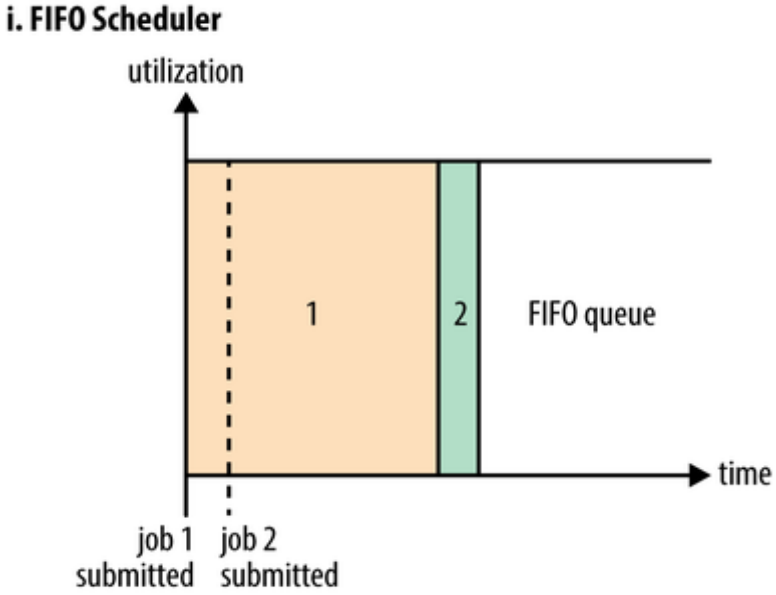
5. Yarn 调度器 Scheduler

理想情况下，我们应用对 Yarn 资源的请求应该立刻得到满足，但现实情况资源往往是有限的，特别是在一个很繁忙的集群，一个应用资源的请求经常需要等待一段时间才能的到相应的资源。在 Yarn 中，负责给应用分配资源的就是 Scheduler。其实调度本身就是一个难题，很难找到一个完美的策略可以解决所有的应用场景。为此，Yarn 提供了多种调度器和可配置的策略供我们选择。

在 Yarn 中有三种调度器可以选择：FIFO Scheduler ，Capacity Scheduler，Fair Scheduler。

5.1. FIFO Scheduler

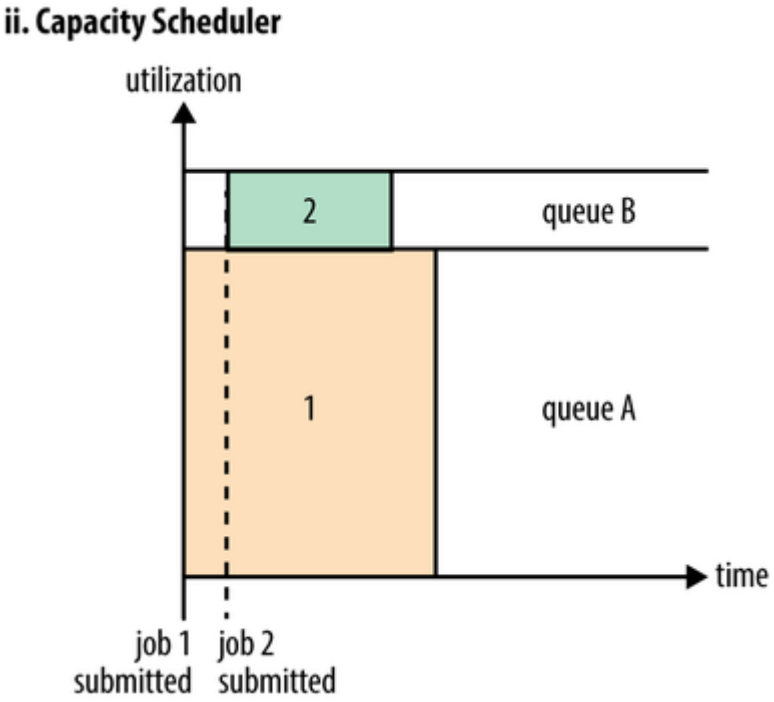
FIFO Scheduler 把应用按提交的顺序排成一个队列，这是一个先进先出队列，在进行资源分配的时候，先给队列中最头上的应用进行分配资源，待最头上的应用需求满足后再给下一个分配，以此类推。



FIFO Scheduler 是最简单也是最容易理解的调度器，也不需要任何配置，但它并不适用于共享集群。大的应用可能会占用所有集群资源，这就导致其它应用被阻塞。在共享集群中，更适合采用 Capacity Scheduler 或 Fair Scheduler，这两个调度器都允许大任务和小任务在提交的同时获得一定的系统资源。

5.2. Capacity Scheduler

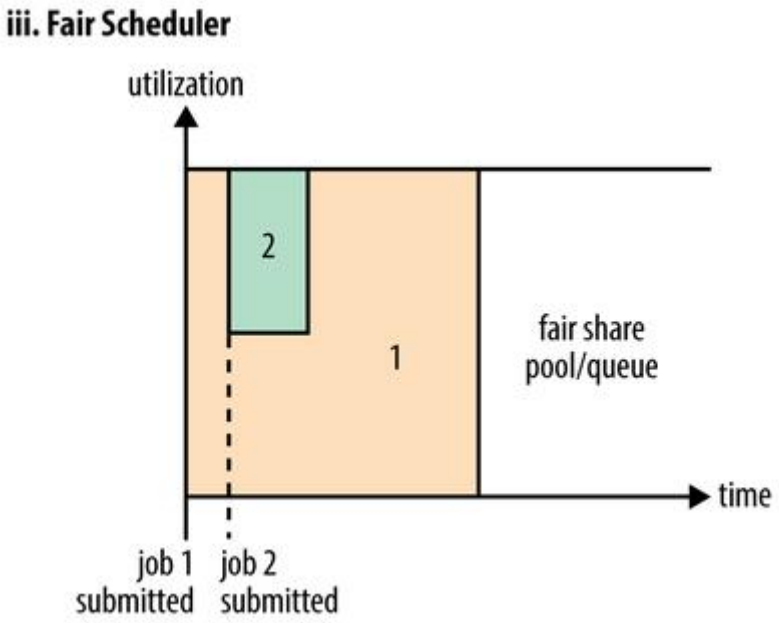
Capacity 调度器允许多个组织共享整个集群，每个组织可以获得集群的一部分计算能力。通过为每个组织分配专门的队列，然后再为每个队列分配一定的集群资源，这样整个集群就可以通过设置多个队列的方式给多个组织提供服务了。除此之外，队列内部又可以垂直划分，这样一个组织内部的多个成员就可以共享这个队列资源了，在一个队列内部，资源的调度是采用的是先进先出(FIFO)策略。



5.3. Fair Scheduler

在 Fair 调度器中，我们不需要预先占用一定的系统资源，Fair 调度器会为所有运行的 job 动态的调整系统资源。如下图所示，当第一个大 job 提交时，只有这一个 job 在运行，此时它获得了所有集群资源；当第二个小任务提交后，Fair 调度器会分配一半资源给这个小任务，让这两个任务公平的共享集群资源。

需要注意的是，在下图 Fair 调度器中，从第二个任务提交到获得资源会有一定的延迟，因为它需要等待第一个任务释放占用的 Container。小任务执行完成之后也会释放自己占用的资源，大任务又获得了全部的系统资源。最终效果就是 Fair 调度器即得到了高的资源利用率又能保证小任务及时完成。



5.4. 示例：Capacity 调度器配置使用

调度器的使用是通过 yarn-site.xml 配置文件中的 yarn.resourcemanager.scheduler.class 参数进行配置的，默认采用 Capacity Scheduler 调度器。

假设我们有如下层次的队列：

```
root
├── prod
└── dev
    ├── mapreduce
    └── spark
```

下面是一个简单的 Capacity 调度器的配置文件，文件名为 capacity-scheduler.xml。在这个配置中，在 root 队列下面定义了两个子队列 prod 和 dev，分别占 40%和 60%的容量。需要注意，一个队列的配置是通过属性 yarn.sheduler.capacity.<queue-path>.<sub-property>指定的，<queue-path>代表的是队列的继承树，如 root.prod 队列，<sub-property>一般指 capacity 和 maximum-capacity。

```
<configuration>
 <property>
    <name>yarn.scheduler.capacity.root.queues</name>
    <value>prod,dev</value>
  </property>
 <property>
    <name>yarn.scheduler.capacity.root.dev.queues</name>
    <value>mapreduce,spark</value>
  </property>
    <property>
    <name>yarn.scheduler.capacity.root.prod.capacity</name>
    <value>40</value>
  </property>
    <property>
```

```
    <name>yarn.scheduler.capacity.root.dev.capacity</name>
    <value>60</value>
  </property>
    <property>
    <name>yarn.scheduler.capacity.root.dev.maximum-capacity</name>
    <value>75</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.mapreduce.capacity</name>
    <value>50</value>
  </property>
   <property>
    <name>yarn.scheduler.capacity.root.dev.spark.capacity</name>
    <value>50</value>
  </property>
</configuration>
```

我们可以看到，dev 队列又被分成了 mapreduce 和 spark 两个相同容量的子队列。dev 的 maximum-capacity 属性被设置成了 75%，所以即使 prod 队列完全空闲 dev 也不会占用全部集群资源，也就是说，prod 队列仍有 25%的可用资源用来应急。我们注意到，mapreduce 和 spark 两个队列没有设置 maximum-capacity 属性，也就是说 mapreduce 或 spark 队列中的 job 可能会用到整个 dev 队列的所有资源（最多为集群的 75%）。而类似的，prod 由于没有设置 maximum-capacity 属性，它有可能会占用集群全部资源。

关于队列的设置，这取决于我们具体的应用。比如，在 MapReduce 中，我们可以通过 mapreduce.job.queuename 属性指定要用的队列。如果队列不存在，我们在提交任务时就会收到错误。如果我们没有定义任何队列，所有的应用将会放在一个 default 队列中。

注意：对于 Capacity 调度器，我们的队列名必须是队列树中的最后一部分，如果我们使用队列树则不会被识别。比如，在上面配置中，我们使用 prod 和 mapreduce 作为队列名是可以的，但是如果我们用 root.dev.mapreduce 或者 dev. mapreduce 是无效的。