

Hadoop MapReduce & YARN 3.3.4



1. MapReduce设计思想

- **分而治之**: 对付大数据并行处理，将大的数据切分成多个小数据，交给更多的节点参与运算。注意：不可拆分的计算任务或相互间有依赖关系的数据无法进行并行计算。
- **抽象模型**: Input、Split、Map、Shuffle、Reduce、Output。
 - Input: 读取数据。
 - Split: 对数据进行粗粒度切分。
 - Map: 对数据进行细粒度切分。
 - Shuffle: 洗牌。将各个 MapTask 结果合并输出到 Reduce。
 - Reduce: 对 Shuffle 进行汇总并输出到指定存储。
 - Output: HDFS、Hive、Spark、Flume.....
- **统一架构**: 程序员需要考虑数据存储、划分、分发、结果收集、错误恢复等诸多细节。为此，MapReduce 设计并提供了统一的计算框架，为程序员隐藏了绝大多数系统层面的处理细节。程序员只需要集中于应用问题和算法本身，而不需要关注其他系统层的处理细节，大大减轻了程序员开发程序的负担。
- **离线框架**: 可以实现上千台服务器集群并发工作，适合 PB 级以上海量数据的离线处理。
 - 不擅长实时计算：MapReduce 无法像 MySQL 一样，在毫秒或者秒级内返回结果。如果数据量小，使用 MR 反而不合适。
 - 不擅长流式计算：流式计算的输入数据是动态的，而 MapReduce 的输入数据集是静态的，不能动态变化。这是因为 MapReduce 自身的设计特点决定了数据源必须是静态的。
 - 不擅长 DAG (有向图) 计算：多个应用程序存在依赖关系，后一个应用程序的输入为前一个的输出。在这种情况下，MapReduce 并不是不能做，而是使用后，每个 MapReduce 作业的输出结果都会写入到磁盘，会造成大量的磁盘 IO，导致性能非常的低下。
- **计算向数据靠拢**: 将计算放在数据节点上进行工作。
- **顺序处理数据、避免随机访问数据**: 大规模数据处理的特点决定了大量的数据记录不可能存放在内存、而只可能放在外存中进行处理。磁盘的顺序访问和随即访问在性能上有巨大的差异。例：100 亿个数据记录，每个记录 100B，共计 1TB 的数据库。更新 1% 的记录（随机访问）需要 1 个月时间；而顺序访问并重写所有数据记录仅需 1 天时间。
- **失效被认为是常态**: MapReduce 集群中使用大量的低端服务器（Google 目前在全球共使用百万台以上的服务器节点），因此，节点硬件失效和软件出错是常态。因而：一个良好设计、具有容错性的并行计算系统不能因为节点失效而影响计算服务的质量，任何节点失效都不应当导致结果的不一致或不确定性；任何一个节点失效时，其它节点要能够无缝接管失效节点的计算任务；当失效节点恢复后应能自动无缝加入集群，而不需要管理员人工进行系统配置。MapReduce 并行计算软件框架使用了多种有效的机制，如节点自动重启技术，使集群和计算框架具有对付节点失效的健壮性，能有效处理失效节点的检测和恢复。

2. 常用排序算法

2.1. 分类

2.1.1. 不值钱

所谓不值钱指的是学过计算机的基本都会的算法，大家都会了变的普遍了，并不是说算法不值钱。

- 冒泡排序
- 选择排序
- 插入排序

2.1.2. 进阶型

- 希尔排序（高级插入）
- 堆排序（高级选择）

2.1.3. 常用型

- 快速排序
- 归并排序

2.1.4. 偏方型

所谓偏方治大病，在某些场景下会有奇效。

- 计数排序
- 桶排序
- 基数排序

2.2. 快速排序

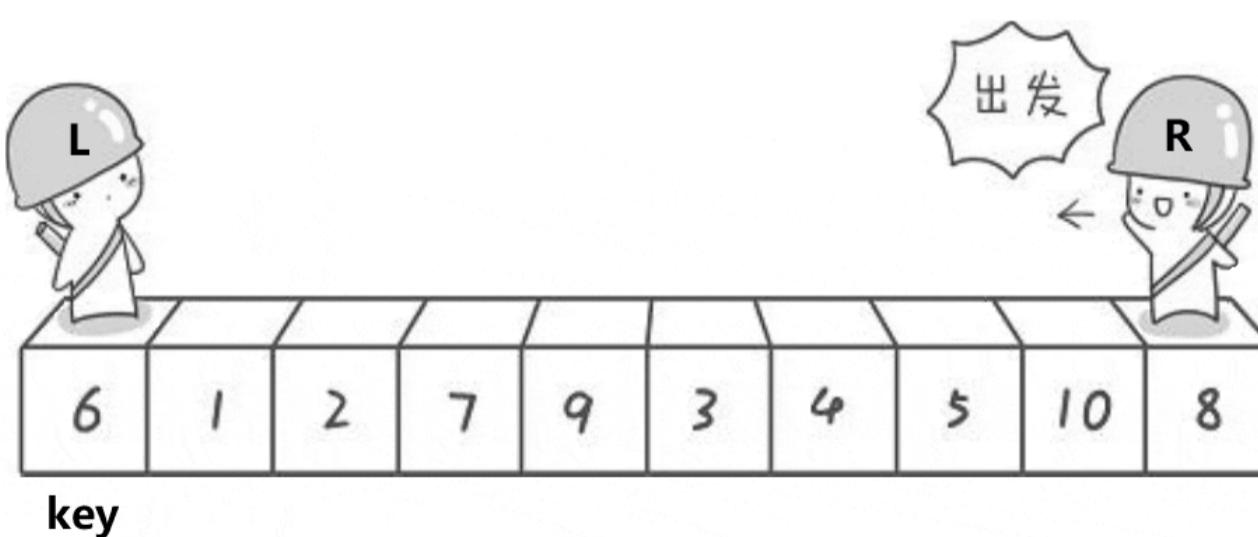
快速排序（Quick Sort）是从冒泡排序算法演变而来的，实际上是在冒泡排序基础上的递归分治法。快速排序在每一轮挑选一个基准元素，并让其他比它大的元素移动到数列的一边，比它小的元素移动到数列的另一边，从而把数列拆解成了两个部分。快速排序又分为：
Hoare 法（左右指针法）、前后指针法、挖坑法。

2.2.1. 左右指针法

Hoare 法（左右指针法）：快排的创始人 Hoare 命名的，也是快排最初的实现版本。

计算机领域的爵士——托尼·霍尔（Tony Hoare），发明了快速排序算法，1980 年获得图灵奖。

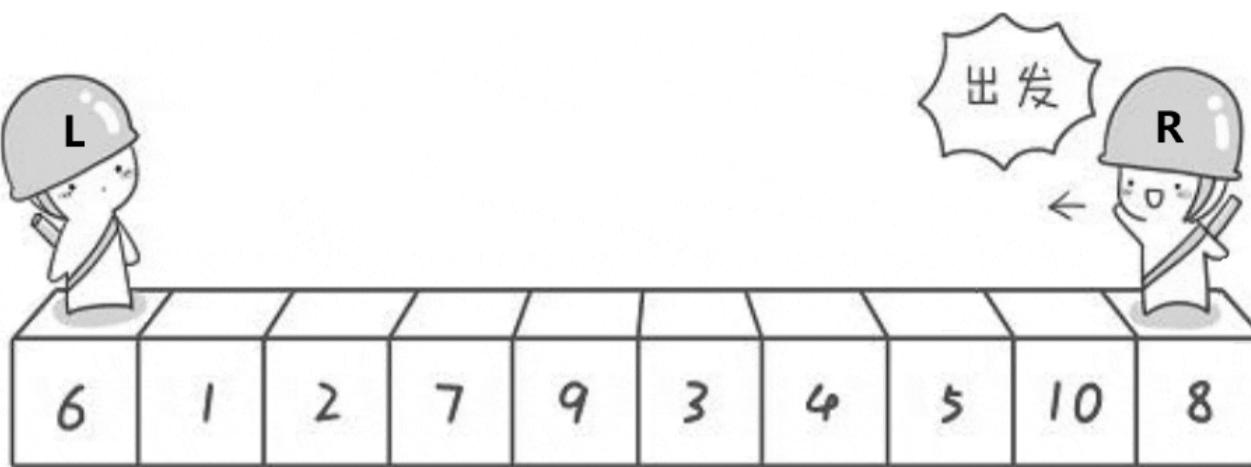
左右指针法算法步骤：定义一个 Begin 指向第一个元素，定义一个 End 指向最后一个元素。令第一个元素为 Key，Begin 向后找大于 Key 的值，End 向前找小于 Key 的值，找到之后把 Begin 跟 End 位置的值交换，直到 Begin 的索引大于等于 End 的索引时结束，然后将 Key 和 End 指针值交换。再将 Key 的左右两边重复上述操作，最终有序。



2.2.2. 挖坑法

先将第一个数据存放在临时变量 key 中，形成一个坑位

key =



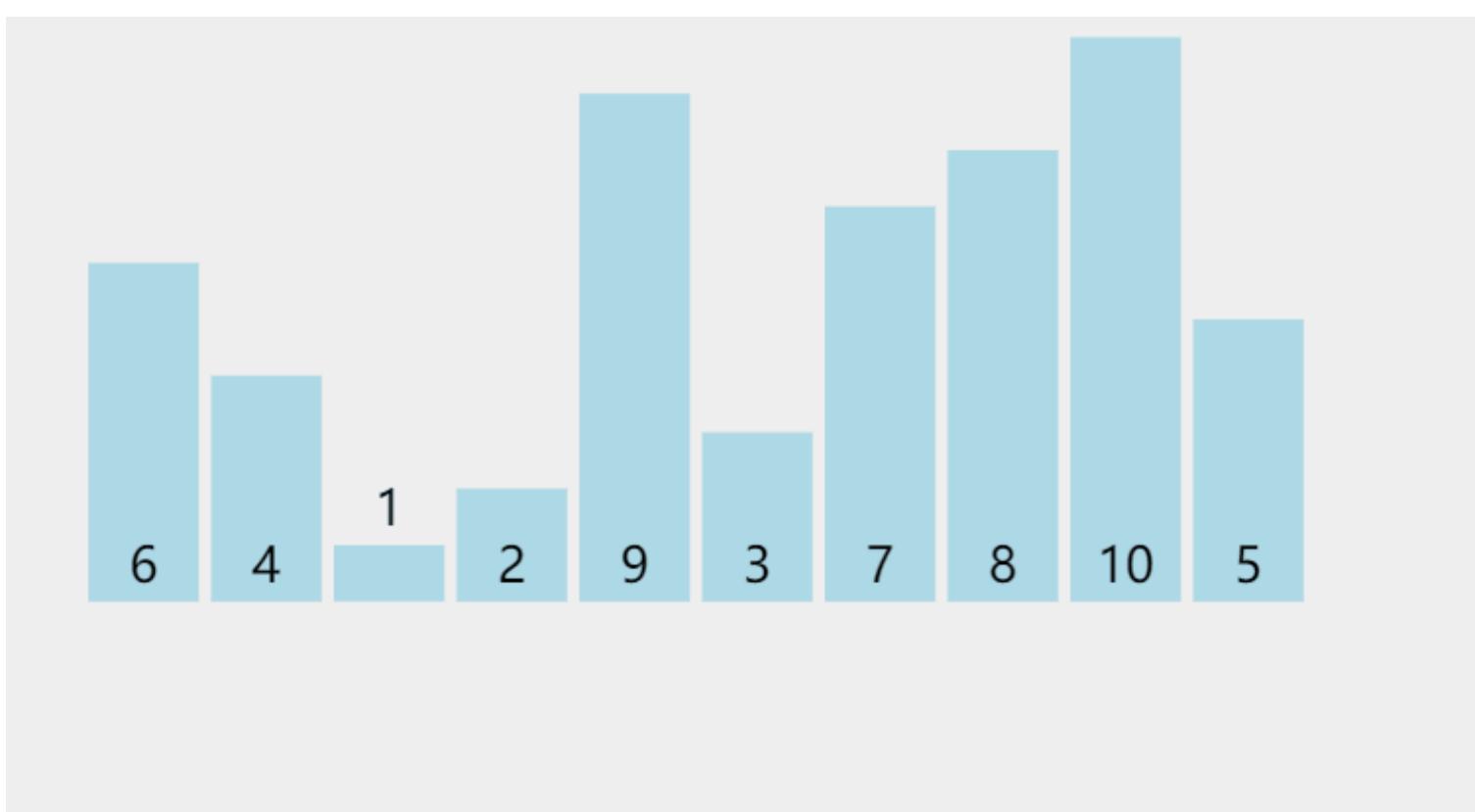
挖坑法算法步骤：

- 从数列中挑出一个元素，称为 "基准" (pivot)。例如：`arr = [5, 2, 4, 6, 1, 7, 8, 3]`，基准为 5；
- 重新排序数列，所有元素比基准值小的摆放在基准左边，所有元素比基准值大的摆在基准的右边；
 - 一开始基准在最左边，所以从右向左依次和基准进行比较，如果小于基准就和基准进行交换，如果大于基准保持不变，此时 `arr = [3, 2, 4, 6, 1, 7, 8, 5]`；
 - 此时基准跑到了右边，所以从左向右依次和基准进行比较，如果大于基准就和基准进行交换，如果小于基准保持不变，此时 `arr = [3, 2, 4, 5, 1, 7, 8, 6]`；
 - 此时基准跑到了左边，所以从右向左依次和基准进行比较，如果小于基准就和基准进行交换，如果大于基准保持不变，此时 `arr = [3, 2, 4, 1, 5, 7, 8, 6]`；
 - 此时基准跑到了右边，所以从左向右依次和基准进行比较，如果大于基准就和基准进行交换，如果小于基准保持不变，此时数列已无需改动，基准的位置是正确的，左右两边无序。
- 整体排序后，基准的位置是正确的，左右两边无序；
 - 左边：`[3, 2, 4, 1]`
 - 右边：`[7, 8, 6]`
- 将左右两边各当作一个新的序列继续挑出基准元素；
 - 左边：基准为 3
 - `[1, 2, 4, 3]`
 - `[1, 2, 3, 4]`
 - 右边：基准为 7
 - `[6, 7, 8]`
- 递归地 (recursive) 把小于基准元素的子数列和大于基准元素的子数列排序。

2.2.3. 前后指针法

前后指针法算法步骤：

- 选择一端下标设为基准值，例如：`arr = [5, 2, 4, 6, 1, 7, 8, 3]`，基准为 5；
- 然后初始设置 Cur、Prev 两个标志指针，Prev 标志序列第一个元素，Cur 标志 Prev 后一个位置；`Prev = 5, Cur = 2`；
- Cur 位置的元素若大于基准值，Cur 向前前进。若小于基准值，对 Prev 进行加一，然后判断是否与 Cur 的位置相等：
 - 若相等，Cur 继续向前前进；
 - 若不相等，则交换 Cur 和 Prev 的值。这样就能保证 Cur 与 Prev 之间的元素都大于基准值，Prev 之前的元素都小于基准值。
- 重复上述过程，直到 Cur 超过序列最右侧位置，最后进行一次判断，若 Prev 标记位置不是序列最后一个位置，则将基准值交换到 Prev 位置，即完成左右子序列划分，再对左右子序列重复上述过程，直到整个序列完成排序。



2.2.4. 优化选 Key

快速排序在选 Key 的时候，最理想，效率最高的情况就是每次都能选到中间值。但是，快速排序在没有优化前，对数据有序的情况进行排序，那么它每次选 Key 的值都在最左边或最右边，效率就会大大降低。所以：**当待排序数组有序时，快速排序的时间复杂的最差。**

例如：`arr = [1, 2, 3, 4, 5, 6, 7, 8]`，基准为 1。重新排序数列，所有元素比基准值小的摆放在基准左边，所有元素比基准值大的摆在基准的右边；

- 一开始基准在最左边，所以从右向左依次和基准进行比较，如果小于基准就和基准进行交换，如果大于基准保持不变，此时 `arr = [1, 2, 3, 4, 5, 6, 7, 8]`（从右向左比较了 7 次）；
- 此时基准移动到第二位，继续从右向左依次和基准进行比较，如果小于基准就和基准进行交换，如果大于基准保持不变，此时 `arr = [1, 2, 3, 4, 5, 6, 7, 8]`（从右向左比较了 6 次）；
- 此时基准移动到第三位，继续从右向左依次和基准进行比较，如果小于基准就和基准进行交换，如果大于基准保持不变，此时 `arr = [1, 2, 3, 4, 5, 6, 7, 8]`（从右向左比较了 5 次）；
-以此类推，将这个有序的数组排完，共比较了 $7+6+5+\dots+1 = 28$ 次。

为解决这一情况引入一个较为巧妙的方法，**三数取中**。“三数取中”即取三个数中不是最大也不是最小的数，将这一概念引入快速排序中，取首中尾三个元素的中间值（排序三个数，取中间数）作为待排序区间的 Key，再把这个元素和队头元素互换，即可解决这一问题。

例如：`arr = [1, 2, 3, 4, 5, 6, 7, 8]`，中间数为 5，根据三数取中优化规则得到：`arr = [5, 2, 3, 4, 1, 6, 7, 8]`。

- 然后重新排序数列，所有元素比基准值小的摆放在基准左边，所有元素比基准值大的摆在基准的右边；
- 一开始基准在最左边，所以从右向左依次和基准进行比较，如果小于基准就和基准进行交换，如果大于基准保持不变，此时 `arr = [1, 2, 3, 4, 5, 6, 7, 8]`（从右向左比较了 4 次）；
- 此时基准跑到了右边，所以从左向右依次和基准进行比较，如果大于基准就和基准进行交换，如果小于基准保持不变，此时 `arr = [1, 2, 3, 4, 5, 6, 7, 8]`（从左向右比较了 4 次）；
- 此时数列已无需改动，基准的位置是正确的，开始排序左右两边。将左右两边各当作一个新的序列继续挑出基准元素：
 - 左边：基准为 3（比较了 4 次）
 - `[1, 2, 3, 4]`
 - `[3, 2, 1, 4]`
 - `[1, 2, 3, 4]`
 - 右边：基准为 7（比较了 2 次）
 - `[6, 7, 8]`
-以此类推，将这个有序的数组排完，共比较了 $4+4+4+2 = 14$ 次。

可以看出引入“三数取中”后变量移动的次数比没有引入前少了许多。除此之外还可以使用以下方法：

- 随机选 Key；
- 针对有序情况，选正中间数据为 Key（前提是知道有序）。

2.3. 归并排序

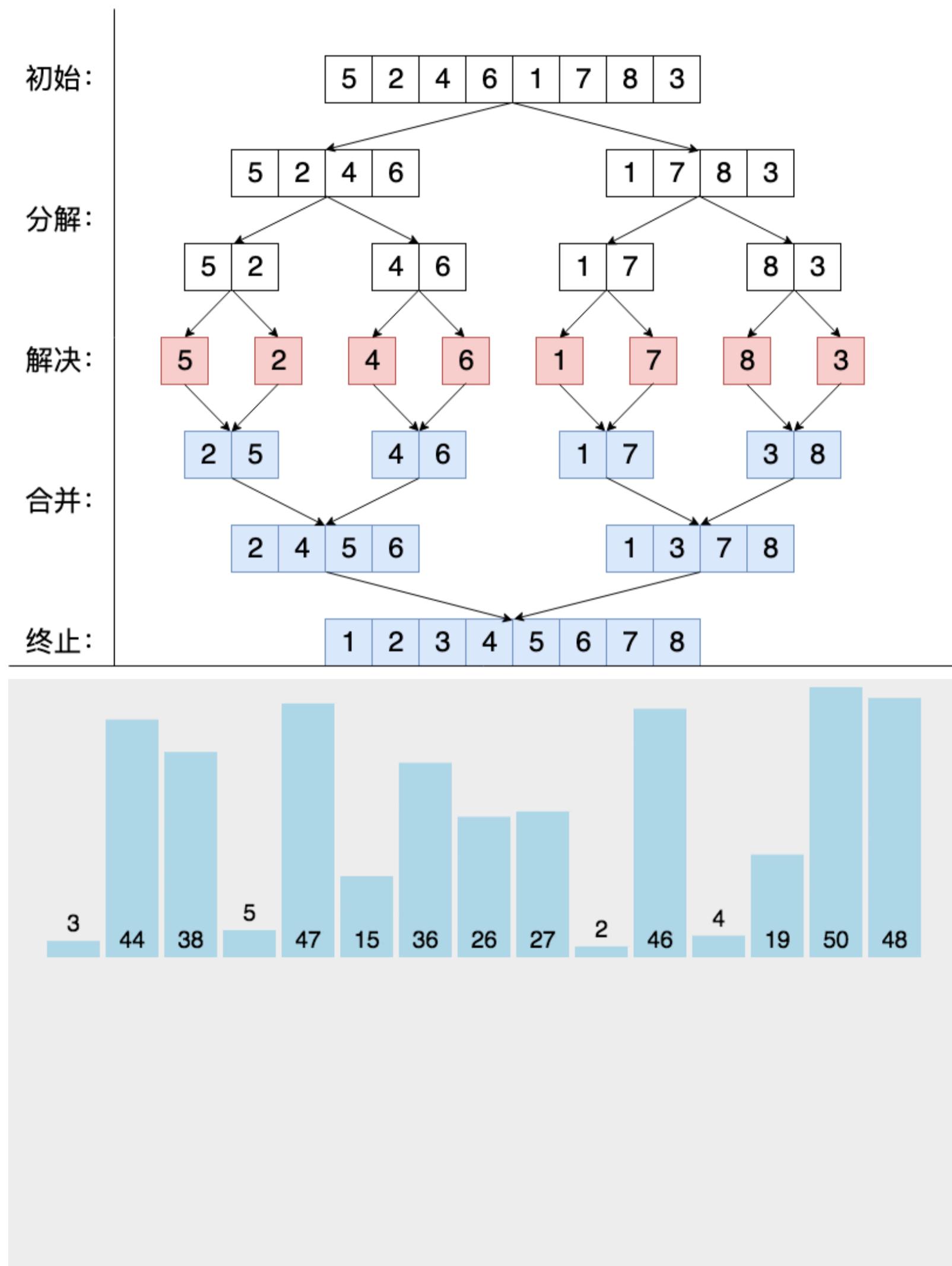
归并排序（Merge Sort）是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。

归并排序的基本思想是：将原序列不断拆分，一直拆到每个子序列只有一个元素时，再用插入的方式归并。即先使每个子序列有序，再使子序列段间有序。将已有序的子序列合并，得到完全有序的序列。将两个有序表合并成一个有序表的过程称为二路归并。

和选择排序一样，归并排序的性能不受输入数据的影响，但表现比选择排序好的多，因为始终都是 $O(n \log n)$ 的时间复杂度。代价是需要额外的内存空间。

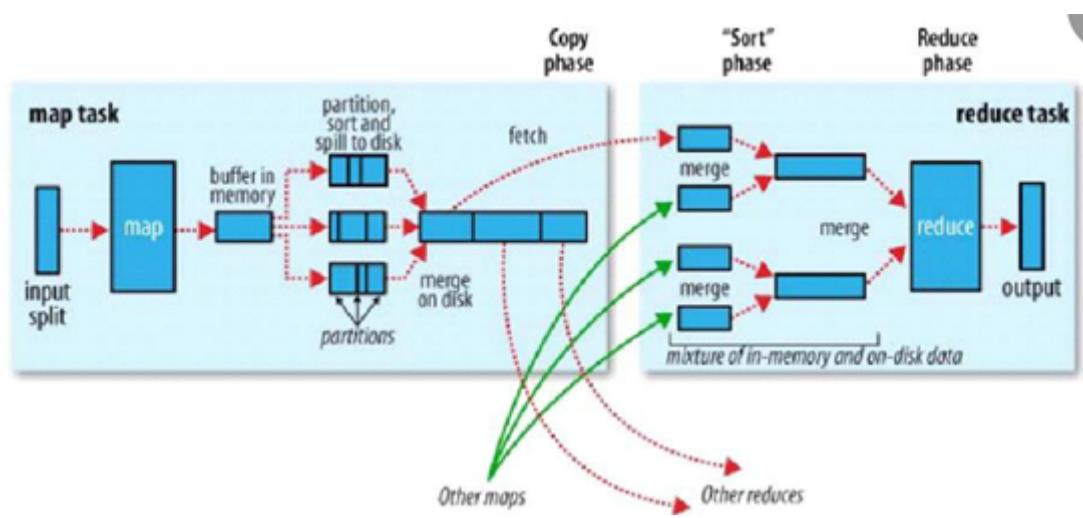
归并排序算法步骤：

- 将原序列二等分，二等分之后的子序列继续二等分；
- 直到每个子序列只有一个元素时，停止拆分；
- 再按照分割的顺序进行归并。



3. MapReduce计算流程

- 计算1T数据中每个单词出现的次数--> wordcount



3.1. 原始数据File

- 1T数据被切分成块存放在HDFS上，每一个块有128M大小

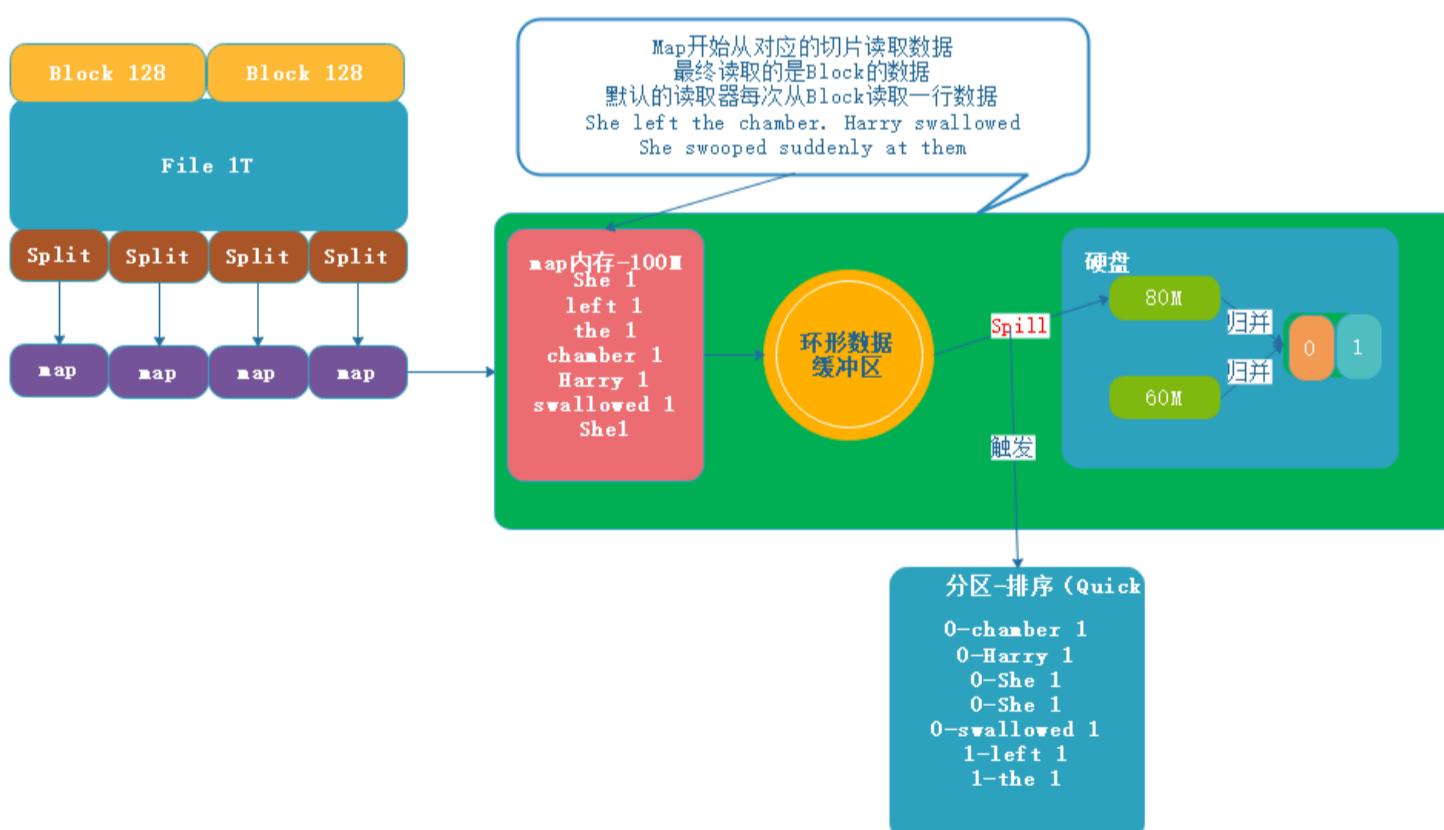
3.2. 数据块Block

- hdfs上数据存储的一个单元,同一个文件中块的大小都是相同的
- 因为数据存储到HDFS上不可变, 所以有可能块的数量和集群的计算能力不匹配
- 我们需要一个动态调整本次参与计算节点数量的一个单位
- 我们可以动态的改变这个单位--> 参与的节点

3.3. 切片Split

- 切片是一个逻辑概念
- 在不改变现在数据存储的情况下, 可以控制参与计算的节点数目
- 通过切片大小可以达到控制计算节点数量的目的
 - 有多少个切片就会执行多少个Map任务
- 一般切片大小为Block的整数倍(2~1/2)
 - 防止多余创建和很多的数据连接
- 如果Split>Block ,计算节点少了
- 如果Split<Block ,计算节点多了
- 默认情况下, Split切片的大小等于Block的大小,默认128M
- 一个切片对应一个MapTask

3.4. MapTask

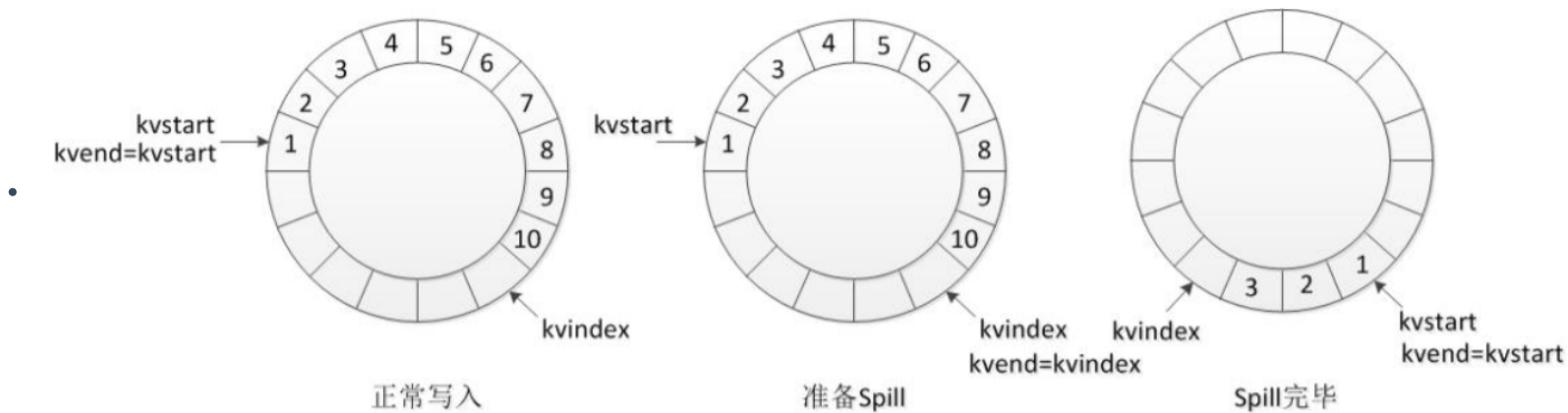


- map默认从所属切片读取数据,每次读取一行 (默认读取器) 到内存中
- 我们可以根据自己的逻辑(空格分隔)计算每个单词出现的次数
- 这是会产生 (Map<String, Integer>) 临时数据, 存放在内存中
- 但是内存大小是有限的, 如果多个任务同时执行有可能内存溢出 (OOM)
- 如果把数据都直接存放到硬盘, 效率太低
- 我们需要在OOM和效率低之间提供一个有效方案
 - 可以在内存中写入一部分, 然后写出到硬盘

3.5. 环形数据缓冲区

可以循环利用这块内存区域，减少数据溢写时map的停止时间

- 每一个Map可以独享的一个内存区域
- 在内存中构建一个环形数据缓冲区(kvBuffer),默认大小为100M
- 设置缓冲区的阈值为80%,当缓冲区的数据达到80M开始向外溢写到硬盘
- 溢写的时候还有20M的空间可以被使用效率并不会被减缓
- 而且将数据循环写到硬盘，不用担心OOM问题



3.6. 分区Partation

- 根据Key直接计算出对应的Reduce
- 分区的数量和Reduce的数量是相等的
- $\text{hash}(\text{key}) \% \text{partation} = \text{num}$
- 默认分区的算法是Hash然后取余
 - Object的hashCode()-->equals()
 - 如果两个对象equals,那么两个对象的hashcode一定相等
 - 如果两个对象的hashcode相等,但是对象不一定equals

3.7. 排序Sort

- 对要溢写的数据进行排序 (QuickSort)
- 按照先Partation后Key的顺序排序-->相同分区在一起,相同Key的在一起
- 我们将来溢写出的小文件也都是有序的

3.8. 溢写Spill

- 将内存中的数据循环写到硬盘,不用担心OOM问题
- 每次会产生一个80M的文件
- 如果本次Map产生的数据较多,可能会溢写多个文件

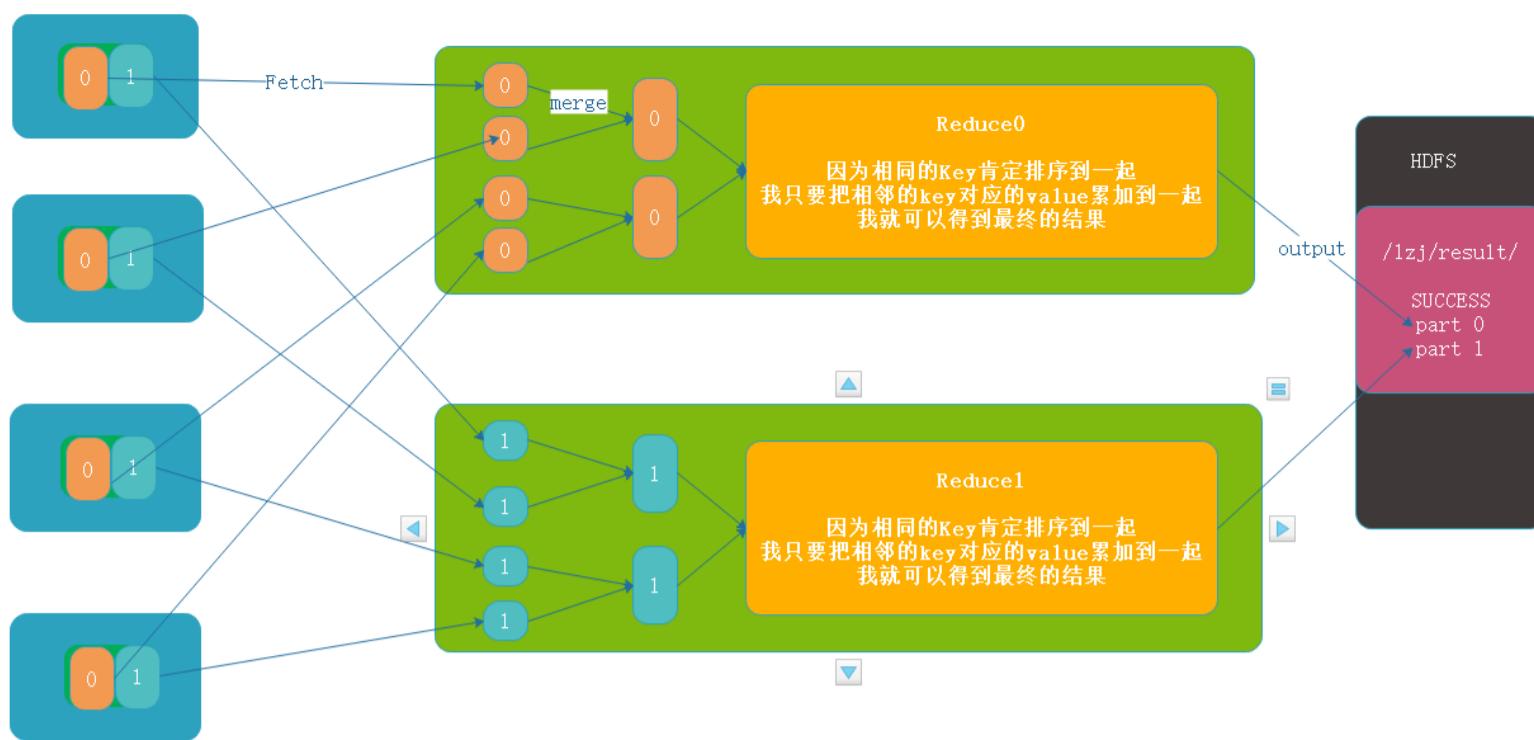
3.9. 合并Merge

- 因为溢写会产生很多有序(分区 key)的小文件,而且小文件的数目不确定
- 后面向reduce传递数据带来很大的问题
- 所以将小文件合并成一个大文件,将来拉取的数据直接从大文件拉取即可
- 合并小文件的时候同样进行排序(归并排序),最终产生一个有序的大文件
- Merge 是为了让传输的文件数量变少,但是网络传输数据量并没有改变,只是减少了网络 IO 次数

3.10. 组合器combiner

- 集群的带宽限制了mapreduce作业的数量,因此应该尽量避免map和reduce任务之间的数据传输。hadoop允许用户对map的输出数据进行处理,用户可自定义combiner函数(如同map函数和reduce函数一般),其逻辑一般和reduce函数一样,combiner的输入是map的输出,combiner的输出作为reduce的输入,很多情况下可以直接将reduce函数作为combiner函数来使用
`(job.setCombinerClass(FlowCountReducer.class);)`。
- combiner属于优化方案,所以无法确定combiner函数会调用多少次,可以在环形缓存区溢出文件时调用combiner函数,也可以在溢出的小文件合并成大文件时调用combiner。但要保证不管调用几次combiner函数都不会影响最终的结果,所以不是所有处理逻辑都可以使用combiner组件,有些逻辑如果在使用了combiner函数后会改变最后rereduce的输出结果(如求几个数的平均值,就不能先用combiner求一次各个map输出结果的平均值,再求这些平均值的平均值,这将导致结果错误)。
- Combiner 的意义是对每一个 MapTask 的输出进行局部汇总,以减小网络传输量。
 - 原先传给reduce的数据是 a1 a1 a1 a1 a1
 - 第一次combiner组合之后变为a{1,1,1,1,..}
 - 第二次combiner后传给reduce的数据变为a{4,2,3,5...}

3.11. 拉取Fetch



- 我们需要将Map的临时结果拉取到Reduce节点
- 原则：
 - 相同的Key必须拉取到同一个Reduce节点
 - 但是一个Reduce节点可以有多个Key
- 未排序前拉取数据的时候必须对Map产生的最终的合并文件做全序遍历
 - 而且每一个reduce都要做一个全序遍历
- 如果map产生的大文件是有序的，每一个reduce只需要从文件中读取自己所需的即可

3.12. 合并Merge

- 因为reduce拉取的时候，会从多个map拉取数据
- 那么每个map都会产生一个小文件,这些小文件 (文件与文件之间无序, 文件内部有序)
- 为了方便计算 (没必要读取N个小文件) ,需要合并文件
- 归并算法合并成2个(qishishilia)
- 相同的key都在一起

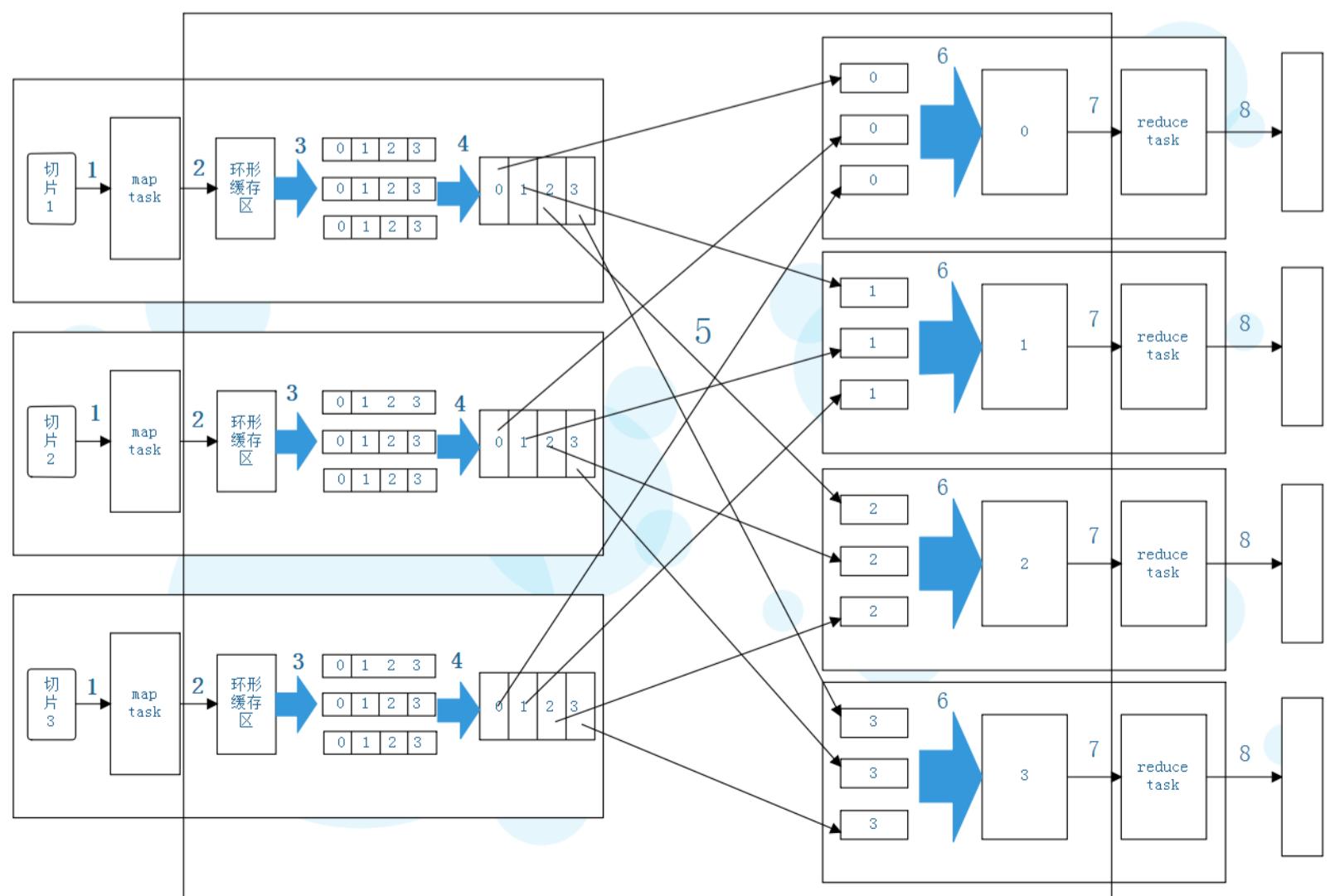
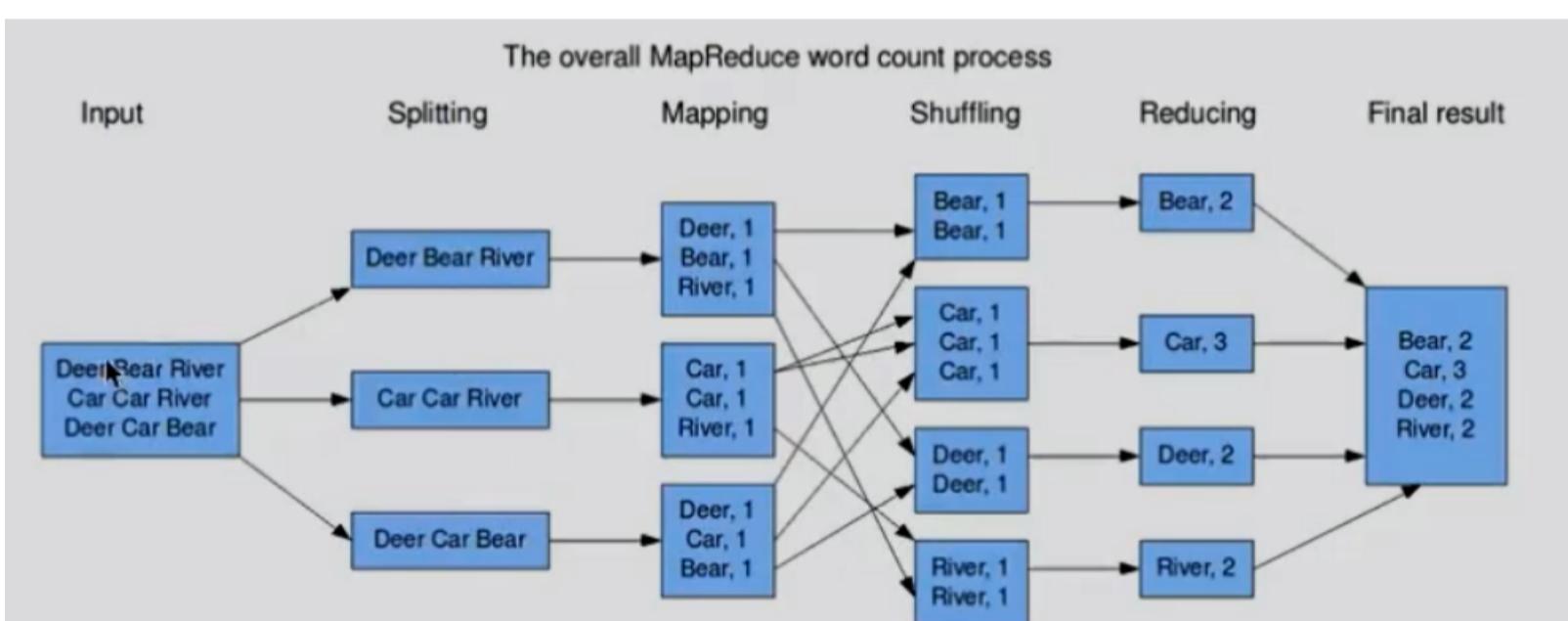
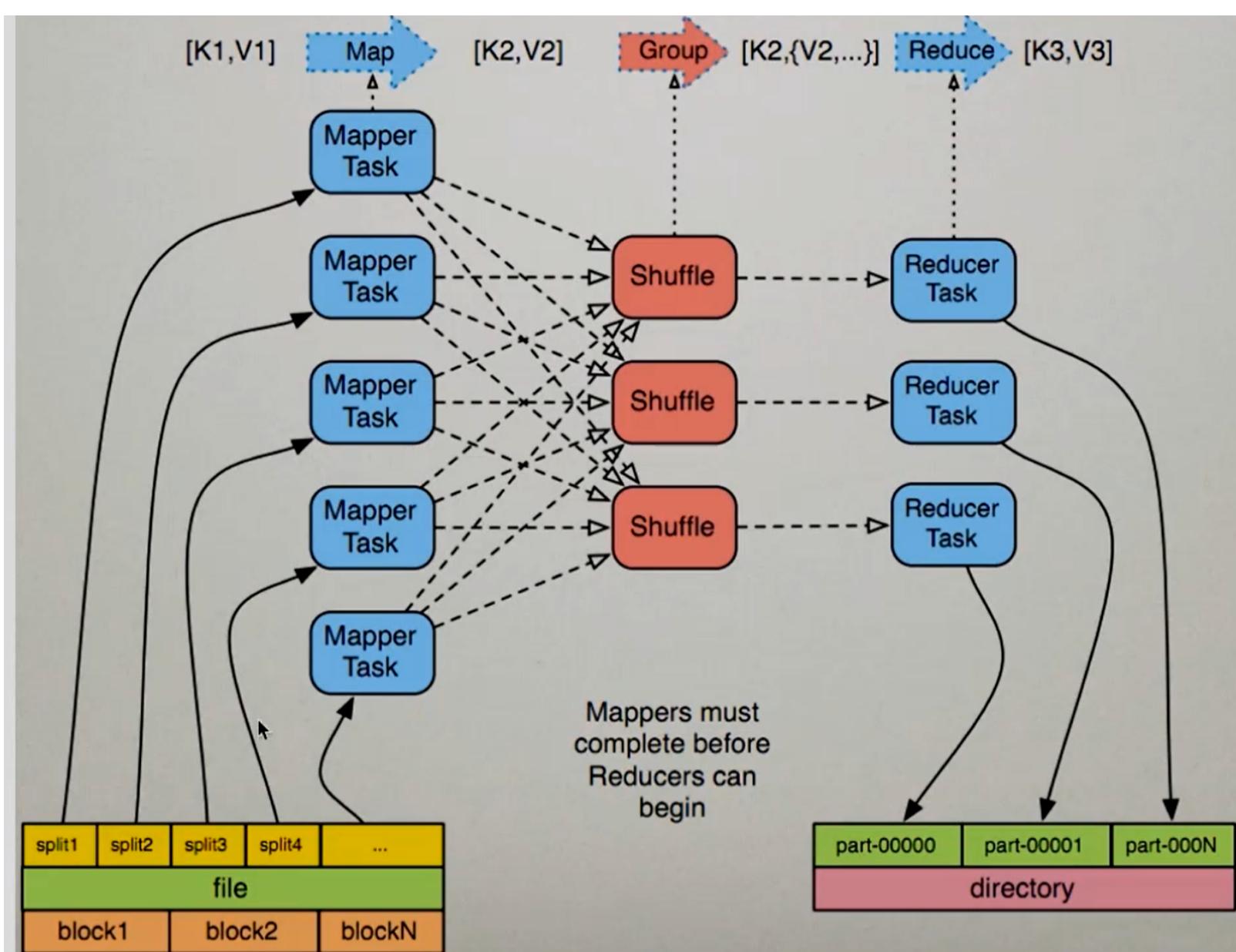
3.13. 归并Reduce

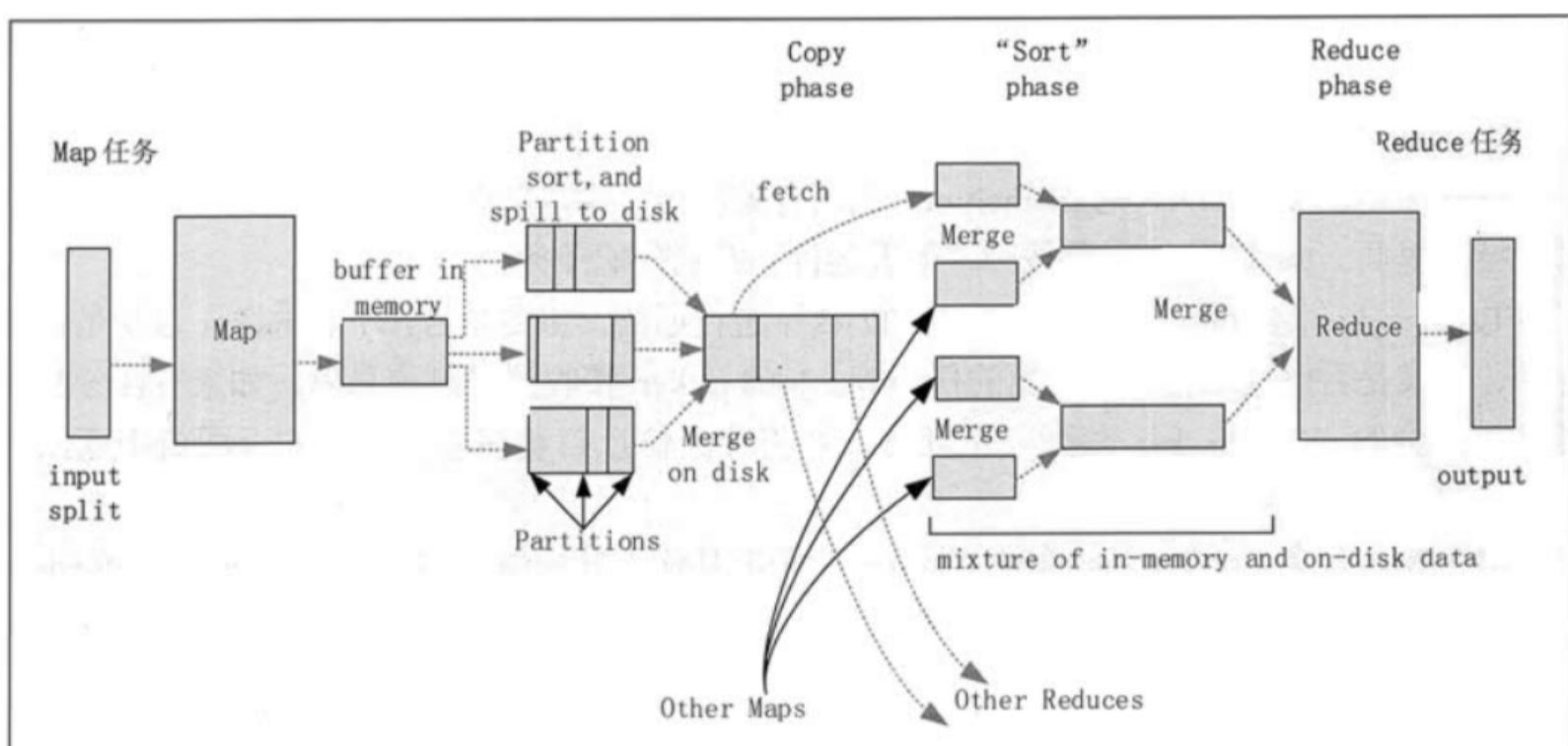
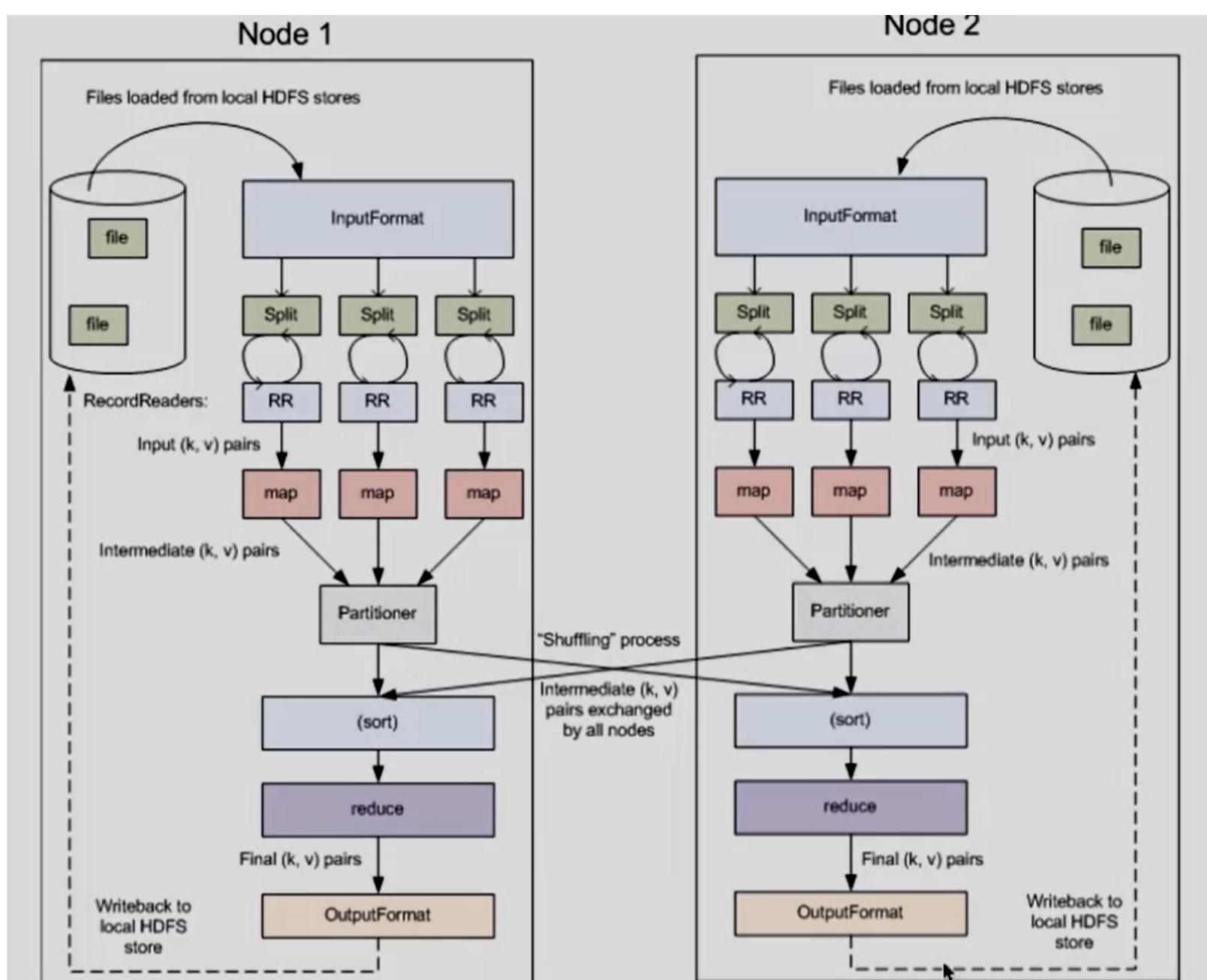
- 将文件中的数据读取到内存中
- 一次性将相同的key全部读取到内存中
- 直接将相同的key得到结果-->最终结果

3.14. 写出Output

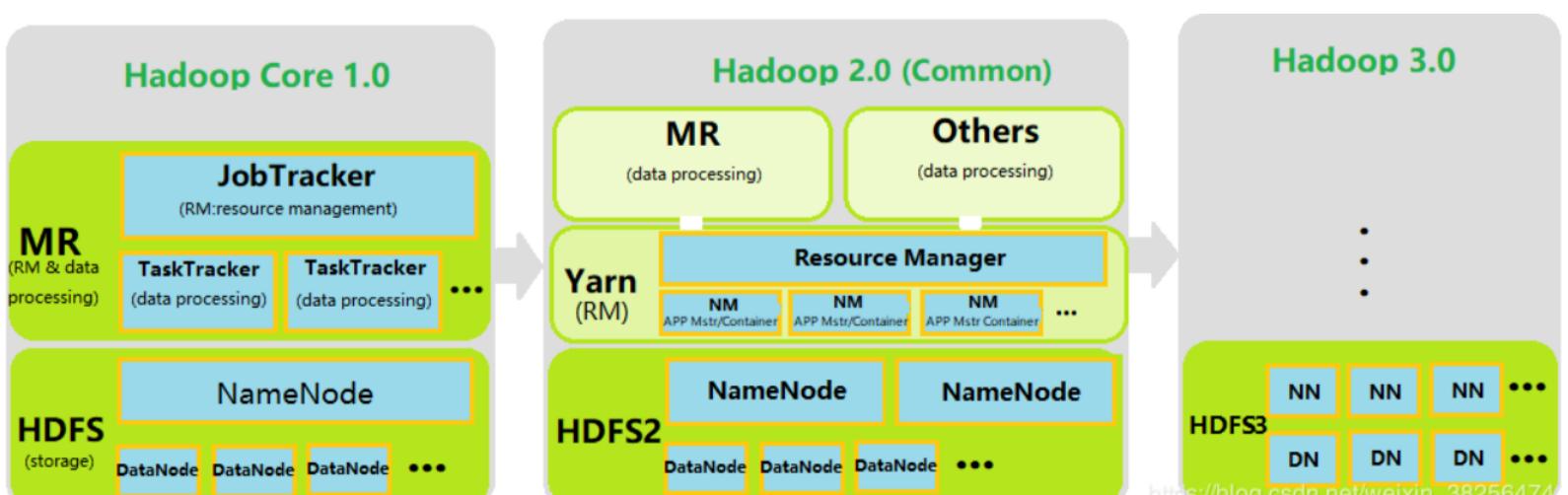
- 每个reduce将自己计算的最终结果都会存放到HDFS上

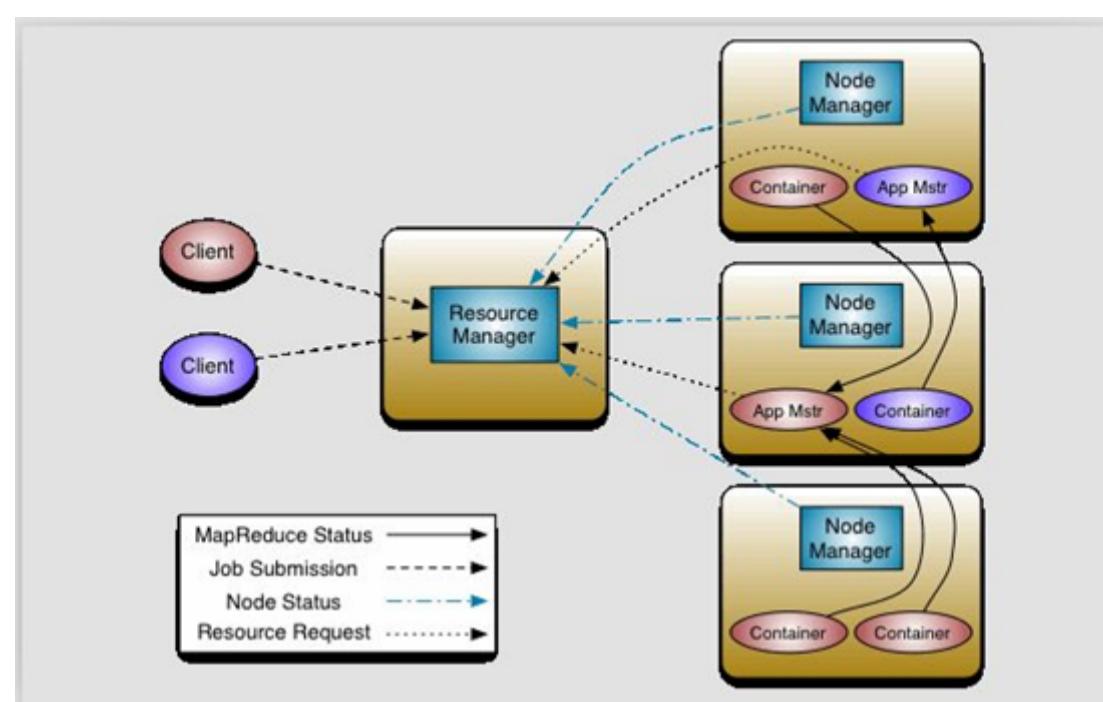
3.15. MapReduce过程截图





4. Hadoop-YARN架构





4.1. 基本概念

- 2.x开始使用Yarn (Yet Another Resource Negotiator, 另一种资源协调者) 统一管理资源
- 以后其他的计算框架可以直接访问yarn获取当前集群的空闲节点
- client
 - 客户端发送mr任务到集群
 - 客户端的种类有很多种
- ResourceManager
 - 资源协调框架的管理者
 - 分为主节点和备用节点 (防止单点故障)
 - 主备的切换基于Zookeeper进行管理
 - 时刻与NodeManager保持心跳，接受NodeManager的汇报
 - NodeManager汇报当前节点的资源情况
 - 当有外部框架要使用资源的时候直接访问ResourceManager即可
 - 如果有MR任务，先去ResourceManager申请资源，ResourceManager根据汇报相对灵活分配资源
 - 资源在NodeManager1，NodeManager1要负责开辟资源
- NodeManager
 - 资源协调框架的执行者
 - 每一个DataNode上默认有一个NodeManager
 - NodeManager汇报自己的信息到ResourceManager
- Container
 - 2.x资源的代名词
 - Container动态分配的
- ApplicationMaster
 - 我们本次Job任务的主导者
 - 负责调度本次被分配的资源Container
 - 当所有的节点任务全部完成，application告诉ResourceManager请求杀死当前ApplicationMaster线程
 - 本次任务所有的资源都会被释放
- Task(MapTask--ReduceTask)
 - 开始按照MR的流程执行业务
 - 当任务完成时，ApplicationMaster接收到当前节点的回馈

4.2. 工作流程

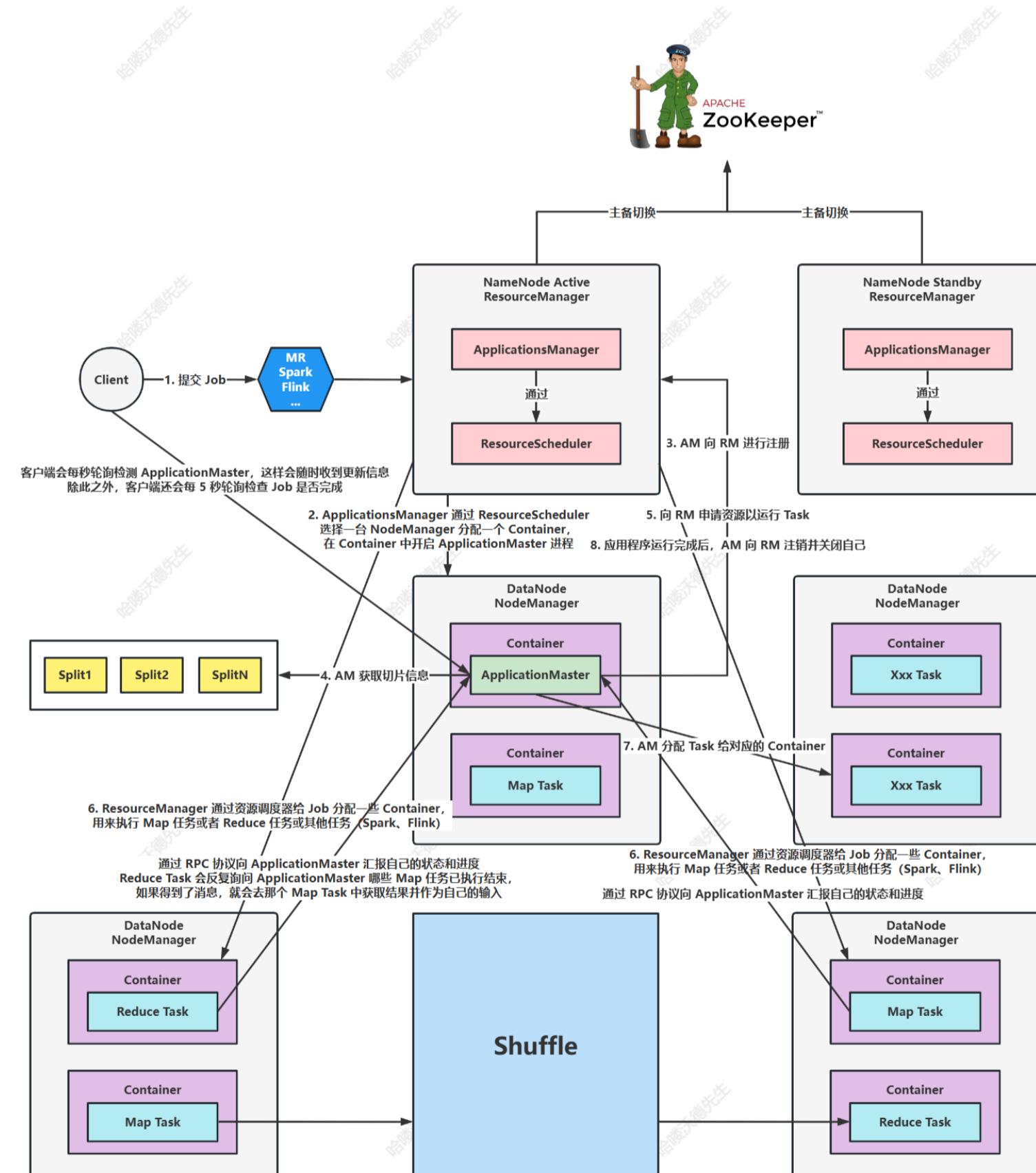
首先确认执行 MapReduce 作业的运行时框架，根据 `mapreduce.framework.name` 变量进行配置：

- 如果等于 `yarn`：则创建 YARNRunner 对象；
- 如果等于 `local`：则创建 LocalJobRunner 对象。

如果是 YARN 平台，客户端将对 ResourceManager 发起提交作业申请，具体流程如下：

- Client 对 ResourceManager 发起提交作业申请；
- ResourceManager 返回 JobID (即 Application ID) 和保存数据资源 (作业的 Jar 文件，配置文件，计算所得输入分片，资源信息等) 的临时目录 (使用 JobID 命名的目录，`hdfs://xxx/staging/xxx`)；

- 接着 Client 计算分片，拷贝资源(作业的 Jar 文件，配置文件，计算所得输入分片，资源信息等)到 HDFS，最后用 submitApplication 函数提交 Job 给 ResourceManager；
- ApplicationManager 接收 submitApplication 方法提交的 Job，并将其交给 ResourceScheduler (调度器) 处理；
- ResourceManager 选择一台 NodeManager 分配一个 Container，在 Container 中开启 ApplicationMaster 进程；
- 首先 ApplicationMaster 向 ResourceManager 进行注册，这样用户可以直接通过 ResourceManager 查看应用程序的运行状态；然后 ApplicationMaster 收集计算后的输入分片情况来向 ResourceManager 申请对应的资源以运行 Task；最后 ApplicationMaster 初始化一定数量的记录对象(bookkeeping)来跟踪 Job 的运行进度，并收集每个 Task 的进度和完成情况，直到运行结束；
 - ApplicationMaster 采用轮询的方式，通过 RPC 协议向 ResourceManager 申请和领取资源；
 - ApplicationMaster 申请到资源后，会与对应的 NodeManager 进行通讯，要求它启动 Container；
 - NodeManager 为任务设置好运行环境（包括环境变量、Jar 包、二进制程序等）后，将 Task 启动命令写到一个脚本中，并通过运行该脚本启动对应的任务；
 - 各个任务通过 RPC 协议向 ApplicationMaster 汇报自己的状态和进度，方便 ApplicationMaster 随时掌握各个任务的运行状态，从而可以在任务失败的时候重新启动任务；
 - 此期间，客户端会每秒轮询检测 ApplicationMaster，这样就会随时收到更新信息，这些信息可以通过 Web UI 来进行查看。除此之外，客户端还会每 5 秒轮询检查 Job 是否完成，需要调用 Job 类下的 waitForCompletion() 方法，Job 结束后该方法返回。轮询时间间隔可以通过 mapreduce.client.completion.pollinterval 进行设置。
- 应用程序运行完成后，ApplicationMaster 向 ResourceManager 注销并关闭自己。



是否可以直接从本地 IDEA 将程序提交到 YARN 平台？可以，核心代码如下。

```
// 加载配置文件
Configuration configuration = new Configuration(true);
// HDFS 文件系统主机地址
conf.set("fs.defaultFS", "hdfs://node01:8020");
// MapReduce 作业的运行时框架，Local 本地模式（学习环境） YARN 模式（正式环境）
conf.set("mapreduce.framework.name", "yarn");
// 设置 ResourceManager 的主机地址，默认为 0.0.0.0
conf.set("yarn.resourcemanager.hostname", "node01");
// 允许跨平台提交（因为 Windows 和 Linux 系统的结构不一样，默认使用 Linux 系统的提交方式）
// 不开启跨平台提交，在 Windows 提交应用会报 /bin/bash: line 0: fg: no job control 错误
```

```

conf.set("mapreduce.app-submission.cross-platform", "true");
// 创建作业
Job job = Job.getInstance(conf);
// 要提交的应用程序的 Jar 包位置
job.setJar("D:\\Projects\\IdeaProjects\\yjxxt\\hadoop-demo\\target\\hadoop-demo-1.0-SNAPSHOT.jar");

```

注意：如果在 resources 目录下添加了 Hadoop 集群的 core-site.xml hdfs-site.xml mapred-site.xml yarn-site.xml，那么可以注释以上除 Jar 包之外的配置代码。这里的配置文件要和 Hadoop 集群中的配置文件一致，否则会出错，最好的做法是从集群中直接 Copy 出来。

5. Hadoop-YARN环境搭建

5.1. 目标环境

节点	NN01	NN02	DN	Zookeeper	ZKFC	JournalNode	ResourceManager	NodeManager
node01	✓		✓	✓	✓	✓	✓	✓
node02		✓	✓	✓	✓	✓		✓
node03			✓	✓		✓	✓	✓

YARN 环境的搭建基于前面的 HA 环境。

5.2. 修改配置文件

修改环境配置文件 `hadoop-env.sh` :

```
[root@node01 ~]# cd /opt/yjx/hadoop-3.3.4/etc/hadoop/
[root@node01 hadoop]# vim hadoop-env.sh
```

在文件末尾添加以下内容：

```
export JAVA_HOME=/usr/java/jdk1.8.0_351-amd64
export HDFS_NAMENODE_USER=root
export HDFS_DATANODE_USER=root
export HDFS_ZKFC_USER=root
export HDFS_JOURNALNODE_USER=root
export YARN_RESOURCEMANAGER_USER=root
export YARN_NODEMANAGER_USER=root
```

修改 Map 配置文件 `mapred-site.xml` :

```
[root@node01 hadoop]# vim mapred-site.xml
```

在 `configuration` 节点中添加以下内容：

```
<!-- 设置执行 MapReduce 任务的运行时框架为 YARN -->
<property>
<name>mapreduce.framework.name</name>
<value>yarn</value>
</property>
<!-- 设置 MapReduce JobHistory 服务器的地址 -->
<property>
<name>mapreduce.jobhistory.address</name>
<value>node01:10020</value>
</property>
<!-- 设置 MapReduce JobHistory 服务器的 Web 地址 -->
<property>
<name>mapreduce.jobhistory.webapp.address</name>
<value>node01:19888</value>
</property>
<!-- 设置已经运行完的 Hadoop 作业记录的存放路径（HDFS 文件系统中的目录），默认是 ${yarn.app.mapreduce.am.staging-dir}/history/done -->
<property>
<name>mapreduce.jobhistory.done-dir</name>
<value>/history/done</value>
</property>
```

```

<!-- 设置正在运行中的 Hadoop 作业记录的存放路径 (HDFS 文件系统中的目录) , 默认是 ${yarn.app.mapreduce.am.staging-dir}/history/done_intermediate -->
<property>
<name>mapreduce.jobhistory.intermediate-done-dir</name>
<value>/history/done_intermediate</value>
</property>
<!-- 设置需要加载的 jar 包和环境配置 -->
<property>
<name>mapreduce.application.classpath</name>
<value>
/opt/yjx/hadoop-3.3.4/etc/hadoop,
/opt/yjx/hadoop-3.3.4/share/hadoop/common/*,
/opt/yjx/hadoop-3.3.4/share/hadoop/common/lib/*,
/opt/yjx/hadoop-3.3.4/share/hadoop/hdfs/*,
/opt/yjx/hadoop-3.3.4/share/hadoop/hdfs/lib/*,
/opt/yjx/hadoop-3.3.4/share/hadoop/mapreduce/*,
/opt/yjx/hadoop-3.3.4/share/hadoop/yarn/*,
/opt/yjx/hadoop-3.3.4/share/hadoop/yarn/lib/*
</value>
</property>

```

修改 YARN 配置文件 `yarn-site.xml` :

```
[root@node01 hadoop]# vim yarn-site.xml
```

在 `configuration` 节点中添加以下内容:

```

<!-- 提交 MapReduce 作业的 staging 目录 (HDFS 文件系统中的目录) , 默认是 /tmp/hadoop-yarn/staging -->
<property>
<name>yarn.app.mapreduce.am.staging-dir</name>
<value>/tmp/hadoop-yarn/staging</value>
</property>
<!-- 设置开启 ResourceManager 高可用 -->
<property>
<name>yarn.resourcemanager.ha.enabled</name>
<value>true</value>
</property>
<!-- 设置 ResourceManager 的集群 ID -->
<property>
<name>yarn.resourcemanager.cluster-id</name>
<value>yarn-yjx</value>
</property>
<!-- 设置 ResourceManager 节点的名字 -->
<property>
<name>yarn.resourcemanager.ha.rm-ids</name>
<value>rm1,rm2</value>
</property>
<!-- 设置 ResourceManager 服务器的地址 -->
<property>
<name>yarn.resourcemanager.hostname.rm1</name>
<value>node01</value>
</property>
<!-- 设置 ResourceManager 服务器的地址 -->
<property>
<name>yarn.resourcemanager.hostname.rm2</name>
<value>node03</value>
</property>
<!-- 设置 ResourceManager 服务器的 Web 地址 -->
<property>
<name>yarn.resourcemanager.webapp.address.rm1</name>
<value>node01:8088</value>
</property>
<!-- 设置 ResourceManager 服务器的 Web 地址 -->
<property>
<name>yarn.resourcemanager.webapp.address.rm2</name>
<value>node03:8088</value>
</property>
<!-- 设置 YARN 的 ZK 集群地址, 以逗号分隔 -->
<property>
<name>yarn.resourcemanager.zk-address</name>
<value>node01:2181,node02:2181,node03:2181</value>
</property>
<!-- 定义用户自定义服务或者系统服务, 以逗号分隔, 服务名称只能包含 A-zA-Z0-9, 不能以数字开头, 例如: mapreduce_shuffle,spark_shuffle -->
<property>
<name>yarn.nodemanager.aux-services</name>
<value>mapreduce_shuffle</value>
</property>
<!-- MapReduce 是在各个机器上运行的, 在运行过程中产生的日志存在于不同的机器上, 为了能够统一查看各个机器的运行日志, 将日志集中存放在 HDFS 上, 这个过程就是日志聚合 -->

```

```

<!-- 设置开启日志聚合，日志聚合会收集每个容器的日志，并在应用程序完成后将这些日志移动到文件系统，例如 HDFS -->
<property>
<name>yarn.log-aggregation-enable</name>
<value>true</value>
</property>
<!-- 设置聚合日志的保留时间 -->
<property>
<name>yarn.log-aggregation.retain-seconds</name>
<value>640800</value>
</property>
<!-- 设置是否启用自动恢复，如果为 true 则必须指定 yarn.resourcemanager.store.class -->
<property>
<name>yarn.resourcemanager.recovery.enabled</name>
<value>true</value>
</property>
<!-- 设置 ResourceManager 的状态信息存储在 ZooKeeper 集群 -->
<property>
<name>yarn.resourcemanager.store.class</name>
<value>org.apache.hadoop.yarn.server.resourcemanager.recovery.ZKRMStateStore</value>
</property>
<!-- 设置是否对容器强制执行物理内存限制 -->
<!-- 是否启动一个线程检查每个任务正在使用的物理内存量，如果任务超出分配值，则将其直接杀掉，默认为 true -->
<property>
<name>yarn.nodemanager.pmem-check-enabled</name>
<value>false</value>
</property>
<!-- 设置是否对容器强制执行虚拟内存限制 -->
<!-- 是否启动一个线程检查每个任务正在使用的虚拟内存量，如果任务超出分配值，则将其直接杀掉，默认为 true -->
<property>
<name>yarn.nodemanager.vmem-check-enabled</name>
<value>false</value>
</property>
<!-- 设置容器的虚拟内存限制，虚拟内存与物理内存之间的比率。作用：在物理内存不够用的情况下，如果占用了大量虚拟内存并且超过了一定阈值，那么就认为当前集群的性能比较差 -->
<property>
<name>yarn.nodemanager.vmem-pmem-ratio</name>
<value>2.1</value>
</property>
<!-- 配置 JobHistory -->
<property>
<name>yarn.log.server.url</name>
<value>http://node01:19888/jobhistory/logs</value>
</property>

```

提示：如果 `yarn.nodemanager.aux-services` 选项配置为 `spark_shuffle`，需要拷贝 `$SPARK_HOME/yarn/spark-x.y.z-yarn-shuffle.jar` 到 `$HADOOP_HOME/share/hadoop/yarn/lib` 目录。

5.3. 拷贝至其他节点

将 node01 已配置好的 YARN 拷贝至 node02 和 node03。

```

[root@node01 hadoop]# pwd
/opt/yjx/hadoop-3.3.4/etc/hadoop
[root@node01 hadoop]# scp mapred-site.xml yarn-site.xml root@node02:`pwd`
[root@node01 hadoop]# scp mapred-site.xml yarn-site.xml root@node03:`pwd`

# 或者使用分发脚本
[root@node01 hadoop]# yjxrsync mapred-site.xml yarn-site.xml

```

5.4. 启动

首先启动 ZooKeeper（三台机器都需要执行）。

```

zkServer.sh start
zkServer.sh status

```

启动 HDFS。

```
[root@node01 hadoop]# start-dfs.sh
```

启动 YARN。

```
[root@node01 hadoop]# start-yarn.sh
```

启动 JobHistory。

```
[root@node01 hadoop]# mapred --daemon start historyserver
```

后期只需要先启动 ZooKeeper 然后启动 Hadoop (start-all.sh) 再启动 JobHistory 即可。

5.5. 访问

访问: <http://node01:9870/> 和 <http://node02:9870/> 结果如下。

Overview 'node01:8020' (active)

Namespace:	hdfs-bd
Namenode ID:	nn1
Started:	Mon Feb 20 15:55:37 +0800 2023
Version:	3.3.4, ra585a73c3e02ac62350c136643a5e7f6095a3dbb
Compiled:	Fri Jul 29 20:32:00 +0800 2022 by stevel from branch-3.3.4
Cluster ID:	CID-06562890-fc0f-4996-938e-c37bd2ceefc5
Block Pool ID:	BP-2016768711-192.168.100.101-1676819347502

Summary

Security is off.
Safemode is off.
4 files and directories, 0 blocks (0 replicated blocks, 0 erasure coded block groups) = 4 total filesystem object(s).
Heap Memory used 57.83 MB of 144 MB Heap Memory. Max Heap Memory is 944 MB.
Non Heap Memory used 64.35 MB of 68.06 MB Committed Non Heap Memory. Max Non Heap Memory is <unbounded>.

Overview 'node02:8020' (standby)

Namespace:	hdfs-bd
Namenode ID:	nn2
Started:	Mon Feb 20 15:55:44 +0800 2023
Version:	3.3.4, ra585a73c3e02ac62350c136643a5e7f6095a3dbb
Compiled:	Fri Jul 29 20:32:00 +0800 2022 by stevel from branch-3.3.4
Cluster ID:	CID-06562890-fc0f-4996-938e-c37bd2ceefc5
Block Pool ID:	BP-2016768711-192.168.100.101-1676819347502

Summary

Security is off.
Safemode is off.
4 files and directories, 0 blocks (0 replicated blocks, 0 erasure coded block groups) = 4 total filesystem object(s).
Heap Memory used 43.83 MB of 60.23 MB Heap Memory. Max Heap Memory is 440.81 MB.
Non Heap Memory used 61.25 MB of 65 MB Committed Non Heap Memory. Max Non Heap Memory is <unbounded>.

访问: <http://node01:8088> 或者 <http://node03:8088>，会被自动转发到 ResourceManager 的主节点。

All Applications

ID	User	Name	Application Type	Application Tags	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU Vcores	Allocated Memory MB	Allocated GPUs	Reserved CPU Vcores	
No data available in table																	

访问：<http://node01:19888/jobhistory> 结果如下。

也可以使用 `jps` 命令查看各节点服务运行情况。

```
[root@node01 hadoop]# jps
8992 Jps
8612 NodeManager
8852 JobHistoryServer
8455 ResourceManager
7176 QuorumPeerMain
7499 NameNode
7643 DataNode
7887 JournalNode
8095 DFSZKFailoverController
[root@node01 hadoop]# 

[root@node02 ~]# jps
5376 NameNode
5808 NodeManager
6032 Jps
5698 DFSZKFailoverController
5460 DataNode
5560 JournalNode
5149 QuorumPeerMain
[root@node02 ~]# 

[root@node03 ~]# jps
5697 NodeManager
5490 JournalNode
5604 ResourceManager
6101 Jps
5150 QuorumPeerMain
5390 DataNode
[root@node03 ~]# ]
```

5.6. 关闭

先关闭 Hadoop 和 JobHistory。

```
[root@node01 hadoop]# mapred --daemon stop historyserver
[root@node01 hadoop]# stop-all.sh
```

再关闭 ZooKeeper (三台机器都需要执行)。

```
zkServer.sh stop
```

环境搭建成功后 `shutdown -h now` 关机拍摄快照。

6. MapReduce案例[WordCount]

6.1. Java代码实现

继续在学习 HDFS 时创建的 `hadoop-demo` 项目基础上进行编写。将新搭建的 YARN 环境的配置文件重新拷贝一份至项目。主要拷贝以下配置文件：

- core-site.xml
- hdfs-site.xml
- mapred-site.xml
- yarn-site.xml
- log4j.properties

然后修改 `hadoop-demo` 项目的 `pom.xml` 文件，在 `<project></project>` 节点中添加以下内容（打包插件）：

```
<build>
```

```

<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>3.4.2</version>
    <configuration>
      <!-- 生成的 jar 包名称中是否追加 appendAssemblyId -->
      <appendAssemblyId>false</appendAssemblyId>
      <descriptorRefs>
        <!-- 将项目依赖的 jar 包打到当前 jar 包 -->
        <descriptorRef>jar-with-dependencies</descriptorRef>
      </descriptorRefs>
      <archive>
        <manifest>
          <!-- 打成可执行的 jar 包的主方法入口类 -->
          <mainClass>xxx.xxx.xxx.xxx.XxxXxx</mainClass>
        </manifest>
      </archive>
    </configuration>
    <!-- 插件目标列表 -->
    <executions>
      <!-- 将插件目标与生命周期阶段绑定 -->
      <execution>
        <!-- 插件目标 -->
        <goals>
          <!-- 只运行一次 -->
          <goal>single</goal>
        </goals>
        <!-- 生命周期阶段 -->
        <phase>package</phase>
      </execution>
    </executions>
  </plugin>
</plugins>
</build>

```

- Job代码

```

package com.yjxxt.mapred.wordcount;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

public class WordCountJob {

  public static void main(String[] args) throws IOException,
    InterruptedException, ClassNotFoundException {
    // 加载配置文件
    Configuration configuration = new Configuration(true);
    // 本地模式运行
    configuration.set("mapreduce.framework.name", "local");
    // 创建作业
    Job job = Job.getInstance(configuration);
    // 设置作业主类
    job.setJarByClass(WordCountJob.class);
    // 设置作业名称
    job.setJobName("yjx-wordcount-" + System.currentTimeMillis());
    // 设置 Reduce 的数量
    job.setNumReduceTasks(2);
    // 设置数据的输入路径（需要计算的数据从哪里读）
    FileInputFormat.setInputPaths(job, new Path("/yjx/harry potter.txt"));
    // 设置数据的输出路径（计算后的数据输出到哪里）
    FileOutputFormat.setOutputPath(job, new Path("/yjx/result/" + job.getJobName()));
    // 设置 Map 的输出的 Key 和 Value 的类型
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);
    // 设置 Map 和 Reduce 的处理类
    job.setMapperClass(WordCountMapper.class);
    job.setReducerClass(WordCountReducer.class);
    // 将作业提交到集群并等待完成
    job.waitForCompletion(true);
  }
}

```

```
}
```

- Job的Mapper代码

```
package com.yjxxt.mapred.wordcount;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

/**
 * KEYIN    当前行的偏移量
 * VALUEIN  当前行的数据
 * KEYOUT   输出的数据的 Key
 * VALUEOUT 输出的数据的 Value
 */
public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

    private IntWritable one = new IntWritable(1);

    /**
     * @param key      前行的偏移量
     * @param value    当前行的数据
     * @param context  可以理解为环形数据缓冲区
     * @throws IOException
     * @throws InterruptedException
     */
    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        // 替换特殊字符
        /*
         正则表达式解释:
         ^: 取反
         \w: 匹配字母、数字、下划线。等价于 [A-Za-z0-9_]
         ': 匹配 '
         -: 匹配 -
         +: 匹配 1 次或多次
         \s: 匹配所有空白符，包括换行
        */
        String line = value.toString().replaceAll("[^\\w'-]+", " ");
        // 切分字符串
        String[] words = line.split("\\s+");
        // 写出数据
        for (String word : words) {
            context.write(new Text(word), one);
        }
    }
}
```

- Job的Reducer代码

```
package com.yjxxt.mapred.wordcount;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

    /**
     * @param key      Map 的输出的 Key
     * @param values   Map 的输出的 Value
     * @param context
     * @throws IOException
     * @throws InterruptedException
     */
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
        // 声明计数器
        int count = 0;
        // 循环处理
        for (IntWritable value : values) {
```

```

        count += value.get();
    }
    // 写出数据
    context.write(key, new IntWritable(count));
}

}

```

6.2. Job提交方式

- Linux端执行方式
 - hadoop jar wordcount.jar com.yjxxt.mapred.wordcount.WordCountJob
- window端本地化执行
 - 拷贝Hadoop配置文件
 - configuration.set("mapreduce.framework.name", "local");

6.3. YARN 常用命令

- `yarn node -list -all` : 列出所有节点;
- `yarn application -list` : 列出所有 Application;
- `yarn application -list -appStates` : 根据 Application 的状态过滤 (所有状态: ALL、NEW、NEW_SAVING、SUBMITTED、ACCEPTED、RUNNING、FINISHED、FAILED、KILLED) ;
- `yarn logs -applicationId application_1667293209556_0001` : 查看 Application 的日志;
- `yarn logs -applicationId application_1667293209556_0001 -containerId container_e01_1667293209556_0001_01_000002` : 查询 Container 的日志;
- `yarn application -kill application_1667293209556_0001` : 根据 ApplicationID 杀死应用;
- `yarn container -list appattempt_1667293209556_0001_000001` : 列出 Application 的所有 Container;
- `yarn container -status container_e01_1667293209556_0001_01_000002` : 打印 Container 的状态; 只有在任务运行的过程中才能看到 Container 的状态。

7. MapReduce案例[充值记录]

7.1. Mock数据



```

package com.yjxxt.gok;

import org.apache.commons.io.FileUtils;
import org.apache.commons.lang3.RandomStringUtils;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import java.io.File;
import java.io.IOException;
import java.time.LocalDateTime;
import java.time.ZoneOffset;
import java.util.ArrayList;

```

```

import java.util.List;
import java.util.Random;

@DisplayName("王者荣耀测试类")
public class GloryOfKingsTest {

    // 开始时间
    private final long startTime = LocalDateTime.of(2021, 1, 1, 0, 0,
0).toInstant(ZoneOffset.of("+8")).toEpochMilli();
    // 结束时间
    private final long endTime = LocalDateTime.of(2022, 1, 1, 0, 0, 0).toInstant(ZoneOffset.of("+8")).toEpochMilli();

    @DisplayName("随机生成一万条充值记录")
    @Test
    public void mockData() throws IOException {
        // 声明一个容器存放数据
        List<String> recordList = new ArrayList<>();
        recordList.add("平台,大区,用户名,充值时间,性别,充值金额");
        // 数据格式: 平台,大区,用户名,充值时间,性别,充值金额
        Random random = new Random();
        for (int i = 0; i < 10000; i++) {
            // 随机平台
            String platform = random.nextInt(2) == 1 ? "微信" : "QQ";
            // 随机大区
            int area = random.nextInt(100) + 1;
            // 随机用户
            String username = RandomStringUtils.randomAlphanumeric(10);
            // 随机时间
            long time = (long) (Math.random() * (endTime - startTime) + startTime);
            // 随机性别 (男生多女生少)
            String gender = random.nextDouble() > 0.01 ? "man" : "woman";
            // 随机充值金额
            int money = random.nextInt(648) + 1;
            // 保存
            recordList.add(platform + "," + area + "," + username + "," + time + "," + gender + "," + money);
        }
        // 将数据写出到文件 (当前项目根路径下)
        FileUtils.writeLines(new File("mockdata/gok.txt"), recordList);
    }

}

```

- 按照性别分类统计出男女用户充值的总金额

8. MapReduce案例[天气信息]

北京主要地区天气预报

海淀	 阴	22 ~ 31°C	>	朝阳	 阴	24 ~ 32°C	>	顺义	 阴	23 ~ 32°C	>
怀柔	 阴	21 ~ 32°C	>	通州	 阴	23 ~ 32°C	>	昌平	 阴	21 ~ 32°C	>
丰台	 阴	21 ~ 31°C	>	石景山	 阴	22 ~ 31°C	>	大兴	 小雨	22 ~ 31°C	>
房山	 小雨	22 ~ 32°C	>	门头沟	 阴	21 ~ 32°C	>			展开	+

海淀15天天气详情

[切换] 2022年7月22日-2022年8月05日

今天天气

明天天气

一周天气

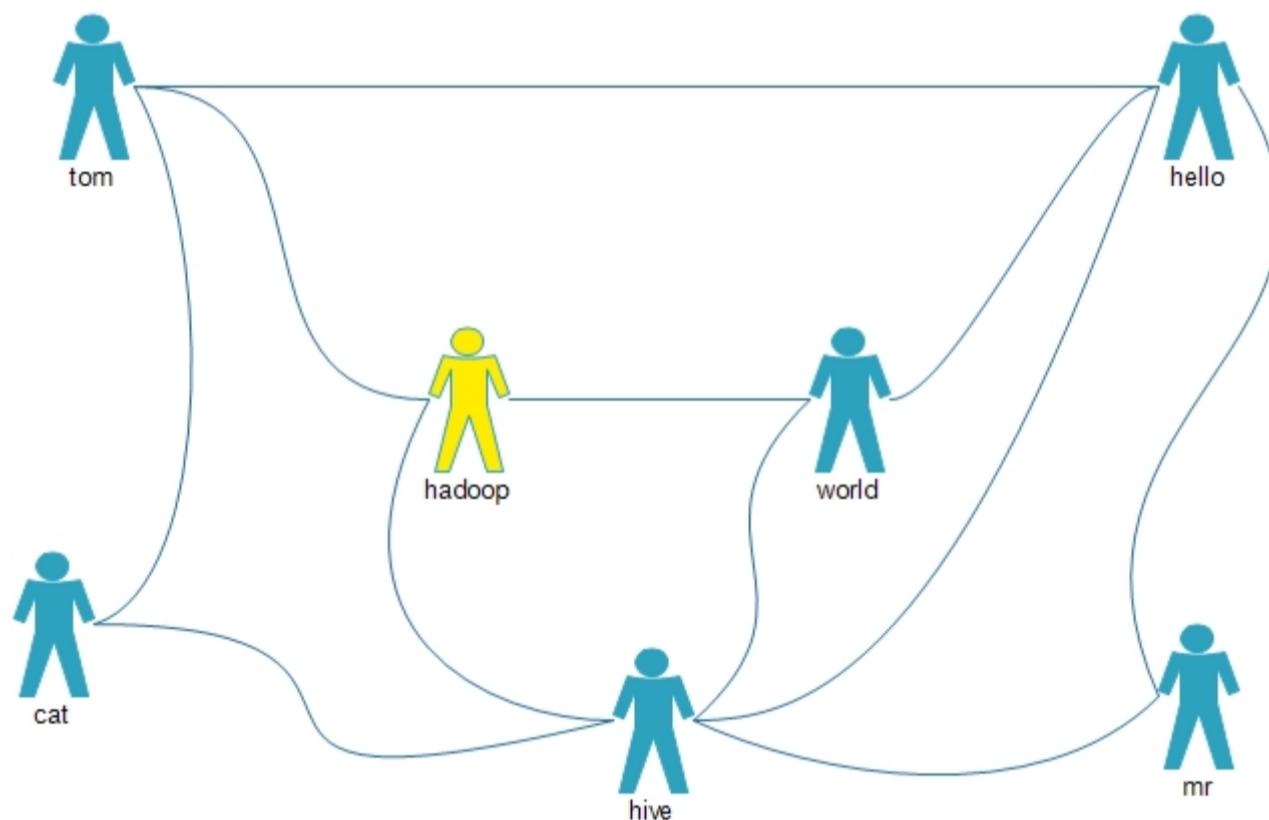
15天气



需求:

- 每个地区，每天的最高温度和最低温度分别是多少？
- 每个地区，每个月温度最高的三天是哪几天？

9. MapReduce案例[好友推荐]



```

tom hello hadoop cat
world hadoop hello hive
cat tom hive
mr hive hello
hive cat hadoop world hello mr
hadoop tom hive world
hello tom world hive mr
  
```

需求：如上图所示，两条线直接联系表示互为好友，例如 tom - cat， tom - hadoop，现在要根据这个图来给 hadoop 推荐好友。类似的功能很常见，比如抖音发现好友，微博感兴趣的人等。

10. MapReduce压缩

10.1. 概述

在时下大数据环境中，虽然机器性能好，节点多，但是并不代表我们的数据不需要做任何的压缩就开始处理。所以在某些情况下，我们还是需要对数据做压缩处理。**压缩技术能够有效减少存储系统的读写字节数，提高网络带宽和磁盘空间的效率。**

在 Hadoop 中，当数据规模很大，工作负载非常密集时，I/O 操作和网络数据传输需要花费大量的时间，Shuffle 与 Merge 过程同样也面临着巨大的 I/O 压力。在这种情况下，数据压缩的重要性不言而喻。而在 Hive 中则体现在数据文件最终存储的格式是否启用压缩。

鉴于磁盘 I/O 和网络带宽是 Hadoop 的宝贵资源，数据压缩对于节省资源、最小化磁盘 I/O 和网络传输非常有帮助，但其性能的提升和资源的节省并非没有代价（增加了 CPU 的运算负担）。如果磁盘 I/O 和网络带宽影响了 MapReduce 作业性能，在任意 MapReduce 阶段启用压缩都可以改善端到端处理时间并减少 I/O 和网络流量。

注意：压缩特性运用得当能提高性能，但运用不当也可能降低性能。

10.2. 条件与优缺点

- 为什么使用压缩（优点）：减少存储系统读写字节数、提高网络带宽和磁盘空间的效率。
- 压缩的缺点：使用数据时需要先对文件解压，加重 CPU 负载，压缩算法越复杂，解压时间越长。
- 压缩的条件：空间和 CPU 要充裕。如果机器 CPU 比较紧张，慎用压缩。
- 压缩的技术：
 - 有损压缩（LOSSY COMPRESSION）：压缩和解压的过程中数据有丢失，使用场景：视频。
 - 无损压缩（LOSSLESS COMPRESSION）：压缩和解压的过程中数据没有丢失，使用场景：日志数据。
- 对称和非对称：
 - 对称：压缩和解压的时间一致。
 - 非对称：压缩和解压的时间不一致。

提示：CPU 核心数是指一个 CPU 由几个核心组成，核心数越多，CPU 运行速度越快，比如处理同一份数据，单核是指一个人处理，双核是指两个人处理，所以核心数越多，CPU 性能越好。

10.3. 基本原则

计算密集型（CPU-Intensive）作业，少用压缩。

- 特点：要进行大量的计算，消耗 CPU 资源。比如计算圆周率、对视频进行高清解码等等，全靠 CPU 的运算能力。
- 计算密集型任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU 执行任务的效率就越低，所以，要最高效地利用 CPU，计算密集型任务同时进行的数量应当等于 CPU 的核心数。
- 计算密集型任务由于主要消耗 CPU 资源，因此，代码运行效率至关重要。Python 这样的脚本语言运行效率很低，完全不适合计算密集型任务。对于计算密集型任务，最好用 C 语言编写。

IO 密集型（IO-Intensive）作业，多用压缩。

- 特点：CPU 消耗很少，任务的大部分时间都在等待 IO 操作完成（因为 IO 的速度远远低于 CPU 和内存的速度）。
- 涉及到网络、磁盘 IO 的任务都是 IO 密集型任务。对于 IO 密集型任务，任务越多，CPU 效率越高，但也有一个限度。常见的大部分任务都是 IO 密集型任务，比如 Web 应用。
- IO 密集型任务执行期间，99% 的时间都花在 IO 上，花在 CPU 上的时间很少，因此，用运行速度极快的 C 语言替换 Python 这样运行速度极低的脚本语言，完全无法提升运行效率。对于 IO 密集型任务，最合适的语言就是开发效率最高（代码量最少）的语言，脚本语言是首选，C 语言最差。

10.4. 压缩实践

10.4.1. 压缩支持

使用 `hadoop checknative` 命令，可以查看是否有相应压缩算法的库，如果显示为 false，则需要额外安装。

```
[root@node01 ~]# hadoop checknative
2023-02-20 19:55:59,761 INFO bzip2.Bzip2Factory: Successfully loaded & initialized native-bzip2 library system-native
2023-02-20 19:55:59,764 INFO zlib.ZlibFactory: Successfully loaded & initialized native-zlib library
2023-02-20 19:55:59,771 WARN zstd.ZStandardCompressor: Error loading zstandard native libraries:
java.lang.InternalError: Cannot load libzstd.so.1 (libzstd.so.1: cannot open shared object file: No such file or
directory)!
2023-02-20 19:55:59,773 WARN erasurecode.ErasureCodeNative: Loading ISA-L failed: Failed to load libisal.so.2
(libisal.so.2: cannot open shared object file: No such file or directory)
```

```

2023-02-20 19:55:59,773 WARN erasurecode.ErasureCodeNative: ISA-L support is not available in your platform... using
builtin-java codec where applicable
2023-02-20 19:55:59,829 INFO nativeio.NativeIO: The native code was built without PMDK support.
Native library checking:
hadoop: true /opt/yjx/hadoop-3.3.4/lib/native/libhadoop.so.1.0.0
zlib: true /lib64/libz.so.1
zstd : false
bzip2: true /lib64/libbz2.so.1
openssl: false Cannot load libcrypto.so (libcrypto.so: cannot open shared object file: No such file or directory)!
ISA-L: false Loading ISA-L failed: Failed to load libisal.so.2 (libisal.so.2: cannot open shared object file: No
such file or directory)
PMDK: false The native code was built without PMDK support.

```

P.S.: Hadoop 2.X 版本已经集成了 Snappy、LZ4、BZip2 等压缩算法的编/解码器，会自动调用对应的本地库，而 CentOS7 中又自带了 Snappy 的依赖库，所以无需安装 Snappy 依赖。

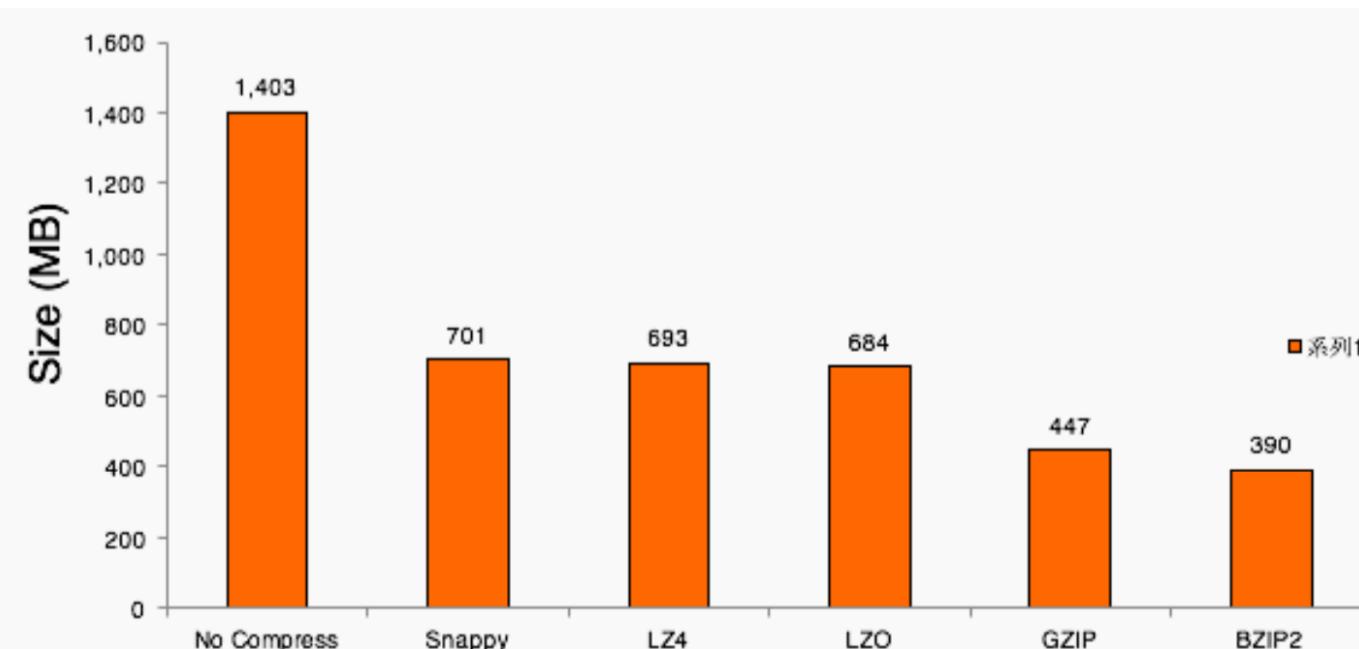
10.4.2. 压缩比较

压缩格式	算法	文件扩展名	是否可切分	压缩比	压缩速度	解压速度
DEFLATE	DEFLATE	.deflate	否	高	低	低
Gzip	DEFLATE	.gz	否	高	低	低
BZip2	BZip2	.bz2	是	高	低	低
LZO	LZO	.lzo	是 (需建索引)	低	高	高
LZ4	LZ4	.lz4	否	低	高	高
Snappy	Snappy	.snappy	否	低	高	高
Zstd	Zstd	.zst	否	高	高	高

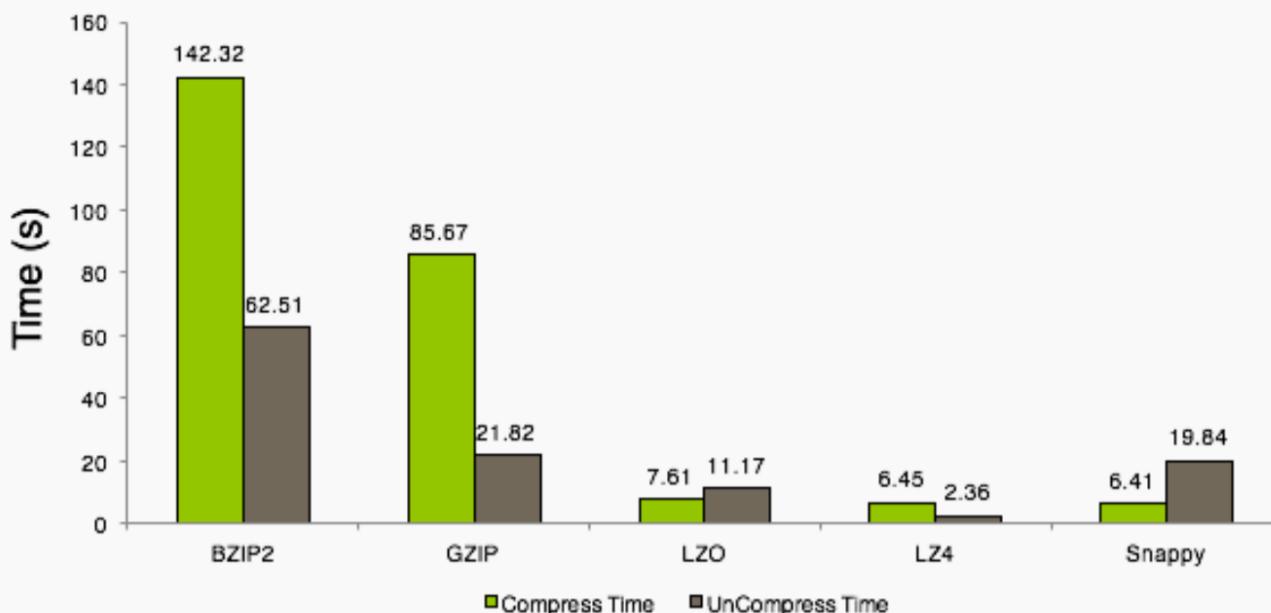
- 文件扩展名：压缩后的数据文件的后缀名。
- 是否可切分：表示压缩后的数据文件在被 MapReduce 读取的时候，是否会产生多个 InputSplit。如果这个压缩格式产生的文件不可切分，那也就意味着，无论这个压缩文件有多大，在 MapReduce 中都只会产生 1 个 Map 任务。如果压缩后的文件不大，也就 100M 左右，这样对性能没有多大影响。但是如果压缩后的文件比较大，达到了 1 个 G，由于不可切分，这样只能使用 1 个 Map 任务去计算，性能就比较差了，这个时候就没有办法达到并行计算的效果了。所以是否可切分这个特性是非常重要的，特别是当我们无法控制单个压缩文件大小的时候。
- 压缩比：表示压缩格式的压缩效果，压缩比越高，说明压缩效果越好，对应产生的压缩文件就越小。如果集群的存储空间有限，则需要重点关注压缩比，这个时候需要选择尽可能高的压缩比。
- 压缩速度：表示将原始文件压缩为指定压缩格式消耗的时间。压缩功能消耗的时间会体现在任务最终消耗的时间里面，所以这个指标也需要重点考虑。
- 解压速度：表示将指定压缩格式的数据文件解压为原始文件消耗的时间。因为 MapReduce 在使用压缩文件的时候需要先进行解压才能使用，解压消耗的时间也会体现在任务最终消耗的时间里面，所以这个指标也需要重点考虑。

存放数据到 HDFS 时，可以通过配置指定数据的压缩方式。当 MapReduce 程序读取数据时，会根据扩展名自动解压。例如：如果文件扩展名为 `.snappy`，Hadoop 框架会自动使用 SnappyCodec 解压缩文件。

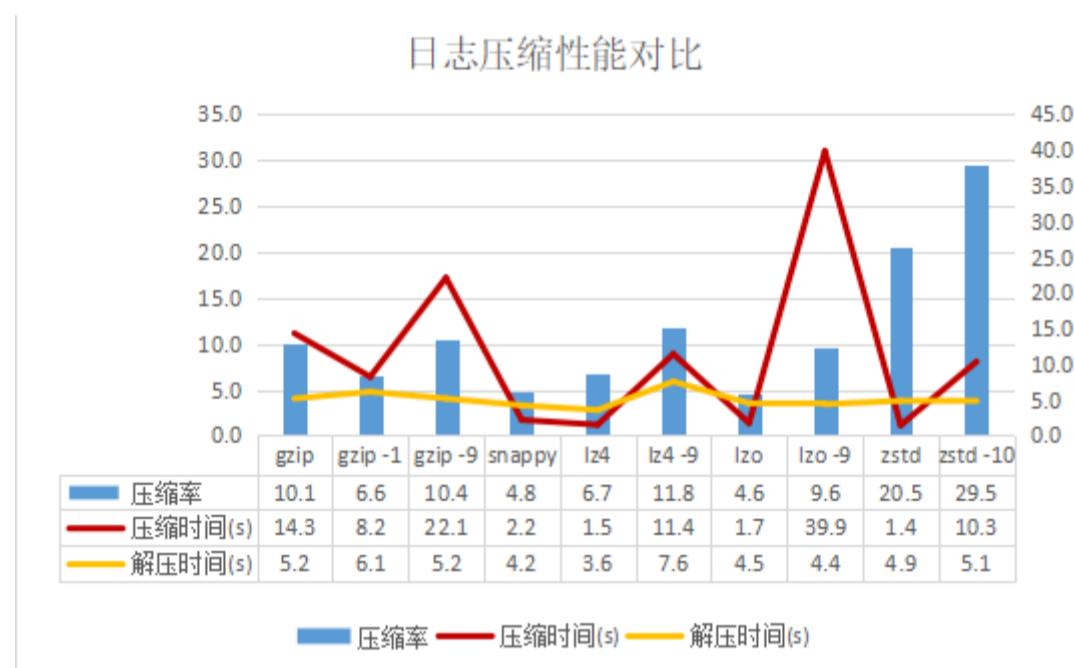
下图为压缩比。



下图为压缩/解压缩时间。



下图为 100 万行不重复的日志文件（大小为 977MB）的压缩性能对比结果。



LZO, LZ4, Snappy 等压缩算法专注于压缩和解压缩性能，Zstd 在性能不错的同时号称压缩率跟 Deflate (Zip/Gzip 的算法) 相当。Linux 内核、HTTP 协议、以及一系列的大数据工具 (包括 Hadoop 3.0.0, HBase 2.0.0, Spark 2.3.0, Kafka 2.1.0) 等都已经加入了对 Zstd 的支持。

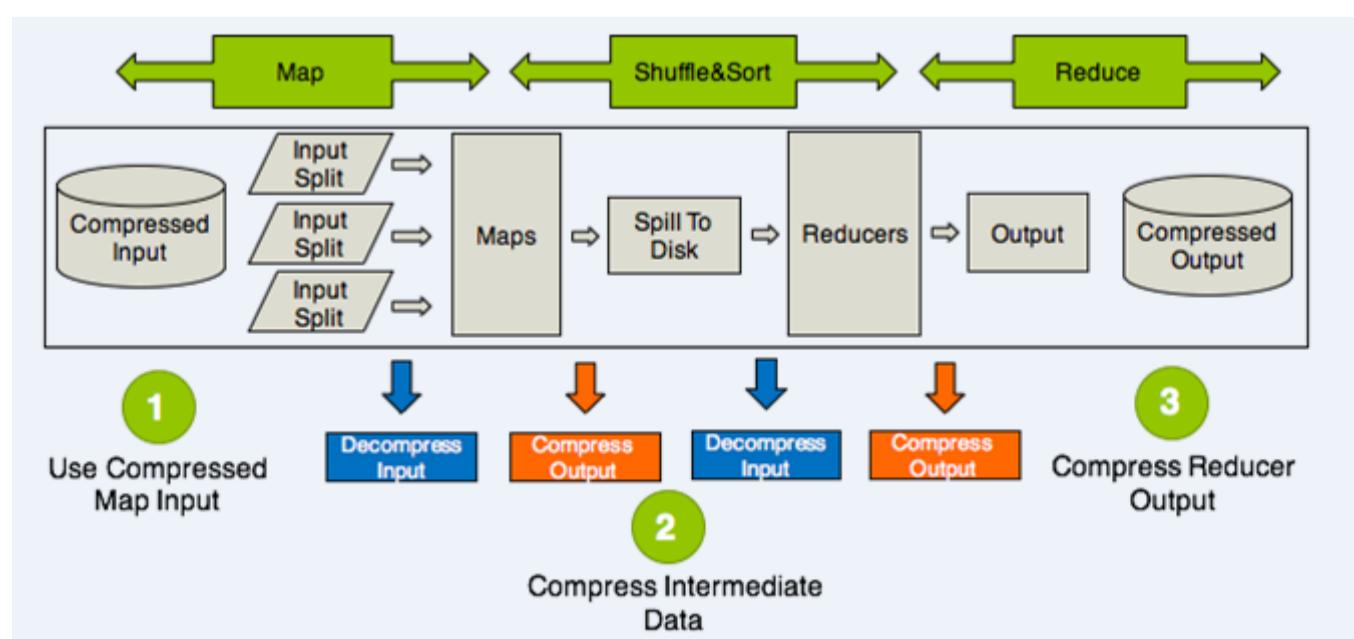
10.4.3. 压缩选择

这些压缩文件最终都要经过 MR 程序处理，所以我们需要知道 MR 程序在哪些地方可以压缩，分别选什么样的压缩算法比较合适。

如下图，MR 主要在三个地方会用到数据压缩：

- **Input**：数据来源
- **Transformation**：中间计算
- **Output**：最后的输出

下面我们就针对这三个部分，来做对应的压缩选型。



1 Use Compressed Map Input	2 Compress Intermediate Data	3 Compress Reducer Output
<ul style="list-style-type: none"> Mapreduce jobs read input from HDFS Compress if input data is large. This will reduce disk read cost. Compress with splittable algorithms like Bzip2 Or use compression with splittable file structures such as Sequence Files, RC Files etc. 	<ul style="list-style-type: none"> Map output is written to disk (spill) and transferred across the network Always use compression to reduce both disk write, and network transfer load Beneficial in performance point of view even if input and output is uncompressed Use faster codecs such as Snappy, LZO 	<ul style="list-style-type: none"> Mapreduce output used for both archiving or chaining mapreduce jobs Use compression to reduce disk space for archiving Compression is also beneficial for chaining jobs especially with limited disk throughput resource. Use compression methods with higher compress ratio to save more disk space

Use Compressed Map Input : 第一次传入压缩文件，应选用可以切片的压缩方式，否则整个文件将只有一个 Map 执行。建议：从 HDFS 中读取文件进行 MapReduce 作业，如果数据很大，可以使用压缩并且选择支持分片的压缩方式，例如 BZip2、LZO，这样可以实现并行处理，提高效率，减少磁盘读取时间，同时选择合适的存储格式例如 Sequence Files、RC、ORC 等。

Compress Intermediate Data : 第二次压缩应选择压缩解压速度快的压缩方式。建议：Map 的输出作为 Reducer 的输入，需要经过 Shuffle 这一过程，需要把数据读取到环形数据缓冲区，然后再读取到本地磁盘，所以选择压缩可以减少了存储文件所占空间，提升了数据传输速率，建议使用压缩/解压速度快的压缩方式，例如 Snappy、LZO、LZ4、Zstd。

Compress ReducerOutput : 第三次压缩有两种场景分别是：

- 当输出的文件为下一个 Job 的输入时，建议：选择可切分的压缩方式例如：BZip2。
- 当输出的文件直接存到 HDFS 作为归档时，建议：选择压缩比高的压缩方式。Reduce 阶段数据落盘通常使用 Gzip 或 BZip2 进行压缩（减少磁盘使用）。

总结：

- Gzip: Hadoop 内置支持，压缩比高，不支持 Split。
 - 用途：通常用来放不常访问的冷数据，较高的压缩比可以极大的节省磁盘空间。
 - 对应的编码/解码器：`org.apache.hadoop.io.compress.GzipCodec`。
- BZip2: Hadoop 内置支持，压缩比高，支持 Split，支持多文件，缺点就是慢。
 - 用途：适用于对处理速度要求不高的场景。一般不常用。
 - 对应的编码/解码器：`org.apache.hadoop.io.compress.BZip2Codec`。
- LZO: 压缩比一般，支持 Split (需要建索引，文件修改后需要重新建索引)，压缩/解压速度快，支持 Hadoop Native 库，需要自己安装。
 - 用途：适合于经常访问的热数据。
 - 对应的编码/解码器：`com.hadoop.compression.lzo.LzopCodec`。
- LZ4: 压缩比一般，不支持 Split，压缩/解压速度快，支持 Hadoop Native 库，需要自己安装。
 - 用途：和 LZO 性能类似，但不支持 Split，可以用于 Map 中间结果的压缩。
 - 对应的编码/解码器：`org.apache.hadoop.io.compress.Lz4Codec`。
- Snappy: 压缩比一般，不支持 Split，压缩/解压速度快，支持 Hadoop Native 库，需要自己安装。
 - 用途：和 LZO 性能类似，但不支持 Split，可以用于 Map 中间结果的压缩。
 - 对应的编码/解码器：`org.apache.hadoop.io.compress.SnappyCodec`。
- Zstd: 压缩比高跟 Deflate (Gzip 算法) 相当，不支持 Split，压缩/解压速度快，支持 Hadoop Native 库，需要自己安装。
 - 用途：和 LZO 性能类似，但不支持 Split，可以用于 Map 中间结果的压缩。
 - 对应的编码/解码器：`org.apache.hadoop.io.compress.ZStandardCodec`。

10.4.4. 压缩配置

要在 Hadoop 中启用压缩，需要配置以下参数。

`core-site.xml`

```
<!-- 可用于压缩/解压缩的编解码器，用逗号分隔列表 -->
<property>
    <name>io.compression.codecs</name>
    <value>
        org.apache.hadoop.io.compress.DefaultCodec,
        org.apache.hadoop.io.compress.GzipCodec,
        org.apache.hadoop.io.compress.BZip2Codec,
        com.hadoop.compression.lzo.LzopCodec,
        org.apache.hadoop.io.compress.Lz4Codec,
        org.apache.hadoop.io.compress.SnappyCodec,
        org.apache.hadoop.io.compress.ZStandardCodec
    </value>
</property>
```

mapred-site.xml

```
<!-- 开启 Mapper 输出压缩 -->
<property>
    <name>mapreduce.map.output.compress</name>
    <value>true</value>
</property>
<!-- 设置 Mapper 输出压缩的压缩方式 -->
<property>
    <name>mapreduce.map.output.compress.codec</name>
    <value>org.apache.hadoop.io.compress.SnappyCodec</value>
</property>
<!-- 开启 Reducer 输出压缩 -->
<property>
    <name>mapreduce.output.fileoutputformat.compress</name>
    <value>true</value>
</property>
<!-- 设置 Reducer 输出压缩的压缩方式 -->
<property>
    <name>mapreduce.output.fileoutputformat.compress.codec</name>
    <value>org.apache.hadoop.io.compress.BZip2Codec</value>
</property>
<!-- SequenceFiles 输出可以使用的压缩类型：NONE、RECORD 或者 BLOCK -->
<!-- 如果作业输出被压缩为 SequenceFiles，该属性用来控制使用的压缩格式。默认为 RECORD，即针对每条记录进行压缩，如果将其改为 BLOCK，将针对一组记录进行压缩，这是推荐的压缩策略，因为它的压缩效率更高。 -->
<property>
    <name>mapreduce.output.fileoutputformat.compress.type</name>
    <value>BLOCK</value>
</property>
```

除了使用配置文件的方式指定压缩器（优先考虑）外，还可以使用编码的方式进行配置。

```
// 加载配置文件
Configuration configuration = new Configuration(true);
// 开启 Mapper 输出压缩
configuration.setBoolean(Job.MAP_OUTPUT_COMPRESS, true);
configuration.setClass(Job.MAP_OUTPUT_COMPRESS_CODEC, SnappyCodec.class, CompressionCodec.class);
// 创建作业
Job job = Job.getInstance(configuration);

// ... Job 的其他设置

// 开启 Reducer 输出压缩
FileOutputFormat.setCompressOutput(job, true);
FileOutputFormat.setOutputCompressorClass(job, BZip2Codec.class);
// 将作业提交到集群并等待完成
job.waitForCompletion(true);
```

10.4.5. 压缩实践

修改之前 MR 的天气信息 Job 代码如下（Mapper 和 Reducer 的代码不动）：

```
package com.yjxxt.mapred.weather.job01;

import com.yjxxt.wordcount.WordCountJob;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.compress.BZip2Codec;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.hadoop.io.compress.SnappyCodec;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
```

```

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

public class WeatherCompressJob {

    public static void main(String[] args) {
        try {
            // 加载配置文件
            Configuration configuration = new Configuration(true);
            // 本地模式运行
            // configuration.set("mapreduce.framework.name", "local");
            // 开启 Mapper 输出压缩
            configuration.setBoolean(Job.MAP_OUTPUT_COMPRESS, true);
            configuration.setClass(Job.MAP_OUTPUT_CODEC, SnappyCodec.class, CompressionCodec.class);
            // 创建作业
            Job job = Job.getInstance(configuration);
            // 设置作业主类
            job.setJarByClass(WeatherCompressJob.class);
            // 设置作业名称
            job.setJobName("yjx-weather-compress-" + System.currentTimeMillis());
            // 设置 Reduce 的数量
            job.setNumReduceTasks(2);
            // 设置数据的输入路径（需要计算的数据从哪里读）
            FileInputFormat.setInputPaths(job, new Path("/yjx/weather.csv"));
            // 设置数据的输出路径（计算后的数据输出到哪里）
            FileOutputFormat.setOutputPath(job, new Path("/yjx/result/" + job.getJobName()));
            // 设置 Map 的输出的 Key 和 Value 的类型
            job.setMapOutputKeyClass(Text.class);
            job.setMapOutputValueClass(IntWritable.class);
            // 设置 Map 和 Reduce 的处理类
            job.setMapperClass(WeatherMapper.class);
            job.setReducerClass(WeatherReducer.class);
            // 开启 Reducer 输出压缩
            FileOutputFormat.setCompressOutput(job, true);
            FileOutputFormat.setOutputCompressorClass(job, BZip2Codec.class);
            // 将作业提交到集群并等待完成
            job.waitForCompletion(true);
        } catch (IOException | InterruptedException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

```

将 Job 打包并上传至 Hadoop 服务器。

运行命令: `hadoop jar xxxx.jar com.yjxxt.mapred.weather.compress.WeatherCompressJob`

不压缩运行结果如下:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	root	supergroup	0 B	Feb 20 17:05	2	128 MB	_SUCCESS
-rw-r--r--	root	supergroup	142.57 KB	Feb 20 17:05	2	128 MB	part-r-00000
-rw-r--r--	root	supergroup	143.42 KB	Feb 20 17:05	2	128 MB	part-r-00001

压缩运行结果如下:

Hadoop Overview Datanodes Datanode Volume Failures Snapshot Startup Progress Utilities ▾

Browse Directory

/bd/result/bd-weather-compress-1676894427888 Go! File Up Download Print

Show 25 entries Search:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	root	supergroup	0 B	Feb 20 20:01	2	128 MB	_SUCCESS
-rw-r--r--	root	supergroup	6.05 KB	Feb 20 20:01	2	128 MB	part-r-00000.bz2
-rw-r--r--	root	supergroup	6.01 KB	Feb 20 20:01	2	128 MB	part-r-00001.bz2

Showing 1 to 3 of 3 entries Previous 1 Next

Hadoop, 2022.



Map 到 Reduce 阶段的压缩可以通过日志查看：

Browsing HDFS node01:19888/jobhistory/logs × node01:19888/jobhistory/logs/node03:39024/container_e01_1676880964827_0003_01_00002/attempt_1676880964827_0003_m_000000_0... 已暂停

Log Type: syslog
Log Upload Time: Mon Feb 20 20:01:10 +0800 2023
Log Length: 49459

```

2023-02-20 20:00:50,165 INFO [main] org.apache.hadoop.security.SecurityUtil: Updating Configuration
2023-02-20 20:00:50,323 INFO [main] org.apache.hadoop.metrics2.impl.MetricsConfig: Loaded properties from hadoop-metrics2.properties
2023-02-20 20:00:50,494 INFO [main] org.apache.hadoop.metrics2.impl.MetricsSystemImpl: Scheduled Metric snapshot period at 10 second(s).
2023-02-20 20:00:50,619 INFO [main] org.apache.hadoop.mapred.YarnChild: Executing with tokens: [Kind: mapreduce.job.Service: job_1676880964827_0003, Ident: (org.apache.hadoop.mapreduce.service.JobService@1676880964827_0003, org.apache.hadoop.mapreduce.jobhistory.JobHistory@1676880964827_0003)]
2023-02-20 20:00:50,678 INFO [main] org.apache.hadoop.mapred.YarnChild: Sleeping for 0ms before retrying again. Got null now.
2023-02-20 20:00:50,963 INFO [main] org.apache.hadoop.mapred.YarnChild: mapreduce.cluster.local.dir for child: /var/bd/hadoop/ha/nm-local-dir/usercache/root/appcache/application_1676880964827_0003
2023-02-20 20:00:51,577 INFO [main] org.apache.hadoop.mapred.YarnChild: *****
[system properties]
os.name: Linux
os.version: 3.10.0-1160.e17.x86_64
java.home: /usr/java/jdk-11.0.16.1
java.runtime.version: 11.0.16.1+1-LTS-1
java.vendor: Oracle Corporation
java.version: 11.0.16.1
java.vm.name: Java HotSpot(TM) 64-Bit Server VM
java.class.path: /var/bd/hadoop/ha/nm-local-dir/usercache/root/appcache/application_1676880964827_0003/container_e01_1676880964827_0003_01_000002:/opt/bd/hadoop-3.3.4/etc/hadoop:/opt/bd/hadoop-3.3.4/libexec
java.io.tmpdir: /var/bd/hadoop/ha/nm-local-dir/usercache/root/appcache/application_1676880964827_0003/container_e01_1676880964827_0003_01_000002/tmp
user.dir: /var/bd/hadoop/ha/nm-local-dir/usercache/root/appcache/application_1676880964827_0003/container_e01_1676880964827_0003_01_000002
user.name: root
*****
Task Overview Counters
*****
Tools

```



11. MapReduce源码分析[学习思想]

11.1. Split

- 快捷键

```

ctrl + alt + 方向键 : 查看上一个或者下一个方法
ctrl + shift + alt + C 拷贝方法的全名
ctrl + alt + b 查看当前接口的实现类

```

- 源代码的分析从提交任务开始

```

job.waitForCompletion(true);

```

- org.apache.hadoop.mapreduce.Job#waitForCompletion

```
//Submit the job to the cluster and wait for it to finish.
//判断当前的状态
if (state == JobState.DEFINE) {
    //-----关键代码
    submit();
}
//监控任务的运行状态
if (verbose) {
    //Monitor a job and print status in real-time as progress is made and tasks fail.
    monitorAndPrintJob();
}
//返回任务状态
return.isSuccessful();
```

- org.apache.hadoop.mapreduce.Job#submit

```
//确认当前任务的状态
ensureState(JobState.DEFINE);
//mapreduce1.x和2.x,但是2的时候将1的好多方法进行了优化
setUseNewAPI();
//获取当前任务所运行的集群
connect();
//Provides a way to access information about the map/reduce cluster.
cluster = new Cluster(getConfiguration());
//创建Job的提交器
final JobSubmitter submitter = getJobSubmitter(cluster.getFileSystem(), cluster.getClient());
//提交任务到系统去执行
//-----关键代码
//Internal method for submitting jobs to the system
status = submitter.submitJobInternal(Job.this, cluster)
//任务的状态修改为运行
state = JobState.RUNNING;
```

- org.apache.hadoop.mapreduce.JobSubmitter#submitJobInternal

```
//验证job输出
checkSpecs(job);
//生成并设置新的JobID
JobID jobId = submitClient.getNewJobID();
job.setJobID(jobId);
//获取任务的提交目录
Path submitJobDir = new Path(jobStagingArea, jobId.toString());
//-----关键代码
// Create the splits for the job 197行
int maps = writeSplits(job, submitJobDir);
//设置map的数量, 其中map的数量就等于切片的数量
conf.setInt(MRJobConfig.NUM_MAPS, maps);
```

- org.apache.hadoop.mapreduce.JobSubmitter#writeSplits

```
//-----关键代码
//使用新API
maps = writeNewSplits(job, jobSubmitDir);
```

- org.apache.hadoop.mapreduce.JobSubmitter#writeNewSplits

```
//获取配置文件
Configuration conf = job.getConfiguration();
//-----关键代码InputFormat
//获取文件读取器 org.apache.hadoop.mapreduce.lib.input.TextInputFormat
InputFormat<?, ?> input = ReflectionUtils.newInstance(job.getInputFormatClass(), conf);

//-----关键代码getSplits
List<InputSplit> splits = input.getSplits(job);
//将List转成数组
T[] array = (T[]) splits.toArray(new InputSplit[splits.size()]);

// sort the splits into order based on size, so that the biggest
Arrays.sort(array, new SplitComparator());
//任务创建切片文件
JobSplitWriter.createSplitFiles(jobSubmitDir, conf, jobSubmitDir.getFileSystem(conf), array);
//返回切片的数目
return array.length;
```

- org.apache.hadoop.mapreduce.task.JobContextImpl#getInputFormatClass

```
//返回创建的TextInputFormat对象
return (Class<? extends InputFormat<?, ?>>) conf.getClass(INPUT_FORMAT_CLASS_ATTR, TextInputFormat.class);

//getClass的操作如果有值返回值, 没有的话使用默认值
getClass(String name, Class<?> defaultValue)
```

- org.apache.hadoop.mapreduce.lib.input.FileInputFormat#getSplits
 - public class TextInputFormat extends FileInputFormat<LongWritable, Text>

```

//Generate the list of files and make them into FileSplits.
//Math.max(1,1)
//getFormatMinSplitSize()一个切片最少应该拥有1个字节
//getMinSplitSize(job) 读取程序员设置的切片的最小值, 如果没有设置默认读取1
long minSize = Math.max(getFormatMinSplitSize(), getMinSplitSize(job));
//读取程序员设置的切片的最大值, 如果没有设置默认读取Long.MAX_VALUE
long maxSize = getMaxSplitSize(job);

//创建一个List存放切片
List<InputSplit> splits = new ArrayList<InputSplit>();
//获取要分析的文件列表
List<FileStatus> files = listStatus(job);

//开始遍历要分析文件的路径
for (FileStatus file : files) {
    //获取文件路径
    Path path = file.getPath();
    //获取文件的长度, 文件拥有的字节数
    long length = file.getLen();
    //如果文件长度不为0
    if (length != 0) {
        //获取文件对应的Blocks信息
        BlockLocation[] blkLocations;
        if (file instanceof LocatedFileStatus) {
            blkLocations = ((LocatedFileStatus) file).getBlockLocations();
        } else {
            FileSystem fs = path.getFileSystem(job.getConfiguration());
            blkLocations = fs.getFileBlockLocations(file, 0, length);
        }
        //判断文件是否可以进行切片
        if (isSplittable(job, path)) {
            //获取Block的大小
            long blockSize = file.getBlockSize();
            //切片的默认大小为 128M
            //blockSize 128M, minSize 1byte, maxSize long.MaxValueBytes
            //return Math.max(minSize, Math.min(maxSize, blockSize));
            //minSize 64M ----> 128M
            //minSize 256M ----> 256M
            //maxSize 64M ----> 64M
            //maxSize 256M ---->128M
            long splitSize = computeSplitSize(blockSize, minSize, maxSize);

            //声明一个变量存放字节的余额 256M
            long bytesRemaining = length;
            //查看剩余的容量是否能达到阈值 SPLIT_SLOP
            //private static final double SPLIT_SLOP = 1.1
            while (((double) bytesRemaining) / splitSize > SPLIT_SLOP) {
                int blkIndex = getBlockIndex(blkLocations, length - bytesRemaining);
                //这个方法专门用来创建切片
                //切片生成之后添加到List
                //org.apache.hadoop.mapreduce.lib.input.FileInputFormat#makeSplit
                splits.add(makeSplit(path, length - bytesRemaining, splitSize,
                                     blkLocations[blkIndex].getHosts(),
                                     blkLocations[blkIndex].getCachedHosts()));
                //每次创建切片后, 将使用的部分删除
                bytesRemaining -= splitSize;
            }
            //判断剩余的容量是否为0
            //最后一个切片的数据范围是(0,1 , 1.1]
            if (bytesRemaining != 0) {
                int blkIndex = getBlockIndex(blkLocations, length - bytesRemaining);
                splits.add(makeSplit(path, length - bytesRemaining, bytesRemaining,
                                     blkLocations[blkIndex].getHosts(),
                                     blkLocations[blkIndex].getCachedHosts()));
            }
            } else { // not splittable
                //如果发现文件不能切片, 将整个文件作为一个切片
                splits.add(makeSplit(path, 0, length, blkLocations[0].getHosts(),
                                     blkLocations[0].getCachedHosts()));
            }
        } else {
            //Create empty hosts array for zero length files
            splits.add(makeSplit(path, 0, length, new String[0]));
        }
    }
    // Save the number of input files for metrics/loadgen
    job.getConfiguration().setLong(NUM_INPUT_FILES, files.size());
}

```

```
//返回切片的容器
return splits;
```

11.2. MapTask

- org.apache.hadoop.mapred.MapTask#run

```
//使用新的API
boolean useNewApi = job.getUseNewMapper();
//-----关键代码initialize
//初始化MapTask
initialize(job, getJobID(), reporter, useNewApi);
//-----关键代码runNewMapper
//开始运行Task
runNewMapper(job, splitMetaInfo, umbilical, reporter);
```

- org.apache.hadoop.mapred.Task#initialize

```
//JOB的上下文参数
jobContext = new JobContextImpl(job, id, reporter);
//Map的上下文参数
taskContext = new TaskAttemptContextImpl(job, taskId, reporter);
//创建Map数据的写出器
outputFormat = ReflectionUtils.newInstance(taskContext.getOutputFormatClass(), job);
    //真正的写出对象
    org.apache.hadoop.mapreduce.task.JobContextImpl#getOutputFormatClass
        return (Class<? extends OutputFormat<?, ?>>)
            conf.getClass(OUTPUT_FORMAT_CLASS_ATTR, TextOutputFormat.class);
//创建Map任务的提交器
committer = outputFormat.getOutputCommitter(taskContext);
    //真正的提交器对象
    org.apache.hadoop.mapreduce.lib.output.FileOutputFormat#getOutputCommitter
        committer = new FileOutputCommitter(output, context);
//获取写出的路径
Path outputPath = FileOutputFormat.getOutputPath(conf);
```

- org.apache.hadoop.mapred.MapTask#runNewMapper

```
// make a task context so we can get the classes
org.apache.hadoop.mapreduce.TaskAttemptContext taskContext =
    new org.apache.hadoop.mapreduce.task.TaskAttemptContextImpl(job, getTaskID(), reporter);
// make a mapper--com.yjx.wordcount.WordCountMapper
org.apache.hadoop.mapreduce.Mapper<INKEY, INVALUE, OUTKEY, OUTVALUE> mapper =
    (org.apache.hadoop.mapreduce.Mapper<INKEY, INVALUE, OUTKEY, OUTVALUE>)
        ReflectionUtils.newInstance(taskContext.getMapperClass(), job);
// make the input format--org.apache.hadoop.mapreduce.lib.input.TextInputFormat
org.apache.hadoop.mapreduce.InputFormat<INKEY, INVALUE> inputFormat =
    (org.apache.hadoop.mapreduce.InputFormat<INKEY, INVALUE>)
        ReflectionUtils.newInstance(taskContext.getInputFormatClass(), job);
// rebuild the input split
org.apache.hadoop.mapreduce.InputSplit split = null;
split = getSplitDetails(new Path(splitIndex.getSplitLocation()), splitIndex.getStartOffset());
// 创建记录读取器
org.apache.hadoop.mapreduce.RecordReader<INKEY, INVALUE> input =
    new NewTrackingRecordReader<INKEY, INVALUE>(split, inputFormat, reporter, taskContext);
    //创建真正的读取器
    //org.apache.hadoop.mapred.MapTask.NewTrackingRecordReader#NewTrackingRecordReader
    this.real = inputFormat.createRecordReader(split, taskContext);
        //使用inputFormat创建读取器
        //org.apache.hadoop.mapreduce.lib.input.TextInputFormat#createRecordReader
        return new LineRecordReader(recordDelimiterBytes);

// 创建记录写出器
org.apache.hadoop.mapreduce.RecordWriter output = null;
output = new NewOutputCollector(taskContext, job, umbilical, reporter);
// 创建Map的上下文对象
org.apache.hadoop.mapreduce.MapContext<INKEY, INVALUE, OUTKEY, OUTVALUE>
    mapContext =
        new MapContextImpl<INKEY, INVALUE, OUTKEY, OUTVALUE>(job, getTaskID(),
            input, output,
            committer,
            reporter, split);
// 创建mapContext的包装类
org.apache.hadoop.mapreduce.Mapper<INKEY, INVALUE, OUTKEY, OUTVALUE>.Context
    mapperContext =
        new WrappedMapper<INKEY, INVALUE, OUTKEY, OUTVALUE>().getMapContext(
            mapContext);
// 初始化切片信息
```

```

input.initialize(split, mapperContext);
//开始执行Mapper方法，就是自己的Mapper实现类
mapper.run(mapperContext);
mapPhase.complete();
setPhase(TaskStatus.Phase.SORT);
statusUpdate(umbilical);
//关闭输入
input.close();
input = null;
//关闭输出（将缓冲区最后的数据写出，并合并这些文件）
output.close(mapperContext);
output = null;

```

- org.apache.hadoop.mapred.MapTask.NewTrackingRecordReader#initialize

```

//LineRecordReader执行初始化
real.initialize(split, context);

```

- org.apache.hadoop.mapreduce.lib.input.LineRecordReader#initialize

```

//获取切片
FileSplit split = (FileSplit) genericSplit;
//配置信息
Configuration job = context.getConfiguration();
//一行最多读取的数据量
this.maxLineLength = job.getInt(MAX_LINE_LENGTH, Integer.MAX_VALUE);
//获取切片的开始和结束偏移量
start = split.getStart();
end = start + split.getLength();
//获取文件路径
final Path file = split.getPath();

// open the file and seek to the start of the split
final FileSystem fs = file.getFileSystem(job);
fileIn = fs.open(file);

//将读取器定位到切片的开始位置
fileIn.seek(start);
//创建输入流
in = new UncompressedSplitLineReader(fileIn, job, this.recordDelimiterBytes, split.getLength());
filePosition = fileIn;
// If this is not the first split, we always throw away first record
// because we always (except the last split) read one extra line in
// next() method.
if (start != 0) {
    start += in.readLine(new Text(), 0, maxBytesToConsume(start));
}
this.pos = start;

```

- org.apache.hadoop.mapreduce.Mapper#run

```

//初始化
setup(context);
try {
    //1判断是否为最后一行2为key设置值3为value设置值
    while (context.nextKeyValue()) {
        //三个参数分别为：key value
        map(context.getCurrentKey(), context.getCurrentValue(), context);
    }
} finally {
    //清空操作
    cleanup(context);
}

```

- org.apache.hadoop.mapreduce.lib.input.LineRecordReader#nextKeyValue

```

//偏移量
key = new LongWritable();
//设置本次读取的开始位置
key.set(pos);
//一行数据
value = new Text();
//We always read one extra line 读取一行的数据
if (pos == 0) {
    newSize = skipUtfByteOrderMark();
} else {
    newSize = in.readLine(value, maxLineLength, maxBytesToConsume(pos));
    //下次读取数据的位置
    pos += newSize;
}

```

- org.apache.hadoop.mapreduce.lib.input.LineRecordReader#skipUtfByteOrderMark

```
//每次空读一行数据，绕过第一行代码
int newSize = in.readLine(value, newMaxLineLength, maxBytesToConsume(pos));
pos += newSize;
```

11.3. KvBuffer

- org.apache.hadoop.mapred.MapTask.NewOutputCollector#NewOutputCollector

```
//创建收集器
collector = createSortingCollector(job, reporter);
//获取reduce的数量
partitions = jobContext.getNumReduceTasks();
if (partitions > 1) {
    partitioner = (org.apache.hadoop.mapreduce.Partitioner<K, V>)
        ReflectionUtils.newInstance(jobContext.getPartitionerClass(), job);
} else {
    partitioner = new org.apache.hadoop.mapreduce.Partitioner<K, V>() {
        @Override
        public int getPartition(K key, V value, int numPartitions) {
            return partitions - 1;
        }
    };
}
```

- org.apache.hadoop.mapred.MapTask#createSortingCollector

```
//获取上下文对象
MapOutputCollector.Context context = new MapOutputCollector.Context(this, job, reporter);
//获取收集器的Class
Class<?>[] collectorClasses = job.getClasses(
    JobContext.MAP_OUTPUT_COLLECTOR_CLASS_ATTR, MapOutputBuffer.class);
//获取MapOutputCollector的自雷
Class<? extends MapOutputCollector> subclazz = clazz.asSubclass(MapOutputCollector.class);
//通过反射创建一个收集器--org.apache.hadoop.mapred.MapTask.MapOutputBuffer
MapOutputCollector<KEY, VALUE> collector = ReflectionUtils.newInstance(subclazz, job);
//执行初始化操作
collector.init(context);
//最终将创建的写出器返回
return collector;
```

- org.apache.hadoop.mapred.MapTask.MapOutputBuffer#init

```
//获取溢写的阈值
final float spillper = job.getFloat(JobContext.MAP_SORT_SPILL_PERCENT, (float)0.8);
//缓冲区数据的大小100M
final int sortmb = job.getInt(JobContext.IO_SORT_MB, 100);
//数据的大小 1024*1024
indexCacheMemoryLimit = job.getInt(JobContext.INDEX_CACHE_MEMORY_LIMIT,
    INDEX_CACHE_MEMORY_LIMIT_DEFAULT)
//获取排序器--快速排序
sorter = ReflectionUtils.newInstance(job.getClass("map.sort.class",
    QuickSort.class, IndexedSorter.class), job);
//设置容量 100M
int maxMemUsage = sortmb << 20;
//结果肯定是16的整数倍
maxMemUsage -= maxMemUsage % METASIZE;
//缓冲区
kvbuffer = new byte[maxMemUsage];
//kvbuffer开始进行初始化
bufvoid = kvbuffer.length;
kvmeta = ByteBuffer.wrap(kvbuffer).order(ByteOrder.nativeOrder()).asIntBuffer();
setEquator(0);
bufstart = bufend = bufindex = equator;
kvstart = kvend = kvindex;

maxRec = kvmeta.capacity() / NMETA;
softLimit = (int)(kvbuffer.length * spillper);
bufferRemaining = softLimit;

//获取比较器
comparator = job.getOutputKeyComparator();
//获取Map的key和value输出类型
keyClass = (Class<K>)job.getMapOutputKeyClass();
valClass = (Class<V>)job.getMapOutputValueClass();
//序列化Key和Value
keySerializer = serializationFactory.getSerializer(keyClass);
```

```

keySerializer.open(bb);
valSerializer = serializationFactory.getSerializer(valClass);
valSerializer.open(bb);
//创建溢写线程，并让溢写线程处于等待，当达到阈值的时候开始溢写
spillInProgress = false;
minSpillsForCombine = job.getInt(JobContext.MAP_COMBINE_MIN_SPILLS, 3);
spillThread.setDaemon(true);
spillThread.setName("SpillThread");
spillLock.lock();
try {
    spillThread.start();
    while (!spillThreadRunning) {
        spillDone.await();
    }
}

```

- org.apache.hadoop.mapred.JobConf#getOutputKeyComparator

```

//获取比较器
Class<? extends RawComparator> ts = getClass(JobContext.KEY_COMPARATOR, null, RawComparator.class);
//如果自定义了比较器，创建自定义比较器对象
if (ts != null)
    return ReflectionUtils.newInstance(ts, this);
//如果没有创建比较器
return WritableComparator.get(getMapOutputKeyClass().asSubclass(WritableComparable.class), this);
//默认的比较器对象--org.apache.hadoop.io.WritableComparator
comparator = new WritableComparator(c, conf, true);

```

- org.apache.hadoop.mapreduce.task.JobContextImpl#getPartitionerClass

```

//创建分区器
return (Class<? extends Partitioner<, ?>)
    conf.getClass(PARTITIONER_CLASS_ATTR, HashPartitioner.class);
//分区器具体执行的代码
//org.apache.hadoop.mapreduce.lib.partition.HashPartitioner#getPartition
return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;

```

11.4. Spill

- org.apache.hadoop.mapred.MapTask.NewOutputCollector#write

```

//开始收集数据
collector.collect(key, value, partitioner.getPartition(key, value, partitions));

```

- org.apache.hadoop.mapred.MapTask.MapOutputBuffer#collect

```

//元数据存储区
bufferRemaining -= METASIZE;
//判断是否需要进行溢写，如果需要进行准备工作
//如果需要溢写唤醒SpillThread线程，调用run方法，开始SortAndSpill
org.apache.hadoop.mapred.MapTask.MapOutputBuffer#sortAndSpill
//如果不满足将数据存储到KvBuffer

```

11.5. Merge

- org.apache.hadoop.mapred.MapTask.MapOutputBuffer#flush

```

//将缓冲区中不满80%的数据也写出到硬盘
sortAndSpill();
//合并曾经溢写出的数据块
mergeParts();
//当前Map准备好进入到下一个阶段
sortPhase.startNextPhase();

```

11.6. ReduceTask

- org.apache.hadoop.mapred.ReduceTask#run

```

//进行初始化操作
initialize(job, getJobID(), reporter, useNewApi);
//获取Key和Value的迭代器

```

```

RawKeyValueIterator rIter = null;

//创建一个
Class combinerClass = conf.getCombinerClass();
CombineOutputCollector combineCollector =
    (null != combinerClass) ?
        new CombineOutputCollector(reduceCombineOutputCounter, reporter, conf) : null;
//创建一个Shuffer
Class<? extends ShuffleConsumerPlugin> clazz =
    job.getClass(MRConfig.SHUFFLE_CONSUMER_PLUGIN, Shuffle.class, ShuffleConsumerPlugin.class);
shuffleConsumerPlugin = ReflectionUtils.newInstance(clazz, job);

//创建一个上下文对象，并且对Shuffer进行初始化
ShuffleConsumerPlugin.Context shuffleContext =
    new ShuffleConsumerPlugin.Context(getTaskID(), job, FileSystem.getLocal(job), umbilical,
        super.lDirAlloc, reporter, codec,
        combinerClass, combineCollector,
        spilledRecordsCounter, reduceCombineInputCounter,
        shuffledMapsCounter,
        reduceShuffleBytes, failedShuffleCounter,
        mergedMapOutputsCounter,
        taskStatus, copyPhase, sortPhase, this,
        mapOutputFile, localMapFiles);

//已经初始化了合并器
shuffleConsumerPlugin.init(shuffleContext);
//执行Shuffer，并且返回key value的迭代器---MergeQueue
rIter = shuffleConsumerPlugin.run();

//获取Key的输出类型
Class keyClass = job.getMapOutputKeyClass();
Class valueClass = job.getMapOutputValueClass();
//获取分组比较器（reduce阶段优先使用分组比较器，如果没有设置就使用原来的比较器）
RawComparator comparator = job.getOutputValueGroupingComparator();
//开始执行Reduce任务
runNewReducer(job, umbilical, reporter, rIter, comparator, keyClass, valueClass);

```

- org.apache.hadoop.mapred.Task#initialize

```

//获取Job和Reduce的Context
jobContext = new JobContextImpl(job, id, reporter);
taskContext = new TaskAttemptContextImpl(job, taskId, reporter);
//返回数据的写出对象---org.apache.hadoop.mapreduce.lib.output.TextOutputFormat
outputFormat = ReflectionUtils.newInstance(taskContext.getOutputFormatClass(), job);
//创建提交对象---org.apache.hadoop.mapreduce.lib.output.FileOutputCommitter
committer = outputFormat.getOutputCommitter(taskContext);
//获取数据输出的路径
Path outputPath = FileOutputFormat.getOutputPath(conf);

```

- org.apache.hadoop.mapreduce.task.reduce.Shuffle#run

```

// Start the map-completion events fetcher thread
final EventFetcher<K, V> eventFetcher =
    new EventFetcher<K, V>(reduceId, umbilical, scheduler, this, maxEventsToFetch);
eventFetcher.start();
//判断map和reduce是否在一个节点
boolean isLocal = localMapFiles != null;
//开启拉取的线程数，本地为1，其他节点为5
final int numFetchers = isLocal ? 1 : jobConf.getInt(MRJobConfig.SHUFFLE_PARALLEL_COPIES, 5);
Fetcher<K, V>[] fetchers = new Fetcher[numFetchers];
//开始去拉取数据
fetchers[0].start();
//关闭拉取事件
eventFetcher.shutDown();
// Stop the map-output fetcher threads
for (Fetcher<K, V> fetcher : fetchers) {
    fetcher.shutDown();
}
//开始获取KeyValue的迭代器
RawKeyValueIterator kvIter = merger.close();

return kvIter;

```

- org.apache.hadoop.mapreduce.task.reduce.MergeManagerImpl#close

```

//返回最后一次合并的数据的迭代器
return finalMerge(jobConf, rfs, memory, disk);

```

- org.apache.hadoop.mapreduce.task.reduce.MergeManagerImpl#finalMerge

```
//获取map输出数据的类型
Class<K> keyClass = (Class<K>) job.getMapOutputKeyClass();
Class<V> valueClass = (Class<V>) job.getMapOutputValueClass();
//获取一个比较器 ---org.apache.hadoop.io.WritableComparator
final RawComparator<K> comparator = (RawComparator<K>) job.getOutputKeyComparator();
//返回key迭代器---org.apache.hadoop.mapred.Merger.MergeQueue
final RawKeyValueIterator rIter = Merger.merge(job, fs,
    keyClass, valueClass, memDiskSegments, numMemDiskSegments,
    tmpDir, comparator, reporter, spilledRecordsCounter, null,
    mergePhase);
```

- org.apache.hadoop.mapred.ReduceTask#runNewReducer

```
//获取迭代器
final RawKeyValueIterator rawIter = rIter;
//使用匿名内部类创建一个新的对象
rIter = new RawKeyValueIterator() {
    public void close() throws IOException {
        rawIter.close();
    }

    public DataInputBuffer getKey() throws IOException {
        return rawIter.getKey();
    }

    public Progress getProgress() {
        return rawIter.getProgress();
    }

    public DataInputBuffer getValue() throws IOException {
        return rawIter.getValue();
    }

    public boolean next() throws IOException {
        boolean ret = rawIter.next();
        reporter.setProgress(rawIter.getProgress().getProgress());
        return ret;
    }
};

//本次任务的上下文对象
org.apache.hadoop.mapreduce.TaskAttemptContext taskContext =
    new org.apache.hadoop.mapreduce.task.TaskAttemptContextImpl(job, getTaskID(), reporter);
//本次要执行的Reducer --com.yjx.WordCountReducer
org.apache.hadoop.mapreduce.Reducer<INKEY, INVALUE, OUTKEY, OUTVALUE> reducer =
    (org.apache.hadoop.mapreduce.Reducer<INKEY, INVALUE, OUTKEY, OUTVALUE>)
        ReflectionUtils.newInstance(taskContext.getReducerClass(), job);
//数据的写出器---org.apache.hadoop.mapreduce.lib.output.TextOutputFormat.LineRecordWriter
org.apache.hadoop.mapreduce.RecordWriter<OUTKEY, OUTVALUE> trackedRW =
    new NewTrackingRecordWriter<OUTKEY, OUTVALUE>(this, taskContext);
//创建Reduce的上下文对象
org.apache.hadoop.mapreduce.Reducer.Context
    reducerContext = createReduceContext(reducer, job, getTaskID(),
        rIter, reduceInputKeyCounter,
        reduceInputValueCounter,
        trackedRW,
        committer,
        reporter, comparator, keyClass,
        valueClass);

//开始执行reduce任务
reducer.run(reducerContext);
```

- org.apache.hadoop.mapreduce.Reducer#run

```
//判断是否还有数据可以读取，相同的key只会执行一次 (hello hello hello hi hi hi 2次)
while (context.nextKey()) {
    //context.getValues()-->private ValueIterable iterable = new ValueIterable();
    //values.iterator -->private ValueIterator iterator = new ValueIterator();
    //iterator.hasNext -->return firstValue || nextKeyIsSame;
    //iterator.next-->(firstValue?value:nextKeyValue())
    //
    reduce(context.getCurrentKey(), context.getValues(), context);
}
```

- org.apache.hadoop.mapreduce.task.ReduceContextImpl#nextKey

```
//hashMore 判断是否还有数据可以读取
//是否还有数据
if (hasMore) {
    //开始读取下一行
    return nextKeyValue();
} else {
    //所有数据处理完成, reduce结束
    return false;
}
```

- org.apache.hadoop.mapreduce.task.ReduceContextImpl#nextKeyValue

```
//判断是否为key的第一个值
firstValue = !nextKeyIsSame;
//获取key
key = keyDeserializer.deserialize(key);
value = valueDeserializer.deserialize(value);
//获取序列化时key和value的长度
currentKeyLength = nextKey.getLength() - nextKey.getPosition();
currentValueLength = nextVal.getLength() - nextVal.getPosition();
//将数据写入到备份存储
if (isMarked) {
    backupStore.write(nextKey, nextVal);
}
//判断下一次是否可以继续读取数据
hasMore = input.next();
//如果后面还有数据, 我要判断nextKeyIsSame
if (hasMore) {
    //获取下个key
    nextKey = input.getKey();
    //首先是组比较器, 否则就是默认的比较器
    nextKeyIsSame = comparator.compare(currentRawKey.getBytes(), 0,
                                         currentRawKey.getLength(),
                                         nextKey.getData(),
                                         nextKey.getPosition(),
                                         nextKey.getLength() - nextKey.getPosition()
                                         ) == 0;
} else {
    //如果读取不到数据, 也就没有下一个了
    nextKeyIsSame = false;
}
```

- org.apache.hadoop.mapreduce.lib.output.TextOutputFormat.LineRecordWriter#write

```
//以行的方式将数据写出
out.write(newline);
```

12. MapReduce优化

12.1. 概述

优化前我们需要知道 Hadoop 适合干什么活，适合什么场景，在工作中，我们要知道业务是怎样的，才能结合平台资源达到最优优化。除了这些我们还要知道 MapReduce 的执行流程，比如从文件的读取，Map 处理，Shuffle 过程，Reduce 处理，文件的输出或者存储压缩等等。

在工作中，往往平台的参数都是固定的，不可能为了某一个作业去修改整个平台的参数，所以在作业的执行过程中，需要对作业进行单独的设定，这样既不会对其他作业产生影响，也能很好的提高作业的性能，提高优化的灵活性。

接下来，回顾一下 Hadoop 的优势（适用场景）：

- 可构建在廉价机器上，设备成本相对较低；
- 高容错性，HDFS 将数据自动保存为多个副本，副本丢失后，自动恢复，防止数据丢失或损坏；
- 适合批处理，HDFS 适合一次写入、多次查询（读取）的情况，适合在已有数据的情况下进行多次分析，稳定性好；
- 适合存储大文件，其中的大可以表示存储单个大文件，因为是分块存储的。也可以表示存储大量的数据，但不适合小文件。

12.2. 小文件优化

从概述中我们知道，很明显 Hadoop 适合大文件的处理和存储，那为什么不适合小文件呢？

- 从存储方面来说：Hadoop 存储的每个文件都会在 NameNode 上记录元数据，如果同样大小的文件，文件很小的话，就会产生很多元数据文件，造成 NameNode 的压力；
- 从读取方面来说：同样大小的文件分为很多小文件的话，会增加磁盘寻址次数，降低性能；
- 从计算方面来说：我们知道一个 MapTask 默认处理一个分片或者一个文件，如果 MapTask 的启动时间比数据处理的时间还要长，那么就会造成低性能。而且在 Map 端溢写磁盘的时候每一个 MapTask 最终会产生 Reduce 数量个数的中间结果，如果 MapTask 数量特别多，就会造成临时文件很多，造成 Reduce 拉取数据的时候增加磁盘的 IO。

明白小文件造成的弊端之后，那我们应该怎么处理这些小文件呢？

- 从源头解决问题，也就是在 HDFS 上不要存储小文件，在数据上传至 HDFS 的时候提前合并小文件；
- 如果小文件何必后的文件过大，可以更换文件存储格式或压缩存储，当然压缩存储需要考虑是否能切片的问题；
- 如果小文件已经存储至 HDFS 了，那么在 FileInputFormat 读取数据的时候使用实现类 CombineFileInputFormat 读取数据，在读取数据的时候进行合并。

12.3. 数据倾斜

MapReduce 是一个并行处理框架（分布式），那么处理的时间肯定是作业中所有任务最慢的那个了，可谓木桶效应。为什么会这样呢？

- 数据倾斜，每个 Reduce 处理的数据量大小不一致，导致有些已经跑完了，有些还在执行；
- 还有可能就是某些作业所在的 NodeManager 有问题或者 Container 有问题或者 JVM GC 等，导致作业执行缓慢。

那么为什么会产生数据倾斜呢？比如数据本身就不平衡，所以在默认的 HashPartition 时造成分区数据不一致问题，还有就是代码设计不合理等。

那如何解决数据倾斜的问题呢？

- 不使用默认的 Hash 分区算法，采用自定义分区，结合业务特点，使得每个分区数据基本平衡；
- 或者既然有默认的分区算法，那么我们可以修改分区的键，让其符合 Hash 分区，并且使得最后的分区平衡，比如在 Key 前加随机数或盐 n-key；
- 既然 Reduce 处理慢，那么可以增加 Reduce 的 memory 和 vcore，提高性能解决问题，虽然没从根本上解决问题，但是还有效果的；
- 如果是因为只有一个 Reduce 导致作业很慢，可以增加 Reduce 的数量来分摊压力，然后再来一个作业实现最终聚合。

12.4. 推测执行

如果不是数据倾斜带来的问题，而是节点服务有问题造成某些 Map 和 Reduce 执行缓慢呢？可以使用推测执行，你跑的慢，我们可以找个其他节点重启一样的任务进行竞争，谁快以谁为准。推测执行是空间换时间的一种优化思想，会带来集群资源的浪费，给集群增加压力，所以一般情况下集群的推测执行都是关闭的，可以根据实际情况选择是否开启。

推测执行相关参数如下：

```
# 是否启用 MapTask 推测执行，默认为 true
mapreduce.map.speculative=true
# 是否启用 ReduceTask 推测执行，默认为 true
mapreduce.reduce.speculative=true
# 推测任务占当前正在运行的任务数的比例，默认为 0.1
mapreduce.job.speculative.speculative-cap-running-tasks=0.1;
# 推测任务占全部要处理任务数的比例，默认为 0.01
mapreduce.job.speculative.speculative-cap-total-tasks=0.01
# 最少允许同时运行的推测任务数量，默认为 10
mapreduce.job.speculative.minimum-allowed-tasks=10;
# 本次推测没有任务下发，执行下一次推测任务的等待时间，默认为 1000 (ms)
mapreduce.job.speculative.retry-after-no-speculate=1000;
# 本次推测有任务下发，执行下一次推测任务的等待时间，默认为 15000 (ms)
mapreduce.job.speculative.retry-after-speculate=15000;
# 标准差，任务的平均进展率必须低于所有正在运行任务的平均值才会被认为是太慢的任务，默认为 1.0
mapreduce.job.speculative.slowtaskthreshold=1.0;
```

12.5. MapReduce 执行流程优化

12.5.1. Map

12.5.1.1. 临时文件

上面我们从 Hadoop 的某些特定场景下聊了 MapReduce 的优化，接下来我们从 MapReduce 的执行流程进行优化。

前面我们已经聊过小文件在数据读取这里也可以做优化，所以选择一个合适的数据文件的读取类（FileInputFormat 的实现类）也很重要。我们在作业提交的过程中，会把作业 Jar 文件，配置文件，计算所得输入分片，资源信息等提交到 HDFS 的临时目录(Job ID 命名的目录下)，默认 10 个副本，可以通过 `mapreduce.client.submit.file.replication` 参数修改副本数量。后期作业执行时会下载这些文件到本地，中间会产生磁盘 IO。如果集群很大的时候，可以增加该参数的值，这样集群很多副本都可以供 NM 访问，从而提高下载的效率。

12.5.1.2. 分片

回顾一下源码中分片的计算公式：

```
// getFormatMinSplitSize(): 一个切片最少应该拥有 1 个字节
// getMinSplitSize(job): 读取程序员设置的切片的最小值，如果没有设置默认读取 1
long minSize = Math.max(getFormatMinSplitSize(), getMinSplitSize(job));
// 读取程序员设置的切片的最大值，如果没有设置默认读取 Long.MAX_VALUE
long maxSize = getMaxSplitSize(job);

// 获取 Block 的大小（默认为 128M）
long blockSize = file.getBlockSize();
// 获取 Split 的大小，切片的默认大小为 Block 的大小
// return Math.max(minSize, Math.min(maxSize, blockSize));
// minSize 为 64M --> 最终返回 128M, minSize 为 256M --> 最终返回 256M
// maxSize 为 64M --> 最终返回 64M, maxSize 为 256M --> 最终返回 128M
// 如果需要调大切片，则调节 minSize；如果需要调小切片，则调节 maxSize
long splitSize = computeSplitSize(blockSize, minSize, maxSize);
```

因为 Map 数没有具体的参数指定（默认情况下一个切片一个 MapTask），所以可以通过如上的公式调整切片的大小，这样就可以实现动态设置 Map 数了，那么问题来了，Map 数该如何设置呢？

12.5.1.3. 资源

这些东西一定要结合业务，Map 数太多，会产生很多中间结果，导致 Reduce 拉取数据变慢；Map 数太少，每个 Map 处理的时间又很长。那如果数据量就是很大，并且还需要控制 Map 的数量，这个时候每个 Map 的执行时间就比较长了，这时候可以调整每个 Map 的资源来提升 Map 的处理能力，相关参数如下。

```
# MapTask 的执行内存，默认为 1024MB
mapreduce.map.memory.mb=2048
# MapTask 的虚拟核数，默认为 1C
mapreduce.map.cpu.vcores=1
```

这里需要注意的是，单个 Map/Reduce Task 申请的内存大小，其值应该在 ResourceManager 中的最大和最小 Container 值之间，具体如下。

```
# ResourceManager 中每个容器可以申请内存资源的最小值，默认值为 1024MB
yarn.scheduler.minimum-allocation-mb=1024
# ResourceManager 中每个容器可以申请内存资源的最大值，默认值为 8192MB
yarn.scheduler.maximum-allocation-mb=8192
yarn.scheduler.minimum-allocation-vcores=1
yarn.scheduler.maximum-allocation-vcores=32
# NodeManager 节点最大可用内存，结合实际物理内存调整，默认值为 -1。表示该节点上 YARN 可使用的物理内存总量。
# 如果设置为 -1 且 yarn.nodemanager.resource.detect-hardware-capabilities 为 true（默认为 false），则会自动计算（在Windows和Linux环境下）。在其他情况下，默认为 8192MB
yarn.nodemanager.resource.memory-mb=-1
```

12.5.1.4. 环形缓冲区 & 溢写

从源头上确定好 Map 之后，接下来看看 Map 的具体执行过程。首先写环形数据缓冲区，为啥要写环形数据缓冲区呢，为什么不直接写磁盘？这样的目的主要是为了减少磁盘 IO。

每个 Map 任务不断地将键值对输出到内存中构造的一个环形数据结构中。使用环形数据结构是为了更有效地使用内存空间，在内存中放置尽可能多的数据。该缓冲默认为 100M（`mapreduce.task.io.sort.mb` 参数控制），当达到 80%（`mapreduce.map.sort.spill.percent` 参数控制）时就会溢写至磁盘，每达到 80% 都会重写溢写到一个新的文件。

可以根据机器的配置和数据量来设置这两个参数，当内存足够时，增大 `mapreduce.task.io.sort.mb=500` 会提高溢写的过程，而且会减少中间结果的文件数量。

```
mapreduce.task.io.sort.mb=500
mapreduce.map.sort.spill.percent=0.8
```

12.5.1.5. 合并

当文件溢写完后，Map 会对这些文件进行 Merge 合并，默认每次合并 10 个溢写的文件，由参数 `mapreduce.task.io.sort.factor=50` 进行设置。这样可以减少合并的次数，提高合并的并行度，降低对磁盘操作的次数。

```
mapreduce.task.io.sort.factor=50
```

12.5.1.6. 输出

12.5.1.6.1. 组合器

在 Reduce 拉取数据之前，我们可以使用 Combiner 实现 Map-Side 的预聚合（不影响最终结果的情况下），如果自定义了 Combiner，此时会根据 Combiner 定义的函数对 map 方法的结果进行合并，这样可以减少数据的传输，降低磁盘和网络 IO，提升性能。

12.5.1.6.2. 压缩

终于走到了 Map 到 Reduce 的数据传输过程了，这中间主要的影响无非就是磁盘 IO，网络 IO，数据量的大小了（是否压缩），其实减少数据量的大小，就可以做到优化了，所以我们可以选择性压缩数据，压缩后数据量会进一步减少，降低磁盘和网络 IO，提升性能。

开启压缩后，数据会被压缩写入磁盘，Reduce 读的是压缩数据所以需要解压，在实际经验中 Hive 在 Hadoop 的运行的瓶颈一般都是 IO 而不是 CPU，压缩一般可以 10 倍的减少 IO 操作。具体可以通过以下参数进行配置。

```
# Map 的输出在通过网络发送之前是否被压缩，默认为 false 不压缩
mapreduce.map.output.compress=false
# 如果 Map 的输出被压缩，那么应该如何压缩它们，默认为 org.apache.hadoop.io.compress.DefaultCodec
mapreduce.map.output.compress.codec=org.apache.hadoop.io.compress.SnappyCodec
```

12.5.1.6.3. 响应线程

Map 流程完成之后，会通过运行一个 HTTP Server 暴露自身，供 Reduce 端获取数据。这里用来响应 Reduce 数据请求的线程数量是可以配置的，通过 `mapreduce.shuffle.max.threads` 属性进行配置，默认为 0，表示当前机器内核数量的两倍。注意该配置是针对 NodeManager 配置的，而不是每个作业配置。具体如下。

```
mapreduce.shuffle.max.threads=0
```

12.5.1.7. 容错

Reduce 的每一个下载线程在下载某个 Map 数据的时候，有可能因为那个 Map 中间结果所在的机器发生错误，或者中间结果的文件丢失，或者网络中断等等情况，这样 Reduce 的下载就有可能失败，所以 Reduce 的下载线程并不会无休止的等待下去，当一定时间后下载仍然失败，那么下载线程就会放弃这次下载，并在随后尝试从其他的地方下载（因为这段时间 Map 可能会重跑）。

为什么从其他地方下载呢？因为 Map/Reduce Task 有容错机制，当任务执行失败后会尝试重启任务，相关参数如下。

```
# MapTask 最大重试次数，一旦重试次数超过该值，则认为 MapTask 运行失败，其对应的输入数据将不会产生任何结果，默认为 4
mapreduce.map.maxattempts=4
# ReduceTask最大重试次数，一旦重试次数超过该值，则认为ReduceTask运行失败，其对应的输入数据将不会产生任何结果，默认为4
mapreduce.reduce.maxattempts=4
# 当一个 NodeManager 上有超过 3 个任务失败时，ApplicationMaster 会将该节点上的任务调度到其他节点上执行
# 该值必须小于 Map/Reduce Task 最大重试次数，否则失败的任务将永远不在不同的节点上尝试
mapreduce.job.maxtaskfailures.per.tracker=3
# 当 NodeManager 发生故障，停止向 ResourceManager 节点发送心跳信息时，ResourceManager 节点并不会立即移除 NodeManager，而是要等待一段时间，该参数如下，默认为 600000ms
yarn.nm.liveness-monitor.expiry-interval-ms=600000
# 如果一个 Task 在一定时间内没有任务进度的更新（ApplicationMaster 一段时间没有收到任务进度的更新），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，也可能会永远卡住。为了防止 Task 永远 Block 不退出，则设置了一个超时时间（单位毫秒），默认为 600000ms，为 0 表示禁用超时
mapreduce.task.timeout=600000
# YARN 中的应用程序失败之后，最多尝试的次数，默认为 2，即当 ApplicationMaster 失败 2 次以后，运行的任务将会失败
mapreduce.am.max-attempts=2
# YARN 对 ApplicationMaster 的最大尝试次数做了限制，每个在 YARN 中运行的应用程序不能超过这个数量限制
yarn.resourcemanager.am.max-attempts=2
# Hadoop 对 ResourceManager 节点提供了检查点机制，当所有的 ResourceManager 节点失败后，重启 ResourceManager 节点，可以从上一个失败的 ResourceManager 节点保存的检查点进行状态恢复
# 检查点的存储由 yarn-site.xml 配置文件中的 yarn-resourcemanager.store.class 属性进行设置，默认是保存到文件中
yarn.resourcemanager.store.class=org.apache.hadoop.yarn.server.resourcemanager.recovery.FileSystemRMStateStore
```

12.5.2. Reduce

12.5.2.1. 资源

接下来就是 Reduce 了，首先可以通过参数设置合理的 Reduce 数量（`mapreduce.job.reduces` 参数控制），以及通过参数设置每个 Reduce 的资源。具体如下。

```
# 默认为 1
mapreduce.job.reduces=1
# 默认为 1024MB
mapreduce.reduce.memory.mb=4096
# 默认为 1
mapreduce.reduce.cpu.vcores=1
# Map 和 Reduce 共享，当 MapTask 完成的比例达到该值后会为 ReduceTask 申请资源，默认是 0.05
mapreduce.job.reduce.slowstart.completedmaps=0.05
```

12.5.2.2. 拉取

Reduce 在 Copy 的过程中默认使用 5 个（`mapreduce.reduce.shuffle.parallelcopies` 参数控制）并行度进行数据复制，可以将其调大例如 100。

Reduce 的每一个下载线程在下载某个 Map 数据的时候，有可能因为那个 Map 中间结果所在的机器发生错误，或者中间结果的文件丢失，或者网络中断等等情况，这样 Reduce 的下载就有可能失败，所以 Reduce 的下载线程并不会无休止的等待下去，当一定时间后下载仍然失败，那么下载线程就会放弃这次下载，并在随后尝试从其他的地方下载（因为这段时间 Map 可能会重跑）。Reduce 下载线程的最大下载时间段可以通过 `mapreduce.reduce.shuffle.read.timeout`（默认为 180000 秒）进行调整。

12.5.2.3. 缓冲区 & 溢写

Copy 过来的数据会先放入内存缓冲区中，然后当使用内存达到一定量的时候才 Spill 磁盘。这里的缓冲区大小要比 Map 端的更为灵活，它基于 JVM 的 Heap Size 进行设置。该内存大小不像 Map 一样可以通过 `mapreduce.task.io.sort.mb` 来设置，而是通过另外一个参数 `mapreduce.reduce.shuffle.input.buffer.percent`（默认为 0.7）进行设置。意思是说，Shuffle 在 Reduce 内存中的数据最多使用内存量为： $0.7 * \text{maxHeap of reduce task}$ ，内存到磁盘 Merge 的启动门限可以通过 `mapreduce.reduce.shuffle.merge.percent`（默认为 0.66）进行设置。

假设 `mapreduce.reduce.shuffle.input.buffer.percent` 为 0.7，`ReduceTask` 的 `max heapsize` 为 1G，那么用来做拉取数据缓存的内存大概为 700MB 左右。这 700MB 的内存跟 Map 端一样，也不是要等到全部写满才会往磁盘溢写，而是达到指定的阈值就会开始往磁盘溢写（溢写前会先做 sortMerge）。这个限度阈值可以通过参数 `mapreduce.reduce.shuffle.merge.percent` 来设定（默认为 0.66）。整个过程同 Map 类似，如果用户设置了 Combiner，也会被启用，然后磁盘中会生成众多的溢写文件。这种 Merge 方式一直在运行，直到没有 Map 端的数据时才会结束，然后启动磁盘到磁盘的 Merge 方式生成最终的文件。

12.5.2.4. 合并

同 Map 一样，当文件溢写完后，Reduce 会对这些文件进行 Merge 合并。合并因子默认为 10，由参数 `mapreduce.task.io.sort.factor` 进行设置。如果 Map 输出很多，则需要合并很多趟，所以可以减少合并的次数，提高合并的并行度，降低对磁盘操作的次数。

12.5.2.5. 读缓存

默认情况下，数据达到一个阈值的时候，缓冲区中的数据就会写入磁盘，然后 Reduce 会从磁盘中获得所有的数据。也就是说，缓冲区和 Reduce 是没有直接关联的，中间会有多次写磁盘 -> 读磁盘的过程，既然有这个弊端，那么可以通过修改参数，使得缓冲区中的一部分数据可以直接输送到 Reduce（缓冲区 -> 读缓存 -> Reduce），从而减少 IO 开销。

修改参数 `mapreduce.reduce.input.buffer.percent`，默认为 0.0，表示不开启缓存，直接从磁盘读。当该值大于 0 的时候，会保留指定比例的内存用于缓存（缓冲区 -> 读缓存 -> Reduce），从而提升计算的速度。这样一来，设置缓冲区需要内存，读取数据需要内存，Reduce 计算也需要内存，所以要根据作业的用运行情况进行调整。

当 Reduce 计算逻辑消耗内存很小时，可以分一部分内存用来缓存数据，可以提升计算的速度。默认情况下都是从磁盘读取数据，如果内存足够大的话，务必设置该参数让 Reduce 直接从缓存读数据。

