大数据离线阶段

04

一、 课程计划

目录

二、 初识 MapReduce

1. MapReduce 计算模型介绍

1.1. 理解 MapReduce 思想

MapReduce 的思想核心是“分而治之”。

所谓“分而治之”就是把一个复杂的问题按一定的“分解”方法分为规模较小的若干部分，然后逐个解决，分别找出各部分的解，再把把各部分的解组成整个问题的解。

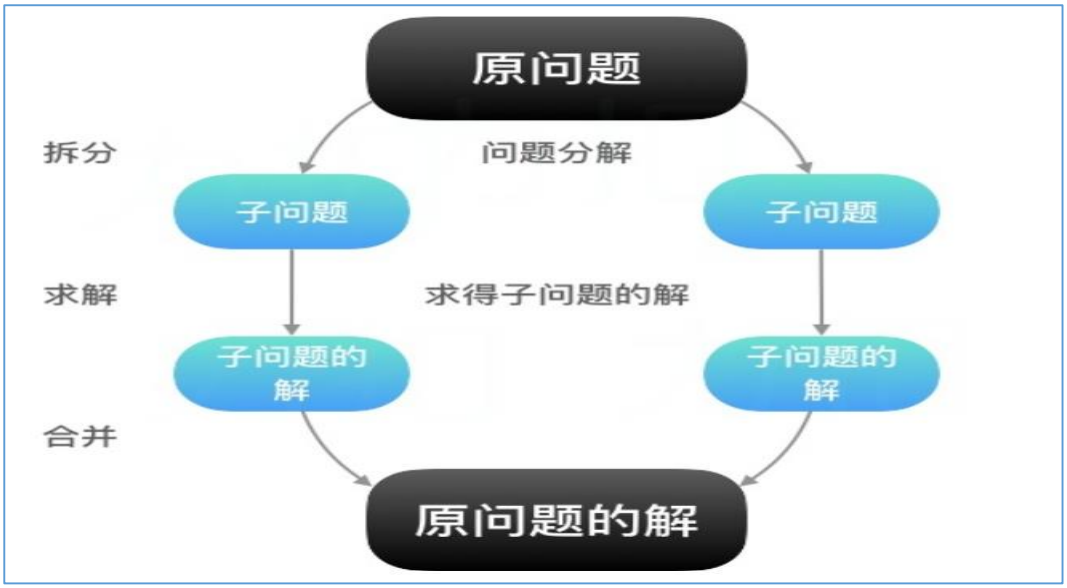
这种朴素的思想来源于人们生活与工作的经验，也完全适合于技术领域。诸如软件的体系结构设计、模块化设计都是分而治之的具体表现。即使是发布过论文实现分布式计算的谷歌也只是实现了这种思想，而不是自己原创。

概况起来，MapReduce 所包含的思想分为两步：

Map 负责“分”，即把复杂的任务分解为若干个“简单的任务”来并行处理。可以进行拆分的前提是这些小任务可以并行计算，彼此间几乎没有依赖关系。

Reduce 负责“合”，即对 map 阶段的结果进行全局汇总。

这两个阶段合起来正是 MapReduce 思想的体现。



还有一个比较形象的语言解释 MapReduce：要数停车场中的所有停放车的总数量。

你数第一列，我数第二列…这就是 Map 阶段，人越多，能够同时数车的人就越多，速度就越快。

数完之后，聚到一起把所有人的统计数加在一起。这就是 Reduce 合并汇总阶段。

1.2. Hadoop MapReduce 设计构思

MapReduce 是一个分布式运算程序的编程框架，核心功能是将用户编写的业务逻辑代码和自带默认组件整合成一个完整的分布式运算程序，并发运行在 Hadoop 集群上。

既然是做计算的框架，那么表现形式就是有个输入（input），MapReduce 操作这个输入（input），通过本身定义好的计算模型，得到一个输出（output）。

对许多开发者来说，自己完完全全实现一个并行计算程序难度太大，而 MapReduce 就是一种简化并行计算的编程模型，降低了开发并行应用的入门门槛。

Hadoop MapReduce 构思体现在如下的三个方面：

- 如何对付大数据处理：分而治之

对相互间不具有计算依赖关系的大数据，实现并行最自然的办法就是采取分而治之的策略。并行计算的第一个重要问题是如何划分计算任务或者计算数据以便对划分的子任务或数据块同时进行计算。不可分拆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！

- 构建抽象模型：Map 和 Reduce

MapReduce 借鉴了函数式语言中的思想，用 Map 和 Reduce 两个函数提供了高层的并行编程抽象模型。

Map: 对一组数据元素进行某种重复式的处理；

Reduce: 对 Map 的中间结果进行某种进一步的结果整理。

MapReduce 中定义了如下的 Map 和 Reduce 两个抽象的编程接口，由用户去编程实现:

map: (k1; v1) → [(k2; v2)]

reduce: (k2; [v2]) → [(k3; v3)]

Map 和 Reduce 为程序员提供了一个清晰的操作接口抽象描述。通过以上两个编程接口，大家可以看出 MapReduce 处理的数据类型是<key,value>键值对。

- 统一构架，隐藏系统层细节

如何提供统一的计算框架，如果没有统一封装底层细节，那么程序员则需要考虑诸如数据存储、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。

MapReduce 最大的亮点在于通过抽象模型和计算框架把需要做什么(what need to do)与具体怎么做(how to do)分开了，为程序员提供一个抽象和高层的编程接口和框架。程序员仅需要关心其应用层的具体计算问题，仅需编写少量的处理应用本身计算问题的程序代码。

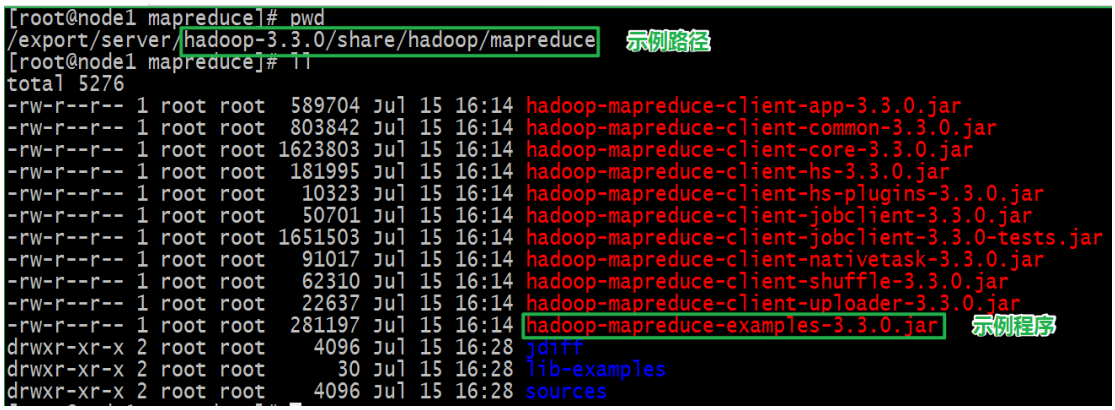
如何具体完成这个并行计算任务所相关的诸多系统层细节被隐藏起来,交给计算框架去处理：从分布代码的执行，到大到数千小到单个节点集群的自动调度使用。

2. 官方 MapReduce 示例

在 Hadoop 的安装包中，官方提供了 MapReduce 程序的示例 examples，以便快速上手体验 MapReduce。

该示例是使用 java 语言编写的，被打包成为了一个 jar 文件。

/export/server/hadoop-3.3.0/share/hadoop/mapreduce



运行该 jar 包程序，可以传入不同的参数实现不同的处理功能。

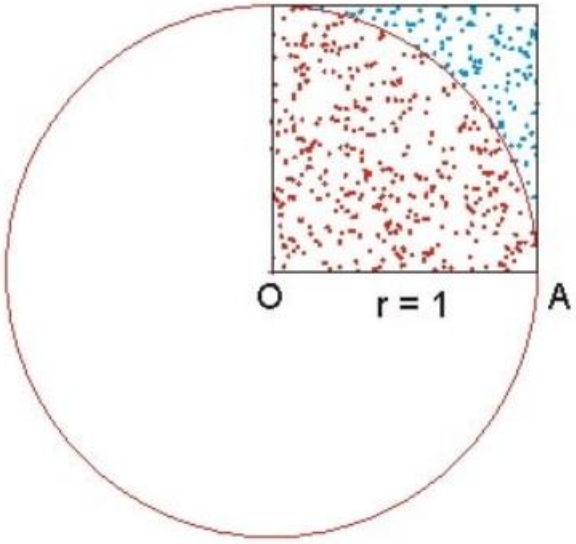
```
hadoop jar hadoop-mapreduce-examples-3.3.0.jar args...
```

2.1. 示例 1：评估圆周率 π （PI）

圆周率 π 大家都不陌生，如何去估算 π 的值呢？

Monte Carlo 方法到基本思想：

当所求解问题是某种随机事件出现的概率，或者是某个随机变量的期望值时，通过某种“实验”的方法，以这种事件出现的频率估计这一随机事件的概率，或者得到这个随机变量的某些数字特征，并将其作为问题的解。



假设正方形边长为 1，圆半径也为 1，那么 1/4 圆的面积为：

$$\frac{1}{4}\pi r^2 = \frac{\pi}{4}$$

在正方形内随机撒点，分布于 1/4 圆内的数量假设为 a ，分布于圆外的数量为 b，N 则是所产生的总数：N=a+b。

那么数量 a 与 N 的比值应与 1/4 圆面积及正方形面积成正比，于是：

$$\frac{\pi}{4} : 1 = a : N$$

$$\pi = \frac{4a}{N}$$

下面来运行 MapReduce 程序评估一下圆周率的值，执行中可以去 YARN 页面上观察程序的执行的情况。

```
[root@node1 mapreduce]# pwd
/export/server/hadoop-3.3.0/share/hadoop/mapreduce
[root@node1 mapreduce]# hadoop jar hadoop-mapreduce-examples-3.3.0.jar pi 10 50
```

第一个参数 pi：表示 MapReduce 程序执行圆周率计算；

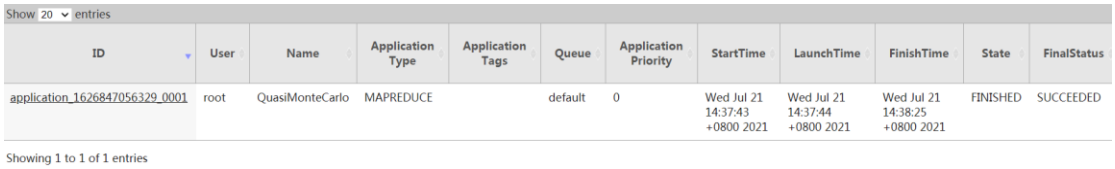
第二个参数：用于指定 map 阶段运行的任务次数，并发度，这是是 10；

第三个参数：用于指定每个 map 任务取样的个数，这里是 50。

```
[root@node1 mapreduce]# hadoop jar hadoop-mapreduce-examples-3.3.0.jar pi 10 50
Number of Maps  = 10
Samples per Map = 50
Wrote input for Map #0
Wrote input for Map #1
Wrote input for Map #2
Wrote input for Map #3
Wrote input for Map #4
Wrote input for Map #5
Wrote input for Map #6
Wrote input for Map #7
Wrote input for Map #8
Wrote input for Map #9
Starting Job
2021-07-21 14:37:41,596 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at node1/192.168.227.151:8032
2021-07-21 14:37:42,367 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/root/.staging/job_1626847056329_0001
```

```
2021-07-21 14:37:42,561 INFO input.FileInputFormat: Total input files to process : 10
2021-07-21 14:37:42,648 INFO mapreduce.JobSubmitter: number of splits:10
2021-07-21 14:37:42,875 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1626847056329_0001
2021-07-21 14:37:42,875 INFO mapreduce.JobSubmitter: Executing with tokens: []
2021-07-21 14:37:43,109 INFO conf.Configuration: resource-types.xml not found
2021-07-21 14:37:43,110 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2021-07-21 14:37:43,490 INFO impl.YarnClientImpl: Submitted application application_1626847056329_0001
2021-07-21 14:37:43,561 INFO mapreduce.Job: The url to track the job: http://node1:8088/proxy/application_1626847056329_0001/
2021-07-21 14:37:43,563 INFO mapreduce.Job: Running job: job_1626847056329_0001
2021-07-21 14:37:55,881 INFO mapreduce.Job: Job job_1626847056329_0001 running in uber mode : false
2021-07-21 14:37:55,883 INFO mapreduce.Job:  map 0% reduce 0%
2021-07-21 14:38:16,714 INFO mapreduce.Job:  map 20% reduce 0%
2021-07-21 14:38:17,760 INFO mapreduce.Job:  map 50% reduce 0%
2021-07-21 14:38:20,917 INFO mapreduce.Job:  map 70% reduce 0%
2021-07-21 14:38:21,924 INFO mapreduce.Job:  map 100% reduce 0%
2021-07-21 14:38:26,953 INFO mapreduce.Job:  map 100% reduce 100%
2021-07-21 14:38:26,986 INFO mapreduce.Job: Job job_1626847056329_0001 completed successfully
```

```
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=1180
        File Output Format Counters
                Bytes Written=97
Job Finished in 45.694 seconds
Estimated value of Pi is 3.16000000000000000000
```



Show 20 entries

ID	User	Name	Application Type	Application Tags	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus
application_1626847056329_0001	root	QuasiMonteCarlo	MAPREDUCE		default	0	Wed Jul 21 14:37:43 +0800 2021	Wed Jul 21 14:37:44 +0800 2021	Wed Jul 21 14:38:25 +0800 2021	FINISHED	SUCCEEDED

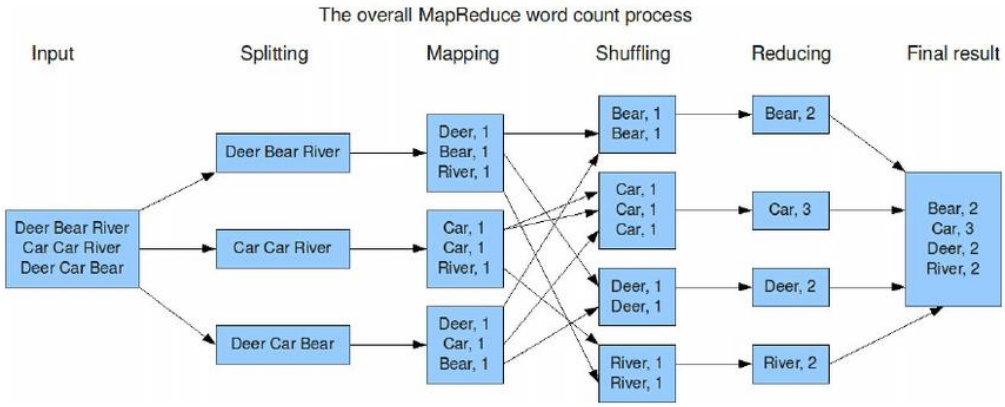
Showing 1 to 1 of 1 entries

2.2. 示例 2：单词词频统计 WordCount

WordCount 算是大数据统计分析领域的经典需求了，相当于编程语言的 HelloWorld。其背后的应用场景十分丰富，比如统计页面点击数，搜索词排行榜等跟 count 相关的需求。

其最基本的应用雏形就是统计文本数据中，相同单词出现的总次数。用 SQL 的角度来理解的话，相当于根据单词进行 group by 分组，相同的单词分为一组，然后每个组内进行 count 聚合统计。

对于 MapReduce 乃至于大数据计算引擎来说，业务需求本身是简单的，重点是当数据量大了之后，如何使用分而治之的思想来处理海量数据进行单词统计。



上传课程资料中的文本文件到 HDFS 文件系统的/input 目录下，如果没有这个目录，使用 shell 创建：

```
hadoop fs -mkdir /input
hadoop fs -put 1.txt /input
```

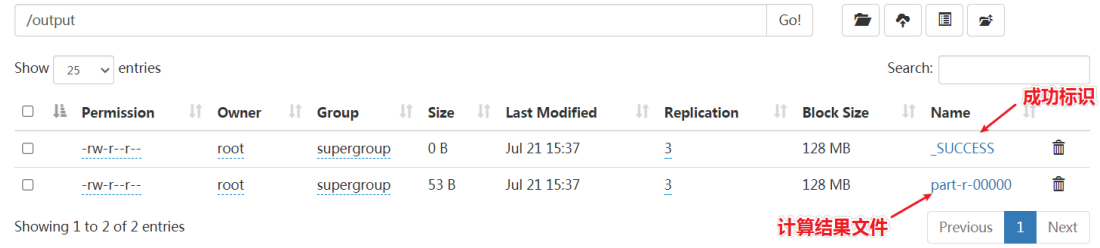
准备好之后，执行官方 MapReduce 实例，对上述文件进行单词次数统计:

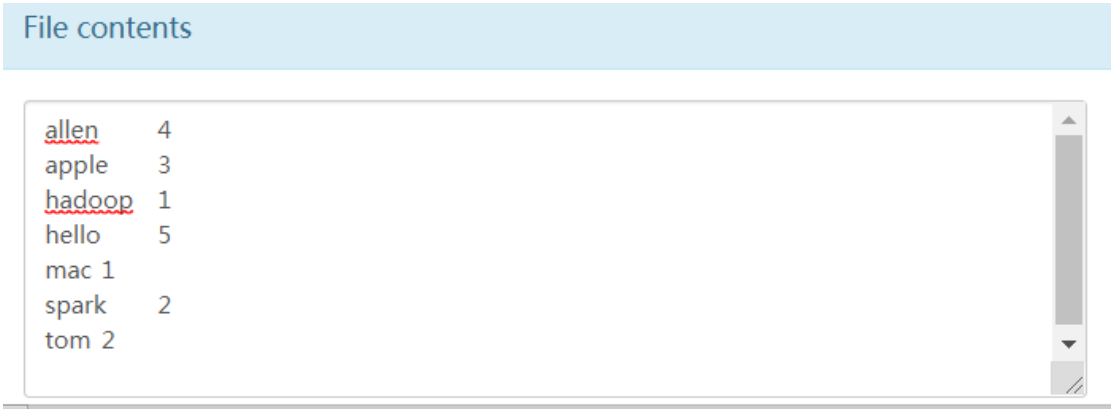
```
[root@node1 mapreduce]# pwd
/export/server/hadoop-3.3.0/share/hadoop/mapreduce
[root@node1 mapreduce]# hadoop jar hadoop-mapreduce-examples-3.3.0.jar wordcount
/input /output
```

第一个参数：wordcount 表示执行单词统计

第二个参数：指定输入文件的路径

第三个参数：指定输出结果的路径（该路径不能已存在）





可以在课程资料中查看 java 代码的具体实现，后续课程中也会学习如何使用 java 编写 MapReduce 程序。

3. MapReduce Python 接口接入

3.1. 前言

虽然 Hadoop 是用 Java 编写的一个框架，但是并不意味着他只能使用 Java 语言来操作，在 Hadoop-0.14.1 版本后，Hadoop 支持了 Python 和 C++语言，在 Hadoop 的文档中也表示可以使用 Python 进行开发。

https://hadoop.apache.org/docs/r3.3.0/hadoop-streaming/HadoopStreaming.html

在 Hadoop 的文档中提到了 Hadoop Streaming，我们可以使用流的方式来操作它.语法是：

```
[root@node1 lib]# pwd
/export/server/hadoop-3.3.0/share/hadoop/tools/lib
[root@node1 lib]# hadoop jar hadoop-streaming-3.3.0.jar
    -input InputDirs \
    -output OutputDir \
    -mapper xxx \
    -reducer xxx
```

在 Python 中的 sys 包中存在, stdin 和 stdout,输入输出流，我们可以利用这个方式来进行 MapReduce 的编写.

3.2. 代码实现

mapper.py

```
# -*- coding:utf-8 -*-
# @Time : 2021/6/21 16:04
# @Author: itcast
# @File : mapper.py
import sys

for line in sys.stdin:
    # 捕获输入流
    line = line.strip()
```

```
    # 根据分隔符切割单词
    words = line.split()
    # 遍历单词列表 每个标记 1
    for word in words:
        print("%s\t%s" % (word, 1))
```

reducer.py

```
# -*- coding:utf-8 -*-
# @Time : 2021/6/21 16:04
# @Author: itcast
# @File : reducer.py
import sys
# 保存单词次数的字典 key:单词 value：总次数
word_dict = {}

for line in sys.stdin:

    line = line.strip()
    word, count = line.split('\t')

    # count 类型转换
    try:
        count = int(count)
    except ValueError:
        continue
    # 如果单词位于字典中 +1，如果不存在 保存并设初始值 1
    if word in word_dict:
        word_dict[word] += 1
    else:
        word_dict.setdefault(word, 1)
# 结果遍历输出
for k, v in word_dict.items():
    print('%s\t%s' % (k, v))
```

3.3. 程序执行

方式 1：本地测试 Python 脚本逻辑是否正确。

方式 2：使用 hadoop streaming 提交 Python 脚本集群运行。

注意：不管哪种方式执行，都需要提前在 Centos 系统上安装好 Python3.详细安装步骤可以参考课程资料。

本地测试

```
#上传待处理文件 和 Python 脚本到 Linux 上
[root@node2 ~]# pwd
/root
[root@node2 ~]# ll
-rw-r--r--  1 root  root          105 May 18 15:12 1.txt
-rwxr--r--  1 root  root          340 Jul 21 16:16 mapper.py
-rwxr--r--  1 root  root          647 Jul 21 16:18 reducer.py

#使用 shell 管道符运行脚本测试
[root@node2 ~]# cat 1.txt | python mapper.py |sort|python reducer.py
allen    4
apple    3
hadoop   1
hello    5
mac      1
spark    2
tom      2
```

hadoop streaming 提交

```
#上传处理的文件到 hdfs、上传 Python 脚本到 linux

#提交程序执行
hadoop jar /export/server/hadoop-3.3.0/share/hadoop/tools/lib/hadoop-streaming-3.3.0.jar \
-D mapred.reduce.tasks=1 \
-mapper "python mapper.py" \
-reducer "python reducer.py" \
-file mapper.py -file reducer.py \
-input /input/* \
-output /outpy
```

执行结果：

```
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=158
        File Output Format Counters
                Bytes Written=53
2021-07-21 17:27:06,727 INFO streaming.StreamJob: Output directory: /outpy
```

/outpy Go!

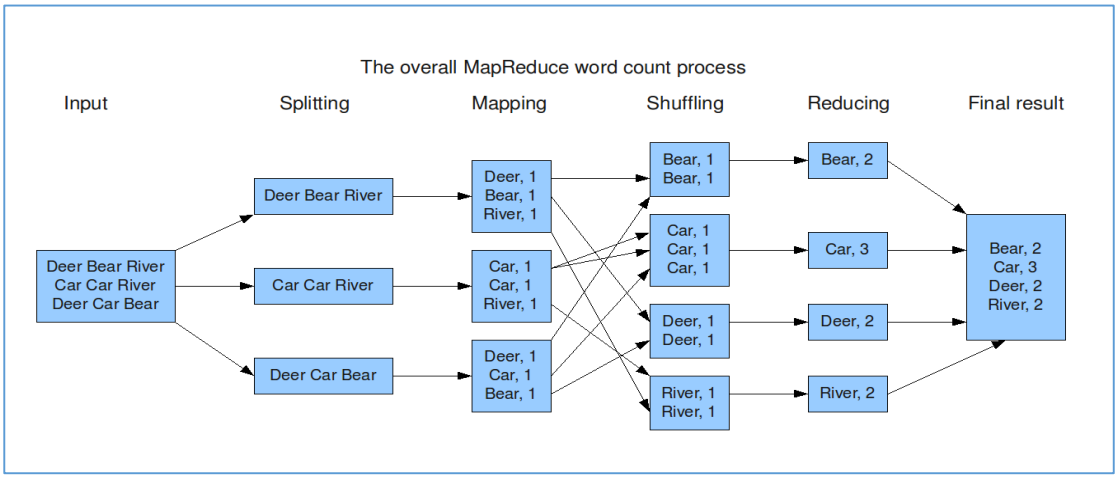
Show 25 entries Search:

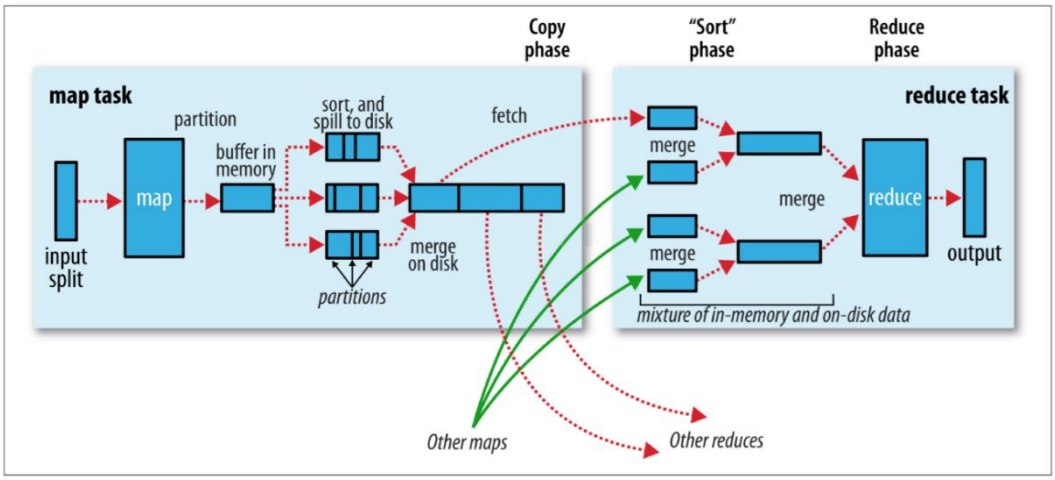
Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	root	supergroup	0 B	Jul 21 17:27	3	128 MB	_SUCCESS
-rw-r--r--	root	supergroup	53 B	Jul 21 17:27	3	128 MB	part-00000

Showing 1 to 2 of 2 entries Previous 1 Next

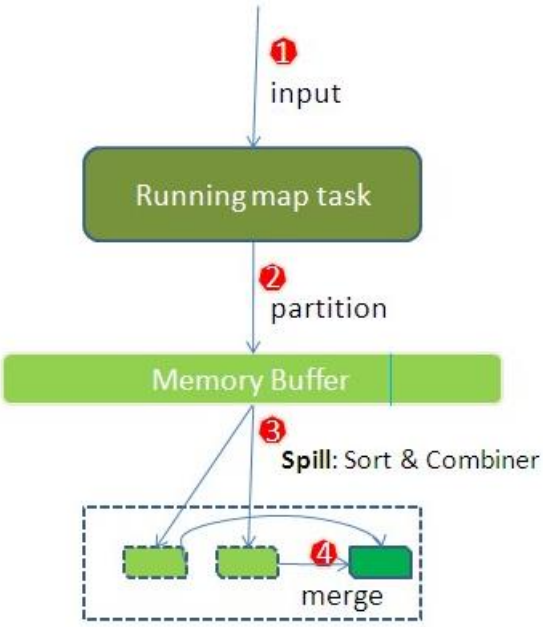
三、 MapReduce 基本原理

1. 整体执行流程图





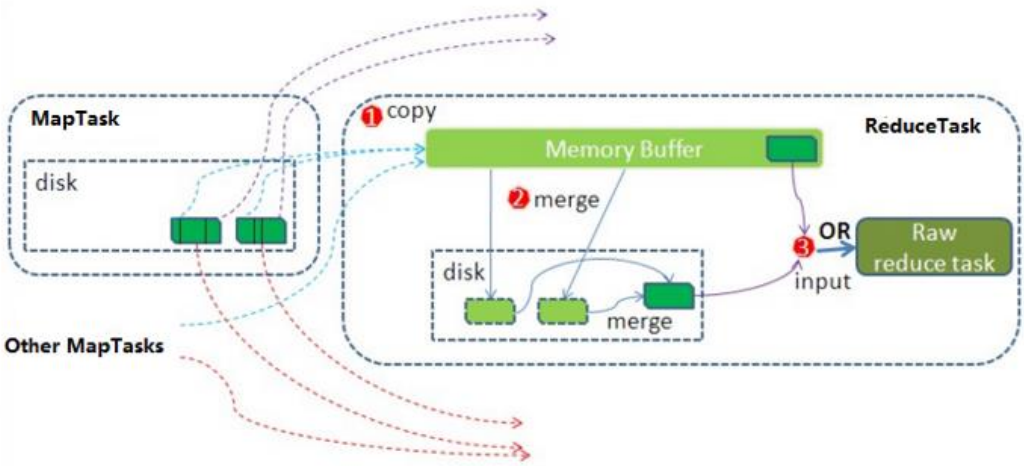
2. Map 阶段执行流程



- **第一阶段**是把输入目录下文件按照一定的标准逐个进行逻辑切片，形成切片规划。默认情况下，Split size = Block size。每一个切片由一个 MapTask 处理。（getSplits）
- **第二阶段**是对切片中的数据按照一定的规则解析成<key,value>对。默认规则是把每一行文本内容解析成键值对。key 是每一行的起始位置(单位是字节)，value 是本行的文本内容。（TextInputFormat）
- **第三阶段**是调用 Mapper 类中的 map 方法。上阶段中每解析出来的一个<k,v>，调用一次 map 方法。每次调用 map 方法会输出零个或多个键值对。
- **第四阶段**是按照一定的规则对第三阶段输出的键值对进行分区。默认是只有一个区。分区的数量就是 Reducer 任务运行的数量。默认只有一个 Reducer 任务。
- **第五阶段**是对每个分区中的键值对进行排序。首先，按照键进行排序，对于键相同的键值对，按照值进行排序。比如三个键值对<2,2>、<1,3>、<2,1>，键和值分别是整数。那么排序后的结果是<1,3>、<2,1>、<2,2>。如果有第六阶段，那么进入第六阶段；如果没有，直接输出到文件中。

- **第六阶段**是对数据进行局部聚合处理，也就是 combiner 处理。键相等的键值对会调用一次 reduce 方法。经过这一阶段，数据量会减少。**本阶段默认是没有的。**

3． Reduce 阶段执行流程

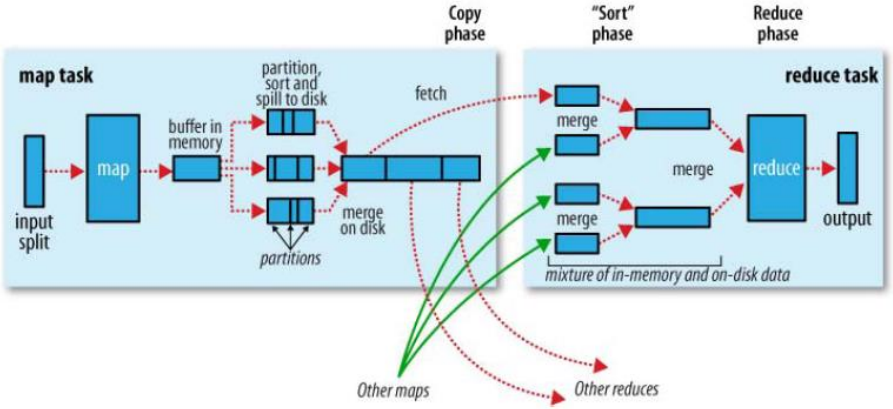


- **第一阶段**是 Reducer 任务会主动从 Mapper 任务复制其输出的键值对。Mapper 任务可能会有很多，因此 Reducer 会复制多个 Mapper 的输出。
- **第二阶段**是把复制到 Reducer 本地数据，全部进行合并，即把分散的数据合并成一个大的数据。再对合并后的数据排序。
- **第三阶段**是对排序后的键值对调用 reduce 方法。键相等的键值对调用一次 reduce 方法，每次调用会产生零个或者多个键值对。最后把这些输出的键值对写入到 HDFS 文件中。

4. 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。

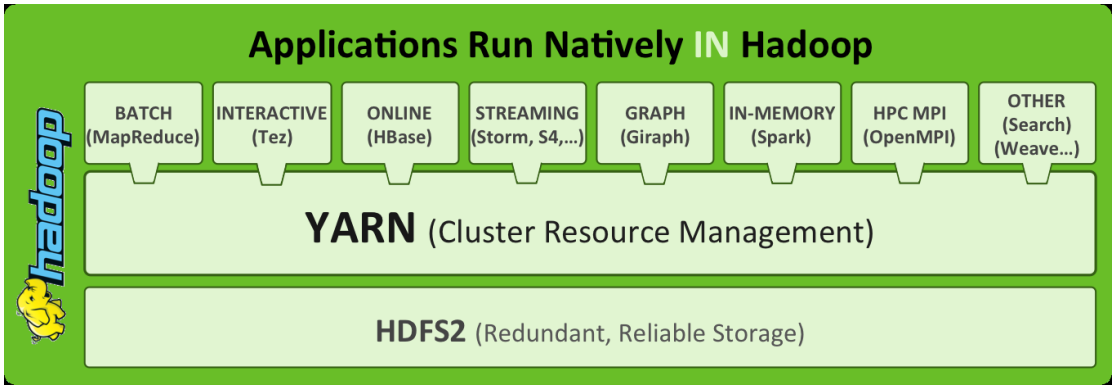
5).Merge 阶段：在 ReduceTask 远程复制数据的同时，会在后台开启两个线程对内存到本地的数据文件进行合并操作。

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

四、 Apache Hadoop YARN

1. Yarn 通俗介绍

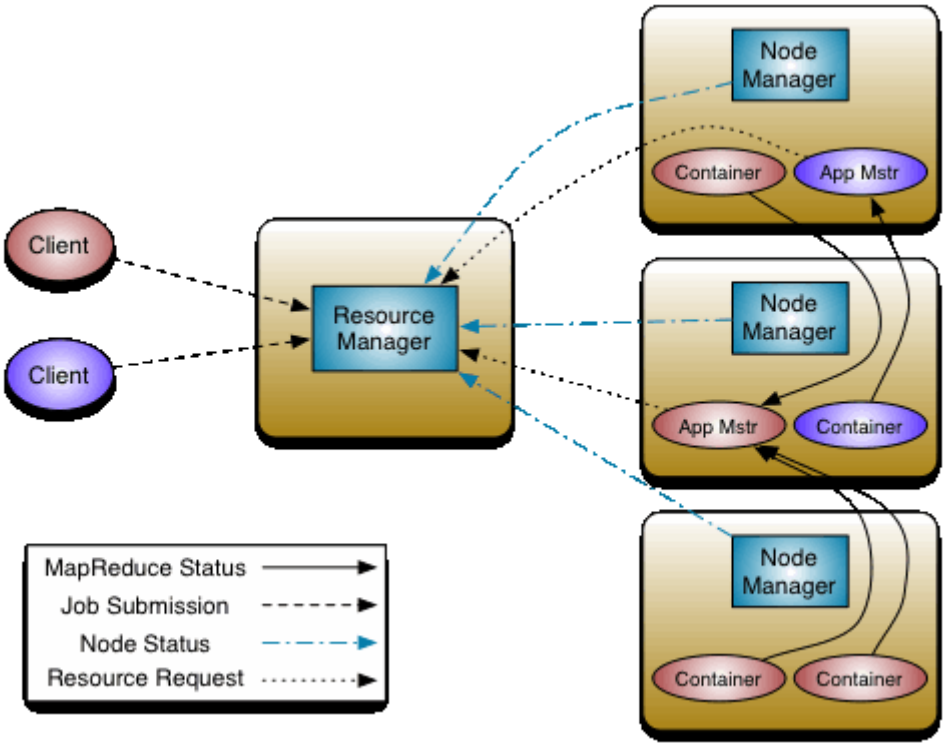


Apache Hadoop YARN （Yet Another Resource Negotiator，另一种资源协调者）是一种新的 Hadoop 资源管理器，它是一个通用资源管理系统和调度平台，可为上层应用提供统一的资源管理和调度，它的引入为集群在利用率、资源统一管理和数据共享等方面带来了巨大好处。

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。

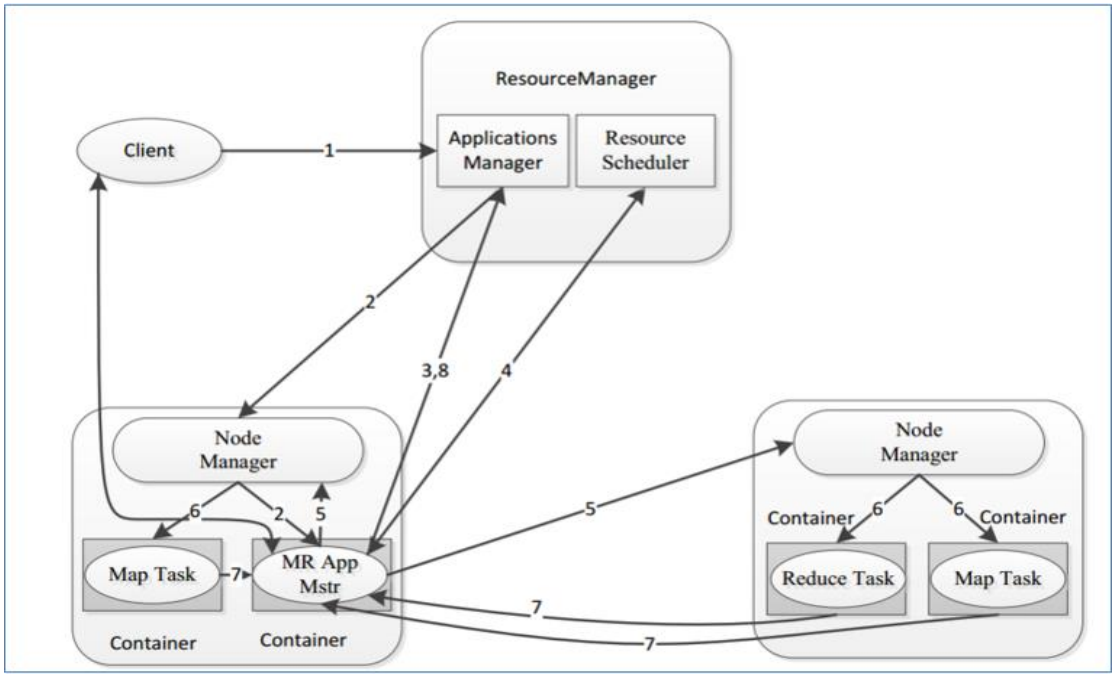
3.3. ApplicationMaster

- 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。



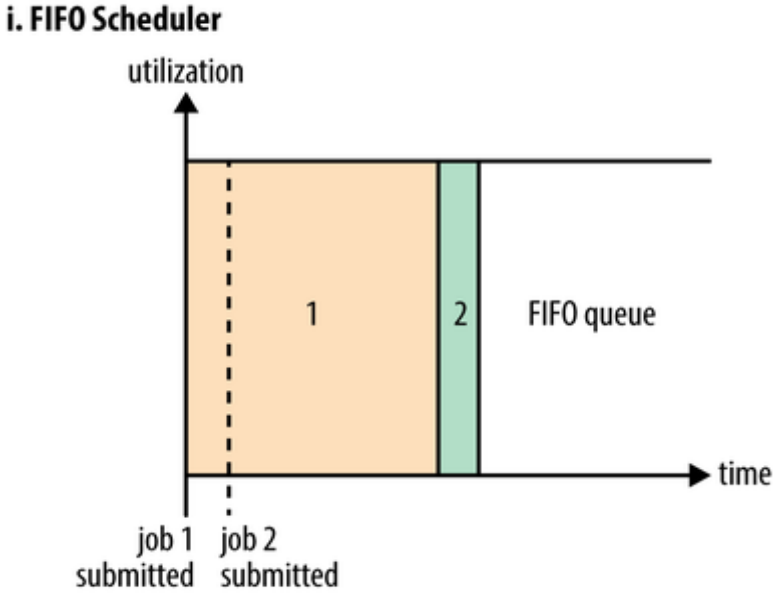
5. Yarn 调度器 Scheduler

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

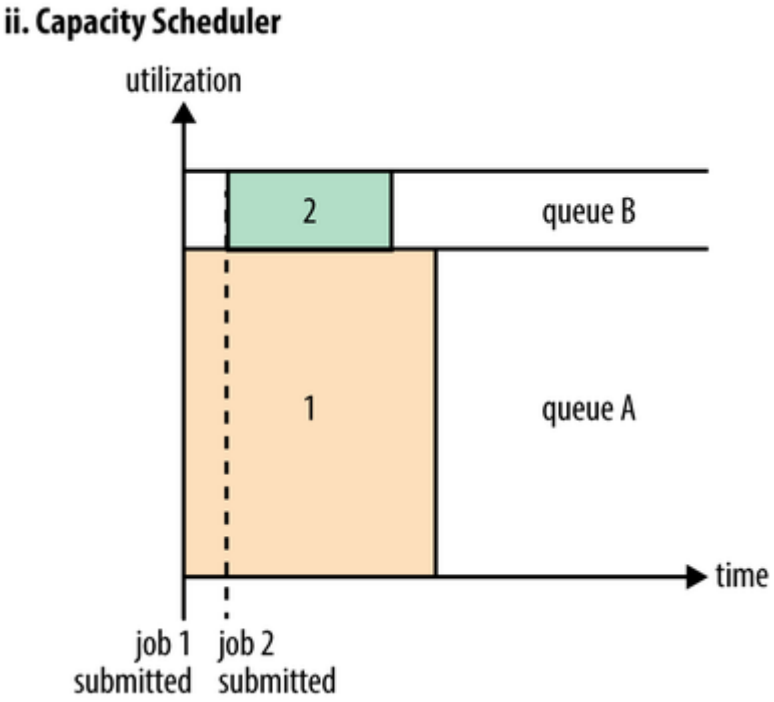
FIFO Scheduler 把应用按提交的顺序排成一个队列，这是一个先进先出队列，在进行资源分配的时候，先给队列中最头上的应用进行分配资源，待最头上的应用需求满足后再给下一个分配，以此类推。



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。



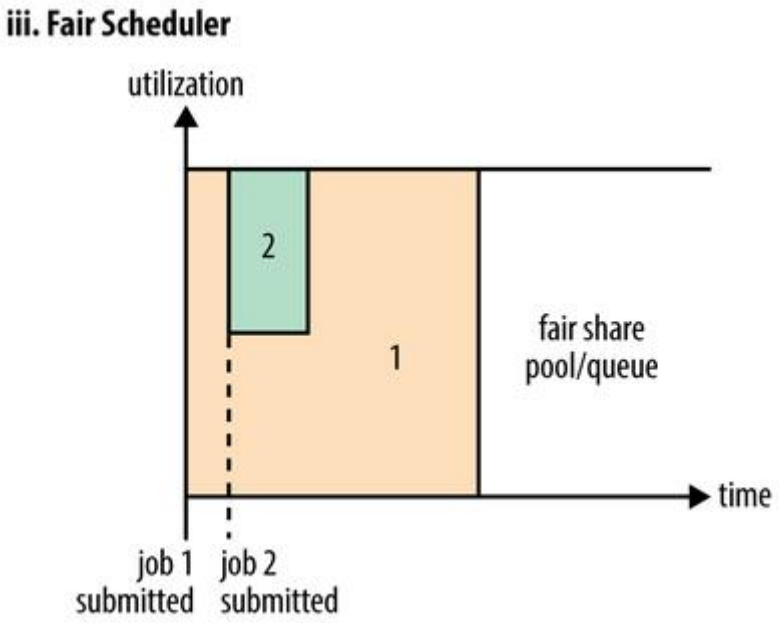
容量调度器 Capacity Scheduler 最初是由 Yahoo 最初开发设计使得 Hadoop 应用能够被多用户使用，且最大化整个集群资源的吞吐量，现被 IBM BigInsights 和 Hortonworks HDP 所采用。

Capacity Scheduler 被设计为允许应用程序在一个可预见的和简单的方式共享集群资源，即"作业队列"。Capacity Scheduler 是根据租户的需要和要求把现有的资源分配给运行的应用程序。Capacity Scheduler 同时允许应用程序访问还没有被使用的资源，以确保队列之间共享其它队列被允许的使用资源。管理员可以控制每个队列的容量，Capacity Scheduler 负责把作业提交到队列中。

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。



公平调度器 Fair Scheduler 最初是由 Facebook 开发设计使得 Hadoop 应用能够被多用户公平地共享整个集群资源，现被 Cloudera CDH 所采用。Fair Scheduler 不需要保留集群的资源，因为它会动态在所有正在运行的作业之间平衡资源。

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。

五、 Hadoop High Availability

HA(High Available)，高可用，是保证业务连续性的有效解决方案，一般有两个或两个以上的节点，分为活动节点（Active）及备用节点（Standby）。通常把正在执行业务的称为活动节点，而作为活动节点的一个备份的则称为备用节点。当活动节点出现问题，导致正在运行的业务（任务）不能正常运行时，备用节点此时就会侦测到，并立即接续活动节点来执行业务。从而实现业务的不中断或短暂中断。

Hadoop1.X 版本，NN 是 HDFS 集群的单点故障点，每一个集群只有一个 NN，如果这个机器或进程不可用，整个集群就无法使用。为了解决这个问题，出现了一堆针对 HDFS HA 的解决方案（如：Linux HA, VMware FT, shared NAS+NFS, BookKeeper, QJM/Quorum Journal Manager, BackupNode 等）。

在 HA 具体实现方法不同情况下，HA 框架的流程是一致的，不一致的就是如何存储、管理、同步 edits 编辑日志文件。

在 Active NN 和 Standby NN 之间要有个共享的存储日志的地方，Active NN 把 edit Log 写到这个共享的存储日志的地方，Standby NN 去读取日志然后执行，这样 Active 和 Standby NN 内存中的 HDFS 元数据保持着同步。一旦发生主从切换 Standby NN 可以尽快接管 Active NN 的工作。

1. Namenode HA

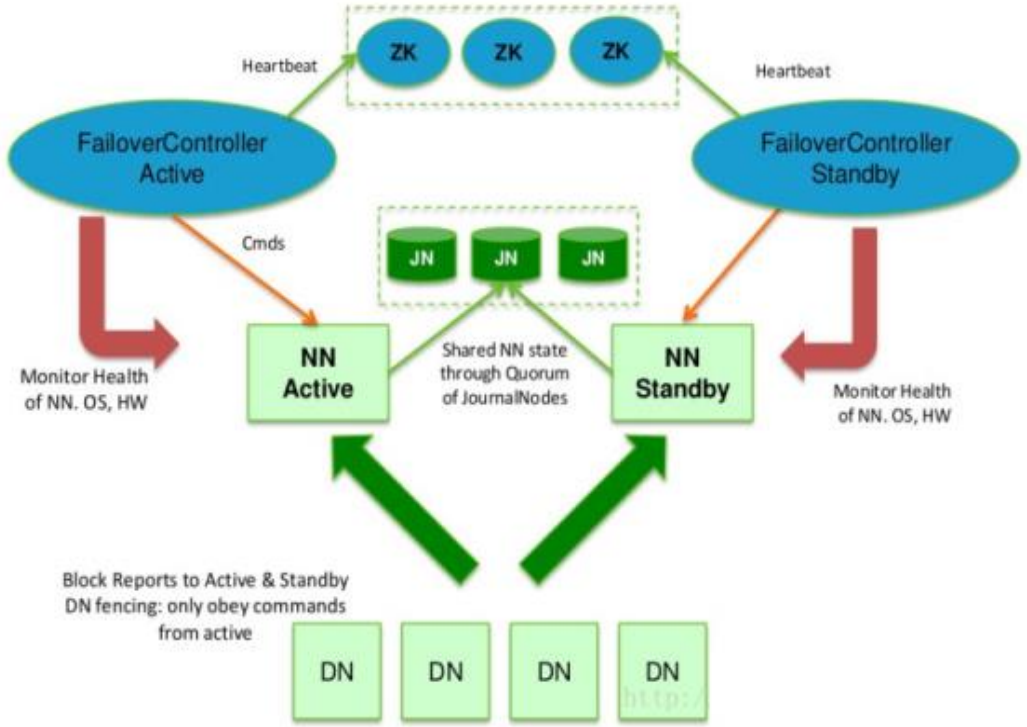
1.1. Namenode HA 详解

hadoop2.x 之后，Clouera 提出了 QJM/Qurom Journal Manager，这是一个基于 Paxos 算法（分布式一致性算法）实现的 HDFS HA 方案，它给出了一种较好的解决思路和方案,QJM 主要优势如下：

不需要配置额外的高共享存储，降低了复杂度和维护成本。

消除 spof(单点故障)。

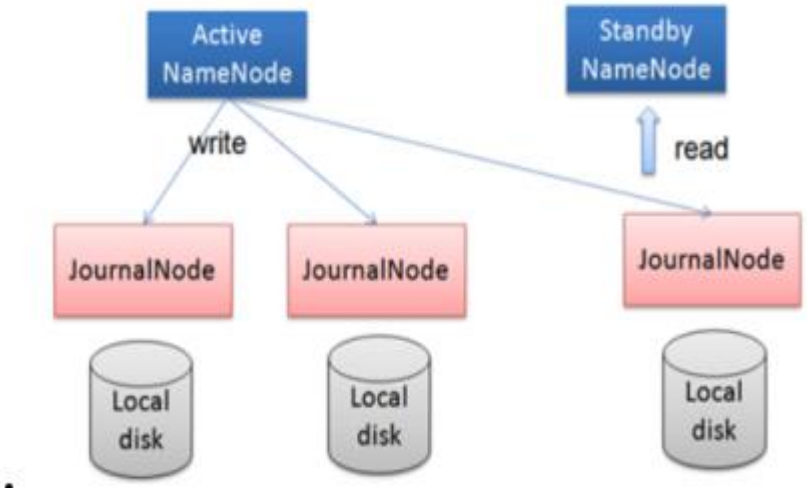
系统鲁棒性(Robust)的程度可配置、可扩展。



基本原理就是用 2N+1 台 JournalNode 存储 EditLog，每次写数据操作有>=N+1 返回成功时即认为该次写成功，数据不会丢失了。当然这个算法所能容忍的是最多有 N 台机器挂掉，如果多于 N 台挂掉，这个算法就失效了。这个原理是基于 Paxos 算法。

在 HA 架构里面 SecondaryNameNode 已经不存在了，为了保持 standby NN 时时的与 Active NN 的元数据保持一致，他们之间交互通过 JournalNode 进行操作同步。

任何修改操作在 Active NN 上执行时，JournalNode 进程同时也会记录修改 log 到至少半数以上的 JN 中，这时 Standby NN 监测到 JN 里面的同步 log 发生变化了会读取 JN 里面的修改 log，然后同步到自己的目录镜像树里面，如下图：



当发生故障时，Active 的 NN 挂掉后，Standby NN 会在它成为 Active NN 前，读取所有的 JN 里面的修改日志，这样就能高可靠的保证与挂掉的 NN 的目录镜像树一致，然后无缝的接替它的职责，维护来自客户端请求，从而达到一个高可用的目的。

在 HA 模式下，datanode 需要确保同一时间有且只有一个 NN 能命令 DN。为此：

每个 NN 改变状态的时候，向 DN 发送自己的状态和一个序列号。

DN 在运行过程中维护此序列号，当 failover 时，新的 NN 在返回 DN 心跳时会返回自己的 active 状态和一个更大的序列号。DN 接收到这个返回则认为该 NN 为新的 active。

如果这时原来的 active NN 恢复，返回给 DN 的心跳信息包含 active 状态和原来的序列号，这时 DN 就会拒绝这个 NN 的命令。

1.2. Failover Controller

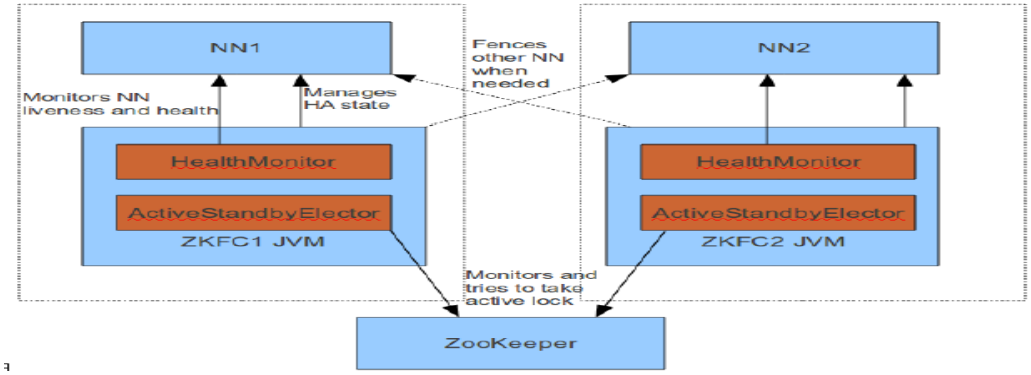
HA 模式下，会将 FailoverController 部署在每个 NameNode 的节点上，作为一个单独的进程用来监视 NN 的健康状态。**FailoverController 主要包括三个组件：**

HealthMonitor：监控 NameNode 是否处于 unavailable 或 unhealthy 状态。当前通过 RPC 调用 NN 相应的方法完成。

ActiveStandbyElector：监控 NN 在 ZK 中的状态。

ZKFailoverController：订阅 HealthMonitor 和 ActiveStandbyElector 的事件，并管理 NN 的状态，另外 zkfc 还负责解决 fencing（也就是脑裂问题）。

上述三个组件都在跑在一个 JVM 中，这个 JVM 与 NN 的 JVM 在同一个机器上。但是两个独立的进程。一个典型的 HA 集群，有两个 NN 组成，每个 NN 都有自己的 ZKFC 进程。



ZKFailoverController 主要职责：

- **健康监测：**周期性的向它监控的 NN 发送健康探测命令，从而来确定某个 NameNode 是否处于健康状态，如果机器宕机，心跳失败，那么 zkfc 就会标记它处于一个不健康的状态
- **会话管理：**如果 NN 是健康的，zkfc 就会在 zookeeper 中保持一个打开的会话，如果 NameNode 同时还是 Active 状态的，那么 zkfc 还会在 Zookeeper 中占有一个类型为短暂类型的 znode，当这个 NN 挂掉时，这个 znode 将会被删除，然后备用的 NN 将会得到这把锁，升级为主 NN，同时标记状态为 Active
- 当宕机的 NN 新启动时，它会再次注册 zookeper，发现已经有 znode 锁了，便会自动变为 Standby 状态，如此往复循环，保证高可靠，需要注意，目前仅仅支持最多配置 2 个 NN
- **master 选举：**通过在 zookeeper 中维持一个短暂类型的 znode，来实现抢占式的锁机制，从而判断那个 NameNode 为 Active 状态

2． Yarn HA

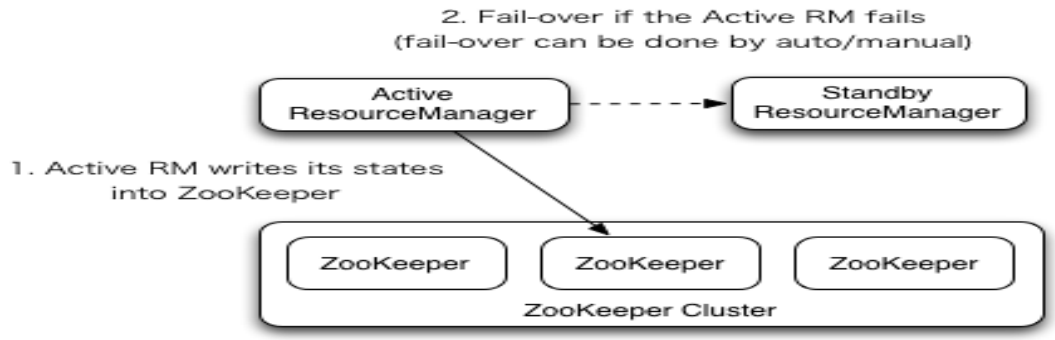
Yarn 作为资源管理系统，是上层计算框架（如 MapReduce,Spark）的基础。在 Hadoop 2.4.0 版本之前，Yarn 存在单点故障（即 ResourceManager 存在单点故障），一旦发生故障，恢复时间较长，且会导致正在运行的 Application 丢失，影响范围较大。从 Hadoop 2.4.0 版本开始，Yarn 实现了 ResourceManager HA，在发生故障时自动 failover，大大提高了服务的可靠性。

ResourceManager（简写为 RM）作为 Yarn 系统中的主控节点，负责整个系统的资源管理和调度，内部维护了各个应用程序的 ApplictionMaster 信息、NodeManager（简写为 NM）信息、资源使用等。由于资源使用情况和 NodeManager 信息都可以通过 NodeManager 的心跳机制重新构建出来，因此只需要对 ApplicationMaster 相关的信息进行持久化存储即可。

在一个典型的 HA 集群中，两台独立的机器被配置成 ResourceManger。在任意时间，有且只允许一个活动的 ResourceManger,另外一个备用。切换分为两种方式：

手动切换：在自动恢复不可用时，管理员可用手动切换状态，或是从 Active 到 Standby，或是从 Standby 到 Active。

自动切换：基于 Zookeeper，但是区别于 HDFS 的 HA，2 个节点间无需配置额外的 ZFKC 守护进程来同步数据。



3． Hadoop HA 集群的搭建

HA 集群搭建的难度主要在于配置文件的编写，**心细，心细，心细**！

详细的搭建安装步骤请参考附件资料。