# Yarn 框架代码详细分析 V0.3

阿里巴巴 封神 2012 年 1 月 http://fengshenwu.com/blog

## 目录

版本介绍

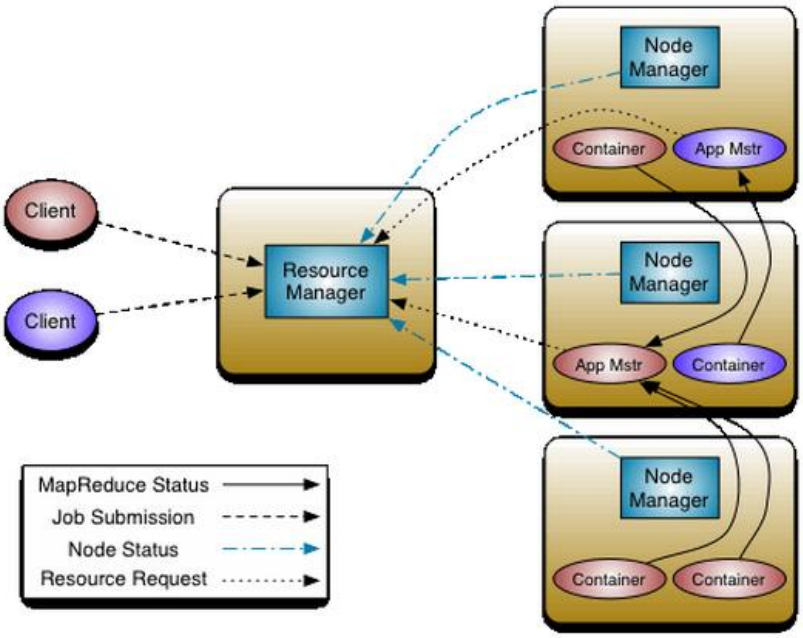
V0.1 yarn 简单介绍、yarn 模块详细分析、Jobhistory 机制介绍
V0.2 添加 RM 调度器、MRAppMaster 分配器
V0.3 完善调度器的描述，添加一些细节描述

# 一、yarn 简单介绍

## 1.1、概述

每个框架一开始都不会考虑太多的问题，也很难预料到未来会发生什么。HADOOP 诞生时的目标就是为了支持几十台机器的规模。随着互联网企业的发展，数据的规模越来越大，在寻找技术方案的时候，越来越多的公司选择了开源的 HADOOP，越来越多的人也参与到了 HADOOP 的发展中。由于海量数据的需要，HADOOP 所需要支持的机器数越来越多，动辄就几千台的规模，目前阿里巴巴的云梯集群已经有 3400 台的规模，且这个数据还在持续增长。HADOOP 自身框架的问题限制了（有一个 JobTracker、NameNode 单点）集群的规模。这主要是两方面的，一方面是存储 HDFS 层面的，这个社区在极力发展 Federation；另一方面是计算层面，一些技术能力较强的公司开始在寻求解决方案(如：facebook 的 corona,和 YARN 极其相似)，HADOOP 开源社区也在积极寻求解决方案，从 2008 年初，就创建了 MAPREDUCE-279 来讨论、跟踪下一代 MAP-REDUCE 的发展。本文主要讲述下一代 MAP-REDUCE（也称为 YARN）的框架设计及代码细节，主要的阅读对象是开发人员或者对 YARN 代码细节感兴趣的同学，一些 YARN 的基本知识请参考 http://hadoop.apache.org。

我们首先看下官方给出的一个比较高层面的框架图，从中我们可以看出还是存在一个全局的 master 节点，这个节点也会限制计算节点的规模。目前，我们只能说，在现在的需求下（大约 10w 计算节点），这个节点应该不会出现问题。在可预计的未来，谁能说这不是一个限制呢？




简单地讲述下这个图，详细的情况请参考：

http://hadoop.apache.org/docs/r0.23.5/hadoop-yarn/hadoop-yarn-site/YARN.html

http://blog.cloudera.com/blog/2012/02/mapreduce-2-0-in-hadoop-0-23/

以前的 JobTracker 功能太多了，在 YARN 中大约分成了 3 快，一部分是 ResourceManager，负责 Scheduler 及 ApplicationsManager；一部分是 ApplicationMaster，负责 job 的生命周期管理；最后一部分是 JobHistroyServer，负责日志的展示。

为了支持更多的计算模型，把以前的 TaskTracker 替换成了 NodeManager。NodeManager 管理各种各样的 container。Container 才是真正干活的。计算模型相关的事情可以放在 NodeManager 的一个扩展服务中，如 MAP-REDUCE 的 shuffle。

## 1.2、YARN 的优势

YARN 本身框架的优势是扩展性与支持多计算模型。对于扩展性目前主要体现在计算节点规模上，以前 JobTracker-TaskTracker 模型下最多大约在 5000 台机器左右，对于 YARN，官方说可以支持大约 10w 台机器，当然这个目前还没有一家公司去试用过，连 300 台机器目前估计也是测试阶段。对于支持多计算模型，目前 YARN 理论是可以支持更多的计算模型的，如：MAP-REDUCE、MPI、Giraph、Spark 等。目前 MAP-REDUCE 是默认支持的，MPI 见 MAPREDUCE-2911。

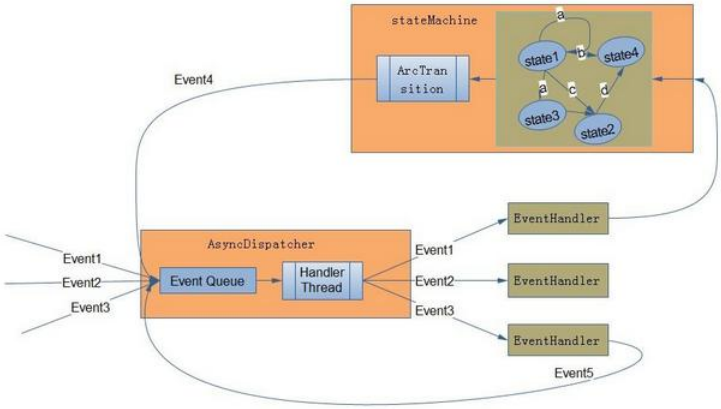
下面一些优势，只能说是重构或者在原来的框架上面也可以实现。目前我整理的有如下几点：

- 调度器的资源模型及资源隔离，JobTracker 时代还是使用 slot，这个是很不合理的。在资源模型包括内存、CPU、磁盘、网络等情况下，机器资源的利用将更加合理。
- JobHistory Sever 从 JobTracker 分离出来，减轻了 master 节点的压力。
- uber appMaster 模式，可以在一台机器上面运行所有的 task。这个其实是可以把计算计算全部算在集群内，local 模式需要使用客户端的资源。
- shuffle 性能提升 30%+，主要是使用 netty 重写了。

不可否认的是在框架层面，YARN 在很多方面都做得比较优秀。但是这个不是没有代价的，代价就是 YARN 更加复杂，更加让人迷糊，可能某些性能更加低下。当然我们也要感谢计算机软件的发展，目前我们有很多的软件设计的思想可以使用，来使我们的软件更加正确、可靠、高性能。但是 YARN 还是复杂的。例如：MAPREDUCE-3561 里面列举的一些 case。

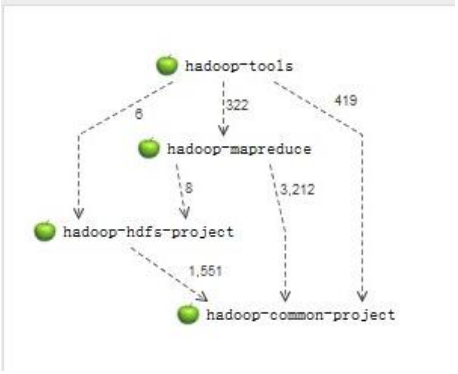
## 1.3、新框架下的软件设计模式

上个月，我写了一篇博客，结合我以前在架构方面的实践，总结了在 yarn 中的一些软件设计模式。其中最主要说了在 yarn 使用最广泛，也是最重要的三个设计模型：服务生命周期管理模式、事件驱动模式、状态驱动模式。详情参见《yarn(hadoop2)框架的一些软件设计模式》，最经典如下图，文中有详细的说明。




## 1.4、HADOOP 2.0.0- alpha 工程结构

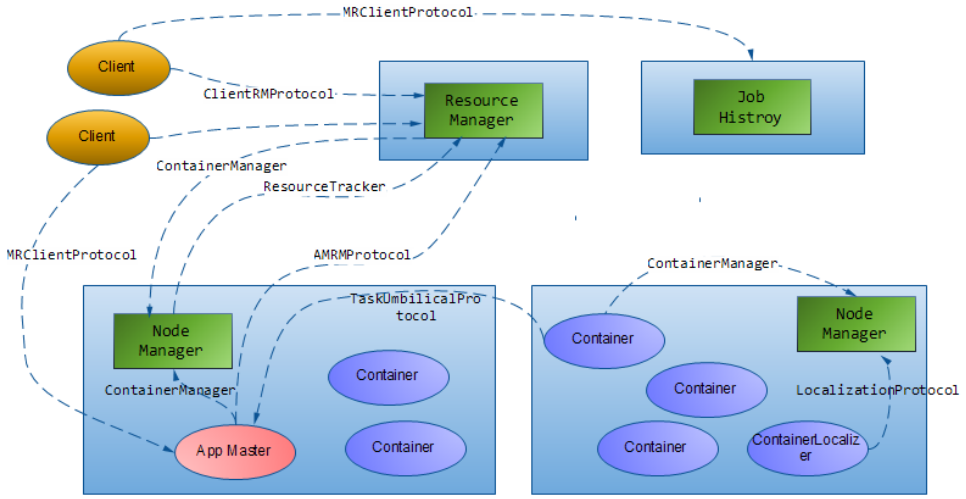
请参考《hadoop2 包结构及包功能大致介绍》，介绍了工程之间的依赖关系及每个工程的大致功能。在 1.x 以后，hadoop 开始采取 maven 来管理工程，代码结构就清晰了很多。在 YARN 版本中就有一个三十几个的包工程。



# 二、yarn 模块详细分析

## 2.1、接口

在 YARN 中，以 map-reduce 为例，守护进程有 RM、NM、JH；客户端的进程有 client；运行期的进程有 AppMaster、ContainerLocalizer、container（MRChild）。这些进程之间的交互如下图所示：




上面的 7 个进程之间大概有 7 个接口交互：

| 接口 | 介绍 |
| --- | --- |
| ClientRMProtocol | The protocol between clients and the ResourceManager to submit/abort jobs and to get information on applications, cluster metrics, nodes, queues and ACLs. |
| MRClientProtocol | 客户端向 MRAppMster 查询相关信息 |
| ContainerManager | The protocol between an ApplicationMaster/ResourceManager and a NodeManager to start/stop containers and to get status of running containers. |
| AMRMProtocol | This is used by the ApplicationMaster to register/unregister and to request and obtain resources in the cluster from the ResourceManager. |
| ResourceTracker | NM 向 RM 注册及心跳 |
| LocalizationProtocol | ContainerLocalizer 进程向 NM 报告资源下载进度 |
| TaskUmbilicalProtocol | MRChild 向 MRAppMaster 获取任务、报告进度等 |

大致的步骤是：

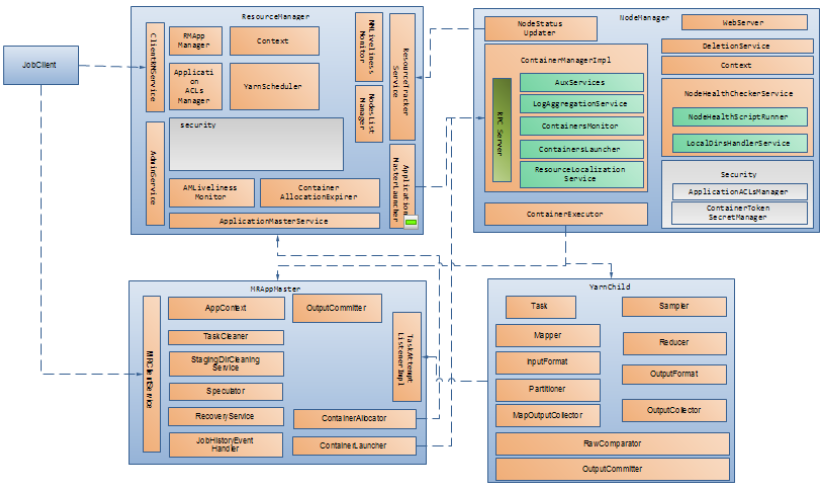
1. client 通过 ClientRMProtocol 向 RM 提交作业
2. RM 通过 ContainerManager 启动 AppMaster
3. AppMaster 通过 AMRMProtocol 向 RM 获取资源
4. AppMaster 通过 ContainerManager 向 NM 启动 MRChild
5. MRChild 通过 TaskUmbilicalProtocol 向 TaskUmbilicalProtocol 报告进度
6. Client 通过 MRClientProtocol 获取 job 的进度等信息

心跳汇报有：

- NM 向 RM 报告
- ContainerLocalizer 向 NM 报告下载资源进度
- MRAppMaster 向申请资源
- MRChild 向 AppMaster 报告进度

## 2.2、各大模块分析

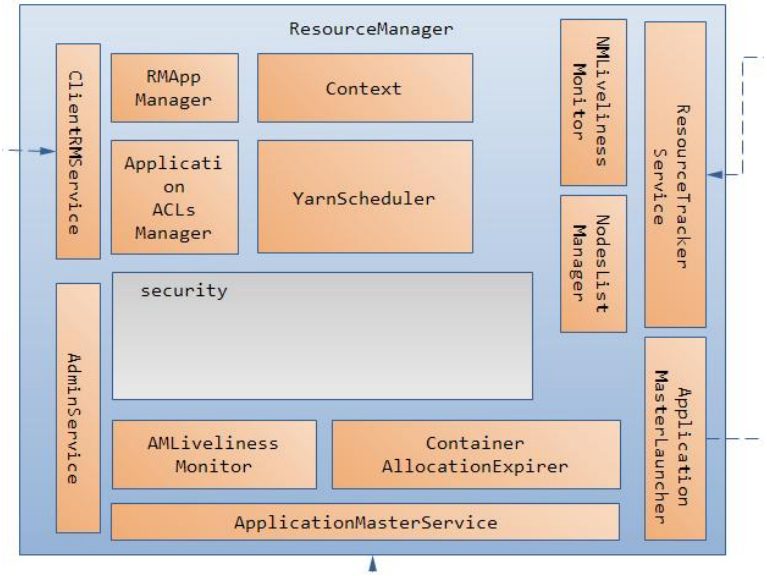
接下来将分析各个重要模块，RM、NM、MRAppMaster、MRYarnChild。如下如所示




每个模块都包括很多组件，这些组件之间通过上面讲的软件的设计模式组织在一起。接下来我们一一了解。

### 2.2.1、RM

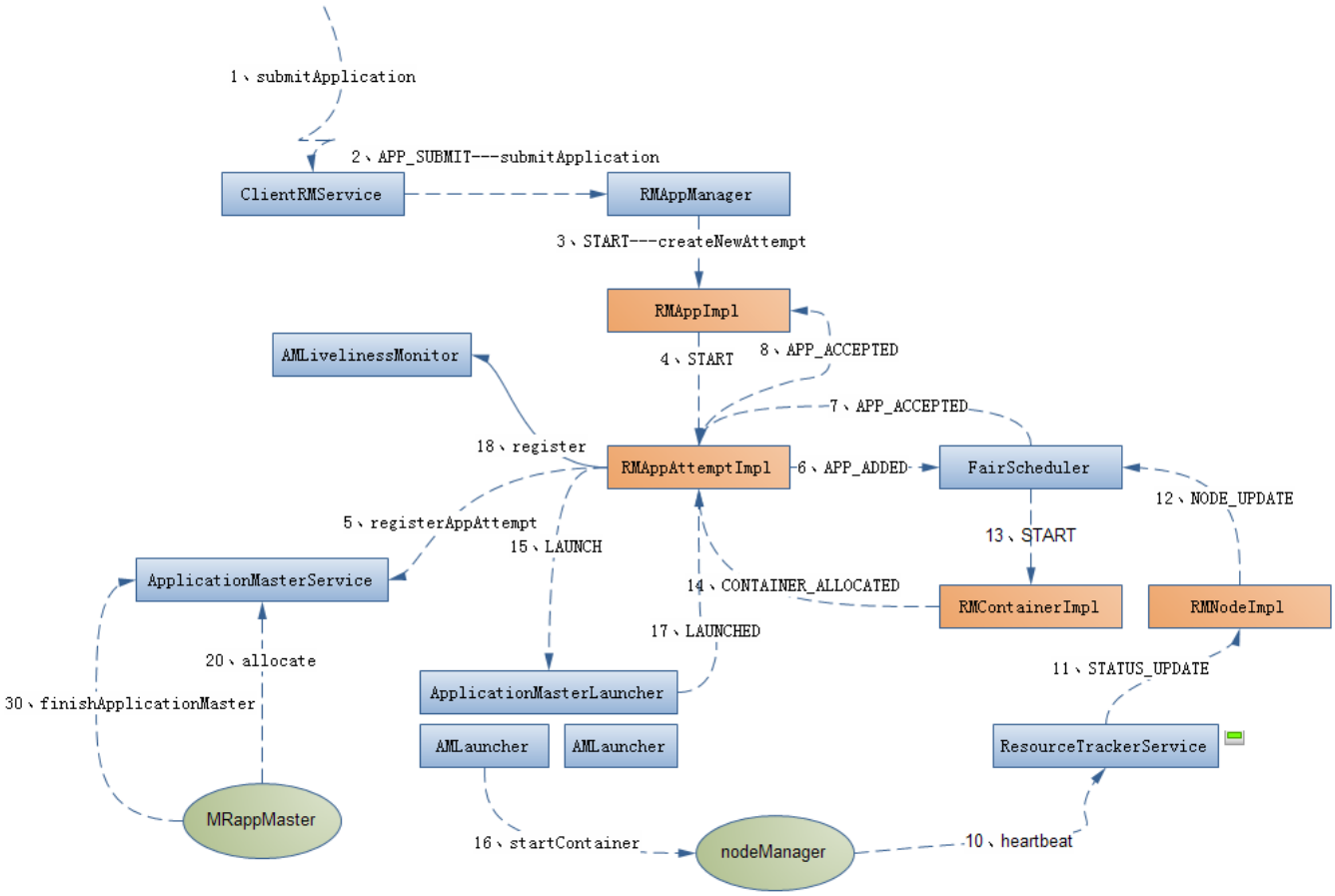
RM 主要做两件事情，一件是资源调度器，另一件是 appMaster、NM 的管理。




可以参考文章 [Apache Hadoop YARN – ResourceManager](), 其中从图中就大致看出，大致分为了 5 大部分：

- 客户端连接：RM 提供作业提交、管理功能。主要体现在 ClientService、AdminService 组件上。
- NM 管理：ResourceTrackerService 直接处理心跳，NMLivelinessMonitor 监控 NM，NodesListManager 提供 NM 的黑白名单等。
- appMaster 管理：ApplicationMasterService 提供 appMaster 的注册，获取资源等，AMLivelinessMonitor 监控 appMaster。
- 资源调度及相关：
  - ApplicationsManager 提供一个缓冲去提交作业，并响应相应的管理命令。
  - ApplicationMasterLauncher 有一个线程组保持最近的发布的记录并且有责任去清理 appMaster
  - YarnScheduler 调度资源，资源模型目前支持内存，以后会支持 CPU、IO、磁盘等。
- 安全：在每个模块中都有安全模块，基本这里没有仔细去深究。基本是 kerberos 的封装及相关的 Token 等

以下是一张作业提交的流程：（所有流程图都是：橙色方框中有状态机，椭圆形是一个进程，蓝色的方框流转的是基本事件机制）



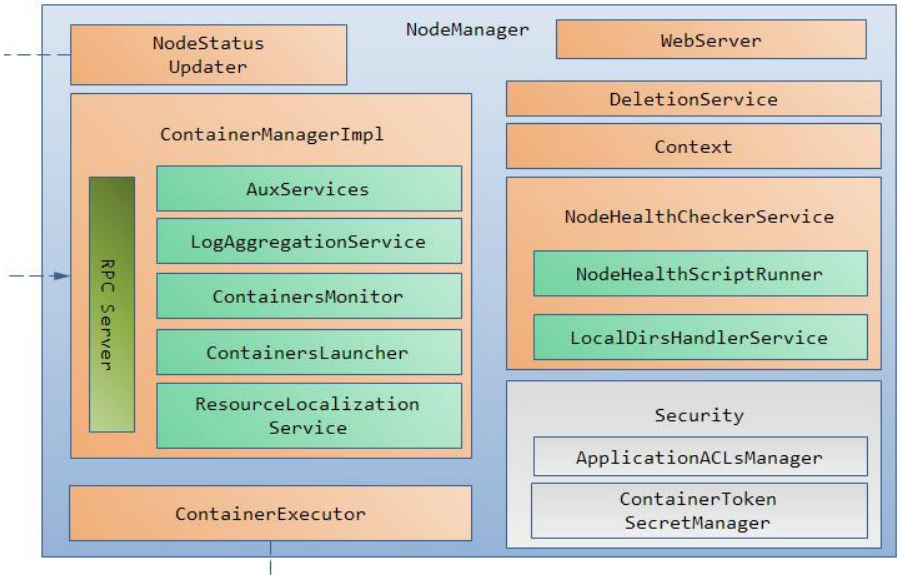

大致的过程：

1. 客户端提交作业后，初始化相关的 RMAppImpl、RMAppAttemptImpl 后直接提交到 Scheduler 组件中。图中的 1、2、3、4、5、6、7、8。
2. NM 产生心跳，ResourceTrackerService 处理后，到调度器选择出 job，最后到 ApplicationMasterLauncher 调用 NM，启动 appMaster。11、12、13、14、15、16、17
3. appMaster 随后就向 RM 注册且向 ApplicationMasterService 申请资源。20
4. appMaster 向 ApplicationMasterService 报告完成任务。30

在 RM 中有四个状态机 RMAPPImpl、RMAppAttemptImpl、RMContainerImpl 及 RMNodeImpl，其中每个状态机都负责相关模块的流转，互相无缝的衔接才能保证程序正确的运行。

## 2.2.2、NM

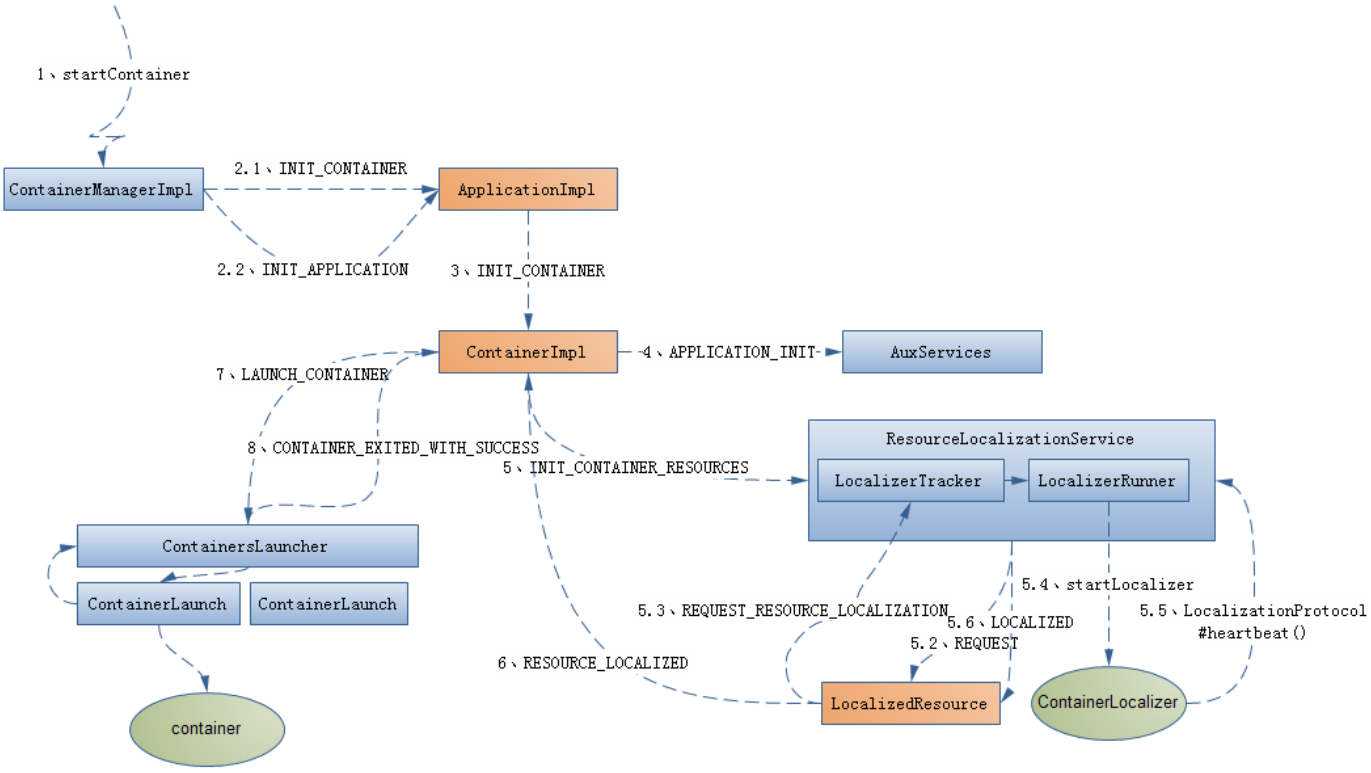
主要做三事情：管理 container、资源下载、健康检测后汇报




可以参考文章 Apache Hadoop YARN – NodeManager 不好分类，就一个个简单描述下。

- ResourceLocalizationService 安全地下载和管理各种资源文件。
- ContainersLauncher：用一个线程池去准备和快速启动各种 container。
- ContainersMonitor：检测 container 资源的利用，如果出现超标情况将会 kill。
- LogAggregationService：收集上传 container 产生的各种日志。
- AuxServices：通过配置服务的方式来扩展 NM 的功能，如 shuffle。
- ContainerExecutor：跟操作系统相关的启动执行器。
- NodeHealthCheckerService：收集相关的运行时期的信息，通过 NodeStatusUpdater 发送给 RM。
- NodeStatusUpdater 其实就是状态更新，定期向 MR 汇报本节点的资源利用情况。

以下是一张启动一个 container 的流程图，大致经过如下的步骤：

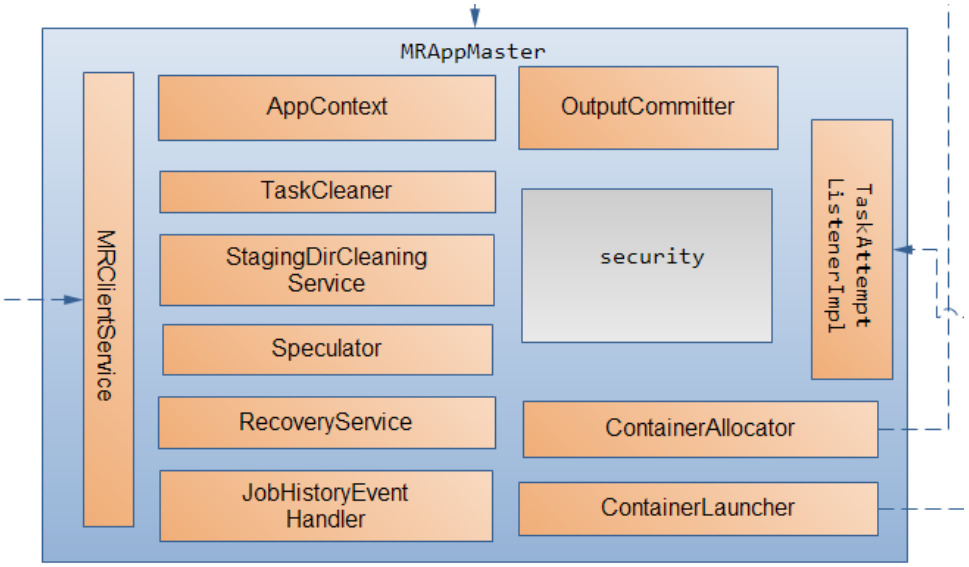



大致的过程：

1. RM或者appMaster通过接口发送startContainer命令后，开始初始化ApplicationImpl及ContainerImpl，注册AuxServices。1、2、3、4
2. 资源下载：5
   a) 启动一个进程containerLocalier专门负责下载
   b) containerLocalier通过心跳报告进度
   c) 下载完成后触发事件给containerImpl
3. containerImpl通过containersLauncher启动container。7
4. 当container退出时，containersLauncher会发送事件给containerImpl，再完成一些资源的释放。8

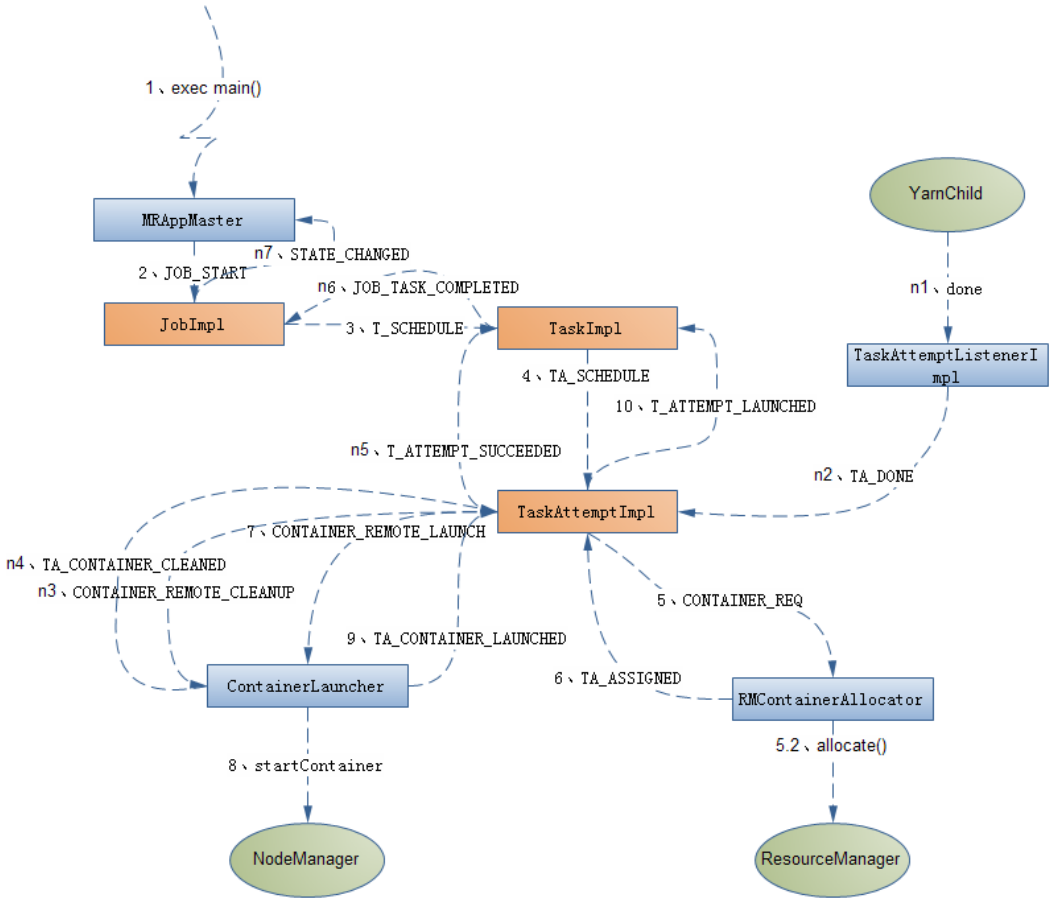
### 2.2.3、MRAppMaster

MRAppMaster主要就是管理job中的各个task的，当然也包括job内的资源分配等




- ContainerAllocator：向RM申请资源，且job内调度task。
- ContainerLauncher：请求NM启动MRYarnChild。
- RecoveryService：MRAppMaster重新启动的时候，恢复上一次运行时的环境。
- MRClientService：提供客户端查询进度等相关信息。
- JobHistoryEventHandler：处理jobhistory信息，目前作业信息存储在hdfs上面。
- Speculator：推测执行

以下是一张启动 MRAppMaster 后的流程图，大致经过如下的步骤

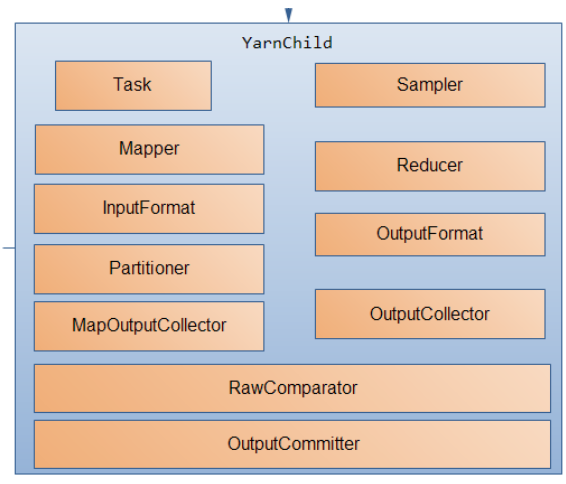



大致的过程：

1. MRAppMaster 启动最后一步会发送启动 JobImpl 的事件，后初始化 TaskImpl 及 TaskAttemptImpl。1、2、3、4、5
2. ContainerAllocator 自身有一个线程固定周期循环获取资源，当获取后再发事件给 TaskAttemptImpl。5.2、6
3. TaskAttemptImpl 发送事件给 ContainerLauncher。7
4. ContainerLauncher 通过 NM 启动 MRYarnChild，再发送消息给 TaskAttemptImpl。8、9、10
5. MRYarnChild 调用接口 done()，后资源清理，后一个 task 就完成了。n1、n2
6. TaskAttemptImpl 会清理 container 资源(基本是 kill)，再发送完成消息给 TaskImpl。n3、n4、n5
7. 每次一个 task 完成，都会通知 JobImpl，会判断是否所有的 task 完成，如果完成则发送作业完成的消息给 MRAppMaster，MRAppMaster 则会启动 stop()流程，依次释放资源，清理资源等。n6、n7

## 2.2.4、MRYarnChild

MRYarnChild 就是真正的干事情的 container 的了。里面有很多的元素，这个工程和 Hadoop MRv1 基没有大的变化。这里不展开了，参考《Hadoop 权威指南》

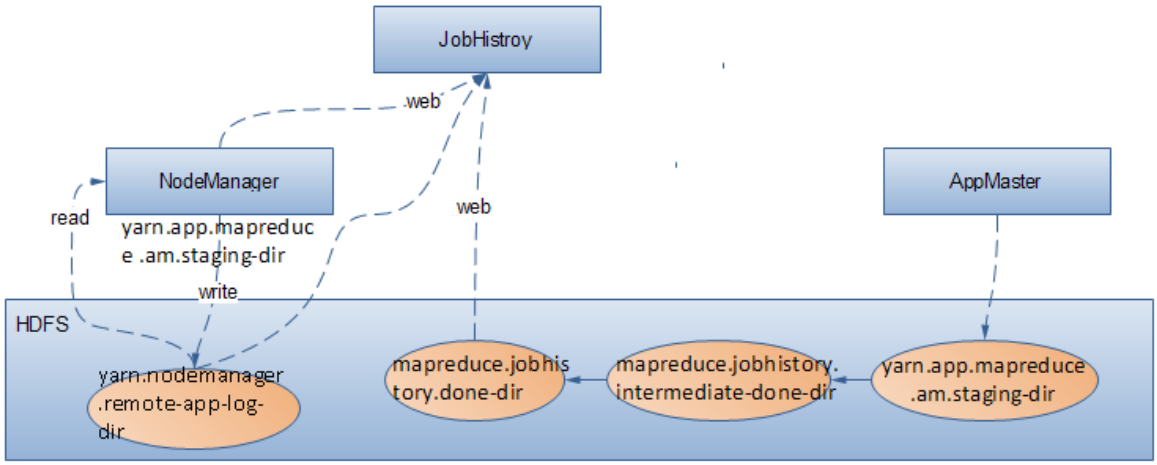


# 三、功能点详细分析

## 3.1、Jobhistory 机制

这个主要涉及到 NM、MRYarnChild 及 JobHistroy server。在 MRv1 中，JobHistroy server 是嵌入在 Jobtracker 中的，当有大量的查询时，对 Jobtracker 造成很大的压力，所以阿里巴巴基本是自己实现了一套 JobHistroy server 服务器。在 YARN 开发过程中，看到了这个问题。

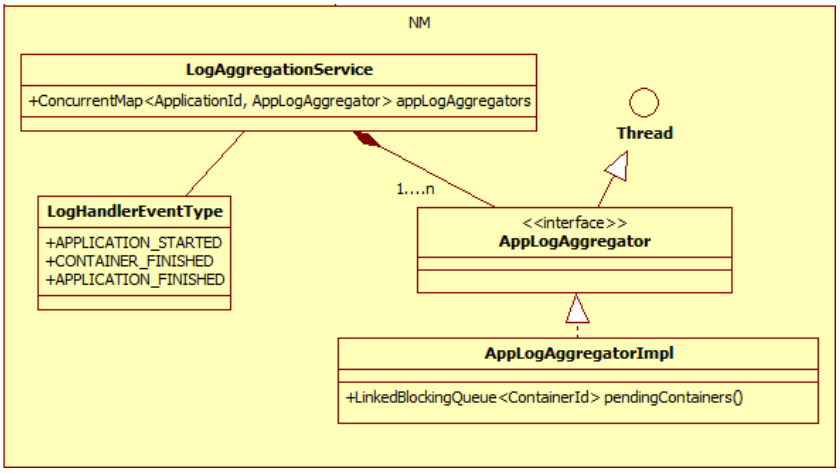
下图是数据流向图：




- NM 会收集 container 的 log 日志，主要包括标准输入、标准输出、log4j。收集完成后会存储到指定的 HDFS 目录中。
- MRAppMaster 运行的过程中会收集一些重要的过程式的事件并存储在${yarn.app.mapreduce.am.stagung-dir}中，job 完成后会再把文件 rename 到${mapreduce.jobhistory.intermediate-done-dir}中。
- JobHistory 服务解析 logs 和 JobHistory 日志，用户可以通过页面来查询。Ps：YARN 的页面比 MRV1 的好看。

### 3.1.1. NM 收集日志

以下是 NM 收集日志的类图：



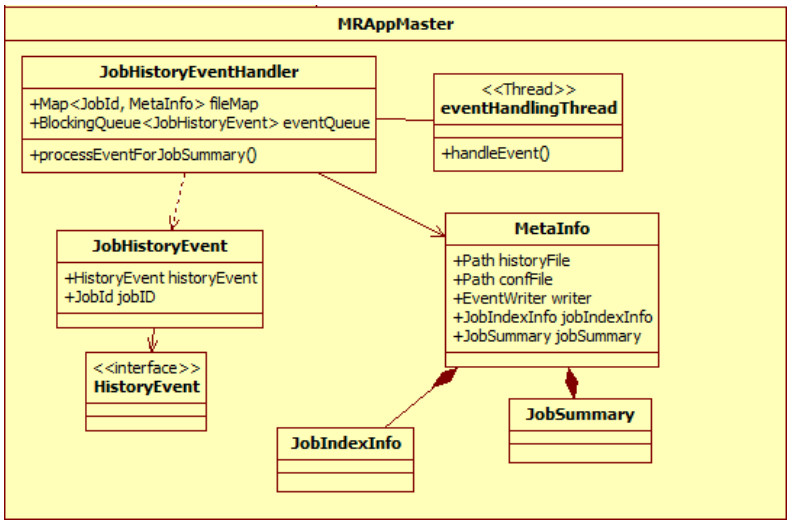

NM 会初始化 LogAggregationService 服务，他是一个 service 也是一个 EventHandler。LogAggregationService 处理的事件如下：

- APPLICATION_STARTED：创建目录,启动上传线程 AppLogAggregatorImpl(线程用 LinkedBlockingQueue 同步)
- CONTAINER_FINISHED：container 完成后就可以上传日志了,并删除本地的日志
- APPLICATION_FINISHED:设置 application 已经完成(appFinishing=true)，这样上传线程就可以不用睡眠等待就可以上传 job 的所有 container 日志

其中一个 AppLogAggregatorImpl 处理一个 job 的日志。

### 3.1.2. MRAppMaster 收集 JobHistory

以下是 MRAppMaster 的 JobHistory 收集类图：




JobHistoryEventHandler 既是一个 service 也是一个 EventHandler，处理大约 30 个左右的事件（这些事件会在 MRAppMaster 的一些关键点产生）。JobHistoryEventHandler 接受到事件后就 push 到阻塞队列，eventHandlingThread 线程会从阻塞队列中获取事件，调用 handleEvent()处理，主要有四步骤：

- setupEventWriter 初始化路径，向 hdfs 写 jobFile.xml 文件
- 事件所带的信息写入到 hdfs 的 xxx.jhist 文件中
- 向内存写 JobSummary 信息及 JobIndexInfo 信息
- 作业完成、被杀、失败，调用 closeEventWriter 把文件写入 hdfs 及重命名。三个文件：jobFile.xml、xxx.jhist、JobSummary 文件。

其中 MetaInfo 持有一些元数据信息，最主要的是 JobIndexInfo 与 JobSummary。

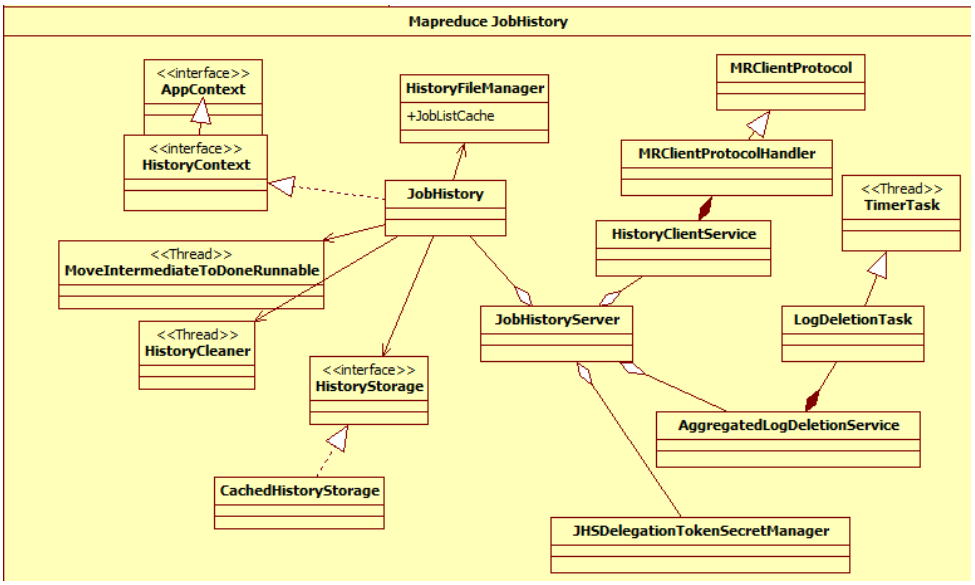
- JobIndexInfo 信息不写入 hdfs，直接组成 xxx.jhist 的文件名称，如
  job_1356079352172_0002-1356080889706-hadoop-word+count-1356080764388-1-1-SUCCEEDED-default.jhist
  即：JobId-StartTime-UserName-JobName-FinishTime-NumMapsNumReduces-JobStatus-QueueName.jhist
- JobSummary 有一些 JobIndexInfo 所没有的信息，JobSummary 是存储在 HDFS 中的。

### 3.1.3. JobHistoryServer

日志服务有一个专门的 historyServer 进程处理。historyServer 是单独部署的，代码也是在单独 hadoop-mapreduce-client-hs 工程中。

主要有以下几个方面：

- JobHistory：加载和管理 Job history 的缓存。
- AggregatedLogDeletionService：日志文件的管理。
- HistoryClientService：提供一些服务供 JobClient 查询。
- HsWebApp：页面的组装，数据的拼接，响应页面请求等。

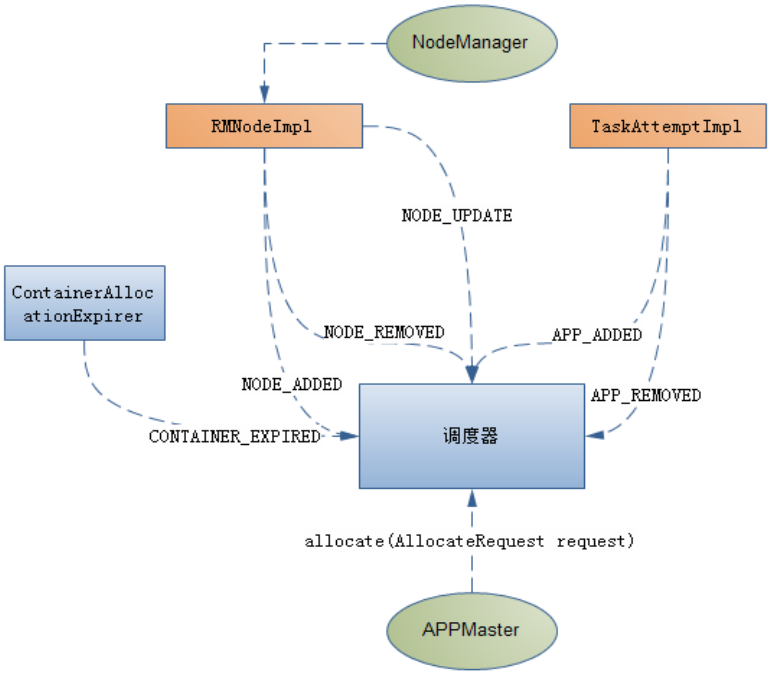


## 3.2、RM 调度器

### 3.2.1、简述

在 HADOOP 中，主要有三种调度器为：FifoScheduler、CapacityScheduler、FairScheduler。前两个调度器相对比较简单，阿里巴巴选择的是 FairScheduler，这里主要讲述下 FairScheduler 的详细情况。

Scheduler 处理的大致流程如图所示：




处理 6 个事件，下表是他们什么时候发生及会产生什么样的效果的一个列表，在设计的时候，只要一个没有考虑到就会出现问题，最严重的就是系统不可用。

当 APPMaster 申请资源是按照一定的请求模型来申请，首先，按照优先级别来划分；在同一优先级内，又分为了三个层次，为：node、rack、*。他们各自的资源数目都表示在这个层次的最大所申请的资源数。起到一个一定范围的限制作用。

- Node 是申请到具体到一个节点，如，我申请 node1 节点 2 个 2048m 的资源。
- rack 是申请到具体到一个机架，如：我申请 rack1 机架的 2 个 1024m 的资源。分配的时候，只要是 rack1 机架上的 node 节点都可以的。
- *是代码可以在整个集群申请资源。

他们之间有一定的约束关系的，例子：

```
priority:{Priority: 20}
        nodeAddress=r02i02009.yh.aliyun.com numContainers=12 request={Priority: {Priority: 20}, Capability: memory: 1024}
        nodeAddress=r02i02010.yh.aliyun.com numContainers=13 request={Priority: {Priority: 20}, Capability: memory: 1024}
        nodeAddress=/default-rack numContainers=16 request={Priority: {Priority: 20}, Capability: memory: 1024}
        nodeAddress=r03f05046.yh.aliyun.com numContainers=14 request={Priority: {Priority: 20}, Capability: memory: 1024}
        nodeAddress=* numContainers=17 request={Priority: {Priority: 20}, Capability: memory: 1024}
        nodeAddress=r03f11008.yh.aliyun.com numContainers=9 request={Priority: {Priority: 20}, Capability: memory: 1024}
```

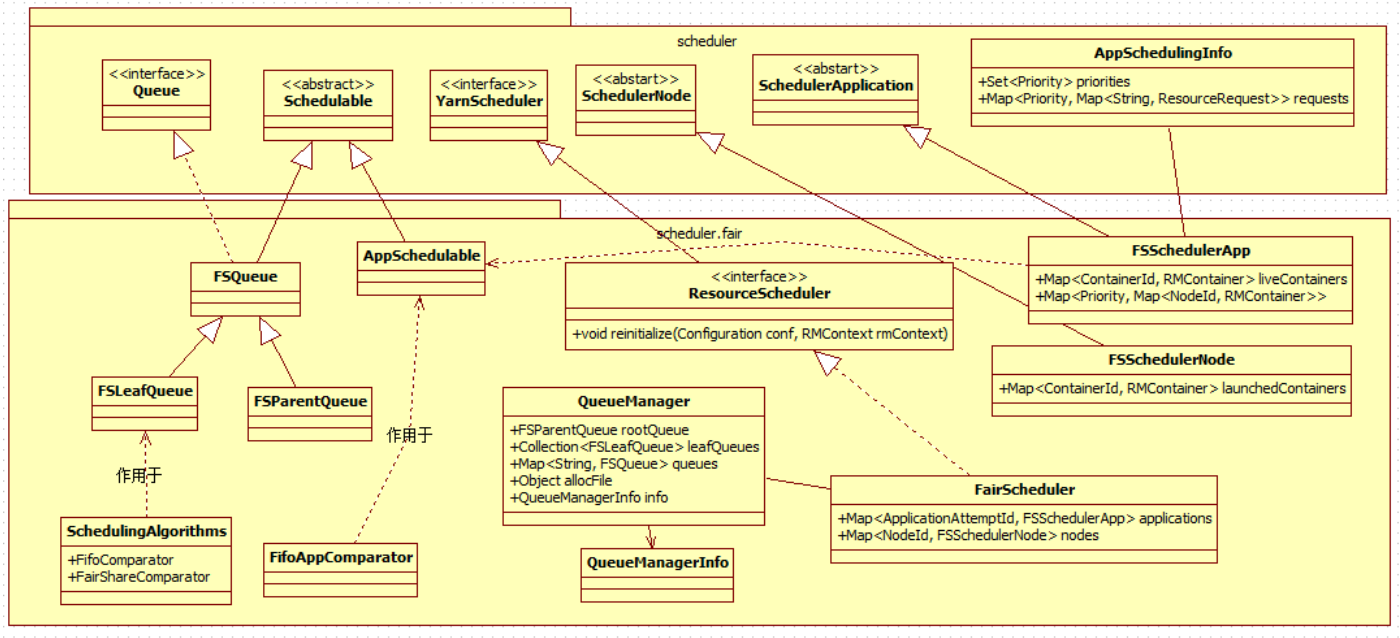
优先级 20 中代表的含义是：共计申请 17 个资源，其中/default-rack 最多为 16 个，每个机器分别最多为相应的数字。

在分配的时候，node1 心跳来了，如果存在 node1 的资源请求，优先分配 1 个资源请求给 node1，则 node1 对应的 rack 和*分别减 1，如果 rack 或者*的资源申请减 1 后小于 0，则不能分配给 node1。rack 类似。

| 事件 | 触发条件 | 产生的效果 |
|---|---|---|
| NODE_ADD | 新节点注册，老节点重新启动，老节点重新报告为健康状态 | 增加集群总容量、放入 node 队列中 |
| NODE_REMOVED | 老节点报告为不健康状态，老节点再次接通且端口有变化，老节点下线(DECOMMISSION), | 减少集群总容量、移除 node 队列、kill 相关正在运行的 container |
| CONTAINER_EXPIRED | 已经分配出资源后,等待被 AppMaster 后,NM 反馈启动成功的时间(在 RMContainer 状态图上,就是停留在 ACQUIRED 的时间)超过预计时间，默认是 10 分钟。有 3 个事件会引起离开，NM 已经 luanch container、NM 已经完成 container、appMaster 主动释放 container。 | 释放相关 container 资源等。 |
| APP_ADDED | 客户提交作业后，初始化一些环境后，由 RMAppAttemptImpl 提交 job | 先验证权限,再向对应的 queue 提交 job |
| APP_REMOVED | 作业完成，被 kill，失败，container Crashed | Kill 所有的 AppMaster 持有的 container 并释放资源 |
| NODE_UPDATE | NM 的心跳触发 | 更新已经完成的和刚启动的 container，在此 node 上面分配和预订资源。这个后面在详细讲述。 |

## 3.2.2、FairScheduler 代码分析

在 2.0.3 中，FairScheduler 的类图大概如下图所示：

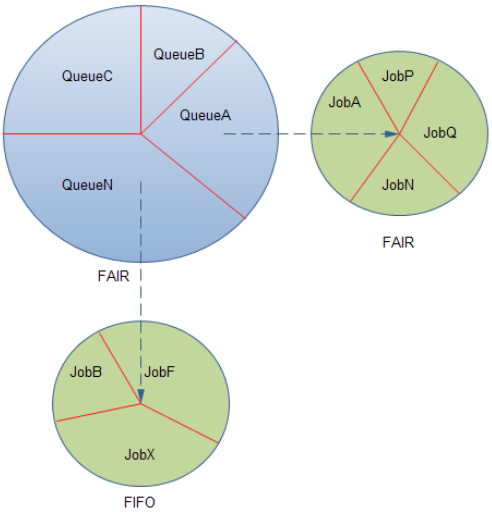



其中各个类的用途描述如下，其中关于度量没有包含其中。

- FairScheduler 是一个 EventHandler，响应上文提到的 6 个事件；allocate 是 AppMaster 获取资源及释放不需要的资源的接口；提供各个报告；内置有一个 UpdateThread 线程，默认每 0.5s 把资源重新预分配一次并确认是否进行抢占。
- QueueManager 提供了解析 fair-scheduler.xml 文件并重新设置 FairScheduler 相关的变量。
- Schedulable 继承体系有 3 个实现类，分别是根节点调度器 FSParentQueue、各个 queue 调度器 FSLeafQueue、每个 App 的调度器 AppSchedulable。
- FSSchedulerApp 封装了 app 的一些状态及维护，一些信息存在 AppSchedulingInfo 中，如：app 的资源请求 Map<Priority, Map<String, ResourceRequest>> requests。
- FSSchedulerNode 封装了 node 的一些状态及维护，如：reservedContainer 与已经分配的 launchedContainers。
- WeightAdjuster 提供了自定义的权重调整方案，默认为 NewAppWeightBooster。
- FifoAppComparator 提供 AppSchedulable 的一个比较器。
- SchedulingAlgorithms 提供相关的比较器并且提供如何计算 R 值的算法。

## 3.2.3、FairScheduler 资源预分配

这里主要涉及的是，把所有的资源总量预分配给各个 queue 及 job 的过程，目前这里的资源是指内存，对于 CPU 的支持社区正考虑中。如何合理地分配有限的资源，这在所有涉及到共享资源的系统中都会涉及。下面主要简单讲下 FairScheduler。FairScheduler 使多个 job 在有限的资源情况下，都能得到合理公平的运行，不会使作业出现长时间的饥饿。




大致的分配思路就是如何把一快蛋糕分完，这里首先需要一个分组 queue，job 是被提交到相应的 queue 内的。每个 queue 都有一个最大和最小资源量。job 没有最大资源量的限制，最小资源量为 0。FairShare 的数值表示预分配给每个 queue 及 job 的资源量，如果没有抢占并不代表在实际分配的时候就一定能拿到，一般的实际的数值基本在 FairShare 左右徘徊。FairShare 算出来主要是为了抢占的时候使用的。

预分配的流程：

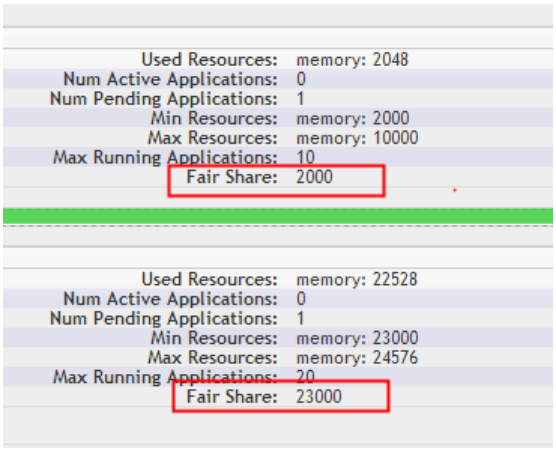
1. 如果配置文件已经更新，则重新 load 配置文件。
2. 先按照 FIFO 排序(优先级、开始时间、application 编号)所有的 job，再判断 job 是否可以运行。如果出现以下情况，则 job 不可运行：
   - 超过单一用户提交的最大 job 数

- 超过单一队列的最大 job 数

3. 更新相关抢占变量，这里主要更新 queue 的关于抢占的相关时间。
4. 更新每个 queue 及 job 的需求量(Demand)
   - 对于 job 是所有的请求量之和+已经使用的量
   - 对于 queue 是 $\mathbf{Min\{QueueMaxResource,\ \sum_{k=1}^{n} Job_k.demand\}}$
5. 计算各个 queue 及 job 的 FairShare 数值。这里存在一个 R 值，会满足如下的等式：

   $$\mathbf{TotalResource} = \sum_{k=1}^{n} \mathbf{min}\{\mathbf{max}(\mathbf{weight_k} * \mathbf{R}, \mathbf{minShare_k}), \mathbf{demand_k}\}$$

   那么相应的 queue 或者 job 的 FairShare 的数值为：$\mathbf{Min\{max(weight*R, minShare), demand\}}$。




我们注意到 weight 在计算过程中非常重要，那么这个值是啥呢？

- 对于 queue，是在 fair-scheduler.xml 配置的。没有配置就是 1.0
- 对于 job，就要复杂点：考虑了大作业及时间因素，当然用户也可以自行设计相关的 WeightAdjuster 调整类。
  - 运行状态为 false，返回 1.0
  - 如果配置了 sizeBasedWeight=true，则 weight=$\sqrt[10]{demand}/\sqrt[2]{2}$，这会让大作业占优势。
  - Weight=weight*Priority（这里优先级都为 1.0）
  - 用户可以自己设置相关的 WeightAdjuster 权重调整的类，默认的为 NewAppWeightBooster，如果开始时间已经过去了 mapred.newjobweightbooster.duration(默认 3 分钟)，则 Weight= Weight* factor(默认是 3)。

## 3.2.4、FairScheduler 抢占资源

计算 queue 及 job 的 FairShare 就是为了抢占，否则 FairShare 就只要在页面看看了。抢占是在 UpdateThread 中最后一步执行，条件是抢占开关开启：yarn.scheduler.fair.preemption 设置为 true，抢占的频率为 15s。每个队列如果想抢占，还需要设置几个时间的变量：fairSharePreemptionTimeout 全局就一个，minSharePreemptionTimeouts 可以为每个 queue 配置一个，如果没有配置默认用 defaultMinSharePreemptionTimeout。在 UpdateThread 线程在当前使用资源小于 MinShare 时会更新 lastTimeAtMinShare，当小于 FairShare 的一半时会更新 lastTimeAtHalfFairShare。计算的步骤如下：

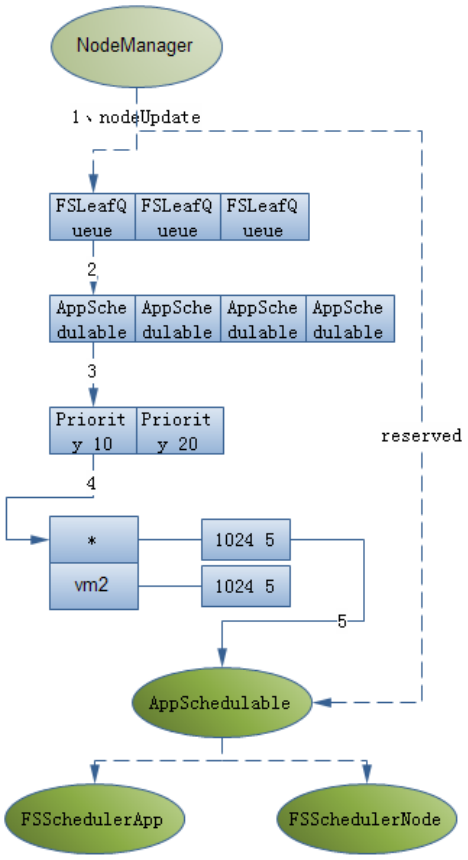
1. 计算需要抢占的资源总量，当 queue 满足 now-lastTimeAtMinShare>minSharePreemptionTimeouts 或者 now-lastTimeAtHalfFairShare>fairSharePreemptionTimeout 时，如果其中一个不满足，则相应的表达为 0。

$$\mathbf{TotalResource} = \sum_{k=1}^{n} \begin{matrix} \mathbf{max}\{\mathbf{max}(0, \mathbf{min}(\mathbf{minshare_k}, \mathbf{demand_k}) - \mathbf{usage_k}, \\ \mathbf{max}(0, \mathbf{min}(\mathbf{fairshare_k}, \mathbf{demand_k}) - \mathbf{usage_k}\} \end{matrix}$$

2. 选出超支的 app，强制 kill 掉优先级低的任务。

## 3.2.5、FairScheduler container 分配

在每次 NM 的心跳触发时，会触发实际的资源的分配。这里主要有一些基本的事情：尽量做到本地化、一次心跳可以分配多个资源、支持在某个节点上面预订一个 container 等。一个基本的流程如右图所示。




如果在这个节点上面已经有预订的资源，则直接跳到 AppSchedulable 分配资源，否则需要选择一个 AppSchedulable 及 ResourceRequest。

- AppSchedulable 选择过程：
  1. 对 FSLeafQueue 排序，用 FAIR，基本是按照 minShare(如果 minShare 小于 demand 则取 demand)是否满足，都不满足再按照 minShare 缺口比重、都满足则按照 ResourceUsage/权重比重，如果还相同则提交时间及 queueName 排序，选取排在第一个的 FSLeafQueue。
  2. 对 FSLeafQueue 内的 AppSchedulable 排序，可以用 FIFO 或者 FAIR。FIFO 是按照优先级、提交时间、作业名称。再选取一个 AppSchedulable。
- ResourceRequest 选择过程：

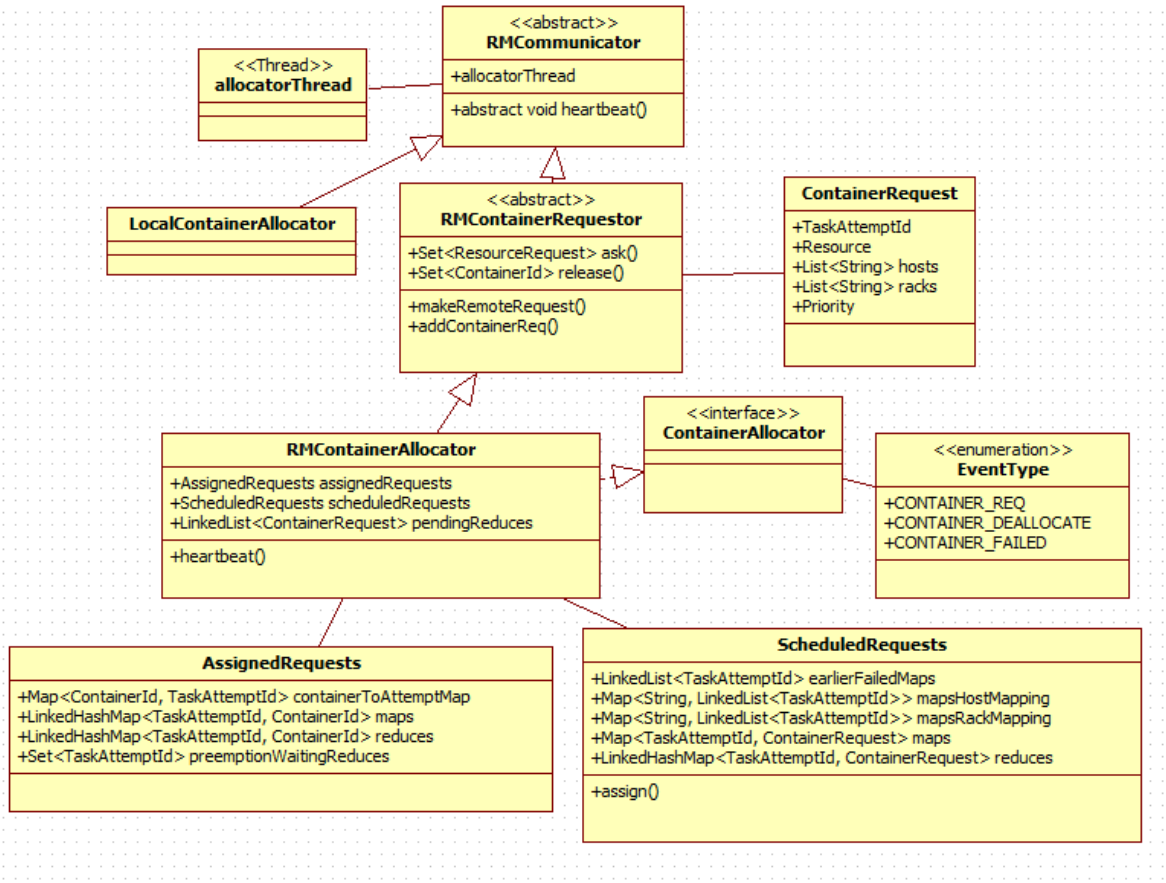
3. 选择所有请求中，按照 priority 排序。<0、5、10、20，分别对应：appMaster、PRIORITY_FAST_FAIL_MAP、PRIORITY_REDUCE、PRIORITY_MAP>。
4. 选取满足条件的 priority。首先获取 AllowedNodeType（在 NODE_LOCAL 重试的次数大于 numNodes * threshold 则降级为 RACK_LOCAL，RACK_LOCAL 重试的次数大于 numNodes * threshold 则降级为 OFF_SWITCH，首次默认为 NODE_LOCAL）
   按照 NODE_LOCAL、RACK_LOCAL、OFF_SWITCH 顺序来选取 ResourceRequest（如果有此节点的 ResourceRequest；如果有此机架上的 ResourceRequest 及 AllowedNodeType 不为 NODE_LOCAL；如果有 ResourceRequest 及 AllowedNodeType 为 OFF_SWITCH）
5. 选择一个 ResourceRequest 后，再分配 container，如果没有资源，则在此节点上预订一个 container。

## 3.3、MRAppMaster 分配器

这里主要的组件是 RMCommunicator 组件，RMCommunicator 是一个服务，启动的时候会初始化一些资源。主要的作用是请求 RM 的资源，分配 job 内的 task 流程。

### 3.3.1、代码分析

类图如下所示：




有一个明显的继承体系，从上到下的功能是：

- RMCommunicator:连接到 RM，并发送心跳到 RM。心跳的内容则由其子类提供。
- LocalContainerAllocator:本地化的 container 分配器，不会请求 container。处理事件 CONTAINER_REQ，处理的过程也十分简单就是返回一个与 AM 一模一样的 container。
- RMContainerRequestor:保持一个数据结构，如：需要请求的 container，需要释放的 container，黑名单等，并提供对这个对象操作的方法。
- RMContainerAllocator:实际的分配器，处理三个事件<CONTAINER_REQ、CONTAINER_DEALLOCATE、CONTAINER_FAILED>。分配 container 给 task，维护 map、reduce 的生命周期。

### 3.3.2、任务周期管理及资源分配

Map 的生命周期为：scheduled->assigned->completed。Reduce 的生命周期为：pending->scheduled->assigned->completed。在任务到达的时候，就直接通过心跳向 RM 请求所有的 map 资源，当 map 完成到 0.05 的比例的时候才容许启动 reduce 任务（要按照资源的要求，不一定就能启动 reduce，大概启动的 reduce 个数为 map 完成的比率乘以当时能申请的所有资源除以每个 reduce 所需的资源）。如果在运行过程中，出现 map 失败等，可能出现抢占 reduce 资源的情况。

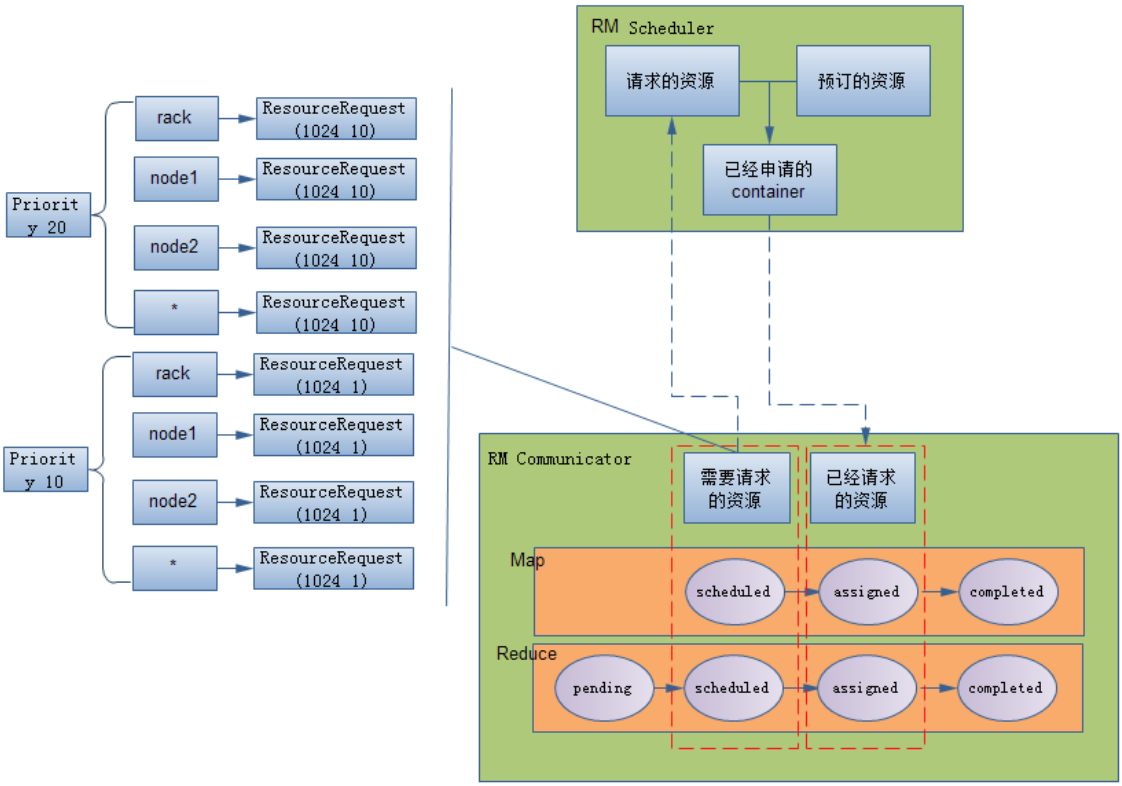
参考下图，简单讲下处理的流程：

- 当一个 TaskAttemptImpl 的 container 请求事件 CONTAINER_REQ 到达 RMContainerAllocator 后，如果是 map，则放入 scheduled 中，并放入到远程资源请求列表 remoteRequestsTable 中及本次心跳请求的 ask 变量中。如果是 reduce 则放到 reduce 的 pending 列表中。
- RMContainerAllocator 心跳到 RM，把 ask 带给 RM，返回获得已经获取的资源（container），并把 ask 清空。
- 分配获取的资源（container）

- 检查资源是否需要（map 已经完成就不需要优先级为 20 的资源了）
- 分配给 task container，把在 scheduled 状态的 task 放到 assigned 中。并再次初始化下次心跳请求的 ask。这里修改 remoteRequests 变量。

- 检查本心跳范围内是否有 map 完成，如果有完成，则
  - 如果当前 scheduled 状态的 map 还存在且给 map 的资源量少于一个 map 的资源量，则会抢占正在运行的 reduce。优先抢占进度相对较慢的。
  - 考虑调度 reduce，从 pending 到 scheduled。调度的条件是 map 已经完成 5%，再满足以下条件：
    - 当前的所有能申请的资源量*min(50%,map 完成量)+map 空闲量 大于单个 reduce 需求量。

另外，申请资源的数据结构如下图所示：资源请求机器的、机架及任意的。RM scheduler 在处理的时候，只要分配了资源就会把相应级别的的请求资源清理掉。



## 3.3、shuffle

待续

## 3.5、NM 的资源下载

待续