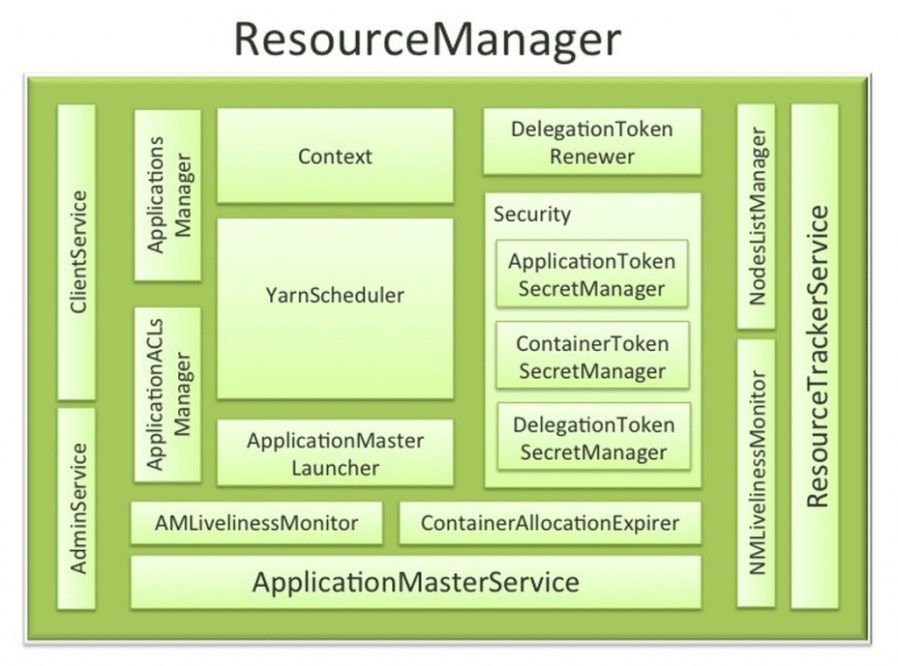
YARN ResourceManager调度器分析

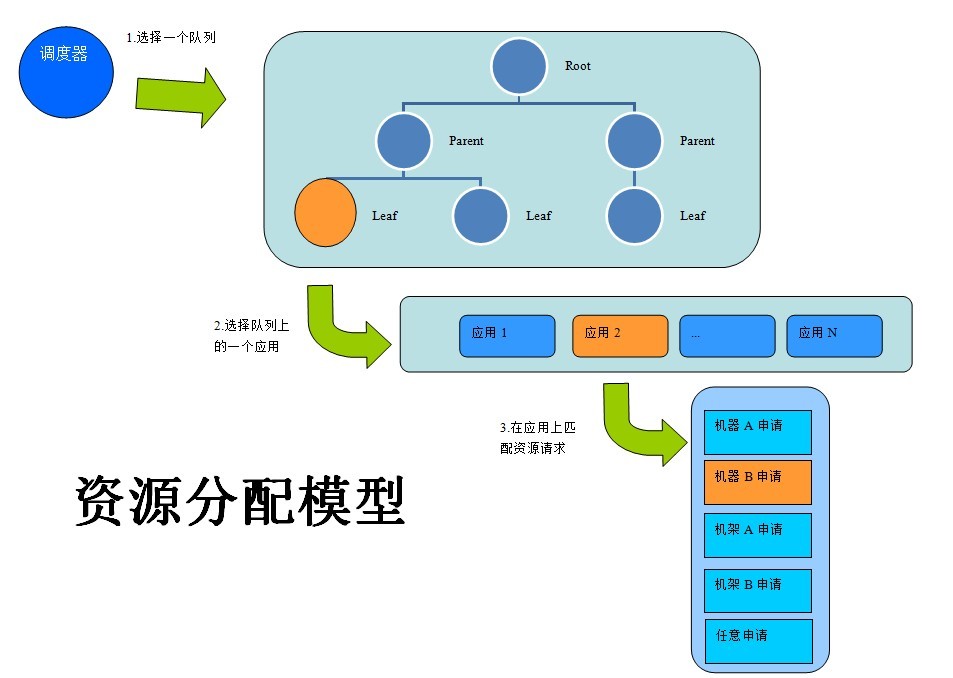
ResourceManager是YARN中的中控模块，负责统一规划资源的使用。NodeManager是资源结点模块，负责启动和管理Container。ApplicationMaster，每一个应用都会启动一个AM，负责向RM申请资源，请求NM启动Container，并告诉Container做什么事情。

下图是RM的总体架构：



RM是YARN资源控制框架的中心模块，负责集群中所有资源的统一管理和分配，它接受来自NM的汇报，建立AM，并将资源派送给AM。

在Hadoop中，调度器在内存中会维护一个队列，用户可以向一个或者多个队列提交应用。每次NM心跳的时候，调度器会根据一定的规则选择一个队列，再在队列上选择一个应用，尝试在这个应用上分配资源。



总体来说，3种调度器就是在回答如何选择一个队列，在一个队列上如何选择一个应用的问题。

# 1、调度器分类

当前YARN中有三种调度器，FifoScheduler，CapacityScheduler和FairScheduler：

| 调度器 | FifoScheduler | CapacityScheduler | FairScheduler |
| --- | --- | --- | --- |
| 设计目的 | 最简单的调度器，易于理解和上手 | 多用户的情况下，最大化集群的吞吐和利用率 | 多用户的情况下，强调用户公平地贡献资源 |
| 队列组织方式 | 单队列 | 树状组织队列。无论父队列还是子队列都会有资源参数限制，子队列的资源限制计算是基于父队列的。应用提交到叶子队列。 | 树状组织队列。但是父队列和子队列没有参数继承关系。父队列的资源限制对子队列没有影响。应用提交到叶子队列。 |
| 资源限制 | 无 | 父子队列之间有容量关系。每个队列限制了资源使用量，全局最大资源使用量，最大活跃应用数量等。 | 每个叶子队列有最小共享量，最大资源量和最大活跃应用数量。用户有最大活跃应用数量的全局配置。 |
| 队列ACL限制 | 可以限制应用提交权限 | 可以限制应用提交权限和队列开关权限，父子队列间的ACL会继承。 | 可以限制应用提交权限，父子队列间的ACL会继承。但是由于支持客户端动态创建队列，需要限制默认队列的应用数量。目前，还看不到关闭动态创建队列的选项。 |
| 队列排序算法 | 无 | 按照队列的资源使用量最小的优先 | 根据公平排序算法排序 |
| 应用选择算法 | 先进先出 | 先进先出 | 先进先出或者公平排序算法 |
| 本地优先分配 | 支持 | 支持 | 支持 |
| 延迟调度 | 不支持 | 不支持 | 支持 |
| 资源抢占 | 不支持 | 不支持 | 支持，看到代码中也有实现。但是，由于本特性还在开发阶段，本文没有真实试验。 |

FifoScheduler: 最简单的调度器，按照先进先出的方式处理应用，只有一个队列可提交应用，所有用户提交到这个队列，可以针对这个队列设置ACL，没有应用优先级可以配置。

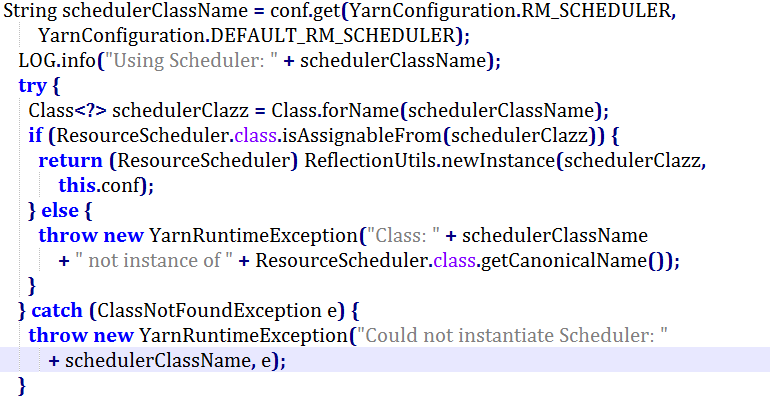
CapacityScheduler: 可以看做是FifoScheduler的多队列版本，每个队列可以限制资源使用量。但是队列间的资源分配以使用量作为排列依据，使得容量最小的队列有竞争优势。集群整体吞吐较大。延迟调度机制使得应用可以放弃，跨机器或者机架的调度机会，争取本地调度。

FairScheduler:多队列，多用户共享资源。特有的客户端创建队列的特性，使得权限控制不太完美。根据队列设定的最小共享量或者权重等参数，按比例共享资源。延迟调度机制跟CapacityScheduler的目的类似，但是实现方式稍有不同。资源抢占特性，是指调度器能够依据公平资源共享算法，计算每个队列应得的资源，将超额资源的队列部分容器释放掉的特性。

# 2、YARN源码分析

## 2.1 基本架构

资源调度器是可插拔式的，可以根据需要实现自己的调度器。ResourceMananger在初始化时会根据用户的配置创建一个资源调度器对象，相关代码如下：



管理通过参数yarn.resourcemananger.scheduler.class设置资源调度器的主类，默认是FariScheduler。

所有的调度器均应该实现接口ResourceScheduler，组织关系如下图：



YARN的资源管理器实际上是一个事件处理器，处理来自外部的6种SchedulerEvent

-Type类型的事件，并根据时间的具体含义进行相应的处理，如下图所示：



ResourceManager收到NodeManager通过心跳机制汇报的信息后，会触发一个NODE\_UPDATE事件，该事件会触发资源调度器最核心的资源分配机制。

## 2.2 资源调度模型

YARN采用了双层资源调度模型：在第一层中，ResourceManager中的资源调度器将资源分配给各个ApplicationMaster，第二层中，ApplicationMaster再进一步将资源分配给它内部的各个任务。

YARN的资源分配过程是异步的，资源调度器将资源分配给一个应用程序后，不会立刻Push给对应的ApplicationMaster，而是暂时放到一个缓冲区中，等待ApplicationMaster通过周期性的心跳主动来取。

步骤如下图所示：



其中资源调度器关注的是步骤4中采用的策略，即如何将节点上空闲的资源分配给各应用程序，至于步骤7中的策略，则完全由应用程序自己决定。

以FairScheduler为例，assignContainers相关类的关系图如下：



其代码执行流程如下所示：



调用关系，代码分析：

分为两个部分，部分1为Client提交到ResourceManager，部分2为NodeManager向RM汇报心跳信息，然后update状态，分配container

部分1：



部分2：



containers的运行，调用关系的代码分析

在上述的步骤1中，RM由ResourceScheduler返回AllocatedResponse，其中有分配的container信息，ApplicationMaster根据Response，与NM通信并启动Container，流程如下图所示：



最后会调用ContainerExecutor在NodeManager中启动一个Container

## 2.3 资源隔离

上节中ContainerExecutor.launchContainer，启动一个Container，在launchContainer中实现资源隔离，但是ContainerExecutor的子类中只有LinuxContainerExecutor具有资源隔离的实现，DefaultContainerExecutor是不支持的。

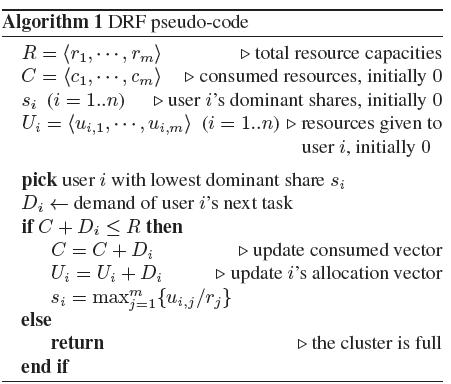
核心代码：



# 3、算法分析

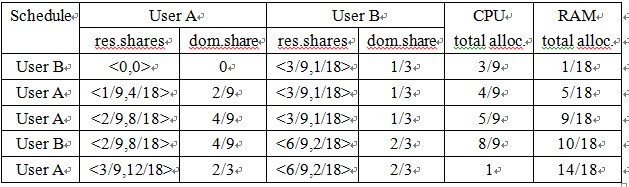
## 3.1 算法伪代码

YARN资源调度器采用了主资源公平调度算法（DRF），该算法扩展了最大最小算法，使其能够支持多维度资源的调度。在DRF算法中，将所需要份额最大的资源成为主资源，而DRF的基本思想则是将最大最小算法应用于主资源中，进而将多维资源调度问题转化为单资源调度的问题，即DRF总是最大化所有主资源中最小，算法伪代码如下：



举例说明，假设系统中共有9 CPUs 和18 GB RAM，有两个user（framework）分别运行了两种任务，分别需要的资源量为<1 CPU, 4 GB> 和 <3 CPUs, 1 GB>。对于用户A，每个task要消耗总CPU的1/9和总内存的2/9，因而A的支配性资源为内存；对于用户B，每个task要消耗总CPU的1/3和总内存的1/18，因而B的支配性资源为CPU。DRF将均衡所有用户的支配性资源，即：A获取的资源量为：<3 CPUs，12 GB>，可运行3个task；而B获取的资源量为<6 CPUs, 2GB>，可运行2个task，这样分配，每个用户获取了相同比例的支配性资源，即：A获取了2/3的RAMs，B获取了2/3的CPUs。

DRF算法的一个可能的调度序列如下图所示：

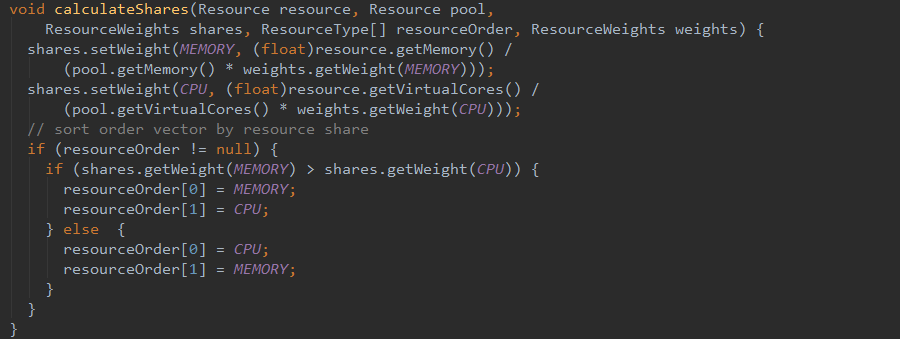


## 3.2 源码分析

算法实现的核心类是SchedulingPolicy，FairScheduler使用的是其实现类DominantResour

ceFairnessPolicy，下面分析核心方法：

1.calculateShares，对Resource Share进行计算和排序，ResourceOrder存储了多种资源中优先被考虑的因素



2.DominantResourceFairnessComparator#compare(Schedulable s1,Schedulable s2)，比较两种调度，优先分配哪个任务

1）调用calaculateShares，分别计算s1和s2的资源(cpu和memory)的share，根据share设置s1和s2的ResourceOrder

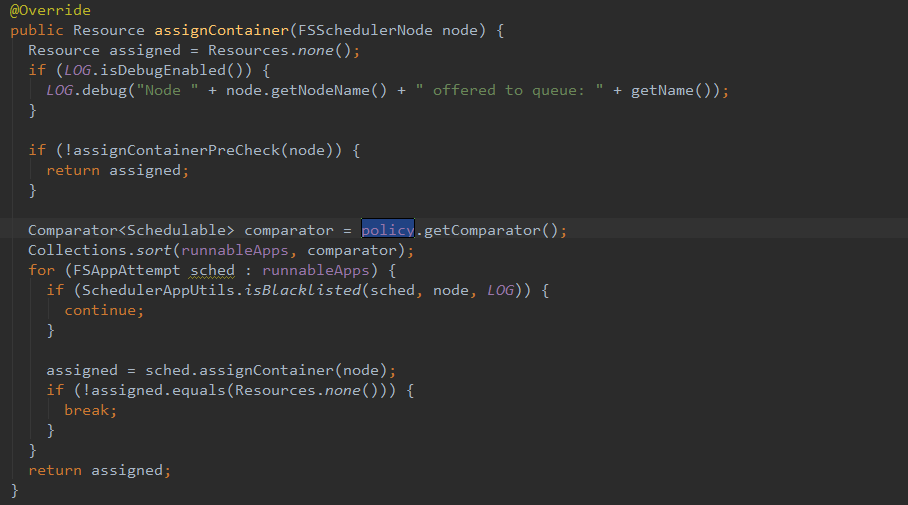
2）根据ResourceOrder选取更急迫的资源

3）对比两者s1和s2的Order，根据算法调度s1或者s2

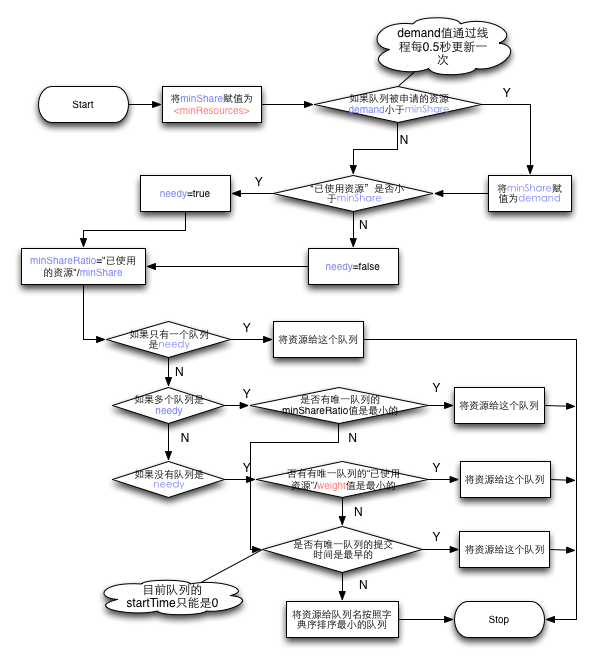
## 3.3 算法执行流程

以FairScheduler调度为例，

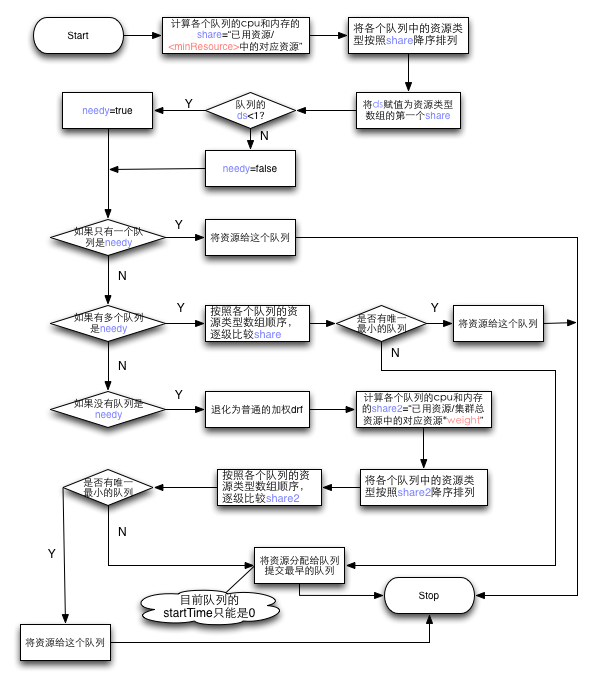
对Policy的使用，在代码FSLeafQueue和FSParentQueue的assighContainers中，源码如下：



流程如下图：



demand值的更新，在FairScheduler#run中调用update来完成；minShare的赋值ComputeFairShare#getFairShareIfFixed中方法来设置。



最终的调用是在FSParentQueue#preemptContainer -> FSLeafQueue#preemptContainer会返回分配的RMContainer，其中preemtContainer的源码如下：

