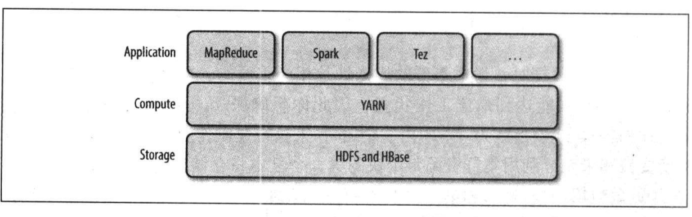
# 4.1 YARN运行机制(重点)

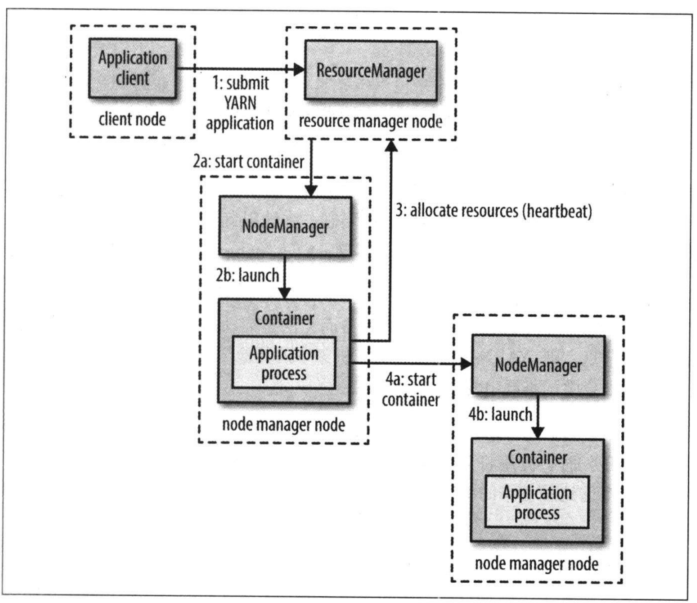
1.Apache YARN(Yet Another Resource Negotiator的缩写)是Hadoop的**集群资源管理系统**。最初是为了改善MapReduce的实现，但它具有足够的通用性，同样可以**支持其他的分布式计算模式**。

2.**YARN提供请求和使用集群资源的API**，但这些API很少直接用于用户代码。相反，用户代码中用的是分布式计算框架提供的更高层API，这些API建立在YARN上且向用户隐藏了资源管理细节。下图对此进行了描述，一些**分布式计算框架**(MapReduce, Spark等等)作为**YARN应用运行在集群计算层(YARN)和集群存储层(HDFS和HBase)上**。



3.YARN通过**两类长期运行的守护进程提供自己的核心服务**：

**管理集群上资源使用的资源管理器**(resource manager)、运行在集群中所有节点上且能够启动和监控**容器**(container)的**节点管理器**(node manager)。容器用于执行特定应用程序的进程，每个容器都有**资源限制**(内存、CPU等)。一个容器可以是一个Unix进程，也可以是一个Linux cgroup，取决于YARN的配置。下图描述了YARN是如何运行一个应用的。



4.为了在YARN上运行一个应用，首先客户端联系**资源管理器**，要求它运行一个**application master进程**(步骤1)。然后，**资源管理器找到一个能够在容器中启动application master的节点管理器**(步骤2a和2b)。准确来说，application master一旦运行起来后能做些什么依赖于应用本身。**有可能是在所处的容器中简单地运行一个计算，并将结果返回给客户端；或是向资源管理器请求更多的容器**(步骤3)，以用于运行一个分布式计算(步骤4a和4b)。后者是MapReduce YARN应用所做的事情。

5.YARN本身**不会为应用的各部分**(客户端、master和进程)**彼此间通信提供任何手段**。大多数重要的**YARN应用使用某种形式的远程通信机制**(例如Hadoop的RPC层)来向客户端传递状态更新和返回结果，但是这些通信机制是专属于各应用的。

## 4.1.1 资源请求

1.**YARN有一个灵活的资源请求模型**。当请求多个容器时，可以指定每个容器需要的计算机资源数量(内存和CPU)，还可以指定对**容器的本地限制要求**。

2.本地化对于**确保分布式数据算法高效使用集群带宽非常重要**，因此，**YARN允许一个应用为所申请的容器指定本地限制**。本地限制可用于**申请位于指定节点或机架，或集群中任何位置**(机架外)的容器。

3.有时**本地限制无法被满足**，这种情况下**要么不分配资源**，或者**可选择放松限制**。例如，一个节点由于已经运行了别的容器而无法再启动新的容器，这时如果有应用请求该节点，则YARN将尝试在**同一机架中的其他节点上启动一个容器**，如果还不行，则会尝试**集群中的任何一个节点**。

4.通常情况下，当启动一个容器用于处理**HDFS数据块**(为了在MapReduce中运行一个map任务)时，应用将会向这样的节点申请容器：**存储该数据块三个复本的节点**，或是**存储这些复本的机架中的一个节点**。如果都申请失败，则**申请集群中的任意节点**。

5.YARN应用**可以在运行中的任意时刻提出资源申请**。例如，可以在最开始提出所有的请求，或者为了满足不断变化的应用需要，采取更为动态的方式在需要更多资源时提出请求。

## 4.1.2 应用生命期

1.YARN**应用生命期差异性很大**：有**几秒的短期应用**，也有连续**运行几天甚至几个月的长期应用**。与其关注应用运行多长时间，不如按照**应用到用户运行的作业之间的映射关系对应用进行分类更有意义**。最简单的模型是一个用户作业对应一个应用，这也是MapReduce采取的方式。

2.第二种模型是，作业的每个**工作流或每个用户会话**(可能并无关联性)**对应一个应用**。这种方法要比第一种情况效率更高，因为**容器可以在作业之间重用**，并且有可能缓存作业之间的中间数据。Spark采取的是这种模型。

# 4.2 YARN中的调度(重点)

## 4.2.1 YARN中的调度选项

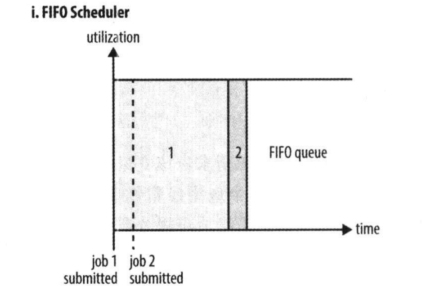
1.YARN中有三种调度器可用：**FIFO调度器**(FIFO Scheduler)，**容量调度器**(Capacity Scheduler)和**公平调度器**(Fair Scheduler)。

2.FIFO调度器将**应用放置在一个队列中**，然后**按照提交的顺序(先进先出)运行应用**。首先为队列中**第一个应用的请求分配资源**，**第一个应用的请求被满足后再一次为队列中下一个应用服务**。

3.**FIFO调度器的优点是，简单易懂，不需要任何配置，但是不适合共享集群**。**大的应用会占用集群中的所有资源，所以每个应用必须等待直到轮到自己运行**。

4.在一个**共享集群**中，更适合使用**容量调度器或公平调度器**。这**两种调度器都允许长时间运行的作业能及时完成**，同时也允许**正在进行较小临时查询的用户能够在合理时间内得到返回结果**。

5.下图描述了调度器之间的差异性，从图中看出，当使用FIFO调度器时，小作业一直被阻塞，直至大作业完成。

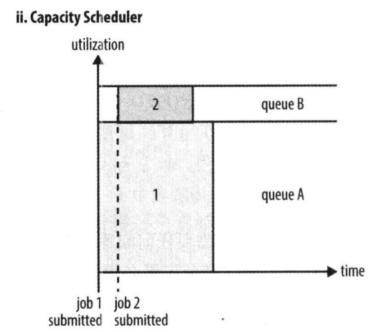


6.使用容量调度器时，**一个独立的专门队列保证小作业一提交就可以启动**，由于队列容量是为那个队列中的作业所保留的，**因此这种策略是以整个集群的利用率为代价的**。**意味着与FIFO相比，大作业执行时间更长**。

容量调度器允许多个组织共享一个Hadoop集群，**每个组织可用分配到全部集群资源的一部分**。**每个组织被配置一个专门的队列，每个队列被配置为可以使用一定的集群资源**。**队列可以进一步按层次划**分，这样每个组织内的不同用户能够共享该组织队列所分配的资源。在一个队列内，**使用FIFO调度策略对应用进行调度**。

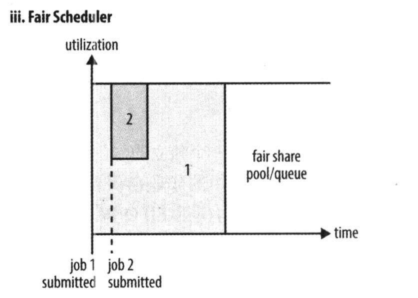
正如下图所示，单个作业使用的资源不会超过队列容量。然而，如果队列中有多个作业，并且队列资源不够用，**如果仍有可用的空闲资源**，那么**容量调度器可能会将空余的资源分配给队列中的作业**，哪怕会超出队列容量。这被称为**弹性队列**。

正常操作时，**容量调度器不会通过强行中止来抢占容器**。因此，如果一个队列一开始资源够用，然后随着需求增长，资源开始短缺时，**那么这个队列就只能等着其他队列释放容器资源**。**缓解这种情况的方法是，为队列设置一个最大容量限制**，这样这个队列就不会过多侵占其他队列的容量了。当然这也是牺牲队列弹性为代价的，因此需要在不断尝试和失败中找到一个折中。

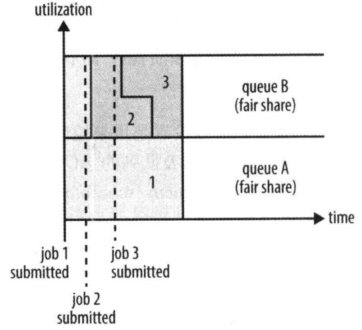


7.使用公平调度器时，**不需要预留一定量的资源，因为调度器会在所有运行的作业之间动态平衡资源**。第一个(大)作业启动时，**它也是唯一运行的作业**，因而**获得集群中所有的资源**。当第**二个(小)作业启动时，它被分配到集群的一半资源**，这样**每个作业都能公平共享资源**。

从**第二个作业的启动到获得公平共享资源之间会有时间滞后**，因为**它必须等待第一个作业使用的容器用完并释放资源**。当**小作业结束且不再申请资源后，大作业将回去再次使用全部的集群资源**。最终的效果是：**既得到了较高的集群利用率，又能保证小作业能及时完成**。



接下来解释资源是如何在队列之间公平共享的。想象两个用户A和B，分别拥有自己的队列(参见图4-4)。A 启动一个作业，**在B没有需求时A会分配到全部可用资源**；当A 的作业仍在运行时B启动一个作业，一段时间后，按照我们先前看到的方式，**每个作业都用到了一半的集群资源**。这时，**如果B启动第二个作业且其他作业仍在运行，那么第二个作业将和B的其他作业(这里是第一个)共享资源**，**因此B的每个作业将占用四分之一的集群资源**，而A仍继续占用一半的集群资源。最终的结果就是资源在用户之间实现了公平共享。



为了使**作业从提交到执行所需的时间可预测**，**公平调度器支持“抢占”功能**。**所谓抢占，就是允许调度器终止那些占用资源超过了其公平共享份额的队列的容器**，这些**容器资源释放后可以分配给资源数量低于应得份额的队列**。注意，抢占会降低整个集群的效率，因为被终止的容器要重新运行。

## 4.2.2 延迟调度

1.所有的**YARN调度器都试图以本地请求为重**。在一个繁忙的集群上，如果一个应用请求某个节点，那么**极有可能此时有其他容器正在该节点上运行**。显而易见的处理是，**立刻放宽本地性需求**，在**同一机架中分配一个容器**。

2.通过实践发现，此时如果等待一小段时间，能够增加在所请求节点上分配到一个容器的机会，**从而可以提高集群的效率**。这个特性称为**延迟调度**。容量调度器和公平调度器都支持延迟调度。

3.当使用延迟调度时，调度器不会简单的使用它收到的第一个调度机会，而是等待设定的最大数目的调度机会发生，然后才**放松本地性限制并接收下一个调度机会**。

## 4.2.3 主导资源公平性

1.YARN调度器解决多类型资源问题的思路是，观察每个用户的主导资源，并将其作为对集群资源使用的一个度量。这个方法称为“主导资源公平性”。

2.想象一个总共有100个CPU和10TB的集群。应用A请求的每份容器资源为2个CPU和300GB内存，应用B请求的每份容器资源为6个CPU和 100GB内存。A请求的资源在集群资源中占比分别为2%和3%，由于内存占比(3%)大于CPU占比(2%)，所以内存是A的主导资源。B请求的资源在集群资源中占比分别为6%和1%，所以CPU是B的主导资源。由于**B申请的资源是A的两倍(6% vs3%)，所以在公平调度下，B将分到一半的容器数**。

## 4.2.4 YARN的工作流程(Job的提交流程，重点)

1.MapReduce程序创建成功后，需要提交到HDFS上运行，MapReduce程序提交到客户端所在的节点。

2.YarnRunner向ResourceManager申请提交一个应用。

3.ResourceManager返回应用程序需要提交的资源路径和应用ID(ApplicationID)。

4.YarnRunner提交资源到HDFS上，包含job的配置文件、分片信息、MapReduce作业代码文件(xxx.jar)，资源提交完毕后，向ResourceManager申请一个运行MRApplicationMaster(协调运行作业中的任务)，一个应用对应一个ApplicationMaster。

5.ResourceManager把提交的Job初始化成一个Task，放入Yarn的调度队列中去。

6.当执行此Task时，其中一台NodeManager领取此任务，然后在此NodeManager上创建一个Container并创建一个MRApplicationMaster，然后下载HDFS上的计算资源到此NodeManager上。

7.MRApplicationMaster向ResourceManager申请运行MapTask的资源，同时ResourceManager向另外两台NodeManager分配执行MapTask的任务，这两台NodeManager分别创建Container，然后MRApplicationMaster将启动脚本发送给这两台NodeManager，启动MapTask任务。

8.当MapTask执行完毕之后，MRApplicationMaster向ResourceManager申请执行ReduceTask的资源，ResourceManager分配资源创建Container，Container下载MapReduce作业代码文件到此Container中，MRApplicationMaster启动此任务。

9.ReduceTask获取MapTask分区数据，进行计算。

10.当ReduceTask运行完毕之后，通知MRApplicationMaster，注销ReduceTask的资源，MRApplicationMaster向ResourceManager申请注销自己。