YARN 架构学习总结

Sep 1, 2018 | 技术 | 6,993 字 | 689 Times

关于 Hadoop 的介绍，这里就不再多说，可以简答来说 Hadoop 的出现真正让更多的互联网公司开始有能力解决大数据场景下的问题，其中的 HDFS 和 YARN 已经成为大数据场景下存储和资源调度的统一解决方案（MR 现在正在被 Spark 所取代，Spark 在计算这块的地位也开始受到其他框架的冲击，流计算上有 Flink，AI 上有 Tensorflow，两面夹击，但是 Spark 的生态建设得很好，其他框架想要在生产环境立马取代还有很长的路要走）。本片文章就是关于 YARN 框架学习的简单总结，目的是希望自己能对分布式调度这块有更深入的了解，当然也希望也这篇文章能够对初学者有所帮助，文章的主要内容来自 《Hadoop 技术内幕：深入解析 YARN 架构设计与实现原理》 和 《大数据日知录：架构与算法》。

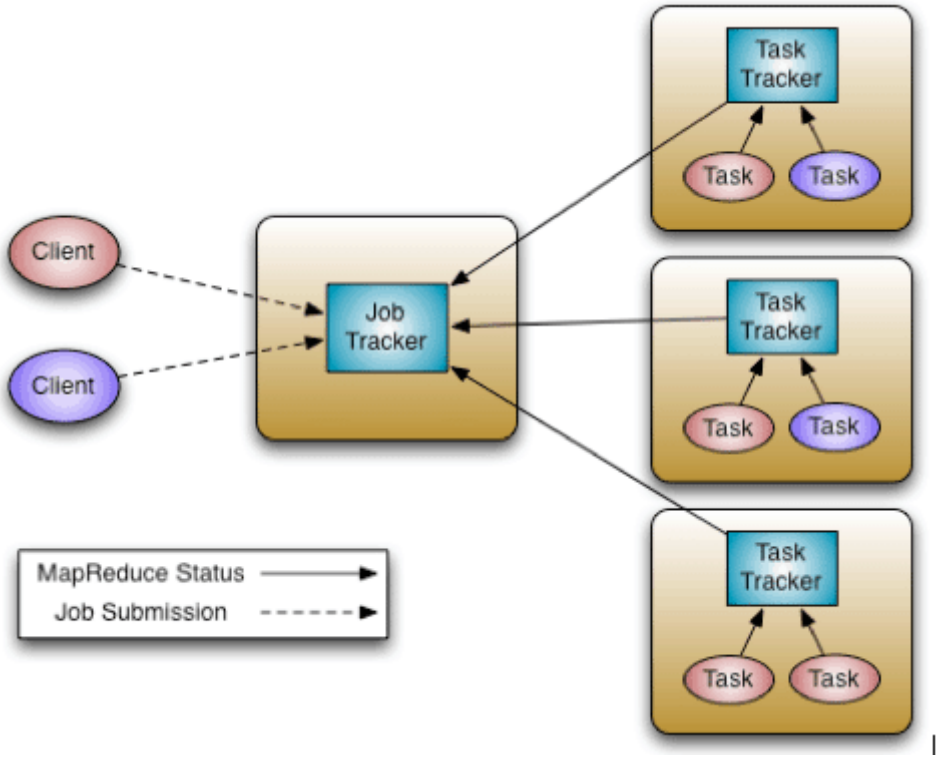
文章目录

Yarn 背景

关于 YARN 出现的背景，还是得从 Hadoop1.0 说起，在 Hadoop1.0 中，MR 作业的调度还是有两个重要的组件：JobTracker 和 TaskTracker，其基础的架构如下图所示，从下图中可以大概看出原 MR 作业启动流程：

1. 首先用户程序 (Client) 提交了一个 job，job 的信息会发送到 JobTracker 中，JobTracker 是 Map-Reduce 框架的中心，它需要与集群中的机器定时通信 (心跳：heartbeat)，需要管理哪些程序应该跑在哪些机器上，需要管理所有 job 失败、重启等操作；

2. TaskTracker 是 Map-Reduce 集群中每台机器都有的一个组件，它做的事情主要是监视自己所在机器的资源使用情况；
3. TaskTracker 同时监视当前机器的 tasks 运行状况。TaskTracker 需要把这些信息通过 heartbeat 发送给 JobTracker，JobTracker 会搜集这些信息以便处理新提交的 job，来决定其应该分配运行在哪些机器上。



Hadoop 1.0 调度的架构图

可以看出原来的调度框架实现非常简答明了，在 Hadoop 推出的最初几年，也获得业界的认可，但是随着集群规模的增大，很多的弊端开始显露出来，主要有以下几点：

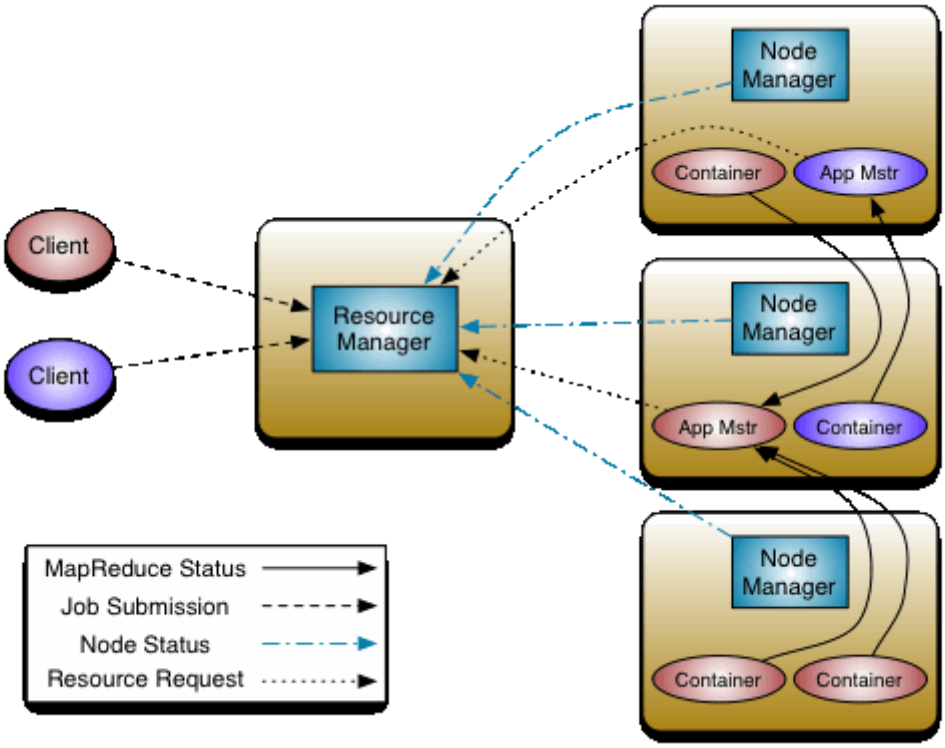
1. JobTracker 是 Map-Reduce 的集中处理点，存在单点故障；
2. JobTracker 赋予的功能太多，导致负载过重，1.0 时未将资源管理与作业控制（包括：作业监控、容错等）分开，导致负载重而且无法支撑更多的计算框架，当集群的作业非常多时，会有很大的内存开销，潜在来说，也增加了 JobTracker fail 的风险，这也是业界普遍总结出 Hadoop1.0 的 Map-Reduce 只能支持 4000 节点主机上限的原因；
3. 在 TaskTracker 端，以 map/reduce task 的数目作为资源的表示过于简单，没有考虑到 cpu/内存的占用情况，如果两个大内存消耗的 task 被调度到了一个节点上，很容易出现 OOM；
4. 在 TaskTracker 端，把资源强制划分为 map task slot 和 reduce task slot, 如果当系统中只有 map task 或者只有 reduce task 的时候，会造成资源的浪费，也就是前面提过的集群资源利用的问题。

Hadoop 2.0 中下一代 MR 框架的基本设计思想就是将 JobTracker 的两个主要功能，资源管理和作业控制（包括作业监控、容错等），分拆成两个独立的进程。资源管理与具体的应用程序无关，它负责

整个集群的资源（内存、CPU、磁盘等）管理，而作业控制进程则是直接与应用程序相关的模块，且每个作业控制进程只负责管理一个作业，这样就是 YARN 诞生的背景，它是在 MapReduce 框架上衍生出的一个资源统一的管理平台。

Yarn 架构

YARN 的全称是 Yet Another Resource Negotiator，YARN 整体上是 Master/Slave 结构，在整个框架中，ResourceManager 为 Master，NodeManager 为 Slave，如下图所示：



YARN 基本架构

ResourceManager（RM）

RM 是一个全局的资源管理器，负责整个系统的资源管理和分配，它主要有两个组件构成：

1. 调度器：Scheduler；
2. 应用程序管理器：Applications Manager，ASM。

调度器

调度器根据容量、☒队列等限制条件（如某个队列分配一定的资源，最多执行一定数量的作业等），将系统中的资源分配给各个正在运行的应用程序。☒要注意的是，该调度器是一个纯调度器，它不再从事任何与应用程序有关的工作，比如不负责重新启动（因应用程序失败或者硬件故障导致的失败），这些均交由应用程序相关的 ApplicationMaster 完成。调度器仅根据各个应用程序的资源需求进行资源分

配，而资源分配单位用一个抽象概念 资源容器(Resource Container，也即 Container)，Container 是一个动态资源分配单位，它将内存、CPU、磁盘、网络等资源封装在一起，从而限定每个任务使用的资源量。此外，该调度器是一个可插拔的组件，用户可根据自己的需求设计新的调度器，YARN 提供了多种直接可用的调度器，比如 Fair Scheduler 和 Capacity Schedule 等。

应用程序管理器

应用程序管理器负责管理整个系统中所有应用程序，包括应用程序提交、与调度器协商资源以 AM、监控 AM 运行状态并在失败时重新启动它等。

NodeManager（NM）

NM 是每个节点上运行的资源和任务管理器，一方面，它会定时向 RM 汇报本节点上的资源使用情况和各个 Container 的运行状态；另一方面，它接收并处理来自 AM 的 Container 启动/停止等各种请求。

ApplicationMaster（AM）

提交的每个作业都会包含一个 AM，主要功能包括：

1. 与 RM 协商以获取资源（用 container 表示）；
2. 将得到的任务进一步分配给内部的任务；
3. 与 NM 通信以启动/停止任务；
4. 监控所有任务的运行状态，当任务有失败时，重新为任务申请资源并重启任务。

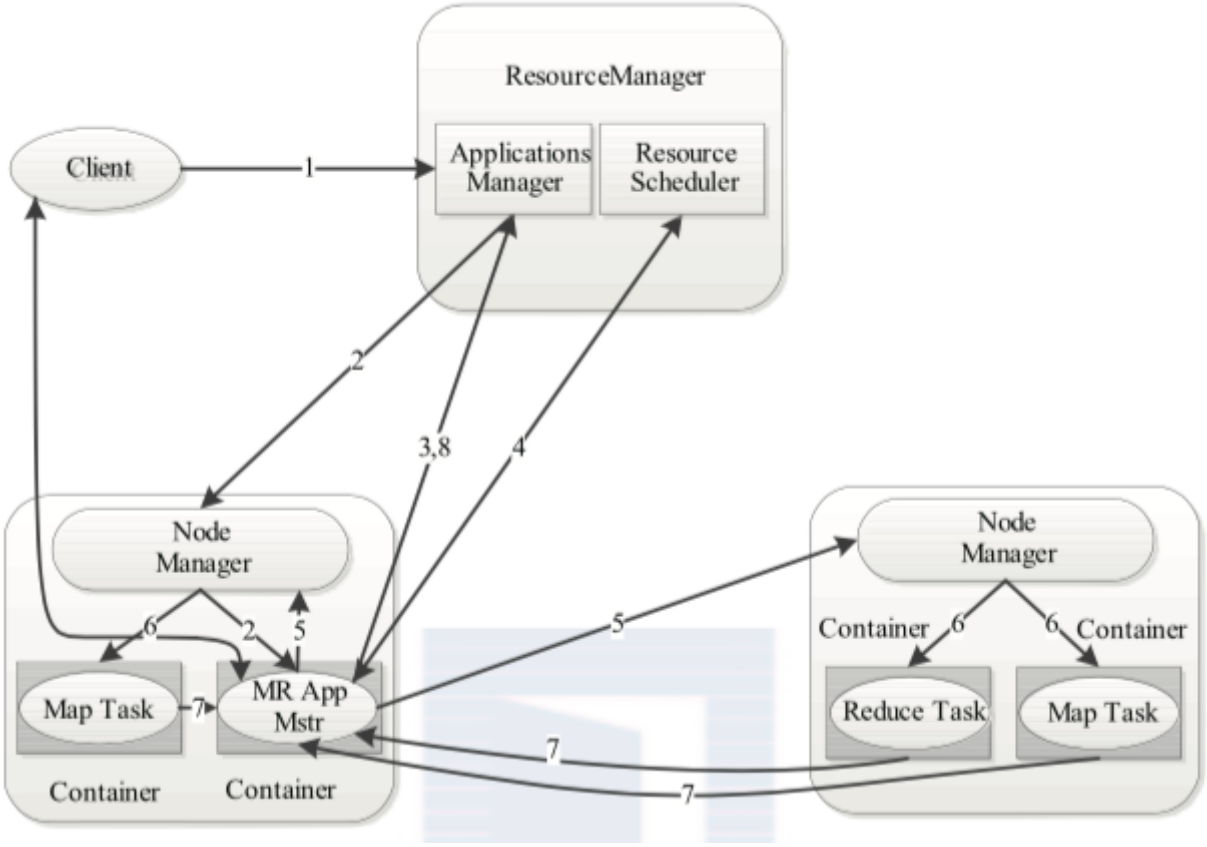
MapReduce 就是原生支持 ON YARN 的一种框架，可以在 YARN 上运行 MapReduce 作业。有很多分布式应用都开发了对应的应用程序框架，用于在 YARN 上运行任务，例如 Spark，Storm、Flink 等。

Container

Container 是 YARN 中的资源抽象，它封装了某个节点上的多维度资源，如内存、CPU、磁盘、网络等，当 AM 向 RM 申请资源时，RM 为 AM 返回的资源便是用 Container 表示的。 YARN 会为每个任务分配一个 Container 且该任务只能使用该 Container 中描述的资源。

YARN 作业提交流程

当用户向 YARN 中提交一个应用程序后，YARN 将分两个阶段运行该应用程序：第一个阶段是启动 ApplicationMaster；第二个阶段是由 ApplicationMaster 创建应用程序，为它申请资源，并监控它的整个运行过程，直到运行完成，如下图所示（此图来自《Hadoop 技术内幕：深入解析 YARN 架构设计与实现原理》）：



YARN 工作流程

上图所示的 YARN 工作流程分为以下几个步骤：

1. 用户向 YARN 提交应用程序，其中包括 ApplicationMaster 程序，启动 ApplicationMaster 命令、用户程序等；
2. RM 为该应用程序分配第一个 Container，并与对应的 NM 通信，要求它在这个 Container 中启动应用程序的 ApplicationMaster；
3. ApplicationMaster 首先向 RM 注册，这样用户可以直接通过 NM 查看应用程序的运行状态，然后它将为各个任务申请资源，并监控它的运行状态，直到运行结束，一直重复下面的 4-7 步；
4. ApplicationMaster 采用轮询的方式通过 RPC 协议向 RM 申请和领取资源；
5. 一旦 ApplicationMaster 申请到资源后，便与对应的 NM 通信，要求它启动任务；
6. NM 为任务设置好运行环境（包括环境变量、jar 包等）后，将任务启动命令写到一个脚本中，并通过运行该脚本启动任务；
7. 各个任务通过某个 RPC 协议向 ApplicationMaster 汇报自己的状态和进度，以让 ApplicationMaster 随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务；
8. 应用程序运行完成后，ApplicationMaster 向 RM 注销并关闭自己（当然像 Storm、Flink 这种常驻应用程序列外）。

调度器

YARN 的调度器是一个可插拔的组件，目前社区已经提供了 FIFO Scheduler（先进先出调度器）、Capacity Scheduler（能力调度器）、Fair Scheduler（公平调度器），用户也可以继承 ResourceScheduler 的接口实现自定义的调度器，就像 app on yarn 流程一样，不同的应用可以自己去实现，这里只是简单讲述上述三种调度器的基本原理。

FIFO Scheduler

FIFO 是最简单的资源调度策略，提交的作业按照提交时间先后顺序或者根据优先级次序将其放入线性队列相应的位置，在资源调度时，按照队列的先后顺序、先进先出地进行调度和资源分配。

很明显这种调度器过于简单，在实际的生产中，应用不是很多，毕竟需要调度的作业是有不同的优先级的。

公平调度器（Fair Scheduler)

公平调度器先将用户的任务分配到多个资源池（Pool）中，每个资源池设定资源分配最低保障和最高上限，管理员也可以指定资源池的优先级，优先级高的资源池将会被分配更多的资源，当一个资源池有剩余时，可以临时将剩余资源共享给其他资源池。公平调度器的调度过程如下：

1. 根据每个资源池的最小资源保障，将系统中的部分资源分配给各个资源池；
2. 根据资源池的指定优先级讲剩余资源按照比例分配给各个资源池；
3. 在各个资源池中，按照作业的优先级或者根据公平策略将资源分配给各个作业；

公平调度器有以下几个特点：

1. 支持抢占式调度，即如果某个资源池长时间未能被分配到公平共享量的资源，则调度器可以杀死过多分配资源的资源池的任务，以空出资源供这个资源池使用；
2. 强调作业之间的公平性：在每个资源池中，公平调度器默认使用公平策略来实现资源分配，这种公平策略是最大最小公平算法的一种具体实现，可以尽可能保证作业间的资源分配公平性；
3. 负载均衡：公平调度器提供了一个基于任务数目的负载均衡机制，该机制尽可能将系统中的任务均匀分配到给各个节点上；
4. 调度策略配置灵活：允许管理员为每个队列单独设置调度策略；
5. 提高小应用程序响应时间：由于采用了最大最小公平算法，小作业可以快速获得资源并运行完成。

能力调度器（Capacity Scheduler)

能力调度器是 Yahool 为 Hadoop 开发的多用户调度器，应用于用户量众多的应用场景，与公平调度器相比，其更强调资源在用户之间而非作业之间的公平性。

它将用户和任务组织成多个队列，每个队列可以设定资源最低保障和使用上限，当一个队列的资源有剩余时，可以将剩余资源暂时分享给其他队列。调度器在调度时，优先将资源分配给资源使用率最低的队列（即队列已使用资源量占分配给队列的资源量比例最小的队列）；在队列内部，则按照作业优先级的先后顺序遵循 FIFO 策略进行调度。

能力调度器有以下几点特点：

1. 容量保证：管理员可为每个队列设置资源最低保证和资源使用上限，而所有提交到该队列的应用程序共享这些资源；
2. 灵活性：如果一个队列资源有剩余，可以暂时共享给那些需要资源的队列，而一旦该队列有新的应用程序提交，则其他队列释放的资源会归还给该队列；
3. 多重租赁：支持多用户共享集群和多应用程序同时运行，为防止单个应用程序、用户或者队列独占集群中的资源，管理员可为之增多多重约束；
4. 安全保证：每个队列有严格的 ACL 列表规定它访问用户，每个用户可指定哪些用户允许查看自己应用程序的运行状态或者控制应用程序；
5. 动态更新配置文件：管理可以根据需要动态修改各种配置参数。

Yarn 容错

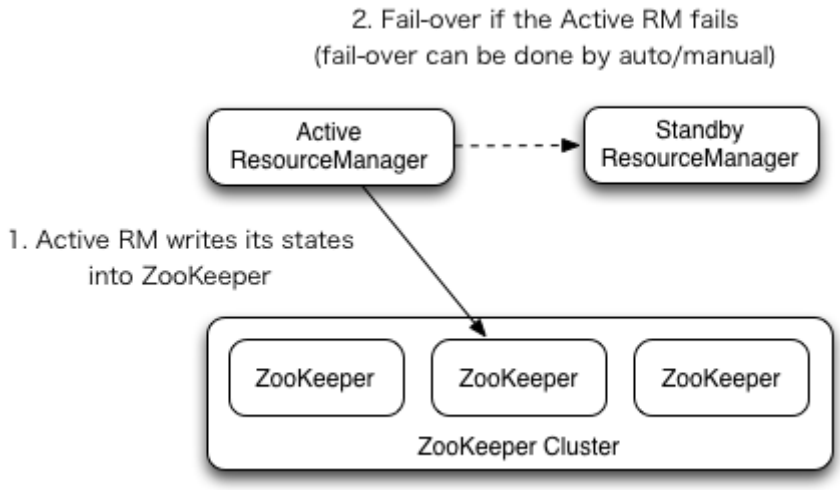
对于分布式系统，不论是调度系统还是其他系统，容错机制都是非常必要的，这里我们简单看下 YARN 的容错机制，YARN 需要做容错的地方，有以下四个地方：

1. ApplicationMaster 容错：ResourceManager 会和 ApplicationMaster 保持通信，一旦发现 ApplicationMaster 失败或者超时，会为其重新分配资源并重启。重启后 ApplicationMaster 的运行状态需要自己恢复，比如 MRAppMaster 会把相关的状态记录到 HDFS 上，重启后从 HDFS 读取运行状态恢复；
2. NodeManager 容错：NodeManager 如果超时，则 ResourceManager 会认为它失败，将其上的所有 container 标记为失败并通知相应的 ApplicationMaster，由 AM 决定如何处理（可以重新分配任务，可以整个作业失败，重新拉起）；
3. container 容错：如果 ApplicationMaster 在一定时间内未启动分配的 container，RM 会将其收回，如果 Container 运行失败，RM 会告诉对应的 AM 由其处理；
4. RM 容错：RM 采用 HA 机制，这里详细讲述一下。

ResourceManager HA

因为 RM 是 YARN 架构中的一个单点，所以他的容错很难做，一般是采用 HA 的方式，有一个 active master 和一个 standby master（可参考：ResourceManager High Availability），HA

的架构方案如下图所示：



YARN RM HA 机制

关于 YARN 的 RM 的 HA 机制，其实现与 HDFS 的很像，可以参考前面关于 HDFS 文章的讲述 HDFS NN HA 实现。

分布式调度器总结

上面基本已经把 YARN 的相关内容总结完了，这个小节主要讲述一下分布式调度系统的一些内容（调度框架只是具体的一种实现方案），主要讲述分布式调度系统要解决的一些问题和分布式调度系统的调度模型。

调度系统设计遇到的基本问题

对于分布式调度系统，在实际的生产环境中，遇到的问题很相似，这个小节就是看下调度系统主要面对的问题。

资源异构性与工作负载异构性

在资源管理与调度场景下，有两类异质性需要考虑：

1. 资源异质性：这个是从系统拥有资源的角度来看，对于数据中心来说非常常见，数据中心的机器很难保证完全一样的配置，有的配置会高一些，有的会低一些；
2. 工作负载异质性：在大型互联网公司中很常见，因为各种服务和功能特性各异，对资源的需求千差万别。

数据局部性（Data Locality）

在大数据场景下，还有一个基本的共识：将计算任务推送到数据所在地进行而不是反过来。因为数据的移动会产生大量低效的数据网络传输开销，而计算代码相比而言数据小得多，所以将计算任务推动到数据所在地是非常常见的，这就是数据局部性，在资源调度中，有三种类型的数据局部性，分别是：

1. 节点局部性（Node Locality）：计算任务分配到数据所在机器节点，无需任务网络传输；
2. 机架局部性（Rack Locality）：虽然计算任务与数据分布在不同的节点，但这两个节点在同一个机架中，这也是效率较高的一种数据性；
3. 全局局部性（Global Locality）：需要跨机架的传输，会产生较大的网络传输开销。

抢占式调度与非抢占式调度

在多用户场景下，面对已经分配的资源，资源管理调度系统可以有两种不同类型的调度方式：

1. 抢占式调度：对于某个计算任务来说，如果空闲资源不足或者出现不同任务共同竞争同一资源，调度系统可以从比当前计算任务优先级低的其他任务中获取已经分配资源，而被抢占资源的计算任务则需要出让资源停止计算；
2. 非抢占式调度：只允许从空闲资源中进行分配，如果当前空闲资源不足，则须等待其他任务释放资源后才能进行。

资源分配粒度（Allocation Granularity）

大数据场景下的计算任务往往由两层结构构成：作业级（Job）和任务级（Task），一个作业由多个并发任务构成，任务之间的依赖关系往往形成有向无环图（DAG），比如：MR 作业，关于作业资源分配的粒度，常见的有两种模式：

1. 群体分配（全分或不分）：需要将作业的所有所需资源一次性分配完成；
2. 增量满足式分配策略：对于某个作业，只要分配部分资源就能启动一些任务开始运行，随着空闲资源的不断出现，可以逐步增量式分配给作业的其他任务以维护作业不断向后进行。

还有一种策略是 资源储备策略，它指的是只有分配到一定量的资源资源才能启动，但是在未获得足够资源的时候，作业可以先持有目前已经分配的资源，并等待其他作业释放资源，这样从调度系统不断获取新资源并进行储备和累积，直到分配到的资源量达到最低标准后开始运行。

饿死（Starvation）与死锁（Dead Lock）问题

饿死和死锁是一个合理的资源调度系统需要避免的两个问题：

1. 饿死：指的是这个计算任务持续上时间无法获得开始执行所需的最少资源量，导致一直处于等待执行的状态，比如在资源紧张的情形下，有些低优先级的任务始终无法获得资源分配机会，如果不断出现新提交的高优先级任务，则这些低优先级任务就会出现饿死现象；
2. 死锁：指的是由于资源调度不当导致整个调度无法继续正常执行，比如前面提高的资源储备策略就有可能导致调度系统进入死锁状态，多个作业占有一定作业的情况下，都在等待新的资源释

放。

资源隔离方法

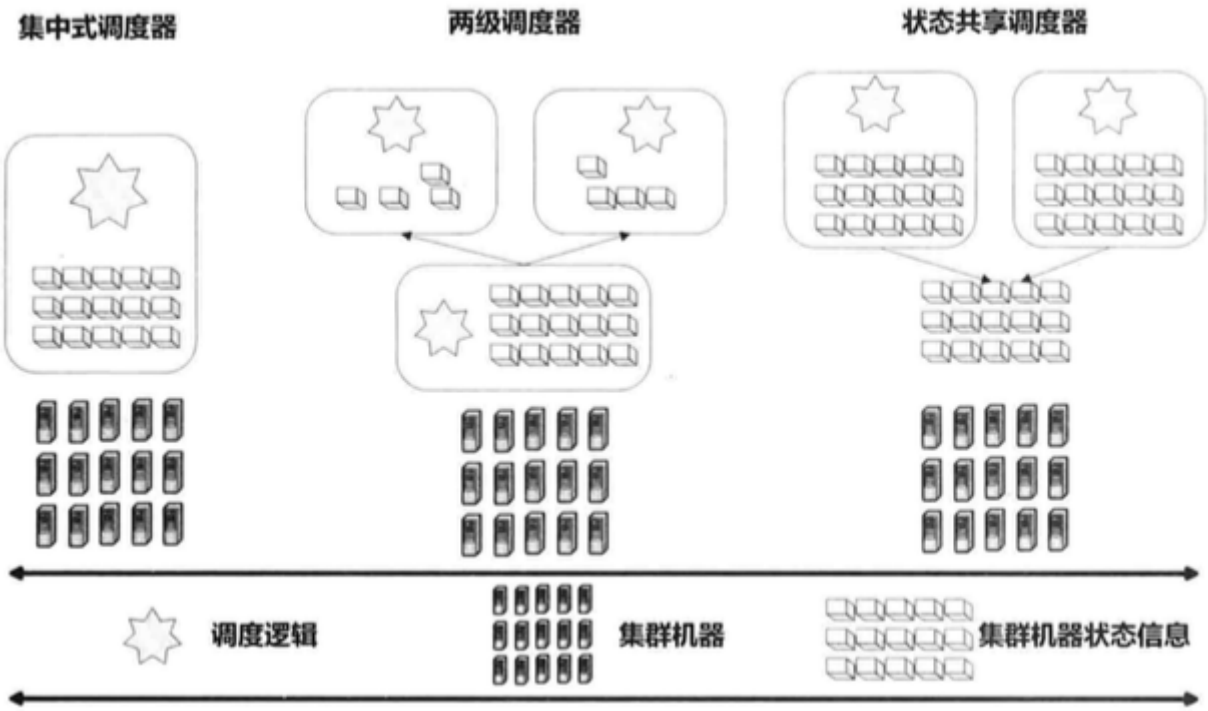
目前对于资源隔离最常用的手段是 Linux 容器（Linux Container，LXC，可以参考什么是 Linux 容器?），YARN 和 Mesos 都是采用了这种方式来实现资源隔离。LXC 是一种轻量级的内核虚拟化技术，可以用来进行资源和进程运行的隔离，通过 LXC 可以在一台物理机上隔离出多个互相隔离的容器。LXC 在资源管理方面依赖于 Linux 内核的 cgroups 子系统，cgroups 子系统是 Linux 内核提供的一个基于进程组的资源管理的框架，可以为特定的进程组限定可以使用的资源。

调度器模型

关于资源管理与调度功能的实际功能，分布式调度器根据运行机制的不同进行分类，可以归纳为三种资源管理与调度系统泛型：

1. 集中式调度器；
2. 两级调度器；
3. 状态共享调度器。

它们的区别与联系如下图所示：



三种调度模型

集中式调度器

集中式调度器在整个系统中只运行一个全局的中央调度器实例，所有之上的框架或者计算任务的资源请求全部经由中央调度器来满足（也就是说：资源的使用及任务的执行状态都由中央调度器管理），因此，整个调度系统缺乏并发性且所有调度逻辑全部由中央调度器来完成。集中式调度器有以下这些特点：

1. 适合批处理任务和吞吐量较大、运行时间较长的任务；
2. 调度逻辑全部融入到了中央调度器，实现逻辑复杂，灵活性和策略的可扩展性不高；
3. 并发性能较差，比较适合小规模的集群系统；
4. 状态同步比较容易且稳定，这是因为资源使用和任务执行的状态被统一管理，降低了状态同步和并发控制的难度。

两级调度器

对于集中式调度器的不足之处，两级调度器是一个很好的解决方案，它可以看做一种策略下放的机制，它将整个系统的调度工作分为两个级别：

1. 中央调度器：中央调度器可以看到集群中所有机器的可用资源并管理其状态，它可以按照一定策略将集群中的所有资源更配各个计算框架，中央调度器级别的资源调度是一种粗粒度的资源调度方式；
2. 框架调度器：各个计算框架在接收到所需资源后，可以根据自身计算任务的特性，使用自身的调度策略来进一步细粒度地分配从中央调度器获得的各种资源。

在这种两级调度器架构中，只有中央调度器能够观察到所有集群资源的状态，而每个框架并无全局资源概念（不知道整个集群资源使用情况），只能看到由中央调度器分配给自己的资源，Mesos、YARN 和 Hadoop on Demand 系统是3个典型的两级调度器。两级调度的缺点也非常明显：

1. 各个框架无法知道整个集群的实时资源使用情况：很多框架不需要知道整个集群的实时资源使用情况就可以运行得很顺畅，但是对于其他一些应用，为之提供实时资源使用情况可以挖掘潜在的优化空间；
2. 采用悲观锁，并发粒度小：悲观锁通常采用锁机制控制并发，这会大大降低性能。

状态共享调度器

通过前面两种模型的介绍，可以发现集群中需要管理的状态主要包括以下两种：

1. 系统中资源分配和使用的状态；
2. 系统中任务调度和执行的状态

在集中式调度器中，这两个状态都由中心调度器管理，并且一并集成了调度等功能；而在双层调度器中，这两个状态分别由中央调度器和框架调度器管理。集中式调度器可以容易地保证全局状态的一致性，但是可扩展性不够；双层调度器对共享状态的管理较难达到好的一致性保证，也不容易检测资源竞争和死锁。

这也就催生出了另一种调度器 —— 状态共享调度器（Shared-State Scheduler），它是 Google 的 Omega 调度系统提出的一种调度器模型。在这种调度器中，每个计算框架可以看到整个集群中的所有资源，并采用互相竞争的方式去获取自己所需的资源，根据自身特性采取不同的具体资源调整策略，同时系统采用了乐观并发控制手段解决不同框架在资源竞争过程中出现的需求冲突。这样，状态共享调度器在一下两个方面对两级调度器做了相应的优化：

1. 乐观并发控制增加了系统的并发性能；
2. 每个计算框架都可以获得全局的资源使用状况；

与两级调度器对比，两者的根本区别在于 中央调度器功能的强弱不同，两级调度器依赖中央调度器来进行第一次资源分配，而 Omega 则严重弱化中央调度器的功能，只是维护一份可恢复的集群资源状态信息的主副本，这份数据被称为 单元状态（Cell State）

1. 每个框架在自身内部会维护 单元状态 的一份私有并不断更新的副本信息，而框架对资源的需求则直接在这份副本信息上进行；
2. 只要框架具有特定的优先级，就可以在这份副本信息上申请相应的闲置资源，也可以抢夺已经分配给其他比自身优先级低的计算任务的资源；
3. 一旦框架做出资源决策，则可以改变私有 单元状态 信息并将其同步到全局的 单元状态 信息中区，这样就完成了资源申请并使得这种变化让其他框架可见；
4. 上述资源竞争过程通过 事务操作 来进行，保证了操作的原子性。

如果两个框架竞争同一份资源，因其决策过程都是在各自私有数据上做出的，并通过原子事务进行提交，系统保证此种情形下只有一个竞争胜出者，而失败者可以后续继续重新申请资源，这是一种乐观并发控制手段，可以增加系统的整体并发性能。

从上面的过程，可以看出，这种架构是一种 以效率优先，不太考虑资源分配公平性 的策略，很明显高优先级的任务总是能够在资源竞争过程中获胜，而低优先级的任务存在由于长时间无法竞争到所需资源而被【饿死】的风险。

参考：

- 《Hadoop 技术内幕：深入解析 YARN 架构设计与实现原理》；
- 《大数据日知录：架构与算法》；
- Hadoop Yarn；
- Hadoop 新 MapReduce 框架 Yarn 详解；
- Hadoop YARN架构设计要点；
- YARN 简介；
- 理解Hadoop YARN架构；
- 这里有7种主流案例，告诉你调度器架构设计通用法则；

博客版权说明 分享到