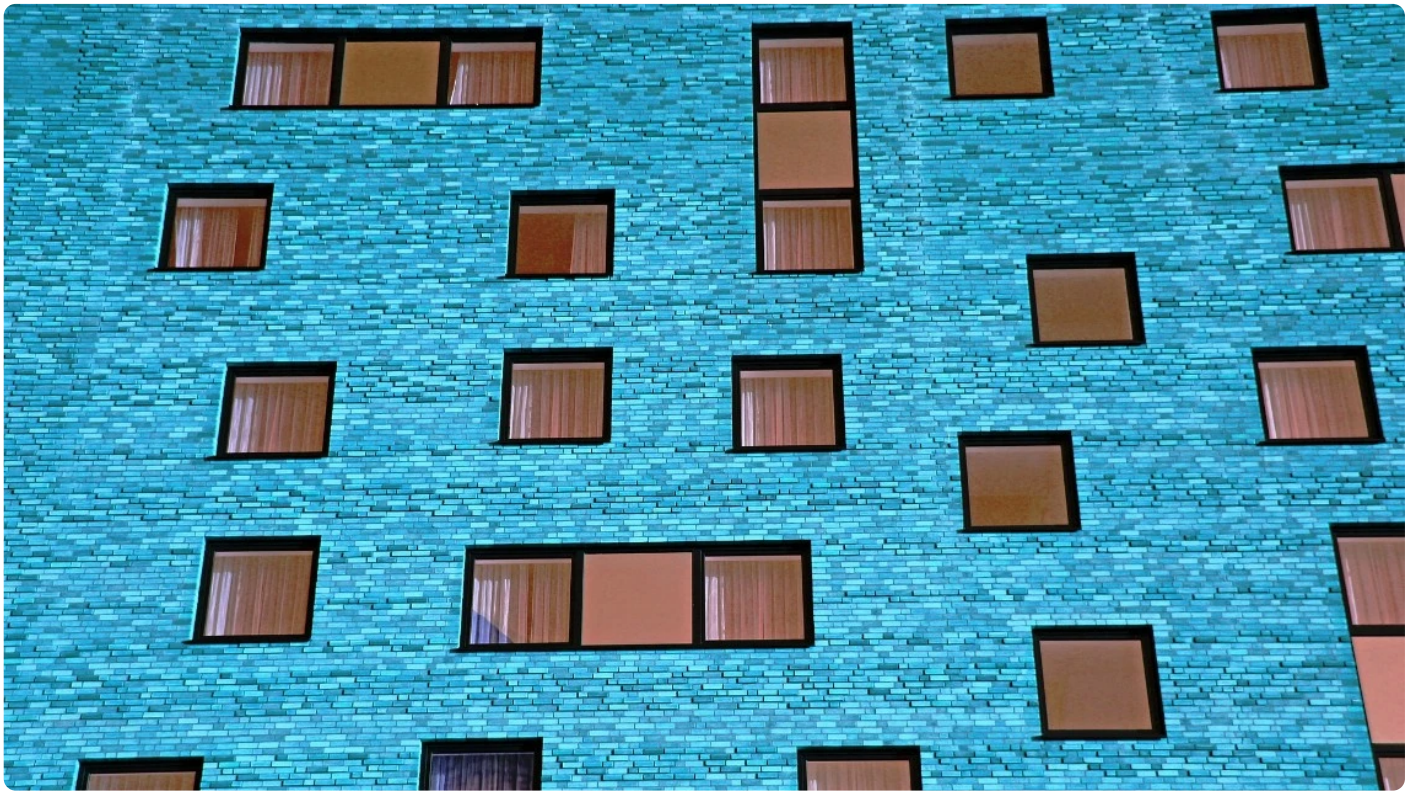
13 | 分布式调度架构之共享状态调度：物质文明、精神文明多手协商抓

2019-10-21 聂鹏程 来自北京

《分布式技术原理与算法解析》



你好，我是聂鹏程。今天，我来继续带你打卡分布式核心技术。

在上一篇文章中，我们一起学习了两层调度。在两层调度架构中，第二层调度只知道集群中的部分资源，无法进行全局最优调度。那么，是否有办法解决全局最优调度的问题呢？

答案是肯定的，解决办法就是我今天要带你打卡的共享状态调度。

接下来，我们就一起看看共享状态调度到底是什么，以及它的架构和工作原理吧。

什么是共享状态调度？

通过我们前两篇文章的讲述，不难发现，集群中需要管理的对象主要包括两种：

一是，资源的分配和使用状态；

二是，任务的调度和执行状态；

在单体调度中，这两种对象都是由单体调度器管理的，因此可以比较容易地保证全局状态的一致性，但问题是可扩展性较差（支持业务类型受限），且存在单点瓶颈问题。

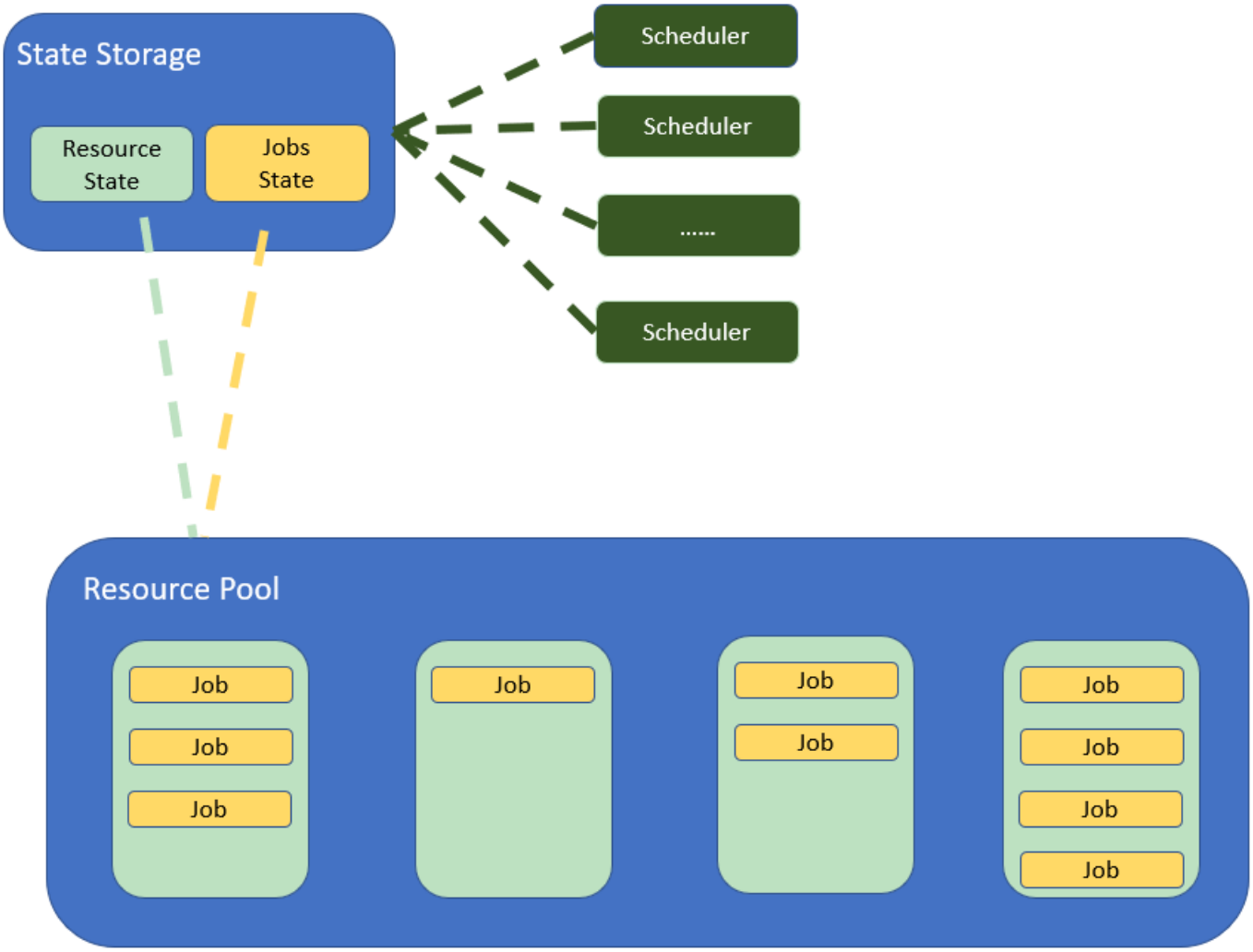
而在两层调度中，这两种对象分别由第一层中央调度器和第二层 Framework 调度器管理，由于 Framwork 调度器只能看到部分资源，因此不能保证全局状态的一致性，也不容易实现全局最优的调度。

为了解决这些问题，一种新的调度器架构被设计了出来。这种架构基本上沿袭了单体调度器的模式，通过将单体调度器分解为多个调度器，每个调度器都有全局的资源状态信息，从而实现最优的任务调度，提供了更好的可扩展性。

也就是说，这种调度架构在支持多种任务类型的同时，还能拥有全局的资源状态信息。要做到这一点，这种调度架构的多个调度器需要共享集群状态，包括资源状态和任务状态等。因此，这种调度架构，我们称之为**共享状态调度器**。

如果我们继续把资源比作物质文明、把任务比作精神文明的话，相对于单体调度和两层调度来说，共享状态调度就是“物质文明与精神文明多手协商抓”。

共享状态调度架构的示意图，如下所示：



可以看出，**共享状态调度架构为了提供高可用性和可扩展性，将集群状态之外的功能抽出来作为独立的服务。**具体来说就是：

State Storage 模块（资源维护模块）负责存储和维护资源及任务状态，以便 Scheduler 查询资源状态和调度任务；

Resource Pool 即为多个节点集群，接收并执行 Scheduler 调度的任务；

而 Scheduler 只包含任务调度操作，而不是像单体调度器那样还需要管理集群资源等。

共享状态调度也支持多种任务类型，但与两层调度架构相比，主要有两个不同之处：

存在多个调度器，每个调度器都可以拥有集群全局的资源状态信息，可以根据该信息进行任务调度；

> 共享状态调度是乐观并发调度，在执行了任务匹配算法后，调度器将其调度结果提交给 State Storage，由其决定是否进行本次调度，从而解决竞争同一种资源而引起的冲突问题，实现全局最优调度。而，两层调度是悲观并发调度，在执行任务之前避免冲突，无法实现全局最优匹配。

看到这里，我再和你说说**乐观并发调度和悲观并发调度的区别**吧。

乐观并发调度，强调事后检测，在事务提交时检查是否避免了冲突：若避免，则提交；否则回滚并自动重新执行。也就是说，它是在执行任务匹配调度算法后，待计算出结果后再进行冲突检测。

悲观并发调度，强调事前预防，在事务执行时检查是否会存在冲突。不存在，则继续执行；否则等待或回滚。也就是说，在执行任务匹配调度算法前，通过给不同的 Framework 发送不同的资源，以避免冲突。

现在，我们已经对共享状态调度有了一个整体印象，知道了它可以解决什么问题。那么接下来，我们再看看这种调度架构是如何设计的吧。

共享状态调度设计

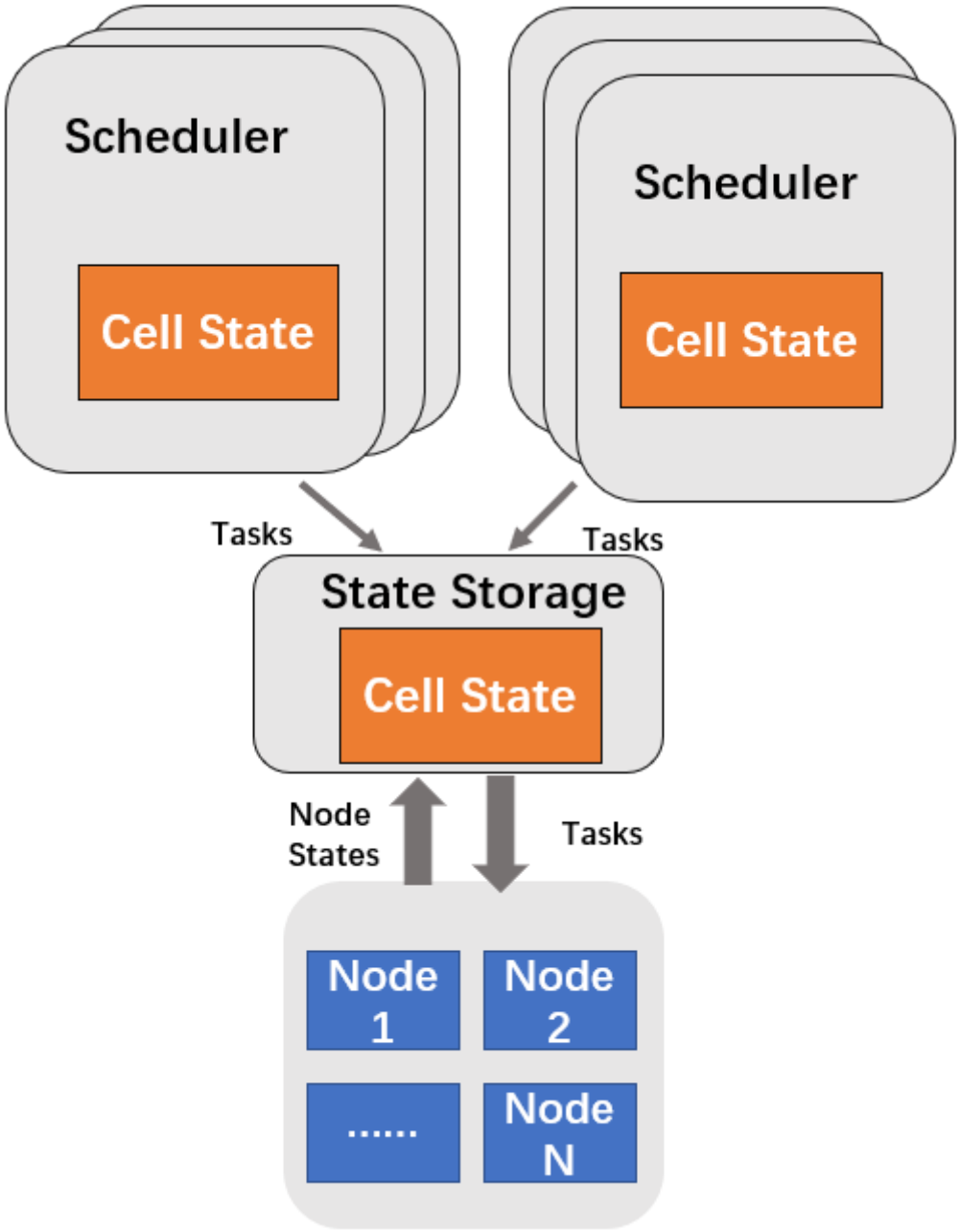
共享状态调度的理念最早是 Google 针对两层调度器的不足，提出的一种调度架构。这种调度结构的典型代表有 Google 的 Omega、微软的 Apollo，以及 Hashicorp 的 Nomad 容器调度器。

作为 Google 公司的第二代集群管理系统，Omega 在设计时参考了 Borg 的设计，吸收了 Borg 的优点，并改进了其不足之处。所以接下来，我就以 Omega 为例和你讲述共享状态调度的架构和工作原理吧。这样一来，你可以对照着第 11 篇文章中 Borg 的调度设计一起理解。

Omega 调度架构

Omega 集群中有一个“Cell”的概念，每个 Cell 管理着部分物理集群，一个集群有多个 Cell。实际上，你可以直接将这里的“Cell”理解为一个集群的子集群或子节点的集合。

Omega 集群的调度架构示意图，如下所示。



我在介绍共享状态调度的架构时提到，State Storage 模块负责存储和维护资源及任务状态，里面有一个 Cell State 文件，记录着全局共享的集群状态。实际上，State Storage 组件中的集群资源状态信息，就是主本，而 Cell State 就是以主副本的形式存在的。每个调度器都包含一个私有的 Cell State 副本，也就是拥有了一个集群资源状态信息的副本，进而达到了共享集群资源状态信息的目的。

在 Omega 系统中，没有中央资源分配器，所有资源分配决策都在调度器（Scheduler）中进行。每个调度器都可以根据私有的 Cell State 副本，来制定调度决策。

调度器可以查看 Cell 的整个状态，并申请任何可用的集群资源。一旦调度器做出资源调度决策，它就会在原子提交中更新本地的 Cell State 的资源状态副本。若同时有多个调度器申请同一份资源，State Storage 模块可以根据任务的优先级，选择优先级最高的那个任务进行调度。

可以看出，在 Omega 系统中的每个调度器，都具有对整个集群资源的访问权限，从而允许多个调度器自由地竞争空闲资源，并在更新集群状态时使用乐观并发控制来调解资源冲突问题。

这样一来，Omega 就有效地解决了两层调度中 Framework 只拥有局部资源，无法实现全局最优的问题。

接下来，我们看一下 Omega 共享调度的工作原理吧。

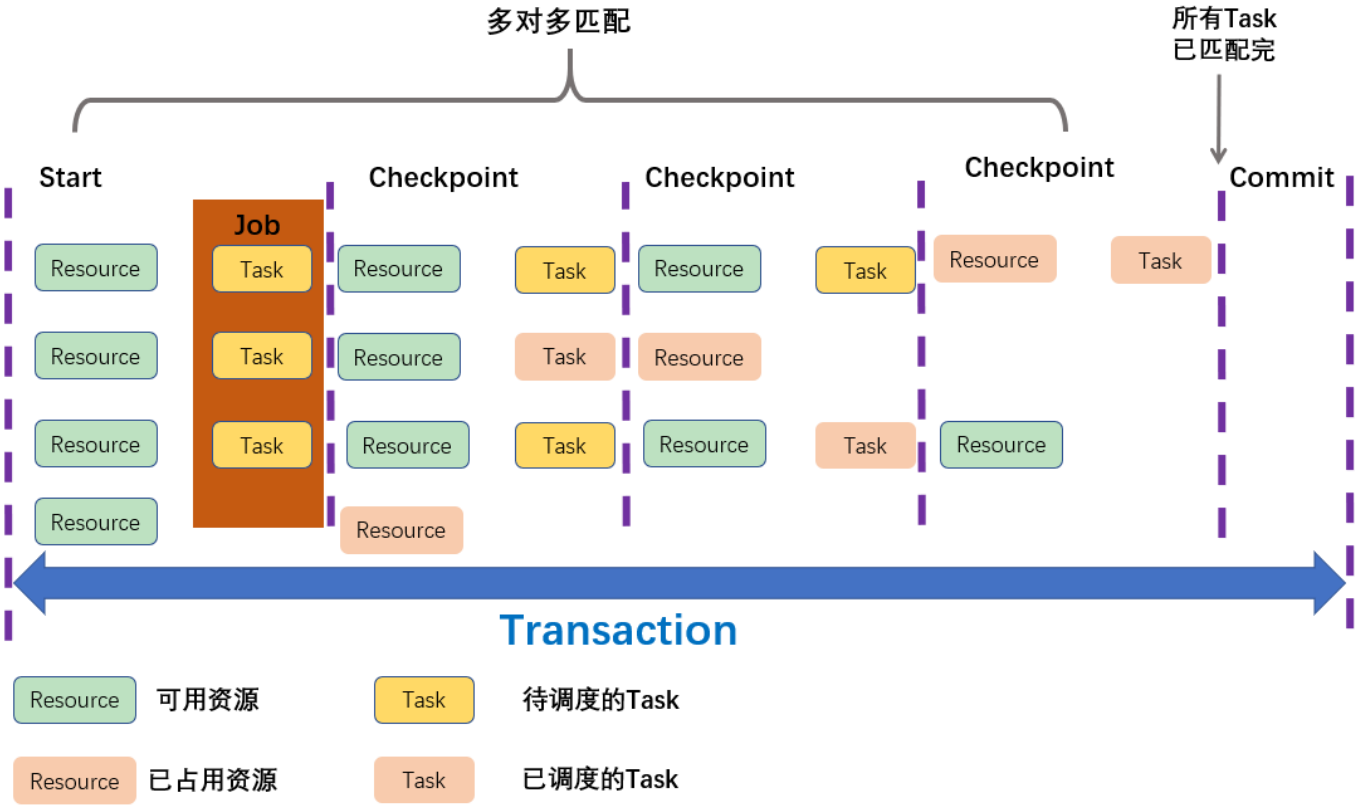
Omega 共享调度工作原理

Omega 使用事务管理状态的设计思想，将集群中资源的使用和任务的调度类似于基于数据库中的一条条事务（Transaction）一样进行管理。显然，数据库是一个共享的状态，对应 Omega 中的 Cell State，而每个调度器都要根据数据库的信息（即集群的资源信息）去独立完成自己的任务调度策略。

接下来，我们就具体看看吧。

如下图所示，在一个应用执行的过程中，调度器会将一个 Job 中的所有 Task 与 Resource 进行匹配，可以说 Task 与 Resource 之间是进行多对多匹配的。其间，调度器会设置多个 Checkpoint 来检测 Resource 是否都已经被占用，只有这个 Job 的所有 Task 可以匹配到可用资源时，这个 Job 才可以被调度。

这里的 Job 相当于一个事务，也就是说，当所有 Task 匹配成功后，这个事务就会被成功 Commit，如果存在 Task 匹配不到可用资源，那么这个事务需要执行回滚操作，Job 调度失败。



无论事务是否执行成功，调度器都会在事务执行之后，重新从主本那里同步更新本地 Cell State 的资源状态副本，以保证本地集群信息状态的有效性。若事务未成功执行，则调度器会在必要时重新运行其调度算法并再次尝试申请资源。

也就是说，调度器对 Job 的调度是具有原子性的，一个 Job 的所有 Task 都是一起调度的，即使部分 Task 调度失败了，调度器再次调度时也必须再次调度整个 Job。多个调度器可以并行调度，无需等待其他调度器调度结果，若存在冲突时，进行冲突处理，比如根据 Job 的优先级，优先级高则获得资源。

由此我们可以看到，**Omega 涉及了 Job 并发调度**。针对这一点，Omega 采用了传统数据库中的乐观锁（MVCC，Multi-Version Concurrency Control，基于多版本的并发访问控制），即每一个应用都发放了所有的可用资源，在更新集群状态时使用乐观并发控制来解决资源冲突问题，来提高 Omega 的并发度。

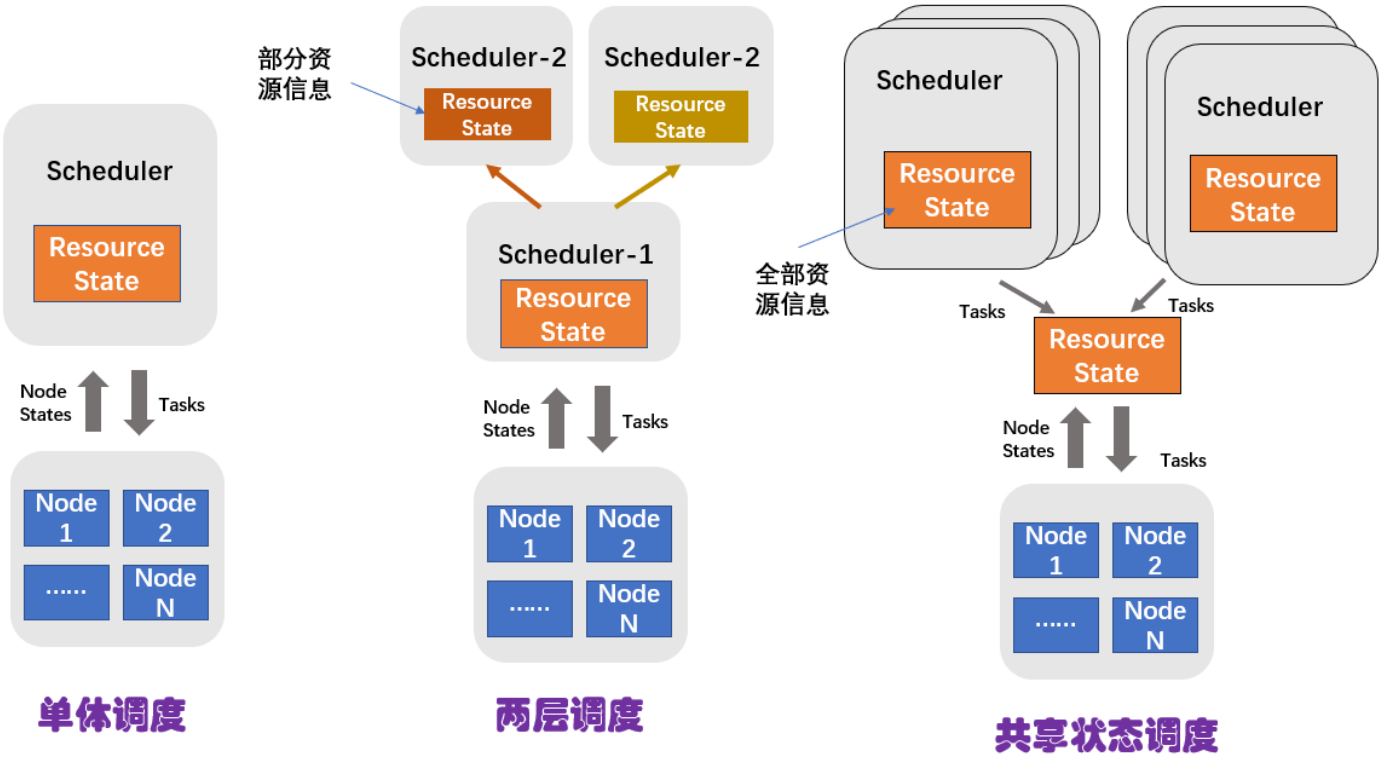
不同的 Omega 调度器可以实现不同的策略，但有一些调度规则是所有调度器必须达成一致的，比如哪些资源是允许分配的、如何评估作业的优先级等。

因此，**Omega 调度器将两层调度器中的集中式资源调度模块简化成了一些持久化的共享数据（状态）和针对这些数据的验证代码。**而这里的“共享数据”，实际上就是整个集群的实时资源状态信息，而验证代码就是解决调度冲突的调度规则。

知识扩展：单体调度、两层调度和共享调度的区别是什么？

现在，我已经带你学习了单体调度、双层调度和共享调度，那么这三种调度的区别是什么呢？接下来，我们就一起回忆并对比下吧。

我把这三种调度的架构示意图放到一起，先帮你有一个整体认识。



单体调度，是由一个中央调度器去管理整个集群的资源信息和任务调度，也就是说所有任务只能通过中央调度器进行调度。

这种调度架构的优点是，中央调度器拥有整个集群的节点资源信息，可以实现全局最优调度。但它的缺点是，无调度并发性，且中央服务器存在单点瓶颈问题，导致支持的调度规模和服务类型受限，同时会限制集群的调度效率。因此，单体调度适用于小规模集群。

两层调度，是将资源管理和任务调度分为两层来调度。其中，第一层调度器负责集群资源管理，并将可用资源发送给第二层调度；第二层调度接收到第一层调度发送的资源，进行任务调度。

这种调度架构的优点是，避免了单体调度的单点瓶颈问题，可以支持更大的服务规模和更多的服务类型。但其缺点是，第二层调度器往往只对全局资源信息有部分可观察性，因此任务匹配算法无法实现全局最优。双层调度适用于中等规模集群。

共享状态调度，多个调度器，每个调度器都可以看到集群的全局资源信息，并根据这些信息进行任务调度。相较于其他两个调度架构来说，共享状态调度架构适用的集群规模最大。

这种调度架构的优点是，每个调度器都可以获取集群中的全局资源信息，因此任务匹配算法可以实现全局最优性。但，也因为每个调度器都可以在全局范围内进行任务匹配，所以多个调度器同时调度时，很可能会匹配到同一个节点，从而造成资源竞争和冲突。

虽然 Omega 的论文宣称可以通过乐观锁机制，避免冲突。但在工程实践中，如果没有妥善处理资源竞争的问题，则很可能会产生资源冲突，从而导致任务调度失败。这时，用户就需要对调度失败的任务进行处理，比如重新调度、任务调度状态维护等，从而进一步增加了任务调度操作的复杂度。

我将单体调度、两层调度、共享状态调度总结在了一张表格中：

	单体调度	两层调度	共享状态调度
调度架构	集中式结构：一个中央调度器	树形结构：一个中央调度器，多个第二层调度器	分布式结构：多个对等调度器
调度形式	单点集中调度	Resource Offer	Transaction
调度单位	Task	Task	Task
任务调度的并发性	无并发	悲观并发调度	乐观并发调度
是否是全局最优调度	是	否	是
系统并发度	共享状态调度>两层调度>单体调度		
调度效率（综合考虑并发度，全局最优性，以及故障问题等因素）	共享状态调度>两层调度>单体调度		
系统可扩展性	共享状态调度>两层调度>单体调度		
是否有具体实现源码	是	是	否
适用场景	小规模集群，适用于业务类型比较单一的场景	中等规模集群，适用于同时具有多种业务类型的场景	大规模集群，适用于同时具有多种业务类型的场景
典型应用	Borg	Mesos、YARN	Omega

总结

今天，我主要与你分享了分布式调度架构设计中的共享状态调度。我们一起来总结下今天的核心知识点吧。

首先，我讲述了什么是共享状态调度。概括地说，共享状态调度是将单体调度器分解为多个服务，由多个服务共享集群状态，包括资源状态和任务状态等。

接下来，我以 Google 的 Omega 集群管理系统为例，和你分享了共享状态调度的架构和工作原理。共享状态调度包含多个调度器，每个调度器都可以看到集群的全局资源信息，并根据这些信息进行任务调度。

最后，我要和你说明的是，共享状态调度是乐观并发调度，调度器将其调度的结果以原子的方式提交给资源维护模块，由其决定是否进行本次调度。

接下来，我整理一张思维导图来帮助你巩固今天的核心知识点。

我想让你知道的是，在分布式领域中，共享状态调度，是 Google 号称的下一代集群管理系统 Omega 的调度机制，可以解决双层调度无法实现全局最优的问题，同时也避免了单体调度的单点瓶颈问题。

但，说到这儿你可能会回想起曾经看到的两句话：

1. 为了达到设计目标，Omega 的实现逻辑变得越来越复杂。 在原有的 Borg 共享状态模型已经能满足绝大部分需要的情况下，Omega 的前景似乎没有那么乐观。
2. Omega 系统缺点是，在小集群下没有优势。

这里，我再与你解释下，为什么说 Omega 是 Google 准备打造的下一代集群管理系统。

从调度架构方面来看，Borg 无法支持同时存在多种业务类型的场景，并且存在单点瓶颈问题。而 Omega 解决了 Borg 的这两个问题，但是当多个调度器并行调度时，可能存在资源冲突，当资源申请产生冲突时，会导致大量任务或任务多次调度失败，增加了任务调度失败的故障处理的复杂度，比如需要进行作业回滚、任务状态维护等。

因此，设计一个好的冲突避免策略是共享状态调度的关键。对于小规模集群来说，其集群规模、任务数量等都不大，使用单体调度就可以满足其任务调度的需求，避免了考虑复杂的冲突避免策略。也就是说，共享状态调度比较适合大规模、同时存在多种业务类型的场景，不太适合小规模集群。

思考题

共享状态调度的核心是解决并发冲突，那你认为有没有什么好的方法可以解决冲突呢？

我是聂鹏程，感谢你的收听，欢迎你在评论区给我留言分享你的观点，也欢迎你把这篇文章分享给更多的朋友一起阅读。我们下期再会！

精选留言 (16)

亢（知行合一的路上）
2020-03-21

单体调度：一个部门经理（调度）给所有下属安排工作；
双层调度：多个项目经理（二层调度）向部门经理（一层调度）申请人力资源；
共享状态：两位部门经理同时给下属安排工作，根据任务优先级解决冲突。

作者回复: 赞，非常有意思，用人力管理来形象介绍单体调度、双层调度和共享状态调度的区别

22

Eternal
2019-10-30

感觉这一节超级棒，老师通过事务的例子来解释共享状态的调度很深刻。
我对老师的问题关于共享状态调度的核心解决并发冲突的思考：
一个job由多个task组成，一个job要么全都执行，要么全都不执行，就像老师说的理解成一个事务；
1.job执行执行的时候如果阻塞等待资源，我们可以将阻塞加上超时时间，超时后还不能获取到资源，当前job主动释放自己已经占有的资源，这可以叫做等待超时时间；
2.如果一个job占有了一些资源，正在执行，我们可以给当前job设置一个超时时间，如果job在超时时间内还没把资源执行完，自己主动释放占有的资源，回滚job的所有task，这可以理解成作业超时时间
3.多个job的多个task在抢资源的时候，我们可以设计一个公平和非公平的抢占队列
4.共享状态下的调度有一个特点是每个调度器都有全局的资源，这个可以这样改进一下：每个调度器的资源分为两部分组成，一部分独占，一部分由共享抢占获取，这样减少并发的程度；
这让我想到了操作系统的内存模型，一个核心占用一部分固定内存，然后其它的内存是共享的，也就是可以抢占的

共 1 条评论 >　　10

上校
2020-05-05

老师说到omega的缺点，感觉可以用多个Borg管理多个集群，然后不同Borg负责不同的任务类型的执行来解决omega的缺点

> 作者回复: 多个borg相当于多个集群，如果采用多个集群，又会引入多集群之间的数据一致性、集群可靠性等问题

共 2 条评论 >　　4

波波安
2019-10-21

解决并发冲突的方法就是加锁，在分布式架构中可以采用分布式锁，具体有前面讲到的三种，基于数据库的，基于redis的，基于zookeeper的分布式锁。

1

leslie
2019-10-21

关于并发其实数据系统中一直很早就在处理这种问题：从早期单机RMDB->读写分离->一主多从->内存库【虽然市面上各种分类众多，可是个人还是偏向这种说法，各种关系太复杂

了】。

谈不上什么特别好的解决方案：就谈谈老师课程中提到的Google对此的做法吧MVCC，其实这确实是一条解决的方式和思路，就像内存库就无法做到ACID特性，只能牺牲部分。目前主流其实就是老师文中所说的先锁或者后验证。

前段时间和本地的同行交流发现其实随着现在系统的越来越复杂：操作系统都开始特性化的时代确实感觉定制化的东西已经越来越多的出现在计算机的各个方面，如何从中获得一个相对中立的方案才是关键吧。

2

集团军群

2021-06-02

老师这课后总结，真是一流的。

西门吹牛

2020-08-20

俩种调度方式，就和单机下通过悲观锁和乐观锁访问共享资源一样，这种思想到处可见，要学习思想才能自己设计。

共享的调度，同样的思想，在计算几上也有类似，比如cpu执行指令，会采用分支预测的方式来尽可能提升cpu的利用率，预测对了，那么就节省了cpu等待条件判断的返回的时间，如果条件判断结果出来，发现错了，那就重来，要知道，cpu是很快的，在大的循环里面，只会在最后一次会预测错误，总的来说利大于弊。

风华笔墨

2020-05-20

我觉得解决并发冲突，在本地调度计算出资源分配的过程中，找出资源的最大值和最小值作为一个区间划分。同时还需要各个cell state每隔一定时间进行广播比对，使没有交叉的区间先执行。

> 作者回复: 解决冲突的策略有很多种，你说的是其中一种策略，根据不同的场景和需求，可以采用不同的策略。

南国

2020-04-17

关于多节点之间的并发冲突问题应该有以下解决方案:
1.应用层做一些调整,使得在每个调度器上的数据申请不同的资源,避免并发问题的最好方法就是不出现并发.
2.乐观的解决方案,多节点一起提交,在资源管理器上收敛于一致状态.
3.分布式锁,这个感觉效率不高,因为每次得到锁得到资源以后还需要一个全局的共识.

作者回复: 1. 避免并发的问题最好的方法就是不出现并发，这个在分布式场景中很难避免的，而且串行的效率一般比较低；
2. 你说的这种情况，其实就是由一个中心资源管理器来解决或协调冲突；
3. 每个算法都是有优点也有其局限的地方，需要根据自己的场景所关注的点进行取舍。

1

亢（知行合一的路上）
2020-03-18

今天又看了一遍，之前都没理解啊😓 看了之后，还得实践，才能加深理解。
现在还不知道调度器的 Cell State 和 State Storage 中的 Cell State 是怎么保持一致的?
新来的 job，调度器根据本地 State 副本进行 task 和资源的映射，完成后，将结果提交到中心？如果无冲突，则将变更同步到本地；如果冲突，也更新，过会再试。
待继续学习……

钱
2020-02-16

并发冲突的解决思路?
1：并行串行化，锁 + 阻塞队列
2：类似CAS，不断尝试
那种方式比较好，需要看场景，竞争大的加锁控制会好一些，竞争小的使用CAS应该可以😄

米 虫
2019-12-13

看了这篇文章，感觉只要有了类似redis和锁后，这个世界已经不存在单点问题了。

阿卡牛
2019-10-29

不同的体系结构是否对选择不同的调试结构有影响？

天天向善
2019-10-22

很抽像，能不能描述一下调度器做了哪些事情，任务举几个例子，共享状态调度的资源状态和任务状态有哪些

Jackey
2019-10-21

能想到的解决冲突的方法就是加锁了，乐观锁或是悲观锁，本质上都是让并行变串行。当然也有一些可以优化的点，比如读读并行。

随心而至
2019-10-21

1.同步
给资源编号，按照一定的顺序加锁，释放锁，以免出现死锁
2.CAS compare and swap
可能还有其他方式，可以参考维基百科的Synchronization条目