36 | 从Omega到Kubernetes：哺育云原生的开源项目

2022-01-17 徐文浩

《大数据经典论文解读》 课程介绍 >

讲述：徐文浩
时长 14:17 大小 13.09M

你好，我是徐文浩。

在前面两节课里，我们一起看过在 2015 年发表的 Borg 的论文。不过，Borg 这个系统的开发与使用，其实要远远早于 2015 年。事实上，在 2004 年 Google 发表的 MapReduce 的论文里，我们就已经隐隐约约可以看到 Borg 的存在了。

而在 2015 年，Borg 也早就已经进行了很多次的进化。在 2013 年，Google 就发表了一篇叫做《Omega: Flexiable, Scalable Schedulers for large compute clusters》的论文，向整个工业界介绍了 **Omega** 这个调度系统。而在不久之后的 2014 年 7 月，Go[illegible] 更是首次发布了开源的 **Kubernetes** 系统。

在 2016 年，Google 更是发表了一篇叫做《Borg, Omega, and Kubernetes: Lessons Learned From Three Container-Management Systems Over A Decade》的文章。这篇文章总结了十多年来，Google 从开发 Borg，到优化调度系统变成 Omega，以及最后重起炉灶开源 Kubernetes 中的经验与教训。

那么，今天我们就一起来了解下，在 2013 和 2016 年发表的这两篇论文到底讲了什么。虽然 Google 并没有开源 Borg 和 Omega，但是从开源的 Kubernetes 和发表的论文中，我们也多少能够一窥这些系统是如何一路走来的。我在这个课程的一开始就说过，我们学习这些论文和知识，重要的并不只是“是什么”“怎么做”的，更重要的是**“为什么”**。了解系统是如何逐渐演变过来的，你才能真的体会到如何在不断地迭代优化中设计系统。

而在学完这节课的内容之后，我希望我们能够一起思考这样两个问题：

> 为什么 Borg 的 Master 要从一个中央式的调度系统，变成 Omega 这样支持同时运行多个调度系统的设计？
>
> 除了资源的调度和隔离之外，我们的编排系统还需要考虑哪些问题？

因为 Borg 和 Omega 的系统都没有开源，虽然我们无法从源代码中去验证我们的思考和答案，不过，做一做这样的思维体操也是很有乐趣的事情，也有助于我们在未来的系统设计中，养成先思考后动手的好习惯。

压力山大的调度系统

其实在学习研究 Borg 的论文的时候，我的心里一直有两个疑惑。

第一个疑惑是，为什么在线服务和批处理任务，在 Borg 看来都是一个 Job，并且是放在一起调度的。Borg 既没有把在线服务和批处理任务拆分成 Service 和 Job，分别有不同的处理方式，也没有在 Borg 里实现两种调度器（Scheduler），分别去调度这两种不同类型的程序。

第二个疑惑是，Borg 是一个单 Master 系统，它真的能够做到去管理 1 万台服务器吗？如果我们有 Kafka、Hadoop、Flink 以及各种 Web Application 这样多个不同类型的在线服务，我们为什么不给里面的每一个集群，都设计一个 Master 去分配资源，然后再在 Borg 层面去管理这些 Master 就好了呢？

Borg 的 Master 只需要分配到某一个业务集群的资源，而业务集群自己再在内部进行资源分配。这样，Borg 的 Master 只需要和少数的服务集群的调度器交互，压力可以大大降低了。

其实这两个问题，在 Omega 这个调度系统的论文里，我们就能找到答案。因为我们的疑惑是正常的，**单一的 Master 的确成为了 Borg 的一个瓶颈**。

Borg 的调度系统，需要同时调度两种类型的任务。一种是**批处理任务**，另一种就是**需要长时间运行的在线服务**。

比如，一个 Nginx 服务器。这些服务，需要长时间在线运行，并且满足严格的可用性和性能指标。所以，Borg 调度的时候，既需要考虑资源分配的问题，还需要考虑让对应的容器调度到不同的服务器、交换机上，以确保硬件故障下的容错能力。而这个问题，是一个 NP-Hard 问题，也就是说，它没有在多项式时间内的解法。所以，在 Borg 里，调度这样一个在线服务，可能需要花上几十秒钟的计算时间。

一个在线运行的服务，往往要在线上运行几天乃至几个月，所以多花上个几十秒钟调度的确不是什么大问题。但问题是，一个 BI 分析师的一个 SQL 在 Borg 里同样也是一个 Job。而且在上节课里我们就说过，Borg 会按照**优先级**来进行 Task 的调度。

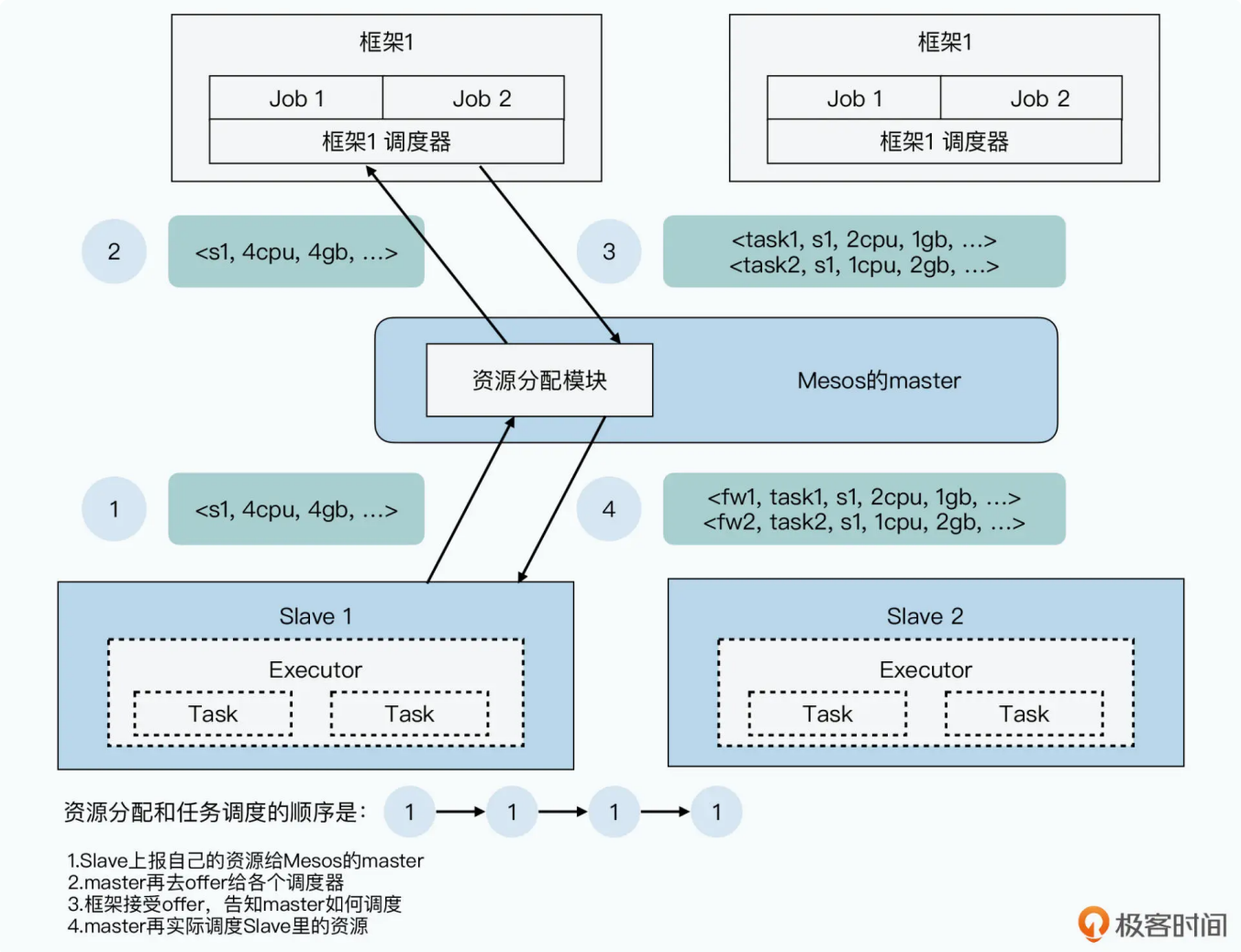
那么，如果同时有人向 Borg 集群提交了一个 Nginx Web 服务器的调度请求，这个 SQL 的 Job 就需要等待 Web 服务器调度完成。而这就意味着，我们的 SQL 虽然只需要 5 秒钟来实际运行，但是却要等待几十秒，去等待前面的调度完成。所以这个体验是很糟糕的，我们好不容易通过 Dremel 实现了列存储，缩短了 SQL 运行需要的时间，但是前端的 BI 分析师的体验却完全没有变化。

从解决这个问题的视角来看，**一个可行的方式就是我们需要多个调度器**。我们的一次性运行的 Job，不需要考虑太多的因素，因为毕竟它只运行几秒钟。最差情况下，无非对应的 Task 被抢占了，我们再重新运行一次就好了。

而一旦有了多个调度器，那么我们就要回答一个新的问题：在多个调度器之间，应该如何协调。因为它们毕竟调度的是同一组服务器。于是，在调度器层面，我们也面临在单机多线程程序类似的问题，就是**不能让两个 Job 被调度到同一份资源上**。

一种最笨的办法，当然是对集群进行静态分区。所谓静态分区，也就是我们让集群中的一部分机器，或者一部分资源专门给一种用途。比如，划出 5000 台服务器给在线服务，5000 台服务器给批处理任务；或者 1 万台机器，每台都留一半的 CPU/ 内存 / 磁盘 IO 来给在线服务，剩下的一半给批处理任务。

但是，这显然就违背了上节课我们了解过的 Borg，能够通过超卖以及抢占资源的方式，来提高集群利用率的思路了。



在Borg的论文发表之前，Mesos已经实现了一个两层动态分区的调度器

那么，针对静态分区的优化，自然就是**动态分区**。也就是，虽然我们对集群进行了功能层面的划分，但是每个集群分配的资源会动态调整。这样，当我们的在线服务的资源有空余的时候，就可以匀出一些给批处理任务。开源的 **Mesos 系统**就是采用这样的方案。

这个时候，整个集群的调度系统就和我之前提到的第二个疑惑类似，它分成了**两层**。第一层是 Master 在多个调度器之间分配资源，而第二层则是每个调度器自己在内部分配资源。

在 Mesos 里，第一层的分配是采用一种叫做 **Resource Offer** 的机制。也就是，资源并不是由下层的调度器提出申请的，而是由 Master 向集群主动提供的，调度器只能选择接受还是不接受。但是调度器自己，没有权力去抢占资源。

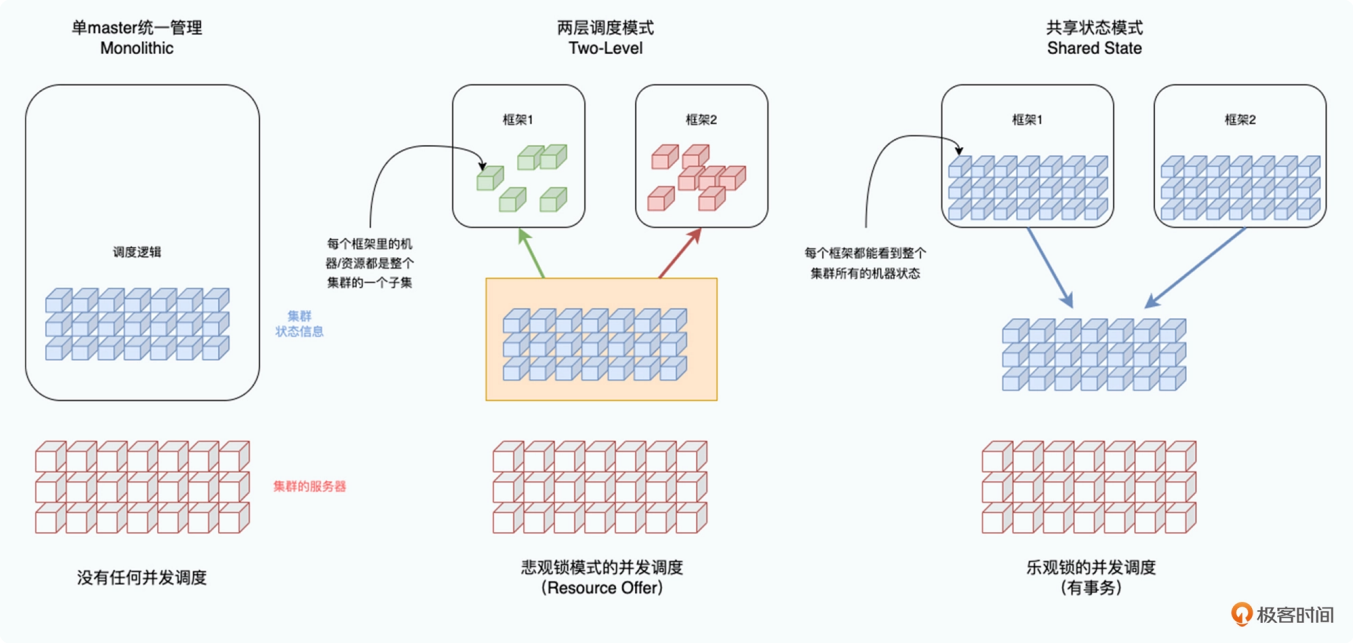
这种两层调度的模式，本质上类似于数据库里的悲观锁。下游的调度器，必须要先通过“锁”锁住资源，才能开始执行任务。但是，在 Borg 这个模式下，这个“锁”的动作可能就会占用很长时间。

前面我们刚提到过，Borg 的在线服务任务的调度，往往需要计算几十秒钟。在这几十秒里，因为我们部署的 Task 需要分配到不同的交换机、服务器下，而在调度程序计算完之前，我们又不知道到底应该是哪些交换机和服务器。于是，我们就不得不锁住大量我们可能用不上的资源。而这些资源，我们的批处理任务的调度器就拿不到了。

于是，虽然我们有了一个动态分区的、两层的调度系统，但是一旦实际用起来，和我们之前一层的系统效果并没有什么分别。

把 Master 调度器变成数据库

既然，预先锁住资源不是一个靠谱的办法，那么我们自然可以借鉴在数据库里用过的办法，也就是**把悲观锁换成乐观锁**。



参见论文里的图1，Omega采用了让两个框架/调度器通过事务进行资源竞争的模式

我们先让多个不同的调度器都能看到全局的资源，然后大家都可以去竞争同一份资源。两个调度器对自己 Job 的调度去抢占服务器资源，就好像并发的两个事务请求到了数据库要更新同一条记录一样。如果第一个成功了，那么第二个“事务”就失败了，需要重试。

不过，这个时候你可能会问：那我们的在线服务，每次调度都需要花很多时间计算，岂不是很容易在事务竞争中失败呢？这个其实没有关系。不要忘了，在 Borg 里，我们的在线服务，是可以去抢占批处理任务的资源的。即使批处理任务先拿到了对应的资源，我们仍然可以重试一次，把这个资源强行占为己有。而所有的相同优先级的在线服务，又都是使用同一个调度器，互相之间不会产生并发和竞争。

在这个模式下，虽然看似我们是两层调度器，但是，第一层其实已经不是一个调度器了，而更像一个**数据库**。只不过，在这个数据库里，除了基本的调度信息的读写之外，还需要确保 Borg 里一系列的调度分配规则限制，是始终有效的。

比如，我们只能超卖批处理任务，但是不能超卖生产任务等等。而各个调度器，就像是在并发读写这个数据库的程序。这个数据库里，实现了 **MVCC** 这样带版本信息的多副本机制，当我们去调度分配资源的时候，就是在这个数据库里发起了一个事务。

如果我们的事务执行成功了，调度也就完成了。对应的，数据库里相应“资源”的版本信息也就更新了。这个更新的信息，也会同步给另外一个调度器的副本里。而此时，如果另外一个调度器在尝试使用相同的资源，对应的调度事务也就会失败，需要重试。

Borg 没有说的那些事儿

无论是 Omega 还是 Borg，论文的核心关注点还是 Job 和 Task 的调度。但是，读论文是一回事儿，当我们到真实世界里去实现这样一个系统，其实会遇到更多的问题。这个，也是当 Kubernetes 开源之后，我们遇到的实际情况。

首先，在同一台服务器上有运行多个不同的应用，需要隔离的不仅仅是 CPU、内存这样的硬件资源，**还需要把运行的依赖环境隔离出来**。不然的话，我们就会遭遇到“DLL 地狱”这样的问题。

也许你的程序依赖的是 Python 2.7，而我的程序依赖的是 Python 3.5。即使是几个人用的一两台服务器，这也已经是一个需要解决的问题了。当这样的场景放到上万台服务器、几千个工程师共同协作的场景下，就成了一个大问题。

而对这个问题的解决方案，就是**容器的“环境隔离”**的作用。在我们目前使用的主流的 Kubernetes+Docker 的组合下，我们的容器还是一个“镜像”，里面会包含我们程序依赖的所有运行环境。如果是一个 Java 程序，那么我们的容器镜像里，会带上自己需要版本的 JVM；如果我们是一个 C++ 程序，那么对应的程序库，应该是通过静态链接编译在程序里，而不是通过动态链接调用外部的程序库。

其次，我们通过容器部署的这些**任务还需要进行“管理”**。

所有部署的任务，其实都需要一些公共的基础设施，比如 Borg 里说过的内置的 HTTP 服务器，让每个任务都可以对外暴露一系列的指标（Metrics），状态（Status），日志（Logs）。对于批处理任务，可能这些就够了。而对于在线服务来说，我们往往还需要**服务发现机制和负载均衡**。

一般来说，像 Web 应用这样的在线服务，往往是在一个负载均衡器后面，并且是可以水平扩容的。而我们的 Kubernetes 又可以在不同的服务器上去动态调度容器，就意味着容器的 IP 地址也会动态发生变化。不同的服务之间会相互调用，服务 A 需要调用服务 B，这也就意味着当 B 扩容的时候、IP 发生变化的时候，A 都需要能够感知到。

在没有 Kubernetes 这样的系统之前，其实各个公司都有一套自己的解决方案，这些方案往往也和自家的 RPC 框架绑定在一起。通过引入特定的函数库，每个应用都可以方便地实现日志、状态、指标。以及把自己的服务器注册在 ZooKeeper 上，或者通过 ZooKeeper 去获取到要调用服务器的 IP 地址。典型的，你就可以去参考 Twitter 的 Fingale这样的项目。

不过，这样的方案往往有比较强的“显式侵入性”，也就是你的应用程序，需要引入一个函数库，并要在自己的代码层面去调用这个函数库。而在 Kubernetes 的容器管理机制下，它通过每一个服务器节点上引入了一个 **kube-proxy**，来一次性解决了这个问题。你只需要去申明依赖的服务，Kubernetes 会帮助你找到依赖的服务在哪里，然后对应地把请求转发过去。

可以看到，一旦当我们把容器编排实际使用起来，它就不再是一个简单的资源调度问题，而变成了一个全功能的集群服务管理问题了。Kubernetes 不仅仅要提供资源的调度，还要把服务依赖、负载均衡、服务发现，乃至于每个应用方便 debug 的特性实现并考虑进来。

其实，光一个 Kubernetes，就足够开一门课程来做讲解了。毕竟，它对于我们的“大数据”来说，只是会用到的底层基建之一。如果你想要对 Kubernetes 深入学习，可以去专门看一下《深入剖析 Kubernetes》这个课程，我们在这里也不再太多聊这里的细节问题了。

小结

在 Borg 之后，Google 先是通过 Omega 优化了 Borg 的调度系统。基于单一 Master 的调度系统，在 Borg 管理 1 万台服务器的场景之下，会带来一个“阻塞”的问题。高优先级的在线服务类的任务，会被优先调度，但是由于要既要考虑资源分配，还需要考虑高性能和容错能力，整个调度算法会是一个 NP-Hard 的问题。单个任务的调度就会花上几十秒，这会阻塞很多交互式的查询任务。

所以，最终 Omega 采用了多个调度器，把在线服务和批处理任务，拆分到不同的调度器里。而各个调度器之间则采用了**乐观锁并发竞争**的方式，这个就使得我们不需要为了一个在线服务，长时间锁住大量的资源，从而让整个集群的调度，变得更加可扩展了。

而当我们从 Borg 和 Omega 这些论文的理论中，进入到 Kubernetes 这样的实际系统之后，需要解决的问题就很多了。

一方面，我们的容器不只是进行资源层面的隔离，而是要把整个运行环境也一并隔离出来，避免“**DLL 地狱**”。除此之外，我们还需要考虑**负载均衡、服务发现**等一系列的问题。所以，Kubernetes 也一并通过 metadata、spec 和 status 的统一 API，来解决了这个问题。

所以，Kubernetes 并不是一个调度器，而是被叫做容器“编排”系统。

而即使到今天，对于服务和服务之间的依赖，Kubernetes 的方案离完美也还差得远。这也是为什么，Kubernetes 本身作为一个系统仍然在快速进化的原因。

推荐阅读

如果你对于大型分布式集群的调度问题，还想要深入研究的话，那么来自 Berkeley 的这一篇《Multi-agent Cluster Scheduling for Scalability and Flexibility》应该是一份很好

的阅读学习材料。这篇 Survey，总结了各种类型的分布式调度系统，里面也对未来的分布式调度工作的优化指出了方向。

思考题

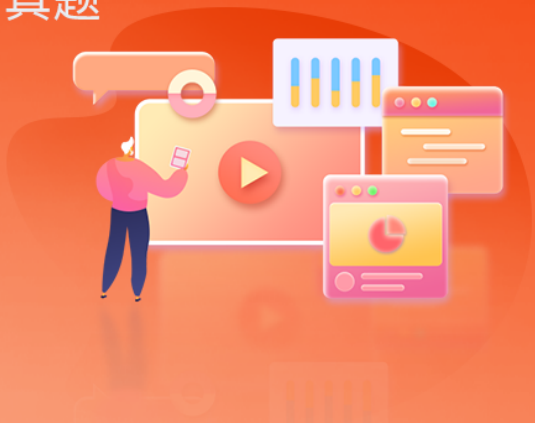
到今天这节课为止，我们就已经把要讲解的各个系统的论文讲解完了。后面的课程中，我们会来一起探索来自 Twitter 和 Facebook 这两家公司公开发表的实战经验。

那么，按照惯例，我还是给你留一个思考题。在过去这么多节课里，你觉得哪些大数据系统已经相对完善了，而哪些系统后续还有较大的改进空间，这个改进需求源自哪里呢？

欢迎在留言区说说你的想法，也欢迎你把今天的内容分享给更多的朋友。

上一篇 35 | Borg（二）：互不"信任"的调度系统

下一篇 37 | 当数据遇上AI，Twitter的数据挖掘实战（一）



精选留言 (1)

写留言

在路上

2022-01-17

徐老师好，批处理能够高吞吐地处理历史数据，但是对即席查询而言有点慢。流式处理能够低延迟地处理实时数据，但是需要先写逻辑再消费数据。Dremel查询性能好，但是它的数据文件采用列式存储，不可修改，数据难以以分钟级别的延迟落到数据库中。我认为实时数据湖是一个方向，数据以低成本、低延迟的方式落入湖中，支持快速查询、快速分析。

展开