34 | Borg（一）：当电力成为成本瓶颈

2022-01-07 徐文浩

《大数据经典论文解读》 课程介绍 >

讲述：徐文浩

时长 14:20 大小 13.14M

你好，我是徐文浩。

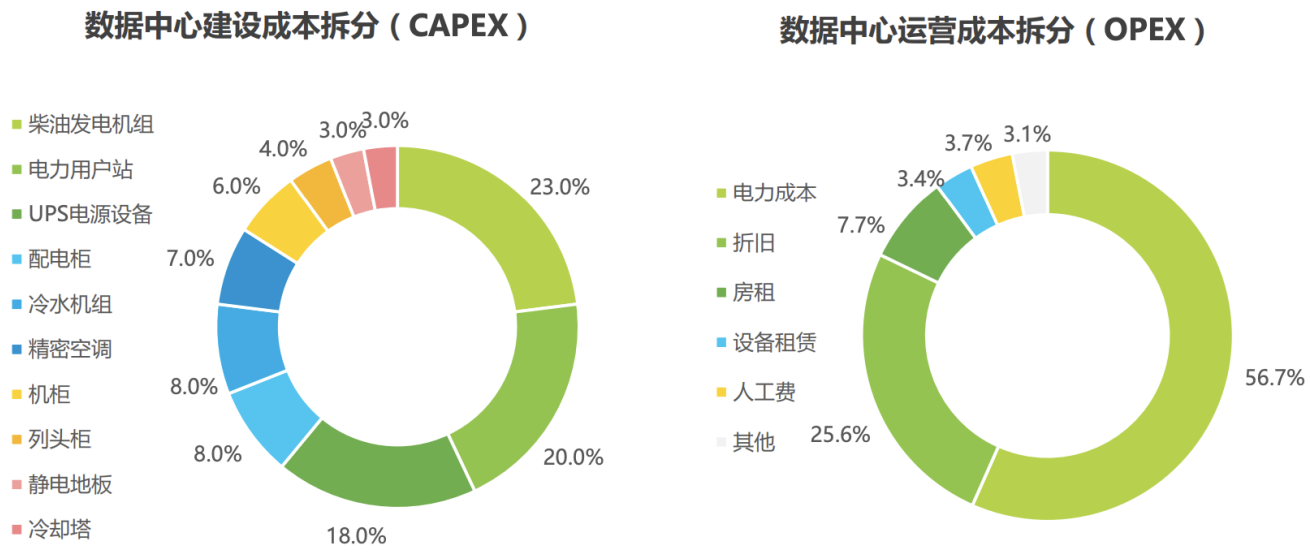
从 GFS 这样的分布式文件系统，到 MapReduce 这样的数据批处理系统；从 Bigtable 这样的分布式 KV 数据库，到 Spanner 这样全球部署的强一致性关系数据库；从 Storm 这样只能做到“至少一次”的流式系统，到 Dataflow 这样真正做到“流批一体”的统一数据处理系统。在过去的 30 多讲里，我和你一起看过了各式各样的大数据系统。

在研究这些大数据系统的时候，我们其实有一个假设。这个假设，就是其中的每一个系统，都需要占用一组独立的服务器。而在一个完整的大数据体系中，我们既需要有 GF 样的文件系统，也需要 MapReduce/Spark 这样的批处理系统，还需要 Bigtable 这样的 KV 数据库、Hive 这样的数据仓库、Kafka 这样的消息队列，以及 Flink 这样的流式系统。这样一算，我们需要的服务器可真不算少。

成本 - 混合编排的需求起源

但是，当我们采购了很多服务器，搭建起了一系列的大数据系统，我们又会发现这些服务器在很多时候负载不高，显得非常浪费。因为我们在采购服务器的时候，需要根据平时的峰值流量来确定服务器数量。比如，像 Kafka 这样的消息队列，肯定是在早晚高峰，和中午用户比较多的时候，流量比较大，到了半夜流量就比较小。如果我们高峰时间的 CPU 占用要有 60%，那么在低谷时刻，可能只有 10%。

如果就只是采购服务器的硬件成本，那还好，毕竟服务器我们已经采购完了。但是，对于一个数据中心来说，硬件成本只是一小部分，最大的一头在电力成本。根据艾瑞发表的「2020 年中国数据中心行业研究报告」，数据中心的运营成本，有超过 50% 是电力成本。所以，能不能尽量少用一点服务器，就变成一个很有价值的问题了。



上面的研究报告中，引用自Bloomberg的数据，数据中心的成本大头来自于电费

一个很自然的想法，就是对我们服务器的使用进行“削峰填谷”，我们让原本在高峰时间运行的一些任务，挪到半夜这样的低谷时段去。这个思路，对于离线进行数据分析的任务，当然很容易做到，所以一般来说，我们的 Hadoop 或者 Spark 集群的 CPU 整体利用率往往很高。

但是对于 Kafka、Dataflow 这样提供近实时服务的数据系统，我们是没办法把峰值时段的任务，也挪到第二天半夜才计算的。所以，一个新的想法自然也就冒出来了，那就是，我们是不是可以把 MapReduce 这样的离线分析任务，也放到 Dataflow 的集群上运行呢？在半夜 Dataflow 没有什么流量的时候，我们完全可以把这部分服务器的资源用起来。

所以，我们需要有一个办法，能够让我们的各种大数据系统混合编排在同一批服务器上。

事实上，这个混合编排并不局限于大数据系统，我们的业务系统也可以一并考虑进来，毕竟半夜里，我们业务系统的访问量肯定也很小，对应的服务器资源一样是闲置着的。

正是这么一个朴素的、尽量利用服务器资源的思路，就催生了 Google 的 Borg 系统，并最终从中进化出了 Kubernetes 这个开源的容器编排系统。如今，几乎所有的公司都在把系统往容器化的方向迁移，Kubernetes 也成为了云原生系统的标准。那么今天，我就带你一起来看看，这篇 Kubernetes 的老祖宗的论文《Large-scale cluster management at Google with Borg》是怎么一回事儿。

通过这节课的学习，希望你能先对 Borg 系统的基础概念、系统架构有所认识和了解。之后，我会再花一节课时间，来具体为你讲解 Borg 是如何进行资源的调度分配，以及评估整个系统的性能的。

好了，废话少说，让我们接着往下学习吧。

CGroups-Linux 下隔离资源的解决方案

把不同类型的程序，部署在同一台服务器上，我们面临的第一个问题，就是这两个程序会**竞争资源**。比如，我们在同一台服务器上，既部署了 Kafka 的 Broker 进程，也部署了 Bigtable 的 Tabet Server 进程。那么，当我们在流量高峰的时候，Kafka 传输的流量占满了整个网络带宽之后，我们的 Tablet Server 就无法对外提供服务了，所有的请求都会因为超时而失败。

要解决这个问题，本质上，我们需要把服务器的资源拆开来，然后把对应的一组应用程序隔离出来，只允许它们去使用服务器的一部分资源。而这个解决方案，就是 Linux 下的**CGroup 功能**。CGroups的全称叫做 Linux Control Group，它可以限制一组 Linux 进程使用的资源。具体来说，它主要能做到这样几点：

资源限制，比如限制一组进程总共可以使用的内存。

优先级，比如限制这组进程能够拿到的 CPU 和 I/O 的吞吐量。

结算，也就是统计这组进程实际使用了多少资源。

控制，可以冻结一组进程的运行，或者反过来恢复它们的运行。

这么一看你就清楚了，有了 CGroups，我们就可以把 Kafka 相关的进程分成一组，然后交给 CGroups 去限制它占用的内存、CPU、硬盘 I/O 和网络 I/O，然后 Tablet Server 分成另一组，也限制了对应使用的资源。这样，我们就可以在一个服务器上，同时运行两个不同系统的程序，而且不用相互干扰了。

对于 CGroups 的具体功能，你可以去这里看看简单的一些运行示例，有个直观的体会。事实上，Borg 里部署的一个个的任务，其实就是基于 CGroups，封装好的 LXC 的容器。如果你有兴趣的话，也可以去仔细研究一下 LXC 和 CGroups 相关的知识。

Borg：典型的 Master-Slave 系统

我们的各种系统的进程都被封装进了一个个的 LXC 容器，这样我们就可以通过一个集群管理系统，把不同的 LXC 容器分配到不同的服务器上运行，这个集群管理系统，就是我们的 Borg。

从用户视角看 Borg

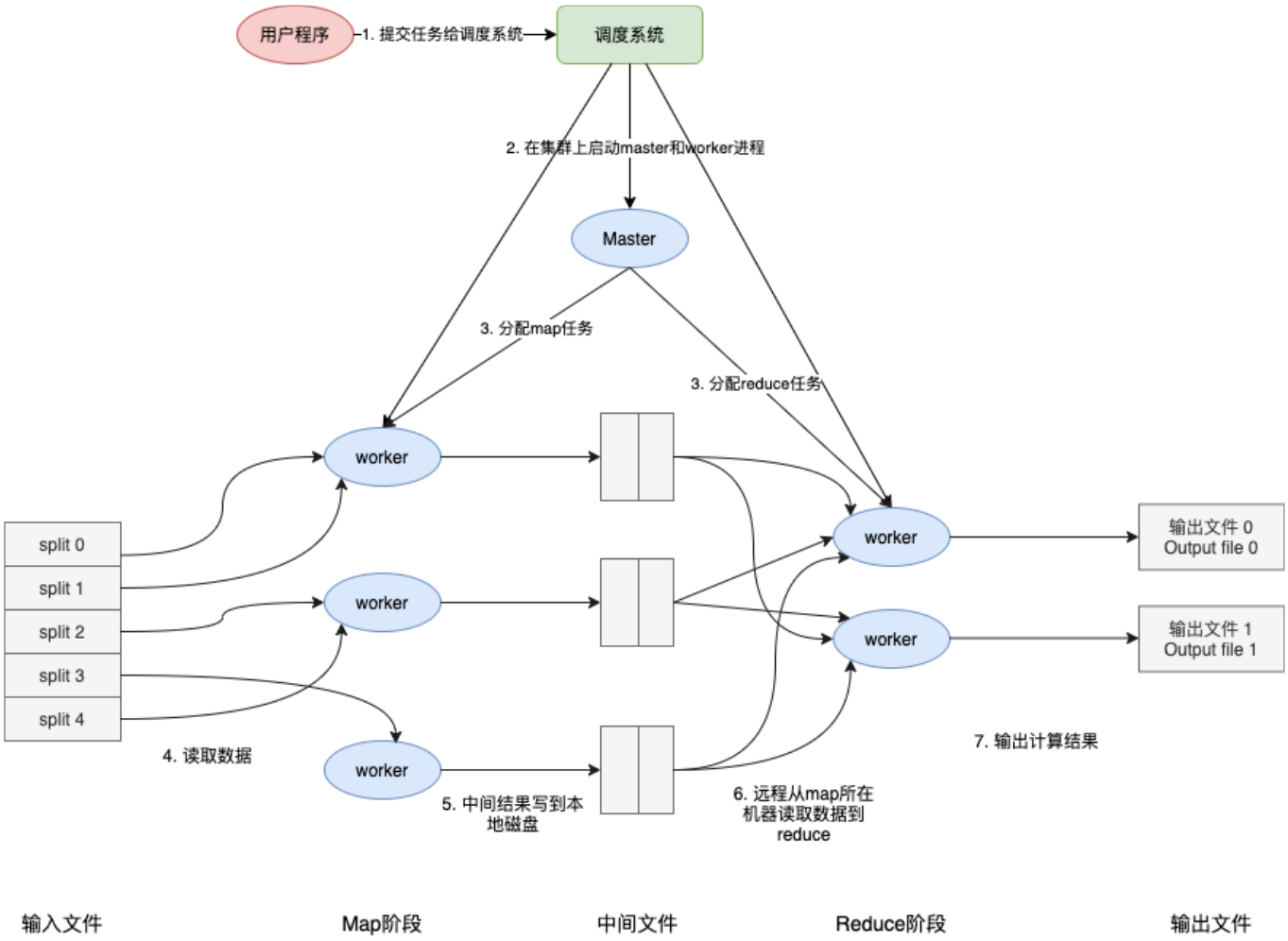
一个 Borg 的集群被称之为一个 **Cell**，通常被部署在一个数据中心里。而在 Google，一个中等规模的 Cell 通常有 1 万台机器。

一个集群有这么多服务器，意味着 Borg 的“混合部署”，并不是简单地把两三个应用放在一起，让它们能够错开各自的性能峰值，而是把一个数据中心里，成千上万个应用混合部署在一起。而且，这么多的服务器，往往是分多个批次采购的，也就意味着服务器的配置，比如 CPU、内存、硬盘等等可能是各不相同的。

而 Borg，需要对于各个应用的开发人员，做到隐藏所有的资源管理细节，让应用不需要关心它们具体会被部署在什么样的硬件配置上。而且，即使当前部署的硬件故障了，Borg 也需要能够立刻快速恢复，把对应的容器另外寻找一台服务器，部署上去。

当然，在 Borg 里，部署的并不是一个裸的 CGroup，而是一个个的 **Tasks**。用户会向 Borg 提交一个个的 Job。这些 Job 里，既有像长期驻留的服务，也有一次性运行批处理任务。**一个 Job 其实就是一个二进制的程序**，这个 Job 程序会被部署到一台或者多台服务器上。每个服务器上实际运行的进程就叫做 Tasks。

这个其实我们在MapReduce 的论文讲解里就看到过了。提交的 MapReduce 的 Job，就是先提交给了 Borg，然后 Job 对应的 Map 和 Reduce 的 Tasks，其实就是由 Borg 分配到不同的服务器上，实际执行的。



MapReduce的系统架构图

其实你可以把worker看成是alloc，Job还是MapReduce的Job，map和reduce的Task就是Borg里的Task

那么，在这 1 万台机器的集群里的 Tasks，有的是提供在线服务的，对于服务的响应延时有要求；有的只是一个离线任务，只要能够执行完成就好。而我们为了尽可能充分利用硬件资源，就会遇到某一个时间点，很多 Tasks 都想要使用资源但是资源不够用了的情况。

针对这个问题，Borg 采用了**配额和优先级的机制**。

和我们操作系统里的进程类似，Borg 里的 Job 都有一个优先级，从高到低分别是监控（Monitoring），生产（Production），批处理（Batch）和尽最大努力（Best Effort）这四种，同一种类的优先级下，还能根据一个整数的 priority 参数，来区分不同任务的优先级大小。Job 提交之后部署到具体的服务器上，就会变成一个个 Task，这些个 Task 也就继承了 Job 上的优先级的属性。

生产类型的 Tasks，可以去抢占批处理 Tasks 的资源，但是生产类型的 Tasks 互相抢占资源是不允许的，因为大家都要提供在线服务。不过，这个时候问题来了，像 MapReduce 这样的 Job，自然应该归在批处理 Job 里，对应的 Task 按理也都是随时会被抢占的批处理 Task。

但是，我们同样会有很多 MapReduce 的 Job，需要在规定的时间范围内完成。比如，我们可能每天半夜里都会利用 MapReduce，来进行过去一天的报表计算，我们需要这些任务在早上 8 点大家都来上班之前完成。但是，如果批处理的 Task，随时会被生产类型的 Task 抢占，我们就有可能无法保障这一点。

为了解决这一类的问题，Borg 有一种叫做 **alloc** 的机制。alloc 是一组可以预留的资源，不管这个服务器资源是否被用到了，这个资源我们始终会分配给 alloc。这样，对于 MapReduce 类型的任务，我们可以始终通过 alloc，在我们的 Borg 集群里预留一部分资源，这些资源是其他生产类型的 Job 抢不走的。

在 Borg 里，你可以向一个 alloc 集合来提交你的 Job，这个 Job 对应拆分出来的 Task，就会使用 alloc 预留的资源运行，这些资源是不会被其他的生产 Task 抢占走的。

Borg 的系统架构

好了，我们之前已经看过了太多的分布式系统，相信你想一想也能差不离地猜出 Borg 的整体架构是怎么样的了。

Borg 是一个典型的 **Master-Slave** 类型的系统。一个 Borg Cell 通常由这样几部分组成。

首先是**用户界面**，你可以通过配置文件、命令行以及浏览器内的 Web 界面，向 Borg 发起请求，所有这些请求都会统一发送给 Borg 的 Master。和我们之前看过的 MapReduce 和 Hive 一样，Borg 的 UI 界面里，同样有一个对于所有 Job 和 Task 的监控系统。每一个直接部署执行的 Task，都会内置一个 HTTP 服务器，你通过浏览器，就能拿到 Job 和 Task 的各种监控指标和运行日志，这个系统在 Borg 里叫做 Sigma。

然后就是一个 **Master 集群**，为了保障高可用性，Master 的所有数据当然是通过 Paxos 协议来维护多个同步复制的副本的。它一样，也是通过 Checkpoint 来建立快照，通过日志记录所有的操作，整个 Master 的集群里，也有一个选举出来的 master 中的 master。而这个 master 中的 master，则是通过获取一个 Chubby 里的锁来确保唯一性。

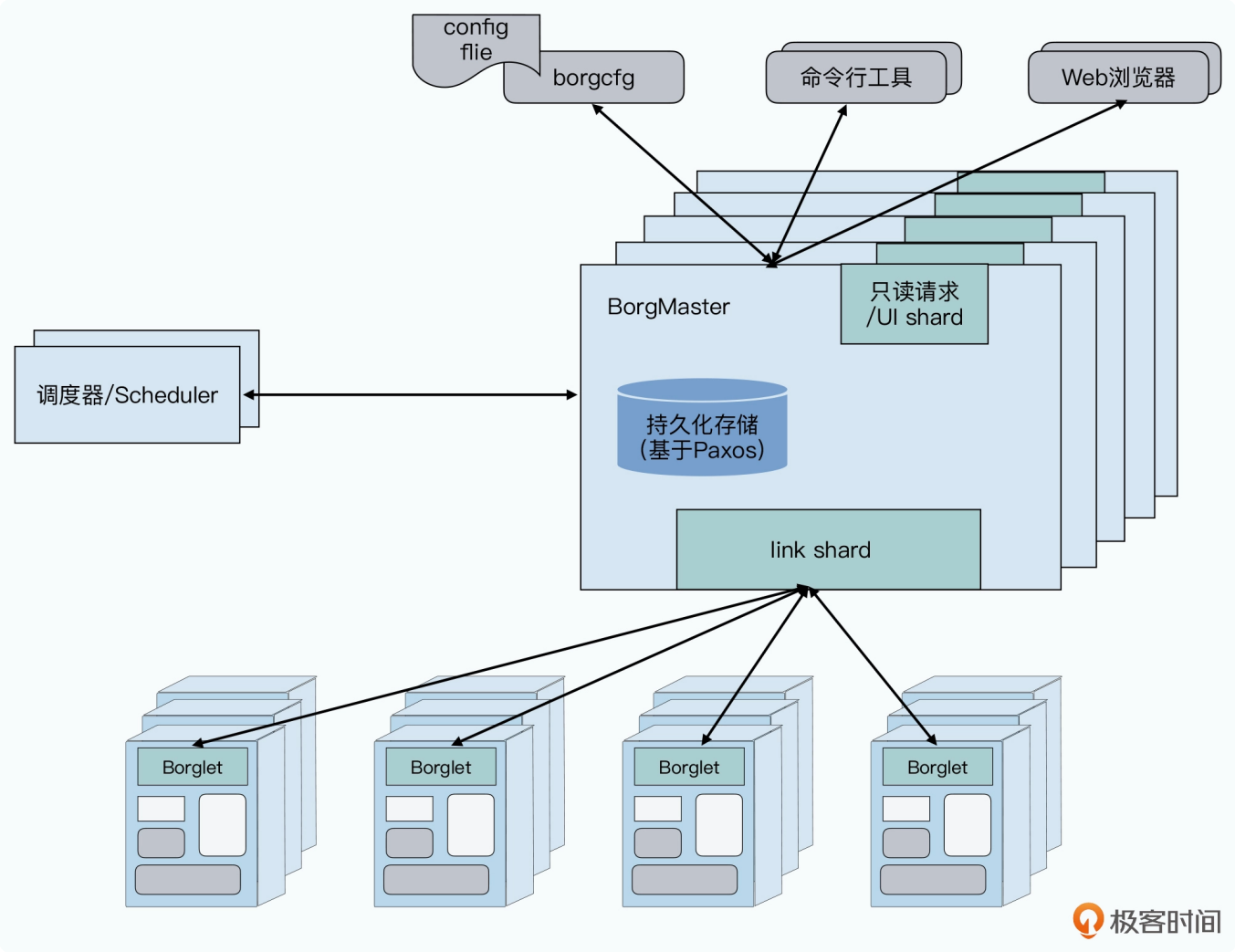
这一系列的设计是一个典型的 Master-Slave 分布式系统的设计，我们之前在 GFS/Bigtable 这样的系统里已经反复看到过了。Master 集群只负责管理所有的元数据，以及和外部发来的 RPC 请求。这里的元数据，包括集群里所有对象的状态，也就是集群本身的资源，也就是有哪些 Borglet 服务器，有哪些 Job、Task 以及 Allocs。然后，它也要和 Borglets 通信，确保自己了解整个集群的状态。

不过，实际 Task 分配给哪个服务器执行，却不是由 Master 来决定的。Borg 把这部分职责单独从 Master 里剥离出来，给了一个叫做 **Scheduler 的服务器**。当一个 Job 提交给 Master 之后，Master 会把它变成一个个有待调度的 Tasks，然后加入到一个队列里。而我们的 Scheduler 则会异步遍历这些 Tasks，当它判断我们的集群有足够的资源，可以满足整个 Job 的需求的时候，它就会把 Task 分配到 Slave 服务器里。

Slave 服务器，就是我们实际负责去运行 Task 的服务器。每个 Slave 服务器上，都会有一个叫做 Borglet 的进程。Master 并不会直接和每个 Slave 服务器上的 Task 进程通信，而只是和 Borglet 进程通信。

Borglet 进程会负责启动和停止 Task，如果 Task 失败了就重启。因为 Borg 的单个集群，要有 1 万台服务器，所以，并不是由 Master 一台台机器来主动拉 Slave 里的信息的。而是由 Borglet 上报对应的信息，Master 只会**定期轮询**Borglet，获取这些 Slave 服务器的状态。

而因为有 1 万台之多的服务器，Master 集群里虽然选出了一个 master（本质上是 leader），Borg 还是会把所有的 Slave 机器分片，然后让每一个 master 的副本，都负责一部分 Borglet 的通信。然后，这个副本会把 Borglet 上报的最新信息，和 Master 已经知道信息的差（Diff），交给 Master 里的 master，以减少这一台特定机器的负载。



论文中的图1，Borg的整体架构

可以看到，从整个集群的架构来说，Borg 并没有什么特殊之处。Master-Slave 的架构我们在 MapReduce/Bigtable 里都已经见过。对外提供命令行、Web 的一系列的 UI，我们在 Hive 里面也见过。每个 Slave 服务器里，通过一个 Borglet 进程来管理，我们在 Megastore 的 Coordinator 里还是见过。

其实，**分布式系统的架构设计都是大同小异的**。不过，对于 Borg 来说，最有挑战的一点，还是在于如何去分配和调度一个个的 Task，以及如何衡量一个混合部署了成百上千个 Job 的系统的性能。而这些问题，我们会在下节课里进行解答。

小结

其实 Borg 系统整体并不难理解。为了尽可能地利用服务器资源，我们需要能够让不同性质系统的进程部署到一整个大集群里，通过一个集群管理系统来统一管理。而为了做到这一点，我们先要能够对同一台服务器上的一组进程打包起来，进行资源的限制和权限的隔离，这个功能是来自于 Linux 内核的 CGroup。

在有了 CGroup 之后，Borg 系统的整体架构并不复杂，是一个典型的 Master-Slave 系统。它通过 Paxos 实现 Master 的高可用性和一致性，通过 Borglet 来管理每个服务器上的 Tasks，而具体的 Tasks，则不需要依赖 Borg 本身的 Master 和 Borglet 来执行。这一系列的设计，和我们之前看过的那么多的分布式系统，并没有本质上的区别。

在任务的调度上，Borg 是把任务按照优先级，从高到低分成了监控任务、生产任务、批处理任务，以及尽最大努力完成的任务。生产级的任务可以抢占非生产级的任务，而 Borg 也添加了预留资源的 alloc 机制，使得 MapReduce 这样都是批处理的任务，也都能长期保留一些资源，确保任务能够按时完成。

我们可以看到，在整体架构上，Borg 并没有什么出人意料的思路。不过在实践的工程上，能够管理 1 万台集群的服务器，的确让人叹为观止。可惜 Google 并没有把 Borg 这样的系统开源出来，我们也就没有办法深入到代码层面去一探究竟了。

不过，好在在论文里，Google 还是对他们如何管理调度集群里的任何资源，以及如何评估混合部署，为我们的系统带来的性能影响进行了说明。而这些，也是我们下节课的主题。

推荐阅读

Borg 能够有效运转的前提，是通过 Linux 下的 CGroups，对我们运行的程序进行资源层面的限制。你可以去看看美团技术团队的博客中，关于CGroups的介绍，也可以看看酷壳上的这篇博客，可以手把手体验一下 CGroups 的这些功能。

思考题

Borg 的一个中等规模的 Cell 就会有 1 万台服务器。你能想一下，比起 100 台服务器的集群管理系统，1 万台服务器会给我们带来哪些额外的挑战吗？

欢迎在留言区分享你的思路，也欢迎把今天的内容分享给更多的朋友。

赞 0 提建议

上一篇 33 | Raft（二）：服务器增减的“自举”实现

下一篇 35 | Borg（二）：互不“信任”的调度系统

精选留言 (2)

写留言

csyangchsh
2022-01-10

1 万台服务器的额外挑战能想到的有两个：网络和机器故障。

那时刻
2022-01-07

请问老师，CGroup进行资源的限制和权限的隔离，而namespace也有权限的隔离，它们的区别是什么呢？