41 | 十字路口上的Kubernetes默认调度器

41 | 十字路口上的Kubernetes默认调度器

张磊 2018-11-26

09:51

讲述：张磊 大小：4.52M

你好，我是张磊。今天我和你分享的主题是：十字路口上的 Kubernetes 默认调度器。

在上一篇文章中，我主要为你介绍了 Kubernetes 里关于资源模型和资源管理的设计方法。而在今天这篇文章中，我就来为你介绍一下 Kubernetes 的默认调度器（default scheduler）。

在 Kubernetes 项目中，默认调度器的主要职责，就是为一个新创建出来的 Pod，寻找一个最合适的节点（Node）。

而这里“最合适”的含义，包括两层：

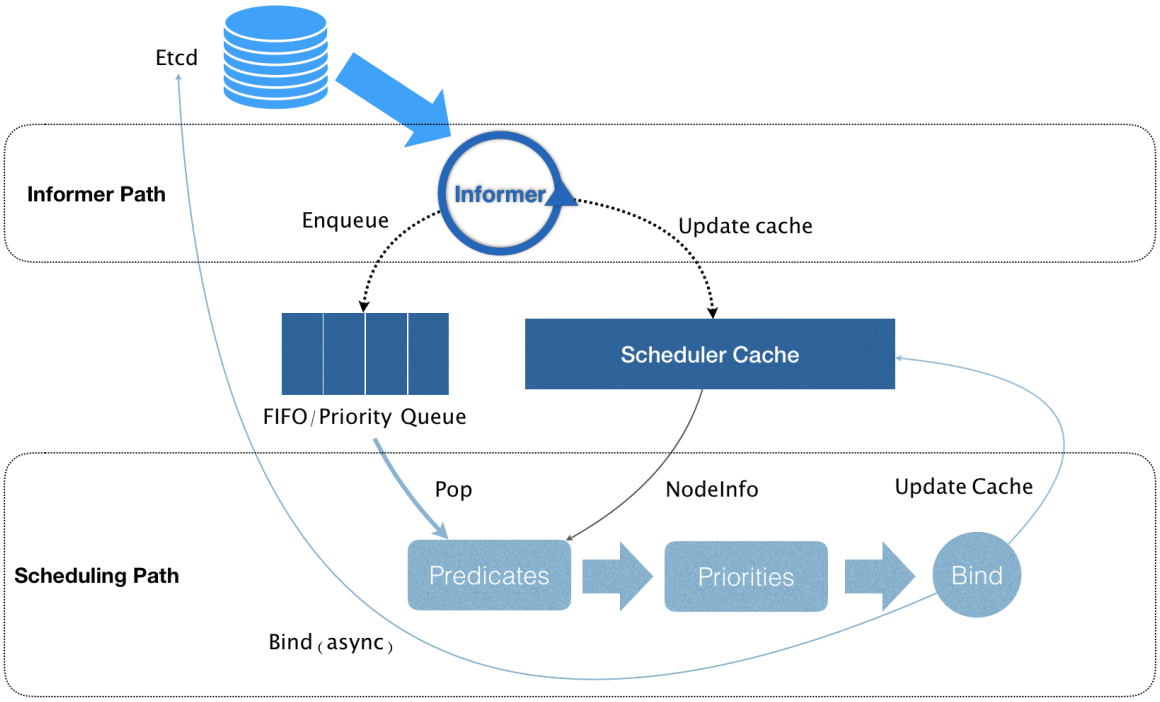
1. 从集群所有的节点中，根据调度算法挑选出所有可以运行该 Pod 的节点；
2. 从第一步的结果中，再根据调度算法挑选一个最符合条件的节点作为最终结果。

所以在具体的调度流程中，默认调度器会首先调用一组叫作 Predicate 的调度算法，来检查每个 Node。然后，再调用一组叫作 Priority 的调度算法，来给上一步得到的结果里的每个 Node 打分。最终的调度结果，就是得分最高的那个 Node。

而我在前面的文章中曾经介绍过，调度器对一个 Pod 调度成功，实际上就是将它的 spec.nodeName 字段填上调度结果的节点名字。

> *备注：这里你可以再回顾下第 14 篇文章《深入解析 Pod 对象（一）：基本概念》中的相关内容。*

在 Kubernetes 中，上述调度机制的工作原理，可以用如下所示的一幅示意图来表示。



可以看到，Kubernetes 的调度器的核心，实际上就是两个相互独立的控制循环。

其中，**第一个控制循环，我们可以称之为 Informer Path**。它的主要目的，是启动一系列 Informer，用来监听（Watch）Etcd 中 Pod、Node、Service 等与调度相关的 API 对象的变化。比如，当一个待调度 Pod（即：它的 nodeName 字段是空的）被创建出来之后，调度器就会通过 Pod Informer 的 Handler，将这个待调度 Pod 添加进调度队列。

在默认情况下，Kubernetes 的调度队列是一个 PriorityQueue（优先级队列），并且当某些集群信息发生变化的时候，调度器还会对调度队列里的内容进行一些特殊操作。这里的设计，主要是出于调度优先级和抢占的考虑，我会在后面的文章中再详细介绍这部分内容。

此外，Kubernetes 的默认调度器还要负责对调度器缓存（即：scheduler cache）进行更新。事实上，Kubernetes 调度部分进行性能优化的一个最根本原则，就是尽最大可能将集群信息 Cache 化，以便从根本上提高 Predicate 和 Priority 调度算法的执行效率。

而**第二个控制循环，是调度器负责 Pod 调度的主循环，我们可以称之为 Scheduling Path。**

Scheduling Path 的主要逻辑，就是不断地从调度队列里出队一个 Pod。然后，调用 Predicates 算法进行“过滤”。这一步“过滤”得到的一组 Node，就是所有可以运行这个 Pod 的宿主机列表。当然，Predicates 算法需要的 Node 信息，都是从 Scheduler Cache 里直接拿到的，这是调度器保证算法执行效率的主要手段之一。

接下来，调度器就会再调用 Priorities 算法为上述列表里的 Node 打分，分数从 0 到 10。得分最高的 Node，就会作为这次调度的结果。

调度算法执行完成后，调度器就需要将 Pod 对象的 nodeName 字段的值，修改为上述 Node 的名字。**这个步骤在 Kubernetes 里面被称作 Bind。**

但是，为了不在关键调度路径里远程访问 APIServer，Kubernetes 的默认调度器在 Bind 阶段，只会更新 Scheduler Cache 里的 Pod 和 Node 的信息。**这种基于“乐观”假设的 API 对象更新方式，在 Kubernetes 里被称作 Assume。**

Assume 之后，调度器才会创建一个 Goroutine 来异步地向 APIServer 发起更新 Pod 的请求，来真正完成 Bind 操作。如果这次异步的 Bind 过程失败了，其实也没有太大关系，等 Scheduler Cache 同步之后一切就会恢复正常。

当然，正是由于上述 Kubernetes 调度器的“乐观”绑定的设计，当一个新的 Pod 完成调度需要在某个节点上运行起来之前，该节点上的 kubelet 还会通过一个叫作 Admit 的操作来再次验证该 Pod 是否确实能够运行在该节点上。这一步 Admit 操作，实际上就是把一组叫作 GeneralPredicates 的、最基本的调度算法，比如：“资源是否可用”“端口是否冲突”等再执行一遍，作为 kubelet 端的二次确认。

> *备注：关于 Kubernetes 默认调度器的调度算法，我会在下一篇文章里为你讲解。*

除了上述的“Cache 化”和“乐观绑定”，Kubernetes 默认调度器还有一个重要的设计，那就是“无锁化”。

在 Scheduling Path 上，调度器会启动多个 Goroutine 以节点为粒度并发执行 Predicates 算法，从而提高这一阶段的执行效率。而与之类似的，Priorities 算法也会以 MapReduce 的方式并行计算然后再进行汇总。而在这些所有需要并发的路径上，调度器会避免设置任何全局的竞争资源，从而免去了使用锁进行同步带来的巨大的性能损耗。

所以，在这种思想的指导下，如果你再去查看一下前面的调度器原理图，你就会发现，Kubernetes 调度器只有对调度队列和 Scheduler Cache 进行操作时，才需要加锁。而这两部分操作，都不在 Scheduling Path 的算法执行路径上。

当然，Kubernetes 调度器的上述设计思想，也是在集群规模不断增长的演进过程中逐步实现的。尤其是 **“Cache 化”，这个变化其实是最近几年 Kubernetes 调度器性能得以提升的一个关键演化。**

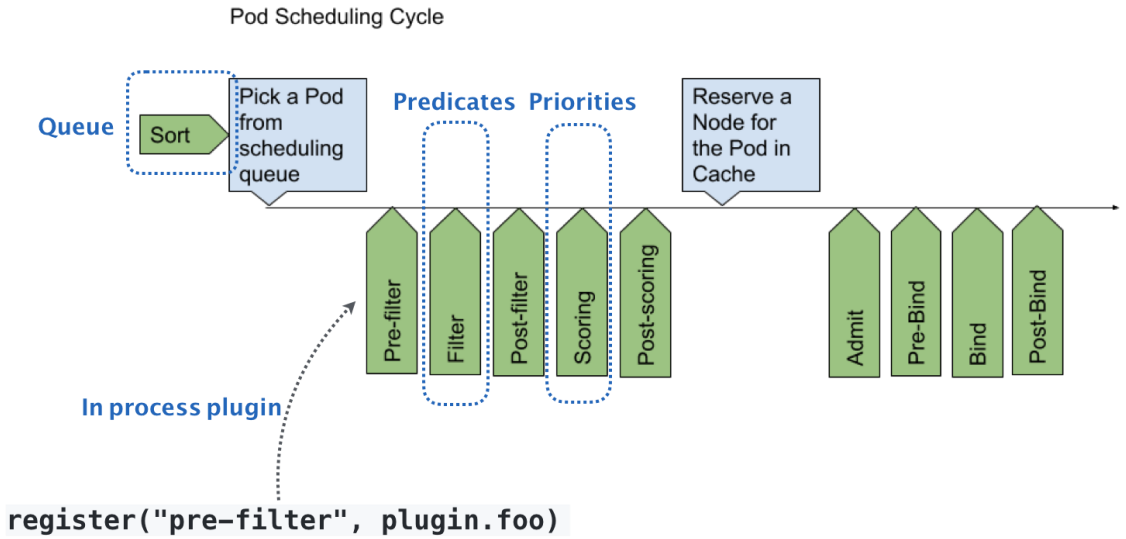
不过，随着 Kubernetes 项目发展到今天，它的默认调度器也已经来到了一个关键的十字路口。事实上，Kubernetes 现今发展的主旋律，是整个开源项目的“民主化”。也就是说，Kubernetes 下一步发展的方向，是组件的轻量化、接口化和插件化。所以，我们才有了 CRI、CNI、CSI、CRD、Aggregated APIServer、Initializer、Device Plugin 等各个层级的可扩展能力。可是，默认调度器，却成了 Kubernetes 项目里最后一个没有对外暴露出良好定义过的、可扩展接口的组件。

当然，这是有一定的历史原因的。在过去几年，Kubernetes 发展的重点，都是以功能性需求的实现和完善为核心。在这个过程中，它的很多决策，还是以优先服务公有云的需求为主，而性能和规模则居于相对次要的位置。

而现在，随着 Kubernetes 项目逐步趋于稳定，越来越多的用户开始把 Kubernetes 用在规模更大、业务更加复杂的私有集群当中。很多以前的 Mesos 用户，也开始尝试使用 Kubernetes 来替代其原有架构。在这些场景下，对默认调度器进行扩展和重新实现，就成了社区对 Kubernetes 项目最主要的一个诉求。

所以，Kubernetes 的默认调度器，是目前这个项目里为数不多的、正在经历大量重构的核心组件之一。这些正在进行的重构的目的，一方面是将默认调度器里大量的“技术债”清理干净；另一方面，就是为默认调度器的可扩展性设计进行铺垫。

而 Kubernetes 默认调度器的可扩展性设计，可以用如下所示的一幅示意图来描述：



可以看到，默认调度器的可扩展机制，在 Kubernetes 里面叫作 Scheduler Framework。顾名思义，这个设计的主要目的，就是在调度器生命周期的各个关键点上，为用户暴露出可以进行扩展和实现的接口，从而实现由用户自定义调度器的能力。

上图中，每一个绿色的箭头都是一个可以插入自定义逻辑的接口。比如，上面的 Queue 部分，就意味着你可以在这一部分提供一个自己的调度队列的实现，从而控制每个 Pod 开始被调度（出队）的时机。

而 Predicates 部分，则意味着你可以提供自己的过滤算法实现，根据自己的需求，来决定选择哪些机器。

需要注意的是，上述这些可插拔式逻辑，都是标准的 Go 语言插件机制（Go plugin 机制），也就是说，你需要在编译的时候选择把哪些插件编译进去。

有了上述设计之后，扩展和自定义 Kubernetes 的默认调度器就变成了一件非常容易实现的事情。这也意味着默认调度器在后面的发展过程中，必然不会在现在的实现上再添加太多的功能，反而还会对现在的实现进行精简，最终成为 Scheduler Framework 的一个最小实现。而调度领域更多的创新和工程工作，就可以交给整个社区来完成了。这个思路，是完全符合我在前面提到的 Kubernetes 的“民主化”设计的。

不过，这样的 Scheduler Framework 也有一个不小的问题，那就是一旦这些插入点的接口设计不合理，就会导致整个生态没办法很好地把这个插件机制使用起来。而与此同时，这些接口本身的变更又是一个费时费力的过程，一旦把控不好，就很可能会把社区推向另一个极端，即：Scheduler Framework 没法实际落地，大家只好都再次 fork kube-scheduler。

总结

在本篇文章中，我为你详细讲解了 Kubernetes 里默认调度器的设计与实现，分析了它现在正在经历的重构，以及未来的走向。

不难看到，在 Kubernetes 的整体架构中，kube-scheduler 的责任虽然重大，但其实它却是在社区里最少受到关注的组件之一。这里的原因也很简单，调度这个事情，在不同的公司和团队里的实际需求一定是大相径庭的，上游社区不可能提供一个大而全的方案出来。所以，将默认调度器进一步做轻做薄，并且插件化，才是 kube-scheduler 正确的演进方向。

思考题

请问，Kubernetes 默认调度器与 Mesos 的“两级”调度器，有什么异同呢？

精选留言(7)

CYH

问题回答：messos二级调度是资源调度和业务调度分开；优点：插件化调度框架（用户可以自定义自己调度器然后注册到messos资源调度框架即可），灵活可扩展性高.缺点：资源和业务调度分开无法获取资源使用情况，进而无法做更细粒度的调度.k8s调度是统一调度也就是业务和资源调度进行统一调度，可以进行更细粒度的调度；缺点其调度器扩展性差。以上是本人拙见，请老师指正。

作者回复: 一点都不拙！

2018-11-29

□ 1

□ 18
loda

想请教个问题，kubernetes如何保证用户能提前知道这次调度能否成功？
场景：很多公司都希望提前就发现资源池余量是否充足，从而决定是要加机器还是可以继续扩容
拙见：
1.scheduler的缓存其实是一个非常重要的数据，可以提供当前时刻调度能否成

功的视图，但是直接暴露出来不太符合kubernetes以apiserver为依据的原则
2.提供余量检查接口，实时查询apiserver中所有pod和node，根据扩容参数算出剩余资源量。不过规模大后，对集群压力太大
3.定时采集上述指标，缺点是实时性太差
4.监听pod crud，自己独立维护一个和scheduler一样的缓存或持久话数据，每次基于这个数据判断剩余量。缺点是维护成本较高，容易出现数据不一致

想问下，有没有更合适的方式?

2018-12-09

□

□ 2
宋晓明

老师 刚才遇到一个问题 我初始化的时候 分配的ip范围是10.64.200.0/22 然后两个master 3个node kubernetes自己分配master分别是200.0/24 201.0/24 然后其他两个node是202.0/24 203.0/24 其实这些范围已经用满了 在加第三个node 在master看也成功了 但创建多个pod的时候 第三个node上的pod是失败的,后来发现第三个node上kube-bridge网桥上没有IP地址，需求是：因为master上不会有自己创建的pod的，所以master上地址段有点浪费，我如何把master上空闲的地址段用到新增node上，还是不用管，还是怎么样?

2018-11-27

□

□ 1
dgonly

调度器对APIServer的更新失败，Bind过程失败，老师说等schedule cache同步后就恢复正常，这个怎么理解?
我理解是informer path继续watch，发现pod没有被bind到node就继续执行一遍调度流程吗？如果某种原因更新APIServer一直失败，会不会就一直执行重新调度的操作，有没有一种类似重试超过一定比如就丢弃的机制。谢谢!

2018-11-26

□

□ 1
wypsmall

请教一个问题，调度策略中有没有对网络io限制呢？也就是说不希望高网络io的pod被调度到同一个宿主机。

2019-02-13