

服务治理，你应该听得很多了。但是我想说，你所听到的服务治理可能混合了流量调度等其它内容。我们这里会把服务治理和流量调度分开来讲。所以，这里只涉及服务治理上的一些关键技术，主要有以下几点。

- 服务关键程度。
- 服务依赖关系。
- 服务发现。
- 整个架构的版本管理。
- 服务应用生命周期全管理。

服务关键程度和服务的依赖关系

下面，我们先看看服务关键程度和服务的依赖关系。关于服务关键程度，主要是要我们梳理和定义服务的重要程度。这不是使用技术可以完成的，这需要细致地管理对业务的理解，才能定义出架构中各个服务的重要程度。

然后，我们还要梳理出服务间的依赖关系，这点也非常重要。我们常说，“没有依赖，就没有伤害”。这句话的意思就是说，服务间的依赖是一件很易碎的事。依赖越多，依赖越复杂，我们的系统就越易碎。

因为依赖关系就像“铁锁连环”一样，一个服务的问题很容易出现一条链上的问题。因此，传统的SOA希望通过ESB来解决服务间的依赖关系，这也是为什么微服务中希望服务间是没有依赖的，而让上层或是前端业务来整合这些个后台服务。

但是要真正做到服务无依赖，我认为还是比较有困难的，总是会有一些公有服务会被依赖。我们只能是降低服务依赖的深度和广度，从而让管理更为简单和简洁。在这一点上，以Spring boot为首的微服务开发框架给开了一个好头。

微服务是服务依赖最优解的上限，而服务依赖的下限是千万不要有依赖环。如果系统架构中有服务依赖环，那么表明你的架构设计是错误的。循环依赖有很多的副作用，最大的问题是这是一种极强的耦合，会导致服务部署相当复杂和难解，而且会导致无穷尽的递归故障和一些你意想不到的问题。

解决服务依赖环的方案一般是，依赖倒置的设计模式。在分布式架构上，你可以使用一个第三方的服务来解决这个事。比如，通过订阅或发布消息到一个消息中间件，或是把其中的依赖关系抽到一个第三方的服务中，然后由这个第三方的服务来调用这些原本循环依赖的服务。

服务的依赖关系是可以通过技术的手段来发现的，这其中，[Zipkin](#)是一个很不错的服务调用跟踪系统，它是通过 [Google Dapper](#)这篇论文来实现的。这个工具可以帮你梳理服务的依赖关系，以及了解各个服务的性能。

在梳理完服务的重要程度和服务依赖关系之后，我们就相当于知道了整个架构的全局。就好像我们得到了一张城市地图，在这张地图上可以看到城市的关键设施，以及城市的主干道。再加上相关的监控，我们就可以看到城市各条道路上的工作和拥堵情况。这对于我们整个分布式架构是非常非常关键的。

我给很多公司做过相关的咨询。当他们需要我帮忙解决一些高并发或是架构问题的时候，我一般都会向他们要一张这样的“地图”，但是几乎所有的公司都没有这样的地图。

服务状态和生命周期的管理

有了上面这张地图后，我们还需要有一个服务发现的中间件，这个中间件是非常非常关键的。因为这个“架构城市”是非常动态的，有的服务会新加进来，有的会离开，有的会增加更多的实例，有的会减少，有的服务在维护过程中（发布、伸缩等），所以我们需要有一个服务注册中心，来知道这么几个事。

- 整个架构中有多少种服务？
- 这些服务的版本是什么样的？
- 每个服务的实例数有多少个，它们的状态是什么样的？
- 每个服务的状态是什么样的？是在部署中，运行中，故障中，升级中，还是在回滚中，伸缩中，或者是在下线中.....

这个服务注册中心有点像我们系统运维同学说的CMDB这样的东西，它也是非常之关键的，因为没有这个东西，我们将无法知道这些服务运作的状态和情况。

有了这些服务的状态和运行情况之后，你就需要对这些服务的生命周期进行管理了。服务的生命周期通常会有以下几个状态：

- Provision，代表在供应一个新的服务；
- Ready，表示启动成功了；
- Run，表示通过了服务健康检查；
- Update，表示在升级中；
- Rollback，表示在回滚中。
- Scale，表示正在伸缩中（可以有Scale-in和Scale-out两种）。
- Destroy，表示在销毁中。
- Failed，表示失败状态。

这几个状态需要管理好，不然的话，你将不知道这些服务在什么样的状态下。不知道在什么样的状态下，你对整个分布式架构也就无法控制了。

有了这些服务的状态和生命周期的管理，以及服务的重要程度和服务的依赖关系，再加上一个服务运行状态的拟合控制（后面会提到），你一下子就有了管理整个分布式服务的手段了。一个纷乱无比的世界就可以干干净净地管理起来了。

整个架构的版本管理

对于整个架构的版本管理这个事，我只见到亚马逊有这个东西，叫VersionSet，也就是由一堆服务的版本集所形成的整个架构的版本控制。

除了各个项目的版本管理之外，还需要在上面再盖一层版本管理。如果Build过Linux分发包，那么你就会知道，Linux分发包中各个软件的版本上会再盖一层版本控制。毕竟，这些分发包也是有版本依赖的，这样可以解决各个包的版本兼容性问题。

所以，在分布式架构中，我们也需要一个架构的版本，用来控制其中各个服务的版本兼容。比如，A服务的1.2版本只能和B服务的2.2版本一起工作，A服务的上个版本1.1只能和B服务的2.0一起工作。这就是版本兼容性。

如果架构中有这样的问题，那么我们就需要一个上层架构的版本管理。这样，如果我们要回滚一个服务的版本，就可以把与之有版本依赖的服务也一起回滚掉。

当然，一般来说，在设计过程中，我们希望没有版本的依赖性问题。但可能有些时候，我们会有这样的问题，那么就需要在架构版本中记录下这个事，以便可以回滚到上一次相互兼容的版本。

要做到这个事，你需要一个架构的manifest，一个服务清单，这个服务清单定义了所有服务的版本运行环境，其中包括但不限于：

- 服务的软件版本；
- 服务的运行环境—环境变量、CPU、内存、可以运行的结点、文件系统等；
- 服务运行的最大最小实例数。

每一次对这个清单的变更都需要被记录下来，算是一个架构的版本管理。而我们上面所说的那个集群控制系统需要能够解读并执行这个清单中的变更，以操作和管理整个集群中的相关变更。

资源/服务调度

服务和资源的调度有点像操作系统。操作系统一方面把用户进程在硬件资源上进行调度，另一方面提供进程间的通信方式，可以让不同的进程在一起协同工作。服务和资源调度的过程，与操作系统调度进程的方式很相似，主要有以下一些关键技术。

- 服务状态的维持和拟合。
- 服务的弹性伸缩和故障迁移。
- 作业和应用调度。
- 作业工作流编排。
- 服务编排。

服务状态的维持和拟合

所谓服务状态不是服务中的数据状态，而是服务的运行状态。也就是服务的Status，而不是State。也就是上述服务运行时生命周期中的状态—Provision, Ready, Run, Scale, Rollback, Update, Destroy, Failed....

服务运行时的状态是非常关键的。服务运行过程中，状态也是会有变化的，这样的变化有两种。

- 一种是不预期的变化。比如，服务运行因为故障导致一些服务挂掉，或是别的什么原因出现了服务不健康的状态。而一个好的集群管理控制器应该能够强行维护服务的状态。在健康的实例数变少时，控制器会把不健康的服务给摘除，而又启动几个新的，强行维护健康的服务实例数。

- 另外一种预期的变化。比如，我们需要发布新版本，需要伸缩，需要回滚。这时，集群管理控制器就应该把集群从现有状态迁移到另一个新的状态。这个过程并不是一蹴而就的，集群控制器需要一步一步地向集群发送若干控制命令。这个过程叫“拟合”——从一个状态拟合到另一个状态，而且要穷尽所有的可能，玩命地不断地拟合，直到达到目的。

详细说明一下，对于分布式系统的服务管理来说，当需要把一个状态变成另一个状态时，我们需要对集群进行一系列的操作。比如，当需要对集群进行Scale的时候，我们需要：

- 先扩展出几个结点；
- 再往上部署服务；
- 然后启动服务；
- 再检查服务的健康情况；
- 最后把新扩展出来的服务实例加入服务发现中提供服务。

可以看到，这是一个比较稳健和严谨的Scale过程，这需要集群控制器往生产集群中进行若干次操作。

这个操作的过程一定比较“慢”的。一方面，需要对其它操作排队；另一方面，在整个过程中，我们的控制系统需要努力地逼近最终状态，直到完全达到。此外，正在运行的服务可能也会出现问题，离开了我们想要的状态，而控制系统检测到后，会强行地维持服务的状态。

我们把这个过程就叫做“拟合”。基本上来说，集群控制系统都是要干这个事的。没有这种设计的控制系统都不能算做设计精良的控制系统，而且在运行时一定会有很多的坑和bug。

如果研究过Kubernetes这个调度控制系统，你就会看到它的思路就是这个样子的。

服务的弹性伸缩和故障迁移

有了上述的服务状态拟合的基础工作之后，我们就能很容易地管理服务的生命周期了，甚至可以通过底层的支持进行便利的服务弹性伸缩和故障迁移。

对于弹性伸缩，在上面我已经给出了一个服务伸缩所需要的操作步骤。还是比较复杂的，其中涉及到了：

- 底层资源的伸缩；
- 服务的自动化部署；
- 服务的健康检查；
- 服务发现的注册；
- 服务流量的调度。

而对于故障迁移，也就是服务的某个实例出现问题时，我们需要自动地恢复它。对于服务来说，有两种模式，一种是宠物模式，一种是奶牛模式。

- 所谓宠物模式，就是一定要救活，主要是对于stateful 的服务。
- 而奶牛模式，就是不救活了，重新生成一个实例。

对于这两种模式，在运行中也是比较复杂的，其中涉及到了：

- 服务的健康监控（这可能需要一个APM的监控）。
- 如果是宠物模式，需要：服务的重新启动和服务的监控报警（如果重试恢复不成功，需要人工介入）。
- 如果是奶牛模式，需要：服务的资源申请，服务的自动化部署，服务发现的注册，以及服务的流量调度。

我们可以看到，弹性伸缩和故障恢复需要很相似的技术步骤。但是，要完成这些事情并不容易，你需要做很多工作，而且有很多细节上的问题会让你感到焦头烂额。

当然，好消息是，我们非常幸运地生活在了一个比较不错的时代，因为有Docker和Kubernetes这样的技术，可以非常容易地让我们做这个工作。

但是，需要把传统的服务迁移到Docker和Kubernetes上来，再加上更上层的对服务生命周期的控制系统的调度，我们就可以做到一个完全自动化的运维架构了。

服务工作流和编排

正如上面和操作系统做的类比一样，一个好的操作系统需要能够通过一定的机制把一堆独立工作的进程给协同起来。在分布式的服务调度中，这个工作叫做Orchestration，国内把这个词翻译成“编排”。

从《分布式系统架构的冰与火》一文中的SOA架构演化图来看，要完成这个编排工作，传统的SOA是通过ESB（Enterprise Service Bus）—企业服务总线来完成的。ESB的主要功能是服务通信路由、协议转换、服务编制和业务规则应用等。

注意，ESB的服务编制叫Choreography，与我们说的Orchestration是不一样的。

- Orchestration的意思是，一个服务像大脑一样来告诉大家应该怎么交互，就跟乐队的指挥一样。（查看[Service-oriented Design: A Multi-viewpoint Approach](#)，了解更多信息）。
- Choreography的意思是，在各自完成专属自己的工作的基础上，怎样互相协作，就跟芭蕾舞团的舞者一样。

而在微服务中，我们希望使用更为轻量的中间件来取代ESB的服务编排功能。

简单来说，这需要一个API Gateway或一个简单的消息队列来做相应的编排工作。在Spring Cloud中，所有的请求都统一通过API Gateway（Zuul）来访问内部的服务。这个和Kubernetes中的Ingress相似。

我觉得，关于服务的编排会直接导致一个服务编排的工作流引擎中间件的产生，这可能是因为受到了亚马逊的软件工程文化的影响所致——亚马逊是一家超级喜欢工作流引擎的公司。通过工作流引擎，可以非常快速地将若干个服务编排起来形成一个业务流程。（你可以看一下AWS上的Simple Workflow服务。）

这就是所谓的Orchestration中的conductor 指挥了。

小结


好了，今天的分享就这些。总结一下今天的主要内容：我们从服务关键程度、服务依赖关系、整个架构的版本管理等多个方面，全面阐述了分布式系统架构五大关键技术之一——服务资源调度。希望这些内容能对你有所启发。

你现在的公司中是怎样管理和运维线上的服务的呢？欢迎分享一下你的经验和方法。

下一篇文章中，我们将从流量调度和状态数据调度两个方面，来接着聊分布式系统关键技术。

文末有系列文章《分布式系统架构的本质》的目录，供你查看，方便你找到自己感兴趣的内容。


- [分布式系统架构的冰与火](#)
- [从亚马逊的实践，谈分布式系统的难点](#)
- [分布式系统的技术栈](#)
- [分布式系统关键技术：全栈监控](#)
- [分布式系统关键技术：服务调度](#)
- [分布式系统关键技术：流量与数据调度](#)
- [洞悉PaaS平台的本质](#)
- [推荐阅读：分布式系统架构经典资料](#)
- [推荐阅读：分布式数据调度相关论文](#)



左耳朵耗子


全年独家专栏《左耳听风》

每邀请一位好友订阅
你可获得 **36元** 现金返现

获取海报 

嘉宾 陈皓

资深互联网技术专家



戳此获取你的专属海报