从目前可以得到的信息来看，对分布式服务化架构实践最早的应该是亚马逊。因为早在2002年的时候，亚马逊CEO杰夫·贝索斯（Jeff Bezos）就向全公司颁布了下面的这几条架构规定（来自《Steve Yegge对Google平台吐槽》一文）。

1. 所有团队的程序模块都要通过Service Interface方式将其数据与功能开放出来。
2. 团队间程序模块的信息通信，都要通过这些接口。
3. 除此之外没有其它的通信方式。其他形式一概不允许：不能直接链结别的程序（把其他团队的程序当做动态链接库来链接），不能直接读取其他团队的数据库，不能使用共享内存模式，不能使用别人模块的后门，等等。唯一允许的通信方式是调用Service Interface。
4. 任何技术都可以使用。比如：HTTP、CORBA、Pub/Sub、自定义的网络协议等。
5. 所有的Service Interface，毫无例外，都必须从骨子里到表面上设计成能对外界开放的。也就是说，团队必须做好规划与设计，以便未来把接口开放给全世界的程序员，没有任何例外。
6. 不这样做的人会被炒鱿鱼。

这应该就是AWS（Amazon Web Service）出现的基因吧。当然，前面说过，采用分布式系统架构后会出现很多的问题。比如：

- 一个线上故障的工单会在不同的服务和不同的团队中转过来转过去的。
- 每个团队都可能成为一个潜在的DDoS攻击者，除非每个服务都要做好配额和限流。
- 监控和查错变得更为复杂。除非有非常强大的监控手段。
- 服务发现和服务治理也变得非常复杂。

为了克服这些问题，亚马逊这么多年的实践让其可以运维和管理极其复杂的分布式服务架构。我觉得主要有以下几点。

1. **分布式服务的架构需要分布式的团队架构**。在亚马逊，一个服务由一个小团队（Two Pizza Team不超过16个人，两张Pizza可以喂饱的团队）负责，从前端负责到数据，从需求分析负责到上线运维。这是良性的分工策略——按职责分工，而不是按技能分工。
2. **分布式服务查错不容易**。一旦出现比较严重的故障，需要整体查错。出现一个S2的故障，就可以看到每个团队的人都会上线。在工单系统里能看到，在故障发生的一开始，大家都在签到并自查自己的系统。如果没问题，也要在线待命（standby），等问题解决。（我在《故障处理最佳实践：应对故障》一文中详细地讲过这个事）。
3. **没有专职的测试人员，也没有专职的运维人员，开发人员做所有的事情**。开发人员做所有事情的好处是——吃自己的狗粮（Eat Your Own Dog Food） 最微观的实践。自己写的代码自己维护自己养，会让开发人员明白，写代码容易维护代码复杂。这样，开发人员在接需求、做设计、写代码、做工具时都会考虑到软件的长期维护性。
4. **运维优先，崇尚简化和自动化**。为了能够运维如此复杂的系统，亚马逊内部在运维上下了非常大的功夫。现在人们所说的DevOps这个事，亚马逊在10多年前就做到了。亚马逊最为强大的就是运维，拼命地对系统进行简化和自动化，让亚马逊做到了可以轻松运维拥有上千万台虚机的AWS云平台。
5. **内部服务和外部服务一致**。无论是从安全方面，还是接口设计方面，无论是从运维方面，还是故障处理的流程方面，亚马逊的内部系统都和外部系统一样对待。这样做的好处是，内部系统的服务随时都可以开放出来。而且，从第一天开始，服务提供方就有对外服务的能力。可以想像，以这样的标准运作的团队其能力会是什么样的。

在进化的过程中，亚马逊遇到的问题很多，甚至还有很多几乎没有人会想到的非常生僻的东西，它都一一学习和总结了，而且都解决得很好。

构建分布式系统非常难，充满了各种各样的问题，但亚马逊还是毫不犹豫地走了下去。这是因为亚马逊想做平台，不是“像淘宝这样的中介式流量平台”，而是那种“可以对外输出能力的平台”。

亚马逊觉得自己没有像史蒂夫·乔布斯（Steve Jobs）这样的牛人，不可能做出像iPhone这样的爆款产品，而且用户天生就是众口难调，与其做一个大家都不满意的软件，还不如把一些基础能力对外输出，引入外部的力量来一起完成一个用户满意的产品。这其实就是在建立自己的生态圈。虽然在今天看来这个事已经不稀奇了，但是贝索斯早在十五年前就悟到了，实在是个天才。

所以，分布式服务架构是需要从组织，到软件工程，再到技术上的一个大的改造，需要比较长的时间来磨合和改进，并不断地总结教训和成功经验。

分布式系统中需要注意的问题

我们再来看一下分布式系统在技术上需要注意的问题。

问题一：异构系统的不标准问题

这主要表现在：

- 软件和应用不标准。
- 通讯协议不标准。
- 数据格式不标准。
- 开发和运维的过程和方法不标准。

不同的软件，不同的语言会出现不同的兼容性和不同的开发、测试、运维标准。不同的标准会让我们用不同的方式来开发和运维，引起架构复杂度的提升。比如：有的软件修改配置要改它的.conf文件，而有的则是调用管理API接口。

在通讯方面，不同的软件用不同的协议，就算是相同的网络协议里也会出现不同的数据格式。还有，不同的团队因为用不同的技术，会有不同的开发和运维方式。这些不同的东西，会让我们的整个分布式系统架构变得异常复杂。所以，分布式系统架构需要有相应的规范。

比如，我看到，很多服务的API出错不返回HTTP的错误状态码，而是返回个正常的状态码200，然后在HTTP Body里的JSON字符串中写着个：error，bla bla error message。这简直就是一种反人类的做法。我实在不明白为什么会有众多这样的设计。这让监控怎么做啊？现在，你应该使用Swagger的规范了。

再比如，我看到很多公司的软件配置管理里就是一个key-value的东西，这样的东西灵活到可以很容易地被滥用。不规范的配置命名，不规范的值，甚至在配置中直接嵌入前端展示内容……

一个好的配置管理，应该分成三层：底层和操作系统相关，中间层和中间件相关，最上面和业务应用相关。于是底层和中间层是不能让用户灵活修改的，而是只让用户选择。比如：操作系统的相关配置应该形成模板来让人选择，而不是让人乱配置的。只有配置系统形成了规范，我们才hold得住众多的系统。

再比如：数据通讯协议。通常来说，作为一个协议，一定要有协议头和协议体。协议头定义了最基本的协议数据，而协议体才是真正的业务数据。对于协议头，我们需要非常规范地让每一个使用这个协议的团队都使用一套标准的方式来定义，这样我们才容易对请求进行监控、调度和管理。

这样的规范还有很多，我在这就不一一列举了。

问题二：系统架构中的服务依赖性问题

对于传统的单体应用，一台机器挂了，整个软件就挂掉了。但是你千万不要以为在分布式的架构下不会发生这样的事。分布式架构下，服务是会有依赖的，于是一个服务依赖链上，某个服务挂掉了，会导致出现“多米诺骨牌”效应，会倒一片。

所以，在分布式系统中，服务的依赖也会带来一些问题。

- 如果非关键业务被关键业务所依赖，会导致非关键业务变成一个关键业务。
- 服务依赖链中，出现“木桶短板效应”——整个SLA由最差的那个服务所决定。

这是服务治理的内容了。服务治理不但需要我们定义出服务的关键程度，还需要我们定义或是描述出关键业务或服务调用的主要路径。没有这个事情，我们将无法运维或是管理整个系统。

这里需要注意的是，很多分布式架构在应用层上做到了业务隔离，然而，在数据库结点上并没有。如果一个非关键业务把数据库拖死，那么会导致全站不可用。所以，数据库方面也需要做相应的隔离。也就是说，最好一个业务线用一套自己的数据库。这就是亚马逊服务器的实践——系统间不能读取对方的数据库，只通过服务接口耦合。这也是微服务的要求。我们不但要拆分服务，还要为每个服务拆分相应的数据库。

问题三：故障发生的概率更大

在分布式系统中，因为使用的机器和服务会非常多，所以，故障发生的频率会比传统的单体应用更大。只不过，单体应用的故障影响面很大，而分布式系统中，虽然故障的影响面可以被隔离，但是因为机器和服务多，出故障的频率也会多。另一方面，因为管理复杂，而且没人知道整个架构中有什么，所以非常容易犯错误。

你会发现，对分布式系统架构的运维，简直就是一场噩梦。我们会慢慢地明白下面这些道理。

- 出现故障不可怕，故障恢复时间过长才可怕。
- 出现故障不可怕，故障影响面过大才可怕。

运维团队在分布式系统下会非常忙，忙到每时每刻都要处理大大小小的故障。我看到，很多大公司，都在自己的系统里拼命地添加各种监控指标，有的能够添加出几万个监控指标。我觉得这完全是在“使蛮力”。一方面，信息太多等于没有信息，另一方面，SLA要求我们定义出“Key Metrics”，也就是所谓的关键指标。然而，他们却没有。这其实是一种思维上的懒惰。

但是，上述的都是在“救火阶段”而不是“防火阶段”。所谓“防火胜于救火”，我们还要考虑如何防火，这需要我们在设计或运维系统时都要为这些故障考虑，即所谓 Design for Failure。在设计时就要考虑如何减轻故障。如果无法避免，也要使用自动化的方式恢复故障，减少故障影响面。

因为当机器和服务数量越来越多时，你会发现，人类的缺陷就成为了瓶颈。这个缺陷就是人类无法对复杂的事情做到事无巨细的管理，只有机器自动化才能帮助人类。 也就是，人管代码，代码管机器，人不管机器！

问题四：多层架构的运维复杂度更大

通常来说，我们可以把系统分成四层：基础层、平台层、应用层和接入层。

- 基础层就是我们的机器、网络和存储设备等。
- 平台层就是我们的中间件层，Tomcat、MySQL、Redis、Kafka之类的软件。
- 应用层就是我们的业务软件，比如，各种功能的服务。
- 接入层就是接入用户请求的网关、负载均衡或是CDN、DNS这样的东西。

对于这四层，我们需要知道：

- 任何一层的问题都会导致整体的问题；
- 没有统一的视图和管理，导致运维被割裂开来，造成更大的复杂度。

很多公司都是按技能分工是，把技术团队分为产品开发、中间件开发、业务运维、系统运维等子团队。这样的分工导致各管一摊，很多事情完全连不在一起。整个系统会像 “多米诺骨牌”一样，一个环节出现问题，就会倒下去一大片。因为没有一个统一的运维视图，不知道一个服务调用是如何经过每一个服务和资源，也就导致我们在出现故障时要花大量的时间在沟通和定位问题上。

之前我在某云平台的一次经历就是这样的。从接入层到负载均衡，再到服务层，再到操作系统底层，设置的KeepAlive的参数完全不一致，导致用户发现，软件运行的行为和文档中定义的完全不一样。工程师查错的过程简直就是一场恶梦，以为找到了一个，结果还有一个，来来回回花了大量的时间才把所有KeepAlive的参数设置成一致的，浪费了太多的时间。

分工不是问题，问题是分工后的协作是否统一和规范。这点，你一定要重视。

小结

好了，我们来总结一下今天分享的主要内容。首先，我以亚马逊为例，讲述了它是如何做分布式服务架构的，遇到了哪些问题，以及是如何解决的。我认为，亚马逊在分布式服务系统方面的这些实践和经验积累，是AWS出现的基因。随后分享了在分布式系统中需要注意的几个问题，同时给出了应对方案。

我认为，构建分布式服务需要从组织，到软件工程，再到技术上的一次大的改造，需要比较长的时间来磨合和改进，并不断地总结教训和成功经验。下篇文章中，我们讲述分布式系统的技术栈。希望对你有帮助。

也欢迎大家分享一下你在分布式架构中遇到的各种问题。

文末给出了**《分布式系统架构的本质》系列文章**的目录，希望你能在这个列表里找到自己感兴趣的内容。如果你在分布式系统架构方面，有其他想了解的话题和内容，欢迎留言给我。

- 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