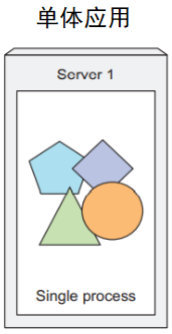
2.开发与部署方式的演变

近年来，应用开发和部署发生了一些变化。这些变化是由两方面促成的，一方面是大型单体应用被拆解为更多小型的微服务，另一方面是应用程序运行所依赖的基础架构发生了变化。理解这些变化，将使我们更好的看到使用k8s和**容器化技术**（比如Docker）带来的好处。

1. 从单体应用到微服务

单体应用由紧密耦合在一起的多个组件构成，由于这些组件在同一个操作系统进程中运行，所以在开发、部署和管理时必须以同一个实体进行。如果对应用某一组件进行了修改，就需要对整个应用进行重新部署。组件缺乏严格的**边界定义**，相互依赖，日积月累导致系统复杂度提升，整体质量也急剧恶化。如下图所示的单体应用：

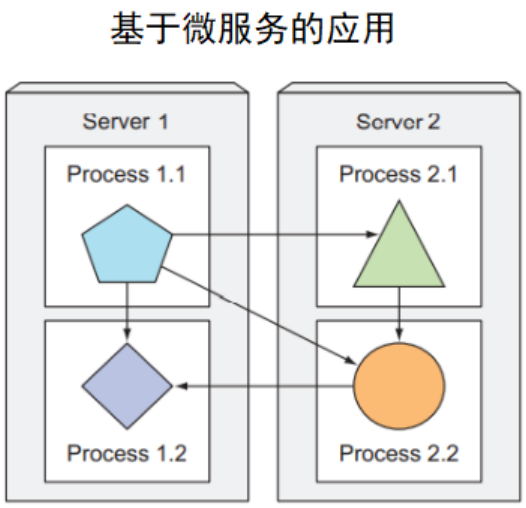


运行一个单体应用程序通常需要少量的能够为运行应用程序提供足够资源的高性能的服务器。为了应对不断增长的系统负荷，我们通常采用增加更多的CPU、内存和其他系统资源的方式来对服务器做**垂直扩展**，或者通过添加额外的服务器和运行应用的多个**副本**的方式对整个系统进行**水平扩展**。虽然垂直扩展

通常不需要应用程序做任何变化，但是成本很快会越来越高，而且在实践中总是有上限。水平扩展在硬件方面相对便宜些，但是需要对应用程序代码做很大的修改，比如应用程序的某些部分非常难于甚至不太可能去做水平扩展（比如关系型数据库）。
如果单体应用的任何一部分不能扩展，那整个应用就不能扩展，除非我们想办法把它拆分开。

将应用拆分为微服务

这些问题迫使我们将复杂的单体应用拆解为更小的、可独立部署的**微服务**组件。每个微服务以一个独立的进程运行，通过简单且定义良好的API接口与其他微服务通信。如下图：

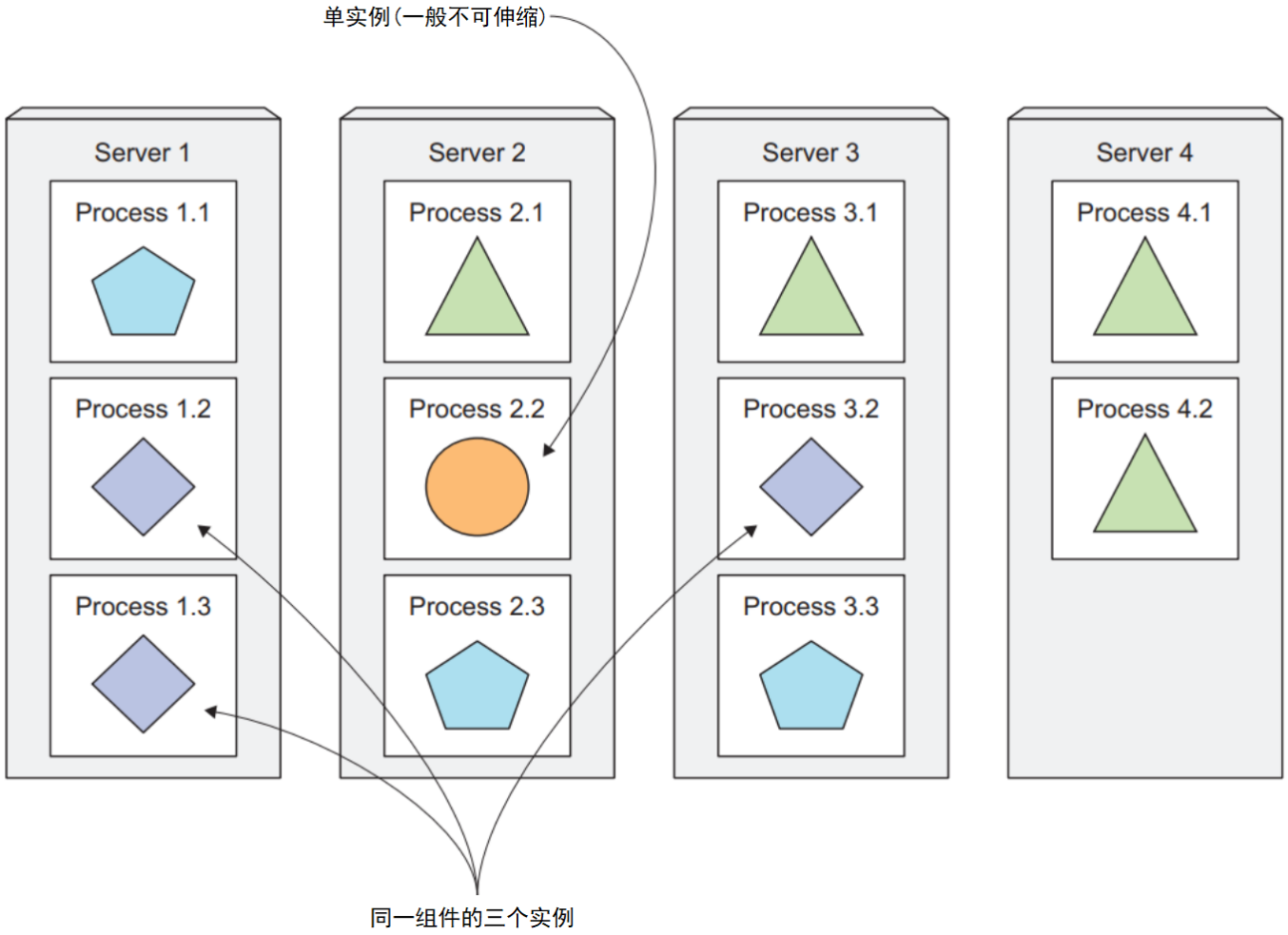


微服务之间可以通过**同步协议**（如HTTP）进行通信，也可通过**异步协议**（如AMQP）进行通信。 基于HTTP，微服务可以对外提供RESTful API接口。这些协议比较简单，能够被大多数开发人员理解，而且不依赖于某一种编程语言。每个微服务可以通过最适合自己的编程语言来实现。

由于每个微服务都是独立的进程，提供相对静态的API，所以可以独立的开发和部署每个微服务。对某个服务进行了修改，并不需要对其他服务进行更改或者重新部署，只要API没有变化，或者只是以向后兼容的方式发生了变化。

扩展微服务

单体应用需要把系统作为一个整体来进行扩展，而微服务的扩展是通过基于单个服务来完成的，这意味着我们可以选择只扩展哪些需要更多资源的服务而让其他服务仍然维持在原有的规模。如下图所示：



某个组件被复制出三份，并且以多进程的方式运行在不同的服务器上。而其他组件却以单个进程的方式运行。当单体应用因为其中一部分无法扩展而整体被限制扩展时，可以把应用拆分成多个微服务，将那些能进行扩展的组件进行水平扩展，不能进行扩展的组件进行垂直扩展。

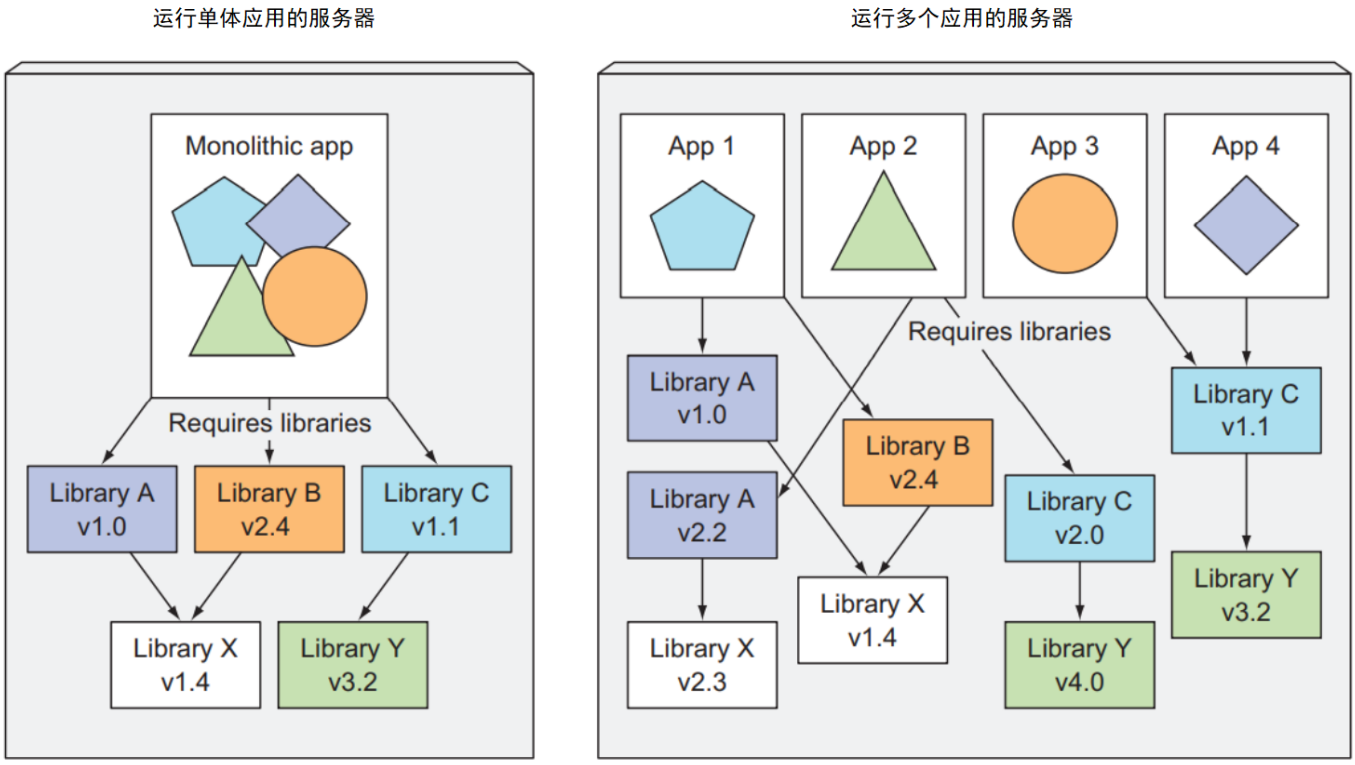
部署微服务

当然，微服务也有缺点。如果你的系统仅包含少许可部署的组件，管理这些组件还是很容易的。完全没必要去选择将每个组件部署到哪里，因为本来选择就不多。当组件数量增加时，部署相关的决定变得越来越困难。因为不仅部署组合数增加了，而且组件间相互依赖的组合数也在增加。

微服务之间是以团队的形式执行工作的，因此它们需要发现彼此并与之通信。部署微服务时，需要正确的配置所有服务以使它们作为一个单一的系统协同工作。随着微服务数量的增加，配置工作将变得繁琐而且更易出错。微服务也会带来一些问题，比如调试代码和追踪程序执行变得困难，因为服务是以多进程方式运行且部署到多个服务器上的。幸运的是，这些问题如今已经被诸如ZipKin这种分布式追踪系统解决了。

2. 为应用程序提供一个一致的环境

微服务中的各个组件不仅是被独立部署的，而且也是被独立开发的。由于它们的独立性，不同团队开发各自的组件是很常见的情况，每个团队都可能使用不同的依赖库并在需求变化的时候替换它们。应用程序组件之间依赖关系的差异性是不可避免的。如下图所示，部署到同一台机器上的多个应用程序会依赖同一个库的不同版本：



如果动态链接的应用依赖不同版本的共享库或者其他环境相关细节，那么在生产服务器对其进行部署和管理将变成运维团队的噩梦。部署到同一台机器上的组件数越多，管理它们各自的依赖以满足需求将会变得更加困难。

无论我们当前正在开发和部署的组件有多少，开发人员和运维团队不得不面对的一个问题就是应用程序**运行环境的差异性**。这种巨大的差异不仅体现在开发和生产环境之间，甚至存在于各个生产服务器之间。另一个不可避免的事实是，生产服务器所处的环境会随着时间的推移而变化。这些差异体现在从硬件到操作系统再到每台机器上可用的依赖库上。生产环境通常由运维团队进行管理，而开发者常常比较关心他们自己的开发环境。这两组人对系统管理的理解程度是不同的，这个理解偏差导致两个环境系统有较大的差异，系统管理员更重视保持系统更新，对系统应用最近的安全补丁，而大多数开发人员则并不太关心这点。

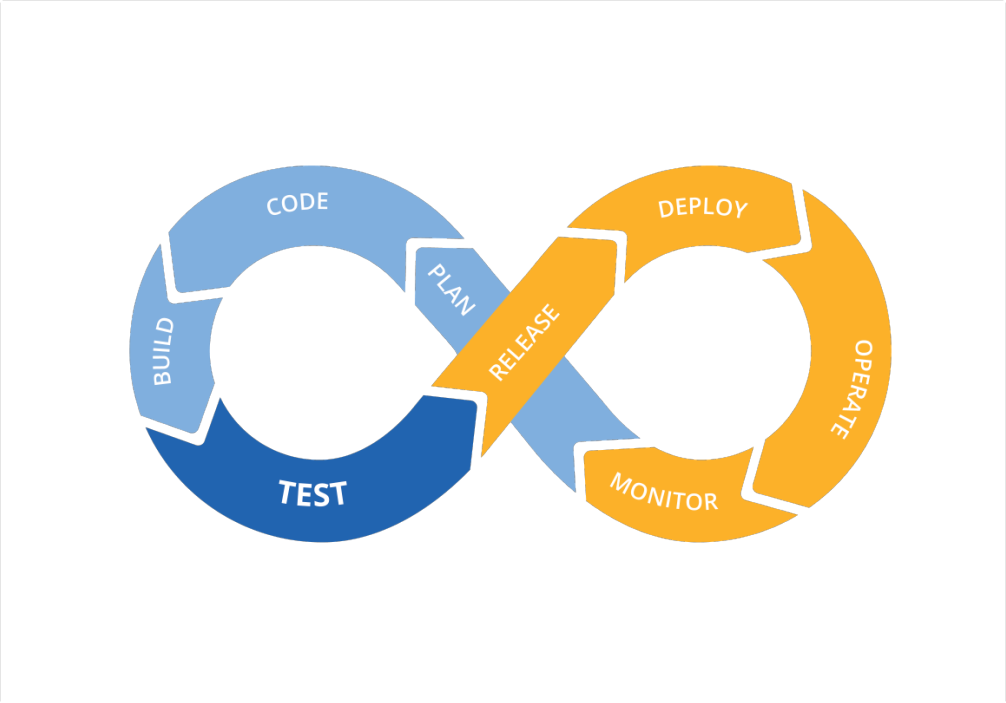
生产系统可能会运行多个应用，这些应用来自开发人员或者部署团队。而开发人员的个人PC就不存在这种情况了。一个生产系统必须提供合适的环境供应用程序部署，即使这些程序需要不同的甚至冲突的依赖库的版本。

为了减少这些在生产环境才会暴露的问题，比较理想的做法是让应用在开发和生产阶段运行在完全一样的环境中，以便应用程序拥有相同的操作系统、库、系统配置、网络环境以及其他所有资源。如果可能

的话，我们不希望这个环境随着时间的推移而变化。同时我们也希望在向服务器部署新应用时不会影响到该服务器上的其他应用。

3.迈向持续交付：DevOps和NoOps

在过去，开发团队完成应用开发之后会将其交付给运维团队，然后由运维团队进行部署和监管。但是现在越来越多的组织开始意识到，由同一个团队负责应用从开发、部署到运维的整个生命周期会更好。这意味着开发人员、QA和运维团队之间的合作需要贯穿整个流程。这种实践被称作DevOps。



有什么优点?

让开发人员更多地在生产环境运行应用，能够使他们对用户的需求、问题以及运维团队维护应用时所遇到的问题有一个更好的理解。应用开发人员如今更倾向于更早地向用户交付应用，然后收集用户的反馈以作进一步的开发。

为了更频繁地发布应用的新版本，我们需要简化部署流程。理想的状态是开发人员能够自己部署应用，而不需要交付给运维人员操作。但是部署应用通常需要具备对数据中心底层基础设施以及硬件架构的理解。开发人员通常不知道甚至不想了解这些细节。

术业有专攻

虽然开发人员和系统管理员的共同目标是成功运行应用并服务于客户，但是他们也有着不同的个人目标和驱动因素。开发人员热衷于创造新的功能和提升用户体验。他们通常不太关心诸如底层操作系统是否已经更新所有安全补丁这种事情，而是更愿意把这些事交给系统管理员。

运维团队负责生产环境的应用部署以及底层的硬件基础设施。他们关心系统安全、资源利用率以及其他对于开发者来说优先级不太高的东西。运维人员不希望处理所有应用组件之间暗含的依赖关系，也不想考虑底层操作系统或者基础设施的改变会怎样影响到应用程序，但是他们不得不关注这些事情。

理想的状态是，开发人员自己部署应用，不需要了解硬件基础设施，不需要与运维团队打交道。这被称作NoOps。很显然，我们仍然需要一些人来关心硬件基础设施，但是理想情况下，他们不用再关心每个应用的独特性。

在后面的学习中，你会看到，Kubernetes能帮我们实现所有这些想法。通过对硬件进行抽象并将其对外暴露为一个单一的平台来部署和运行应用程序，使开发人员能够配置和部署自己的应用程序，而不再需要系统管理员的任何协助。这也使得系统管理员专注于保持底层基础设施运转正常的同时， 不需要关注实际运行在平台上的应用程序。