12-Factors经常被直译为12要素，也被称为12原则，12原则由公有云PaaS的先驱Heroku于2012年提出（原文参见12factor.net），目的是告诉开发者如何利用云平台提供的便利来开发更具可靠性和扩展性、更加易于维护的云原生应用。距离12原则的提出已有五年之久，12原则的有些细节可能已经不那么跟得上时代，也有人批评12原则的提出从一开始就有过于依赖Heroku自身特性的倾向，不过不管怎么说，12原则依旧是业界最为系统的云原生应用开发指南，我们可以把它作为一个非常有力的参考，但是也千万不要教条。

## 原则1：一份基准代码，多份部署

这个原则不管对微服务模式还是其他软件开发模式来说都非常基本，所以被列为12原则的第一条，该原则包括如下四个子原则：

1. 使用代码库管理代码，一般是Git或者SVN，这个要求非常初级，相信本书的读者都会遵守。
2. 一份基准代码（即一个代码库）对应一个应用。如果通过一份基准代码可以编译出多个应用，那么应该考虑将该基准代码按应用拆分为多份；如果一个应用需要多份基准代码，那么要么考虑将多份基准代码合并，要么考虑将该应用按基准代码拆分为多个。
3. 不允许多个应用共享一份基准代码，如果确实需要共享，那就把需要共享的基准代码的稳定版本发布为类库，然后通过依赖管理策略进行加载。
4. 同一应用的多份部署可以使用同一份基准代码的不同版本，但是不可以使用不同的基准代码，类似原则2，使用不同基准代码的应用不应被视为同一应用。

违反子原则2和3，会给代码管理和编译工作带来麻烦：

1. 如果一份基准代码可以编译出多个应用，那么这几个应用之间必然会存在不清晰的依赖关系，随着时间的推移，这种依赖关系会变得愈加混乱，以至于修改一个应用的代码，会给其他应用带来不可预知的影响。这样的基准代码显然极难维护。
2. 基准代码的划分和应用的划分非常类似，也是系统边界的一种体现，如果一个应用需要从多份基准代码编译，那么多数情况下这个应用的内外部边界问题会存在问题。如果边界不存在问题，那么请将多份基准代码合并为一份，而不是维持这种古怪的设计。
3. 如果多个应用不是通过类库，而是直接共享一份基准代码，那么这份被共享的基准代码会很难维护，对这份基准代码的修改必须谨慎考虑对多个应用可能造成的影响。正确的方式是将这份基准代码发布为类库，保持清晰的边界和接口约定供其它应用调用。

## 原则2：显式声明依赖关系

这里的依赖指所有的依赖，包括应用程序本身的类库和操作系统层面被应用程序所使用的库文件或者其他二进制文件，都必须进行显示声明，并对版本做出明确的指定。不要假定运行环境中已经存在应用所需要的任何依赖项，而是应该假定什么都没有（即使有也很可能不是应用所需要的版本）。如果使用容器方式进行部署，容器的基础镜像很可能是Busybox或者Alpine之类的迷你Linux，那么就几乎等于什么都没有。如果使用微服务模式，理想情况下，微服务之间的依赖关系也应该进行显示声明。

以前我们往往不会对依赖做如此严格的管理，因为应用不会有太大规模的部署，也不会进行频繁的发布，如果发现运行环境里缺少某些依赖，那么就临时手工处理一下，也不是什么太大的问题。如今在微服务模式下，应用的部署规模大、发布频率高，还记得前文所说的“不可变服务器”吗？如果这个时候还是使用原有的模式，则会带来混乱。

声明依赖的方式有很多，常见的方式是使用依赖清单，根据依赖清单进行依赖检查，同时使用依赖隔离工具保证应用不会调用系统中存在但是依赖清单中未声明的依赖项；另一种方式是使用容器技术，将应用和依赖打包为容器镜像，依赖的声明和隔离就一并解决了。

## 原则3：在环境中存储配置

首先需要明确的是，这里的配置指与部署环境有关的配置，例如：

* + 数据库、消息代理、缓存系统等后端服务的连接配置和位置信息，如URL、用户名、密码等。
  + 第三方服务的证书。
  + 每份部署独有的配置，例如：域名、连接数、与部署目标环境资源规模有关的JVM参数等。

所有部署中都相同的信息，例如原则2里讲到的依赖信息，不在本原则所讨论的范围内。一些虽然在不同的部署中有所差异、但是和业务相关的信息，例如资金结算的转换比例，也不属于本原则所讨论的配置。

我想大多数的开发者都知道如何通过使用配置文件实现配置和代码的分离，但是这种方式仍然存在一些缺点，例如：

1. 配置文件容易被开发人员不小心提交到代码库中，造成密码、证书等敏感信息泄露。提交到代码库中的配置文件还容易被和应用一起部署到目标环境中，很可能会导致在目标环境中应用了错误的配置或者造成配置冲突。
2. 配置文件会分散在不同的目录中，并且有不同的格式（配置文件的格式往往与开发语言和框架相关），这会给配置的统一管理造成困难。

为了避免上述问题，本原则要求将在环境中存储配置。一种典型的方式是把配置存储在环境变量中，这会使配置和代码彻底的分离，格式上也与开发语言和框架再无瓜葛，并且也不会被误提交到代码库中。还可以使用Spring Cloud Config Server这类配置管理服务进行配置推送，并将配置的历史版本和变更原因也一起管理起来。

## 原则4：把后端服务当作附加资源

这里的后端服务指的是应用运行所依赖的各种服务，例如数据库、消息代理、缓存系统等，对于云原生应用来说，往往还会有日志收集服务、对象存储服务、以及各种通过API访问的服务；当作附加资源指的是把这些服务作为外部的、通过网络调用的资源。

该原则有如下几层含义：

1. 不要将这些服务放在应用本地：云原生应用要求应用本身无状态化，那么状态信息就应该存储在外部服务中（参见不可变服务器）。同时，微服务模式要求应用责权单一以实现可靠性和扩展性，如果在应用本地放置数据库，那么微服务平台将无法通过更换应用的故障实例实现应用的高可用性，也无法通过自动化的横向伸缩实现扩展性，因为应用实例内包含两种性质完全不同的软件（应用和数据库），无法对两者使用同一种方式进行横向扩展。另外，如果将这些服务放在应用本地，那么也无法通过充分利用云平台提供的能力简化运维工作，例如，如果在应用本地放置数据库，而不是使用云平台提供的数据库服务，那么显然无法利用数据库服务提供的自动备份、安全、和高可用等特性。
2. 通过URL或者服务注册/认证中心访问这些后端服务：应用应该能够在不进行任何代码修改的情况下，在不同的目标环境中进行部署，应用不应该和后端服务的任何一种具体实现存在紧耦合关系。
3. 类似“显式声明依赖关系”原则，应用最好也能够对其使用的这些后端服务进行显示声明，以方便云平台对服务资源进行自动绑定，在后端服务出现故障的时候，云平台也能够对其进行自动恢复。

## 原则5：严格分离构建、发布和运行

在本原则中，构建、发布和运行这三个概念可能和从前有所不同，因此有必要首先对其进行明确：

* + 构建指的是将应用代码转化为执行体的过程：构建时会拉取特定版本的代码和依赖项，将其编译为二进制文件（针对编译型语言），并和资源文件一起打包。
  + 发布指的是将构建的结果和部署所需的配置相结合，并将其放置于运行环境之中。
  + 运行指的是将发布的结果启动为运行环境中的一个或多个进程。

本原则要求构建、发布和运行这三个步骤严格区分：

1. 禁止直接修改运行状态的代码或者对应用进行打补丁，因为这些修改很难再同步回构建步骤，这时运行状态的代码就成为了“孤本”。同时，也不应该在运行期间修改应用的配置，配置的修改应该仅限于发布阶段（参见不可变服务器）。
2. 运行这一步骤应该非常简单，仅限于启动进程，资源文件的关联应仅限于构建阶段，配置的结合应仅限于发布阶段。

同时，每一次发布都应该对应一个唯一的发布ID，发布的版本应当像一个只能追加的账本，一旦发布就不能修改。这么做的好处是：

1. 每一份运行状态的代码都可以在对应的发布和构建阶段找到它的来源，这是实现重新发布、故障实例的自动替换、发布出错后的版本回退等机制的基础。
2. 运行步骤非常简单，这样在硬件重启、实例故障和横向扩展等情况下，应用可以简单和快速的实现重启。

## 原则6：以一个或多个无状态的进程运行应用

本原则要求应用进程的内部不要保存状态信息，任何状态信息都应该被保存在数据库、缓存系统等外部服务中。应用实例之间的数据共享也要通过数据库和缓存系统等外部服务进行，直接的数据共享不但违反无状态原则，还引入了串行化的单点，这会为应用的横向扩展带来障碍。在微服务模式下，应用不应该在自身进程内部缓存数据以供将来的请求使用，因为微服务模式以多实例方式运行应用，将来的请求多半会被路由到其他实例，此时虽然可以使用粘滞会话将请求保持在同一个实例上，但是无论是云原生应用还是微服务模式都极力反对使用粘滞会话，原因如下：

1. 很难对粘滞会话实现负载均衡，因为粘滞会话的均衡性不仅决定于负载均衡策略，还和会话本身的行为相关，例如，可能存在应用某些实例上的会话已经大量退出，而另一些实例上的会话依然处于活动状态，此时这两部分实例的负载处于不均衡状态，而负载均衡器无法将活动会话转移到空闲的应用实例。
2. 启动新的应用实例不会立即提高应用的整体处理能力，因为这些新实例只能承接新会话，旧的会话依旧粘滞在旧的应用实例上。
3. 应用实例退出会导致会话丢失，所以在实例发生故障时，即使云平台可以对故障实例进行自动替换，也会导致用户数据丢失。即使是对应用实例进行人工维护，也需要在维护之前对该实例上的会话进行转移，这往往意味着需要编写复杂的业务代码。在传统模式下，可以通过在双机之间进行会话复制来实现对用户无感知的单机下线维护（虽然会付出处理能力减半的代价），但是在微服务模式下，应用的实例数量往往远不止两个，在大量的实例之间进行会话复制会使实例之间原本非常简单的逻辑关系复杂化，此时将无法通过云平台对其进行无差别的自动化维护。另外，在实例之间进行会话复制也意味着实例之间存在着直接的数据共享，这会为应用的横向扩展带来障碍。

所以，粘滞会话是应用实现可用性和扩展性的重要障碍，使用粘滞会话显然是种得不偿失的选择。更好的实现方式是将会话信息存储在缓存服务中。

## 原则7：通过端口绑定提供服务

服务端应用通过网络端口提供服务，这点毋庸置疑，但是本原则还有如下两个深层次的含义：

1. 无论是云原生应用还是微服务模式都要求应用应该完全自我包含，而不是依赖于外部的应用服务器，端口绑定指的是应用直接与端口绑定，而不是通过应用服务器进行端口绑定。如果一定要使用应用服务器，那就使用嵌入式应用服务器，无论是云原生应用还是微服务模式都极力反对将多个应用放置于同一个应用服务器上运行，因为在这种模式下，一个应用出错会对同一个应用服务器上的其他应用造成影响，也无法针对单一应用做横向扩展。
2. 端口绑定工作应该由云平台自动进行，云平台在实现应用到端口的绑定之外，还需要实现内部端口到外部端口的映射和外部端口到域名的映射。在应用的整个生命周期内，应用实例会经历多次的重新部署、重启或者横向扩展，端口会发生变化，但URL会保持不变。

## 原则8：通过进程模型进行扩展

与通过进程模型进行扩展相反的方式是通过线程模型进行扩展，这是一种相对较为传统的方式，典型的例子是Java应用。当我们启动一个Java进程的时候，通常会通过JVM参数为其设置各个内存区域的容量上下限，同时还可能会在应用层面为其设置一个或者多个线程池的容量上下限，当外部负载变化时，进程所占用的内存容量和进程内部的线程数量可以在这些预先设置好的上下限之间进行扩展，这种方式也被称为纵向扩展或者垂直扩展。但是这种方式存在一些问题，首先，在进程的内存容量和线程数量提高时，应用的某些性能指标可能不会得到同步提高，甚至可能会下降（这往往是因为程序对某些无法扩展的资源进行争用所造成的），这种参差不齐的性能扩展对外部负载提高的承接能力会很不理想，有时甚至会适得其反；其次，为了使进程本身可以完成纵向扩展，还需要在虚拟机层面或者容器层面为其预留内存资源和对应的CPU资源，这会造成大量的资源浪费（当然，也可以使虚拟机或者容器跟随进程一起进行纵向扩展，这在技术上是可行的，但是会为虚拟机或者容器管理平台的资源调度造成一些不必要的困难，例如频繁的虚拟机迁移或者容器重启）。

所以，现在更为推崇使用“固定的”进程（对前面Java应用的例子来说，就是固定的内存容量和线程池容量），在外部负载提高时，启动更多的进程，在外部负载降低时，停止一部分进程，这种方式就是本原则所说的通过进程模型进行扩展，有时候也被称为横向扩展或者水平扩展。这种扩展方式的好处是，在进程数量增加的时候，应用的各种性能指标会得到同步的提高，这种提高即使不是线性的，也会按照一种平滑和可预期的曲线展开，可以更为稳定的应对外部负载的变化。

云原生应用和微服务模式极力推崇将通过进程模型进行扩展作为唯一的扩展方式，除了前文所述，还有一个原因是进程是云平台可以操作的最小运行单元（当然，可以通过其他技术手段去操作线程，但是那不会成为云平台的通用技术特性），云平台可以根据各个层面的监控数据，通过预设规则决定是否为应用增加或者减少进程，例如，当前端的负载均衡器检测到访问某个后端应用的并发用户数超过某个阈值时，可以立即为这个后端应用启动更多的进程，以承接更大的负载，同时还可以选择是否对该应用后端的数据库进行扩展。如果此时选择对应用进行纵向扩展，则云平台既不知道应用处理能力的变化，也无法对这种变化进行预期管理，更无法使应用的前后端对这种变化进行联动，即该应用的扩展行为脱离了云平台的管理。在微服务模式下，如果大量的进程都采用纵向扩展方式，则会为平台的资源调度带来极大的混乱。

*注3：该原则似乎更适合被称为横向扩展原则，但是为了和12原则的原文保持一致，这里我们仍然将其称为“通过进程模型进行扩展”。*

## 原则9：快速启动和优雅终止可最大化健壮性

该原则要求应用可以瞬间（理想情况下是数秒或者更短）启动和停止，因为这将有利于应用快速进行横向扩展和变更或者故障后的重新部署，而这两者都是程序健壮性的体现。前文不止一次提到过应用的快速启动，在理念章节的开头，我们提到过平价的进程生成对多道程序设计至关重要，而微服务模式在某种程度上可以认为是多道程序设计在Web领域和分布式系统下的进一步扩展，这里所说的平价进程生成指的是操作系统的一种特性，是应用快速启动的基础，除此之外为了保证应用可以在数秒内完成启动，还需要大量的优化工作，需要开发人员掌握复杂的调优技术与工具，有些工作必须在应用的初始设计阶段完成，例如：如果应用体积过大或者是引用了太多的库文件，那么再多的后期优化也无法将启动时间降低到数秒以内。在“原则5：严格分离构建、发布和运行”中我们还提到，应用的运行步骤应该非常简单，这里的“简单”也隐含着快速的意思，目的是为了在硬件重启、实例故障和横向扩展等情况下，应用可以快速的实现重启。除此之外，“原则6：以一个或多个无状态的进程运行应用”也与应用的快速启动有关，遵守无状态原则，使用云平台提供的缓存服务，而不是在应用内部加载缓存，可以避免在应用启动期间进行耗时的缓存预热。

比起应用的快速启动，优雅终止（Graceful Shutdown）需要考虑的问题会更为广泛一些。优雅终止需要尽可能降低应用终止对用户造成的不良影响（对于微服务应用，用户可能是人，也可能是其他微服务），对于短任务来说，这一般意味着拒绝所有新的请求，并将已经接收的请求处理完毕后再终止；对于长任务来说，这一般意味着应用重启后的客户端重连和为任务设置断点并在重启后继续执行。除此之外，优雅终止还需要释放所有被进程锁定的资源，并对事务的完整性和操作的幂等性做出完备的考虑。

最后，应用还必须应对突如其来的退出，在硬件出现故障时或者进程崩溃时，应用需要保证不会对其使用的数据造成损坏，遵守无状态原则、将数据交由后端服务处理的应用可以很容易的将应对突然退出的复杂度外部化。

## 原则10：开发环境与线上环境等价

本原则的浅层次含义是要求在开发环境和线上环境中使用相同的软件栈，并尽可能为这些软件栈使用相同的配置，以避免“It works on my machine.”这类问题。本原则反对在不同的环境中使用不同的后端服务，虽然可以使用适配器或者在代码中做出兼容性考虑以消除后端服务的差异，但是这将牵扯开发人员和测试人员大量的精力以保证这些适配器和代码确实可以按预期工作，在应用的整个开发周期中，这将积累极大的额外工作量，是一种非常不必要的资源浪费。

近年来个人电脑的性能大幅提高，开发人员一度得以在本地开发环境中运行与生产环境中一致的软件栈，而不是像曾经那样采用轻量的替代方案。但是随着云原生应用和微服务模式的流行，情况又发生了微妙的变化：开发微服务时需要依赖云平台提供的基础服务和其他微服务，越来越难以把这些服务完整的运行在本地，与此同时，完全的在线开发愈发成为一种趋势，那样的话至少在软件栈上开发环境和线上环境就真的没有任何区别了。在我编写这段文字的时候，Red Hat公司刚好在洽购在线开发环境创业公司Codenvy用以充实他们的云平台产品OpenShift，而另一家与Codenvy类似的创业公司Cloud9在差不多一年前被Amazon公司收购。

本原则的深层次含义是尽量缩小开发环境和线上环境中时间和人员的差异。开发环境中的代码每天都在更新，而这些更新往往会累积数周甚至数月才会被发布到线上环境，这是开发环境和线上环境在时间上的巨大差异；开发人员只关心开发环境，运维人员只关心线上环境，开发人员和运维人员在工作上鲜有交集，这是开发环境和线上环境在人员上的巨大差异。对于前一个差异，本原则要求更为密集和频繁的向线上环境发布更新，要求建立机制以保障开发人员可以在数小时甚至数分钟内既可将更新发布到线上，这也正是本章理念部分中持续交付所提倡的；对于后一个差异，本原则要求开发人员不能只关心开发环境中自己的代码，更要密切关注代码的部署过程和代码在线上的运行情况，这也正是DevOps所提倡的。

## 原则11：把日志当作事件流

应用程序应该将其产生的事件以每个事件一行的格式按时间顺序输出，这点毋庸置疑，但是本原则想说的其实是：应用程序不要自行管理日志文件。

以前我们习惯将应用程序产生的事件分门别类的输出到不同的日志文件，并为每个日志文件指定在本地文件系统上的存储位置，为了避免单一日志文件过大，还会为它们配置轮转策略。该原则极力反对上述做法，而是要求应用程序将日志以事件流的方式输出到标准输出STDOUT和标准错误输出STDERR，然后由运行环境捕获这些事件流，并转发到专门的日志处理服务进行处理。这样做的原因是：

1. “原则6：以一个或多个无状态的进程运行应用”要求应用程序无状态，那么应用程序就不应该将日志文件这种价值信息存储在本地文件系统上。当然，可以在本地运行一个日志收集进程读取日志文件，并将其转发到专门的日志处理服务，以保证价值信息不被意外丢弃，但是这将带来如下问题：
   * 需要提供一种机制以保证日志收集进程可靠运行。
   * 需要通过配置文件告知日志收集进程去哪里读取日志文件。
   * 需要在应用程序所在的虚拟机或者容器上为日志收集进程开放一个网络端口以供其发送日志内容，这不仅增加了网络的复杂度，还给网络安全带来了隐患。

由此可见，直接将日志输出到STDOUT和STDERR并由运行环境对其进行捕获远比这种方案来的简洁和可靠。

1. 在存在专门的日志处理服务时，由应用程序自行对日志进行分类显得死板和毫无必要；只需将日志以事件流方式发送给日志处理服务，日志处理服务可以对这些日志按不同视角进行灵活的分类，而不是受限于一种既定的分类规则。
2. “原则6：以一个或多个无状态的进程运行应用”中还提到“微服务模式以多实例方式运行应用，将来的请求多半会被路由到其他实例”，所以单个应用实例的日志无法描述完整的业务操作，不具备业务层面的价值。必须将应用所有实例的日志汇总到日志处理服务，由日志处理服务按特定规则（如按用户ID或者对象ID）对其进行聚合，才能完整展现应用在业务层面的操作过程。应用在以多实例方式运行时，应用的单个实例可能会因为软硬件故障而重启，或者被横向扩展机制创建和销毁，所以必须将应用所有实例的日志汇总，才能完整的描述应用的运行情况。

## 原则12：后台管理任务当作一次性进程运行

这是12原则的最后一条，也是最晦涩的一条。如果你在看过原文之后觉得哪里有些不大对，不必担心，因为很多人的想法和你一样。在本节的开头曾提到有人批评12原有过于依赖Heroku自身特性的倾向，这些批评多半可能是本原则导致的。事实上，通过SSH接入线上环境并使用脚本语言执行管理任务的做法已经不再被提倡，无论是云原生应用还是微服务模式都极力反对这种做法，原因可以参见“理念五：不可变服务器”和“理念六：提供声明式接口”。另外还有一个原因显而易见：你的应用有数个或者数十个实例，那么应该登录到哪个实例中执行管理任务呢？如果在管理任务执行的过程中，所在实例因为软硬件故障重启，或者被横向扩展机制销毁，那又该怎么办？

正确的做法是，如果管理任务是修改应用配置，那么应该通过配置管理服务进行操作，参见“原则3：在环境中存储配置”；如果管理任务是批处理任务，例如数据的迁移、清洗或者检查，那么应该通过云平台的批处理机制进行操作，大多数的云平台都会提供这种机制，例如Kubernetes的Jobs。本原则还提到“应用的管理进程应该和应用的常驻进程运行于同一环境，并使用相同的代码、版本和配置”，这是一条比较有价值的建议，可以避免由于环境或代码等不一致造成的一些潜藏问题。虽然现在不提倡通过SSH接入应用常驻进程所在的环境并执行管理任务，但是如果你使用容器技术，那么很容易通过容器模板创建一个和应用常驻进程一致的运行环境，并在其中执行管理任务。