Beyond the twelve factors app 中文版

项目主页

- beyond-twelve-factors-app 中文版
- 本书是译本，从上面的项目地址可以下载到原英文版本
- 由bitocean1024:GitHub 翻译

序言

今时今日，了解如何设计云上应用变得越来越重要。云计算已经快速的从一个由初创和前沿公司拥抱的技术转变成了企业级系统未来的基础设施。为了占有今天的市场，大型和小型公司都在拥抱并实践云架构

在Pivotal，我的工作是让云基础设施成功赋能我们的客户。以典型场景为例，我绝大多数的精力都花在了与运维团队合作去安装和配置云平台，包括训练他们如何管理，维护和监控云平台。我们交付了一个产品级别的，全自动化的云应用运行时，让开发者们能够充分享受云平台带来的好处，但是怎么做到这一点呢？开发者们经常会问我们，他们应该遵守什么准则和最佳实践去设计他们的应用程序才能充分的利用云带来的便利性。本书就是为了回答这些问题的

不管你是在构建一个云上应用还是想迁移一个已经存在的应用到云上面，本书都是开发者和架构师们的一个极佳的应该常伴左右的指导书

-- Dan Nemeth,

Solutions Architect,Pivotal

前言

之所以会产生流行的技术术语是因为我们有必要去构建一个共同的语言，在这个语言下，我们可以讨论一些问题而不必停下来先介绍背景知识。共享的流行术语不仅仅是为了方便，同样对于讨论问题，架构设计都很重要

Twelve-Factor应用这个词是工程师们在会议当中，在午餐闲聊当中，在架构设计方案评审中被牵引出来，并且不断传递下去的

问题是对于流行的专业术语并非每一个人都能理解到词背后的上下文，也并非每个人的理解都是一样的。就twelve-Factor来说，不同的人的理解可能完全不同，甚至正在阅读本书的读者对于12 factors 可能之前没有任何的接触

本书的目标就是为了详细的解释清楚到底什么是Twelve-Factor应用，希望能够让读者对于Twelve-Factor有共同的理解。而且本书能够让读者对于Twelve-Factor有更多的理解，让大家不仅仅只是使用云，而是一起努力让云原生更繁荣

原始的12 Factors

在云计算的早期，一些初创公司带了一个新的物种，对于这个新的物种，很少有开发者和公司对它有处理经验。这是一个新的抽象级别，它使IT专业人员可以不受一整套非功能性需求的困扰。对某些人来说，这是一个黑暗和恐怖的领域，但是有一些人则是极力的拥抱这个领域，从目前看来他们当初所有的期盼的都已经实现了

最早公开宣称要涉足这个领域的先驱是Heroku，它为你的程序提供了一个平台，有了这个平台，你需要做的事情仅仅是上传你的代码到git，并且构建你的应用，然后云将会接管后面所有的事情，你的程序将会神奇的在线上跑起来

Heroku承诺你永远不需要去关心基础设施，所有你要做的事情就是充分的利用云去构建你的应用，云将会让你的程序运行的很好

问题是大多数人并不知道如何去构建一个“云友好”的应用，正如本书要讨论的，“云友好”的应用并非只是跑在云上面就完了，他们需要拥抱弹性扩展，瞬时文件系统，无状态，并且把所有的东西都当作服务来看待。以这种方式构建的应用能够快速的扩展和部署，允许它们的开发团队能够快速响应市场需求，为应该添加新的功能

时至今日，依然有很多违反云原生的模式。早期需要适应云的人并不知道对于云他们应该做些什么，不应该做些什么，也不知道为了构建一个云原生的应用，他们在设计构架的时候需要考虑哪些点。这是一种新的，几乎没有参考框架的构建应用程序的方式

为了解决这个问题（也为了增加他们平台的适配性）Heroku的一组人在2012年提出了12 Factors。这是构建云原生应用需要遵守的基本规则

这12 Factors的目的是教会开发人员如何构建可用于自动化和声明式设置，与基础操作系统具有明确约束，可动态扩展的云就绪应用程序

> 对于许多人来说，云原生和12 Factors是等价的，本书的其中一个目的是为了阐明清楚，云原生的含义远比12 Factors要丰富的多。在Heroku的场景里面，云原生真正意味着“在Heroku上工作良好”

这12 Factors 被用于指导开发者去开发新的云应用，同时也可以用来评估已经存在的项目到底与云之间有多少适配性

Codebase

一个版本管理系统的代码仓库对应多个部署

Dependencies

严格的声明和隔离依赖

Configuration

将配置存储到环境当中

Backing Services

把所有附加的资源都当作后端服务

Build,release,run

严格的区分构建和运行阶段

Processes

用一个或者多个无状态进程来运行应用程序

Port binding

通过端口绑定来暴露服务

Concurrency

通过进程模式来扩展

Disposabolity

支持快速重启和快速的优雅停机提升程序的鲁棒性

Dev/prod parity

保持开发、测试、生产环境相同

Logs

把logs当作事件流

Admin proceddes

用一次性的进程来跑管理任务

12 Factors可以很好地说明在云中构建和部署应用程序以及为团队建立弹性伸缩应用程序生产线需严格要遵守的纪律。然而技术从它被创造的那一天起就会一直演进下去，所以在某些情况下，有必要详细说明初始准则，并添加旨在满足现代应用程序开发标准的新准则

Beyond the Twelve-Factor Application

在这本书里面，首先我呈现了一些基于原始12 Factors 创造出来的新的指导准则。在本书中，我改变了Factor之间的顺序，用以明确区分Factor之间的优先级.同时新增了一些应该被考虑到的Factors，比如说遥测，加密，和“API First”，如果你的应用需要跑在云上面。而且，我可能会为原始的factors增加一些警告和异常，用以反应今时今日的最佳实践

把优先顺序的改变，重新定义，新增都考虑进去，本书将要描述以下的云原生应用需要遵守的factors:

1. One codebase,one applicatiopn
2. API first
3. 依赖管理
4. 设计，构建，发布，运行
5. 配置，证书，和代码
6. 日志
7. 可处置性
8. 背后服务
9. 同等环境
10. 管理性进程
11. 端口绑定
12. 无状态进程
13. 并发
14. 遥测
15. 认证和授权

12factor.net 提供了一个极佳的起点，可以作为一个用来衡量应用与云适配性的标尺。正如你接下来要看到的一样，这些因素是相互关联的，可能遵守了其中一个能够让你很容易遵守下一个，进入一个良性循环。一旦人们进入这个良性循环，他们经常会反问自己当初为什么要用其他的方式来构建应用呢

无论您是在开发新的应用程序并且没有一行传统代码的负担，还是正在分析包含数百个传统应用程序的企业投资组合，本书将会为你开发云原生应用提供一个完备的指导方针

CHAPTER 1

Once Codebase,One Application

第一个原始的因素是：代码仓库（codebase）,原始的说法是：“一个版本追踪系统的代码仓库，多份部署”

当需要管理一个开发团队方方面面的时候，代码的组织，组件的管理等细节通常都会被忽略掉。但是合理的应用准则和管理应该能够明确出来“按月开发”和“按天开发”的产品在代码管理上的区别

云原生应用必须一直存在与一个版本管理系统的代码仓库，一个代码仓库是一个版本控制系统的repository或者一批repository但是共享相同的库根

单个应用的仓库代码被用于产生许多不可变的被部署于不同环境的release。遵守这个准则能够迫使团队成员去分析他们的应用边界以及分析是否存在一些单体服务应该被拆分成多个微服务。如果你的一个应用有多个代码仓库，可能你的系统需要被解耦合了，而不应该是一个大的单体应用

> 1、一个不可变的release是一个构建出来的组件，这个组件是不可变的，这一点将会在本书的后面多次提及，这种类型的组件需要被测试，确保dev/prod环境相同，提前预测部署结果

> 2、在微服务拆分上可以参考Building Microservices by Sam Newman (O’Reilly) 获取更多的指导

最简单的违反这条准则的例子就是你的应用确实是由多个代码仓库组成的，这样的做法让你的应用的构建和部署阶段几乎不可能做到自动化

另外一个经常打破这条准则的的方式是一个主应用和一个紧耦合的worker程序(or an en-queuer and de-queuer,etc 不知道该怎么翻译这个)，两者一起合作完成一项工作。在这种场景下，确实是多个代码仓库支持一个应用，尽管他们共享一个根目录。这就是为什么我认为代码仓库这个概念应该是表明一个有凝聚力的单元，而不仅仅是一个版本控制系统的repository

相反的，当一个代码仓库被用于产生多个应用的时候，这条准则依然是被打破了。举个例子，一个代码仓库是一个包装模块，但是包装模块确是由多个启动脚本或者由多个不同时间点执行的脚本组成。在Java里面，EAR包是一个打破 One Codebase的毒药，在解释语言的世界里（e.g.，Ruby）,你可能会在一个codebase里面包含多个启动脚本，每个脚本干了完全不同的事情

多个应用在一个仓库通常是一种迹象，这个迹象表明可能是多个不同的团队维护了相同的codebase，这种方式后续将会由于各种原因变得很丑陋。康威定律表明组织架构最终会反应到产品架构里面，换句话说，功能紊乱的差劲的组织架构，缺乏秩序的团队通常会导致功能紊乱和缺乏秩序的代码

如果你确实面临着多个团度维护一个代码仓库，你可能需要利用康威定律把你大的团队拆分称小的团队，每个小团队去独立负责自己的应用或者说是微服务

如果你审视你的应用并且决定找机会重构你的代码仓库成为一个个小的产品，你会发现多个仓库对应的一个应用可能可以被拆分称一个一个微服务，微服务通过暴露API能够被其他应用所复用

换句话说“一个代码仓库，一个应用”并不是说不允许你在多个应用中共享代码，它仅仅意味着你的共享代码应该是一个单独的仓库

这条准则也不是说所有共享的代码都需要成为一个微服务，准确的说，你需要评估你的共享代码是否应该被拆分成一个独立发布的产品，这个产品以供应商的身份作为你的应用的一个依赖

CHAPTER 2

API First

这个章节讨论一个当代应用开发需要具备的但是没有被原始的12 factors覆盖到的一个方面。不考虑你正在开发的应用类型，假设你正在开发一个云原生应用，然后你的终极目标是让这个应用成为一个微服务系统内的一部分

Why API First

假设你已经全面拥抱了本书讨论的其他因素，你正在构建云原生应用，在你提交完代码到你的仓库之后，自动跑完了测试用例，同时也在几分钟以内发布了一个版本到某个环境

现在，您组织中的另一个团队开始构建与您的代码进行交互的服务，很快，您将有多个团队，所有团队都在水平依赖别人的服务上构建自己服务，这些服务都在不同的发布节奏上

如果没有一个准则来约束这个过程，系统集成将会成为一个噩梦。为了避免集成失败，也为了让你意识到API应该是你开发过程中的一等公民，API first给了团队更具抗干扰性的公有约束，而不用关心内部的开发细节

尽管可能你并不打算开发某个生态中的一个微服务，花费少量的精力去遵守API first这条准则在将来仍然有可能给你带来不错的回报

Building Services API first

如今，“移动优先”的概念越来越受关注。它指的是从项目一开始就抱定一个想法：你的应用最终可能会给移动设备提供接口。同样，API优先意味着您要构建的是供客户端应用程序和服务使用的API

正如本书开头提及的，云原生不仅仅是一个规则列表，它是一种哲学，对于我们中的某些人来说，甚至是一种生活方式。这些准则不一定对应到一个特定的云平台提出的要求，但对于构建现代应用程序的人和组织至关重要，这些习惯将为将来云环境的变化做好准备

在做出的每个决定和编写的每一行代码的时候都需要考虑到这条准则，即通过使用API去满足应用程序的每个功能要求。甚至是用户界面，无论是网络界面还是移动界面，实际上都不过是API的使用者

通过首先设计你的API,你能够真正开始编码之前就与你的利益相关者（你的内部团队成员，你的客户，甚至是你的组织内其他有可能消费你的API的人）开始沟通了，这种合作方式允许你更淡定的去编写功能代码，mock你的接口，编写公开化的文档

如今，您会发现有无数工具和标准来支持API First开发。 API规范有一种标准格式，该格式使用类似于Markdown的语法，称为API Blueprint。这种格式比JSON（或WSDL，博物馆中的遗物）更具人类可读性，并且可以被代码用来生成文档甚至生成服务器mock程序，这在测试服务生态系统中具有不可估量的价值。诸如Apiary之类的工具套件提供了诸如GitHub集成和服务器模拟之类的功能。如果有人想调用你的API，您要做的就是给它一个指向您在Apiary上的应用程序的链接，在那里她可以阅读您的API Blueprint，查看调用API的示例代码，甚至可以模拟调用执行

换句话说，没有理由去拒绝API first,这种模式可以应用于非云软件开发，但是它特别适合云开发，因为它具有云原生应用程序开发的很多特点，比如说快速原型制作，支持构建微服务生态以及促进自动化部署测试和持续交付的能力

这种模式是“合约优先”开发模式的扩展，这种模式下，开发者专注于应用边界，开发团队各自维护自己的服务，只要通过集成测试服务器保证接口集成点测试通过，那么就可以假设服务之间调用是工作良好的

API first将组织从瀑布模式和过度提前设计的模式中解放出来，允许产品各自演进成独立的系统，可以满足未来不可预测的需求

如果您建立了一个大的单体应用，甚至一个大的紧密耦的生态，那么您响应新需求或创建现有功能的新使用者的能力就会受到阻碍。从另外一个角度来讲，如果你接受了这种观念，即所有的应用都是一种后端服务，并且他们被设计成API first ,然后你的系统将能够自由生长去适应新的需求，去为已经存在的服务容纳新的消费者

生活，饮食和呼吸API-first生活方式，您的投资将成倍增长

CHAPTER 3

Dependeny Management

12 factors的第二个因素：“依赖”，指的是怎样，在什么地方，什么时候去管理应用的依赖

Reliannce on the Mommy Server

在经典的企业环境里面，我们曾经使用过一个叫做“Mommy Server”的服务器，这种服务器为应用提供了所有需要的资源，从满足应用的依赖到提供一个web server去承载应用。“Mommy server”的对立面是嵌入式的，自启动的服务器，所有运行应用需要的依

赖都被包含在了一个构建好的组件里面

云模型是经典企业模型的成熟版本，因此，我们的应用需要不断的成长去充分利用云。应用不能假设服务器或者容器有它们需要的一切，相反应用需要自己去包含这些依赖。迁移到云，意味着需要把你的组织从“Mommy server”里面解脱出来

如果你构建应用的时候没有依赖容器模型的语言或者框架（Ruby,Go,Java with Spring Boot），你就已经走在了前面，不用担心“Mommy server”的问题了

现代化的依赖管理

大多数当代编程语言都会有一些机制去管理应用依赖，Maven 和 Gradle是Java世界比较流行的两个工具，.NET开发者用的是NuGet,Ruby用的是Bundler,go编程者用的是godeps.不考虑工具本身，这些工具都提供了类似的功能：他们允许开发者声明依赖，同时让工具去负责满足应用的依赖

许多工具同样有能力去隔离依赖，通常的做法是分析声明的依赖并且把这些依赖捆绑到某个子结构之下，或者在应用组件本身的内部

云原生应用程序永远不会隐性的依赖系统范围的软件包。对于Java，这意味着您的应用程序不能假定容器将管理服务器上的类路径，对于.NET，你的应用不能依赖全局的包缓存机制。Ruby开发者不能依赖于一个已经存在的中央位置。总而言之，你的代码不能依赖部署服务器上的已经存在的依赖

不合理的依赖隔离会引起数不清的问题，其中一些依赖相关的问题里面，比如某个开发者正在本地开发，并且使用到了某个依赖包的X版本，但是在生产环境的中央位置里面放了一个X+1的版本，这可能引起不可预知，很难察觉到的运行失败问题。如果任由这种情况发展，这种类型的失败可能会摧毁整个服务器，甚至给公司带来百万美元的损失

> 隔离依赖是指让依赖紧紧跟随应用一起发布，而不是将依赖放到某个中央的共享位置

正确管理应用程序的依赖关系是关于可重复部署的概念，应用程序应该力求走自动化路线。在理想的情况下，应用的容器是捆绑到应用发布的artifact里面，或者更好的情况下是应用最好没有容器

然后，对于一些企业，把server容器嵌入到应用里面是不太实际的，所以必须能够把发版的应用和server容器组合到一起，在云原生的Heroku上面，buildpack就是用来干这个事情的

遵守应用依赖管理的规则能够让你的应用离云原生更进一步

CHAPTER 4

Design,Build,Release,Run

原始的Factor 5,release,run的意思是严格的区分构建和运行阶段，这是一个非常棒的建议，如果你没有遵守这条规则将会给你带来很多麻烦。除了构建，发版和运行，设计这一步也很重要

在下面的4-1，你能看见一个从设计到运行的流程图，注意这不是一个瀑布流程：从设计到编码再到运行是一个个迭代周期，迭代周期可以根据时间来自由调配。如果你的团队有一个自动的CI/CD管理线，从设计到在生产环境运行可能只需要几分钟

单个代码仓库通过构建得到一个artifact，这个artifact然后与程序外部的应用配置合并成一个不可变的release。这个不可变的release被分发到云环境并且运行起来，本节的要点是强调上面的四个阶段是完全隔离并且独立发生的

Design

在瀑布应用的开发过程中，我们在真正开始编码前花费了大量的时间去设计这个应用。这种开发流程已经不适应现代的快速迭代的需求了

并不是说我们不需要去设计，而是说，我们设计一个个能够单独发布的小功能，小的设计是整个迭代周期里面的其中一部分

应用开发者最好理解应用依赖，在设计阶段就去明确哪些依赖是供应商提供，哪些依赖是自己提供。换句话说，开发者在设计阶段决定最终哪些依赖会与开发好的应用打包到一起发布

Build

构建阶段是将代码仓库的源代码转换成一个带版本号的二进制的artifact.在这个阶段，设计阶段声明的依赖会被提取并且捆绑到构建好的artifact（通常简化成“build”）。在java的世界里面，一个build可能是一个WAR或者一个JAR文件，或者是一个ZIP文件，或者是一个可执行的二进制文件

构建通常是由一个持续集成的服务器完成，构建和部署之间通常是一个1:N的关系。单个构建可以被发布或者部署到任意多的环境，并且每一个不可变的构建应该是符合期望的运行。坚持这种不可变的artifact和其他的factor（特别是环境相同），将会给你足够的信心：如果你的程序能在测试环境运行良好，那么也应该能在生产环境运行良好

如果你曾经也遇到了“在我的电脑上运行的好好的”这种问题，那说明你的程序的四个阶段有可能没有被完全分离。强迫你的团队成员使用CI服务器可能前期看会有由点投入，但是一旦坚持下来，你会发现“one build,many deploys” 开始产生效果了

一旦你确信自己的代码库可以在任何地方运行，并且不再担心生产版本问题，你将开始看到云原生理念的真正惊人的好处，例如，持续部署和发布可以在需求接收之后的数小时之内发生

Release

在云原生的世界里，发布通常是指将你的artifact推送到云环境，构建阶段产生的输出和环境，以及应用配置信息一起组成了另外一个不可变的artifact，一个release

每一个Releases需要有一个唯一的ID来标识，可能是一个时间戳，也可能是一个自增长的数字。由于build和release之间的1:N的关系，所以release不能直接使用build的ID

我们假设你的CI系统已经构建好了你的应用，并且标记为build-1234,CI系统然后释放build的输出到dev，测试和生产环境，每一个release都应该有一个唯一的id，因为这个release不仅仅包含原始的build输出，还包括环境的特定信息

如果发布的程序有问题，如有必要，你可以回滚到上一个release，这就是其中一个很重要的原因：为什么release是不可变，并且唯一标识的

Run

运行阶段同样是由云提供者来完成的（尽管开发者需要在本地运行），不同的云服务供应商在细节上有所不同，但是通用的模式是你的服务将会放置到一个容器里面（Docker,Garden,Warden,etc）,然后一个进程会去启动你的应用

值得注意的是，要确保开发人员可以在本地运行应用程序，同时能够通过CD管道将其部署到多个云中，这通常是一个很难解决的问题。但是，这值得解决，因为开发人员可以毫无阻碍的开发自己的云原生应用

当程序运行起来，云环境然后将保证它的存活，监控它的健康，同时采集它的logs，还有一些其他的管理任务像动态扩容和错误容忍

最后，这条准则的目标是最大化的提升你的发布速度，同时通过自动测试和自动部署来让你对本次发布抱有信心。我们从云上面得到了开箱即用的敏捷部署，同时不会因为速度而失去自己的程序将会在云上工作良好的信心

CHAPTER 5

Configuration,Credentials,and Code

原始的12因素只是声称你需要将配置存储到环境里面，但是我觉得配置这条规则应该需要更清晰一点

> 把配置，凭证和代码当作易挥发的物质，这些物质一旦合并会发生化学反应

这听起来可能有点苛刻，但如果不遵守这条规则，可能会给您带来难以言表的挫折感，离应用程序上线越近越会产生问题

为了能够把配置和代码、证书隔离开，我们需要对配置有一个清晰的定义。配置指会随着部署环境发生变化的值，这个包括：

- 后端服务的URLs和其他信息，比如web service和SMTP 服务
- 连接到数据库的必要信息
- 调用第三方服务比如Amazon AWS， Google Maps, Twitter, and Facebook的证书
- 可能需要放到XMI或者YMAL配置文件的其他属性

配置并不包括程序本身的内部信息，也就是说，如果某一个值在所有的部署环境都是一样的（很明显他是你的不可变的build artifact的一部分），然后它就不是配置

证书是极其敏感的与代码无关的信息，通常来说，程序员会把证书从程序源代码里面提取出来让好把他们放到属性文件里面，但是这样做并没有解决问题。因为配置文件本身仍然是代码仓库的一部分，这意味着证书与你的程序一起发布，这本身就打破了这条规则

> 把你的程序当作开源项目，一个最简单的检验你的证书和配置是否合理的方式，就是假设你现在需要把你的源代码push到GitHub上面

如果外面的人需要访问你的代码，你是否把你依赖的服务的敏感信息给暴露出去了呢？不是你的组织内部的人是否看到了内部后端服务的URLs，证书或者其他敏感的信息呢？

如果你能够在不暴露任何敏感信息的情况下去开源你的代码，那么你可能在隔离你的代码、配置、证书方面做的很好了

不把证书暴露出去是显而易见的，但是不把外部配置暴露出去却显得不那么好理解。外部化的配置能够帮助我们通过CD管道把不可变的builds部署到不同的环境，同时管理自己的开发和生产环境

外部化的配置

都知道要将配置外部化，但是实施起来却不是那么回事了。举几个例子，如果你正在开发一个Java程序，然后你将自己的配置放到属性文件，并且与代码放到一起构建，其他类型的语言可能会有一个YAML文件，.NET程序会有一个XML文件

你应该想到上面的所有做法都是不符合云原生模式的，所有的场景都会由于跨环境可变的配置而导致你无法你构建一个不可变的release artifact

有一种比较笨的外化配置的方法是把你的所有代码里面的配置文件全部抽出来，然后修改你的代码从环境变量去读取。环境变量被认为是外部化配置的最佳实践，尤其是在Cloud Foundry或Heroku等云平台上

根据您的云提供商，您可能可以使用其工具来管理后端服务或绑定服务，以安全的方式向应用程序公开包含服务凭证和URL的结构化环境变量

另外一个推荐的外化配置的方式是使用一个配置中心，比如开源的Spring Cloud Configuration Server，或者其他的也有不少。购买配置服务器产品时应注意的一件事是支持版本控制。如果要对配置进行外部化，则应该能够保护数据更改并获得由谁进行更改以及何时进行更改的历史记录。正是这一要求可以使用仓库版本控制系统比如git来作为一个配置服务器

CHAPTER 6

Logs

本章节，我讨论第11个factor，logs

Logs应该被当作一种事件流，也就是说，logs是程序内部发射出来的时间有序的事件序列。就像在原始的12 factors声明的那样，在云原生范式里面最关键的一点，一个真正云原生的应用从来不关心如何路由和存储他的输出流

有时，这个理念需要慢慢习惯一下，应用程序开发人员，尤其是大型企业的开发人员，通常习惯于严格控制其日志的形式和目的地。在配置文件里面设置设置磁盘日志文件所在的位置，日志滚动策略以处理日志文件的大小和其他相关问题

云原生应用不能假定他们运行在哪个文件系统上面，事实上，它是瞬时的。一个云原生应用会将他们的日志输出到标准输出和标准错误，这样做可能会吓到一批人，害怕失去对日志的控制

你应该将日志的聚合，处理和存储视为一项非功能性要求，该要求不是由您的应用程序满足，而是由您的云提供商或与或者平台上面的其他工具套件来满足。您可以使用ELK技术栈（ElasticSearch，Logstash和Kibana），Splunk，Sumologic等工具或任何其他工具来捕获和分析您的日志

拥抱这种理念，能让你少做一些关于日志的工作

当你的程序从log存储，处理，和分析中解耦，你的代码将会变得简单，你可以依赖工业级的标准工具和技术栈去处理log,而且如果你需要更改你处理log的方式，你可以不用修改你的代码

你的应用程序不应该控制日志的最终输出方式的其中一个原因是由于弹性扩展。当你在固定数量的服务器上具有固定数量的应用实例时，将日志存储在磁盘上似乎能起作用。但是，当您的应用程序可以从1个正在运行的实例动态变为100个，并且您不知道这些实例在何处运行时，您需要您的云提供商来替你你汇总这些日志

简化应用程序的日志处理，可以减少代码量，让你更加专注于你的业务

CHAPTER 7

Disposability

可处置性是原始的12 factors中的第九个

作为一个云实例，一个应用程序的生命周期和支持它的基础设施一样是朝生夕灭的。一个云原生应用的进程应该是可处置的，这意味着他们可以快速的重启和停止。如果一个应用程序不能快速的启动和优雅的关闭，那么它就不能快速的扩展，部署，发布和恢复。我们不仅要在构建应用的时候想到这一点，关键是我们做到这一点就能充分利用云平台的优势

以前，在企业容器中部署的应用可能需要数分钟的时间去启动，较长的启动时间不仅限于旧版或企业应用程序，用解释性语言编写或写得不好的软件可能也需要很长时间才能启动

如果您正在启动一个应用程序，并且要花几分钟时间才能进入稳定状态，那么在当今的高流量世界中，这可能意味着在应用程序启动时会拒绝成百上千个请求。更重要的是，视应用程序部署所在的平台而定，如此缓慢的启动时间实际上可能会触发警报或警告，因为应用程序无法通过其运行状况检查。极慢的启动时间甚至可能阻止您的应用程序完全在云中启动

如果你的应用程序正在一个高负载的场景下，你需要快速的启动更多的实例去处理这个负载，但是慢启动会阻碍你通过扩展去处理较高的负载。如果应用程序不能快速和优雅的关闭，他们同样会失去失败场景下快速恢复的能力，不能快速关闭的能力同样会有耗尽资源的风险

许多应用程序会在启动过程中一些耗时教长的动作，例如获取数据缓存起来或者加载一些依赖项。为了真正地接受云原生架构，你需要单独处理这种动作。例如，您可以将缓存外部化为后端服务，以便您的应用程序可以快速启动和关闭，而无需执行启动前加载动作

CHAPTER 8

Backing Services

因素4的意思是你需要把后端服务当作绑定的资源

上面说的听起来确实是个好的建议，但是为了能够正确的理解这个建议，我们需要去了解到底什么是后端服务以及什么是绑定的资源

后端服务是一个你需要依赖它的功能的应用程序，这是一个相当宽泛的定义，而且是有意定义的这么的宽泛的。其中一些常见的后端服务包括数据存储，消息中心，缓存系统，以及很多其他类型的服务比如说提供业务功能的服务或者加密服务

当你在构建一个云原生的应用的时候，你必须把文件系统也当作瞬时的，你必须把文件存储或者磁盘都当作后端服务，你不能像常规的企业应用一样去从磁盘读取或者写入文件。相反，文件存储应该被当作你应用程序的一个必然存在的资源来考虑

图 8-1 阐明了一个应用，以及一系列的后端服务，和这些服务绑定的资源（连接线）。而且，注意这些文件存储是抽象的（eg,Amazon S3），并不是直接系统级别访问的磁盘

一个绑定的资源是指把你的应用连接到后端服务，绑定到数据库的资源可能需要一个用户名，一个密码，以及一个URI，有了这些你的应用就能消费后端服务了

本书的前面就提到过，我们需要外部化配置（与代码和证书隔离开来）以及我们释放的产品应该是不可变的。把这些规则与一个应用消费后端服务结合起来，我们得到了一些资源绑定的规则：

- 一个应用需要声明它需要的后端服务，但是必须允许云平台去实施资源绑定
- 应用与后端服务的绑定应该是通过外化的配置来完成的
- 如果一个应用想的话，它可以随时在不需要重新部署的前提下去依附或者分离后端服务

举个例子，假设你的应用需要去连接一个Oracle数据库，你在编码的时候可以先假设你已经拥有了一个可以信赖的Oracle数据库（与特定的语言或者工具集相关），程序源代码假设只要配置好了，就能将外部的资源绑定到应用

这意味着你的应用程序没有一行代码会与某个特定的服务耦合到一起，同样的，你可能会有一个用来发送邮件的后端服务，所以你知道你需要通过SMTP连接到邮件服务。但是邮件服务的准确的实现细节从来不会影响到你的应用，同时你的应用也不会依赖于某个固定位置的SMTP服务器，也不需要依赖特定的证书

最终，当你开发的时候把后端服务当作绑定的资源这样考虑的时候，其中一个最大的好处是你可以随时依附和分离资源

> 断路器
> 一些工具包提供断路器允许你的程序在后端服务出现问题的时候与它们断开连接，并且提供了降级的方式。由于断路器位于你的应用和后端服务中间，所以你必须首先拥抱后端服务，你才能利用断路器的优势

一个已经拥抱了后端服务的应用有选择的机会，一个管理员注意到了一个数据库实例已经死了，他可以再启动一个新的数据库实例，然后把你的应用绑定到新的数据库实例

这种与后端服务的灵活性，弹性以及松耦合组成了云原生应用真正现代化的标志

CHAPTER 9

Environment Parity

12 factors中的第十个，开发/生产对等，告诉我们尽可能的保持我们的环境一样

可能有一些组织已经做到了环境对等，但是我们中的大多数人都是工作在下面这样的环境中的：QA环境，以及有不同扩展性和稳定性要求的共享的开发环境。数据库驱动在开发环境，测试环境与生产环境都是不一样的，加密规则，防火墙以及其他的环境配置

也是不同的。一些人有能力部署其中的一些环境，但是总有一些环境无法部署成功。最终结果是，开发人员将会害怕环境，他们即便能够在生产环境部署成功，也基本上没有信心能够在其他环境部署成功

前面讨论设计，构建，发布，运行阶段的时候，我指出“在我们的机器上工作的很好”（在其他环境不行）本身是违反云原生模式的。同样的，“在QA上工作良好”，“在生产环境工作良好”也是违反云原生模式的

遵循环境对等的规则是为了给你的团队和组织信心：应用程序将会在所有环境都工作良好

然后有很多因素都会导致环境的不一致，下面这些是最常见的：

*时间
*人员
*资源

时间

在很多组织里面，从开发人员提交代码到代码真正的部署到生产环境，中间可能会间隔数周或者数月。在这样的组织里面，你可能会经常听到类似“四分之三版本” 或者 “12月20xx版本”，的词语，这样的词语是为了提醒其他人注意一点

如果发生了这样的间隔，人们通常会忘记有哪些功能将会被发版进去（尽管可以通过release notes来回顾），而且更重要的是，开发人员已经忘记了代码是什么样的

一种更现代的方式是，组织应该努力去缩短代码提交到生产发布的时间间隔，将它从数周或者数月缩短到几分钟或者几小时。一个合理的CICD管线应该自动在不同的环境执行测试，并且在测试通过之后将程序推送到生产环境。随着云支持“零停工”部署，这种方式将成为标准

这种想法可能会吓到很多人，但是一旦开发者习惯了这种开发方式，并且知道他们的代码将会在提交的同一天就会发布到生产环境，组织纪律和代码规范将会得到极大的提升

人员

从历史来看，部署应用程序的人与公司的规模直接相关。小公司里面，开发人员编写完代码之后还要去部署程序，但是大公司里面，更多的人和团队会介入

原始的12因素表明开发者和部署者应该是同一个人，这个在应用部署的平台是一个像Heroku一样的黑盒的公开的云平台确实很有意义；但是这个做法在一个大型组织里面，需要把应用部署到一个私有云上面通常会不奏效

此外，我认为人类不应该部署应用程序，至少不应该部署到他们自己的工作站或实验室以外的任何环境中。如果存在适当的构建管道，应用程序将自动部署到所有适用的环境中，并且可以根据CI工具和目标云平台中的安全限制手动部署到其他环境中

事实上，即使您的云是公共云提供商，仍然可以使用CloudBees或Wercker之类的云托管CD工具来自动化您的测试和部署

尽管总是有例外，但我认为，如果您不能通过按一个按钮来部署，或者不能自动响应某些事件，那么您就错了

资源

当我们坐在办公桌前，在本地开发的时候，由于要快速启动和运行某些东西时，我们就会做出妥协。这些妥协可能会给我们留下一点技术债务，也可能使我们陷入灾难性的失败

其中一个折衷办法我们使用和提供后端服务的方式。我们的应用程序可能需要一个数据库，而且我们知道在生产环境中，我们会将它连接到Oracle或Postgres服务器，但是在本地搭一套数据库太麻烦了，所以我们会妥协并使用类似于目标数据库的内存中的数据库

每当我们做出这样的妥协时，我们的开发环境和生产环境之间的差距就越大；这种差距越大，我们对应用程序行为的可预测性就越低。随着可预测性的下降，可靠性也会下降；如果可靠性下降，我们就失去了敏捷开发持续发布的能力。它给我们所做的每件事都增加了一种脆弱感；最糟糕的是，我们通常不能提前知道增加开发/生产环境差距的后果，直到知道时已经为时已晚

现如今，开发人员有太多的工具去规避环境不一致的问题了，开发者能够通过PaaS平台的一个服务，或者可以通过类似Docker的容器工具来保证他们的本地工作的数据库实例与生产环境几乎一样

当您在构建云原生应用程序的过程中评估开发生命周期中的每一步时，每一个增加部署环境之间的功能差距的决策都需要被标记和质疑，并且您需要抵制通过允许你的环境存在差异来给开发带来便利性的冲动，即使这种差异在当时看来无关紧要

每一个提交都是一个部署候选

本节以及本书的后面将会多次提及这个规则：每个提交都是一个部署候选

在以云计算方式构建应用程序时，每次提交更改时，该更改都应在一段短时间后进入生产环境：基本上是运行所有测试、根据所有集成套件检查更改以及部署到预生产环境所需的时间

如果您的开发环境、测试环境和生产环境不同，甚至是在你认为无关紧要的地方产生了不同，那么你就失去了准确预测代码变更在生产环境中的行为的能力。这种对代码部署到生产环境的信心，对于持续交付、快速部署（允许应用程序及其开发团队在云中蓬勃发展）至关重要

CHAPTER 10

Administrative Processes

第十二个也是最后一个原始的因素说的是“将管理/管理任务作为一次性进程运行”。我觉得这个因素可能会产生误导。管理进程的概念本身并没有什么错，但是依然有很多你不应该使用它们的原因

原始的12要素关于这一点的问题在于，它固执的认为管理进程应该用解释型语言例如Ruby或者交互型编程方式例如shell来编写。管理进程是一种作者希望用户去使用的功能

我认为，在某些情况下，使用管理进程实际上是一个坏主意，你应该始终扪心自问，管理进程究竟是不是你想要的，又或者是否存在更合适你的设计或者架构。可能应该重构为其他内容的管理进程示例包括：

- 数据迁移
- 交互式控制台
- 运行定时脚本，比如一个凌晨的批量任务
- 运行一次性的自定义的程序

首先，让我们看看计时器的问题（通常由Autosys或Cron等应用程序管理）

一种想法可能是将计时器内部化，让应用程序每n小时唤醒一次以执行其批处理操作。从表面上看，这看起来是一个很好的解决方案，但是当有20个应用程序实例在一个可用性区域中运行，而另外15个实例在另一个可用性区域中运行时，会发生什么情况呢？如果它们都在计时器上执行相同的批处理操作，基本上就是在引发混乱，而损坏或重复的数据将只是这种模式产生的众多麻烦之一

又或者用交互式shell促发来解决（指批量处理的功能以REPL的方式暴露），可是这样也有很多问题，其中最大的一个问题是，即使可以访问该shell，也只能与单个实例的临时内存进行交互。如果应用程序是作为无状态进程正确构建的，那么我认为在进程内部暴露REPL几乎没有任何价值

接下来，让我们看看触发定时或批处理管理进程的机制。这通常发生在一些外部计时器刺激（如cron或Autosys）执行shell脚本时。在云中，您不能指望能够调用这些命令，因此您需要找到其他方法来触发应用程序中的特殊活动

在图10-1中，您可以看到一个应用程序的经典企业体系结构，它有自己的常规职责，并通过执行shell脚本支持批处理或定时操作。这在云中显然行不通

Figure 10-1

有几种解决方案可以解决这个问题，但我发现最吸引人的一种解决方案，特别是在将应用程序的其余部分迁移到云平台上时，就是公开一个RESTful端点，该端点可用于调用特殊功能，如图10-2所示。

另一种可选的方法是从主应用程序中提取与批处理相关的代码，并创建一个单独的微服务，它也采用类似于上图中的体系结构

这仍然允许随意调用定时功能，但它会将此操作的促发转移到应用程序之外。此外，此方法还解决了动态扩容实例上内部计时器最多执行一次的问题。批处理操作由一个应用程序实例处理一次，然后您可以与其他后端服务交互以完成任务。保护批处理端点也应该相当简单，这样就只能由授权人员操作。更有用的是，您的批处理操作现在可以弹性伸缩，并利用所有其他云优势

即使是上面的解决方案，在架构上也存在多种形式，一些形式可以让你无需在程序内部暴露批量处理的功能

如果您仍然觉得需要使用管理进程，那么您应该确保您这样做的方式符合云提供商提供的功能。换句话说，不要使用您最喜欢的编程语言来派生新的进程来运行任务；使用以云原生方式运行的一次性任务的程序。在这种情况下，您可以使用类似Amazon Web Services Lambdas的解决方案，这些功能可以按需调用，并且不需要像前面的microser-vice示例中那样在配置好的服务器中运行

CHAPTER 11

Port Binding

因素7表明云原声应用通过端口绑定的方式暴露服务

避免web容器决定端口

Web应用程序，尤其是那些已经在企业中运行的应用程序，通常在某种服务器容器中执行。Java世界充满了像Tomcat、JBoss、Liberty和Web-Sphere这样的容器。其他web应用程序可能在其他容器中运行，如Microsoft Internet Information Server（IIS）

在非云环境中，web应用程序被部署到这些容器中，然后容器负责在应用程序启动时为其分配端口

在管理自己的web服务器的企业中，一种极为常见的模式是在同一容器中承载多个应用程序，通过端口号（或URL层次）将应用程序分隔开来，然后使用DNS在该服务器周围提供用户友好的外观。例如，您可能有一个名为appserver的（虚拟或物理）主机，以及许多已分配端口8080到8090的应用程序。为了避免让用户记住端口号，DNS用来将app1这样的域名与appsever:8080，aap2与appserver:8081相关联，等等

避免微观管理端口分配

拥抱平台即服务（platform-as-a-service）允许开发人员和devops不再需要执行这种微观管理。您的云提供商应该为您管理端口分配，因为它可能还管理路由、扩展、高可用性和容错，所有这些都需要云提供商管理网络的某些方面，包括将主机名路由到端口以及将外部端口号映射到容器-内部端口

端口绑定最初的12个因素之所以使用export这个词，是因为它假定云原生应用程序是自包含web容器的，并且从不注入任何类型的外部应用程序服务器或容器

现有企业应用程序的实用性和性质可能会使以这种方式构建应用程序变得困难或不可能。因此，限制性稍小的准则是，应用程序和应用程序服务器之间必须始终保持1:1的相关性。换句话说，您的云提供商可能支持web应用容器，但它不太可能支持在同一容器中托管多个应用程序，因为这使得持久性、可扩展性和可恢复性变得几乎不可能

现代应用程序的端口绑定对开发人员的影响相当简单：您的应用程序可能以http://localhost:12001运行在开发人员本机上，在QA中，它可能运行在http://192.168.1.10:2000上，生产环境运行在http://app.company.com。使用导出端口绑定开发的应用程序支持这种特定于环境的端口绑定，而无需更改任何代码

应用即后端服务

最后，为允许外部化的运行时端口绑定而开发的应用程序可以充当另一个应用程序的支持服务。这种灵活性，加上在云上运行的所有其他好处，是非常强大的

CHAPTER 12

Stateless Processes

因素6，进程，讨论支持云原生应用的无状态特点

应用程序应作为单个无状态进程执行。正如本书前面提到的，我对管理和辅助进程的使用有很强的看法，现代的云原生应用程序应该由一个单一的无状态进程组成

这与原始的12个因素稍有矛盾，后者的要求更为宽松，允许应用程序由多个进程组成

准确定义无状态

我经常提出的一个问题是无状态的概念是什么？人们想知道如何构建一个没有任何状态的进程，毕竟，每个应用程序都需要某种状态，对吗？即使是最简单的应用程序也会留下一些浮动的数据，所以您怎么可能有一个真正的无状态进程呢？

无状态应用程序在处理请求之前不假设内存内容，也不在处理请求之后假设内存内容。应用程序可以在处理请求或处理事务的过程中创建和使用瞬态，但在客户机收到响应时，这些数据应该已经全部消失

简单地说，所有持久状态都必须在应用程序外部，由后端服务提供。因此，无状态不是说不存在状态，而是说状态不能在应用程序中维护

例如，一个提供用户管理功能的微服务是无状态的，因此所有用户的列表都在后端服务（例如Oracle或MongoDB数据库）中维护。显然，让数据库无状态的是没有意义的

无共享模式

进程通常通过共享公共资源来相互通信。即使不考虑向云端迁移，采用无共享模式也会带来很多好处

首先，进程之间共享的任何东西都是一种责任，这使得所有这些进程都变得更加脆弱。在许多高可用性模式中，进程将通过各种技术共享数据，以选举集群领导者，决定进程是主进程还是备份进程，等等

如果你的程序跑在云上面，你要规避上面这些做法。你的进程可以在没有任何警告的情况下在一瞬间消失，这是一件好事。进程来来去去，水平和垂直扩展，而且是高度可处置性的。这意味着进程之间共享的任何内容也可能消失，从而可能导致连锁失败

文件系统准确的说并不是一个后端服务，这意味着你不能通过文件来共享程序间的数据，云上面的磁盘是瞬时的，并且在某些场景下是只读的

如果进程间需要共享数据，比如web集群间的session状态，那么session状态应该外部化，同时通过一个后端服务存储和读取状态

数据缓存

一种常见的模式，尤其是在长时间运行、基于容器的web应用程序会在进程启动期间缓存频繁使用的数据，这一点本书已经提到，进程需要快速启动和停止，而花很长时间来预热内存缓存违反了这一原则

更糟糕的是，将应用程序必须用到的的数据缓存在内存可能会使应用程序膨胀，使每个实例（应具有弹性伸缩性）占用的RAM远远超过所需要的

有几十种第三方缓存产品，包括Gem-fire和Redis，它们都被设计为应用程序的后端服务用来缓存状态数据。它们可以用于缓存会话状态，但也可以用于缓存进程在启动过程中可能需要的数据，并避免进程之间紧密耦合的数据共享

CHAPTER 13

Concurrency

因素8，并发性，建议我们云原生应用程序应该使用进程模型进行扩展。曾经有一段时间，如果应用程序达到了它的容量极限，那么解决方法就是增加它的大小。如果一个应用程序每分钟只能处理一定数量的请求，那么首选的解决方案就是将应用程序变大

向一个大块的单体应用增加CPU核数，增加内存和其他的资源（虚拟的或者物理的），这种做法被称之为垂直扩展，这种做法在现如今的社会是不受欢迎的做法

一个更加理想的现代化的扩展方式是水平扩展，相比较于把一个本来就很大的进程弄的越来越大，你可以创建多个进程，然后在这些进程中分配负载

大多数云提供商已经完善了这一功能，你甚至可以配置规则，根据负载或系统中可用的其他运行时遥测动态扩展应用程序实例的数量

如果你正在构建一个可处置性的，无状态的，无共享进程，你就能够充分享受云提供的水平扩展的能力，你的应用将能够在云上面以多进程运行的方式茁壮成长

CHAPTER 14

遥测

遥测的概念不在最初的12个因素之列。遥测的字典定义意味着使用特殊设备对某物进行特定测量，然后使用无线电将这些测量传输到其他地方。遥测点到源头的测量是需要通过远程，远距离，无实物连接的方式来完成的

虽然我建议使用比无线电更现代的东西，但是遥测的使用应该是任何云原生应用程序的一个重要部分

在你本地跑程序有一些云上面没有的便利性，你可以执行debug，也可以跑数百个任务去可视化你程序的内部行为

你没有这种直接访问云应用程序内部的能力。你的应用程序实例可能会在几乎没有任何警告的情况下从美国东海岸移动到西海岸。你的应用程序可能启动的时候只有一个实例，几分钟后，你可能会有几百个正在运行的应用程序副本。这些都是非常强大、有用的特性，但它们为实时应用程序监视和遥测提供了一种不熟悉的模式

> 像对待太空探测器一样对待你的应用程序
> 我喜欢把向云上部署应用想像成向太空中发射一个科学探测器

如果你的作品已经跑到了千里之外，你无法在物理上接触它，更没法用锤子捶打的方式去迫使它运动，这种情况下，你想要什么样的遥测呢？你需要什么数据和控制去让你的作平在空中自由漂移呢？

当需要监控你的应用的时候，这里通常会有多种不同类型的数据：

- 应用性能监控
- 特定领域的遥测
- 健康和系统日志

其中第一个是APM，它由事件流组成，云端以外的工具可以使用这些事件来监视应用程序的性能。你的责任是定义和量化你的应用程序的特定的性能指标。用于组成APM仪表盘的数据通常是通用的，可以来自多个业务线的多个应用程序

第二点，特定领域的遥测，也取决于你。这是指对你的业务有意义的事件和数据流，您可以将其用于自己的分析和报告。这种类型的事件流通常被输入到一个“大数据”系统中，用于存储、分析和预测

APM和特定业务遥测会有些许不同，可以这样想：APM可以提供给你每秒的HTTP请求数，然后特定业务遥测可能会告诉你在最近的20分钟你卖出去了多少iPads

最后，健康和系统日志可能是由你的云服务供应商提供的，它们由一系列的事件流组成。比如服务的启动，关闭，扩展，web请求追溯。以及周期性的健康检测的结果

云让很多事情变得简单，但是监控和遥测仍然很困难，可能比传统的企业应用程序监控更困难。当你盯着消防水管看着水流，流里面包含定期健康检查、请求审核、业务级别事件、跟踪数据和性能指标的数据时，这是一个难以置信的数据量

在规划监控策略时，你要考虑清楚到底需要聚合多少信息、信息传入的速率以及要存储多少信息。如果应用程序从1个实例动态扩展到100个实例，那么也会导致日志流量增加百倍

审计和监控云应用程序是那种常常被忽视但是却又很重要的事情，特别是当你构建的是真正产品级别的应用程序。如果你不会盲目地将卫星送入轨道而没有办法对其进行监控，你就不应该对你的云应用程序做同样的事情

正确的遥测你的应用程序是你的应用程序在云上面运行成功或者失败的关键

CHAPTER 15

Authentication and Authorization

原始的12 因素里面没有讨论安全，认证，授权

安全是任何应用程序和云环境的重要组成部分，安全绝不应是事后诸葛亮

很多时候，我们过于关注应用程序的功能需求，以至于忽略了交付任何应用程序最重要的一个方面(安全)，而不管该应用程序是面向企业、移动设备还是云

云原生应用程序应该是一个安全的应用程序。你的代码，无论是编译的还是原始的，都会跨多个数据中心传输，在多个容器中执行，并被无数客户端访问，这些客户端有一些是合法的，但是绝大多数是邪恶的

即使您在应用程序中实现安全性的唯一原因是需要对哪个用户做了什么更改做审计追踪，因为这一个理由去加密你的应用程序就已经足够了，这么做可以在花费少量时间精力的情况下获得极大的好处

在理想情况下，所有云原生应用程序都将使用RBAC（基于角色的访问控制）保护其所有端点。对应用程序资源的每个请求都应该知道是谁发出请求，以及该使用者所属的角色。这些角色决定调用客户端是否有足够的权限让应用程序接受请求

有了OAuth2、OpenID Connect等工具，各种SSO服务器和标准，以及各种语言特定的身份验证和授权库，安全性应该从一开始就融入到应用程序的开发中，而不应该是在应用程序在生产环境中运行一段时间之后再加上的一个小功能

CHAPTER 16

A Word on Cloud Native

既然您已经阅读了一篇超越12因素应用程序的讨论，并且了解到人们经常交替使用“12 factors”和“cloud native”，那么值得花点时间讨论一下cloud native这个术语

什么是云原生?

像“SOA”、“云原生”和“微服务”这样的流行语和短语都是因为我们需要一种更快、更有效的方式来交流我们对某个主题的看法。这对于促进复杂话题上有意义的对话至关重要，我们最终会建立一个共享的背景或共同的语言

这些流行语的问题在于，它们依赖于多方之间的相互理解或共同理解。就像经典的“电话传话”游戏一样，这种所谓的共同理解随着信息的传播逐步迅速恶化为相互之间的困惑

我们在SOA（面向服务的体系结构）中看到了这一点，并在云原生概念中再次看到了这一点。似乎每次共享此概念时，含义都会改变，然后我们对云原生的理解会越来越多

要理解“云原生”，我们必须首先理解“云”。许多人认为“云”是公开、不受限制地接触互联网的同义词。虽然有一些云产品属于这种类型，但是这离完整的定义相差甚远

在本书的上下文中，云就是指PaaS(平台即服务)。PaaS提供者公开了一个平台，该平台对应用程序开发人员隐藏了基础设施的详细信息，该平台位于基础设施即服务（IaaS）之上。PaaS提供商的例子包括googleappengine、Redhat openshift、Pivotal Cloud Foundry、Heroku、AppHarbor和Amazon AWS

关键的一点是，云不一定是public的同义词，企业程序也可以运行在自己的数据中心、在自己的IaaS之上或在第三方IaaS提供商（如VMware或Citrix）之上建立自己的私有云

接下来，我对短语“cloud native”中的“native”一词表示异议。这造成了一种误解，即只有新开发的符合云部署流程的应用程序才能被视为云原生应用程序。这是完全不现实的，但由于“云原生”这个短语现在无处不在，并且在大多数IT圈中迅速扩散，我不能使用“云友好型”、“云就绪型”或“云优化型”这样的短语，因为它们既没有现在流行的原始短语那么吸引人，也没有得到广泛认可变成我们的方言。下面是我对云原生应用程序的一个简单定义：

> 一个云原生应用是一个被设计成并且最终运行在PaaS平台上的并且可以任意水平扩展的应用程序

一旦开始为一个概念添加更多细节，你就必须开始从其他人的角度来看待什么是cloud native，你可能陷入“实用主义与纯粹主义”的争论当中（本章稍后讨论）

什么是云原生?

不久前，部署应用程序还需要跑到空调机房里面找到某个具体的物理服务器，然后安装应用程序

裸机部署充满了问题和风险：我们无法动态扩展应用程序，部署过程很困难，硬件的更改可能会导致应用程序失败，而硬件故障通常会导致大量数据丢失和严重停机

这导致了虚拟化革命。每个人都同意裸机不再是发展的方向，于是hypervisor诞生了。业界决定在硬件上加一层抽象层，这样我们就可以更容易地进行部署，横向扩展我们的应用程序，并希望能够防止大量的停机和硬件故障带来的损失

在当今这个设备和软件不断智能化，并且不断互联的世界里，你可能很难找到一家不以软件开发作为公司重点的公司了。即使是在传统的制造业，存在一些需要制造硬件的物理的产品，可是如果没有软件，制造业也不会发生。没有软件，人类就无法被组织起来去高效的构建大规模的产品，没有软件，你当然无法参与全球市场

不管你在哪个行业，如果没有快速交付可靠软件的能力，你就无法在当今的市场上竞争。它需要能够动态扩展以处理以前闻所未闻的大量数据。如果你不能处理大数据，你的竞争对手就会。如果您不能生产出能够处理大量负载、保持响应能力和像市场一样快速变化的软件，您的竞争对手将找到一种方法

这让我们了解了云原生的本质。那种公司有大量的时间和精力花费在运维工作上，结果构建和维护出来的的却是脆弱的基础设施，以及提心吊胆的每月进行一次发布的时代已经一去不复返了

今天，我们需要专注于更加重要的事情（指的是业务相关），我们要做的比竞争对手更好，然后让平台来满足我们的非功能性需求。吉姆·柯林斯在他的《Good to Great》（HarperBus-iness）一书中提出了一个问题：你是一只刺猬，还是一只狐狸?

希腊诗人和唯利是图的阿奇洛胡斯首先讨论了这个概念，他说：“狐狸知道很多事情，但刺猬知道一件大事。”这句话的核心迫使我们审视我们的时间和资源花费在了哪里，并且想一想花费时间和资源的事情到底是不是我们最重要的事情。你的公司或团队想要完成什么？很可能，您没有用故障转移、快速回复、弹性可伸缩性或自动部署之类的东西来回答这个问题。不，你想做的是把你和其他人区别开来。你想构建一个能让你立于不败之地的东西，把其他的琐碎的事情都留给别人去做

现在是云时代，我们需要以一种拥抱云的方式构建我们的应用程序。我们需要把大部分时间花在刺猬（一件大事）上，让别人或其他人来处理狐狸的许多小事。以最快的速度上线你的产品是如今的市场要求我们必须做到的；它是避免我们被自己的的竞争对手抛在后面的必要条件。我们希望能够将我们的资源投入到我们的业务领域，让其他专家去做那些比我们做得更好的事情

通过拥抱云原生架构，遵循一切皆是服务原则构建出来的并且部署到云上面的应用将能够享受到极大的好处。问题不是为什么要拥抱云原生？问题是你为什么不去拥抱云原生？

纯粹主义者vs实用主义者

从SOA到REST的所有模式，以及介于两者之间的所有模式，既有象牙塔顶上闪耀的理想，也有现实世界中的开发团队、预算和约束的现实。诀窍在于决定哪些理想是你不会让步的，哪些理想你会稍微妥协，以满足实际的需要，让产品按时上线

在这本书中，我提到了对理想的妥协在哪里是可能的，甚至是常见的，但是也清楚地表明了依据我们的经验哪些地方是不能妥协的。最终决定权在于你，如果我们创建的每个应用程序都是一个从未违反本书中任何一条指导原则的纯云原生应用程序，我们会非常高兴，但现实和经验表明，对纯粹主义理想的妥协就像死亡和税收一样永远存在

在规划和实施云原生应用程序时，学习何时何地在本书中的指导原则上妥协可能是最重要的一项技能，而不是完全照搬所有原则或者完全抛弃这些原则

CHAPTER 17

Summary

12 factors 应用程序是构建在云中运行的应用程序的一个很好的开端，但是要构建真正在云中蓬勃发展的云原生应用程序，你需要看看 beyond the 12 factors

我对你提出的的挑战是：根据本书中提出的指导原则评估现有的应用程序，并开始计划如何让它们在云中运行。抛开所有其他好处不谈，最终，一切都将是基于云的，就像今天一切都在虚拟化上运行一样

如果你正在构建一个新的应用，必须硬性规定它是以云原生的方式构建出来的

拥抱持续集成、持续交付和设计在云上蓬勃发展的应用程序，你所收获的好处将远远超过云原生世界本身带来的好处

关于作者

> 译者说明:本节在原书的最后面，这里将它放到前面，是为了让读者先去了解一下作者

Kevin Hoffman是Pivotal Cloud Foundry的解决方案架构师，他在那里培训组织成员，教会他们如何构建云原生应用程序、如何将应用程序迁移到云，以及主动去拥抱云和微服务的方方面面。他为几乎所有类型的行业编写了应用程序，包括四架直升机的自动驾驶仪软件、废物管理、金融服务和生物识别安全

在业余时间，当他不编写代码、修复bug或学习新技术的时候，他还写幻想和科幻小说

项目主页

- beyond-twelve-factors-app 中文版