Python 微服务开发

[法] 塔里克 • 齐亚德(Tarek Ziadé) 著

和坚　张渊　　译

清华大学出版社

北　京

Copyright Packt Publishing 2017. First published in the English language under the title 'Python Microservices Development – (9781785881114).

北京市版权局著作权合同登记号　图字：01-2018-4395

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

Python微服务开发 / (法)塔里克 • 齐亚德(Tarek Ziadé) 著；和坚，张渊 译. —北京：清华大学出版社，2019
书名原文：Python Microservices Development
ISBN 978-7-302-52412-0

Ⅰ. ①P… Ⅱ. ①塔… ②和… ③张… Ⅲ. ①软件工具—程序设计 Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字(2019)第 041781 号

责任编辑：王　军　韩宏志
封面设计：周晓亮
版式设计：孔祥峰
责任校对：牛艳敏
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：北京鑫丰华彩印有限公司
装 订 者：三河市溧源装订厂
经　　销：全国新华书店
开　　本：170mm×240mm　　**印　　张**：17.75　　**字　　数**：358 千字
版　　次：2019 年 4 月第 1 版　　**印　　次**：2019 年 4 月第 1 次印刷
定　　价：59.00 元

产品编号：080803-01

译 者 序

近年来，“微服务”技术风靡全球。随着传统互联网和移动互联网的蓬勃发展，企业在快速迭代中积累了大规模服务化开发和运维经验。为通过提高 IT 的响应能力来提升竞争力，微服务架构成为传统企业的“救命稻草”。开发团队在变革中改造和重建技术架构，企业管理者们切实感受到微服务带来的巨大好处。

但不可否认，微服务架构带来了额外复杂性。对开发者提出了更高的要求，开发者不仅要编写业务代码，还需要具有部署和运维等能力。

十多年前，当 Ruby on Rails 和 Django 发布时，人们热衷于追逐包罗万象、开箱即用的全栈式 Web 框架，只需要运行几行脚手架命令，就能快速编写一个包含 Web 页面和数据存储的 Todo List 应用。但时至今日，那些小巧灵便的框架变得越来越受欢迎，开发者更愿意选择“微框架”，通过谨慎地综合运用不同工具来开发应用；随着开发的进行，灵活地升级或替换其中的某些部分。Flask 即是这种微框架之一。

本书模拟真实场景，从一个单体 Flask 应用开始，提出问题，分析和比较方案，作出权衡，最终解决问题(可能引出又一个“问题”——但并非当前的优先级)，逐渐拆分出多个微服务，解决随之而来的部署、监控、安全等新问题，在最后利用异步编程优化性能。期间牵涉大量工具，但本书并未详细介绍它们，只是“点到为止”。本书内容紧贴实用，面向想要开阔眼界和动手实践的开发者和架构师。通过阅读本书，读者将能对微服务开发实践有系统性认识，以便在开发早期进行规划和技术选型。

这里要感谢清华大学出版社的编辑们，他们为本书的翻译投入了巨大热情并付出了很多心血。没有他们的帮助和严谨的要求，本书不可能顺利付梓。

本书涉及大量的实践和专业术语，虽然译者倾力而为，力求译文准确易懂。但毕竟水平有限，失误在所难免，如有任何意见和建议，请不吝指正！本书主要章节由和坚和张渊翻译，参与本书翻译的还有张坤、吴邦、刘易斯、赵汝达、何睿智、刘娟娟、罗冬哲等。

最后，希望读者能通过本书对微服务开发有更深入的理解，并能继续探索解决已知问题的工具和方法。

作 者 简 介

Tarek Ziadé是一位 Python 开发人员，在 Mozilla 的服务团队工作，已使用法语和英语撰写多本 Python 书籍。Tarek 创建了一个名为 Afpy 的法国 Python 用户组，现居住在法国第戎市郊区。在工作之余，Tarek 不忘陪伴家人。他另有两个爱好：跑步和吹小号。

可访问 Tarek 的个人博客(Fetchez le Python)，并在 Twitter 上关注他(@tarek_ziade)。还可在亚马逊上找到他撰写的另一本书 *Expert Python Programming*，该书已由 Packt 出版。

感谢Packt团队，以及帮助过我的以下技术精英：Stéfane Fermigier、William Kahn-Greene、Chris Kolosiwsky、Julien Vehent和Ryan Kelly。

感谢 Amina、Milo、Suki 和 Freya 给予我的爱和耐心支持。

希望在阅读时，你能享受到和我写本书时同样的乐趣！

审校者简介

自 20 世纪 90 年代末以来，William Kahn-Greene 一直在编写 Python 代码和构建 Web 应用。

他在 Mozilla crash ingestion pipeline 的 crash-stats 小组工作，并维护着多种 Python 库，如 bleach。在等待 CI 测试代码改动时，William 会摆弄木制品，照料他种的番茄，并烹饪 4 个人的饭食。

序　言

7 年前，当我开始在 Mozilla 工作时，为一些 Firefox 功能编写 Web 服务。它们中的一些最终蜕变成微服务。这种变化是随着时间的推移逐渐发生的。促成这种转变的第一个因素是，我们将所有服务转移到云厂商上，并开始与一些第三方服务交互。在云服务上托管应用时，微服务架构成为自然之选。另一个驱动因素是 Firefox 的 Account 项目。我们想在 Firefox 上为用户提供独立身份，以便用户与我们的服务交互。这样一来，所有服务必须与同一个身份提供方(Identity Provider)交互，一些服务器端部分开始重新设计为微服务，以便更高效地工作。

许多 Web 开发者有类似经历，或正在经历这个过程。我也相信 Python 是用来编写小型和高效微服务的最佳语言。Python 生态系统生机勃勃，最新的 Python 3 的特性让它在这个领域中能与过去 5 年中迅猛发展的 Node.js 一决高下。

这就是本书的全部内容。我想分享自己使用 Python 编写微服务的经验，并为此创建了一个简单示例——Runnerly。它位于 GitHub，可供你学习。你可在 GitHub 上与我直接交流，请指出你看到的任何错误，我们可共同切磋如何编写优秀的 Python 应用。

前　言

为将 Web 应用部署到云，代码需要与很多第三方服务进行交互。使用微服务架构，可构建能管理这些交互的大型应用。但这带来一系列挑战，每项挑战都有独特的复杂性。这本通俗易懂的指南旨在帮助你克服这些挑战。书中将介绍如何以最合理的方式设计、开发、测试和部署微服务，紧贴实用的示例将帮助 Python 开发者用最高效的方式创建 Python 微服务。阅读完本书，读者将掌握基于小型标准单元构建大型应用的技能。本书将使用成熟的最佳实践，并分析如何规避常见陷阱。此外，对于正将单体设计转换成新型“微服务”开发范式的社区开发者来说，本书也颇具价值。

本书内容

第 1 章“理解微服务”定义什么是微服务，以及微服务在现代 Web 应用中扮演的角色。还介绍 Python，并解释为什么用 Python 构建微服务是上佳之选。

第 2 章“Flask 框架”介绍 Flask 的主要特性。通过一个 Web 应用示例来展示这个框架，Flask 是构建微服务的基础。

第 3 章“良性循环：编程、测试和写文档”，介绍测试驱动开发方法和持续集成方法，以及在构建和打包 Flask 应用的实践中如何使用这些方法。

第 4 章“设计 Runnerly”基于应用特性和用户案例，首先构建一个单体应用，然后讲述如何将其拆解成微服务，并实现微服务之间的数据交互。还将介绍用来描述 HTTP API 的 Open API 2.0(ex-Swagger)规范。

第 5 章“与其他服务交互”介绍一个服务如何与后台服务进行交互，如何处理网络拆分问题，以及其他交互问题，另外介绍如何独立地测试一个服务。

第 6 章“监控服务”介绍如何在代码中添加日志和指标，清晰地掌控全局，确定发生了什么，并能追查问题和了解服务利用率。

第 7 章“保护服务”介绍如何保护微服务，如何处理用户身份验证、服务间身份验证以及用户管理。还介绍针对服务的欺诈和滥用，以及如何缓解这些问题。

第 8 章“综合运用”描述在终端用户界面中，如何设计和构建一个使用微服务的 JavaScript 应用。

第 9 章“打包和运行 Runnerly”描述如何打包、构建和运行整个应用。开发者必须能够将应用打包到一个开发环境中，确保所有部分都可以运行。

第 10 章“容器化服务”解释什么是虚拟化，如何使用 Docker，如何将服务做成 Docker 镜像。

第 11 章“在 AWS 上部署”首先介绍当前的云服务厂商和 AWS 世界。然后演示如何使用 AWS 来实例化一个基于微服务架构的应用。另外介绍 CoreOS，这是一个专门用于在云上发布 Docker 容器的 Linux 分支。

第 12 章“接下来做什么？”总结全书，在如何构建独立于云厂商和虚拟化技术的微服务问题上，给出一些提示来避免将鸡蛋放入同一个篮子里。还将帮助你巩固第 9 章中学到的知识。

阅读本书需要准备什么

要执行本书的命令和应用，系统需要安装 Python 3.*x*、virtualenv 1.*x* 和 Docker CE。正文中也会根据需要详细列出安装说明。

读者对象

作为一名开发者，如果你了解 Python 基本概念、命令行，以及基于 HTTP 的应用设计原则，并想学习如何构建、测试、扩展和管理 Python 3 微服务，那么本书适合你。阅读本书，你不必具有用 Python 编写微服务的任何经验。

本书约定

代码块按以下样式显示：

```
import time

def application(environ, start_response):
   headers = [('Content-type', 'application/json')]
   start_response('200 OK', headers)
return bytes(json.dumps({'time': time.time()}), 'utf8')
```

会用粗体来显示需要重点关注的代码：

```
from greenlet import greenlet
   def test1(x, y):
   z = gr2.switch(x+y)
print(z)
```

任何命令行的输入或输出都按以下样式显示：

```
docker-compose up
```

警告或重要注释会这样显示。

提示和技巧会这样显示。

读者反馈

欢迎读者提出反馈意见，这样我们能了解你对本书的看法，喜欢什么或不喜欢什么。反馈意见很重要，能帮助我们开发读者真正想了解的主题。只需要发邮件给feedback@packpub.com，并在邮件标题中提及本书，即可将反馈意见发给我们。

如果你是某个主题的专家，有兴趣写书，或愿意为写书做贡献，请到www.packtpub.com/authors 页面查阅作者指南。

下载示例代码

本书相关的代码放在 GitHub 上，网址是 https://github.com/PacktPublishing/Python-Microservices-Development。还有其他代码包和视频，欢迎通过 https://github.com/PacktPublishing/页面下载。

另外，读者可扫描本书封底的二维码直接下载代码。

下载文件后，用以下工具的最新版本来解压缩：

- 在 Windows 系统中使用 WinRAR /7-Zip。
- 在 Mac 系统中使用 Zipeg /iZip /UnRarX。
- 在 Linux 系统中使用 7-Zip /PeaZip。

勘误

尽管我们已经非常小心地确保内容的准确性，但还是会发生失误。如果你在书中发现了错误，可能是文本错误或代码错误，你能向我们报告此事，我们将不胜感激。通过这样做，可减少其他读者的阅读痛苦，并帮助我们改进本书的后续版本。如果你发现任何勘误，请访问 http://www.packtpub.com/submit-errata 页面来报告它们，选择你购买的书，单击 Errata Submission Form 链接，输入勘误的详细信息。一旦填写的勘误被确认，你的提交将被接受，然后勘误将被上传到我们的网站上，或添加到任何现有的勘误列表中。现有的勘误列表位于 Errata 标题的下面。

要查看之前提交的勘误，可访问 https://www.packtpub.com/books/content/support，然后在查找输入框内输入书名。要查找的信息会显示在 Errata 下面。

盗版行为

互联网上的盗版行为是所有媒体一直头疼的问题。在 Packt，我们将尽力处理盗版问题。我们会非常认真地对待版权和许可证的保护。如果你在互联网上遇到任何我们作品的非法拷贝，请立即向我们提供网址或网站名称，以便我们能采取补救措施。

请通过 copyright@packtpub.com 联系我们，并附带上有侵权嫌疑的材料。

非常感激你能帮助保护我们的作者以及我们的工作。这样我们可持续为你带来有价值的内容。

问题

关于本书的任何问题，欢迎通过 questions@packtpub.com 联系。

目　录

第 1 章

理解微服务

软件行业一直在尝试改良软件构建方式。不用说，与穿孔卡片时代相比，当今的软件构建过程已改良了很多。

微服务是过去几年里涌现出的一种改良方法，部分原因是很多公司想缩短发布周期。他们希望能尽快向客户交付新产品或新特性，希望通过迭代达到“敏捷”目的，希望能做到交付、交付、再交付。

如果有大量客户正使用你的服务，相对于发布之前反复测试产品，直接在运行的产品上推送一个试验性功能或移除一项无用的功能是更好的实践方法。

现在，Netflix 等公司正在倡导持续交付技术，这是一种可让小改动频繁上线，然后在一小部分用户中进行测试的技术。他们开发了很多工具，例如 Spinnaker(http://www.spinnaker.io/)就是通过自动执行尽可能多的步骤来更新生产环境，改动的特性通过相互独立的微服务发布到云上。

但如果阅读 Hacker News 或 Reddit，你会发现，梳理出哪些概念真正有用，哪些概念只是随波逐流的新闻体裁是非常困难的。

“写一篇承诺救赎的论文，使它成为‘结构化’或‘虚拟化’的东西，或使用抽象、分布式、高阶、可适用等概念，你几乎肯定在宣扬一门新邪教。”

——Edsger W. Dijkstra

本章将讲解什么是微服务，然后重点介绍多个使用 Python 实现微服务的方法。本章要点如下：

- 面向服务架构
- 使用单体方式构建应用
- 使用微服务方式构建应用

- 微服务的益处
- 微服务的缺陷
- 使用 Python 实现微服务

希望当读到本章结尾时，你能深入了解微服务的构建，并明白微服务是什么，以及如何使用 Python。

1.1 SOA 的起源

关于微服务有很多种定义，并没有一个官方标准。在试着解释微服务时，人们通常会提到面向服务架构(Service-Oriented Architecture，SOA)。

SOA 早于微服务，其核心原则是将应用组织成一个独立的功能单元，可远程访问并单独进行操作和更新。

——Wikipedia

上述定义中的每个单元都是一个独立服务，它实现业务的一个方面，并通过接口提供功能。

虽然SOA清楚地指出服务应当是独立的进程，但并未强制使用哪种协议进行交互，对如何部署和编排应用还是相当模糊的。

在少数专家于 2009 年发布的 SOA 宣言(http://www.soa-manifesto.org)中，甚至没有提及服务是否通过网络进行交互。

SOA服务可在同一个机器上使用套接字(socket)通过IPC(Inter-Process Communication，进程间通信)方式来交互，如使用共享内存、间接消息队列或远程过程调用(Remote Procedure Call，RPC)。选项非常广泛，只要没有在单个进程中运行所有应用，SOA就可以是任何东西。

常见的说法是，过年几年开始涌现的微服务是 SOA 的一种特定实现方式。它们实现了 SOA 的一些目标，也就是用独立组件来构建应用，组件之间进行着交互。

如果想给出微服务的完整定义，最好先分析一下大多数软件是如何设计架构的。

1.2 单体架构

让我们先通过一个非常简单的例子来介绍传统的单体应用：一个酒店预订网站。

除了静态的 HTML 内容，网站有一个预订功能，可让全球任何城市的用户通过网站预订酒店。用户可搜索酒店，然后用信用卡付款。

当用户搜索酒店网站时，应用将执行以下操作：

(1) 针对酒店数据库执行一些 SQL 查询。

(2) 给合作伙伴的服务发送 HTTP 请求，将更多酒店添加到列表中。

(3) 使用 HTML 模板引擎生成 HTML 结果页面。

一旦用户找到满意的酒店并单击“预订”，应用将执行以下步骤：

(1) 如有必要，在数据库中创建客户，然后进行身份验证。

(2) 通过与银行网络服务交互来完成付款。

(3) 按法律要求，应用需要将支付详情保存到数据库。

(4) 使用 PDF 生成器生成收据。

(5) 用电子邮件服务向用户发送一份用于确认的电子邮件。

(6) 用电子邮件服务将预订电子邮件转发给第三方酒店。

(7) 在数据库中添加用于追踪订单的条目。

上面是一个简化的过程，但紧贴实用。

应用和数据库的交互包括酒店信息、预订信息、支付信息和用户信息等。它还与外部服务进行交互来发送邮件，完成支付，从合作伙伴获取更多酒店。

在经典的 LAMP(Linux-Apache-MySQL-Perl/PHP/Python)架构中，每个传入的请求都会在数据库生成关联的 SQL 查询，以及少量对外部服务的网络请求，然后服务器使用模板引擎生成 HTML 响应。

图 1-1 描述了这种中心化架构。

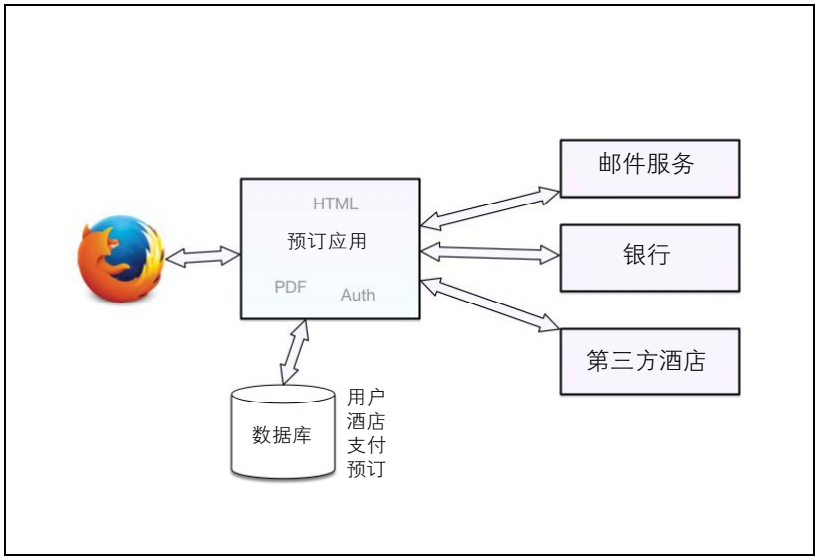

图 1-1　中心化架构

这是一个典型的单体应用，它有很多显而易见的好处。

最大的好处是整个应用程序在一个代码库中，这样开始项目编码就变得十分简单。很容易构建良好的测试覆盖率，还可在一个代码库内以干净和结构化的方式组织代码。将所有数据存储到单一数据库中也简化了应用的开发。可调整数据模型以及代码查询它的方式。

部署也很容易：可给代码库打标签，构建应用包，然后运行它。如果需要扩大规模，可运行多个预订应用的实例，并使用复制机制建立多个数据库。

如果应用一直都很小，这种模型将非常好用，对于单个团队来说，维护也很容易。

但项目通常都会增长，都比最初计划的要大。在一个代码库中维护整个应用会遇到很多棘手的问题。例如，如果要进行一次大范围的彻底修改，如更改银行服务或数据库层，则整个应用会陷入不稳定状态。这些改变在项目生命周期中是个较大的问题，只有通过大量的额外测试才能部署一个新版本。一个项目的生命周期中，这样的改变难免发生。

由于系统的不同部分要求不同的正常运行时间和稳定性，因此一些小变化也会产生附带破坏。例如创建 PDF 出错而导致服务器崩溃，会将付款和预订流程置于风险中，很明显这是存在问题的。

失控性增长是另一个问题，应用迅速添加了很多新特性，不断有开发者离开或加入项目，代码结构变得混乱不堪，测试速度越来越慢。通常，这种增长的最终结果是一个难以维护的意大利面条式的代码库，每次当开发者重构数据模型时，“长毛”的数据库都需要一个复杂的数据迁移计划。

大型软件项目通常需要经历数年时间才能走向成熟，此后，会慢慢地变得难以理解和陷入混乱，最终很难进行维护。这不是因为开发者水平糟糕导致的，而是因为复杂度在增加，很少有人完全理解他们所做的每一个小改动会产生的影响，他们只试图在代码库的某个角落孤立地工作。当从 1 万英尺的高空鸟瞰项目时，看到的只有混乱。

这些都是我们亲身经历过的。

过程是很痛苦的，一个项目开始时，开发者梦想能用最新的架构来构建应用。但紧接着，他们通常会再次陷入同样的困局——熟悉的场景再次上演。

下面总结一下单体应用的优缺点：

- 用单体模式开始一个项目是容易的，可能还是最好的方法。
- 中心化的数据库简化了数据的设计和组织。
- 部署应用较简单。
- 对代码的任何改动会影响原本不相关的功能。对某部分的错误修改可能导致整个应用的崩溃。

- 扩展应用的解决方案存在限制：可部署多个实例，但若其中一个特定功能占用了所有资源，则会影响整个应用。
- 随着代码库的增长，很难保证代码的干净和可控性。

当然也有一些办法可避免上述问题。

常见的解决方案是将应用拆分为不同的部分，而最后生成的代码仍将在单个进程中运行。开发人员通过使用外部库或框架来重构应用从而做到这一点。这些工具可以是内部的，或来自开源软件(Open Source Software，OSS)社区。

如果使用诸如 Flask 的框架在 Python 中构建 Web 应用，可将焦点放在业务逻辑上。最吸引人的地方是能把自己的代码外部化，变成 Flask 的扩展和较小的 Python 包。将代码拆分是控制应用程序增长的好方法。

“小而美”

——UNIX 哲学

例如，可使用 Reportlab 和一些模板，将酒店预订应用中的 PDF 生成器拆分成 Python 包。

这个软件包可在其他一些应用中重用，甚至可发布到 Python 包索引(PyPI)中。

但你构建的依然是一个单体应用，很多问题依然存在。例如无法按照不同的部分扩展，缺陷依赖会导致任何间接错误。

还会因为构建时使用了依赖而遇到新挑战。其中一个问题是依赖地狱，如果应用的一部分使用了某个工具库，但 PDF 生成器只能用这个工具库的特定版本，最终将不得不使用一些怪异的解决方案来处理，甚至在分支上定制开发一个修复。

当然本节中描述的所有问题都不可能在项目的第一天出现，而是随着时间推移慢慢堆积起来。

下面看看如果使用微服务来构建相同的应用，会是什么样的。

1.3　微服务架构

如果使用微服务构建相同功能的应用，就可用拆分出的多个组件来管理代码，每个组件运行在独立的线程中。我们不需要使用单一应用负责所有事项，而是如图 1-2 所示拆分成多个微服务。

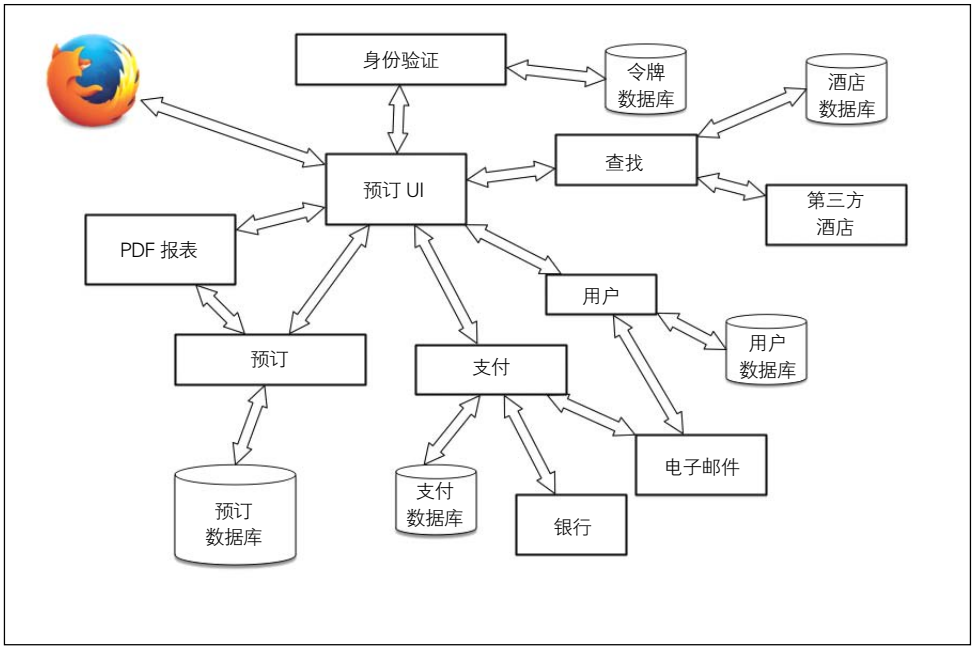

图 1-2　拆分成多个微服务

图中显示的组件数量较多，但不必望而生畏。在单体应用中，内部交互只对内部的某个部分可见。我们已经转移了一些复杂性，最终得到 7 个独立组件：

(1) 预订 UI：一个前端服务，用来生成 Web UI 界面，会与其他所有微服务发生交互。

(2) “PDF 报表”服务：一个非常简单的服务，通过给定的模板和数据把收据或者任何文档创建成 PDF。

(3) 查找：一个可根据城市名查询酒店列表的服务，这个服务有自己的数据库。

(4) 支付：一个和第三方银行服务交互的服务，管理记账数据库。支付成功时会发送电子邮件。

(5) 预订：存储预订信息，并生成 PDF。

(6) 用户：存储用户信息，通过电子邮件和用户交互。

(7) 身份验证：一个基于 OAuth2 来返回身份验证令牌的服务，每个微服务都可在请求其他服务时用它进行身份验证。

这些微服务，连同诸如电子邮件的外部服务，将提供和单体应用相同的功能集。这个架构中的每个组件都使用 HTTP 协议进行通信，通过 REST 风格的 Web 接口提供服务。

由于每个微服务都在内部处理自己的数据结构，所以不需要中心化的数据库，使

用和语言无关的格式(如 JSON)输入和输出数据。可使用任何程序语言都能生成和使用的 XML 或 YAML 格式，最后通过 HTTP 请求和响应进行传输。

“预订UI”服务有些不同，因为它主要用来生成UI页面。依赖于UI使用的前端框架，“预订UI”服务输出的可能是混合的HTML和JSON；如果使用基于静态JavaScript的客户端工具直接在浏览器中生成界面，甚至可以是普通JSON。

除了这个特殊的 UI 情形，使用微服务架构设计的 Web 应用由多个使用 HTTP 进行交互的微服务组成。

这种场景下，微服务是聚焦于特定任务的逻辑单元。这里尝试给出一个完整定义：

微服务是一个轻量级应用，它通过定义良好的契约提供一组有限的功能。它是具有单一责任的组件，可独立开发和部署。

此定义没有提及 HTTP 或 JSON，因为也可以考虑一个基于 UDP 来交换二进制数据的微服务。

但在本书的案例中，所有微服务都是使用 HTTP 协议的简单 Web 应用，都使用和生成 JSON(UI 情形除外)。

1.4　微服务的益处

虽然微服务架构看起来比单体架构复杂得多，但其益处颇多：

- 分离团队的关注点
- 处理更小的项目
- 更多的扩展和部署选项

下面将详细讨论这些内容。

1.4.1　分离团队的关注点

首先，每个微服务可由一个团队独立开发。例如，构建“预订”服务可以是一个完整项目。只要有一个良好的 HTTP API 说明文档，负责开发的团队可使用任何编程语言和数据库。

其次，这意味着应用的演进比单体架构更容易控制。例如，如果支付系统更改其与银行的交互，则影响范围只限于该服务内部，其余应用将保持稳定，甚至完全不受影响。

这种松耦合极大地提高了整个项目的开发速度，从服务层面讲，这是一种类似于

“单一职责原则”的哲学逻辑。

单一职责原则由 Robert Martin 定义，一个类应该只有一个令其改变的原因。换句话说，每个类应该提供单一的、定义良好的功能。在微服务层面，每个微服务都应该聚焦在单个角色上。

1.4.2 更小的项目

第二个益处是降低了项目的复杂度。例如向应用添加一个“PDF 报表”功能时，即便代码很干净，仍会让代码库变得更大，更复杂，有时还会更慢。而在单独应用中构建该功能可避免此问题，因为可更容易地使用任何工具来编程。可频繁地重构它，缩短发布周期，聚焦在最重要的事项上。应用的增长尽在掌控中。

在完善应用时，小项目还能减少风险：如果团队想尝试最新的编程语言或框架，他们可在实现相同微服务 API 的原型上通过快速迭代来试一下，然后决定是否继续使用新的编程语言或框架。

一个浮现在我脑海中的真实例子是 Firefox 的同步存储微服务，通过一些实验，从当前的 Python+MySQL 实现切换到基于 Go 语言的实现，会将用户的数据存储在独立的 SQLite 数据库中。该原型具有很高的实验性质，但由于我们已将存储功能和定义良好的 HTTP API 隔离在一个微服务中，很容易让一小部分用户试用新方案。

1.4.3 扩展和部署

最后，将应用程序拆分为组件更便于在限制下进行扩展。假设每天都有很多顾客预订酒店，而 PDF 生成开始消耗更多 CPU。这时你可将这个特定服务部署到具有更大 CPU 的服务器上。

另一个典型例子是消耗内存的微服务，例如与诸如 Redis 或 Memcache 的内存数据库进行交互。你可通过将微服务部署到具有更少 CPU、更多内存的服务器上来调整部署。

总之，微服务的益处如下：

- 团队可独立开发每个微服务，使用任何技术栈都没问题。可自行定义发布周期。而全部需要定义的只是与语言无关的 HTTP API。
- 开发者将复杂的应用拆分成逻辑单元，每个微服务只关注将自己的事情做好。
- 由于微服务是独立应用，因此可对部署进行更精细的控制，让扩展变得更容易。

微服务架构有利于解决应用开始增长后可能出现的诸多问题。然而，我们需要意

识到，在实践中，微服务架构也会带来一些新问题。

1.5　微服务的缺陷

如前所述，用微服务构建应用有很多益处，但它并非是万能的。

下面列出在编写微服务时，可能需要处理的主要问题：

- 不合理的拆分
- 更多的网络交互
- 数据存储和分享
- 兼容性问题
- 测试

下面将详细讨论这些问题。

1.5.1　不合理的拆分

微服务体系架构的第一个问题是：它如何被设计出来？在第一次尝试中，团队不可能马上想出完美的微服务架构。诸如 PDF 生成器的微服务是个明显的用例。但是，处理业务逻辑时，在你领悟到如何拆分出正确的微服务集之前，你的代码很可能是摇摆不定的。

通过不断试错，设计才能逐渐趋于成熟。而添加或删除微服务可能比重构单体应用更令人痛苦。如果没有证据表明需要拆分，不必先将应用拆分成微服务。

过早拆分是万恶之源。

如果对拆分的意义心存疑问，保持代码在同一个应用中是安全的选择。因为拆分决定可能是错的，所以晚一点把代码拆分到一个新的微服务中比把两个微服务重新合并到一个代码库更容易一些。

例如，如果你总是必须一起部署两个微服务，或一个微服务的改变会影响另一个数据模型，很可能是你没有正确地拆分应用，这两个服务应该重新合并。

1.5.2　更多的网络交互

用微服务构建应用时，第二个问题是会增加网络交互。而在单体版本中，即使代码变得混乱，所有处理也都在一个进程中，可在不需要调用太多后端服务的情况下生成实际响应，然后返回结果。

对于微服务架构，需要额外注意每个后端服务的调用方式，下面是可能出现的问题：

- 如果由于网络隔离或服务延迟，“预订 UI”服务无法调用“PDF 报表”服务，会有什么后果？
- “预订 UI”服务请求其他服务是同步的还是异步的？
- 这将如何影响响应时间？

我们需要有一个坚定的战略来回答所有这些问题，这个主题将在第 5 章中讨论。

1.5.3 数据的存储和分享

另一个问题是数据的存储和分享，有效的微服务需要独立于其他微服务，理想情况下，不应该共享数据库。那么，这对酒店预订应用意味着什么？

再次引发了许多问题：

- 是否在所有数据库中使用相同的用户 ID，或者每个服务都有独立的 ID 并作为隐藏的实现细节来保存？
- 一旦用户添加到系统中，能否通过诸如“数据抽取”的策略将用户信息复制到其他服务数据库中，这么做是不是过度重复了？
- 如何处理数据删除？

以上都是难以回答的问题，本书将介绍许多不同的方法来解决它们。

在设计基于微服务的应用时，一个最大的挑战是如何在保持微服务隔离的同时尽量避免数据重复。

1.5.4 兼容性问题

另一个问题发生在当功能更改影响到多个微服务时。如果不能向后兼容，而且更改影响到服务之间的数据传输方式，你将遇到很多麻烦。

你部署的新服务能否与旧版本的其他服务一起使用？或者你是否需要一次修改和部署多个服务？这是不是说你可能无意中发现一些应该合并在一起的服务？

良好的版本控制和干净的 API 设计有助于缓解这些问题，本书后面将详细讲解这个问题。

1.5.5 测试

最后，如果要进行端到端的测试并部署整个应用，现在需要测试很多积木一样的

微服务。要有一个强健且敏捷的过程才能高效部署。在开发时要顾及完整应用。你不可能做到只根据其中一个微服务就完整地测试整个应用。

幸运的是，现在有许多工具可帮助部署使用多个组件构建的应用，我们也将在书中学到这些工具。所有这些工具推动了微服务架构的成功和采用；如果没有这些工具，微服务也不会是今天的面貌。

微服务风格架构促进了部署工具的革新，部署工具降低了微服务风格架构的获准门槛。

下面总结一下微服务的缺陷：

- 过早将应用拆分成微服务可能导致架构设计问题。
- 微服务之间的网络交互增加了开销。
- 测试和部署微服务较为麻烦。
- 最大的挑战：不同微服务之间很难共享数据。

本节提到的所有这些缺陷其实都不必过于担心。它们看起来似乎难以应对，传统的单体应用好像更安全一些。但从长远看，通过将项目拆分成微服务，开发和运维工作都变得更容易了。

1.6　使用 Python 实现微服务

Python 是一门神奇的多用途语言。

你可能已经知道，Python 可用来构建很多不同类型的应用程序，从用来执行服务器任务的简单系统脚本，到为数百万用户提供服务的大型面向对象应用。

根据 Philip Guo 在 2014 年发布在美国计算机协会(Association for Computing Machinery)网站上的一项研究，Python 在美国顶尖大学的使用率已经超过 Java，成为学习计算机科学最流行的语言。

这一趋势在软件行业也是如此。Python 现在位列 TIOBE 索引(http://www.tiobe.com/tiobe-index/)的前五名，在 Web 开发领域的市场份额可能更大，因为像 C 这样的语言很少被用来构建 Web 应用程序。

本书假设你已经熟悉 Python 编程语言。如果你还不是一个富有经验的 Python 开发者，可阅读本书作者的另一本书 *Expert Python Programming, Second Edition*，来学习高阶 Python 编程技能。

有些开发者批评 Python 的速度慢，不适合构建 Web 服务。Python 的确有些慢，

但依然是构建微服务的语言选项，许多大公司都乐意使用它。

本节将给出使用不同方法来构建 Python 微服务的背景，还给出关于异步与同步编程的一些深刻见解，最后总结有关 Python 性能的一些细节。

本节包含 5 个部分：

- WSGI 标准
- greenlet 和 gevent 模块
- Twisted 和 Tornado 模块
- asyncio 模块
- 语言性能

1.6.1 WSGI 标准

可以很方便地用 Python 创建和运行 Web 应用，这是 Python 吸引大多数 Web 开发者的原因。

受到公共网关接口(Common Gateway Interface，CGI)的启发，Python Web 社区建立了一个标准，称为 WSGI (Web Server Gateway Interface，Web 服务器网关接口)。有了 WSGI，可更方便地编写 Python 应用来支持 HTTP 请求。

使用这个标准编码时，即可通过 uwsgi 或 mod_wsgi 等 WSGI 扩展，由 Apache 或 nginx 等标准服务器执行项目。

应用只需要处理传入请求并返回 JSON 响应，Python 在标准库中包含了所有实现细节。

使用普通 Python 模块，只需要不到 10 行代码，即可创建一个返回服务器本地时间的功能完备的微服务。下面是代码：

```
import json
import time

def application(environ, start_response):
    headers = [('Content-type', 'application/json')]
    start_response('200 OK', headers)
    return [bytes(json.dumps({'time': time.time()}), 'utf8')]
```

自从 WSGI 协议建立，它就成为一个重要标准，Python Web 社区广泛采用了它。开发者通过编写可挂在 WSGI 应用之前或之后的功能性中间件，在 Web 应用环境中完成不同的事情。

一些 Web 框架，如 Bottle(http://bottlepy.org)，就是专门根据该标准创建的。此后，

很多框架都可在 WSGI 协议下使用。

使用 WSGI 的最大问题是原生同步性。对于每个传入请求，都会调用上述代码中的 application 函数一次，当函数返回时，必须发回响应。这意味着，每次调用 application 函数时，在响应准备好之前，都会阻塞。

对于这种情况下编写的微服务，代码总是需要等待各种网络资源的响应。换句话说，这时你的应用是停顿的，在所有东西准备好之前，客户端会被阻塞。

对 HTTP API 来说，这样做问题不大。我们并非讨论构建双向应用(如基于 Web 套接字的应用)。但是，当你的应用同时收到多个调用请求时会发生什么？

WSGI 服务器允许运行一个线程池，来并发服务多个请求。但你不可能运行几千个线程；一旦线程池耗尽，即便微服务除等待后端服务响应外无所事事，下一个请求还是会阻塞客户的访问。

出于上述原因及其他因素，诸如 Twisted 或 Tornado 的非 WSGI 框架，以及 JavaScript 领域的 Node.js 都大获成功，因为它们完全是异步框架。

在编写 Twisted 应用时，可使用回调来暂停和恢复生成响应的工作。此时，可接受一个新请求并开始处理。该模型极大地缩短了进程的停顿时间。可服务数千个并发请求。当然，这并非说应用会更快地返回每个响应，只是说一个进程可接受更多并发请求，在数据准备好之前，能在请求间进行切换。

在 WSGI 标准中没有一个简单方式可做到同样的事情，虽然社区内争论了多年，但最终没能达成共识。社区最终可能放弃 WSGI 标准。

同时，如果你的部署考虑到 WSGI 标准的“一个请求对应一个线程”限制，那么使用同步框架构建微服务仍然是可能的。

不过还有一个诀窍来提升同步的 Web 应用，这就是 greenlet，下一节将对此进行解释。

1.6.2　greenlet 和 gevent 模块

异步编程的一般原则是，让进程处理多个并发执行的上下文来模拟并行处理方式。

异步应用使用一个事件循环，当一个事件触发时暂停或恢复执行上下文；只有一个上下文处于活动状态，上下文之间进行轮替。代码中的显式指令将告诉事件循环，哪里可暂停执行。这时，进程将查找其他待处理的线程进行恢复。最终，进程将回到函数暂停的地方并继续运行。从一个执行上下文移到另一个称为“切换”。

greenlet 项目(https://github.com/python-greenlet/greenlet)是根据 Stackless 项目构建的程序包，是一个特别的 CPython 实现。

greenlet 是易于实例化的伪线程，可用来调用 Python 函数。在这些函数中，可切

换到另一个函数，即将控制权交给另一个函数。切换是通过事件循环完成的，允许使用类似线程的接口范式来编写异步应用。

下面是 greenlet 文档中的一个例子：

```
from greenlet import greenlet
def test1(x, y):
    z = gr2.switch(x+y)
    print(z)

def test2(u):
    print (u)
    gr1.switch(42)

gr1 = greenlet(test1)
gr2 = greenlet(test2)
gr1.switch("hello", " world")
```

上例中的两个 greenlet 显式地从一个切换到另一个。

为构建基于 WSGI 标准的微服务，如果底层代码使用 greenlet，我们可接受多个并发请求，当知道一个调用将阻塞请求(如 I/O 请求)时，只需要从一个请求切换到另一个。

不过，从一个 greenlet 切换到另一个需要显式地完成，这会导致代码变得混乱，难以理解。这就轮到 gevent 大显身手了。

gevent(http://www.gevent.org/)项目构建在greenlet的上层，能采用隐性方式在greenlet之间自动切换，它还有其他许多功能。

gevent 提供了 socket 模块的协作版本，当 socket 中的一些数据准备好后，会使用 greenlet 自动地暂停或恢复执行。甚至有一个 monkey patch 功能，可自动用 gevent 版本的 socket 来替代标准库 socket。只需要通过一行额外代码就可让你的标准同步代码魔术般地每次都异步使用 socket：

```
from gevent import monkey; monkey.patch_all()

def application(environ, start_response):
    headers = [('Content-type', 'application/json')]
    start_response('200 OK', headers)
    # ...do something with sockets here...
    return result
```

当然这种隐含的魔力是有代价的。为使 gevent 正常工作，所有底层代码都需要与 gevent 补丁兼容。来自社区的一些包可能因此继续阻塞或返回非期望的结果；如果这些包使用 C 语言扩展，并绕过了打上 gevent 补丁的标准库的一些功能，这表现得尤其明显。

不过大多数情况下都能正常工作。而且，与 gevent 配合良好的项目会被标记成绿色，如果一个库不能与 gevent 配合，社区通常会要求库的作者修复成绿色。

例如，在 Mozilla，这用来扩展 Firefox Sync 服务。

1.6.3　Twisted 和 Tornado 模块

如果增加并发请求数量对你构建的微服务很重要，可尝试放弃WSGI标准，而使用诸如Tornado(http://www.tornadoweb.org/)或Twisted(https://twistedmatrix.com/trac/)的异步框架。

Twisted 已经存在多年。要实现相同的微服务，需要编写的代码要略微长一些，如下所示：

```
import time
import json
from twisted.web import server, resource
from twisted.internet import reactor, endpoints

class Simple(resource.Resource):
    isLeaf = True
    def render_GET(self, request):
    request.responseHeaders.addRawHeader(b"content-type",
                                         b"application/json")
    return bytes(json.dumps({'time': time.time()}), 'utf8')

site = server.Site(Simple())
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8080)
endpoint.listen(site)
reactor.run()
```

虽然 Twisted 是一个非常健壮和高效的框架，但使用它构建 HTTP 微服务时，你可能遇到下列问题：

- 必须使用从 Resource 类派生的类(该类实现了每个支持的方法)来实现微服务的每个端点。对于简单的 API 来说，它增加了很多样板代码。

- 由于是原生异步的，Twisted 代码可能很难理解和调试。
- 当你串联了很多功能，又将其逐一触发时，很容易掉进回调地狱——代码会变得混乱。
- 很难正确测试 Twisted 应用，你必须使用 Twisted 特定的单元测试模型。

Tornado基于类似的模型，但在某些领域做得更好。它有一个量级更轻的路由系统，并尽可能使代码更接近普通Python。Tornado也使用回调模型，所以调试难度较大。

但这两个框架都努力借助 Python 3 中引入的新异步功能来弥补差距。

1.6.4 asyncio 模块

当 Guido van Rossum 开始在 Python 3 中添加异步功能时，社区的一部分人认为，以同步、顺序化方式编写应用更合理一些，不必像 Tornado 或 Twisted 那样必须添加显式回调。这些人推荐使用类似 gevent 的解决方案。

但 Guido 选择了显式技术，并在 Tulip 项目(该项目受到 Twisted 的启发)进行尝试。最后，asyncio 模块从 Tulip 项目中诞生，并添加到 Python 中。

事后看来，在 Python 中实现显式的事件循环机制，而非采用 gevent 的方式是比较合理的。Python 核心开发人员编写了 asyncio，优雅地使用 async 和 await 关键字来扩展 Python 语言实现协程(coroutine)，使用普通 Python 3.5+构建的异步应用代码看起来非常优雅，而且很接近同步编程。

协程是能暂停和恢复程序执行的功能。第 12 章将详细解释如何在 Python 中实现协程以及如何使用协程。

回调语法混乱问题经常出现在 Node.js 和 Twisted(Python 2)应用中。但通过以上方式，Python 很好地避免了此类问题。

除了协程外，Python 3 还在 asyncio 包中引入一套完整的功能和帮助程序来构建异步应用，详情可参阅 https://docs.python.org/3/library/asyncio.html。

Python 现在可像 Lua 这样的表达式语言一样创建基于协程的应用，现在有一些新框架已嵌入这些功能。只有 Python 3.5+版本支持此功能。

KeepSafe 的 aiohttp(http://aiohttp.readthedocs.io)就是其中之一，只需要几行优雅的代码就能构建完全异步的微服务：

```
from aiohttp import web
import time

async def handle(request):
```

```
    return web.json_response({'time': time.time()})

if __name__ == '__main__':
    app = web.Application()
    app.router.add_get('/', handle)
    web.run_app(app)
```

在这个简短示例中，实现方式非常类似于同步应用的实现方式。使用异步的唯一提示是 async 关键字，async 关键字用来指出 handle 函数是协程的。

这就是将在异步 Python 应用的每个级别上使用的方式。这里有另一个使用 aiopg 的例子，来自 asyncio 的 PostgreSQL 库的项目文档：

```
import asyncio
import aiopg

dsn = 'dbname=aiopg user=aiopg password=passwd host=127.0.0.1'

async def go():
    pool = await aiopg.create_pool(dsn)
    async with pool.acquire() as conn:
        async with conn.cursor() as cur:
            await cur.execute("SELECT 1")
            ret = []
            async for row in cur:
                ret.append(row)
            assert ret == [(1,)]

loop = asyncio.get_event_loop()
loop.run_until_complete(go())
```

通过添加少量 async 和 await 前缀，让执行 SQL 查询和返回结果的函数看起来接近于同步函数。

但基于Python 3的异步框架和库还处于兴起阶段，如果你使用asyncio或诸如aiohttp的框架，都必须在每个需要的功能中使用特定的异步实现方式。

如果代码中需要使用非异步的库，那么需要完成一些额外的和富有挑战性的工作，以免阻塞事件循环。

如果你的微服务处理的资源数量有限，则是可管理的。但在撰写本书期间，坚持使用已经成熟的同步框架可能比使用异步框架更安全。让我们先享用目前成熟的程序包的生态系统，并期盼 asyncio 生态系统早日走向成熟！

Python 中有很多同步框架可用来构建微服务，如 Bottle、Pyramid with Cornice、Flask 等。

本书的下一个版本很可能使用异步框架。但这一版中还是使用 Flask 框架。Flask 已经存在了一段时间，非常健壮和成熟。但请记住，无论使用什么 Python Web 框架，都应能接替运行本书中的所有示例。这是因为所有构建微服务的代码都非常接近于普通 Python，而框架的主要作用是路由请求并提供一些帮助。

1.6.5 语言性能

前一节探讨了两种编写微服务的不同方式：异步和同步。无论使用哪种技术，Python 的速度直接影响着微服务性能。

当然，每个人都知道 Python 比 Java 和 Go 要慢，但执行速度并非总是最重要的。微服务通常是一层薄薄的代码，它的大部分时间都在等待来自其他服务的网络响应。Postgres 服务器需要快速返回 SQL 查询结果(因为构建响应时，其中花费的时间最多)，与此相比，对于微服务而言，内核速度就没那么重要了。

当然，让应用尽快运行是合情合理的要求。

在Python社区中，围绕语言加速的一个有争议的话题是GIL(Global Interpreter Lock)互斥会破坏性能，因为多线程应用不能使用多个进程。

GIL 的存在是有理由的，它可保护 CPython 解释器的非线程安全部分，也存在于 Ruby 等其他语言中。到目前为止，所有试图将其删除的尝试都未能加快 CPython 实现的速度。

Larry Hasting 正在研究一个名为Gilectomy(https://github.com/larryhastings/gilectomy)的无GIL CPython项目。它的最低目标是提出一个无GIL实现，能使单线程应用的运行速度达到CPython级别。但到撰写本书时为止，还是慢于CPython的。跟踪这个项目，看它能否达到“速度相同”的那一天很有趣，那时，非GIL CPython将非常有吸引力。

对于微服务，除了阻止在同一进程中使用多个内核外，GIL 也会由于互斥锁带来的系统调用开销而降低高负载时的系统性能。

然而，围绕 GIL 的所有关注是有益的，在过去几年，已经做了一些工作来减少解释器中的 GIL 争夺，这让 Python 的性能在一些领域有了极大提高。

注意，即使核心团队删除 GIL，由于 Python 是一门解释性和垃圾收集语言，也会遭受这些特性带来的性能损失。

如果你对解释器如何分解函数感兴趣，可分析一下 Python 提供的 dis 模块。下面的示例中，解释器将一个生成自增值的函数分解成不少于 29 步的指令操作序列：

```
>>> def myfunc(data):
...   for value in data:
...       yield value + 1
...
>>> import dis
>>> dis.dis(myfunc)
 2           0 SETUP_LOOP              23 (to 26)
             3 LOAD_FAST                0 (data)
             6 GET_ITER
       >>    7 FOR_ITER                15 (to 25)
            10 STORE_FAST               1 (value)

 3          13 LOAD_FAST                1 (value)
            16 LOAD_CONST               1 (1)
            19 BINARY_ADD
            20 YIELD_VALUE
            21 POP_TOP
            22 JUMP_ABSOLUTE            7
       >>   25 POP_BLOCK
       >>   26 LOAD_CONST               0 (None)
            29 RETURN_VALUE
```

用静态编译的语言编写的类似函数将大大减少生成相同结果所需的操作数。不过，还有一些方法可加快 Python 的执行速度。

一种方法是构建 C 语言扩展，或使用诸如 Cython(http://cython.org/)的语言静态扩展，将代码的一部分写入编译代码，但这会使代码更复杂。

另一种最有前景的解决方案是，仅使用 PyPy 解释器(http://pypy.org/)来运行应用。

PyPy 实现了 JIT(Just-In-Time)编译器，此编译器在运行时直接用 CPU 可使用的机器码替换 Python 片段。JIT 的策略是在执行前，实时检测何时编译以及如何编译。

虽然 PyPy 总比 CPython 滞后几个 Python 版本，但你可在生产环境中使用它，而且它的性能惊人。我们有一个 Mozilla 项目需要快速执行，PyPy 版本的程序几乎与 Go 版本的程序一样快，因此我们在那里改用 Python。

要了解 PyPy 与 CPython 的不同之处，PyPy Speed Center 网站(http://speed.pypy.org/)是可供访问的绝佳场所。

如果你的程序使用了 C 扩展，则需要针对 PyPy 重新编译，这是一个问题。如果其他开发人员维护你使用的某些扩展，这尤其麻烦。

不过，如果你用一组标准库构建微服务，则很可能可直接与 PyPy 解释器一起工作，所以这是值得一试的。

对于大多数项目而言，Python 及其生态系统的好处大大超出本节描述的性能问题，因为微服务的性能开销很少是一个问题。即使算作一个问题，微服务方法也允许你在不影响系统其余部分的情况下，重新编写“性能关键型”组件。

1.7 本章小结

本章比较了构建 Web 应用所用的单体方法和微服务方法。很明显，这两个选项并非严重对立，你并不需要在第一天就做出抉择并坚持到底。

可使用单体来启动项目，然后用微服务加以改进。随着项目逐渐成熟，部分服务逻辑应迁移到微服务中。这是你从本章学到的有用方法，但需要小心地避免落入一些常见陷阱。

另一个要点是，Python 被认为是编写 Web 应用的最佳语言之一，也是编写微服务的最佳语言之一。由于 Python 提供了大量成熟的框架和包，在其他领域也被经常使用。

本章简单介绍了几个同步或异步框架，在本书的后续章节，将使用 Flask 框架。

下一章将介绍奇妙的 Flask 框架，即使你之前不熟悉它，也很可能会喜欢上它。

最后，Python 是一个较慢的语言，在某些特定的情况下这可能是一个问题。但通过弄清楚是什么让它变慢，几个避免缓慢的解决方案足以让这个问题变得不再重要。