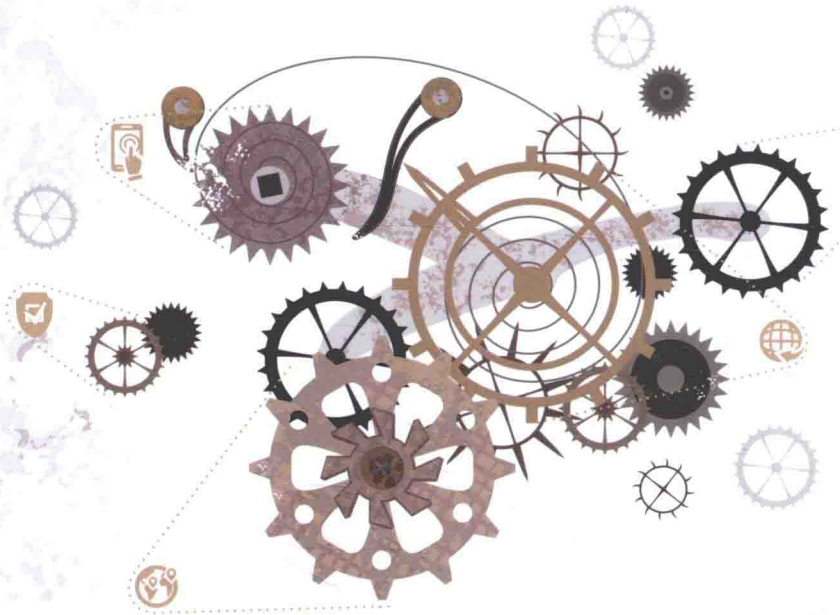
Broadview®
www.broadview.com.cn

追随Google | Facebook | Amazon | Netflix

化大而复杂为小而简单，用快速交付支撑持续创新

微服务架构与实践

王磊◎著

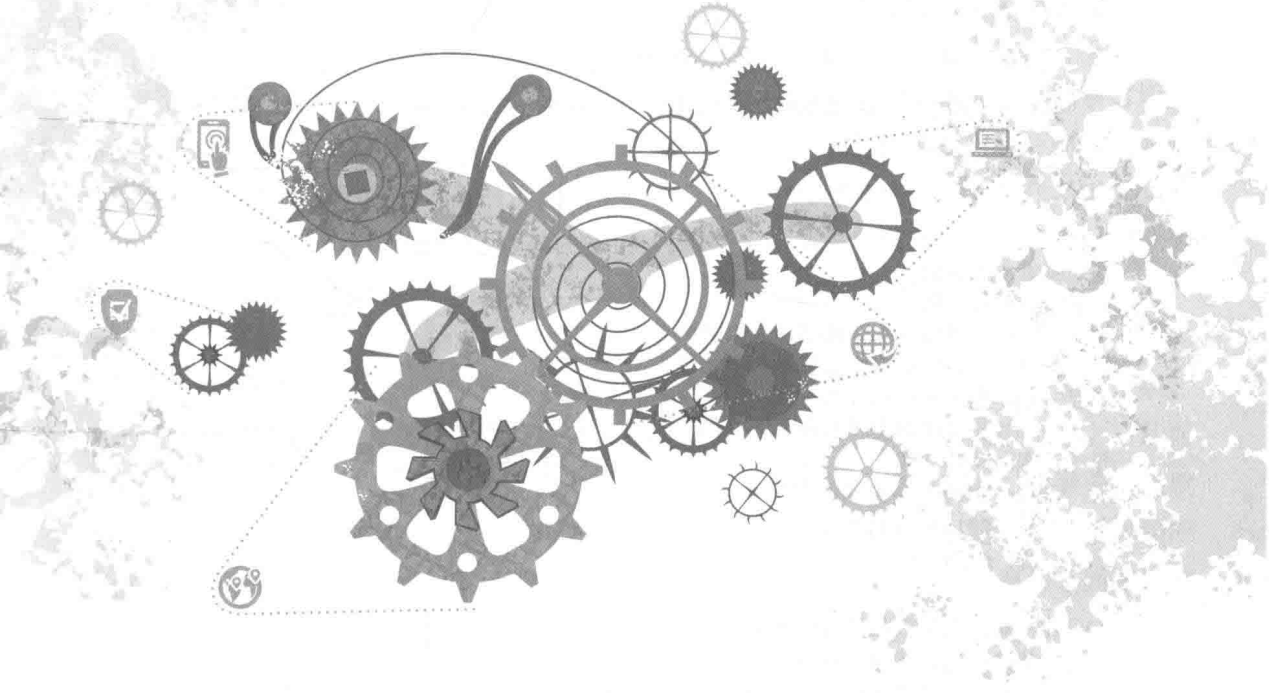


中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

微服务
架构与实践

王磊◎著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

随着 RESTful、云计算、DevOps、持续交付等概念的深入人心，微服务架构逐渐成为系统架构的一个代名词。本书首先从理论出发，介绍了微服务架构的概念、诞生背景、本质特征以及优缺点；然后基于实践，探讨了如何从零开始构建第一个微服务，包括 Hello World API、Docker 映像构建与部署、日志聚合、监控告警、持续交付流水线等；最后，在进阶部分讨论了微服务的轻量级通信、消费者驱动的契约测试，并通过一个真实的案例描述了如何使用微服务架构改造遗留系统。全书内容丰富，条理清晰，通俗易懂，是一本理论结合实践的微服务架构的实用书籍。

本书不仅适合架构师、开发人员、测试人员以及运维人员阅读，也适合正在尝试使用微服务架构解耦历史遗留系统的团队或者个人参考，希望本书能在实际工作中对读者有所帮助。

图书在版编目（CIP）数据

微服务架构与实践/王磊著.—北京：电子工业出版社，2016.1
ISBN 978-7-121-27591-3

Ⅰ.①微… Ⅱ.①王… Ⅲ.①互联网络－网络服务器 Ⅳ.①TP368.5

中国版本图书馆 CIP 数据核字（2015）第 273639 号

策划编辑：张春雨
责任编辑：刘　舫
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：14.75　　字数：312 千字
版　　次：2016 年 1 月第 1 版
印　　次：2016 年 1 月第 1 次印刷
定　　价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件到 dbqq@phei.com.cn。

服务热线：（010）88258888。

推荐语

天下武功，唯“快”不破。在当下互联网环境下，相信每一个 IT 从业者都能够深切地体会到“快”这个字对应用开发的影响。互联网产品的需求来得快，变得快，你的产品必须持续创新，不断给用户带来新的价值，否则用户会毫不犹豫弃你而去。用户期望的交付周期也极大缩短了，导致传统以月为单位的交付周期不得不被压缩到天甚至小时，这就要求互联网产品必须小步快跑，快速迭代，总之，就是要“快”。我拜访过很多互联网公司，知道做互联网要“快”，但是现实情况是他们中的大部分都面临着产品迭代速度越来越慢的问题。分析原因可以发现一个共同点，就是随着产品功能的累积，应用实现越来越复杂，代码规模越来越大，开发团队工作在一个逻辑复杂、模块耦合的单块架构应用之上，从而导致应用难于维护和更新，发布过程很长，而且随时面临发布失败的风险。微服务架构就能够很好地解决这个问题。微服务架构自 2010 年开始逐渐被大家熟知，通过对传统单块应用进行服务化切分，把一个大而复杂的问题化解为多个小而简单的问题，服务之间通过契约来约定依赖，做到服务独立发布和演进。今天，微服务架构已经被广泛运用在像 Google、Facebook 这样的大型互联网公司，为他们的快速交付和持续创新提供软件架构支撑。本书

中有大量微服务架构实战经验的总结，不仅仅有应用架构设计的内容，还涵盖了微服务大背景下应用测试、发布、日志、监控等方面，让读者可以全面应对微服务架构需求。

有人把微服务比作一把双刃剑，一方面它把单个问题域的复杂性降低了，服务可以独立更新、快速交付；但另一方面，面对一个由不同技术栈支撑的整体系统，运维和交付的难度增大了。在本书中，作者通过把微服务架构和当下热门的 Docker 容器技术、AWS 自动化部署相结合，向读者介绍了具有前瞻性的微服务自动化运维最佳实践，同时详细阐述了微服务化应用的持续交付流程和设计要领，不乏独到见解和技术细节，相信企业 CIO、软件设计师、架构师们读完这本书一定会受益匪浅。

郭峰

DaoCloud 联合创始人

微服务的出现，为运维又打开了一扇窗。微服务将整个业务系统拆分为相对独立的业务模块，并强调各个微服务都可以独立测试、独立部署、独立运行；微服务之间是一种真正的低耦合，就像汽车的各个零部件，哪个坏了，拆掉换个新的就能组装上；微服务面向产品而不是项目，这样，开发、测试、运维（系统、DBA 等）可形成更稳定的“小”团队，而不是项目周期一到，各个职能解散，各回各家；微服务配以 Docker，更可谓珠联璧合。这些都对运维提出了新的机遇和挑战，熟悉 DevOps、懂 Docker、沟通能力强的综合型运维人员，市场需求和价值更加突显。

纵览全书，说理清楚，用清晰明了的文字，帮助大家理清了很多似是而非的概念；图文并茂，图片既清晰又贴切，语言朴实、平易近人，没有从国外翻译过来的书籍那种生硬、别扭的感觉；理论结合实际，更多融合了作者实施微服务的一线经验。是一本非常用心、又可以实际落地的好书。

萧田国

开放运维联盟联合主席，高效运维社区创始人

当我开始阅读本书时，我并不知道微服务是否适合我。然而在读完前几章后，我发现这正是我在寻求的。微服务其实已经无处不在，在全球各处的技术会议上每个人都在讨论微服务。但它不是新东西，亚马逊和 Netflix 等大公司已经使用微服务架构建造分布式系统多年。在本书中，作者跟随领先者在本领域深入浅出地解释了一些重要的概念，并给出许多有价值的实践指导。最后，本书蒸馏出很多它处寻不到的信息，提供给大家一个通用的、坚固的开始。

龚勇

辉门（中国）有限公司，亚太区信息技术总监

微服务架构作为 SOA 在众多互联网公司中的成功新实践，是广大企业在互联网化进程中必须理解的概念。本书不仅讲述了微服务的基础理论，而且通过实例，深入浅出地涵盖了微服务构建过程中持续集成、构建、部署、持续交付以及日志聚合和运维的过程，体现了作者深厚的理论功底与扎实的实践经验，推荐阅读。

徐唤春

上海商派软件有限公司技术副总裁

微服务的概念初看简单清晰、容易理解，但在企业中的实际实施其实是一件很困难的事情。尤其很多计划实施微服务的公司在服务划分、DevOps 和相应的组织结构变化方面毫无经验，付出了实施的代价，却很难真正享受到微服务带来的好处。这本书总结了作者两年多在真实大型软件系统上实施微服务的经验和心得，具体指导了微服务实施在技术方面的实践，非常值得参考。

杨云

ThoughtWorks 首席咨询师，前支付宝资深架构师

随着应用系统的不断发展演进，单体应用变得越来越大，越来越复杂，导致扩展性差，资源优化难，维护成本高等问题。为了应对这一挑战，一种更加灵活、轻便、松耦合的设计架构——微服务架构，正受到越来越多应用系统开发者的青睐，它的敏捷开发、灵活部署、易扩展等特性，使它成为解决复杂应用的一把利器。微服务架构在具体实践中是怎样实施的？它在实施过程中存在怎样的困难和挑战?作者在本书中通过理论结合实践的方式，深入浅出地阐述了微服务的本质以及如何有效地、持续地交付微服务，并给出了许多有价值的实践指导，全书内容丰富，理论结合实际，推荐阅读。

薛正华博士

中国计算机学会高级会员，大数据专委会委员

前言

一直以来，系统的架构设计就是IT领域经久不衰的话题之一，是每个系统构建过程中极其关键的一部分，它决定了系统是否能够被正确、有效地构建。系统架构设计描述了在应用系统内部，如何根据业务、技术、组织、灵活性、可扩展性以及可维护性等多种因素，将应用系统划分成不同的部分，并使这些部分之间相互分工、相互协作，从而为用户提供某种特定的价值。多年来，我们一直在技术的浪潮中乘风破浪，扬帆奋进，寻找更优秀的系统架构设计方式来构建系统。

由来

随着RESTful、云计算、DevOps、持续交付等概念的深入人心，微服务架构逐渐成为系统架构的一个代名词。那么微服务是否是业界期待已久的企业架构解决方案呢？在微服务架构的实施过程中存在着怎样的困难和挑战呢？

在过去两年多的时间里，笔者一直在探索和实践，并助力国外某房地产互联网门户，将其复杂的业务支撑系统逐渐演进为基于微服务架构的系统。

这期间也经历了从微服务的理论认识，到小范围实践、迭代，再到多个基于微服务构建的项目已经成功上线的过程。在感受微服务为开发实践、测试策略、部署、运维等带来改变的同时，也切身体会到使用微服务架构，对系统灵活性、可伸缩性方面的提升，以及对团队应对变化能力的提升。

结构

鉴于此，本书从笔者实践的角度出发，首先阐述了单块架构存在的弊端以及微服务的理论基础。接着通过实践部分，让读者能够体验从零开始搭建第一个微服务的过程，包括代码静态检查、AWS 基础设施构建、Docker 映像构建及部署、持续交付流水线、服务的日志聚合以及监控和告警。随后，探讨了笔者在微服务的实践过程中所积累的经验，包括基于 HAL 的通信机制、消费者驱动的测试，并通过一个真实的案例，帮助读者更好地理解微服务架构所带来的灵活性、易扩展性和独立性。

全书分成 3 部分，共 14 章。

第 1 部分为基础部分。包括第 1 章和第 2 章，概述了三层应用架构以及微服务架构。

第 2 部分为实践部分。包括第 3 章至第 10 章，通过一个具体的实例，从头到尾介绍了一个服务从需求到实现，再到构建、部署以及运维的整个过程。

第 3 部分为进阶部分。包括第 11 章至第 14 章，讨论了微服务的持续交付、测试策略、通信机制，并描述了一个使用微服务改造遗留系统的真实案例。

部分和部分之间几乎是相互独立的，没有必然的前后依赖关系，因此，读者可以从任何一个感兴趣的部分开始阅读。但是，每部分中的各章节之间的内容是相互关联的，建议按照章节的顺序阅读。

结合作者本人的工作经验和使用习惯，书中的大部分案例代码均采用 Ruby 编写，并且运行在 Mac OS 环境上。因此，读者最好对 Ruby 语言有一定了解，并且熟悉 Mac OS 或者 Linux 环境下的基本操作，以便能够更加顺利地阅读本书。

感悟

2014 年 10 月，我在 InfoQ 上发表了一篇题为《使用微服务架构改造遗留系统》的文章，

收到了很多朋友的反馈与建议，并有一些朋友和我积极探讨微服务架构的实践与心得。2015年初，基于在项目中积累的经验，我开始在博客上连载《解析微服务架构》系列的文章。机缘巧合，认识了电子工业出版社计算机出版分社的张春雨编辑，并在他的建议下，开始构思如何将自己在微服务实践中积累的经验以书的方式展现出来。但由于工作项目进度一直很紧，直到2015年4月，参加完北京QCon会议后，才开始真正动笔。

当时，自己曾信心满满地认为，应该能够比较顺利地完成这本书，因为大部分内容都比较熟悉，而且平时工作中也有笔记和积累。但当真正动笔之后才发现，理解领悟和用文字表述清楚是截然不同的两件事。有时候看似很容易的知识点，用文字解释清楚却并非易事。有时候工作中天天遵循的实践，总结清楚却需要花些工夫。当然，整个写作的过程，也是自己将微服务的相关知识点以及积累的经验从头到尾梳理的过程。通过编写本书，使得自己对这些知识的认识和理解更加深入和全面，受益匪浅。每次重新回过头来审阅书稿时，总会觉得某些知识点讲述得不够透彻，需要进行补充，或者应该用更好的方式将其展示出来。

微服务作为当下热门的话题之一，其涉及的部分已经不仅仅局限在技术层面，而是关注整个产品或者组织的价值。因此，它不仅涉及技术选择、服务划分、服务注册、服务安全、服务测试、服务运维，也包括微服务下团队组织架构的变化，全功能团队的构建，可靠的持续交付流水线，DevOps文化等。作为一本微服务架构的书，很难做到面面俱到。另外，由于时间仓促以及作者自身水平有限，书中难免有疏漏之处，在此敬请广大读者批评指正。在阅读本书的过程中，如有任何问题，可通过微信号：5109343或邮箱：wldandan@gmail.com与我联系。

最后，希望读者能享受微服务架构的实践之路。

王磊

2015年10月7日于西安

致谢

首先，要感谢我的家人，特别是我的妻子。在我占用大量周末、休假的时间进行写作的时候，她给予了极大的宽容、支持和理解，并对我悉心照顾且承担了全部的家务，让我能够全身心地投入到写作之中，而无须操心家庭琐事。没有她的支持和鼓励，这本书是无法完成的。

同时，要感谢 ThoughtWorks 提供了优秀的环境和平台，使我的技能能够得以施展，并且身处一群技术牛人中间，也得到了很多学习和成长的机会。

另外，感谢我的同事赵国庆和陈熙，他们为本书“Pact 实现契约测试”小节提供了示例代码。也要感谢我所在的团队对我一直以来的鼓励与支持，并在日常的工作中给予了我很多帮助。同时，感谢张凯峰、吕健、崔福罡、周星、邱俊涛（排名不分先后）等同事的帮助，在我写书的过程中，他们提出了很多宝贵的建议。谢谢各位！

最后，还要感谢电子工业出版社计算机出版分社的张春雨编辑和刘舫编辑，在我写作期间给予了很多指导和帮助。根据他们的建议，我对本书的内容做了很多修正，使内容更加充实，也更加易懂，本书能够出版，离不开他们的敬业精神和一丝不苟的工作态度。

目录

第 1 部分　基础篇

第 1 部分

基础篇

一直以来，系统的架构设计是 IT 领域经久不衰的话题之一，是每个系统构建过程中极其关键的一部分，它决定了系统是否能够被正确、有效地构建。

那什么是系统架构设计？

系统架构设计描述了在应用系统的内部，如何根据业务、技术、组织、灵活性、可扩展性以及可维护性等多种因素，将应用系统划分成不同的部分，并使这些部分彼此之间相互分工、相互协作，从而为用户提供某种特定价值的方式。

多年来，我们一直在技术的浪潮中乘风破浪，扬帆奋进，寻找更优秀的系统架构设计方式来构建系统。

随着面向对象分析、设计模式、企业架构模式等方法论的深入人心，从功能实现、代码组织的角度考虑，系统中不同职责的部分逐渐被划分到了如下三个层次：

- 表示层，聚焦数据显示和用户交互。
- 业务逻辑层，聚焦业务逻辑处理。
- 数据访问层，聚焦数据的存储与访问。

每一层负责的部分更趋向于具体化、细致化，这就是最初的软件三层架构。三层架构的出现，不仅解决了系统间调用复杂、职责不清的问题，更有效地降低了层与层之间的依赖关系，成为软件架构的经典模式之一。虽然三层架构将系统在逻辑上分成了三层，但它并不是物理上的分层。也就是说，对不同层的代码而言，经历编译、打包、部署后，所有的代码最终还是运行在同一个进程中。

对于这种功能集中、代码中心化、一个发布包、部署后运行在同一进程的应用程序，我们通常称之为单块架构应用。典型的单块架构应用，莫过于传统的 J2EE 项目所构建的产品或者项目，它们存在的形态一般是 WAR 包或者 EAR 包。当部署这类应用时，通常是将整个包作为一个整体，部署在某个 Web 容器，如 Tomcat 或者 Jetty 中。当这类应用运行起来后，所有的功能也都运行在同一个进程中。

随着业务的不断扩大，需求功能的持续增加，单块架构已经很难满足业务快速变化的需要。一方面，代码的可维护性、扩展性、灵活性在降低；另一方面，系统的修改成本、构建以及维护成本在显著增加。因此，单块架构应用的改造与重构势在必行。

第 1 章

单块架构及其面临的挑战

随着面向对象分析、面向对象设计、面向对象原则、设计模式、企业架构模式等理念以及方法论的不断发展，从为用户提供功能以及有效组织软件结构的角度考虑，系统中不同职责的部分逐渐被定义在了不同的层次，每一层负责的部分更趋向于具体化、细致化，于是软件的三层架构逐渐出现了。三层架构通常包括表示层、业务逻辑层以及数据访问层。

三层架构的出现，解决了系统间调用复杂、职责不清的问题，也有效降低了层与层之间的依赖关系，成为软件架构的经典模式之一。

虽然三层架构将系统在逻辑上分成了三层，但它并不是物理上的分层。也就是说，对不同层的代码而言，经历编译、打包、部署后，所有的代码最终还是运行在同一个进程中。

对于这种功能集中、代码中心化、一个发布包、部署后运行在同一进程的应用程序，我们通常称之为单块架构应用。

随着业务的不断扩大，需求功能的持续增加，单块架构已经很难满足业务快速变化的需要。一方面，代码的可维护性、扩展性、灵活性在降低；另一方面，系统的修改成本、构建以及维护成本在显著增加。因此，单块架构应用的改造与重构势在必行。

1.1 三层应用架构

1.1.1 三层应用架构的发展

现实生活中，“层”这个字的含义，大家一点都不陌生。我们经常说楼房高多少层，蛋糕有几层等。通常来说，层有好几种定义，但其中最耳熟能详的，莫过于“层”能帮助我们划分出构成某整体事物的、上下相互支撑的不同部分。譬如，我们喜欢吃的蛋糕，一般是由三层组成：第一层的蛋糕体、第二层的奶油和第三层的水果。从顶部至底部，每一层依赖于下一层，从底部到顶部，每一层又支撑着上一层。在软件架构模式领域，经过多年的发展，也有了层的概念：

- 层能够被单独构造。
- 每层具有区别于其他层的显著特点。
- 层与层之间能够互相连接，互相支撑，互相作用，相互协作，从而构成一个整体。
- 层的内部可以被替换成其他可工作的部分，但对整体的影响不大。

以 Web 应用程序为例，在 Web 应用程序开发的早期，由于受到面向过程的思维及设计方式的影响，所有的逻辑代码并没有明显的区分，因此代码之间的调用相互交错，错综复杂。譬如，我们早期使用的 ASP、JSP 以及 PHP，都是将所有的页面逻辑、业务逻辑以及数据库访问逻辑放在一起，这是我们通常提到的一层架构，如图 1-1 所示。

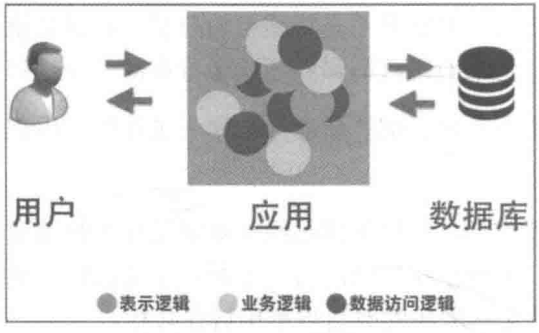


图 1-1 一层架构

随着 Java、.NET 等高级语言的快速发展，这些语言为开发者提供了越来越方便的数据访问机制，如 Java 语言的 JDBC、IO 流，或者.NET 的 ADO.NET 等。这时候，数据访问部分的代码逐渐有了清晰的结构，但表示逻辑和业务逻辑依然交织在一起，我们称这个阶段为二层架构阶段，如图 1-2 所示。

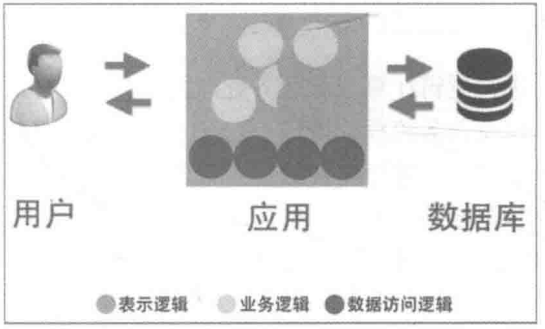


图 1-2 二层架构

随着面向对象分析、面向对象设计、面向对象原则、设计模式、企业架构模式等理念以及方法论的不断发展，从为用户提供功能以及有效组织软件结构的角度考虑，Web 应用中不同职责的部分逐渐被定义在了不同的层次，每一层负责的部分更趋向于具体化、细致化，于是软件的三层架构逐渐出现了，如图 1-3 所示。

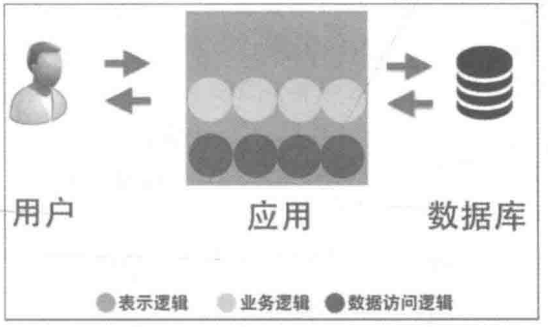


图 1-3 三层架构

1.1.2 什么是三层架构

三层架构通常包括表示层、业务逻辑层以及数据访问层。

- 表示层

表示层部分通常指当用户使用应用程序时，看见的、听见的、输入的或者交互的部分。譬如，有可能是信息的显示、音乐的播放、可以输入的文本框、单选按钮，以及可单击的按钮等。通过这些元素，用户同软件进行交互并获取期望的结果。目前的用户接口大部分情况下为 Web 方式，当然也可以是桌面软件的形式，例如.NET 的 WINFORM 或者 Java 的 SWING。

- 业务逻辑层

业务逻辑部分是根据用户输入的信息，进行逻辑计算或者业务处理的部分。业务逻辑层主要聚焦应用程序对业务问题的逻辑处理，以及业务流程的操作，它是大部分软件系统区别于其他系统的核心。譬如，当用户单击一个按钮后，它可能会触发业务逻辑部分的代码进行运算，生成用户期望的结果。举例来说，在一个电子商务平台中，作为用户，当我们下单购买某商品后，应用程序的业务逻辑层会对订单进行处理，如计算折扣、进行配送等。

- 数据访问层

在用户同应用程序交互的过程中，会产生数据。这类数据需要通过某种机制被有效地保存，并在将来能够被重复使用，或者提供给其他系统。这种机制或者方法就是数据访问层最关注的部分。也就是说，它关注的是应用程序是如何有效地将数据存储到数据库、文件系统或者其他存储介质中。有一点要注意的是，它关注的是对原始数据的操作（数据库或者文件系统等存取数据的形式以及接口），而非原始数据的存储介质本身（数据库或者文件系统本身）。

1.1.3 三层架构的优势

三层架构的出现，一方面是为解决应用程序中代码间调用复杂、代码职责不清的问题。其通过在各层间定义接口，并将接口与实现分离，可以很容易地用不同的实现来替换原有层次的实现，从而有效降低层与层之间的依赖。这种方式不仅有利于帮助团队理解整个应用架构，降低后期维护成本，同时也有利于制定整个应用程序架构的标准。

另一方面，三层架构的出现从某种程度上解决了企业内部如何有效根据技能调配人员，提高生产效率的问题。在大环境下，有效地分层能使不同职责的人员各司其职，更聚焦于个人专业技能的发展和培养。

三层架构的出现不仅标准化了复杂系统的逻辑划分，更帮助企业解决了如何有效形成技术人员组织结构的问题，因此在很长一段时间里，它一直是软件架构的经典模式之一。

1.2 单块架构

1.2.1 什么是单块架构

虽然软件的三层架构帮助我们将应用在逻辑上分成了三层，但它并不是物理上的分层。

这也就意味着，即便我们将应用架构分成了所谓的三层，经过开发团队对不同层的代码实现，经历编译（非静态语言则忽略编译阶段）、打包、部署后，不考虑负载均衡以及水平扩展的情况，最终还是运行在同一台机器的同一个进程中。对于这种功能集中、代码和数据中心化、一个发布包、部署后运行在同一进程的应用程序，我们通常称之为单块架构应用。

典型的单块架构应用，莫过于传统的 J2EE 项目所构建的产品或者项目，它们存在的形态一般是 WAR 包或者 EAR 包。当部署这类应用时，通常是将整个一块作为一个整体，部署在同一个 Web 容器，如 Tomcat 或者 Jetty 中。当这类应用运行起来后，所有的功能也都运行在同一个进程中。

类似的，基于 Ruby On Rails 的单块架构应用，一般在逻辑上分为控制器层、模型层以及视图层，同时代码存放在遵循一定层级结构的目录中。当部署这类应用的时候，通常是使用 SSH 或者其他一些工具，如 Capistrano，将整个目录部署在 Passenger 或者其他 Web 容器中。当这类应用运行起来后，所有的功能也都运行在同一个进程中，如图 1-4 所示。

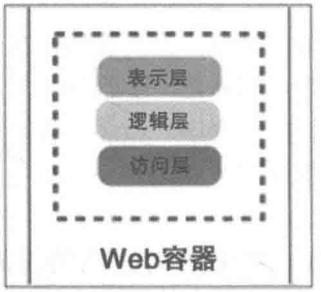


图 1-4　Web 容器中的三层架构

因此，对于单块架构应用的定义，其实是在分层软件架构设计的系统基础之上，从部署模式、运行模式角度去考虑的一种定义方式。

1.2.2　单块架构的优势

单块架构有如下一些优势。

- 易于开发

对单块架构的应用程序而言，开发方式相对简单。首先从概念上，现有的大部分工具、应用服务器、框架都是这类单块架构应用程序，容易理解而且为人所熟知。如果从实践角度出发，现有的集成开发工具比较适合单块架构的应用程序，像 NetBeans、Eclipse、IDEA 等，它们都能够有效加载并配置整个应用程序的依赖，方便开发人员开发、运行、调试等。

- 易于测试

单块架构应用程序也非常容易被测试，因为所有的功能都运行在一个进程中，启动集成开发环境或者将发布包部署到某一环境，一旦启动该进程，就可以立即开始系统测试或

者功能测试。

- 易于部署

对单块架构的应用程序而言，部署也比较容易。实际上，由于所有的功能最终都会打成一个包，因此只需复制该软件包到服务器相应的位置即可。当然，部署的方式可以有很多种，最简单的可以使用 SCP 远程复制到指定的目录下，当然也可以使用某些自动化的工具来完成。

- 易于水平伸缩

对单块架构的应用程序而言，水平伸缩也比较容易。实际上，由于所有的功能最终都会打成一个包，且只能运行在一个进程中，因此单块架构的水平伸缩，更确切地理解其实是克隆，即新建一个服务器节点，配置好该节点的运行环境，复制软件包到相应的位置，运行该应用程序。当然，必须要确保负载均衡器能采取某种分发策略，有效地将请求分发到新创建的节点。

1.2.3 单块架构面临的挑战

随着最近几年互联网行业的迅猛发展，随着公司或者组织业务的不断扩张，需求的不断增加以及用户量的不断增加，单块架构的优势已逐渐无法适应互联网时代的快速变化，面临着越来越多的挑战。譬如，一方面，随着业务的扩大，如何为用户提供可靠的服务，如何有效处理用户增多后导致并发请求数增多和响应慢的问题，以及如何有效解决用户增多后带来的大数据量的问题等。另外一方面，随着公司或者组织业务的不断扩张，需求不断增加，越来越多的人加入开发团队，代码库也在急剧膨胀。在这种情况下，单块架构的可维护性、灵活性在降低，而测试成本、构建成本以及维护成本却在显著增加。

- 维护成本增加

随着应用程序的功能越来越多，团队越来越大，相应的沟通成本、管理成本、人员协调成本必然会显著增加。譬如，对于使用 Java 编写的中型应用而言，当代码量为几万行时，可能只需要几人左右的团队维护。当代码量上升到几十万行级别时，可能需要几十人甚至是上百人的团队。

随着应用程序功能的增多，当出现缺陷时，有可能引起缺陷的原因组合就会比较多，这也会导致分析缺陷、定位缺陷、修复缺陷的成本相应增高，也就意味着缺陷的平均修复

周期可能会花费更长时间。

另外，随着代码量的增加，在开发人员对全局功能缺乏深度理解的情况下，修复一个缺陷，还有可能引入其他缺陷。在自动化测试机制不完善的情况下，很可能导致该过程陷入“修复越多，缺陷越多”的恶性循环，最终导致维护成本高居不下，如图 1-5 所示。

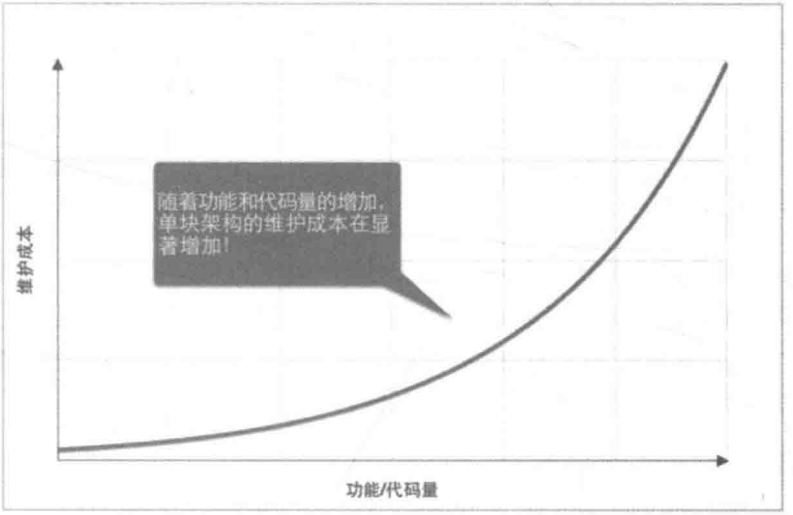


图 1-5　单块架构的维护成本

- **持续交付周期长**

随着应用程序的功能越来越多，代码越来越复杂，构建和部署时间也会相应增加。在现有部署流水线稳定工作的情况下，对单块架构应用程序做任何细微的修改以及代码提交，都会触发部署流水线，令其对整个应用程序进行代码编译、运行单元测试、代码检查、构建并生成部署包、验证功能等，这也就意味着流水线的反馈周期变长，单位时间内构建的效率变低了。

另一方面，团队人员的增多，部署流水线运行的时间增加，开发人员能够提交代码的时间窗口就相应减少（因为在流水线运行的过程中是禁止提交代码的），可能出现长时间等待代码提交却无法提交的情况，极大破坏了团队的灵活性并降低了团队的工作效率。几年前，我曾经工作在一个 50 万行代码的单块架构应用上，整个应用由一个 50 人左右的分布式团队负责。通常情况下，从开发人员提交代码到运行单元测试、构建发布包、运行功能测试、标记为可发布状态，大概需要 40 分钟，时间稍微有点长，但还能忍受。关键的问题是开发人员通常都是集中在下午 3 点左右完成特定功能后提交代码，结果就导致 3 点至 5 点那个时间

段，成为代码提交的瓶颈，极大影响了该应用持续集成和构建的效率。

- 新人培养周期长

随着应用程序的功能越来越多，代码变得越来越复杂，对于新加入团队的成员而言，了解行业背景、熟悉应用程序业务、配置本地开发环境这些看似简单的任务，将会花费越来越长的时间。

曾经有个朋友，在加入一家世界 500 强的知名 IT 公司后，被安排到了一个百万行级代码的产品组里。他花了将近 1 个月的时间来熟悉产品文档、配置开发环境，才在本地成功地运行了这个应用。在他从事这份新工作的头一个月里，我好几次问及他对新工作的印象，得到的答案都是一样“看文档，装环境”。

对个人而言，花一个月时间来配置本地开发环境，其中滋味大家可想而知，我估计人世间最痛苦的事情也莫过于此。而对公司或者部门而言，本期望员工花几天就能配置好的环境，却花了一个月才完成，这更是极大的浪费。更有甚者，在第一次配置完开发环境后，好几年都不愿意再升级或者重装系统，真是“一朝被蛇咬，十年怕井绳”。

- 技术选型成本高

传统的单块架构系统倾向于采用统一的技术平台或方案来解决所有问题。通常，技术栈的决策是在团队开发之前经过架构师、技术经理慎重评估后选定的，每个团队成员都必须使用相同的开发语言、持久化存储及消息系统，而且要使用类似的工具。随着应用程序的复杂性逐渐增加以及功能越来越多，如果团队希望尝试引入新的框架、技术，或者对现有技术栈升级，通常会面临不小的风险。

另一方面，互联网行业不仅市场变化快，而且技术变化也快。譬如，短短几年时间，仅前端的 JavaScript 框架，就出现了好几十个，从早期的 Backbone、Ember 到近来的 AngularJS、Ractive 等。类似的，后端的框架、工具等也是层出不穷，有兴趣的朋友可以参考 ThoughtWorks 的技术雷达[1]。

因此，对单块架构的应用而言，初始的技术选型严重限制了其将来采用不同语言或框架的能力。如果想尝试新的编程语言或者框架，没有完备的功能测试集，很难平滑完成替换，而且系统规模越大，风险越高。

---

[1] ThoughtWorks 技术雷达是 ThoughtWorks 定期对业界技术、工具、语言等发展趋势的分析以及预测报告，包括技术、工具、平台、语言和架构，更多细节请访问 http://www. ThoughtWorks.com/radar/techniques。

- 可扩展性差

无论是垂直扩展还是水平扩展，单块架构都存在不同程度的扩展性问题。

  - 垂直扩展

    如果应用程序的所有功能代码都运行在同一个服务器上，将会导致应用程序的扩展非常困难。如果扩展要求紧急，那么垂直扩展（Vertical Scaling 或 Scale-up）可能是最容易的（如果钱不是问题的话）。

    在大多数情况下，如果舍得砸钱上 IBM 的服务器、Oracle 的数据库或者来自 EMC 的存储设备，不用改变一行代码，整个世界都会变好的。不幸的是，伴随着业务的增加，数据的增加，垂直扩展可能会变得一次比一次吃力，成本越来越高。这也是为什么很多公司开始尝试使用开源，并放弃昂贵的 IOE 产品的原因。有兴趣的朋友可以看看讲述淘宝技术架构演进的《淘宝技术这十年》一书，其中讲了很多类似的故事。

  - 水平扩展

    与此相对，水平扩展（Horizontal Scaling 或 Scale-out）通常的做法是建立一个集群，通过在集群中不断添加新节点，然后借助前端的负载均衡器，将用户的请求按照某种算法，譬如轮转法、散列法或者最小连接法等合理地将请求分配到不同的节点上。

    但是，对于单块架构而言，由于所有程序代码都运行在服务器上的同一个进程中，会导致应用程序的水平扩展成本非常高。譬如，如果应用程序某部分的功能是内存密集型的，需要缓存大量数据，而另外一部分功能是 CPU 密集型的，需要进行大量的运算，那么每次实施水平扩展，运行该应用的服务器都必须有足够的内存和强劲的 CPU 来满足需求。

    因此，鉴于每个服务器都要提供该应用系统所需要的各种资源，基础设施的整体花费可能会非常高。另外，如果某些节点内部保持了状态，如用户登录后的会话信息等，更会大大增加水平扩展的难度。

- 构建全功能团队难

最后，非常微妙的是，随着应用程序的功能越来越多，代码变得越来越复杂，其应用程序的复杂结构也会逐渐映射到研发团队的结构上。康威定律指出：一个组织的设计成果，

其结构往往对应于这个组织中的沟通结构。

单块架构的开发模式在分工时往往以技能为单位，比如用户体验团队、服务端团队和数据库团队等，这样的分工可能会导致任何功能上的改变都需要跨团队沟通和协调，如图 1-6 所示。

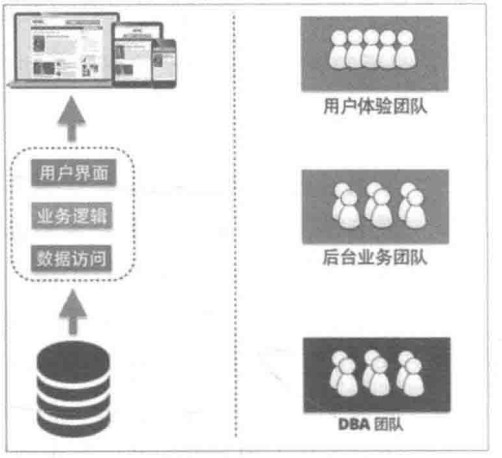


图 1-6　按技能划分团队

譬如，用户体验工程师（UX）更专注负责用户接口部分，业务层开发者则负责建立服务器后端的业务逻辑，数据库工程师和 DBA 更关注数据访问组件和数据库。鉴于这些问题，随着时间的推移，不仅代码越来越难以管理，其对团队结构的影响也越来越明显。

1.3　小结

综上所述，随着业务的不断扩大，需求功能的持续增加，单块架构已经很难满足业务快速变化的需要。一方面，代码的可维护性、扩展性、灵活性在降低；而另一方面，系统的测试成本、构建成本以及维护成本在显著增加。因此，随着项目或者产品规模的不断扩大，单块架构应用的改造与重构势在必行。

互联网时代的产品通常有几个特点：创新成本低、需求变化快，用户群体庞大，它和几年前我们熟悉的单块架构应用有着本质的不同。随着市场变化加快、用户需求变化加快、用户访问量增加，单块架构应用的维护成本、人员培养成本、缺陷修复成本、技术架构演进的成本以及系统扩展成本等都在相应增加，因此单块架构曾经的优势已逐渐无法适应互联网时代的快速变化，面临着越来越多的挑战。

第 2 章

微服务架构综述

微服务架构模式（Microservice Architect Pattern）是近两年在软件架构模式领域出现的一个新名词。虽然其诞生时间不长，但其在各种演讲、文章、书籍中出现的频率已经让很多人意识到它对软件领域所带来的影响。

到底什么是微服务架构呢？当我们谈论微服务时，它代表着一种什么样的含义？同传统的面向服务的架构 SOA 相比，它有什么异同点？

本章，我们将揭开微服务的神秘面纱。

本章的主要内容包括：

- 什么是微服务架构
- 微服务架构与 SOA
- 微服务的本质
- 微服务不是银弹

2.1 什么是微服务架构

其实，很难对微服务下一个准确的定义。就像 NoSQL，我们谈论了好几年的 NoSQL，知道 NoSQL 的大致含义，也可以根据不同的应用场景选择不同的 NoSQL 数据库，但是我们还是很难对它下一个准确的定义。类似的，关于什么是“函数式编程”，也或多或少存在同样的窘境。我们可以轻松地选择不同的函数式编程语言，可以轻松地写出函数式编程风

格的代码，但很难对什么是函数式编程下一个准确的定义。

实际上，从业界的讨论来看，微服务本身并没有一个严格的定义。不过，ThoughtWorks的首席科学家——马丁·福勒（Martin Fowler）先生，对微服务的这段描述，似乎更加通俗易懂：

> 微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务与服务间采用轻量级的通信机制互相沟通（通常是基于 HTTP 的 RESTful API）。每个服务都围绕着具体业务进行构建，并且能够被独立地部署到生产环境、类生产环境等。另外，应尽量避免统一的、集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言、工具对其进行构建。
>
> ——摘自马丁·福勒先生的博客
> （http://martinfowler.com/articles/microservices.html）

2.1.1 多微才够微

微服务架构通过对特定业务领域的分析与建模，将复杂的应用分解成小而专一、耦合度低并且高度自治的一组服务，每个服务都是很小的应用。那么，微服务中提到的“微”，到底是个什么样的“微”呢？

实际上，关于多微的服务才合适，是一个非常有趣的话题。有人觉得使用代码行数来作为“微”的衡量标准比较合适，而有些人认为，既然是微服务，就应该简单，应该在很短的时间内，譬如 2 周内，能够容易地重写该服务，这样才符合微的定义。

- **代码行数**

我们知道，不同的语言有不同的特点。静态类型语言的主要优点在于其结构规范，存在编译期的语法检查、便于调试、类型安全性高，通常其继承关系简洁明了，IDE 对其支持也更加友好；但其缺点是为此需要写更多的类型相关代码。因此如果要实现同样的功能，代码量相对稍多，这类语言的典型代表有 Java、C++等。

动态语言，其灵活性较高，运行时可以改变内存结构，无类型检查，无须写较多的类型相关的代码；但缺点是不方便调试，无编译期检查，因此对个体的能力要求较高。尤其是项目复杂度高或者代码量较大的项目，如果没有足够高的测试覆盖率，维护起来更是举步维艰。典型的代表如 JavaScript、Ruby 或者 Python 等。

譬如，对于经典的康威生命游戏而言，游戏开始时，细胞随机地被指定为存活或者死亡状态（黑色表示存活，白色表示死亡），每个细胞都会不断演进，并且在演进的过程中，每个细胞下一代的状态由该细胞当前周围 8 个细胞的状态所决定，其具体规则如图 2-1 所示。

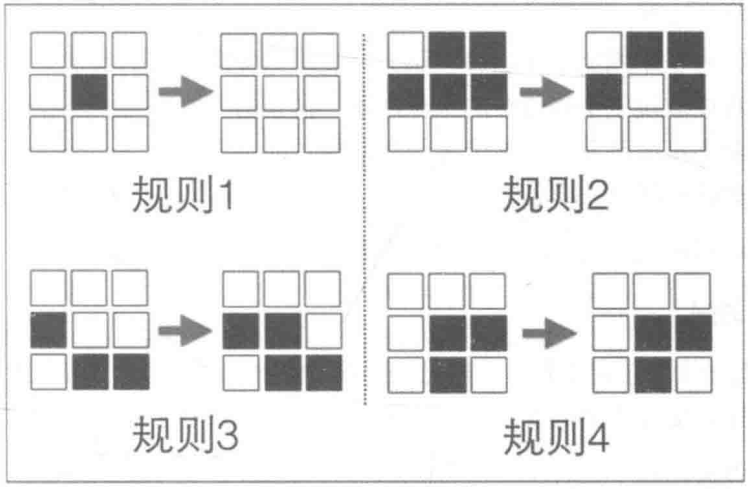


图 2-1　康威生命游戏的规则

- 规则 1，如果一个细胞周围有少于 2 个存活细胞，则该细胞无论存活或者死亡，下一代将死亡。
- 规则 2，如果一个细胞周围有多于 3 个存活细胞，则该细胞无论存活或者死亡，下一代将死亡。
- 规则 3，如果一个细胞周围有 3 个存活细胞，则该细胞下一代将存活。
- 规则 4，如果一个细胞周围且仅有 2 个存活细胞，则该细胞下一代状态保持不变。

感兴趣的读者，可以尝试使用自己熟悉的开发语言来解决这个问题。通常，由于语言特性的差异，静态语言实现的代码量会略大于动态语言。譬如用 Java 实现的版本，代码量会略多于用 Ruby 实现的版本。

有一种数学语言，APL（A Programing Language），如果我们使用它来实现康威生命游戏的题目，只需如下一行代码就可以解决问题。

```
life←{↑1 ω∨.∧3 4=+/,¯1 0 1∘.θ¯1 0 1∘.⌽⊂ω}
```

因此，对于实现同样的功能，选择不同的语言，代码的行数会千差万别。因此，代码行数这种量化的数字显然无法成为衡量微服务是否够“微”的决定因素。

- **重写时间**

有些人认为，既然是微服务，就应该简单，应该在很短的时间内（譬如 2 周），能容易地重写当前服务，这样才符合“微”的概念。

实际上，同样的周期，对于不同的个体而言，结果可能不尽相同。我们知道，对于功能的替换或者重写，很大程度取决于个体成员的工作经验、擅长的开发语言、对业务背景的了解等。譬如，工作经验丰富的开发者通常情况下对其擅长的语言更熟练；而对业务理解较好的开发者，能在更短的时间内完成重写。

因此，多长时间能够重写该服务也不能作为衡量其是否“微”的重要因素。

- **团队觉得好才是真的好**

微服务的“微”并不是一个真正可衡量、看得见、摸得着的微。这个“微”所表达的，是一种设计思想和指导方针，是需要团队或者组织共同努力找到的一个平衡点。

所以，微服务到底有多微，是个仁者见仁，智者见智的问题，最重要的是团队觉得合适。但注意，如果达成“团队觉得合适”的结论，至少还应该遵循以下两个基本前提。

- **业务独立性**

  首先，应该保证微服务是具有业务独立性的单元，并不能只是为了微而微。关于如何判断业务的独立性，也有不同的考量。譬如，可以将某一领域的模型作为独立的业务单元，譬如订单、产品、合同等；也可以将某业务行为作为独立的业务单元，譬如发送邮件、单点登录验证、不同数据库之间的业务数据同步等。更多关于业务建模与划分的资料，请参考 Eric Evans 的《领域驱动模型》一书。

- **团队自主性**

  其次，考虑到团队的沟通及协作成本，一般不建议超过 10 个人。当团队超过 10 个人，在沟通、协作上所耗费的成本会显著增加，而这也是大部分敏捷实践里提

倡的。当团队成员超过 10 个人的时候，可以考虑继续再划分子团队，让不同的子团队承担独立的工作，这也是笔者在实践中通常采用的做法。除此之外，团队应该由不同技能、不同角色的成员组成，是一个全功能的团队。

2.1.2 单一职责

从我们接触编程的第一天起，老师就教我们，编写代码的原则要符合“高内聚、低耦合”。所谓高内聚，是一个模块内各个元素彼此结合的紧密程度高。而低耦合，则是指对于一个完整的系统，模块与模块之间，尽可能独立存在。换句话说，对于每个模块，尽可能独立完成某个特定的子功能。高内聚、低耦合的系统有什么好处呢？在系统持续发展的过程中，高内聚、低耦合的系统具有更好的重用性、可维护性和扩展性，能够持续支持业务的发展，而不会成为业务发展的障碍。

在面向对象的设计中，更是有放之四海而皆准的“SOLID 原则”。熟悉的读者一定知道，SOLID 原则中的 S 表示的是 SRP（Single Responsibility Principle，单一职责原则）：即一个对象应该只有一个发生变化的原因，如果一个对象可被多个原因改变，那么就说明这个对象承担了多个职责。更多关于单一职责原则的解释，请参考罗伯特·C·马丁（Robert C. Martin）的《敏捷软件开发：原则、模式和实践》一书。

实际上，UNIX 的设计是这一原则的完美体现者：在 UNIX 中，各个命令都独立负责一个单一的功能，但命令和命令之间，可以通过管道连接起来，组合实现更强大的功能。譬如，将 du、sort 以及 sed 几个命令用管道连接起来，能方便地获取当前目录下占用空间最多的 5 个资源，如下代码所示：

```
du -s * | sort -nr | sed 5q
```

类似的，对于每个服务而言，我们希望它处理的业务逻辑能够单一，在服务架构层面遵循单一职责原则。也就是说，微服务架构中的每个服务，都是具有业务逻辑的，符合高内聚、低耦合原则以及单一职责原则的单元，不同的服务通过“管道”的方式灵活组合，从而构建出庞大的系统。

2.1.3 轻量级通信

服务之间应通过轻量级的通信机制，实现彼此间的互通互联，互相协作。所谓轻量级通信机制，通常指语言无关、平台无关的交互方式。譬如，某些复杂的系统，由 Java、Ruby

以及 Node 等不同开发语言实现的部分组成，不同部分之间能够采用语言无关、平台无关的方式进行交互，如图 2-2 所示。

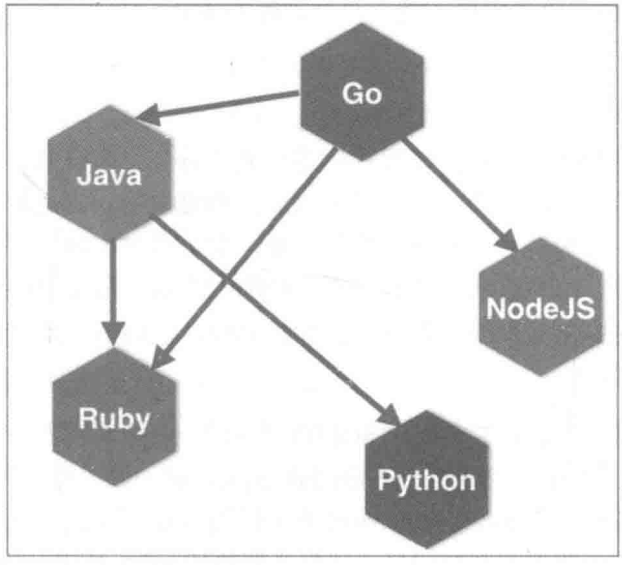


图 2-2　语言无关、平台无关的通信机制

对于轻量级通信的格式而言，我们熟悉的 XML 或者 JSON，它们的解析和使用基本与语言无关、平台无关。对于轻量级通信的协议而言，通常基于 HTTP，能让服务间的通信变得标准化并且无状态化。目前，大家所熟悉的 REST（Representational State Transfer），是实现服务之间互相协作的轻量级通信机制之一。

对于我们所熟悉的 Java RMI 或者.NET Remoting 等，虽然这类机制能够使用 RPC（Remote Procedure Call）的方式简化消费者端的调用，使消费者一端像调用本地接口一样调用远端的接口，但其最大的劣势在于：其对语言或者平台有较强的耦合性，同时灵活性和扩展性较差。譬如，如果使用 Java 的 RMI 作为通信协议，则通信的两端必须采用运行在 JVM 之上的开发语言实现。

对于微服务而言，通过使用语言无关、平台无关的轻量级通信机制，使服务与服务之间的协作变得更加标准化，也就意味着在保持服务外部通信机制轻量级的情况下，团队可以选择更适合的语言、工具或者平台来开发服务本身。

2.1.4 独立性

独立性指在应用的交付过程中，开发、测试以及部署的独立。

在传统的单块架构应用中，所有功能都存在于同一个代码库中。当修改了代码库中的某个功能，很容易出现功能之间相互影响的情况。尤其是随着代码量、功能的不断增加，风险也会逐渐增加。换句话说，功能的开发不具有独立性。

除此之外，当多个特性被不同小组实现完毕，需要经过集成，经过回归测试，团队才有足够的信心，保证功能相互配合、正常工作并且互不影响。因此，测试过程不是一个独立的过程。

当所有测试验证完毕，单块架构应用将被构建成一个部署包，并标记相应的版本。在部署过程中，单块架构部署包将被部署到生产环境或者类生产环境，如果其中某个特性存在缺陷，则有可能导致整个部署过程的失败或者回滚。

因此，在单块架构中，功能的开发、测试、构建以及部署耦合度较高，相互影响，如图 2-3 所示。

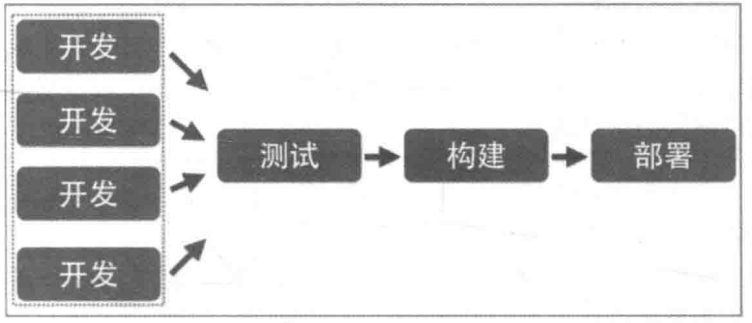


图 2-3 耦合度高

在微服务架构中，每个服务都是一个独立的业务单元，当对某个服务进行改变时，对其他的服务不会产生影响。换句话说，服务和服务之间是独立的。

对于每个服务，都有独立的代码库。当对当前服务的代码进行修改后，并不会影响其他服务。从代码库的层面而言，服务与服务是隔离的。

对于每个服务，都有独立的测试机制，并不必担心破坏其他功能而需要建立大范围的回归测试。也就是说，从测试的角度而言，服务和服务之间是松耦合的。

由于构建包是独立的，部署流程也就能够独立，因此服务能够运行在不同的进程中。从部署的角度考虑，服务和服务之间也是高度解耦的，如图 2-4 所示。

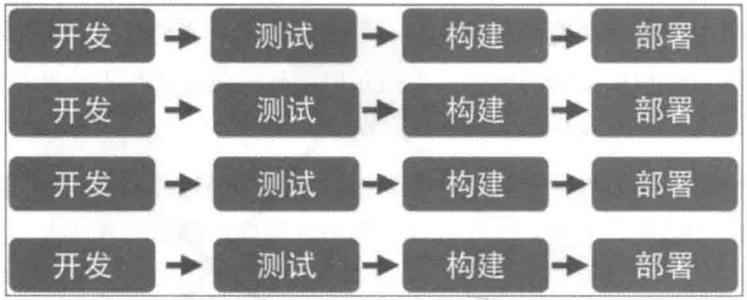


图 2-4　独立性高

对于微服务架构中的每个服务而言，与其他服务高度解耦。只改变当前服务本身，就可以完成独立的测试、构建以及部署等。

2.1.5　进程隔离

在单块架构中，整个系统运行在同一个进程中，虽然我们将应用程序的代码分成逻辑上的三层、四层甚至更多层，但它并不是物理上的分层。这也意味着，经过开发团队对不同层的代码实现，经历过编译、打包、部署后，不考虑负载均衡以及水平扩展的情况，应用的所有功能会运行在某机器的同一个进程中。

所有功能都运行在同一个进程中，也就意味着，当对应用进行部署时，必须停掉当前正在运行的应用，部署完成后，再重新启动进程，无法做到独立部署。如果当前某应用中包含定时任务的功能，则要考虑在什么时间窗口适合部署，是否先停掉消息队列或者切断与数据源的联系，以防止数据被读入应用程序内存，但还未处理完，应用就被停止而导致的数据不一致性。

多年前，我曾经接触过一个 Java 项目，应用程序本身实现了较复杂的业务逻辑，同时还包括一个定时任务的消息发送功能。该定时任务每隔 5 秒会从数据库读入数据、暂存队列，并将其以消息的形式发送出去。在这种情况下，每次应用程序业务相关部分的缺陷修复或者特性增加，都要耗费大量的时间完成部署。因为在部署过程中，团队需要先停掉数据网关，避免新的消息进入队列，然后等待消息被处理完毕，再关闭应用程序，以防止数据被读入到内存中，但还未被处理完而导致数据不一致。

同时，为了提高代码的重用以及可维护性，在应用开发中，我们有时也会将重复的代码提取出来，封装成组件。注意，这里所说的组件，指的是可以独立升级、独立替换掉的部分。在传统的单块架构中，组件通常的形态叫共享库，譬如 JVM 平台下的 JAR 包或者 Windows 下的 DLL 等，它们都是组件的一种形态。当应用程序在运行期时，所有的组件最终也会被加载到同一个进程中运行，如图 2-5 所示。

图 2-5　组件运行在同一进程中

但在微服务架构中，应用程序由多个服务组成，每个服务都是一个具有高度自治的独立业务实体。通常情况下，每个服务都能运行在一个独立的操作系统进程中，这就意味着不同的服务能非常容易地被部署到不同的主机上，如图 2-6 所示。

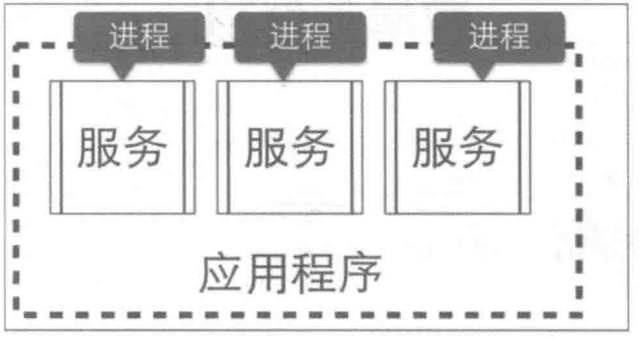


图 2-6　服务运行在不同的进程中

理论上，虽然能够将多个服务部署到同一个节点，并让它们运行在不同的进程中，这

种方式是可行的，但并不推荐这么做。作为运行微服务的环境，我们希望它能够保持高度自治性和隔离性。如果多个服务运行在同一个服务器节点上，虽然省去了节点的开销，但是增加了部署和扩展的复杂度。譬如，当部署某个新服务时，如果当前节点已经运行多个服务，则可能会对这些现有的服务造成影响。

另一方面，当多个服务运行在同一个服务器节点上，如果某些服务随着业务的发展需要水平扩容，但某些服务却不需要，如何有效组织这些服务，哪些服务需要一起被水平扩容，这些问题会为服务的水平扩容带来不必要的麻烦。

综上所述，微服务架构其实是将单一的应用程序划分成一组小的服务，每个服务都是具有业务属性的独立单元，同时能够被独立开发、独立运行、独立测试以及独立部署。

2.2 微服务的诞生背景

微服务的诞生并非偶然。它是在互联网高速发展，技术日新月异的变化以及传统架构无法适应快速变化等多重因素的推动下诞生的产物，如图 2-7 所示。

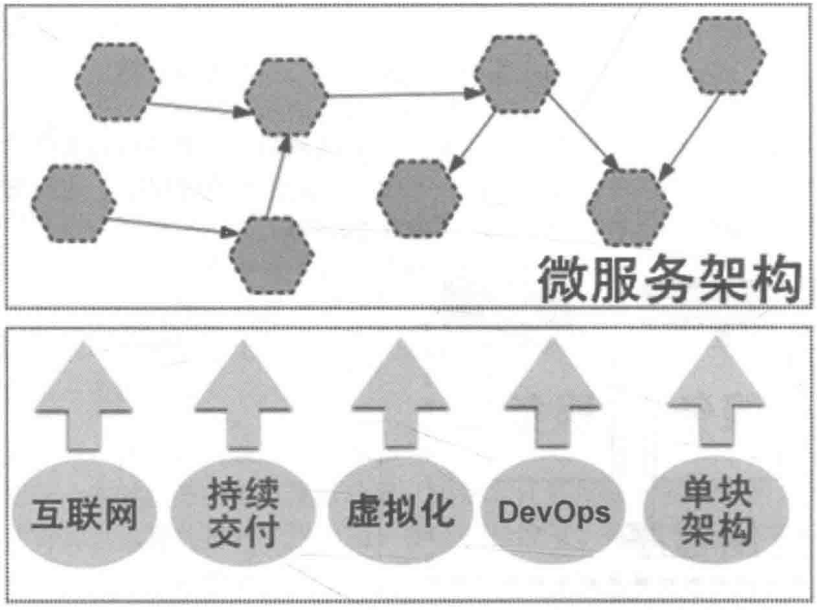


图 2-7 微服务的诞生背景

2.2.1 互联网行业的快速发展

过去的十年中，互联网给我们的生活带来了翻天覆地的变化。购物、出行、订餐、支付，甚至美甲、洗车等，想到的，想不到的活动都可以通过互联网完成，越来越多的传统行业也开始依赖互联网技术打造其核心竞争优势。互联网时代的产品通常有两类特点：需求变化快和用户群体庞大。在这种情况下，如何从系统架构的角度出发，构建灵活、易扩展的系统，快速应对需求的变化；同时，随着用户量的增加，如何保证系统的可伸缩性、高可用性，成为系统架构面临的挑战。

2.2.2 敏捷、精益方法论的深入人心

纵观 IT 行业过去的十年，敏捷、精益、持续交付等价值观和方法论的提出以及实践，让很多组织意识到应对市场变化、提高响应力的重要性。精益创业（Lean Startup）通过迭代持续改进，帮助组织分析并建立最小可实行产品（Minimum Viable Product）；敏捷方法帮助组织消除浪费，通过反馈不断找到正确的方向；持续交付帮助组织构建更快、更可靠、可频繁发布的交付机制；云、虚拟化和基础设施自动化（Infrastructure As Code）的使用则极大简化了基础设施的创建、配置以及系统的安装和部署；DevOps 文化的推行更是打破了传统开发与运维之间的壁垒，帮助组织形成全功能化的高效能团队。经过这些方法论以及实践的推行和尝试后，从宏观上而言，大部分组织已经基本上形成了一套可遵循、可参考、可实施的交付体系。这时候，逐渐完善并改进各个细节的需求就会更加强烈。所谓细节，就是类似如何找到灵活性高、扩展性好的架构方式，如何用更有效的技术、工具解决业务问题等。

2.2.3 单块架构系统面临的挑战

几年前我们熟悉的传统 IT 系统，也可以称之为单块架构系统，是以技术作为切分的要素，譬如逻辑分层、数据中心化等。但随着用户需求个性化、产品生命周期变短、市场需求不稳定等因素的出现，单块架构系统面临着越来越多的挑战。因此，如何找到一种更有效、更灵活、更能适应当前互联网时代需求的系统架构方式，成为大家关注的焦点。

2.2.4 容器虚拟化技术

Docker 是一个开源的应用容器（Linux Container）引擎，允许开发者将他们的应用以及依赖包打包到一个可移植的容器中，然后发布到任何装有 Docker 的 Linux 机器上。

同传统的虚拟化技术相比，基于容器技术的 Docker，不需要额外的 hypervisor 机制的支持，因此具有更高的性能和效率。另外，Docker 的优势也主要体现在以下几个方面：

- 更快速地交付和部署。开发者可以使用一个标准的镜像来构建镜像，开发完成之后，运维人员可以直接使用这个镜像来部署。
- 更轻松地迁移和扩展。Docker 容器可以在任意平台上运行，包括物理机、虚拟机、公有云、私有云等。这种兼容性可以低成本地将应用程序从一个平台直接迁移到另一个。
- 更简单地管理。使用 Docker，所有镜像的修改都能以增量的方式被分发和更新，从而实现自动化并且高效的管理。

Docker 的出现，有效地解决了微服务架构下，服务粒度细、服务数量多所导致的开发环境搭建、部署以及运维成本高的问题。同时，利用 Docker 的容器化技术，能够实现在一个节点上运行成百甚至上千的 Docker 容器，每个容器都能独立运行一个服务，因此极大降低了随着微服务数量增多所导致的节点数量增多的成本。

如果说之前的敏捷、精益、持续交付以及 DevOps 是微服务诞生的催化剂，那 Docker 的出现，则有效解决了微服务的环境搭建、部署以及运维成本高的问题，为微服务朝大规模应用起到了推波助澜的关键作用。

所以，微服务的诞生绝不是偶然，是多重因素推动下的必然产物。

2.3 微服务架构与 SOA

2.3.1 SOA 概述

早在 1996 年，Gartner 就提出了面向服务架构（SOA）。SOA 阐述了“对于复杂的企业 IT 系统，应按照不同的、可重用的粒度划分，将功能相关的一组功能提供者组织在一起为消费者提供服务”，其目的是为解决企业内部不同 IT 资源之间无法互联而导致的信息孤岛问题。更多关于 SOA 的介绍，请参考 https://en.wikipedia.org/wiki/Service-oriented_architecture。

2002 年，SOA 被称作“现代应用开发领域最重要的课题之一”，其正在帮助企业从资源利用的角度出发，将 IT 资源整合成可操作的、基于标准的服务，使其能被重新组合和应用。

由于 SOA 本身的广义性以及抽象性，在其诞生的相当长一段时间内，人们对 SOA 存

在着不同的认知和理解，如图 2-8 所示。

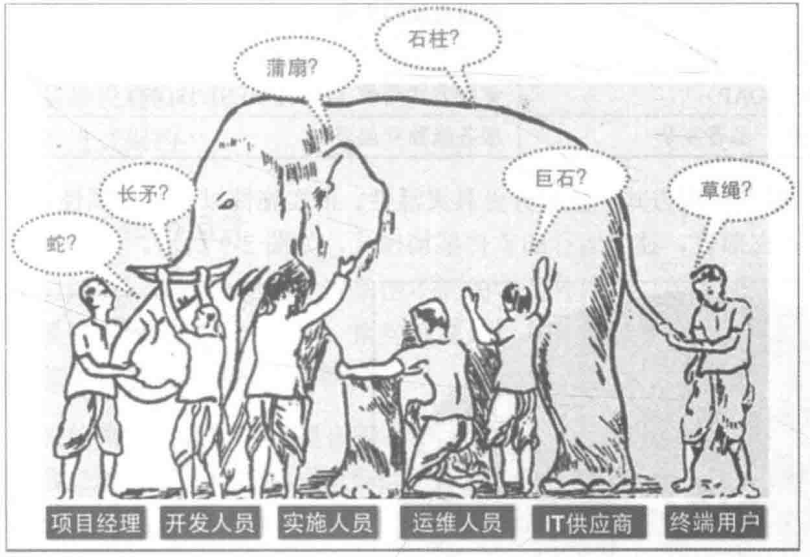


图 2-8　对 SOA 的不同认知（原图来自百度百科-盲人摸象）

直到 2000 年左右，ESB（Enterprise Service Bus）、WebService、SOAP 等这类技术的出现，才使得 SOA 渐渐落地。同时，更多的厂商，像 IBM、Oracle 等也分别提出基于 SOA 的解决方案或者产品。

2.3.2　微服务与 SOA

实际上，微服务架构并不是一个全新的概念。仔细分析 SOA 的概念，就会发现，其和我们今天所谈到的微服务思想几乎一致。那在 SOA 诞生这么多年后，为什么又提出了微服务架构呢？

鉴于过去十几年互联网行业的高速发展，以及敏捷、持续集成、持续交付、DevOps、云技术等的深入人心，服务架构的开发、测试、部署以及监控等，相比我们提到的传统的 SOA 实现，已经大相径庭，主要区别如表 2-1 所示

表 2-1　SOA 与微服务的区别

SOA实现	微服务架构实现
企业级，自顶向下开展实施	团队级，自底向上开展实施
服务由多个子系统组成，粒度大	一个系统被拆分成多个服务，粒度细

续表

SOA实现	微服务架构实现
企业服务总线，集中式的服务架构	无集中式总线，松散的服务架构
集成方式复杂（ESB/WS/SOAP）	集成方式简单（HTTP/REST/JSON）
单块架构系统，相互依赖，部署复杂	服务能独立部署

相比传统 SOA 的服务实现方式，微服务更具灵活性、可实施性以及可扩展性，其强调的是一种独立测试、独立部署、独立运行的软件架构模式，如图 2-9 所示。

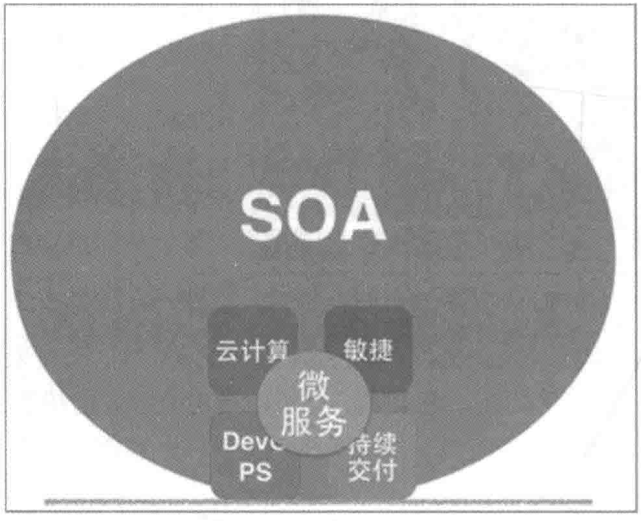


图 2-9　微服务是 SOA 的子集

综上所述，对于微服务的概念而言，它是传统 SOA 的定义的一个子集；而对于其实现方式而言，它是一种更符合现代化互联网发展趋势的实践，是一种更容易帮助企业或组织有效并成功实施服务架构的实践。

2.4　微服务的本质

所谓本质特征，就是当我们谈论同一件事情的时候，不同的人们所关注的相同的部分。从业界的讨论来看，微服务的本质特征通常包括如下几个部分：

- 服务作为组件
- 围绕业务组织团队

- 关注产品而非项目
- 技术多样性
- 业务数据独立
- 基础设施自动化
- 演进式架构

2.4.1 服务作为组件

我们知道，在制造领域，汽车由不同的零部件组成，当某个零部件出现故障时，修理中心能使用备用的零部件替换。换句话说，汽车的每个零件都是可以独立升级、独立替换的，有着较好的灵活性和可替换性。

在软件领域，我们一直提倡使用组件（Component）的方式，将应用模块化并为其构建相对独立的单元。传统实现组件的方式是隔离独立的部分或抽取公用的部分，构建共享库（Library），从而达到解耦和复用的效果。不过，对于共享库而言，通常它是语言相关、平台相关的，并且是和应用程序运行在同一个进程中的，如图 2-10 所示。

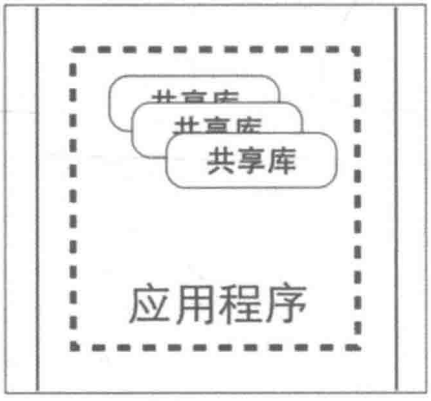


图 2-10 共享库作为组件

换句话说，共享库的变化意味着整个应用要被更新，并且需要被重新部署。如果应用由多个共享库组件组成，那么任何库的变更都将导致应用重新发布。

实际上，微服务也可以被认为是一种组件。如果把微服务作为组件，则同传统使用组件方式最大的区别是，组件可以被独立部署。譬如，应用由多个共享库组成，并且运行在一个进程中，那么任何共享库的变更都将导致整体应用的重新发布。但如果应用由多个服

务构成，并且服务是独立的，则大部分情况下，每个服务的变更仅需要部署自身，并不会影响其他服务。

因此，微服务架构的一个显著优势是，能以松散的服务方式，构建可独立化部署的模块化应用，如图 2-11 所示。

图 2-11　服务作为组件

把服务当成组件的另外一个优点是，在组件和组件之间定义了清晰的、语言无关、平台无关的接口。许多开发语言虽然定义了良好的公共调用接口，也提供了详尽的文档和规范说明，但由于共享库本身的特性，充分依赖于特定平台、特定语言，因此组件间的耦合度较高。

同共享库相比，微服务是通过语言无关、平台无关的轻量级通信机制协作，因此灵活性非常高。当然，使用微服务也有它的不足之处，就是分布式调用比进程内调用更消耗时间，并且严重依赖于网络的可靠性与稳定性。

2.4.2　围绕业务组织团队

在单块应用架构时代，为了节省成本、提高人员效率，企业或者组织一般都会根据技能划分团队。譬如，用户体验设计师，一般都被划分到设计团队；懂服务器端的开发人员，一般都被归类为后端业务逻辑开发团队；精通数据库技能的开发者，一般会在 DBA 团队中找到他们的身影，如图 2-12 所示。

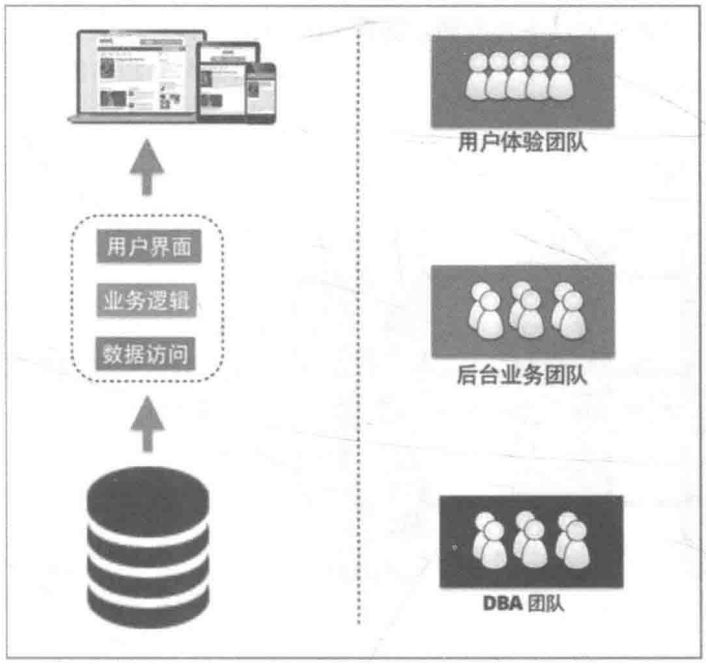


图 2-12　按技能划分团队

当团队被按照这个策略或者维度划分后，即便是某些简单的需求变更，都有可能出现跨组织、跨团队的协作，沟通协作成本较高。

正如康威定律所述，一个组织的设计成果，其结构往往对应于这个组织中的沟通和组织结构。

微服务架构的团队组织方式不同于传统的团队组织方式，它提倡以业务为核心，按业务能力来组织团队，团队中的成员具有多样性的技能。

2.4.3　关注产品而非项目

单块架构应用大部分都是基于项目模式构建的。什么是项目模式？就是当项目启动后，企业或者组织会从不同的技能资源池中抽取相关的资源，组成团队并完成项目。譬如，从开发团队中抽出一部分开发人员、从测试团队中抽出一部分测试人员，以及从其他不同技能团队中抽出需要的人员，组成一个项目团队，然后设置一个时间期限，团队在规定时间内完成项目，就算取得了成功。当项目结束后，所有的资源都会被释放。

对于这种模式，其存在的弊端在于以下几方面，如图 2-13 所示。

- 团队成员缺乏主人翁精神
- 难以制定有效的奖惩机制
- 团队成员缺乏产品成就感

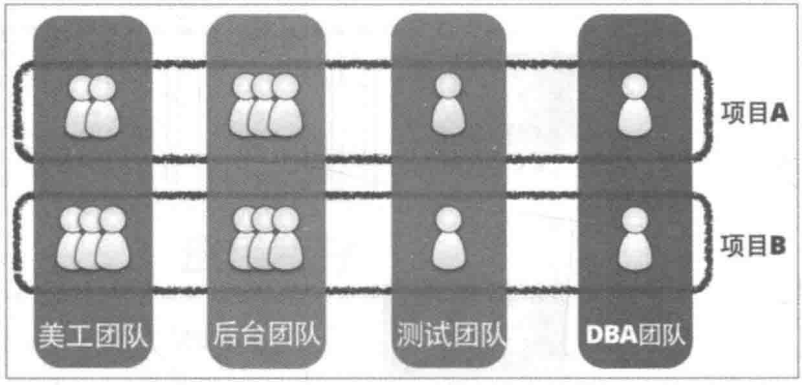


图 2-13　基于项目构建团队

微服务架构提倡的是采用产品模式构建，即更倾向于让团队负责整个服务的生命周期。从服务的分析、开发、测试、部署、运维。所有成员的个人目标和团队的目标一致，都是为了更有效、高效、以可持续性发展的方式为消费者提供业务功能，如图 2-14 所示。

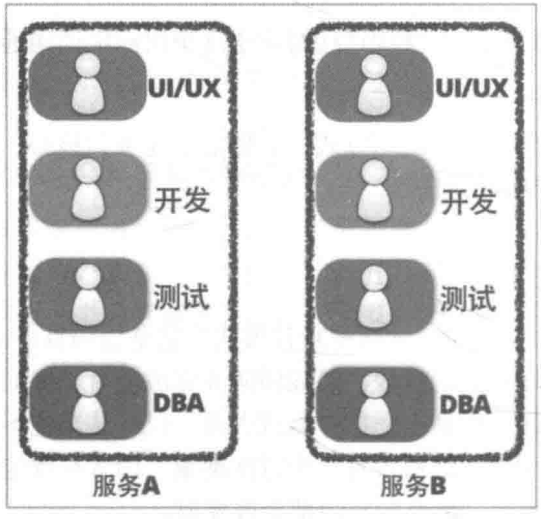


图 2-14　基于产品构建团队

当然，团队有可能负责多个服务，因为最终的目标是通过多个服务的协调、组合实现产品的功能，以及传递价值。

亚马逊的 CTO Werner Vogels 曾经说过一句话，“You build it, you run it”。对团队而言，产品就是团队的，也是每个成员的。团队中的每个人都有责任、有义务确保产品的快速发展以及演进。

2.4.4 技术多样性

传统的单块应用架构，倾向于采用统一的技术平台或方案来解决所有问题。其实我们知道，在现实生活中，并不是每个问题都是钉子，也不是每个解决方案都是锤子。问题有其具体性，解决方案也应有其针对性。用最适合的技术方案去解决具体的问题，往往会事半功倍。

在微服务架构中，提倡针对不同的业务特征选择适合的技术方案，有针对性地解决具体的业务问题，示意图如图 2-15 所示。

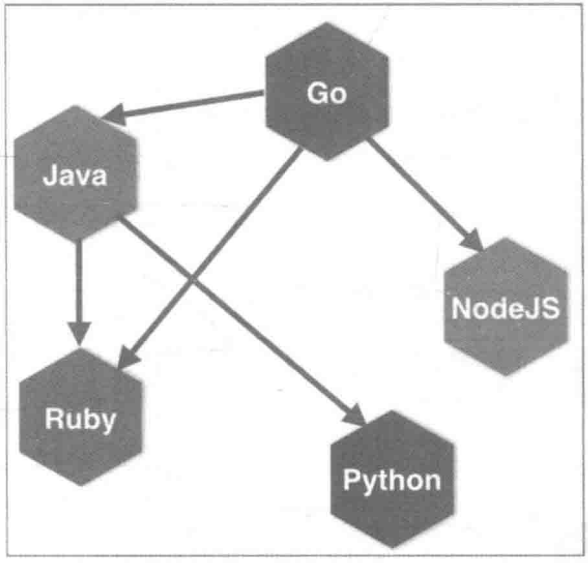


图 2-15 技术多样性

另外，对于单块架构系统，初始的技术选型严重限制其将来能否采用不同语言或框架的能力。如果想尝试新的编程语言或者框架，没有完备的功能测试集，很难平滑地完成替换，而且系统规模越大，风险越高。

微服务架构，使我们更容易在系统上尝试新的技术或解决方案。譬如，可以先挑选风险最小的服务作为尝试，快速得到反馈后再决定是否适用于其他服务。这也意味着，即便对一项新技术的尝试失败，也可以抛弃这个方案，并不会对整个产品带来风险。

2.4.5 业务数据独立

传统的单块应用架构，倾向于采用统一的数据存储平台来存储所有的数据。随着业务的快速发展，需求的不断变化，一方面，数据变得越来越复杂，难以管理；另一方面，随着应用系统的业务逻辑不断更新和发展，数据库不仅承担着数据存储的作用，还承担着不同系统之间的集成作用。

同时，传统的数据库大多是关系型数据库，存储的数据都是以结构化信息为主，但随着互联网的快速发展，数据的结构并不具有确定性，或者说结构发生变化的频率非常快，因此，对于如何有效维护业务数据，也成了一个难题，相应的维护成本会越来越高。

微服务架构，提倡服务自主管理其相关的业务数据。这样存在几个非常明显的优势：

- 能够随着业务的发展，提供业务数据接口集成，而不是以数据库的方式同其他服务集成。
- 能够随着业务的发展，选择更适合的工具管理或者迁移业务理数据。

譬如，在一个复杂的后台 CRM 系统中，产品数据的种类繁多，更新也比较频繁，可以使用类似 MongoDB 这种文档数据库，可灵活地根据需求动态调整结构。对于用户访问系统时产生的会话信息，则可以使用 Redis 等键值系统进行存储；报表数据的结构变化不大，而且要求数据的高一致性，则可以使用传统的关系型数据库，如图 2-16 所示。

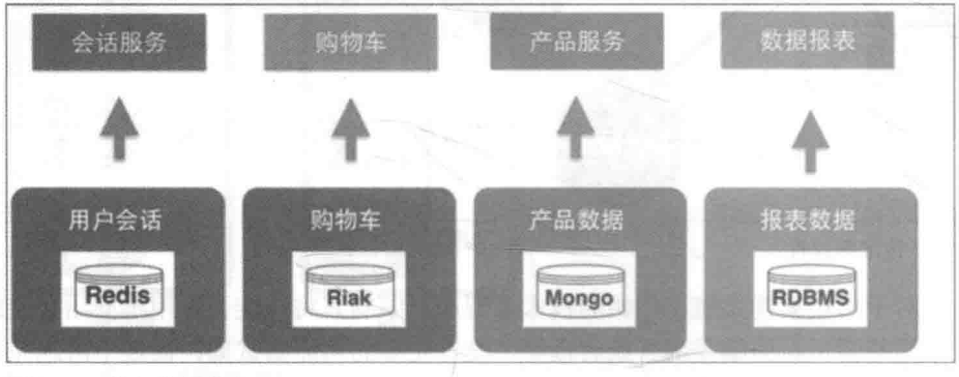


图 2-16 业务数据独立

2.4.6 基础设施自动化

微服务架构将应用程序本身分成多个小的服务，每个服务都是一个独立的部署单元。因此，传统只需要部署一次就能上线的单块架构应用，采用微服务架构后，将需要对每个部分分别进行部署。另外，每个服务都需要部署带来的健康监控、错误回滚、日志分析等，成本也会显著增加。换句话说，部署与运维的成本会随着服务的增多呈指数级增长。

因此，在微服务的实践过程中，对持续交付和部署流水线要求较高。微服务的粒度越细，就意味着需要部署的业务单元越多，业务单元越多，就需要更稳定的基础设置自动化机制，能够创建运行环境，安装依赖，部署应用等。

不过随着云技术的大规模推广与使用，部署和运维的复杂度在大幅度降低。利用云，我们可以快速创建系统需要的资源，降低应用的部署成本。同时，由于持续集成、持续交付等实践的深入人心，很多团队都开始在构建软件的过程中，使用持续交付提倡的基础设施自动化技术。

因此，微服务的实践将促使企业或者团队不断寻找更高效的方式完成基础设施的自动化以及 DevOps 运维能力的提升。

2.4.7 演进式架构

在过去十年中，敏捷方法论及其实践已经被越来越多的组织尝试并认可。敏捷方法论正在帮助组织以拥抱变化的心态，不断尝试，不断获取反馈，从而以高效的方式构建正确的应用系统。实际上，敏捷并不是一种静止的状态，它是组织一直在拥抱变化、尝试改变、获取反馈的演进式发展的一个动态过程。

类似的，架构设计也应该是随着业务的发展而不断发展，随着需求的变化而不断变化的。当我们试图构建一个单块应用架构系统时，会面临非常艰难的技术选型。哪种方案才是合理的？那种方案才是最正确的？

在传统的单块架构设计中，企业或者组织通常是希望构建一个大而全、无所不能的平台，但是在技术发展如此之快的今天，单一的技术平台已经无法适应市场的快速变化，组织应该随着业务的发展，随着企业的发展，不断尝试并改进架构设计，真正做到业务驱动架构，架构服务于业务。

微服务架构将一个复杂应用拆分成多个服务，每个服务都是一个独立的、可部署的业务单元。同时，每个服务都能独立运行在独立的进程中，并且服务之间通过轻量级的通信机制建立联系。

从某种程度上而言，这些特点保证了在应用软件随着业务发展而不断变化的过程中，企业或者组织能够不断调整软件的架构，将不需要的服务（业务）抛弃，将需要的服务（业务）升级，并采用合适的技术或者工具不断优化架构，保持其处于一个不断演进的状态。

2.5 微服务不是银弹

在之前的章节中，我们已经讨论了什么是微服务，微服务的本质特征以及它区别于传统单块架构的优势所在。对于微服务而言，其优势是很明显的。

- **独立性**

每个服务都是独立的业务单元，能够被独立地开发、测试、构建，并且能够被直接部署。

- **单一职责**

每个服务聚焦于某业务功能，通过清晰的边界划分，更容易被团队理解和维护。

- **技术多样性**

通过服务之间的协作以及轻量级的通信机制，组织或者团队能使用适合的语言、工具解决业务问题。

除此之外，微服务的实施也会推动基础设施自动化以及 DevOps 文化在团队中的发展，并有利于构建全功能的团队。

微服务的优势，解决了传统的单块架构系统随着业务需求的快速变化而面临的挑战，使得其看起来像一个完美的解决方案。那微服务是否是传说中的银弹？

实际上，在微服务的实施过程中，需要考虑如下因素：

- 分布式系统的复杂度
- 运维成本
- 部署自动化
- DevOps 与组织架构

- 服务间依赖测试
- 服务间依赖管理

2.5.1 分布式系统的复杂度

微服务架构是一种基于分布式的系统，从交付的角度出发，构建分布式系统必然会带来额外的开销。通常，分布式系统的复杂度主要包括以下几点。

- **性能**

同传统的单块架构相比，分布式系统由于组件与组件的调用是跨进程、跨网络的调用，因此必然要考虑网络延迟以及带宽的影响。尤其要考虑，当某业务场景需要多个服务相互协作时，响应时间以及性能对系统的影响。

- **可靠性**

在分布式系统中，由于网络、带块、节点自身的可靠性等因素，任何一次组件间的远程调用都有可能失败。而且，随着微服务数量的增多，还会出现更多的潜在故障点。因此，如何提高系统的可靠性，降低由于网络、组件等引起的单点故障率，也增加了系统构建的挑战。

- **异步**

对于跨网络的调用，需要考虑异步的通信机制。我们知道，同步通信的过程一般是发送请求，接收响应并处理，整个过程实现简单，但会造成阻塞。异步通信的过程则是发送请求后立即返回，不会造成阻塞，一般适用于耗时操作的处理。但异步通信在享受非阻塞的优势的同时，也大大增加了功能实现的复杂度，并且当出现缺陷时，定位问题、调试问题的难度也更大。

- **数据一致性**

在分布式系统中，为了保证数据的一致性，通常我们会考虑使用分布式事务管理。但由于分布式事务管理需要跨多个节点来保证数据的瞬时一致性，因此比起传统的单块架构的事务，成本要高得多。另外，在分布式系统中，通常也会考虑通过数据的最终一致性来解决数据瞬时一致带来的系统不可用。

- **工具**

虽然现有的 IDE 以及开发工具对单块架构系统的开发支持较好，但并没有为开发分布

式系统提供足够的支持。因此，相比传统的单块架构，分布式架构的开发、调试存在较大的复杂度。

2.5.2 运维成本

通常，我们说的运维主要包括以下几个方面。

- **配置**
  主要包括应用相关的配置信息，譬如参数、依赖部分、数据库地址、缓存地址等。
- **部署**
  主要包括将应用部署到指定的环境中。
- **监控与告警**
  主要包括对应用的健康状况进行监控，并当发现故障时能及时告警。
- **日志收集**
  主要包括日志收集，并提供搜索等方式，帮助团队通过日志快速定位问题。

相比传统的单块架构应用，微服务将系统分成多个独立的部分，每个部分都是可以独立部署的业务单元。这就意味着，原来适用于单块架构的集中式的部署、配置、监控或者日志收集等方式，在微服务架构下，随着服务数量的增多，每个服务都需要独立的配置、部署、监控、日志收集等，因此成本呈指数级增长。

2.5.3 部署自动化

对于单块架构的系统而言，部署包通常只有一个或者很少的几个，部署的操作成本较低。同时，在单块架构的时代，通常组织的交付周期都以周、月为单位。在这种情况下，由手动方式来完成系统的部署，是可以满足需求的。

对于微服务架构而言，每个服务都是一个独立可部署的业务单元，每个服务的修改，都需要独立部署。同时，随着互联网时代的快速发展、用户需求的个性化以及市场需求不稳定等因素的出现，系统的交付周期越来越短，部署的频率越来越高，譬如像亚马逊，每天都要执行数十次、甚至上百次的部署，如果还是依赖手动部署、人工审查等机制，已经无法适应互联网时代的快速变化。因此，微服务系统的部署成本较高。

因此，如何有效地构建自动化部署流水线，降低部署成本、提高部署频率，是微服务架构下需要面临的一个挑战。

2.5.4 DevOps 与组织架构

对于传统单块架构，团队通常是按照技能划分。譬如开发部、测试部、运维部等，并通过项目的方式协作，完成系统的交付。

而在微服务架构的实施过程中，除了如上所述的交付、运维上存在的挑战，在组织或者团队层面，如何传递 DevOps 文化的价值，让团队理解 DevOps 文化的价值，并构建全功能团队，也是一个不小的挑战。

实际上，微服务不仅表现出一种架构模型，同样也表现出一种组织模型。这种新型的组织模型意味着开发人员和运维的角色发生了变化，开发者将承担起服务整个生命周期的责任，包括部署和监控，而运维也越来越多地表现出一种顾问式的角色，尽早考虑服务如何部署。

因此，如何在微服务的实施中，按需调整组织架构，构建全功能的团队，是一个不小的挑战。

2.5.5 服务间的依赖测试

对于传统的单块架构而言，通常使用集成测试验证系统是否和其外部的依赖之间正常协作。

而对于微服务架构而言，由于系统被拆分成多个可独立部署的、分布式的业务单元，因此，服务之间的交互主要通过接口完成。在服务数量逐渐增多的情况下，如何有效地保证服务之间能有效按照接口的约定正常工作，成为微服务实施过程中，测试面临的主要挑战。

2.5.6 服务间的依赖管理

对于传统的单块架构而言，功能实现比较集中，大部分功能都运行在同一个应用中，同其他系统依赖较少。

对于微服务架构而言，当把传统的系统拆分成多个相互协作的独立服务后，随着微服务个数的增多，如何清晰有效地展示服务之间的依赖关系，逐渐成为挑战。这也直接影响着团队对系统的理解和对其维护的信心。

综上所述，微服务架构的出现，解决了传统的单块架构系统随着业务需求快速变化所面临的挑战。但在微服务的实施过程中，注定不是一帆风顺的。因此，清楚地了解微服务

架构所带来的风险，能帮助组织或者团队找到更适合的方式，从而有效地实施微服务。

2.6 小结

微服务架构将一个应用拆分成多个独立的、具有业务属性的服务，每个服务运行在不同的进程中，服务与服务之间通过轻量级的通信机制互相协作、互相配合，从而为终端用户提供业务价值。同时，每个服务可以根据业务逻辑，采用不同的语言、框架、工具以及存储技术来解决业务问题。因此，微服务架构强调的是一种独立开发、独立测试、独立部署、独立运行的高度自治的架构模式，也是一种更灵活、更开放、更松散的演进式架构。

通过本章所介绍的微服务的定义、核心特征以及优缺点，帮助我们更清晰、深刻地理解了微服务架构的理论基础。

第 2 部分

实践篇

俗话说“不积跬步，无以至千里”，当我们掌握了如何构建一个微服务，就可以通过不断地扩展、衍生，构建更多的服务来完成一个复杂的系统。

对于每个服务而言，由于其本身粒度较小，模块化也较清晰，因此在一开始，最重要的事情不是立刻实现功能上的代码，而是优先考虑如何在交付的过程中，从工程实践出发，组织好代码结构、配置、测试、部署、运维、监控的整个过程，从而有效体现微服务的独立性与可部署性。有了这个过程，团队就能够在接下来的时间里，频繁、有序地实现业务逻辑，并进行快速迭代，从而驾驭微服务的快速交付。

在之前的基础篇中，笔者和大家探讨了什么是微服务、微服务的优缺点以及微服务的本质特征，这些是帮助我们理解微服务的理论基础。在接下来的实践篇中，笔者就以构建产品服务（products-service）为例，谈谈如何从工程实践的角度出发，构建一个微服务。

本部分的内容主要包括：

- 产品服务分析。分析并定义服务的功能与接口。

- Hello World API。实现一个最简单的“Hello World” API，定义项目的基本结构并尽快将其部署到测试环境、类生产环境。
- 构建 Docker 映像。利用 Docker 构建服务的部署映像。
- 部署 Docker 映像。基于亚马逊 AWS 构建服务的基础设施，并使用 AWS CloudFormation 完成基础设施的自动化构建与配置。然后，使用 Docker 在 AWS EC2 节点上部署 Docker 映像。
- 持续交付流水线。基于 SNAP-CI，构建持续交付流水线，并定义提交、验证、构建以及发布等 4 个阶段。
- 日志聚合、监控与告警。
- 功能迭代。

第 3 章

构建第一个服务

3.1 场景分析

某企业的 CRM 系统，随着业务的发展，需要将产品的模块，从现有的单块架构中剥离出来，形成一个用于专门提供产品数据的服务，便于维护和扩展。

假定产品具有如下属性：产品名称、产品分类、价格、创建时间、修改时间等（真实情况下产品的属性可能会更复杂）。首先，我们来分析一下如何构建这样一个提供产品数据的服务，应具备哪些功能以及接口。

如下的部分描述了产品服务基于 REST 的接口。

- 创建一个新产品

```
Path:         /products
Verb:         POST
Content-Type:  application/json
Body:         {
               "name":     "Microservice in action",
               "category": "Book",
               "price":    50.00
              }
```

- 获取产品列表

```
Path:         /products
```

```
Verb:         GET
Content-Type: application/json
```

- 获取某产品明细

```
Path:         /products/:id
Verb:         GET
Content-Type: application/json
```

- 修改某产品明细

```
Path:         /products/:id
Verb:         PUT
Body:         {
                 "name":     "Microservice in action",
                 "category": "Book",
                 "price":    99.00
              }
```

- 删除某产品

```
Path:         /products/:id
Verb:         DELETE
```

关于REST

表述性状态转移（Representational State Transfer，简称REST）是Roy Fielding博士在2000年他的博士论文中提出的一种软件架构风格。这是一种针对分布式应用系统设计和开发的方式，能有效降低系统的复杂性，提高可扩展性。

同传统的SOAP或者XML-RPC相比，REST是一种更简洁和轻量级的架构风格，已经有越来越多的Web服务开始采用REST风格设计和实现相关系统。需要注意的是，REST是设计风格而不是标准。REST通常使用HTTP、URI和XML以及JSON这些现有的广泛流行的协议和标准。

REST以资源为中心，将外部访问的所有信息定义成不同的资源，并使用HTTP的verbs，映射为操作资源的行为。如果想了解更多关于REST的知识，请参考Roy Fielding博士的文章*Architectural Styles and the Design of Network-based Software Architectures*。

3.2 任务拆分

之前我们提到，使用微服务构建系统的前期、部署、运维以及持续交付的重要性远远大于实现功能本身。因为微服务的粒度较小，功能和边界都比较清晰，因此在整个交付过程中，功能实现存在的风险较小。相比而言，如何从工程实践出发，尽早将开发、测试、部署、运维、监控的流水线打通，才能帮助团队更好地驾驭微服务。

在之前的章节中，我们已经清楚了业务上需要达成的目标以及技术选型，那接下来，让我们看看如何从工程实践出发，尽早将开发、测试、部署、运维、监控的流水线打通。

通常，在笔者的实践中，动手之前会将整体要做的工作内容划分成小的任务，这也是敏捷实践里经常推荐的方式，这样做的益处在于：

- 同一时间聚焦一个任务。
- 能对每次完成的部分做持续集成。
- 整体的进度容易追踪。

因此，在动手开始实现 products-service 之前，我们先将其划分成如下几个任务。通常，在笔者的实践中，习惯用思维导图的方式将工作划分成不同的任务，如图 3-1 所示。

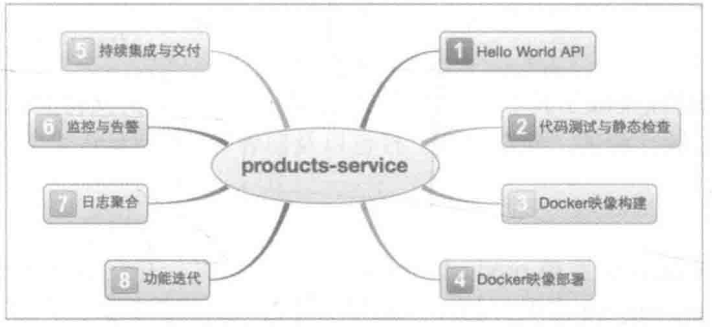


图 3-1 任务拆分

接下来，就让我们看看具体每个任务包括什么内容。

（1）Hello World API

作为程序员，“Hello World”对我们来说一点都不陌生。每当学习新的语言或者技术时候，我们总喜欢用输出 Hello World 来表示迈开的伟大的第一步。

在笔者参与的项目中，当开始构建一个服务时，做的第一步是实现一个“Hello World”

版本的 API，然后提供测试、静态检查等机制并尽快将其部署到测试、类生产环境或者生产环境中，尽早打通整个交付的流水线。

因此，我们首先将着手构建一个最简单的 API。

（2）代码测试与静态检查

代码测试的重要性已经毋庸置疑，它能帮助团队以较低成本的方式发现缺陷。同时，也能帮助团队有效地对存在的功能进行自动化的回归测试。

我们将为 products-service 搭建代码测试与静态检查机制。

（3）构建 Docker 映像

我们将使用 Docker，基于 products-service 的代码库，构建可以部署的映像。

（4）部署 Docker 映像

我们将基于 products-service 的 Docker 映像，利用脚本将其部署到测试环境、类生产环境或者生产环境中。同时，我们也会讨论如何使用亚马逊 AWS EC2 以及 Cloudformation 等工具自动化 products-service 的基础设施。

（5）持续集成与交付

在该部分中，我们将借助持续集成工具 Snap-CI，构建 products-service 的交付流水线，包括自动检测代码提交事件、运行测试、静态检查、打包以及部署等。

（6）监控与告警

在该部分中，我们将使用 Nagios 监控 products-service 的可用状况，并使用 PagerDuty 完成消息提醒，确保当 products-service 出现问题时，能够及时通知到相关的责任人。

（7）日志聚合

在该部分中，我们将使用 Splunk，对不同节点的日志进行聚合，方便团队通过日志分析问题。

（8）功能迭代

有了如上所述的 API、测试、静态检查、流水线部署以及监控与告警，团队就可以在此基础上，以不断迭代的方式，实现后续的业务功能，并持续交付价值。

第 4 章

Hello World API

作为程序员，“Hello World”对我们一点都不陌生。每当学习新的语言或者技术的时候，我们总喜欢用输出“Hello World”来表示迈开了伟大的第一步。

在笔者参与的项目中，当开始构建一个服务时，第一步要做的也是实现一个“Hello World”的版本，然后将其部署到测试环境、类生产环境或者生产环境中，从而尽早打通整个交付流水线。

接下来的部分，笔者将首先介绍使用的开发语言——Ruby 和 Grape，并完成一个“Hello World”API 的实现。

4.1 API 实现

4.1.1 开发语言——Ruby

近几年，笔者一直使用 Ruby 语言从事微服务相关的开发。Ruby 是一门面向对象的动态语言，相比于传统的 Java、C++等面向对象语言，Ruby 的面向对象机制实现得更彻底。在 Ruby 语言中，万物皆对象，包括方法、类，甚至字符、数字等基本数据类型，参见图 4-1。

除此之外，Ruby 的动态语言特性为其提供了强大的元编程能力，包括在运行期修改类或者实例的行为，通过模块（Module）定义抽象的方法集合、Mixin 机制、Block 等。

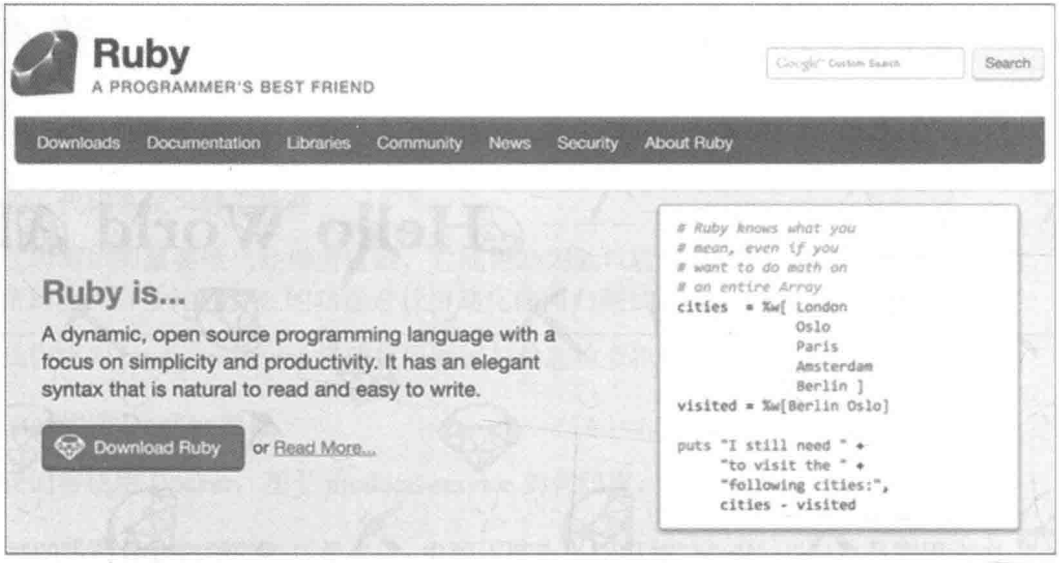


图 4-1　Ruby（图片来自 http://www.ruby-lang.org）

对于当前的例子，笔者将使用 Ruby 作为开发语言。读者也可以根据经验或者喜好，使用其他开发语言。

Ruby语言是一门面向对象的动态语言。在20世纪90年代中期由日本人松本行弘（Matz）设计并开发。Ruby的灵感与特性来自于Perl、Smalltalk、Eiffel、Ada以及Lisp语言。松本行弘于1993年2月开始编写Ruby，直至1995年12月才正式公开发布。其名称的来历和Perl语言有异曲同工之妙，因为Perl的发音与6月的诞生石pearl（珍珠）相同，而Ruby选择以7月的诞生石ruby（红宝石）命名。

2004年，Ruby on Rails框架诞生，这使得Ruby更加广为人知。2006年，Ruby获选TIOBE年度编程语言。同时，在其他语言社区，还发展出JRuby（Java平台）、IronRuby（.NET平台）等其他平台的 Ruby语言替代品。

4.1.2　Web 框架——Grape

在 Ruby 的世界里，有许多用于构建 API 的框架。Ruby on Rails 便是其中最著名的框架之一。Ruby on Rails 是由 DHH（David Heinemeier Hansson）于 2003 年基于 Ruby 开发的 MVC 框架，其包含了很多优秀的工具和库，譬如 ActiveRecord、ActiveMailer、ActiveJob 等。

Ruby on Rails 最大的优势在于提供了大量易于构建 Web 应用的工具库，包括前端页面交互、展示以及后端处理逻辑，还包括类似数据库的创建、Schema 的版本管理等。随着过去十多年来社区对 Rails 的支持和贡献，Rails 提供的功能越来越强大，引入的库越来越多，其本身也变得越来越重量级。渐渐的，Ruby 社区又诞生了很多轻量级的框架。具有代表性的主要有：

- Sinatra
- Webmachine
- Grape

Grape 作为一个轻量级的 API 框架，相比其他框架而言，其主要优势在于：

- 符合 REST 风格。
- 轻量级，能够运行在 Rack 服务器上。
- 文档友好，使用 DSL 简化了 API 的开发。
- 简洁的版本控制、路由管理以及接口参数管理。

基于 Grape 的如上优势，在当前例子中，笔者使用 Grape 作为 Web 框架。更多关于 Grape 的信息，请参考其官方文档（http://intridea.github.io/grape）。

4.1.3 API 的具体实现

接下来，我们就来看看如何实现这个“Hello World”API。

（1）首先，创建一个空的目录 products-service，代码如下所示：

```
$> mkdir products-service
$> cd products-service
```

（2）创建.ruby-version 文件，并添加如下内容：

```
2.1.5
```

.ruby-version 文件用于指定当前应用使用的 Ruby 的版本。

如上代码所示，在当前 products-service 中，使用的 Ruby 版本为 2.1.5。

（3）创建 Gemfile 文件，并添加如下内容：

```
source 'http://ruby.taobao.org'     #使用淘宝的 Gems 镜像
source 'https://rubygems.org'       #使用官方的 Gems 镜像
```

```
gem 'grape'
```

如上代码所示，首先指定 Ruby Gem 的镜像源，国内用户如果使用 rubygems.org 较慢的话，可以尝试淘宝提供的 Gem 镜像 ruby.taobao.org。

另外，向 Gemfile 中添加 Grape 并使用如下命令安装 Gem：

```
$> bundle install
```

（4）创建 api/api.rb 文件，并添加如下内容：

```
require 'grape'
class API < Grape::API
  format :json

  get '/' do
    'Hello World'
  end
end
```

如上代码所示，定义类 API 并使其继承自 Grape::API，同时定义当接受 HTTP 请求为'/'时，返回字符串“Hello world”。

（5）新建 lib/init.rb 文件，并添加如下内容：

```
project_root = File.dirname(__FILE__) + '/..'
$LOAD_PATH << "#{project_root}/api"
require 'grape'
require 'api'
```

如上代码所示，第一行首先定义了项目的根路径，然后将其设置到 Ruby 的 LOAD_PATH 中。第三行开始声明项目启动时依赖的库，包括 grape 和之前定义的 API 类。

（6）新建文件 config.ru，并添加如下内容：

```
require_relative 'lib/init'
run Rack::Cascade.new [API]
```

如上代码所示，第一行首先加载 init 配置，然后使用 Rack 的方式启动 API。

什么是Rack

Rack是Ruby世界构建Web应用的抽象接口，其最初的灵感来自于Python的wsgi（http://www.wsgi.org/wsgi）。由于Rack的简单性和有效性，它将框架编写者从为每个Web服务器单独编写处理模块的负担中解脱出来，并很快在Ruby社区成为开发Web应用的标准模块化接口。

什么是Rack Gem

Rack Gem是一个Rack辅助类的集合，它封装了基本的request、response、cookies以及sessions处理的实现，可以更容易地让Ruby应用支持主流的Web服务器，譬如Passenger、Puma、Unicorn等。关于更多Rack Gem的信息，请参考 https://github.com/rack/rack。

至此，products-service 服务的基本目录结构如下所示：

```
├── Gemfile
├── Gemfile.lock
├── api
│   └── api.rb
├── config.ru
└── lib
    └── init.rb
```

（7）执行如下命令，尝试启动该服务：

```
$> rackup
```

此时，应该可以从控制台看到服务启动的结果，类似如下所示：

```
[2015-07-09 17:04:10] INFO  WEBrick 1.3.1
[2015-07-09 17:04:10] INFO  ruby 2.0.0 (2013-02-24) [x86_64-darwin13.2.0]
[2015-07-09 17:04:10] INFO  WEBrick::HTTPServer#start: pid=97710 port=9292
```

默认情况下，Ruby 使用 WEBrick 作为 Web 服务器。接下来，如果读者打开浏览器，访问 http://localhost:9292，将得到如下结果，大功告成，如图 4-2 所示。

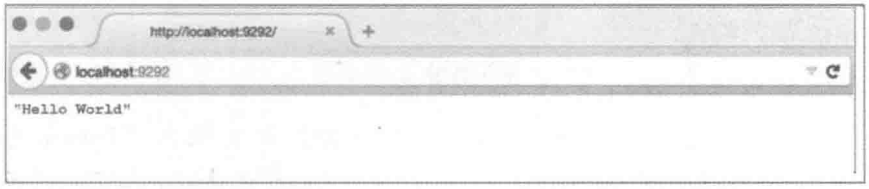


图 4-2　Hello World API 访问结果

到此，Hello World 的 API 构建成功。感兴趣的读者也可以尝试使用其他语言或者库来实现类似的功能。

4.2　代码测试与静态检查

4.2.1　代码测试

随着敏捷开发的实践逐渐深入人心，越来越多的开发者意识到自动化测试的重要性，可以采用 RSpec 作为代码测试工具。接下来，我们将为 products-service 配置 RSpec。

（1）定义 RSpec

进入项目的根目录，打开 Gemfile，添加如下内容：

```
group :development, :test do
  gem 'rspec'
  gem 'rspec-its'
  gem 'byebug'
end
```

（2）初始化 RSpec

执行如下命令安装并初始化 RSpec：

```
$> bundle install
$> rspec --init    # 初始化 RSpec
```

此时，会看到 RSpec 生成两个文件：

```
.rspec
spec/spec_helper.rb
```

注意，.rspec 是 RSpec 运行时首先加载的配置文件。

spec/spec_helper.rb 是 RSpec 运行时的参数配置文件。

（3）定义 RSpec 的 Rake 任务

Rake 是 Ruby 世界里的构建工具，类似 Java 世界中的 Ant 或者 Gradle。

通过定义 Rake 任务，能帮助我们简化 RSpec 的运行。

首先，创建 lib/tasks/spec.rake 文件，并添加如下内容：

```
require 'rspec/core/rake_task'     # 引用 RSpec 库
RSpec::Core::RakeTask.new(:spec)   # 使用 RSpec 的方式定义 Rake 任务
```

接下来，在项目的根目录下新建 Rakefile，并添加如下内容：

```
FileList['./lib/tasks/**/*.rake'].each{ |task| load task } #加载所有的 rake 文件
task default: [:spec]                  #将 spec 作为默认的 rake 任务
```

如上代码所示，Rakefile 将加载 spec.rake，并将 RSpec 的任务作为默认的 Rake 任务。

（4）执行 Rake 任务

之前的部分定义了 RSpec 的 Rake 任务，不过在执行该任务之前，需要先安装 rake 的 gem 包。

在 Gemfile 中加入如下内容：

```
gem 'rake'
```

并执行如下命令安装该 gem：

```
$> bundle install
```

接下来，就可以运行该 Rake 任务：

```
$> bundle exec rake spec
```

如果得到类似如下的结果，则表示当前 RSpec 配置成功。

```
Finished in 0.00034 seconds (files took 0.08908 seconds to load)
0 examples, 0 failures
```

同时，也可以使用 bundle exec rake -T 查看当前可用的 Rake 任务。譬如，当前 products-service 只有一个可执行的 Rake 任务：

```
$> rake spec                 # 运行 RSpec
```

至此，RSpec 就配置成功了。

（5）测试 API

之前的部分，我们实现了最简单的 Hello World API，虽然并没有什么业务逻辑，但存在实现代码，最好应有相应部分的测试。这样有利于提高团队对测试的重视度，并且随着代码的演进，测试也能不断演进，从而逐渐构建完善的测试集。

在 Hello World API 中，我们使用了 Rack 作为 Web 抽象接口。对于 Rack，Ruby 社区也提供了相应的测试工具 rack-test。下面，我们看看如何使用 rack-test 对当前的 API 进行测试。

A）修改 Gemfile，并添加如下代码：

```
gem 'rack-test'
```

B）修改 spec/spec_helper.rb，添加相关的引用：

```
require 'rspec'
require 'rack/test'
require_relative '../lib/init'
......
```

C）新建 api 的测试文件 spec/api_spec.rb，并添加如下代码：

```
require 'spec_helper'

describe API do
  include Rack::Test::Methods

  def app
    API
  end

  describe 'get' do
    before do
      get("/")
    end

    it 'should return Hello world' do
      expect(last_response.body).to eq("\"Hello world\"")
```

```
    end

    it 'should return json format' do
      expect(last_response.content_type).to eq "application/json"
    end
  end
end

def app
  API
end
```

在如上代码中，def app 部分是 rack-test 期望设置的入口类，对于当前 products-service 的例子而言，入口类对应的是类 API。

接下来的部分，使用 RSpec 的关键字 describe、before、it、expect 等定义了对 Hello World API 的测试。其中，describe 描述以 Get 方式发送请求，before 定义测试前的行为为访问根目录（和 XUnit 中的 Setup 类似），it 定义了测试的具体执行内容，expect 定义对测试结果的期望：当使用 Get 请求“/”时，期望返回的 HTTP body 为“Hello world”，同时返回的 Content-type 为 application/json。

测试内容写好了，我们可以使用之前定义的 Rake 任务运行测试：

```
$> bundle exec rake
```

由于默认的任务已经设置成 spec，因此直接运行如上命令即可。最后，期望的输出结果如下所示：

```
..
Finished in 0.07039 seconds (files took 0.53754 seconds to load) 2 examples, 0 failures
```

至此，products-service 的测试框架 Rspec 就配置好了。

4.2.2 测试覆盖率统计

单元测试覆盖率是评价单元测试完整性的重要度量标准之一。SimpleCov 是 Ruby 世界里统计代码测试覆盖率较方便的工具，使用非常简单。

接下来，我们就在当前的 products-service 中添加对测试覆盖率的统计。

（1）修改 Gemfile，在 group :development, :test do 下添加如下代码：

```
gem 'simplecov', require: false
```

（2）在 spec_helper.rb 的开头添加如下代码，引用 Simplecov：

```
require 'simplecov'
```

（3）在 spec_helper.rb 的 rspec 代码片段前添加如下代码，完成初始化：

```
SimpleCov.start
```

（4）在 spec_helper.rb 的最后添加如下代码，设置最小的测试覆盖率：

```
SimpleCov.minimum_coverage 100
```

（5）运行 Rake 任务，此时会看到在结果的最后，多了对测试覆盖率的输出，如下所示：

```
Coverage report generated for RSpec to /Work/microservice-in-action/products-
service/coverage. 9 / 9 LOC (100.0%) covered.
```

（6）按照如上输出的提示，打开目录/Work/microservice-in-action/products-service/coverage/index.html，结果如图 4-3 所示。

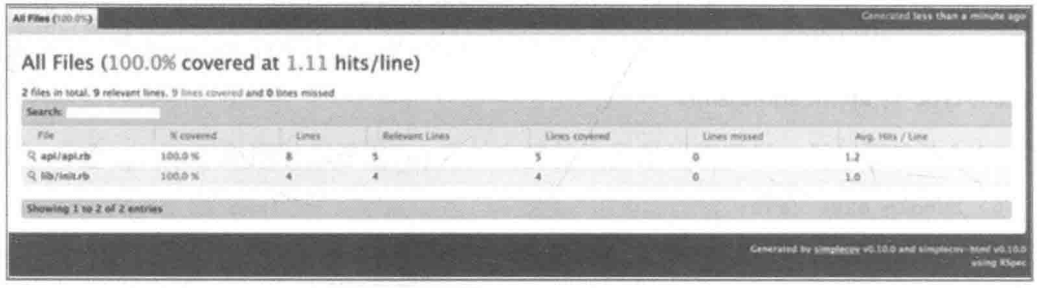


图 4-3　测试覆盖率统计结果

4.2.3　静态检查

文档、作为解释性的文字描述，很难和代码保持同步，因此代码的可读性和质量往往更能成为决定项目是否容易维护的关键要素之一。这就要求开发团队的代码具有良好的命名风格、统一的格式等。如果每次我们都依赖手动的方式检查这些标准是否达到，那势必会影响到交付的效率。

在笔者的实践中，大部分项目都使用 Rubocop 完成代码的静态检查。

首先，我们来看看 Rubocop 的具体配置过程。

（1）定义 Rubocop

打开 Gemfile，添加 rubocop：

```
group :development, :test do
  ......
  gem 'rubocop'
end
```

（2）配置 Rubocop

在根目录下，新建.rubocop.yml 文件：

```
$> cd product-service
$> touch .rubocop.yml
```

并添加如下内容到.rubocop.yml 中：

```
inherit_from: .rubocop_todo.yml

EmptyLinesAroundBlockBody:
  Exclude:
    - 'spec/**/*'

LineLength:
  Max: 120
  Exclude:
    - 'spec/**/*'
    - 'config/initializers/*'

MethodLength:
  Max: 20
  Exclude:
    - 'db/migrate/*'

Metrics/ClassLength:
  Max: 120

Metrics/PerceivedComplexity:
  Max: 10

WordArray:
  MinSize: 2
```

在上面的配置文件中，第一行中的.rubocop 是使用 rubocop --auto-gen-config 生成的，后面的配置规定了在 block 前后必须有空行，每行代码最长不能超过 120 个符号，方法不

能超过 20 行，类不能超过 120 行等。更多配置信息，请参考 rubocop 的官方文档（https://github.com/bbatsov/rubocop）。

（3）定义 Rake 任务

通常，我们会将静态检查和 Rake 任务集成。一方面，能通过 Rake 任务直接调用静态检查机制的运行；另一方面，也可以在 Rake 任务定义中，将多个任务组合在一起，譬如在运行测试的 Rake 任务前，进行静态检查。

首先进入 lib/tasks 目录，并新建 rubocop.rake 文件。

```
$> cd lib/tasks
$> touch rubocop.rake
```

打开 rubocop.rake 文件并添加如下内容：

```
namespace :quality do
  begin
    require 'rubocop/rake_task'

    RuboCop::RakeTask.new(:rubocop) do |task|
      task.patterns = %w{
        app/**/*.rb
        config/**/*.rb
        lib/**/*.rb
        spec/**/*.rb
      }
    end
  rescue LoadError
    warn "rubocop not available, rake task not provided."
  end
end
```

如上代码所示，task.patterns 中定义了筛选文件的策略以及哪些目录下的文件需要被执行静态检查。

接下来，在根目录下执行如下命令：

```
$> bundle exec rake -T
```

查看 rubocop 对应的 Rake 任务，如下所示：

```
$> rake quality:rubocop          # 运行 RuboCop
```

并使用如下命令执行该任务：

```
$> bundle exec rake quality:rubocop
```

如果得到类似如下的结果，则表示 Rubocop 配置成功。

```
Running RuboCop...
Inspecting 2 files
..
2 files inspected, no offenses detected
```

4.2.4 代码复杂度检查

在笔者的实践中，大部分项目都使用 Cane 完成代码的复杂度检查。

接下来，我们看看 Cane 的具体配置过程。

（1）定义 Cane

Cane 的定义和 Rubocop 非常类似。

首先，打开 Gemfile，添加 Cane 到 Gemfile 中：

```
group :development, :test do
  ......
  gem 'cane'
end
```

（2）定义 Rake 任务

和 Rubocop 配置类似，笔者通常也会将 Cane 和 Rake 任务集成，以便当使用 Rake 运行自动化测试的任务时，能同时进行代码的静态检查。

同定义 Rubocop 的任务类似，进入 lib/tasks 目录，并新建 cane.rake 文件。

```
$> cd lib/tasks
$> touch cane.rake
```

打开 cane.rake 文件并添加如下内容：

```
namespace :quality do
  begin
    require 'cane'
    require 'cane/rake_task'
  rescue LoadError
    warn "cane not available, cane task not provided."
```

```
  else
    desc "Run cane to check quality metrics"
    Cane::RakeTask.new(:cane) do |cane|
      cane.abc_max         = 12
      cane.no_doc          = true
      cane.style_glob      = './{app,lib}/**/*.rb'
      cane.style_measure   = 120
      cane.abc_exclude     = []
    end
  end
end
```

接下来，在根目录下执行如下命令：

```
$> bundle exec rake -T
```

查看 Cane 对应的 Rake 任务，如下所示：

```
rake quality:cane
```

接下来，使用如下命令执行该任务：

```
$> bundle exec rake quality:cane
```

如果没有任何错误提示，则表示 Cane 配置成功。

（3）Rake 任务集成

在上面的部分中，我们使用 Rake 分别定义了运行 Rubocop 以及 Cane 的任务。

在笔者的实践中，通常是将这两个任务合并，且定义成一个入口，如下所示。

定义 lib/tasks/quality.rake，并添加如下内容：

```
desc "Run cane quality checks"
task quality: %w(quality:cane quality:rubocop)
```

这时，使用 rake quality 就能运行 Rubocop 和 Cane 了。

（4）默认 Rake 任务设置

另外，还可以将 RSpec、Rubocop 以及 Cane 的运行统一设置成默认运行的 Rake 任务。

修改 Rakefile，代码如下所示：

```
FileList['./lib/tasks/**/*.rake'].each{ |task| load task }
task default: [:quality, :spec]
```

至此，RSpec、Rubocop 以及 Cane 就配置完毕了。

最后完成 RSpec、SimpleCov、Rubocop 以及 Cane 配置后的代码结构如下所示：

```
├── Gemfile
├── Gemfile.lock
├── Rakefile
├── api
│   └── api.rb
├── config.ru
├── lib
│   ├── init.rb
│   └── tasks
│       ├── cane.rake
│       ├── quality
│       │   ├── rubocop.rake
│       │   └── spec.rake
│       └── quality.rake
└── spec
    └── spec_helper.rb
```

通过使用 RSpec、Rubocop、Cane 等配置 products-service 的测试与静态检查机制，使用 SimpleCov 统计代码的测试覆盖率以及使用 Rake 定义相关任务，可简化代码测试、静态检查机制以及测试覆盖率的调用，为后续持续集成环境的配置提供了方便的接口。

第 5 章

构建 Docker 映像

Docker 是一个开源的 Linux 应用容器（Linux Container）引擎，允许开发者将应用及其依赖打包到一个可移植的映像中。开发完成之后，运维人员可以直接使用这个映像，将其发布到任何装有 Docker 的机器上。Docker 的出现，有效地解决了微服务架构下，服务粒度细、服务数量多所导致的开发环境搭建、部署以及运维成本高的问题。

利用 Docker 的容器化技术，能够实现在一个节点上运行成百甚至上千的 Docker 容器，每个容器都能独立运行一个服务，因此也就降低了随着微服务数量增多导致的节点数量增多带来的成本。

本章中，我们将使用 Docker 来完成 products-service 的映像构建。

5.1 定义 Dockerfile

1. 首先，在 products-service 的目录下新建 Dockerfile：

```
$> cd products-service
$> touch Dockerfile
```

2. 在 Dockerfile 中添加如下内容：

```
FROM ruby:2.1.5
MAINTAINER  docker-library <docker-library@github.com>

RUN apt-get update -y
```

```
WORKDIR /app

ADD Gemfile /app/Gemfile
ADD Gemfile.lock /app/Gemfile.lock
RUN bundle install --jobs 8

ADD . /app
EXPOSE 8080

CMD ["rackup","-o","0.0.0.0","-p","8080"]
```

Dockerfile详解

Dockerfile是一个文本文件，它包含了构建Docker映像所需要的指令。通过Dockerfile，Docker能够自动读取并解析其中的指令，并构建Docker映像包。

通常，Dockerfile分为如下4个部分：

- 基础映像（父映像）信息
- 维护者信息
- 映像操作命令
- 容器启动命令

在Dockerfile中，主要内容如下所示。

命令	描述
FROM <image>:<tag>	表示获取Docker基本映像。默认的，如果不明确指定映像URL，则是从Docker Hub上获取映像
MAINTAINER <name><email>	指定维护者的姓名和联系方式
RUN <command>	在Docker容器里运行Shell命令，等价于docker run <image> <command>
WORKDIR <path>	指定RUN、CMD、ENTRYPOINT等命令运行的工作路径
ADD <src> <dest>	添加应用的目录、文件到Docker容器中。<src>是相对于应用程序的相对文件路径，可以是文件，也可以是目录；<dest>是Docker容器内文件或者目录的绝对路径
EXPOSE <port>	Docker容器和Docker主机的端口映射关系
CMD <command>	Docker容器运行时的默认命令

从如上所示的 Dockerfile 可以看出，products-service 的 Docker 映像是一个在 ruby: 2.1.5

的基础映像之上构建的，其构建过程包括如下步骤：

（1）使用 apt-get 更新依赖包。

（2）创建/app 文件夹。

（3）使用 bundler 安装项目中用到的依赖 Gem。

（4）使用 rackup 启动服务并运行在 8080 端口。

通过如上步骤，Docker 便能根据 Dockerfile 构建出映像。

5.2 配置 Docker 主机

我们知道，Docker 使用了 Linux 容器的特性。对于 Linux 系统，Docker 可以直接安装并运行。但对于 Mac 系统而言，其本身并不支持 Linux 应用容器的特性。因此，如果想让 Docker 运行在 Mac 上，我们需要在 Mac 系统创建一个 Linux 的虚拟机作为 Docker 的主机。由于笔者的开发环境为 Mac 系统，因此需要找到合适的机制安装并运行 Docker。

Boot2docker 就是这样一个工具，它能帮助我们在 Mac 下轻松地搭建整个 Docker 运行的环境，其主要包括：

- 一个 VirtualBox 的虚拟机，也称 Docker 主机。
- 安装在 Docker 主机中的 Docker 环境。
- Docker 映像或者容器的管理工具。

为了使用 Boot2docker 创建 Docker 主机，我们需要完成如下几步。

（1）初始化 Docker 主机，代码如下所示：

```
$> boot2docker init
```

（2）配置本地系统与 Docker 主机之间的端口映射，代码如下所示：

```
$> vboxmanage modifyvm "boot2docker-vm" --natpf1
"docker,tcp,127.0.0.1,2376,,2376"
```

（3）启动 Docker 主机：

```
$> boot2docker up
```

（4）启动 Docker 主机后，需按照命令行的输出结果导出如下环境变量：

```
export DOCKER_HOST=tcp://192.168.59.104:2376
export DOCKER_CERT_PATH=/Users/leiwang/.boot2docker/certs/boot2docker-vm
export DOCKER_TLS_VERIFY=1
```

（5）验证 Docker 运行环境。

执行 docker images，显示当前可用的 Docker 映像。

```
$> docker images
```

譬如，如下结果显示了当前可用的 Docker 映像为 postgres:

```
REPOSITORY    TAG     IMAGE ID        CREATED          VIRTUAL SIZE
postgres      9.4     7bf0ec35adaf    4 weeks ago      214 MB
```

如果提示结果如下，则表明当前 Docker 主机并未成功运行：

```
Cannot connect to the Docker daemon. Is 'docker -d' running on this host?
```

类似的工具，还有 Docker-machine。更多关于 Docker-machine 的信息，请读者参考 Docker-machine 的官方文档 https://docs.docker.com/machine/。

5.3 构建 Docker 映像

之前我们定义了 Dockerfile，也使用 Boot2docker 配置了 Docker 主机。

接下来，我们将使用 docker build 构建 Docker 映像，如下所示：

```
cd products-service
docker build -t products-service .
```

执行完成后，查看 Docker 的可用映像列表，执行命令如下：

```
docker images | grep products-service
```

如果看到类似如下的结果，则表示 products-service 的 Docker 映像已经构建成功。

```
products-service latest  befa32e32f36  About an hour ago   809 MB
```

5.4 运行 Docker 容器

已经构建了 Docker 映像，我们可以尝试在 Docker 的环境中运行该映像：

```
docker run -p 8080:8080 products-service
```

如果得到类似如下的结果，则表示 products-service 在 Docker 的容器中已经成功运行。

```
[2015-07-14 03:31:28] INFO  WEBrick 1.3.1
[2015-07-14 03:31:28] INFO  ruby 2.1.5 (2014-11-13) [x86_64-linux]
[2015-07-14 03:31:28] INFO  WEBrick::HTTPServer#start: pid=1 port=8080
```

另外，也可以使用 docker ps 查看当前 products-service 所在容器的运行信息，结果如下所示：

```
CONTAINER-ID  IMAGE            COMMAND                  CREATED   STATUS   PORTS
eb6291344225 products-service "rackup -o 0.0.0.0..." 1m ago   Up 1m  8080->8080/tcp
```

从如上的结果可以看出，一个 Docker 容器正在运行，其 ID 是 eb6291344225，对应的映像是 products-service，容器启动后最后执行的命令是 rackup -o 0.0.0.0 -p 8080。同时，该容器的端口转发配置为 0.0.0.0:8080->8080/tcp。

到目前为止，products-service 所在的 Docker 容器已经成功运行，products-service 也已经启动。不过，在访问 products-service 前，我们必须先获取 Docker 主机的 IP 地址，获取的方式如下所示：

```
$> boot2docker ip
```

在当前例子中，笔者的 Docker 主机的 IP 地址为 192.168.59.103。

接下来，我们打开浏览器并访问如下地址 http://192.168.59.103:8080/，将得到之前 products-service 的运行结果。

```
http://192.168.59.103:8080
```

5.5 发布 Docker 映像

有了可以工作的 Docker 映像，本节我们将完成映像的发布。

- 发布到 Docker Hub

在当前的例子中，笔者使用 Docker Hub（https://registry.hub.docker.com/repos/）的自动构建机制，从 Github 上获取代码，构建 Docker 映像，并发布该映像。

首先，配置 Docker Hub 的工程，如图 5-1 所示。

图 5-1　配置 Docker Hub 工程

如图 5-1 所示，其显示了如何关联 Github 的相关代码库。

另外，从 Docker Hub 的工程中，也可以查看映像构建的状态和当前 Dockerfile 的内容，如图 5-2 所示。

图 5-2　查看 Docker 镜像的构建状态

关于如何配置 Docker Hub，感兴趣的读者可以参考官方文档 http://docs.docker.com/docker-hub/builds。

- **发布到私有的 Docker 仓库**

除了可以发布到 Docker Hub 上，也可以将 Docker 映像发布到内部的 Docker 仓库上。

在笔者的某些项目中，会使用类似如下的脚本，将 Docker 映像发布到内部的 Docker 仓库上。

```
#!/bin/bash

DOCKER_REGISTRY_URL=$DOCKER_REGISTRY_URL
DOCKER_REGISTRY_USER_NAME=$DOCKER_REGISTRY_USER_NAME
APP_NAME=$APP_NAME

BUILD_NUMBER=${BUILD_NUMBER:-dev}
VERSION=${MAJOR_VERSION}.$BUILD_NUMBER

FULL_TAG=$DOCKER_REGISTRY_URL/$DOCKER_REGISTRY_USER_NAME/$APP_NAME:$VERSION
FILE_NAME=$APP_NAME-$VERSION

echo "Building Docker image..."
docker build --tag $FULL_TAG .

if [ $DOCKER_REGISTRY_URL != "localhost" ]; then
  echo "Pushing Docker image to Registry..."
  docker push $FULL_TAG
fi
```

如上代码所示，首先设置 Docker 的私有仓库地址以及访问仓库的用户，接下来，设置构建号 BUILD_NUMBER、版本号 VERSION 等参数，组成 Docker 镜像的 FULL_TAG，然后使用 docker build 构建 Docker 映像，最后使用 docker push 将映像上传到指定的 Docker 仓库中。

- **发布到云存储**

除此之外，考虑到不同网络之间的隔离性、映像访问的安全性等因素，还可将 Docker 映像发布到公有云存储上。

譬如，如果生产环境中所有的应用部署都访问 Docker 仓库，则当某应用受到攻击时，有可能窃取或者破坏 Docker 仓库上的其他 Docker 映像。

在笔者参与的部分项目中，会将 Docker 映像导成 tar 包，存储到 AWS 的 S3 上，因为 S3 有更灵活、细粒度的权限控制能力。示例代码如下所示：

```
#!/bin/bash

APP_NAME=$APP_NAME

BUILD_NUMBER=${BUILD_NUMBER:-dev}
VERSION=${MAJOR_VERSION}.$BUILD_NUMBER

S3_BUCKET=$S3_BUCKET

FULL_TAG=$APP_NAME:$VERSION
FILE_NAME=$APP_NAME-$VERSION

mkdir -p target

echo "Saving Docker image to local file..."
docker save -o target/$FILE_NAME.tar $FULL_TAG

echo "Compressing local Docker image..."
gzip --force target/$FILE_NAME.tar

echo "Uploading docker image to S3..."
aws s3 mv target/$FILE_NAME.tar.gz s3://$S3_BUCKET/$APP_NAME/$FILE_NAME.tar.gz
```

如上代码所示，其和发布到企业私有 Docker 仓库的代码非常类似。这里是使用 docker save 生成 Docker 映像，并使用 aws s3 mv 将映像复制到具体的 Bucket 上。

关于Docker映像存储介质的比较

	私有Docker Registry	云存储	Docker Hub
发布操作	使用Docker push	使用Docker save	集成代码提交
映像类型	Docker映像	保存成文件系统的tar包	Docker映像
获取操作	使用Docker pull	使用Docker load	使用Docker pull
权限控制	访问权限控制不灵活	访问权限控制灵活	访问权限控制灵活
备份成本	备份成本高	云存储平台提供备份机制，备份成本低	备份成本低

5.6 小结

在本章中，我们使用 Docker 构建了 products-service 的映像，同时将该映像发布到 Docker Hub 上。感兴趣的读者也可以使用其他的映像发布机制代替，譬如私有 Docker 仓库或者云存储。

通过使用 Docker 映像，团队可以在任何运行 Docker 的环境中部署该映像，极大地降低了部署的成本。

第 6 章

部署 Docker 映像

在上一章中，我们介绍了如何基于 Docker 构建映像包，并将其发布到 Docker Hub 上。本章让我们来探索如何部署 Docker 映像。

6.1 基础设施 AWS

我们知道，Docker 是基于 Linux 容器的虚拟化技术，因此 Docker 的部署需要将 Docker 映像应用到具体的运行 Linux 操作系统的节点上。另外，除了运行服务的节点本身，还需要包括防火墙配置、路由配置、安全访问控制等，有些情况还需要自动化的扩容机制，这是生产环境中不可缺少的部分，我们也将它们统称为基础设施。

在本例中，我们采用亚马逊的 AWS Web Service 作为部署 Docker 的基础设施，并使用 AWS EC2 节点作为运行 Docker 映像的主机。

什么是Amazon EC2

Amazon EC2（Amazon Elastic Compute Cloud）是亚马逊AWS平台中为用户提供计算的资源，也称为Amazon EC2实例。使用Amazon EC2可以在很大程度上避免前期计划的大规模资源的投入，能使企业根据业务发展需要，动态地按需扩容实例。Amazon EC2主要提供以下功能：

- 构建亚马逊系统映像（AMI），其包括操作系统和运行服务需要的相关软件，从而能够快速启动实例。
- 按需选择实例的CPU、内存、存储等配置。
- 提供持久性存储卷Amazon EBS（Elastic Block Store）。
- 定义区域（Region）和可用区（Available Zone），允许在多个物理隔离的区域存储资源。
- 提供虚拟网络环境配置（Amazon VPC）、负载均衡器（Load Balancer）、自动扩容（Auto Scaling）等机制。

图6-1所示为EC2面板，更多关于AWS EC2的信息，请参考Amazon EC2官方文档（http://aws.amazon.com/cn/documentation/ec2/）。

图6-1　AWS EC2控制面板

对于当前的 products-service 例子而言，基础设施主要包括：

- 虚拟私有云（Virtual Private Cloud）
- 安全组（Security Group）
- 计算实例（Elastic Compute Cloud）
- 自动扩容机制（Auto scaling）
- DNS 解析（Route 53）

6.2 基础设施自动化

基础设施自动化（Infrastructure As Code）最大的优势在于将传统认为最耗时、最容易出错的环境安装、配置等过程采用代码的方式自动化起来，而且业界已经有了一些被广泛使用的基础设施自动化工具，譬如 Chef、Puppet、Ansible 以及 Salt 等。关于这些工具的比较，感兴趣的读者可以参考相关文章，笔者在这里不再赘述。

除了如上所述的基础设施的自动化工具之外，某些云平台厂商也会提供配套的工具来帮助用户完成基础设施的自动化构建。譬如 AWS Web Service 提供的 CloudFormation（http://aws.amazon.com/cloudformation/）、Netflix 提供的 Asgard 等，都能够帮助用户在 AWS Web Service 上高效创建基础设施。

在本案例中，我们将采用亚马逊的 AWS CloudFormation 作为部署和运行 Docker 的基础设施，并使用 AWS EC2 节点运行 Docker 容器。

为使用 CloudFormation，需创建相关的基础设施，主要包括：

- AWS EC2 节点的创建
- AWS Security Group 的创建
- AWS DNS 解析的配置
- AWS Autoscaling Group 自动扩容机制的创建

如下为 CloudFormation 的配置文件：

```
{
   "Resources":{
     "instancesSecurityGroup":{
       "Type":"AWS::EC2::SecurityGroup",
       "Properties":{
         "GroupDescription":"Security Group",
         "VpcId":"vpc-xxxxxx",
         "SecurityGroupIngress":[
           {
             "IpProtocol":"tcp",
             "FromPort":22,
             "ToPort":22,
             "CidrIp":"0.0.0.0/0"
           },
```

```
            {
              "IpProtocol":"tcp",
              "FromPort":80,
              "ToPort":80,
              "CidrIp":"0.0.0.0/0"
            }
          ],
          "Tags":[
            {
              "Key":"Name",
              "Value":"products-service"
            }
          ]
        }
      },
      "loadBalancerSecurityGroup":{
        "Type":"AWS::EC2::SecurityGroup",
        "Properties":{
          "GroupDescription":"Security Group",
          "VpcId":"vpc-xxxxxx",
          "SecurityGroupIngress":[
            {
              "IpProtocol":"tcp",
              "FromPort":80,
              "ToPort":80,
              "CidrIp":"0.0.0.0/0"
            },
          ],
          "Tags":[
            {
              "Key":"Name",
              "Value":"products-service"
            }
          ]
        }
      },
      "dnsRecord":{
        "Type":"AWS::Route53::RecordSet",
        "Properties":{
```

```
      "Comment":"Public Record",
      "HostedZoneName":"products-service.test.microservice-in-action.com",
      "Name":"products-service.test.microservice-in-action.com",
      "Type":"A",
      "AliasTarget":{
        "DNSName":{
          "Fn::GetAtt":[
            "loadBalancer",
            "DNSName"
          ]
        },
        "HostedZoneId":{
          "Fn::GetAtt":[
            "loadBalancer",
            "CanonicalHostedZoneNameID"
          ]
        }
      }
    }
  },
  "loadBalancerSecurityGroup":{
    "Type":"AWS::EC2::SecurityGroup",
    "Properties":{
      "GroupDescription":"Security Group",
      "VpcId":"vpc-xxxxxx",
      "SecurityGroupIngress":[
        {
          "IpProtocol":"tcp",
          "FromPort":80,
          "ToPort":80,
          "CidrIp":"0.0.0.0/0"
        }
      ],
      "Tags":[
        {
          "Key":"Name",
          "Value":"products-service"
        }
      ]
```

```
    }
  },
  "loadBalancer":{
    "Type":"AWS::ElasticLoadBalancing::LoadBalancer",
    "Properties":{
      "Scheme":"internal",
      "Subnets":[
        "subnet-xxxxxx",
        "subnet-xxxxxx"
      ],
      "SecurityGroups":[
        {
          "Ref":"loadBalancerSecurityGroup"
        }
      ],
      "Listeners":[
        {
          "Protocol":"HTTP",
          "LoadBalancerPort":80,
          "InstancePort":80
        }
      ],
      "HealthCheck":{
        "Target":"HTTP:80/diagnostic/status/heartbeat",
        "HealthyThreshold":2,
        "UnhealthyThreshold":4,
        "Interval":10,
        "Timeout":8
      },
      "CrossZone":true,
      "ConnectionDrainingPolicy":{
        "Enabled":true,
        "Timeout":30
      },
      "Tags":[
        {
          "Key":"Name",
          "Value":"products-service"
        }
```

```
      ]
    }
  },
  "dnsRecordservices":{
    "Type":"AWS::Route53::RecordSet",
    "Properties":{
      "Comment":"Public Record",
      "HostedZoneName":"products-service.test.microservice-in-analysis.com",
      "Name":"products-service.test.microservice-in-analysis.com",
      "Type":"A",
      "AliasTarget":{
        "DNSName":{
          "Fn::GetAtt":[
            "loadBalancer",
            "DNSName"
          ]
        },
        "HostedZoneId":{
          "Fn::GetAtt":[
            "loadBalancer",
            "CanonicalHostedZoneNameID"
          ]
        }
      }
    }
  },
  "launchConfiguration-xxxxxx":{
    "Type":"AWS::AutoScaling::LaunchConfiguration",
    "Properties":{
      "IamInstanceProfile":{
        "Ref":"iamInstanceProfile"
      },
      "ImageId":"ami-xxxxxx",
      "InstanceType":"t2.medium",
      "InstanceMonitoring":true,
      "SecurityGroups":[
        {
          "Ref":"instancesSecurityGroup"
        }
```

```
        ],
      }
    },
    "autoScalingGroup-xxxxxx":{
      "CreationPolicy":{
        "ResourceSignal":{
          "Count":1,
          "Timeout":"PT5M"
        }
      },
      "Type":"AWS::AutoScaling::AutoScalingGroup",
      "Properties":{
        "AvailabilityZones":[
          "ap-southeast-2b",
          "ap-southeast-2a"
        ],
        "Cooldown":"120",
        "DesiredCapacity":"1",
        "HealthCheckType":"ELB",
        "LaunchConfigurationName":{
          "Ref":"launchConfiguration-xxxxxx"
        },
        "LoadBalancerNames":[
          {
            "Ref":"loadBalancer"
          }
        ],
        "MaxSize":"1",
        "MinSize":"1",
        "Tags":[
          {
            "Key":"Name",
            "Value":"products-service",
            "PropagateAtLaunch":true
          },
          {
            "Key":"application",
            "Value":"products-service",
            "PropagateAtLaunch":true
```

```
            }
          ],
          "VPCZoneIdentifier":[
            "subnet-xxxxxx",
            "subnet-xxxxxx"
          ]
        }
      },
      "scheduledAction-xxxxxx":{
        "Type":"AWS::AutoScaling::ScheduledAction",
        "Properties":{
          "AutoScalingGroupName":{
            "Ref":"autoScalingGroup-xxxxxx"
          },
          "DesiredCapacity":0,
          "MaxSize":0,
          "MinSize":0,
          "Recurrence":"0 11 * * *"
        }
      },
    },
    "Outputs":{
      "iamRoleArn":{
        "Value":"products-service-iam-role"
      },
      "DNSName":{
        "Value":{
          "Ref":"dnsRecord"
        }
      },
      "loadBalancerAddress":{
        "Value":{
          "Fn::GetAtt":[
            "loadBalancer",
            "DNSName"
          ]
        }
      }
    }
}
```

6.3 部署 Docker 映像

实际上，对于存在的 Docker 映像，之前我们已经演示了如何基于映像来启动一个 Docker 容器，类似执行如下命令：

```
$> docker run products-service
```

对于 products-service 而言，我们已经将 Docker 映像发布到 Docker Hub。因此，可以使用如下脚本 docker-deploy.sh 完成 Docker 映像的部署。

```
#! /bin/bash
set -e

DOCKER_REGISTRY_USER_NAME="wldandan"
APP_NAME="products-service"
APP_VERSION="latest"

FULL_TAG=$DOCKER_REGISTRY_USER_NAME/$APP_NAME:$APP_VERSION

echo "Pulling Docker image from Registry"
docker pull $FULL_TAG

echo "Launching Dokcer Container"
do
```

如上代码所示，该脚本主要包括三个部分：

- 准备 Docker 映像信息
- 获取 Docker 映像
- 启动 Docker 容器

首先，通过环境变量准备 Docker 映像的标签信息，包括 Docker 的仓库地址、映像名称、映像版本号等；然后，根据指定映像的标签信息，从 Docker Hub 上下载对应的 Docker 映像；最后，使用 docker run -d -p 80:8080 在当前的 Docker 环境中启动 Docker 容器。

请注意，命令行中的-p 80:8080 指的是将 Docker 主机的 80 端口映射到 Docker 容器的 8080 端口，这样，我们就可以直接访问 Docker 主机的 80 端口访问容器提供的服务了。

6.4 自动化部署

有了基础设施，有了 Docker 映像的构建和部署，接下来，我们看看如何将二者结合起来，完成自动化的部署。

1. 首先，需要实现自动化部署脚本 deploy/deploy.sh，如下所示。

```
#!/bin/bash
set -e

[[ -z "$1" ]] && echo "Usage: Please Specify deployment file !!!" && exit 1

# 根据环境解析配置文件
echo  "Parsing config file $1 ..."

# 使用 CloudFormation 或者其他机制创建基础设施
echo "Creating resources ..."

# 在节点中获取 Docker 映像
echo "Pulling docker image ..."

# 在节点中运行 Docker 容器
echo "Runing docker container ..."
```

如上代码所示，deploy.sh 文件读取配置文件，根据环境相关的配置，在不同的环境中（譬如生产环境、测试环境）完成资源的创建和更新，包括新建 EC2 节点、更新 Load Balancer 配置、更新 Auto Scaling 配置等。由于这部分资源的创建和更新完全依赖 AWS，为了让读者能聚焦于部署的流程本身，笔者忽略了该部分的实现。感兴趣的读者可以参考 AWS 的相关文档或者书籍。

然后，在创建的 EC2 节点上执行 docker-deploy.sh 的相关内容，获取 Docker 映像并运行 Docker 容器。

为了实现当 EC2 节点启动后，能自动执行 Docker 的映像获取以及部署，docker-deploy.sh 的代码将被嵌入到 AWSCloudFormation 的 User Data 中。

2. 定义生产环境配置文件 deploy/deploy-prod.yml。

在步骤 1 中，我们定义的部署脚本 deploy.sh，会根据配置文件的不同，在不同的环境中执行资源的创建和更新。接下来，我们就定义生产环境的相关配置，如下所示：

```
app:
  name: products-service

docker:
  image: wldandan/products-service:latest

aws:
  vpc: vpc-xxx
  region: xxx
  subnets:
    - subnet-xxx-A
    - subnet-xxx-B
  load_balancers:
     name: products-service-xxx
     dns:  products-service-elb.xxx.amazonaws.com
  instances:
    type: t2.micro
    min: 1
    max: 2

splunk:
  host: splunk-prod-xxx.microservice-in-action.com
  index: products-service

nagios:
  host: nagios-xxx.microservice-in-action.com
```

如上所示，配置文件的内容主要包括：

- app 部分定义了 products-service 的信息。
- docker 部分定义了 Docker 映像的标签信息。
- aws 部分定义了部署的 VPC、Region、Subnets、Load Balancer 以及 EC2 节点等信息。

- splunk 部分定义了 splunk 的相关信息。
- nagios 部分定义了 nagios 的相关信息。

3. 定义测试环境配置文件 deploy/deploy-test.yml。

类似的，我们也需要定义测试环境的相关配置，如下所示：

```
app:
  name: products-service

docker:
  image: wldandan/products-service:latest

aws:
  vpc: vpc-xxx
  region: xxx
  subnets:
    - subnet-xxx-A
    - subnet-xxx-B
  load_balancers:
     name: products-service-xxx
     dns:  products-service-elb.xxx.amazonaws.com
  instances:
    type: t2.micro
    key_pair: products-service
    min: 1
    max: 2

splunk:
  host: splunk-prod-xxx.microservice-in-action.com
  index: products-service

nagios:
  host: nagios-prod-xxx.microservice-in-action.com
```

如上代码所示，部署测试环境的配置文件 deploy-test.yml 和部署生产环境的配置文件 deploy-prod.yml 结构一致，但具体的配置参数略有差异。原因是，在通常情况下，为了保

持安全性和隔离性，生产环境和测试环境是运行在两个独立的 AWS 账号中的。

通过不同环境的配置文件，完成基础设施的自动化配置，然后通过在节点启动时自动加载并运行 Docker 映像，完成 Docker 映像的自动化部署和运行。

6.5 小结

在本章中，我们首先使用 AWS CloudFormation 构建了 products-service 在 AWS 上的基础设施。接下来，通过构建 Docker 映像以及部署脚本，完成了 products-service 映像的自动化部署。

第 7 章

持续交付流水线

到目前为止，我们已经完成了 products-service 的如下部分：

- Hello World API
- 测试与静态检查
- 构建 Docker 映像
- 部署 Docker 映像

接下来，让我们看看如何为 products-service 构建持续交付流水线。

7.1 持续集成环境

在当前例子中，笔者将 products-service 的代码托管在 Github 上，因此可以选用 Snap-CI（https://snap-ci.com）作为持续交付工具。Snap-CI 可以无缝集成 Github 上的代码库，其配置步骤如下所示。

1. 首先选择 Github organisation，当前例子为 Micro-Service-In-Analysis。

2. Snap-CI 会自动加载可用的代码库，选择 products-service。

3. 单击+ Add，稍等片刻，Snap-CI 的控制面板中会出现 products-service。

配置结果如果 7-1 和图 7-2 所示。

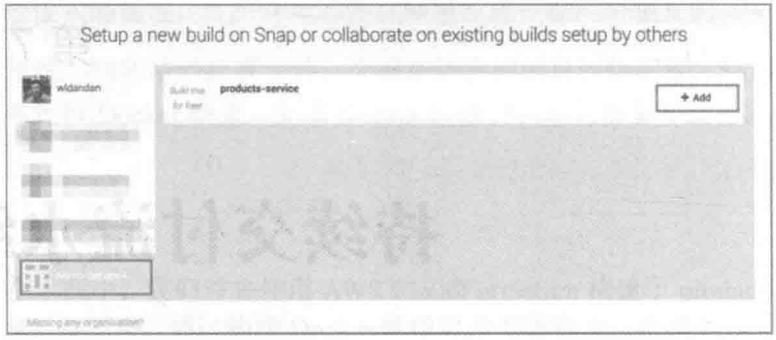


图 7-1　Snap-CI 加载 Github 工程 1

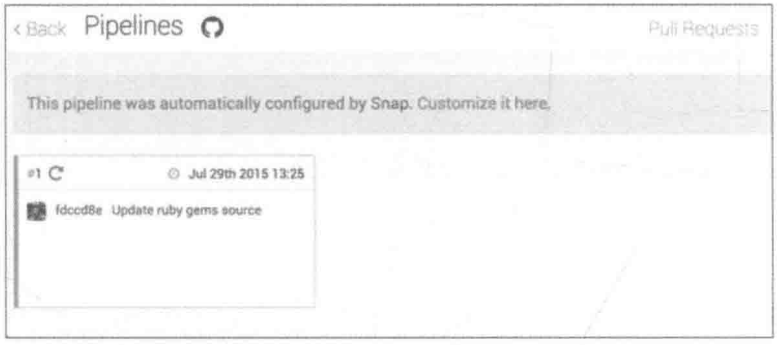


图 7-2　Snap-CI 加载 Github 工程 2

读者也可以根据喜好，选择其他的持续集成服务器，如 Bamboo、Jenkins、GO 以及在线的 Travis-CI、CircleCI 等。

Snap-CI

Snap-CI是一款在线的持续集成工具，由ThoughtWorks公司开发。它可以用于Java、Ruby、Python、JavaScript以及Node.js项目的持续集成构建，同时也支持Capybara、Selenium、PhantomJS等工具的测试，并能将构建包方便地部署到Heroku或者AWS上。

相比其他的持续集成工具，Snap-CI的优势在于：

- 可灵活地按需定义Stage（阶段）
- 对Github的项目支持友好
- 可托管的SAAS平台
- 免费

在笔者参与的项目中，持续交付流水线通常包括 4 个阶段，每个阶段都包括一个或者多个可执行的任务：

- 提交阶段
- 验证阶段
- 发布阶段
- 部署阶段

接下来，就让我们看看如何定义不同阶段执行的任务。

7.2 提交阶段

提交阶段主要检测代码库的变化，并按照配置触发相应的处理。通常，处理包括但不限于：

- 代码编译
- 静态检查
- 运行单元测试

大多数的持续集成服务器，都能通过 WebHook 的方式，检测到代码库的提交事件，并触发相应的处理机制。除此之外，持续集成服务器通常也支持轮询机制，能够定时检测代码库的变化并触发相应的处理机制。

Snap-CI 默认采用轮询的方式定期检测代码库的变化，因此我们不需要做任何设置，即能够自动捕获 products-service 代码库的变化。

我们在 Snap-CI 中定义了一个新的阶段，名称为 COMMIT，触发的方式为 Automatic，并设置当 Snap-CI 检测到代码变化时，执行如下命令：

```
$> bundle install
$> bundle exec rake quality
$> bundle exec rake spec
```

该配置如图 7-3 所示。

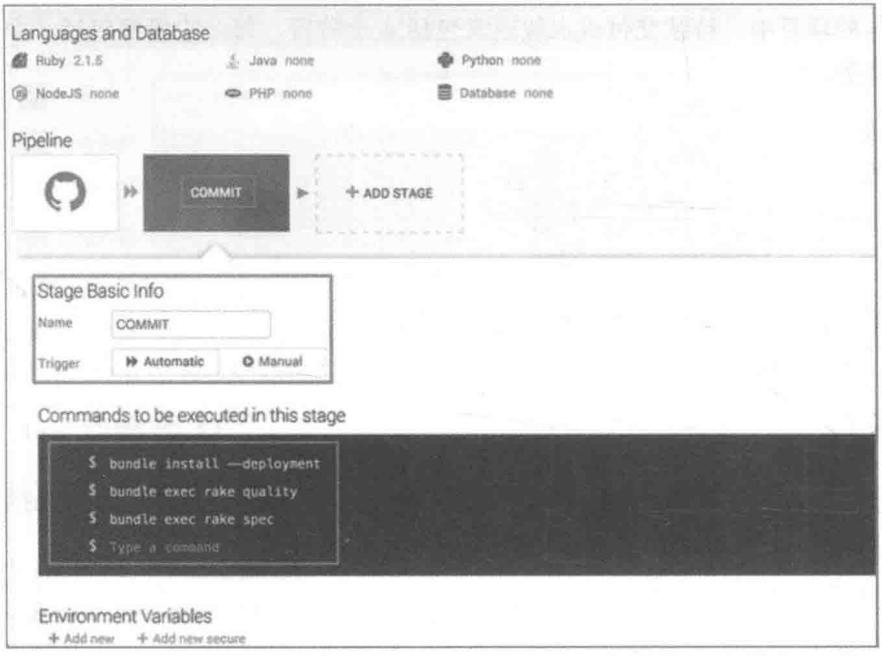


图 7-3　提交阶段配置

注意，如上的 Rake 执行命令来自 4.2 节中，其定义了运行 RSpec 测试和代码静态检查的 Rake 任务。

到此，提交阶段的处理就配置完成了。当开发团队提交代码到 products-service 代码库时，就可以看到 Snap-CI 被触发并开始运行相关的任务了，如下所示：

```
$ bundle install
Fetching gem metadata from https://rubygems.org/.........
Fetching version metadata from https://rubygems.org/..
Installing rake 10.4.2
Installing json 1.8.3 with native extensions
Installing i18n 0.7.0
Installing minitest 5.7.0
Installing thread_safe 0.3.5
Installing ast 2.0.0
Installing ice_nine 0.11.1
Installing builder 3.2.2
Installing columnize 0.9.0
Installing parallel 1.6.1
```

```
Installing diff-lcs 1.2.5
Installing equalizer 0.0.11
Installing hashie 3.4.2
Installing multi_json 1.11.2
Installing multi_xml 0.5.5
Installing rack 1.6.4
Installing powerpack 0.1.1
Installing rainbow 2.0.0
Installing rspec-support 3.3.0
Installing ruby-progressbar 1.7.5
Using bundler 1.10.6
Installing tzinfo 1.2.2
Installing descendants_tracker 0.0.4
Installing parser 2.2.2.6
Installing cane 2.6.2
Installing byebug 5.0.0 with native extensions
Installing rspec-core 3.3.2
Installing rspec-expectations 3.3.1
Installing rspec-mocks 3.3.2
Installing rack-accept 0.4.5
Installing rack-mount 0.8.3
Installing rack-test 0.6.3
Installing axiom-types 0.1.1
Installing coercible 1.0.0
Installing astrolabe 1.3.1
Installing rspec-its 1.2.0
Installing rspec 3.3.0
Installing virtus 1.0.5
Installing rubocop 0.32.1
Installing activesupport 4.2.3
Installing grape 0.12.0
Updating files in vendor/cache
Bundle complete! 8 Gemfile dependencies, 41 gems now installed.
Bundled gems are installed into /var/go/.bundled-gems.
Command bundle install exited successfully with status 0. Took 7.97 seconds.

$ bundle exec rake quality
warning: parser/current is loading parser/ruby21, which recognizes
warning: 2.1.6-compliant syntax, but you are running 2.1.5.
```

```
warning: please see https://github.com/whitequark/parser#compatibility-with-ruby-mri.
Running RuboCop...
Inspecting 3 files
...

3 files inspected, no offenses detected
Command bundle exec rake quality exited successfully with status 0. Took 1.47 seconds.

$ bundle exec rake spec
/opt/local/rbenv/versions/2.1.5/bin/ruby -I/var/go/.bundled-gems/gems/rspec-
core-3.3.2/lib:/var/go/.bundled-gems/gems/rspec-support-3.3.0/lib /var/go/.bundled-
gems/gems/rspec-core-3.3.2/exe/rspec --pattern spec/\*\*\{,/\*/\*\*\}/\*_spec.rb
..

Finished in 0.059 seconds (files took 0.34716 seconds to load)
2 examples, 0 failures

Command bundle exec rake spec exited successfully with status 0. Took 1.41 seconds.

Saving console log for later
```

运行后的结果如图 7-4 所示。

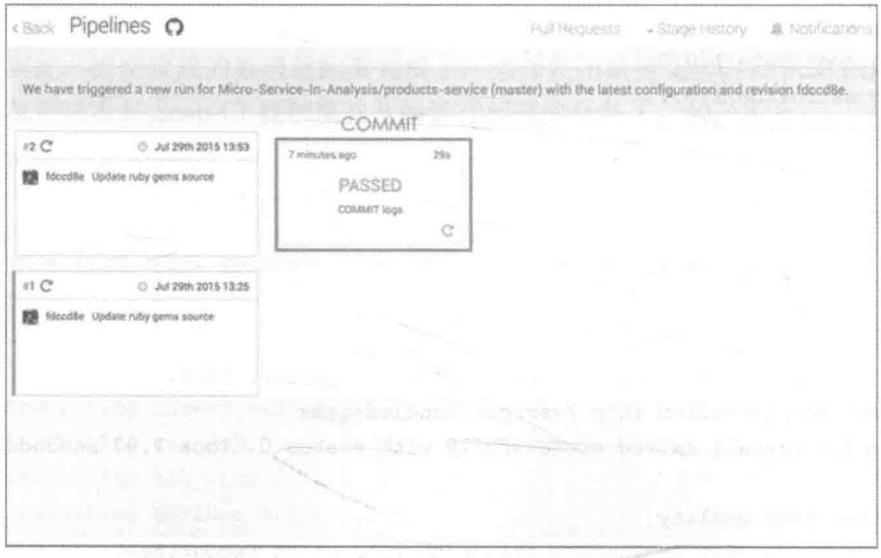


图 7-4　提交阶段运行结果

7.3 验证阶段

验证阶段主要进行功能、性能等的验证。通常，验证阶段包括但不限于如下任务：

- 运行集成测试
- 运行用户行为测试
- 运行组件测试
- 运行性能测试

注意，并不是所有的项目都需要同时定义这些任务，读者可以根据具体的项目情况，定义合适的任务。

实际上，在笔者的项目实践中，每个服务所承担的业务功能高内聚、低耦合而且代码量较少，其验证方式主要以单元测试以及消费者驱动的接口测试为主，通常都将它们放在提交阶段完成了。因此，对于目前的 products-service 例子，我们并不需要配置验证阶段的任务。

7.4 构建阶段

构建阶段的主要任务是构建部署包。

在之前的章节中，我们已经尝试了使用 Docker Hub 自动构建 Docker 映像并存储，同时讨论了组织或者团队也可以根据不同需求，构建私有的 Docker 仓库，或者将构建的 Docker 映像上传到云存储（譬如亚马逊的 AWS S3）上。在当前 products-service 的例子中，为了使读者聚焦于持续交付流程的搭建，而非各种 Docker 映像存储方式的比较，笔者倾向于使用 Docker Hub。

另外，对于 products-service 的持续交付流水线而言，仅当提交阶段的任务运行成功后，才会触发 Docker 映像的构建。因此，我们需要在 Docker Hub 中修改映像构建的触发策略，从之前的代码提交默认触发改成外部事件触发，具体配置流程如图 7-5 和图 7-6 所示。

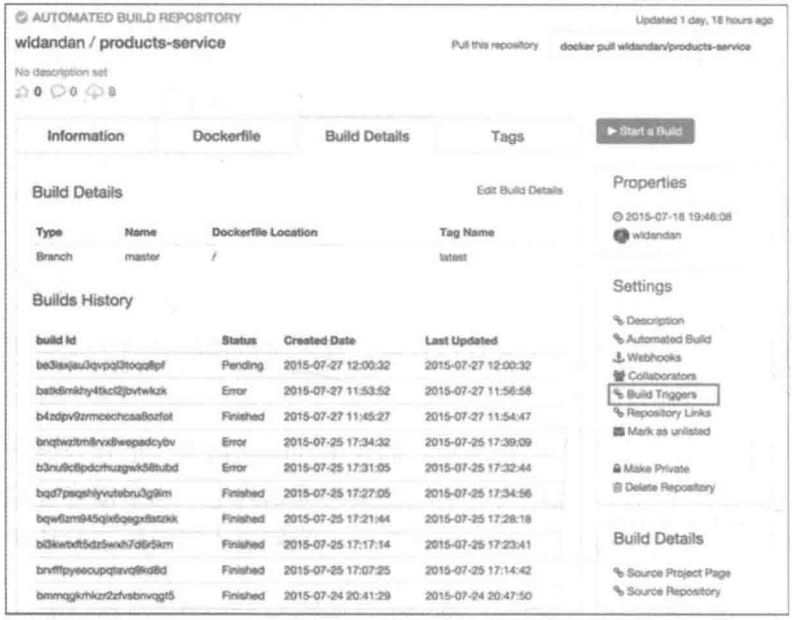


图 7-5　修改 Docker Hub 构建触发策略

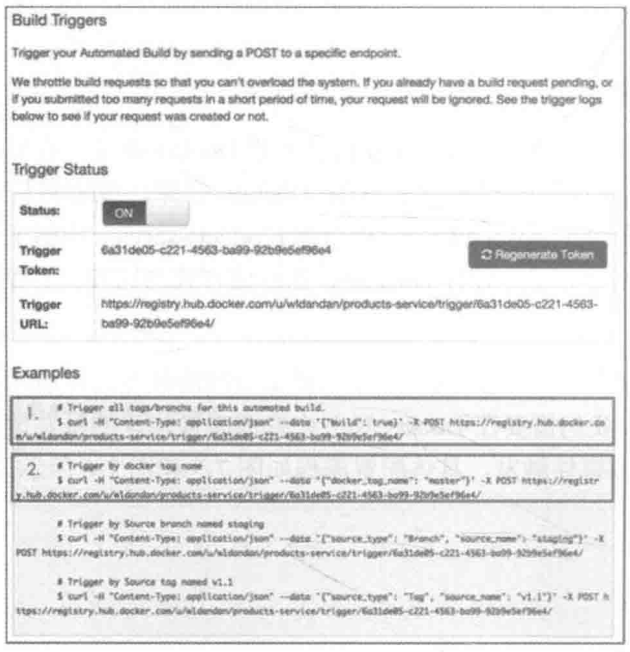


图 7-6　使用 POST 方式显示触发构建

如图 7-6 所示，通过配置 Build Triggers，我们关闭了 Docker Hub 的自动映像构建的功能。同时，只允许通过调用者显式地发送 POST 请求后，才触发 Docker 映像的构建。更多关于如何配置 Docker Hub 事件触发的机制，请参考其官方文档 Remote Build Triggers 部分（https://docs.docker.com/docker-hub/builds/）。

接下来，我们在 Snap-CI 中定义构建阶段的任务，单击“+ ADD STAGE”后，配置构建阶段的参数，如图 7-7 所示。

图 7-7　定义新的阶段 PACKAGE

如图 7-7 所示，我们定义一个新的阶段，名称为 PACKAGE，然后定义其执行的命令为：

```
curl -H "Content-Type: application/json" --data '{"docker_tag_name": "master"}' -X
POST https://registry.hub.docker.com/u/wldandan/products-service/trigger/
6a31de05- c221-4563-ba99-92b9e5ef96e4/
```

通过如上的配置，就意味着仅当提交阶段（COMMIT）的任务运行成功后，构建阶段（PACKAGE）的任务才开始运行，并当执行如上代码，向 Docker Hub 发送 POST 请求时，才触发 Docker Hub 开始构建 Docker 映像，如图 7-8 所示。

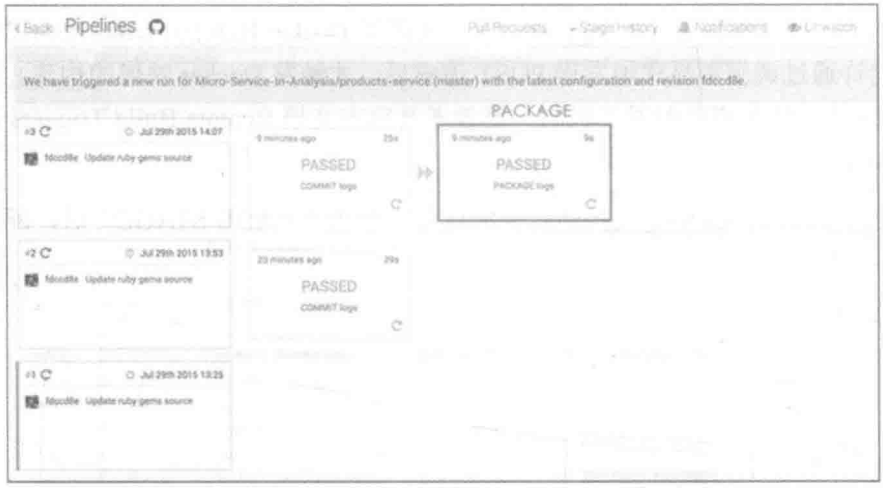


图 7-8　PACKAGE 阶段运行成功

语义化版本

到目前为止，构建的Docker映像一直都是使用latest作为版本标签。

而实际上，对于每次构建的映像以及服务本身，都应该使用语义化的策略定义版本。

语义化版本（Semantic Versioning）是指对于一个给定的版本号，将其定义成MAJOR.MINOR.PATCH（主、次、补丁），其变化的规律通常指：

- MAJOR version（主版本）会在API发生不可向下兼容的改变时增大。
- MINOR version（次版本）会在有向下兼容的新功能加入时增大。
- PATCH version（补丁版本）会在有向下兼容的缺陷被修复时增大。

实际上，这套版本定义规则基于但不仅限用于广泛存在的开源软件的实践。因为版本号及其改变的规则，传递了代码背后的含义，并解释了对于应用本身而言，每个相邻版本之间的变化规律。更多关于语义化版本的细节，请参考相关文档（https://en.wikipedia.org/wiki/Software_versioning）。

7.5　发布阶段

发布阶段主要将部署包发布到具体的环境中。

在笔者的实践中，所谓具体的环境，通常包括三类：

- 测试环境
- 类生产环境
- 生产环境

三类环境的触发条件和使用目的如表 7-1 所示。

表 7-1　不同发布环境的比较

	测试环境	类生产环境	生产环境
触发方式	自动	手动	手动
数据来源	模拟数据	真实数据	真实数据
使用目的	主要用于开发团队内部验证功能为主	基于生产环境的真实数据为业务部门进行演示	为用户提供真实的服务

企业或者组织也可以根据具体的需求，定义其他环境。对于微服务而言，其功能集中并且代码库较小，所以类生产环境和测试环境通常合二为一。在当前的 products-service 例子中，我们只定义两类环境：测试环境和生产环境。

在第 6 章中，我们已经定义了部署脚本和不同环境的配置参数，所以只需要调用如下命令部署测试环境或者生产环境：

```
$> deploy/deploy.sh deploy-test.yml   #部署测试环境
$> deploy/deploy.sh deploy-prod.yml   #部署生产环境
```

有了如上测试环境和生产环境的定义，有了部署脚本 deploy.sh，我们可以抽象出一个更简洁的不同环境的部署脚本 ci-deploy.sh，简化在不同环境下的部署，其内容如下所示：

```
#! /bin/bash
set -e

[[ -z "$1" ]] && echo "Usage: Please Specify environment !!!" && exit 1
./deploy/deploy.sh "deploy/deploy-$1.yml"
```

如上代码所示，该脚本接受传入的参数作为环境名称，执行过程中获取相关的配置信息，然后调用部署脚本 deploy.sh 完成部署操作。通过该脚本，在持续交付环境的配置中，只需要传递部署的环境 prod 或者 test，作为参数即可。

接下来，我们在 Snap-CI 中定义一个新的阶段，名称为 DEPLOY TO TEST，然后定义

其执行的命令：

```
$> ./ci-deploy.sh test  #部署测试环境
```

类似的，我们也可以使用这种方式定义生产环境的部署。

定义一个新的 STAGE，名称为 DEPLOY TO PROD，将其设置为手动触发，然后定义其执行的命令为:

```
$> ./ci-deploy.sh prod  #部署生产环境
```

配置完的结果如图 7-9 所示。

图 7-9　部署测试环境与生产环境

7.6　小结

本章主要讨论了基于 Snap-CI 构建持续交付流水线的过程。对于微服务架构而言，每个服务都是独立开发、独立测试、独立部署的个体，因此尽早建立持续交付流水线，能够帮助团队持续地、频繁地对服务进行改进。这也是笔者在每个服务的开发实践中，首先要做的事情。另外，本章虽然是使用 GitHub、Docker Hub 以及 Snap-CI 完成代码的托管、Docker 映像的存储以及持续集成服务器，但感兴趣的读者可以选择其他工具，构建适合团队的持续交付流水线。

第 8 章

日志聚合

对任何一个系统而言，日志都是至关重要的。通过日志，系统管理员可以查看系统的运行状况，开发人员可以快速定位问题、分析问题。

通常情况下，日志会以某种固定格式输出，并保存到本地文件系统中。当运行服务或者应用的节点数量不多时，查看日志非常简单。只需使用 SSH 登录到相应的节点，使用 grep、awk 或者 sed 等工具，就能定位出错的位置。但是，随着系统不断变复杂，尤其是当系统中的节点增加到成百上千，管理和访问这些节点的日志会变得非常复杂。

由于微服务架构本质上是基于分布式系统之上的应用架构方式，随着服务个数的增多，运行节点的数量必然增多。因此，这种情况下，登录单个节点查看日志、分析日志的工作将会耗费越来越多的成本，如果没有合适的工具，从上百个节点上的上百个日志文件中搜索出错日志会变得很困难，同时也意味着定位问题、发现问题的成本将随着节点数量的增多呈指数增加。

本章，我们将探索微服务的日志聚合。

8.1 日志聚合工具简介

在笔者的项目实践中，日志聚合工具主要以 Splunk 和 LogStash 为主。下面，我们就来看看这两款日志聚合工具。

- Splunk

Splunk（http://www.splunk.com/）是一款功能强大的日志管理工具，它可以用多种方式来添加日志，生成图形化报表，如图 8-1 所示。同时，它最强大的功能是搜索，被誉为“Google for IT”。Splunk 有免费版和收费版，最主要的差别在于每天的索引容量大小（索引是搜索功能的基础），免费版每天索引量最大为 500MB。Splunk 的主要功能包括：

- 日志聚合
- 日志搜索
- 语义提取
- 对结果进行分组、联合、拆分和格式化
- 强大的可视化功能
- 电子邮件提醒功能

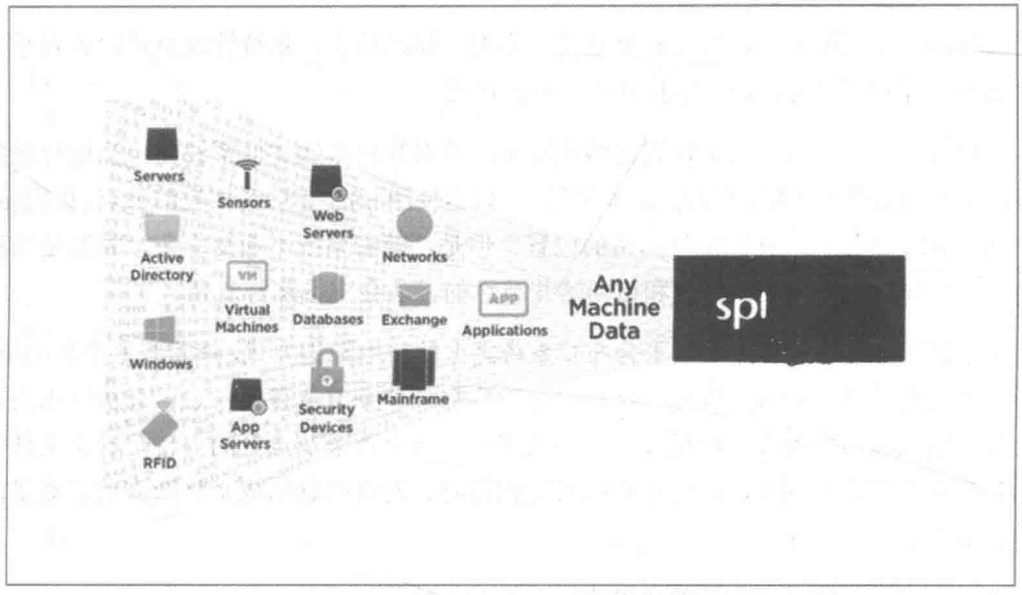


图 8-1　Splunk

Splunk 可以单机部署，也可以分布式部署。更多关于 Splunk 的详细功能，请参考官方文档（http://docs.splunk.com）。

- LogStash

LogStash（http://www.logstash.net）是一款开源的日志管理工具，使用 JRuby 语言实现。

它支持强大的日志输入、过滤器以及输出功能，主要功能包括：

- 收集日志
  LogStash 可以从多个数据源获取日志，譬如标准输入、日志文件或是 syslog 等。同时，内嵌支持多种日志的格式，如系统日志、Web 服务器日志、错误日志、应用日志等。另外，如果日志的输入是 syslog，用户不必在每台服务器上安装日志代理（Log Agent），默认的 rsyslog 客户端就可以直接同步日志。
- 过滤日志
  LogStash 内置了许多过滤器，比如 grep、split、multiline 等，能够方便地为用户定制过滤策略。
- 结果输出
  除了能将日志内容输出到标准输出，LogStash 还能和 ElasticSearch、MongoDB 等配合使用。另外，LogStash 还内置了一个 Web UI 来帮助用户查看和搜索日志。实际上，LogStash 通常会和 ElasticSearch 以及 Kibana 配合使用，组成 ELK（ElasticSearch 作为日志的搜索引擎，LogStash 作为日志的处理引擎，Kibana 作为前端的报表展示）。

本例中，我们就以 Splunk 作为日志聚合工具。Splunk 通过提供强大的索引方式，能够在复杂的企业环境中有效地对日志做索引。同时，Splunk 的界面友好，使用起来也较容易。感兴趣的读者也可以根据喜好或者经验，使用其他的工具完成类似的日志聚合功能。

8.2 Splunk 的核心

Splunk 主要包括如下三个部分。

- 数据（采集器）转发器
  该部分主要负责采集数据。实际上，Splunk 可以实时监控文件或者目录的变化，也可以从标准输出、网络程序中收集数据。
- 数据索引器
  该部分负责将收集的数据进行存储，并采用高效的索引方式对日志进行索引。
- 搜索、分析和可视化
  通过 Splunk 处理语言（Splunk Processing Language）为使用者提供强大的搜索和过滤功能。同时，Splunk 也提供了多样的数据报表显示功能，譬如表格、图表等。另

外，Splunk 还提供了灵活的告警设置，能够在某些项的结果超出阈值时，立刻发出告警通知。

实际上，Splunk 既可以单机部署，也可以分布式部署。在微服务架构中，由于不同的服务部署在不同的主机上，而我们希望对日志进行集中管理，因此通常使用的是分布式部署。分布式部署时，每个部署有服务/应用的主机上都会安装一个日志转发器（Splunk Forwarder），当前节点的日志通过转发器被转发到 Splunk 的重要服务器上，负责接收日志的这台 Splunk 服务器也被称作接收器。

8.3 安装 Splunk 索引器

Splunk 索引器的安装非常简单，步骤如下所示。

1. 首先，从 Splunk 官网下载 Splunk Enterprise 或者 Splunk Light 版本进行安装，如图 8-2 所示。

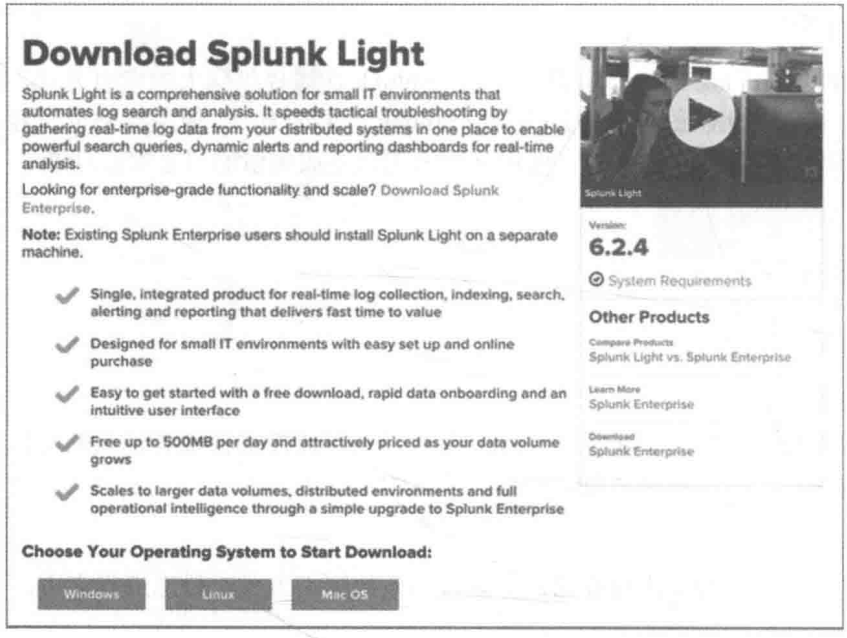


图 8-2 Splunk Light 版本下载

2. 根据向导一步步配置 Splunk 的相关信息。默认的，索引器和搜索器是安装在一起的。

安装完成后便可通过浏览器，访问默认的 8000 端口。

8.4 安装 Splunk 转发器

接下来，我们安装并配置 Splunk 转发器（Splunk Forwarder）。

1. 首先，从官方网站下载 Splunk Forwarder（http://www.splunk.com/en_us/download/universal-forwarder.html），并配置好索引器的 IP 地址和端口号。

2. 然后，设置 inputs.conf 文件，默认安装在$SPLUNK_DIR/etc/system/local/inputs.conf，并在其中添加如下配置：

```
[monitor:///var/log/httpd/error.products_service.log]
index = products-service
sourcetype = products_service_http_error_log

[monitor:///var/log/httpd/access.products_service.log]
index = prducts_service
sourcetype = products_service_http_access_log

[monitor:///var/log/products_service/production.log]
index = products_service
sourcetype = products_service_production_log
```

在如上代码中，“monitor”的配置表明 Splunk 转发器采集的数据源类型。

譬如，如下配置将运行 products-service 的 Apache 日志错误文件添加到 Splunk 转发器中：

```
[monitor:///var/log/httpd/error.products_service.log]
```

如下配置将运行 products-service 的 Apache 日志访问文件添加到 Splunk 转发器中：

```
[monitor:///var/log/httpd/access.products_service.log]
```

如下配置将运行 products-service 的 Apache 应用日志文件添加到 Splunk 转发器中：

```
[monitor:///var/log/products_service/production.log]
```

接下来，index = products_service 的配置参数表示 Splunk 对日志进行索引时的关键字名称。这也意味着当使用 Splunk Web 管理端搜索的时候，可以直接使用类似如下的方式缩小搜索范围：

```
index=products_services
```

sourcetype 的配置参数表明 Splunk 对日志源的分类。通过将日志源进行分组，可以很方便地从可视化面板中对不同的源进行过滤。

关于 Splunk 更多的配置参数，请参考 Splunk 官方文档（http://docs.splunk.com/Documentation/Splunk/6.1/admin/Inputsconf）。

8.5 日志查找

配置完 Splunk 的索引器、Splunk 采集器，我们就可以通过 Splunk 提供的 Web 管理接口对日志进行方便的查询与过滤了。Splunk 提供了很多关键字，可帮助我们有效地进行搜索。

譬如，如果想搜索 products-service 昨天的关键字包含“error”的日志，可以用如下方式进行搜索：

```
index=products_service sourcetype=products_service_production_log earliest=-1d@d
latest=-0d@d error
```

如上代码所示，通过指定 earliest 和 latest，确定搜索的时间段。

同时，Splunk 还提供了自定义报表的功能，使团队可以根据不同时期、项目发展的需要定义合适的报表，如图 8-3 所示。

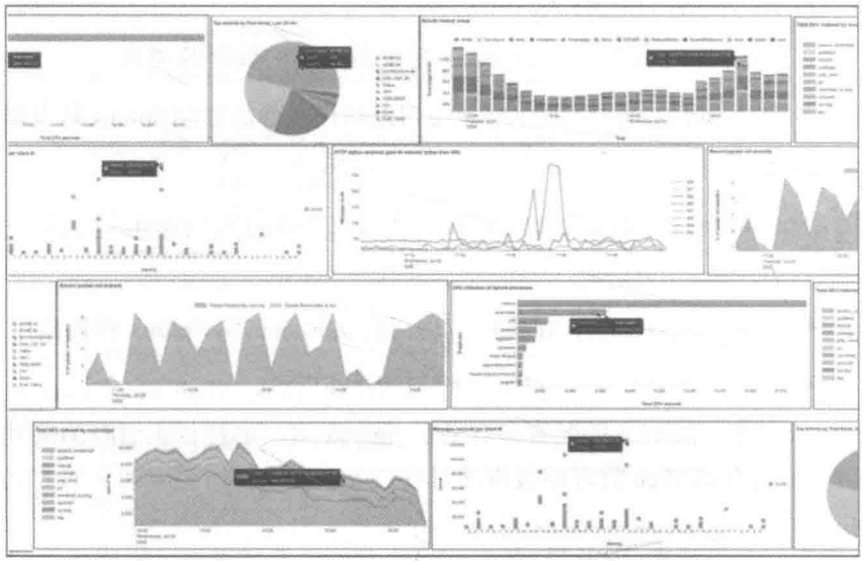


图 8-3 Splunk 日志报表

8.6 告警设置

利用 Splunk，我们还能配置告警功能。譬如，当日志中出现“error”的关键字时，或者当设置的条件达到阈值时，Splunk 能够发出告警邮件并通知给相关的责任人。

图 8-4 Splunk 告警设置

譬如，设置告警主要分成三步，如图 8-4 所示。

1. 保存搜索条件“Save Search”

首先，需要保存搜索条件。所谓搜索条件，其实就是查找日志时的查询语句。譬如，在之前的例子中搜索条件如下所示：

```
index=products_service sourcetype=products_service_production_log earliest=-1d@d
latest=-0d@d error
```

通过保存搜索条件，为后续的告警设置条件。

2. 设置“告警阈值”

基于步骤 1 的搜索结果，设置告警阈值。譬如，在图 8-4 中，告警阈值为“search count > 5”。即如果搜索结果的数量大于 5，则触发告警。

3. 定义“告警响应”

本步骤主要完成告警触发后的处理动作。通常，笔者将其设置为 E-mail 通知。

8.7 小结

在本章中，我们首先介绍了日志聚合机制在微服务实践中的重要性。

接下来，采用 Splunk 作为日志聚合的工具，描述了如何安装、配置 Splunk Server 以及如何配置 Splunk Forwarder。最后，介绍了如何通过 Splunk 查找日志、分析日志、建立控制面板以及设置告警等。

读者也可以根据喜好选择其他的日志聚合工具。但是随着微服务对系统带来的影响和改变，尽早安装并配置日志聚合工具，能为团队持续交付节省时间和人力成本。

第 9 章

监控与告警

在微服务架构中，每个服务都是一个可以独立运行的业务单元，同时每个服务都运行在独立的节点上。因此，我们需要为每个服务建立独立的监控以及告警机制，以监控服务的健康状况，并当异常发生或异常恢复时，能及时提醒用户。所以，监控是整个微服务运维环节，乃至微服务生命周期中非常重要的一环。

关于监控，目前业界已经有许多成熟的产品，譬如 Ganglia、Zabbix、NewRelic、Nagios 和 OneAPM 等。

在笔者的项目实践中，大部分的服务都使用 Nagios 作为监控工具。Nagios 是一个运行在 Linux/UNIX 平台上的开源监控系统，它可以检测主机和应用程序的健康状况，并当异常发生和异常修复时能及时提醒用户。

本章，我们就使用 Nagios 完成对 products-service 的监控。

9.1 Nagios 简介

Nagios 全名为 Nagios Ain't Goona Insist on Saintood，最初的项目名字是 NetSaint。它是一款免费的开源 IT 基础设施监控系统，其功能强大，灵活性强，能有效监控 Windows、Linux、VMware 和 UNIX 主机状态以及交换机、路由器等网络设置。同时，它能够实现错误通知、事件处理等功能。一旦主机或服务状态出现异常，Nagios 会发送邮件或短信第一时间通知 IT 运维人员，并在状态恢复正常后发出邮件或短信通知，如图 9-1 所示。

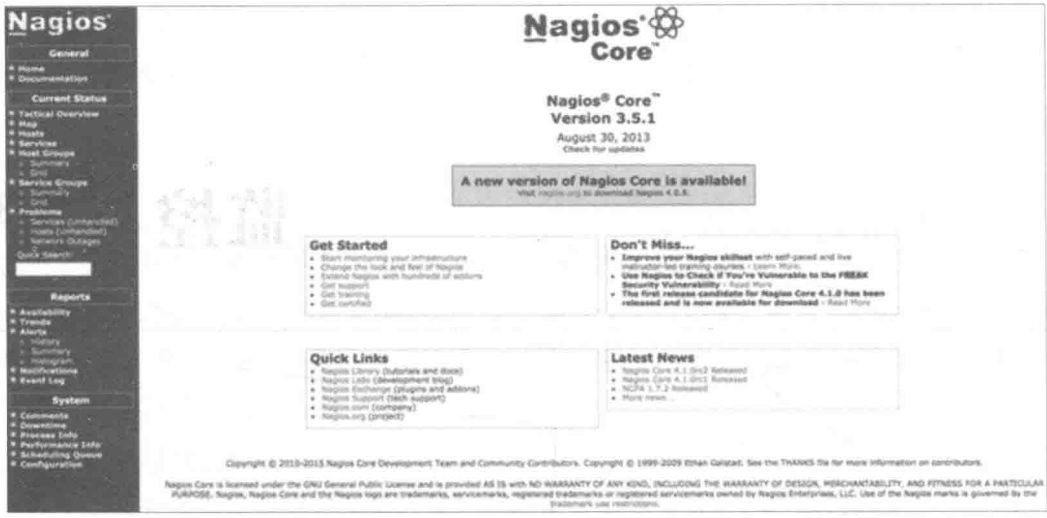


图 9-1　Nagios 管理首页

同传统的 IT 基础设施监控系统相比，Nagios 具有结构简单、可维护性强、学习成本低等优势，受到 IT 运维和管理人员的青睐。同时，Nagios 提供一个 Web 界面，方便 IT 运维人员使用浏览器查看系统的运行状态、网络状态、服务状态、日志信息等。

从 Nagios 的结构来说，主要分成两个部分：

- Nagios 核心
- Nagios 插件

Nagios 的核心部分只提供了很少的监控功能，要搭建一个完善的 IT 监控管理系统，用户还需要在 Nagios 服务器上安装相应的插件，才能提供多样的监控和告警功能。Nagios 的插件可以从 Nagios 官方网站（http://www.nagios.org/）下载，也可以根据实际需求自己编写。

通常，Nagios 可以支持的功能包括：

- 监控网络服务，包括但不限于 SMTP、POP3、HTTP、FTP、PING 等。
- 监控主机资源，譬如 CPU 负荷、磁盘利用率等。
- 自定义插件，用户可以根据实际需求编写监控与告警的相应插件。另外，插件的开发支持多种开发语言，譬如 Shell、Perl、Python 等。
- 告警通知，当服务或主机产生异常时，Nagios 能通过 E-mail、短信等方式发送告警。
- 提供 Web 界面，用于查看当前的网络状态、通知、故障历史和日志文件等。

9.2 Nagios 的工作原理

接下来，我们来看看 Nagios 的工作原理。

首先，Nagios 需要安装在一台 Linux 或 UNIX 服务器上，我们称这台服务器为监控中心；其次，对于每台需要被监控的节点（主机或者服务）而言，都需要运行一个与监控中心服务器进行通信的 Nagios 代理（Agent）；当 Nagios 工作时，监控中心服务器读取配置文件中的指令，然后与远程的 Nagios 代理程序通信，并指示远程的代理程序进行检查。虽然 Nagios 的核心部分必须在 Linux 或 UNIX 操作系统上运行，但是远程被监控的节点可以是任何能够与 Nagios 监控服务器进行通信的主机。根据远程 Nagios 代理的应答，Nagios 监控服务器将依据配置进行回应，如果检测返回值不正确，Nagios 监控服务器将通过一种或多种方式报警。Nagios 的具体工作原理如图 9-2 所示。

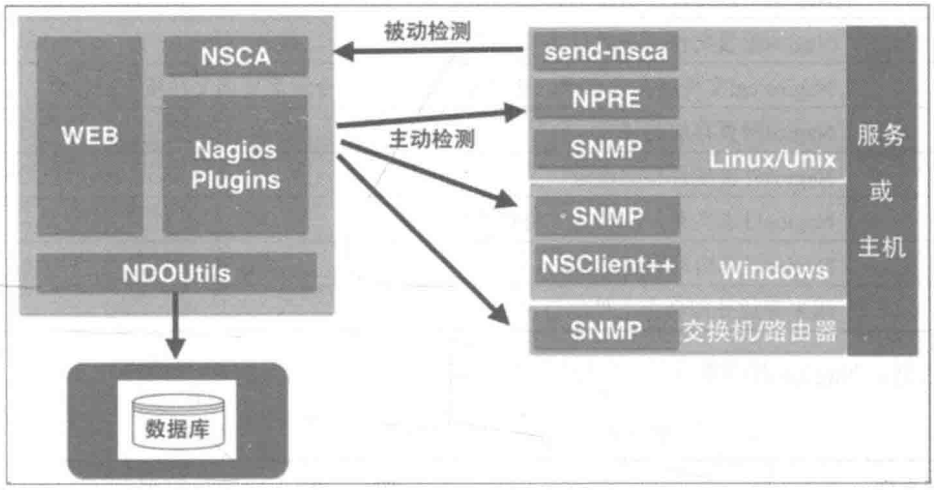


图 9-2 Nagios 的工作原理

另外，Nagios 监控服务器与 Nagios 代理的通信，存在如下两种方式。

- 主动检测

Nagios 监控服务器向 Nagios 代理发起请求，由代理完成实际检测后，返回应答给 Nagios 监控服务器。

- 被动检测

Nagios 代理检测本地的应用运行情况，将检测状态发送给 Nagios 监控服务器。

9.3 Nagios 安装

鉴于篇幅原因，笔者就不赘述如何安装 Nagios 了。更多关于 Nagios 的详细安装信息，请读者参考 Nagios 的官方文档（https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/3/en/quickstart.html）。

Nagios 安装完成后，会在/usr/local/nagios/目录下生成相应的主机、服务、命令、模板等配置文件，默认的配置文件在/usr/local/nagios/etc 目录下。

关于 Nagios 目录以及配置文件更详细的描述，如表 9-1 所示。

表 9-1 Nagios 的安装目录

目录名称	作　用
bin	Nagios可执行程序所在的目录
etc	Nagios配置文件所在的目录
sbin	Nagios cgi文件所在的目录，也就是执行外部命令所需要的文件所在的目录
share	Nagios网页存放路径
libexec	Nagios外部插件存放目录
var	Nagios日志文件、Lock等文件所在的目录
var/archives	Nagios日志自动归档目录
var/rw	用来存放外部命令文件的目录

除此之外，Nagios 相关配置文件的名称及用途如表 9-2 所示。

表 9-2 Nagios 的相关配置文件

配置文件	作　用
nagios.cfg	Nagios的主配置文件
objects	objects是一个目录，在此目录下有很多配置文件模板，用于定义Nagios对象
objects/commands.cfg	命令定义配置文件，其中定义的命令可以被其他配置文件引用
objects/contacts.cfg	定义联系人和联系人组的配置文件
objects/templates.cfg	定义主机和服务的一个模板配置文件，可以在其他配置文件中引用
objects/timeperiods.cfg	定义 Nagios 监控时间段的配置文件
objects/windows.cfg	监控 Windows 主机的一个配置文件模板，默认没有启用此文件

9.4 Nagios 的配置

Nagios 的配置主要包括如下几项：

- **主机**
  表示当前待监控的主机。
- **服务**
  表示当前待监控主机上的服务，一个主机可以对应多个服务。
- **联系人**
  表示监控出现问题时通知的联系人。
- **监控时间**
  表示监控的时间段。
- **监控命令**
  可以自定义监控命令，完成对主机或者服务的监控。

在 Nagios 安装目录中，有一个重要的模板文件 templates.cfg，其示范了如何定义主机、服务以及联系人等。

首先，让我们看看如何定义主机部分，代码如下所示：

```
define host {
  name linux-server #主机名称
  use generic-host
     # use 表示引用，也就是将主机 generic-host 的所有属性引用到 linux-server 中来
  check_period 24x7
     # check_period 设置 nagios 检查主机的时间段
  check_interval 5
     # 对主机的检查时间间隔，这里是 5 分钟
  retry_interval 1
     # 重试检查时间间隔，单位是分钟
  max_check_attempts 10
     # nagios 对主机的最大检查次数
  check_command check-host-alive
     # 指定检查主机状态的命令，其中 check-host-alive 在 commands.cfg 文件中定义
  notification_period workhours
```

```
    # 主机出现故障时，发送通知的时间范围，其中 workhours 在 timeperiods.cfg 中定义
  notification_interval 30
    # 当主机出现异常后，如果故障一直没有解决，nagios 再次发出通知的时间间隔。单位是分钟
  notification_options d,u,r
    # 定义主机在什么状态下可以发送通知给使用者，d 即 down，表示宕机状态，
      u 即 unreachable，表示不可到达状态，r 即 recovery，表示重新恢复状态。
    contact_groups admins
    # 指定联系人组，admins 在 contacts.cfg 文件中定义
   }
```

接下来，定义服务部分，代码如下所示：

```
define service {
  name local-service #定义一个服务名称

  use generic-service
    #引用服务 local-service 的属性信息，local-service 主机在 templates.cfg 文件中进行了定义


  max_check_attempts 4
    #最多检测 4 次，为了确定服务最终状态

  normal_check_interval 5 #每 5 分钟检测一次

  retry_check_interval 1
    #每 1 分钟重新检测服务，最终的状态能被确定
}
```

然后定义联系人部分，代码如下所示：

```
define contact   {
  name generic-contact #联系人名称
  service_notification_period 24x7
      # 当服务出现异常时，发送通知的时间段。时间段是 7×24 小时


  host_notification_period 24x7
      # 当主机出现异常时，发送通知的时间段。时间段是 7×24 小时


  service_notification_options w,u,c,r
      # 定义的是“通知可以被发出的情况”。w 即 warning，表示警告状态，
        u 即 unknown，表示不明状态，c 即 criticle，表示紧急状态，
        r 即 recover，表示恢复状态
```

```
    host_notification_options d,u,r
        # 定义主机在什么状态下需要发送通知给使用者。d 即 down，表示宕机状态，
          u 即 unreachable，表示不可到达状态，r 即 recovery，表示重新恢复状态。

    service_notification_commands notify-service-by-email
        #服务出现故障时，发送通知的方式，可以是邮件和短信。这里发送的方式
         是邮件，其中 notify-service-by-email 在 commands.cfg 文件中
         定义。

    host_notification_commands notify-host-by-email
        #主机出现故障时，发送通知的方式，可以是邮件和短信，这里发送的方式
         是邮件，其中 notify-host-by-email 在 commands.cfg 文件中定
         义。
}
```

9.5 监控 products-service

类似的，针对 products-service，我们将分成 4 个步骤对其进行监控配置。

1. 定义主机和服务监控。

```
define host {
        use                     business-hours-service
        hostgroups              products-service
        host_name               products-service.corp
        address                 products-service.internal.corp
        check_command           check_dig!$HOSTADDRESS$
}

define service {
        use                     generic-service
        host_name               products-service.corp
        service_description     HTTPS Healthcheck
        check_command           check_http_health
}
```

如上代码所示，首先定义主机、服务的监控，以及监控使用的命令。

在当前例子中，对主机监控的命令为 check_dig!$HOSTADDRESS$，主要监控 DNS 解析是否能正常工作。对服务监控的命令为 check_http_health，主要监控服务的 http 响应是否能正常工作。

2. 实现监控命令。

在第 1 步中，定义了主机和服务的监控命令（check_command），分别是 check_dig 和 check_http_health，接下来，我们来实现这两个命令。

在 commands.cfg 中依次添加 check_dig 以及 check_http_health 命令，代码如下所示：

```
# 'check_http_health' command definition
define command {
  command_name    check_dig
  command_line    /usr/lib/nagios/plugins/check_dig --query_address
                     $ARG1$ -H $HOSTADDRESS$ -t 30
}

# 'check_http_health' command definition
define command{
  command_name    check_http_health
  command_line    $USER1$/check_http -H $HOSTADDRESS$ -u
                  "/diagnostic/status/nagios" -w 50 -c 60 -e
                  "HTTP/1.0 200","HTTP/1.1 200" $ARG1$ -f follow
}
```

实际上，这两个命令分别来自 Nagios 的插件 check_dig 和 check_http。

3. 定义监控时间段。

通常，我们都将监控时间段定义在 template 中，代码如下所示：

```
define service {
  contact_groups                OperationsTeam
  name                          Working-hours-service-tmpl
  active_checks_enabled         1
  notifications_enabled         1
  check_period                  24x7
  normal_check_interval         1440
  retry_check_interval          5
  max_check_attempts            10 when a non-OK state is returned
```

```
  first_notification_delay      0
  notification_options          w,u,c,r
}
```

4. 定义联系人。

在这一步中，在 contact.cfg 中定义发生异常时待通知的联系人。

```
define contact {
  contact_name                  products-service-admin
  alias                         Nagios Admin
  use                           generic-contact
  email                         products-servicd@gmail.com
}

define contact {
  contact_name                  products-service-admin-pager
  alias                         Nagios Admin(Phone)
  use                           generic-contact-sms
  pager                         xxxxxx
}
```

如上代码所示，我们在 contact.cfg 中定义了两类联系人，一类为 products-service-admin，通知形式为邮件，另一类为 products-service-admin-pager，通知形式为短信。

9.6 告警

告警是运维环节另外一个非常重要的部分。当系统出现异常时，通过合适的告警机制，能及时、有效地通知相关负责人，做到早发现、早分析、早修复。

针对每个服务，都应该提供有效的告警机制，确保当服务出现异常时，能够准确有效地通知到责任人，并及时解决问题。

在监控部分，我们已经了解到，对于很多现有的监控工具，譬如 Nagios，已经能够提供告警功能。

不过，除了这些带告警功能的监控工具外，业界还有一些功能更强大，也更专业的告警工具，它们不仅能支持多种消息传递的方式，还能更及时地将消息发送给相关责任人，譬如 PagerDuty（https://www.pagerduty.com/）。

PagerDuty 是一款能够在系统出现问题时及时发送消息提醒的应用，提醒的方式包括屏幕显示、电话呼叫、短信通知、邮件通知等，同时还能够集成现有的即时消息通信工具，譬如 Slack、Skype 以及第三方的监控应用，譬如 Newrelic、Nagios、Splunk 等。除此之外，在规定时间内无人应答时，PagerDuty 还能自动将消息的重要性级别提高。

更多关于 PagerDuty 的介绍以及使用，请参考其相关文档。

9.7 小结

本章主要讨论了基于 Nagios 的监控告警设置。针对每个服务，我们都应该提供有效的监控与告警机制，确保当服务出现异常时，能够准确有效地通知到责任人，并及时解决问题。

本章虽然是基于 Nagios 完成的监控与告警，但感兴趣的读者可以选择其他工具，构建适合团队的运维机制。

第 10 章

功能迭代

在之前的部分，我们已经探讨了实现 products-service 的基本流程，如下所示：

- 场景分析。分析并定义了 products-service 的功能。
- 实现 Hello World API。
- 代码测试与静态检查。定义静态检查、复杂度检查以及代码测试覆盖率统计的规范。
- 构建 Docker 映像并部署 Docker 映像。
- 构建持续交付流水线。
- 设置日志聚合。
- 设置监控以及告警机制。

有了基础设施、构建、部署、持续交付流水线以及相应的运维保障机制，接下来，我们就可以通过频繁且持续迭代的方式，完成 products-service 需要的功能。

在着手之前，我们先将 products-service 的实现划分成几个小任务，将复杂的、庞大的工作拆分成小的任务，一方面便于团队跟踪进度，另外一方面也能帮助团队在单位时间内聚焦某一个任务，如图 10-1 所示。

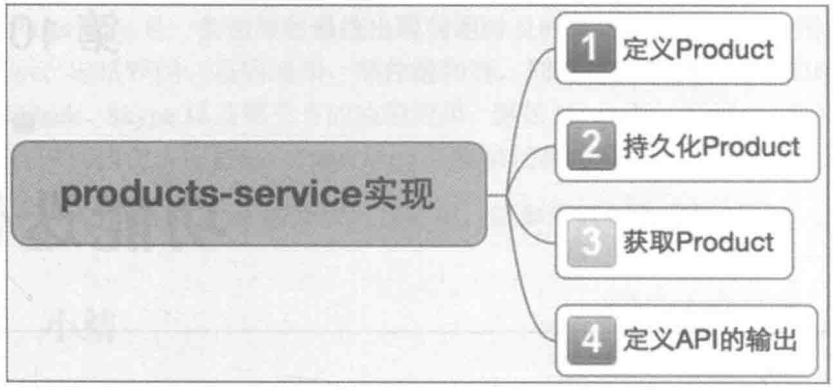


图 10-1　products-services 实现

10.1　定义模型

对当前的 products-service 例子而言，模型相对简单，只有一个 Product 类，具有一些属性，如下所示：

```
class Product
  include Virtus.model

  attribute :id,       Integer
  attribute :name,     String
  attribute :price,    Float
  attribute :category, String
end
```

在当前的例子中，我们使用 Virtus（https://github.com/solnic/virtus）简化模型属性的定义。

什么是Virtus

Virtus是使用Ruby语言实现的一个Gem，可支持并简化业务对象的属性类型、默认值等定义，同时能方便定义属性的约束条件。

10.2 持久化模型

第二步，实现模型的持久化。

我们知道，大部分应用程序都需要存储的功能。应用场景不同，存储的类型也存在很大差异，譬如关系型数据库适合存储持久化的结构化信息，NoSQL 适合存储非结构化信息，文件系统适合存储文本信息，而静态资源存储适合用来存放图片、CSS、JavaScript 等静态文件。

在笔者的实践中，模型的存取通常使用模型存储模式（Repository Pattern）来实现。模型存储模式最早由埃里克·伊文在他的《领域驱动设计》一书中提到，其最大优势在于屏蔽了模型的存储实现细节，让调用者更关注接口部分。

在当前的例子中，笔者使用 rom-sql（https://github.com/rom-rb/rom-sql）完成 Product 的存储以及获取。由于是示例，省去了关于数据连接以及配置等部分的处理，代码如下所示：

```
class ProductRepository
  class << self
    def find(id)
      relation.as(:products).find(id).first || raise(Error, 'Product not found')
    end

    private
    def relation
      Database.db.relation(:products)
    end
  end
end
```

如上代码所示，Database.db.relation(:products)以 Hash 的方式读取数据库 products 表中的所有数据，as(:products)则把 Hash 映射成对应的 Product 对象。

另外，我们需要对 rom-sql 的 relation 以及数据库和 model 之间的映射进行配置，基本配置如下：

```
def self.setup_relations(rom)
  rom.relation(:products) do
    def find(id)
      where(id: id)
    end
```

```
    end
  end

  def self.setup_mappings(rom)
    rom.mappers do
      define(:products) do
        model ProductService::Product
      end
    end
  end
```

对于当前 products-service 的例子而言，数据的存取相对比较简单。对于某些复杂的情况而言，还应该考虑安全性、读写比、可伸缩性等，这时候就会用到命令查询职责分离（Command Query Responsibility Segration）。

命令查询职责分离

在三层架构中，通常是通过数据访问层来修改或者查询数据。在这种情况下，对数据的读写针对的都是同一个业务模型。随着系统业务逻辑变得复杂以及访问量增加，这种设计可能会出现性能、安全、可伸缩性等问题。虽然我们可以通过在数据库层面进行读写分离，解决类似的问题，但如果只是数据库读写分离，业务上读写依旧混合的话，随着业务的复杂度增加以及变化频率的加快，依然会出现难以维护、灵活性不高，甚至性能问题。因此，操作和查询的分离能有效地从业务上将数据的修改和数据的获取职责分离，通过从业务角度区分数据读写，从而提高系统的性能、可扩展性和安全性。

更多关于CQRS的细节，请参考马丁·福勒的这篇CQRS文章（http://martinfowler.com/bliki/CQRS.html）。

模型存储模式与数据访问层

模型存储模式是领域驱动设计中的概念，它强调如何基于业务的需求实现模型的存储，它是一种抽象的设计方法。例如，模型有可能被存储到关系型数据库、NoSQL、文件系统、云存储等，也有可能作为其他服务的输入，继续被处理。

模型存储模式中定义的功能要体现领域模型的意图和约束。使用模型存储模式，隐含着一种思想，就是领域模型需要什么，它才提供什么，不需要的功能、不该提供的功能就不要提供，一切都以业务需求为核心。

数据访问层则更关注数据的存储方式以及存储功能的实现，并不严格受限于业务逻辑。使用数据访问层，其意图在于其能够提供数据访问的所有接口，业务逻辑层需要用哪个数据接口，可以由业务层根据场景来自由选则。

使用模型存储模式的另外一个优势在于，当依赖的环境构建或者访问成本很高的时候，能有效地对数据的存储或者获取的行为做Mock。几个月前，笔者就遇到类似的问题，由于业务需求，需要将某些模型的数据存储到亚马逊的云存储S3上，但由于网络延迟、安全、权限等因素，访问该环境所耗费的成本非常高。为了不让这部分功能影响团队本地的开发，团队就使用Mock的方式，构建一个假的模型存储，使得调用者能够有效地访问该接口并且迅速获取数据。

如果想了解更多关于数据访问层（DAL）与模型存储模式的区别，推荐读者阅读《领域驱动设计》中关于模型存储模式的部分。

模型存储模式与对象关系映射

对象关系映射，是随着面向对象的开发方法和关系型数据库的发展，而诞生的一种将对象和关系型数据库进行映射的开发方法。我们知道，对象和关系数据是业务模型的两种表现形式，业务模型在内存中表现为对象，在数据库中表现为关系数据。因此，对象关系映射一般以中间适配的形式存在，完成对象到关系数据库数据、或者关系数据库中的数据到对象的转换。

因此，对象关系映射可以理解成基于模型存储模式，对关系数据库访问的一种实现方式。

10.3 定义表现形式

第三步，定义 Product 的表现形式。表现形式通常用来描述业务模型如何在应用层显示。

大多数情况下，表现形式的定义包括两部分内容：数据展现和交互方式。

- **数据展现**

该部分描述业务模型中的数据如何在应用层表现出来。

这部分和几年前提出的 DTO（Data Transfer Object）类似。通过适配器，将业务模型的数据转换成表现层需要的数据。

- **交互方式**

该部分定义（数据）消费者如何和生产者（服务）交互，交互的协议和格式遵循什么样的规范。例如在 HTTP 中，使用什么样的 Header、Content-type、Accept-type 等。在当前的例子中，笔者使用基于 REST 的 HAL 协议作为服务间（也就是消费者与生产者间）通信的规范，并使用 Roar（https://github.com/apotonick/roar）对数据进行转换和渲染。

在 Product 的表现形式里，主要包括两部分：

- **明细表现形式**

对于 Product 明细对应的表现形式，代码实现如下所示：

```
module ProductRepresenter
  include Roar::JSON::HAL
  include Roar::Hypermedia
  include Grape::Roar::Representer

  property :id
  property :name
  property :price
  property :category

  link :self do |opts|
    request = Grape::Request.new(opts[:env])
    "#{request.base_url}/products/#{id}"
  end
end
```

在上面的 ProductRepresenter 中声明了 id、name、price、category 以及 self link，最终 Product 明细通过 API，响应的内容如下所示：

```
{
    "id": 1,
    "name": "Building microservice",
    "price": 69.00,
    "category": "Book",
    "_links": {
        "self": {
        "href": "http://localhost:9292/products/1"
        }
    }
}
```

- 列表表现形式

对于 Product 列表对应的表现形式，代码实现如下所示：

```
module ProductsRepresenter
  include Roar::JSON::HAL
  include Roar::Hypermedia
  include Grape::Roar::Representer

  collection :entries, extend: ProductRepresenter, as: :products
end
```

在上面的 ProductsRepresenter 中，我们使用 collection 定义响应的结果为一组资源。同时，定义每个资源的内容都为 ProductRepresenter 的输出。

输出结果如下所示：

```
{
  "products": [
    {
      "id": 1,
      "name": "The Docker Book",
      "price": 69.00,
      "category": "Book",
      "_links": {
          "self": {
          "href": "http://localhost:9292/products/1"
          }
      }
    },
```

```
    {
      "id": 2,
      "name": "Building microservice",
      "price": 100.00,
      "category": "Book",
      "_links": {
          "self": {
          "href": "http://localhost:9292/products/2"
          }
      }
    }
  ]
}
```

10.4 实现 API

最后，我们需要构建 API，为使用者提供管理产品信息的接口。

这里的 API 主要用来接收请求、调度处理逻辑并返回响应结果，和我们所熟悉的传统 Web 框架中的 controller 作用类似，譬如 Spring MVC 中的 controller，Rails 框架中的 controller 等。

在当前的例子中，笔者使用 Grape（https://github.com/intridea/grape）实现 API 相关的功能，具体实现如下：

```
module ProductService
  class API < Grape::API

    # 定义默认的 content-type 以及内容协商机制
    content_type :json, 'application/hal+json'
    format :json

    # 使用 Roar 渲染模型
    formatter :json, Grape::Formatter::Roar

    # 定义 Error
    rescue_from RecordNotFoundError do |error|
```

```
  Rack::Response.new({ errors: error.message }.to_json, 404).finish
end

# 定义资源以及操作
resource :products do
  get do
    present ProductRepository.all, with: ProductsRepresenter
  end

  route_param :id do
    get do
      present ProductRepository.find(params[:id]), with:
      ProductRepresenter
    end

    params do
      requires :product, type: Hash do
        optional :name, type: String, allow_blank: false
        optional :price, type: Float, allow_blank: false
        optional :category, type: Integer, allow_blank: false
      end
    end

    put do
      present ProductRepository.update(params[:product].merge(id: params[:id])),
      with: ProductRepresenter
    end

    delete do
      ProductRepository.delete(params[:id])
      body false
    end
  end

  params do
    requires :product, type: Hash do
      requires :name, type: String, allow_blank: false
      requires :price, type: Float, allow_blank: false
      requires :category, type: Integer, allow_blank: false
```

```
      end
    end

    post do
      present ProductRepository.create(params[:product]), with: ProductRepresenter
    end
  end

  get '/' do
    { '_links' => HAL::Index.links }
  end
 end
end
```

对应的 API 的测试代码如下所示：

```
describe 'GET /products/:id' do
 subject { get "/products/#{id}" }

 context 'when the product does not exist' do
  let(:id) { 12345 }
  it 'should return a 404' do
   subject
   expect(last_response.status).to eq 404
  end
 end

 context 'when the product does exist' do
  let(:product) { ProductRepository.create ModelFactory.build_product_attributes }
  let(:id) { product.id }
  it 'should return the details of the product identified by the id provided' do
   subject
   expect(last_response.status).to eq 200
   expect(json_response).to be_json_of_product product
  end
 end
end
```

这里我们看到，API 将我们之前定义的 ProductRepository 和 ProductRepresenter 联系了起来。当没有查到指定 ID 对应的产品信息时，ProductRepository 会抛出 RecordNotFoundError

并且被 API 捕获，并为调用者返回错误。

基于上面的描述，products-service 实现的代码结构如下所示：

```
├── app
|   ├── api.rb
|   ├── models
|   |   └── product.rb
|   ├── repositories
|   |   ├── product_repository.rb
|   |   └── record_not_found_error.rb
|   └── representers
|       ├── product_representer.rb
|       └── products_representer.rb
```

关于更多的代码实现以及测试，请参考本书相应的源码。

10.5 服务描述文件

通常，对于每个服务而言，都会存在一些描述信息，包括服务的介绍、维护者的信息，以及如何部署、监控和设置告警机制，来帮助团队更好地理解和快速运行服务。这就是服务描述文件。

在笔者的实践中，服务描述文件主要包括如下几个部分：

- 服务介绍
- 维护者信息
- 服务的 SLA
- 服务运行环境
- 开发、测试、构建和部署
- 监控和告警

如下是笔者常用的服务描述文件模板，包括但不限于其中描述的内容。

1. 服务介绍
   - 服务名称

- 服务功能

2. 服务维护者
   - 记录服务的维护者，通常是能直接联系到的个人
3. 服务可用期（SLA，Service Level Agreements）
   - 服务可用期，譬如，周一~周五（9:00~19:00）
4. 运行环境
   - 生产环境地址
     譬如http://product-service.vendor.com
   - 测试环境地址
     譬如http://product-service.test.vendor.com
5. 开发（描述开发相关的信息），通常包括但不限于以下几项。
   - 如何搭建开发环境
   - 如何运行服务
   - 如何调试
6. 测试（描述测试相关的信息），通常包括但不限于以下几项。
   - 测试策略
   - 如何运行测试
   - 如何查看测试的统计结果，譬如覆盖率、运行时间
7. 构建（描述持续集成以及构建的信息），通常包括但不限于以下几项。
   - 持续集成环境
   - 持续集成流程描述
   - 构建后的部署包发布
8. 部署（描述部署相关的信息），通常包括但不限于以下几项。
   - 如何部署到不同环境
   - 部署后的功能验证

9. 运维（描述运维相关的信息），通常包括但不限于以下几项。
   - 日志聚合的访问URL
   - 监控信息的访问URL

10.6 小结

基于前几章实现的微服务交付流水线以及监控、告警机制，本章中，笔者从服务的业务需求出发，通过迭代的方式逐渐实现业务功能，包括 API 的业务逻辑、请求响应的处理、错误异常的处理等。同时，通过定义服务描述文件，帮助团队更好地管理微服务并共享微服务的相关信息。

第 3 部分

进阶篇

之前的部分已经讨论了微服务架构的理论基础，并且通过实践演示了如何开始构建微服务。在本部分中，笔者将和大家讨论在微服务架构的实施中，如何有效地完成持续交付流水线的搭建、如何有效地制定微服务的测试策略，包括单元测试、消费者驱动的契约测试、组件测试等以及如何选择合适的轻量级通信机制。

在本部分的最后，笔者将通过一个真实案例，介绍使用微服务架构改造遗留系统的过程，包括改造策略、微服务的开发模板、代码生成工具以及基于 Netflix 的 Asgard（https://github.com/Netflix/asgard）的一键部署工具。

本部分的内容主要包括：

- 微服务与持续交付
- 微服务与轻量级通信机制
- 微服务与测试
- 使用微服务架构改造遗留系统

第 11 章

微服务与持续交付

十年以前，软件在一年之内的交付次数屈指可数。

过去的十年间，交付的过程一直被不断地优化和改进。从早期的 RUP 模型、敏捷、XP、Scrum，再到近几年的精益创业、DevOps，都力求能更有效地降低交付过程所耗费的成本并提高效率，从而尽早实现软件的价值。

持续交付是一种软件开发策略，用于优化软件交付的流程，以尽快得到高质量、有价值的软件。这种方法能帮助组织更快地验证业务想法，并通过快速迭代的方式持续为用户提供价值。

对于任何一个可交付的软件来说，必然要经历分析、设计、开发、测试、构建、部署、运维的过程。而从持续交付的角度来分析，对于任何一个可部署的独立单元，它都应该有一套独立的交付机制，来有效支撑其开发、测试、构建、部署与运维的整个过程。

如第 2 章所述，微服务将一个应用拆分成多个独立的服务，每个服务都具有业务属性，并且能独立地被开发、测试、构建、部署。换句话说，每个服务都是一个可交付的“系统”。那么在这种细粒度的情况下，如何有效保障每个服务的交付效率，快速实现其业务价值呢？

本章，我们就来探讨微服务与持续交付。本章的内容主要包括：

- 什么是持续交付
- 持续交付的核心
- 微服务与持续交付

从技术上讲，持续交付是软件系统的构建、部署、测试、审核、发布过程的一种自动化实现，而其中的核心则是部署流水线。因为部署流水线能够将这几个环节有效地连接起来。当然，像探索性测试、易用性测试，以及管理人员的审批流程等还是需要一定的手工操作，如图 11-1 所示。

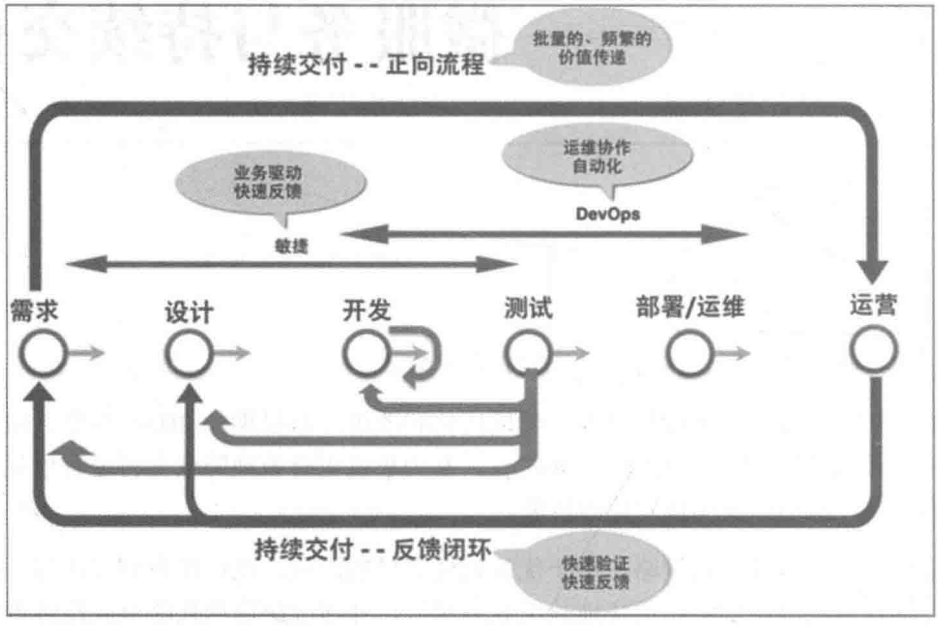


图 11-1　持续交付

11.1　持续交付的核心

在持续交付过程中，需求以小批量形式在团队的各个角色间顺畅流动，并以较短的周期完成小粒度的频繁发布。实际上，频繁的交付不仅能持续为用户提供价值，而且能产生快速的反馈，帮助业务人员制定更好的发布策略。

因此，持续交付的核心在于三个字：小、频、快。

- 小批量价值流动

通过建立自动化的构建及部署机制，将业务功能以小批量的方式，从需求产生端移动到用户端。

- 频繁可发布

通过建立自动化的构建及部署机制，将小批量的业务功能频繁地从需求产生端移动到用户端，持续地交付价值。

- 快速反馈

通过建立高效的反馈机制，快速验证需求是否有效。同时根据反馈，及时指导业务团队并调整策略，优先为用户交付高价值的功能。

持续交付让业务功能在整个软件交付过程中以小批量方式在各角色间顺畅流动，通过更频繁的、低风险的发布快速获得用户反馈，以此来持续达成业务目标。

11.2 微服务架构与持续交付

从交付的角度来分析，对于任何一个可部署的独立单元，它都应该有一套独立的部署流水线，来有效支撑其开发、测试、构建、部署与运维的整个过程。

在微服务架构中，由于每个服务都是一个独立的、可部署的业务单元，因此，每个服务也应该对应着一套独立的持续交付流水线，可谓是“麻雀虽小，五脏俱全”。

接下来，让我们看看在微服务的架构中，如果构建这样一套持续交付的流水线，各个环节需要做什么样的准备。

11.2.1 开发

对于微服务架构而言，如果希望构建独立的持续交付流水线，我们在开发阶段应该尽量做到如下几点。

- 独立代码库

对于每一个服务而言，其代码库和其他服务的代码库在物理上应该是隔离的。所谓物理隔离，是指代码库本身互不干扰，不同的服务有不同的代码库访问地址。譬如，对于我们平时使用的 SVN、GIT 等工具，每个服务都对应且只对应一个独立的代码库 URL。如下所示，分别表示产品信息服务和客户信息服务的代码库。

```
http://github.com/xxxxx/products-service
http://github.com/xxxxx/customers-service
```

除此之外，对不同服务隔离代码库的另一个好处在于，对某服务的代码进行修改，完全不用担心影响其他服务代码库中的代码，在很大程度上避免了修改一处，导致多处发生缺陷的情况。

- **服务说明文件**

对于每一个服务而言，都应有一个清晰的服务说明，描述当前服务的信息，同时帮助团队更快地理解并快速上手。譬如，在笔者的微服务实践过程中，对于每一个代码库，其服务说明都包括如下几个部分。

1. 服务介绍

- 服务提供什么功能，譬如产品服务主要提供产品数据的获取或者存储。
- 谁是服务的消费者。譬如产品服务的消费者为电商的前端网站系统或者 CRM 系统。

2. 服务维护者

- 挑选 1~2 个团队的成员，作为服务的负责人，登记其姓名、电子邮件、电话等联系方式，以便其他团队遇到问题能及时找到服务的负责人。

3. 服务可用期

- 服务可用周期，如 7×24 小时，或周一~周五（7:00~19:00）等。
- 可用率，可用率是指服务可以正常访问的时间占总时间的百分比，如 99.9%或者 99%。如果服务一天内都可以访问，则服务当天的可用率为 100%。如果服务有 3 分钟访问中断，而一天共有 1440 分钟，那么服务的可用率为：((1440 - 3) / 1440) * 100%，也就是 99.79%。
- 响应时间，指服务返回数据的可接受响应时间。譬如为 0.5~1 秒。

4. 定义环境，描述服务运行的具体环境，通常包括：

- 生产环境
- 类生产环境
- 测试环境

5. 开发，描述开发相关的信息，通常包括：

- 如何搭建开发环境

- 如何运行服务
- 如何定位问题

6. 测试，描述测试相关的信息，通常包括：

- 测试策略
- 如何运行测试
- 如何查看测试的统计结果，譬如测试覆盖率、运行时间、性能等。

7. 构建，描述持续集成以及构建相关的信息，通常包括：

- 持续集成访问的 URL
- 持续集成的流程描述
- 构建后的部署包

8. 部署，描述部署相关的信息，通常包括：

- 如何部署到不同环境
- 部署后的功能验证

9. 运维，描述运维相关的信息，通常包括：

- 日志聚合的访问
- 告警信息的访问
- 监控信息的访问

- **代码所有权归团队**

团队的任何成员都能向代码库提交代码，做到任何服务代码的所有权归团队。

代码所有权归团队，它表现的更多的是团队协同工作的观念，即集体工作的价值大于每个个体生产价值的总和。当所有权属于集体的时候，那么每个开发者就不应当出于个人原因来降低代码质量。代码质量上出现的问题应该在整个团队的努力下共同处理。

相反，如果某段代码背后的业务知识没有适当地分享给其他人，那么代码的演变逐渐变为依赖于具体的某个人，瓶颈也就由此而产生。

- **有效的代码版本管理工具**

代码版本管理工具的使用早已经成为开发者必备的核心技能之一，譬如 Git、Mercurial，

以及 CVCS（Centralized Version Control System）等。不过，团队最好能使用分布式版本控制工具（DVCS，Distributed Version Control System），它可以避免由于客户端不能连接服务器所带来的无法提交代码的问题。

- 代码静态检查工具

另外，团队也需要有代码静态检查工具，帮助完成代码的静态检查。譬如 Java 语言的 CheckStyle、Ruby 语言的 Rubocop 等。

另外，代码度量（Code Metrics）工具，譬如常用的 SonarQube、Ruby 的 Cane 等，能够保障团队内部代码的一致性和可维护性。

- 易于本地运行

作为团队的开发人员，当我们从代码库检出（Check out）某服务的代码后，应该花很短的时间、很低的成本就能在本地环境中将服务运行起来。如果依赖于外部资源，并且构建、使用成本较高，就应该考虑采取其他打桩的机制来模拟这些外部资源。这类外部资源通常指数据库、云存储、缓存或者第三方系统等。

譬如，笔者最近参与的一个企业内部系统改造项目，使用了 OKTA 集成单点登录的功能。开发环境下当然也可以使用 OKTA，但由于网络、安全、审批等多种原因，极大影响了开发人员在本地环境访问 OKTA 的效率。最后，团队采用打桩的机制，构建了一套符合 OKTA 协议的模拟 OKTA，在本地使用。在开发环境下，通过加载这个模拟的 OKTA，有效地解决了本地访问 OKTA 时间较长的问题。

另外一个例子是，笔者在系统中使用了 AWS 的 S3 服务。由于权限、网络等多种因素的存在，本地开发时使用 S3 的成本非常高，因此就构建了一套模拟的 S3 环境。当服务运行在开发环境时，加载开发模式的环境变量，访问本地的 Mock S3 环境；而在生产环境，则使用生产模式的 S3 地址。在不改变任何代码的前提下，帮助团队快速在本地搭建运行环境并演示，极大地提高了开发效率，如图 11-2 所示。

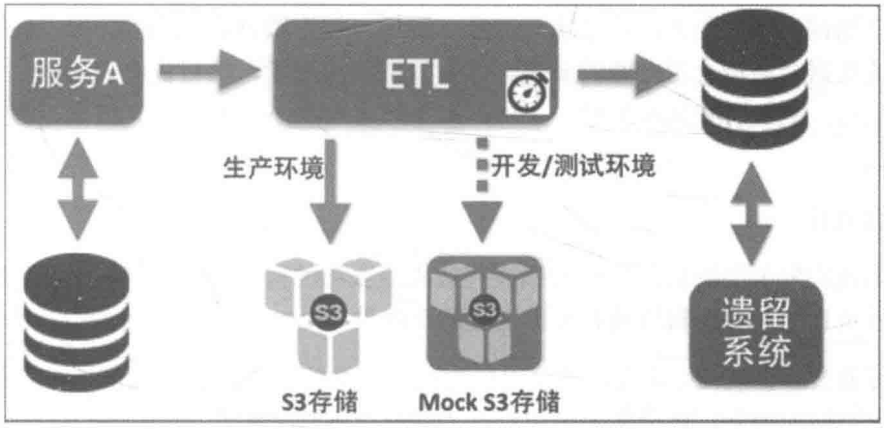


图 11-2 模拟 S3 存储

11.2.2 测试

对于微服务架构，如果希望构建独立的持续交付流水线，我们在测试方面应该注意以下几点。

- 集成测试的二义性

对于任何一个服务而言，单元测试必不可少。但是否需要集成测试，团队可以根据喜好自行决定。笔者个人建议明确定义集成测试的范围，因为“集成”这个词，很难有一个准确的度量机制。到底什么样的组合才叫集成？其可以是对外部不同系统之间的测试组合，也可以是系统内部实现逻辑、类与类之间调用的组合，因此“集成测试”这个术语，在团队和组织内部，容易在沟通过程中产生误解。

- Mock 与 Stub

对于单元测试而言，我们可以使用 Mock 框架帮助我们完成对依赖的模拟（Mock）或者打桩（Stub），譬如 Java 的 Mockito、Ruby 的 RSpec 等。当然，如果对象之间的依赖构建成本不高，也可以使用真实的调用关系而非 Mock 或者 Stub 机制。关于 Mock 和 Stub 的区别，有兴趣的读者可以参考 ThoughtWorks 首席科学家马丁 • 福勒的 *Mocks Aren't Stubs* 这篇文章。

- 接口测试

除了单元测试覆盖代码逻辑外，至少还应该有接口测试来覆盖服务的接口部分。注意，

对于服务的接口测试而言，更关注的是接口部分。譬如，作为数据的生产者，接口测试需要确保其提供的数据能够符合消费者的要求。作为数据的消费者，接口测试需要确保，从生产者获取数据后，能够有效地被处理。另外，对于服务与服务之间的交互过程，最好能设计成无状态的。

- 测试的有效性

如果单元测试的覆盖率够高，接口测试能有效覆盖服务的接口，那么基本上测试机制就保障了服务所负责的业务逻辑以及和外部交互的正确性。

有些朋友可能会存在疑问，是否需要使用行为测试的框架，譬如像 Cucumber、JBehave 等的工具，基于不同的场景，做一些类似用户行为的测试呢？

实际上，这里并没有确定的答案。在笔者参与的项目中，通常都是在最外层做一部分行为测试，原因有以下几点。

1. 通常我们所说的服务，大多是不涉及用户体验部分的。也就是说，作为服务，更关注的是数据的改变，而不是同用户的交互过程。譬如，当我们从电商网站挑选某件商品、下单，一个新的订单就生成了。这时候，订单的状态可能是“新建”。随后，当完成付款时，那么订单的状态可能会被更新成“已支付”。如果忽略状态更新的实现流程，譬如同步更新、异步更新等，那么从服务的角度而言，其并不在意用户到底是从 PC 端的浏览器，还是手机上的 APP 来完成订单的支付，服务自身只完成了一件事：就是完成订单状态从“新建”到“已支付”的更新。

2. 服务作为整个应用的一部分，能够独立存在，那必然有其对应的边界条件。前面提到，单元测试保障内部逻辑，接口测试保障接口，在这样的前提下，服务的正确性和有效性在大部分情况下已经得到了验证。

3. 从经典的测试金字塔来看，越是偏向于用户场景、行为的测试，其成本越高，反馈的周期也越长；相反，越是接近代码级别的测试，成本越低，反馈周期也相对较短，如图 11-3 所示。

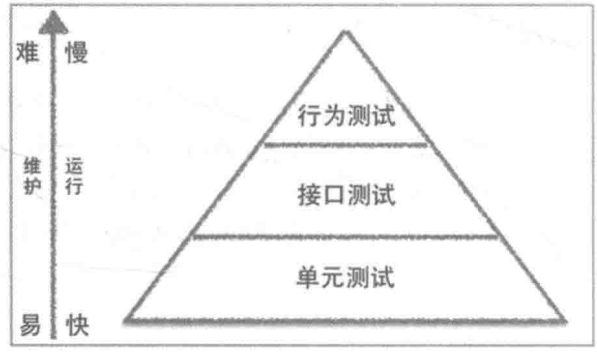


图 11-3　测试金字塔

11.2.3 持续集成

持续集成经过多年的发展，已成为系统构建过程中众所周知的最佳实践之一。对于每个独立的、可部署的服务而言，应为其建立一套持续集成的环境（Continuous Integration Project）。

当团队成员向服务的代码库提交代码后，配置好的持续集成工程会通过定期刷新或者 WebHook 的方式检测到代码变化，触发并执行之前开发阶段定义的静态检查、代码度量、测试以及完成构建的步骤，如图 11-4 所示。

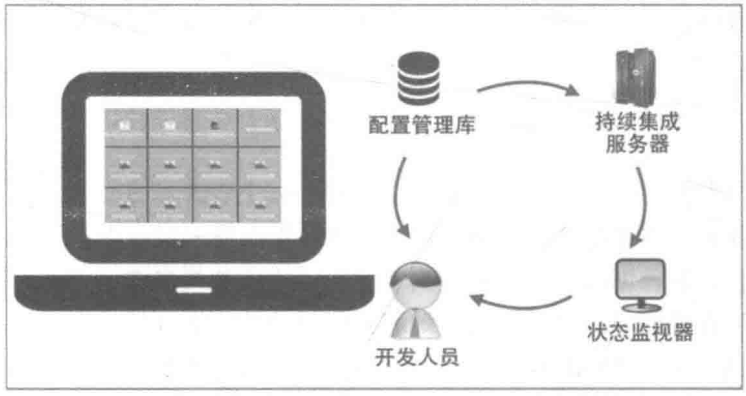


图 11-4 持续集成

常用的企业级持续集成服务器有 Jenkins、Bamboo 以及 GO 等，在线的持续集成平台有 Travis-CI、Snap-CI 等。

更多关于持续集成的细节，请参考 ThoughtWorks 首席科学家马丁·福勒的这篇文章 http://www.martinfowler.com/articles/continuousIntegration.html。

11.2.4 构建

每个服务都是一个可独立部署的业务单元，经过静态检查、代码度量、单元测试、接口测试等阶段后，构建符合需求的部署包。

部署包存在的形式是多种多样的，可以是 deb 包、rpm 包，能在不同 UNIX 操作系统平台直接安装；也可以是 zip 包、war 包等，只需将其复制到指定的目录下，执行某些命令，就可以工作。当然，也有可能是基于某特定的 IAAS 平台，譬如亚马逊的 AMI，我们称之为映像包（Image）。

另外，作为容器化虚拟技术的代表，Docker（一个开源的 Linux 容器）的出现，允许开发者将应用以及依赖包打包到一个可移植的 Docker 容器中，然后发布到任何装有 Docker 的 Linux 机器上。

通过使用 Docker，我们可以方便地构建基于 Docker 的部署镜像包。

11.2.5 部署

对于每个独立的服务而言，如果希望构建独立的持续交付流水线，需要选择部署环境并制定合适的部署方式来完成部署。通常，我们可以从如下两个维度考虑如何进行部署。

1. 部署环境

- 基于云平台

我们知道，云平台是一个很广的概念，其中主要包括 IAAS、PAAS 和 SAAS 三层。当基于云平台部署的时候，要先分清楚部署的环境，即部署将发生在哪一层。由于 SAAS 层是相对于应用的使用者而言，软件即服务。即对使用者而言，不需要再去考虑本地安装、数据维护等因素，直接通过在线的方式享受服务，这和我们讨论的部署环境没有关系。因此，这里我们主要讨论 IAAS 层和 PAAS 层的部署。另外，笔者这里没有区分是公有云还是私有云。公有云指运行在 Internet 上的云服务，私有云则通常指运行在企业内部 Intranet 的云服务。

  - 基于 IAAS 层

    云平台的 IAAS 层，通常包括运行服务的基础资源，譬如计算节点、网络、负载均衡器、防火墙等。因此，对于该层的部署包而言，实际上应该是一个操作系统映像，映像里包含运行服务所需要的基本环境，譬如 JVM 环境、Tomcat 服务器、Ruby 环境或者 Passenger 配置等。当在 IAAS 层部署服务时，不仅可以使用映像创建新的节点，也可以创建其他系统相关的资源，譬如负载均衡器、自动伸缩监控器、防火墙、分布式缓存等。

  - 基于 PAAS 层

    PAAS 层并不关心基础资源的管理，它更关注的是服务或者应用本身。因此，对于该层的部署包而言，通常是能直接在 UNIX 操作系统安装的二进制包（譬如 deb 包或者 rpm 包等），或者是压缩包（譬如 zip 包、tar 包、jar 包或者 war 包等），

将其复制到指定目录下解压缩，启动相关容器，就可以工作。

除此之外，也可以使用 PAAS 平台提供的工具或者 SDK，直接对当前的代码进行部署。譬如 Heroku 提供的命令行，就能很方便地将 Java、Ruby、NodeJS 等代码部署到指定的环境中。

- 基于数据中心

云平台已经成为大家公认的未来趋势之一，但是对于很多传统的企业，由于组织或者企业内部多年业务、数据的积累，以及组织架构、团队、流程固化等原因，无法从现有数据中心一步迁移到云端。而且，针对传统的数据中心，其对应的环境通常比较复杂，既没有 IAAS 那种按需创建资源的灵活性，也没有 PAAS 这种资源能够被自动化调配的可伸缩性。这时候，对于数据中心而言，部署就相对较麻烦，需要投入更多的成本构建环境以及调配资源。

很多企业也开始尝试在数据中心的节点上创建虚拟机（譬如 VMware、Xen 等），以帮助简化资源的创建以及调配。

- 基于容器技术

容器技术，是一种利用容器（Container）实现虚拟化的方式。同传统的虚拟化方式不同的是，容器技术并不是一套完全的硬件虚拟化方法，它无法归属到全虚拟化、部分虚拟化和半虚拟化中的任意一个，它是一个操作系统级的虚拟化方法，能为用户提供更多的资源。

过去两年里，Docker 的快速发展使其成为容器技术的典型代表。Docker 可以运行在任意平台上，包括物理机、虚拟机、公有云、私有云、服务器等，这种兼容性使我们不用担心生产环境的操作系统或者平台的差异性，能够很方便地将 Docker 的映像部署到任何运行 Docker 的环境中。

2. 部署方式

部署方式，是指通过什么样的方法将服务有效地部署到相应的环境。对于服务而言，由于部署环境的不同，采用的部署方式自然也不同，如图 11-5 所示。

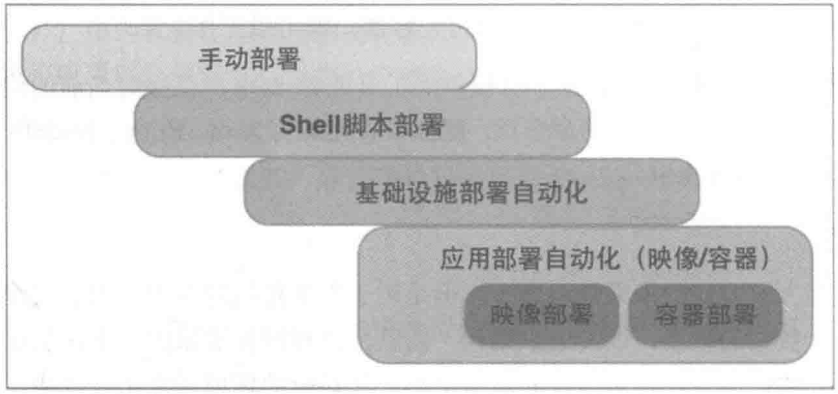


图 11-5　部署方式的演变

- **手动部署**

对于传统的数据中心环境，考虑到资源有限以及安全性等因素，通常的部署方式都是使用 SSH 工具，登录到目标机上，下载需要的部署包，然后复制到指定的位置，最后重启服务。

- **脚本部署**

由于部署团队每次都要手动下载、复制，不仅效率低，而且人为出错的概率也大。因此，很多企业和组织使用 Shell 脚本将这些下载、复制、重启等过程逐渐实现自动化，大幅提升了效率。Shell 脚本的优势是兼容性好，但其弊端在于实现功能所需要的代码量大，可读性也较差，时间长了不易维护。

- **基础设施部署自动化**

随着业务的发展，很多组织以及团队逐渐发现，环境的安装和配置、应用或者服务的部署所耗费的成本越来越高。“基础设施自动化”这个概念的提出，正好有效地解决了这一类问题。于是，越来越多的组织开始尝试使用 Chef、Puppet、Ansible 等工具，完成软件的安装和配置，以及应用或者服务的部署。

- **应用部署自动化**

无须过多的人工干预、一键触发即可完成部署的自动化方式可以说是任何组织，从业务、开发到运维都希望达成的目标。但说起来容易，实现起来却很难，而且这也不是一蹴而就的过程。需要随着组织或者企业在业务的演进过程中、技术的积累过程中，逐渐实现

自动化。通常，应用部署的自动化主要包括以下两部分。

- 映像部署

  私有云、公有云的出现，使得部署方式发生了显著的变化。面对的环境不一样，因此部署包也不一样。以前的 war 包、rpm 包，在基于 IAAS 的云平台上，都可以变成映像。譬如，对于亚马逊的 AWS 云环境，可以方便地使用其提供的系统映像（AMI）来完成部署。

  基于映像的最显著优势在于，能够在应用需要扩容的时候，更有效、迅速地扩容。原因在于，映像本身已经包括了操作系统和应用运行所需要的所有依赖，启动即可提供服务。而基于 war、rpm 包等的部署，扩容时通常需要先启动同构的节点，安装依赖，之后才能部署具体的 war 包或者 rpm 包。

- 容器部署

  利用容器技术，譬如 Docker，构建出的部署包也可以是一个映像。该映像能运行在任何装有 Docker 的环境中，有效地解决了开发与部署环境不一致的问题。同时，由于 Docker 是基于 Linux 容器的虚拟化技术，能够在一台机器上构建多个容器，因此也大大提高了节点的利用率。

所以，就微服务架构本身而言，如何有效地基于部署环境选择合适的部署方式，并最终完成自动化部署，是一个值得团队或者组织不断探讨和实践的过程。

11.2.6 运维

由于每个服务都是一个可以独立运行的业务单元，同时每个服务都运行在不同的独立节点上。因此，需要为服务建立独立的监控、告警、快速分析和定位问题的机制，我们将它们统一归纳为服务的运维。

- 监控

监控是整个运维环节中非常重要的一环。监控通常分为两类：系统监控与应用监控。系统监控关注服务运行所在节点的健康状况，譬如 CPU、内存、磁盘、网络等。应用监控则关注应用本身及其相关依赖的健康状况，譬如服务本身是否可用、其依赖的服务是否能正常访问等。

关于监控，目前业界已经有很多成熟的产品，譬如 Zabbix、NewRelic、Nagios 以及国内的 OneAPM 等。对于笔者参与的项目，服务节点的运行环境大都基于 AWS（使用 EC2、ELB 以及 ASG 等），因此使用 AWS 的 CloudWatch 作为系统监控工具的情况比较多。关于应用监控，通常使用 NewRelic 和 Nagios 作为监控工具。

- 告警

告警是运维环节另外一个非常重要的部分。我们知道，当系统出现异常时，通过监控能发现异常。这时候，通过合适的告警机制，则能及时、有效地通知相关责任人，做到早发现问题，早分析问题，早修复问题。由于每个服务都是独立的个体，因此针对不同的服务，都应该能提供有效的告警机制，确保当该服务出现异常时，能够准确有效地通知到相关责任人，并及时解决问题。

对于告警工具，业界较有名的是 PagerDuty，它支持多种提醒方式，譬如屏幕显示、电话呼叫、短信通知、电邮通知等，而且在无人应答时还会自动将提醒级别提高。除此之外，之前提到的常用的监控产品也能提供告警机制。

- 日志聚合

除此之外，日志聚合也是运维部分必不可少的一环。由于微服务架构本质上是基于分布式系统之上的软件应用架构方式，随着服务的增多、节点的增多，登录节点查看日志、分析日志的工作将会耗费更高的成本。通过日志聚合的方式，能有效将不同节点的日志聚合到集中的地方，便于分析和可视化。

目前，业界最著名的日志聚合工具是 Splunk 和 LogStash，不仅提供了有效的日志转发机制，还提供了很方便的报表和定制化视图。更多关于 Splunk 与 LogStash 的信息，请参考其官方网站。

11.3 小结

本章首先讲述了持续交付的概念及其核心，接着讨论了在微服务架构的实施过程中，如果建立基于服务的细粒度的持续交付流水线，应该考虑的因素。虽然微服务中的服务只是整个应用程序的一个业务单元，但作为一个可以独立发布、独立部署的个体，它必然也要遵循持续交付的机制和流程，包括开发、测试、集成、部署以及运维等，可谓是“麻雀虽小，五脏俱全”。通过搭建稳定的持续交付流水线，能够帮助团队频繁、稳定地交付服务。

第 12 章

微服务与轻量级通信机制

微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。在微服务架构中，服务和服务之间通信时，通常是通过轻量级的通信机制，实现彼此间的互通互联，互相协作。所谓轻量级通信机制，通常是指与语言无关、与平台无关的这类协议。通过轻量级通信机制，使服务与服务之间的协作变得更加简单、标准化。

本章，就和大家探讨一下，如何选择轻量级通信机制，完成服务和服务之间的协作。

12.1 同步通信与异步通信

12.1.1 概述

我们知道，微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。因此，微服务架构本质上是分布式系统。

相比传统的单机系统，由于服务和数据分布在不同的节点上，每次交互都需要跨节点运行，这使得网络成为微服务架构实施考虑的必要因素之一。这也意味着，对于服务节点之间的通信，我们首先需要考虑是采用同步通信方式还是异步通信方式。

12.1.2 同步通信与异步通信的选择

同步通信，是指当客户端（Consumer 或者消费者）发出请求后，在服务端（Provider 或者生产者）处理未结束前，客户端一直处于等待状态，直到最终获得对端的响应。

异步通信，则是指当客户端发出请求后，服务端或者第三方组件会先接受消息并应答，然后在合适的时间对请求进行处理。不过在处理的过程中，客户端不需要一直处于等待状态，图 12-1 所示。

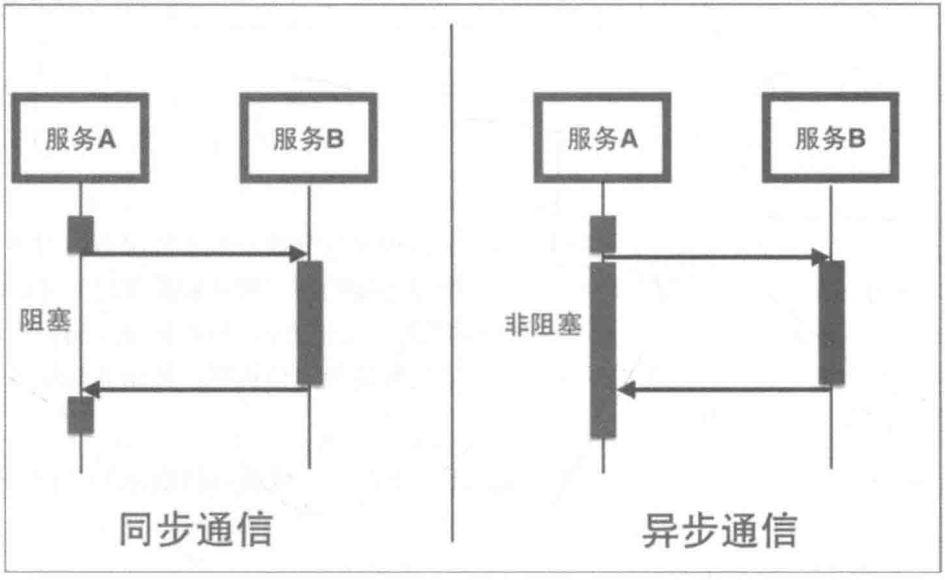


图 12-1 同步通信与异步通信

举个简单的例子，我去书店买书，如果是同步通信机制，店员会告诉我“不确定这本书是否有存货，您稍等，我先查一下”，然后去后台查（中间可能需要 5 分钟、10 分钟，或者 1 个小时），我一直在等待中，最终告诉我是否可以购买。如果是异步通信机制，则店员可能告诉我“不确定这本书是否有存货，我先查一下。您先忙，可以晚点过来查看或者等候通知”，然后我可以先去忙其他的事情，接到电话通知或者方便的时间再过来购买。

也就是说，同步通信的过程一般是发送请求，等待，接收响应并处理。整个过程实现简单，但会造成阻塞操作。而异步通信的过程则是发送请求后立即返回，不会造成阻塞，一般适用于耗时操作的处理。但异步通信在享受非阻塞的优势同时，也大大增加了功能实现的复杂度，并且当出现缺陷时，定位问题、调试问题的难度也更大。

12.2 远程调用 RPC

RPC（Remote Procedure Call），又称远程过程调用，是一种典型的分布式节点间同步通信的实现方式。远程过程调用采用客户端/服务器端的模式，请求的发起者是客户端，提供响应的是服务器端。

客户端通过客户代理存根（Stub），传递函数参数，向服务器端发起函数调用。服务器端通过服务器代理存根（Skeleton），接收到客户端的请求后，对请求进行处理，并在结束后向客户端返回响应，从而完成一次通信。

12.2.1 远程过程调用的核心

远程过程调用采用客户端/服务器端模式。请求的发起者是客户端，提供响应的是服务器端。客户端和服务器端的交互过程如图 12-2 所示。

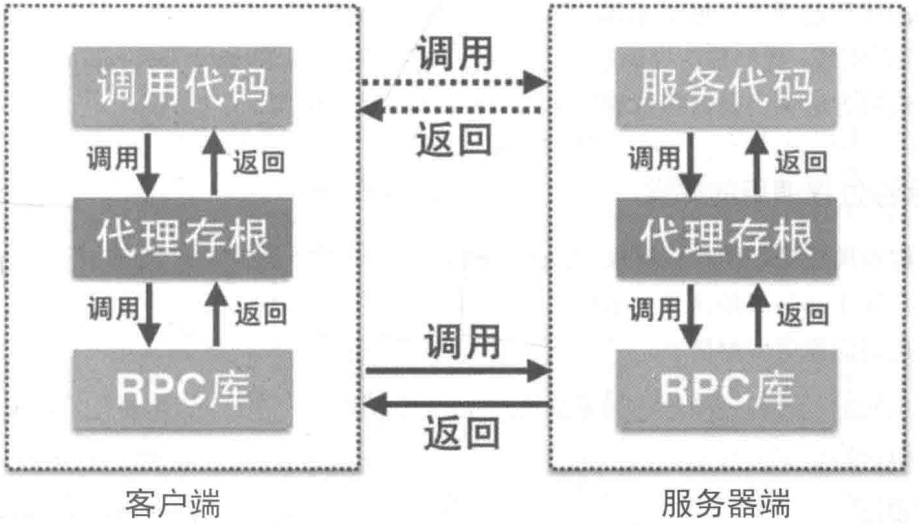


图 12-2 RPC 工作流程

1. 首先，客户端调用本地代理存根，发送请求到服务器端，等待应答信息。

2. 在服务器端，服务代理处于睡眠状态，直到客户端请求到达并将其唤醒。

3. 服务代理获得请求参数后，交由服务器端的服务代码对其进行处理。

4. 应用程序处理结束后，由服务代理向客户端发送应答，等待下一次请求。

5. 客户端代理存根接收应答信息，交给客户端的调用代码进行处理。

如上流程所示，远程过程调用的框架会定义一组代理存根（Stub 与 Skeleton），分别存在于客户端与服务器端环境中，为客户端与服务器端应用程序提供透明的调用和响应机制。

12.2.2 远程方法调用

传统的远程过程调用框架主要包括 Sun RPC（http://web.cs.wpi.edu/~rek/DCS/D04/SunRPC.html）、DCE/RPC（https://en.wikipedia.org/wiki/DCE/RPC），主要基于 C 语言实现，以函数调用为主。

随着面向对象语言的快速发展，序列化/反序列化等特性的诞生，基于不同语言的远程过程调用框架开始提供对象远程访问的功能。譬如 Java RMI、Thrift 或者 protocol buffers 等。同传统的远程过程调用框架相比，有一类框架能够允许客户端通过面向对象的调用方式，调用远端的实现，我们称这类调用为远程方法调用（Remote Method Invocation，RMI）。换句话说，远程方法调用是远程过程调用的一种面向对象的实现。

12.2.3 远程过程调用的弊端

远程过程调用通过使用代理存根（Stub/Skeleton）的方式，屏蔽了通信双方底层的调用细节，让客户端不必显式地区分当前代码级别的方法调用是本地调用还是远程调用，因此使分布式节点间的通信变得简单。

但是，虽然远程过程调用的调用机制屏蔽了调用的细节，简化了调用流程，但其也存在如下的弊端。

- **耦合度高**

大部分远程过程调用的实现机制，是依赖于编程语言或者特定平台的，因此限制了客户端和服务器端采用的技术，耦合度较高。一旦使用后，通信的双方就很难再切换到其他语言或者平台上。譬如，如果使用了 Java RMI 作为通信机制，就意味着对应的通信双方都必须运行在 JVM 平台上，这一点在很大程度上限制了将来技术的替换。Thrift 和 protocol buffers 支持多种语言，可以稍稍地弱化这个缺点。

- 灵活性差

远程过程调用依赖于编程语言以及特定平台，为了提高性能以及为客户端提供透明的调用机制，其传输的格式通常是二进制数据，因此需要客户端与服务器端提供序列化与反序列化的操作。这也就意味着，当通信两端的任一方发生变化时，会对另外一端造成影响。因此，灵活性较差。譬如，当使用 Java RMI 作为通信机制时，Java RMI 会在客户端与服务器端生成代理存根。在这种情况下，如果服务器端或者客户端的任何一方发生变化，都需要修改相应的对端，来保持代理存根的一致性。

远程过程调用，是一种典型的分布式节点间同步通信的实现方式。其为客户端提供了简洁、透明的调用接口，但同时由于其依赖于开发语言以及特定的平台，因此耦合度较高，并且灵活性不太好，并不适合作为一种轻量级的机制满足服务之间通信的要求。

12.3 REST

12.3.1 概述

REST（Representational State Transfer，表述性状态传递）是近几年使用较广泛的分布式节点间同步通信的实现方式。REST 从语义层面将响应结果定义为资源，并使用 HTTP 的标准动词映射为对资源的操作，形成了一种以资源为核心、以 HTTP 为操作方式的，与语言无关、平台无关的服务间的通信机制。这也是目前微服务架构下，服务间通信较常用的一种方式。

REST，不仅仅是同步

实际上，将REST仅列为分布式节点间的同步通信方式并不恰当。REST是由Roy Fielding博士在2000年提出的一种软件架构风格，也是近几年来，API设计和实现所遵循的最佳原则之一。笔者在这里之所以将REST列为同步通信的实现方式，主要是因为REST是基于HTTP的，而HTTP的通信是同步过程（虽然可以使用AJAX等方式将通信的过程异步化，但其本质上还是同步的过程），所以可以认为REST的通信过程是同步的。

不过，也可以将REST的架构风格应用于异步通信过程中，譬如在后台任务系统中，可以使用REST风格作为服务调用的协议。

软件架构风格

软件架构风格是一种研究和评价软件架构设计的方法，它是比架构更加抽象的概念。通常，软件架构风格是由一组相互协作的约束来定义的，譬如软件的交互协议、交互方式以及交互的内容等。

12.3.2 REST 的核心

要深入理解 REST，需要先理解 REST 的四个关键部分，如图 12-3 所示。

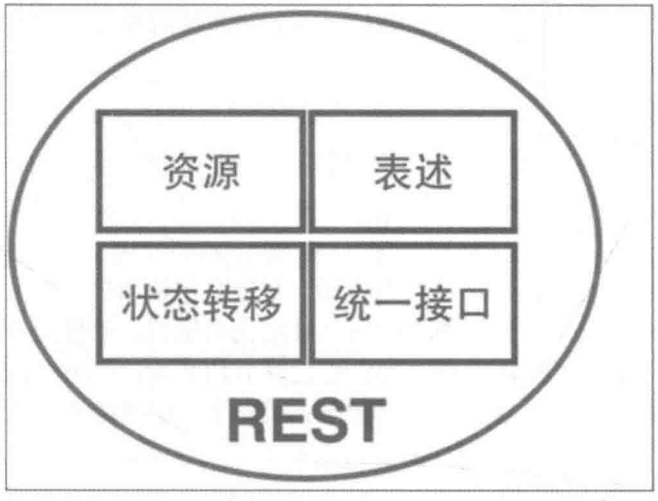


图 12-3 REST 的核心

资源

资源（Resource）是一个抽象的概念，是指对某类信息实体的抽象。那什么是实体？实体是指服务器端需要处理的具体信息，它可以是一段文本、一张图片、一首歌曲，也可以是文件系统中的一个文件、数据库中的一张表等。

与面向对象设计的概念类似，资源通常以名词为核心来定义。每个资源对应一个特定的 URI 作为标识。对某个资源感兴趣的客户端应用，可以访问资源的 URI 与其进行交互。

表述

资源的表述（Representation）是对资源在某个特定时刻的状态的描述。我们知道，资源是一种信息实体，实体在客户端与服务器端进行信息交换时，可以有多种表现形式。譬如，文本可以用 TXT 格式表现，也可以用 HTML 格式、XML 格式、JSON 格式表现，甚至可以采用二进制格式；图片可以用 JPG 格式表现，也可以用 PNG 格式表现。这种资源的表述格式可以在客户端与服务器端通过请求-响应的协商机制来确定。

实际上，URI 仅代表资源的实体，并不代表它的表述。譬如，有些 URL 最后的.html 后缀名并不属于表述范畴，表述应该在 HTTP 请求的头信息中用 Accept 和 Content-Type 字段指定，这两个字段才是对“表述”的描述。

状态转移

状态转移（State Transfer）是指在客户端同服务器端交互的过程中，客户端能够通过资源的表述，实现操作资源的目的。

当我们使用浏览器访问一个网站时，就代表了客户端和服务器端的一个交互过程。在这个过程中，必然会涉及数据或者状态的变化。但我们知道，HTTP 是一个无状态协议，这意味着，所有的状态都保存在服务器端。因此，如果客户端想要操作资源，必须通过某种手段，让服务器端发生状态的转移。而这种转移是建立在资源的表述之上的，所以通常将其称为表述层状态转移。

统一接口（Uniform Interface）

客户端操作资源的方式，通常是基于 HTTP 的 4 个动词（Verb）：GET、POST、PUT、DELETE。它们分别对应 4 种资源的操作方式：

- GET 用来获取资源。
- POST 用来新建资源。
- PUT 用来更新资源。
- DELETE 用来销毁资源。

因为客户端是通过 HTTP 的这 4 个动词操作资源的，也就意味着，不管请求的 URI 是什么，请求的资源有什么不同，但操作资源的接口都是统一的。

12.3.3 REST 的优势

通过资源表述、状态转移以及统一接口，REST 将客户端的请求、服务器端的响应基于资源联系起来，逐渐形成了一种以资源为核心、以 HTTP 为操作方式的，与语言无关、平台无关的通信机制。

同时，由于 HTTP 本身的无状态性，使用 REST，能够有效保持服务/应用的无状态性，利于将来的水平伸缩。

12.3.4 REST 的不足

随着团队或者组织业务的不断增长，服务器端响应内容复杂度的增加，REST 的使用面临如下两个挑战：

- 如何标准化资源结构
- 如何处理资源的相关链接

如何标准化资源结构

我们知道，使用 REST 可以将业务场景的具体信息定义为资源，并基于 JSON 或者 XML 返回给客户端。随着业务的不断增长，逻辑的增加，服务器端对内容的响应结构会变得越来越复杂（所谓响应结构，是指服务器端的响应内容结构，即资源的结构）。

REST 作为指导性的原则，并没有定义服务器端响应结构应该遵循什么标准。这也就意味着，在企业内部，不同的部门，不同的开发小组，对同一类资源，所定义的结构可能不尽相同。譬如，如下是服务器端对客户端获取产品请求的响应结果，两种结构都是合理的，但存在着明显的差异：

响应结构一

```
{
  "name":"Programming Ruby",
  "category": "Book",
  "price":89.00,
  "ref": "http://xxxx/products/12",
  "created_at": "2015-05-01 10:00:00",
  "updated_at": "2015-06-01 11:00:00"
}
```

响应结构二

```
{
  "basic_info":{
    "name":"Programming Ruby",
    "price":89.00,
    "category":"Book"
  },

  "ref":{
    "self": "http://xxxx/product/12",
     "list": "http://xxxx/products&page=1"
  },

  "timestamp":{
    "created_at": "2015-05-01 10:00:00",
    "updated_at": "2015-06-01 11:00:00"
  }
}
```

因此，如何定义一套标准的资源响应结构，成为服务数量增多后使用 REST 面临的一个挑战。

如何有效处理相关资源的链接

大部分 REST 的实现，都是基于 JSON 作为传输格式，但 JSON 最大的遗憾，正如 W3C 所描述的：

"JSON has no built-in support for hyperlinks, which are a fundamental building block on the Web."

JSON 最大的遗憾在于没有对超链接处理做内置的支持，而这部分却恰恰是 Web 的基石。这带来的潜在问题是，对于调用接口的客户端而言，每次需要查看相关的接口文档，才能了解如何修改资源的状态，或者获取相关联资源的信息。譬如，某些社交系统可能会提供类似如下的接口文档列表，来帮助客户端了解如何使用其暴露的接口。

```
https://api.example.com/users/1234567890      GET 获取用户明细
```

```
https://api.example.com/users/[ID]/friends    GET 获取用户相关的好友
https://api.example.com/users/[ID]/posts      GET 获取用户相关的文章
```

如上代码所示，获取某用户信息的请求和响应如下：

```
GET https://api.example.com/user/1234567890
```

```
{
  "id": "1234567890",
  "name": “张三”,
   "age" : 25,
   "group":[
       {"name": "Fishing", "created_at": "2015-05-20 20:00:00"},
       {"name": "Cooking", "created_at": "2015-05-10 20:00:00"}
   ],
}
```

而获取某用户好友列表的请求如下：

```
GET https://api.example.com/user/1234567890/friends
```

```
[
   {
      "id": "1895638109",
      "name": "李四",
      "age" : 30,
   },
   {
      "id": "8371023509",
      "name": "马丁",
      "age" : 40,
   }
]
```

实际上，对于某用户而言，获取其好友列表的资源链接应该是和该用户的链接资源相关的，即用户资源的 JSON 中应该能够自解释这种资源的关联关系。

因此，如何用有效的方式管理 REST 中相关资源的依赖以及链接，是否能够将这种方式标准化，成为服务数量增多后使用 REST 面临的一个挑战。

其他需要考虑的因素

- 性能

  由于 REST 是基于 HTTP 之上的协议，而 HTTP 本身是一个使用广泛的、基于 TCP/IP 的应用层协议，因此 REST 并不是低延时通信的最好选择。也就是说，对于服务之间对低延时要求的场景，可能需要选择不同的底层协议，如 UDP（User Datagram Protocol）来达到希望的性能，或者其他 RPC 框架。

- 开发成本

  使用 REST，其传输格式通常是 XML 或者 JSON 的文本格式。这也就意味着，在享受平台无关、语言无关优势的同时，团队需要编写更多的代码来解析文本格式的协议。不过，目前已经有很多成熟的库帮助我们来做类似的事情，譬如 Java 下解析 JSON 的 JSON-lib、org.json 以及 Ruby 下的 Json Gem。

更多关于 REST 的细节，读者可以参考 *REST in Practice* 一书。

12.3.5 本节小结

通过 REST，我们从语义层面将响应结果定义为资源，并使用 HTTP 的标准动词映射为对资源的操作，形成了一种以资源为核心、以 HTTP 为操作方式的，与语言无关、平台无关的服务间的通信机制。这也是目前微服务架构下，服务间通信较常用的一种方式。

不过，对于 REST，如何标准化 REST 的资源结构，如何有效处理 REST 中相关资源的关联和链接，是一个随着服务数量增多要考虑的问题。另外，对于服务间通信低延时要求的场景，也需要考虑 REST 的适用性。

12.4 HAL

12.4.1 概述

HAL（Hypertext Application Language）是一种轻量级超文本应用描述协议。HAL 的实现基于 REST，并有效地解决了 REST 中资源结构标准化和如何有效定义资源链接的问题。

目前，越来越多的企业和组织开始使用 HAL 提供标准化的服务接口，譬如：

- 亚马逊的 APP Stream API（http://docs.aws.amazon.com/appstream/latest/developerguide/rest-api-application.html）。
- SMXEmail API（https://smxemail.com）。
- 牛津大学官方数据 API（http://api.m.ox.ac.uk/browser/#/）。

更多案例请参考 HAL（http://stateless.co/hal_specification.html）。

12.4.2 HAL 的核心

在 HAL 中，任何服务器端的响应都被定义成一种资源（Resource），这是遵循 REST 原则对资源的定义的。

同 REST 不同的是，在每个资源中，HAL 又将其分成了如下三个标准的部分，如图 12-4 所示。

- 状态（State）
- 链接（Links）
- 子资源（Embedded Resource）

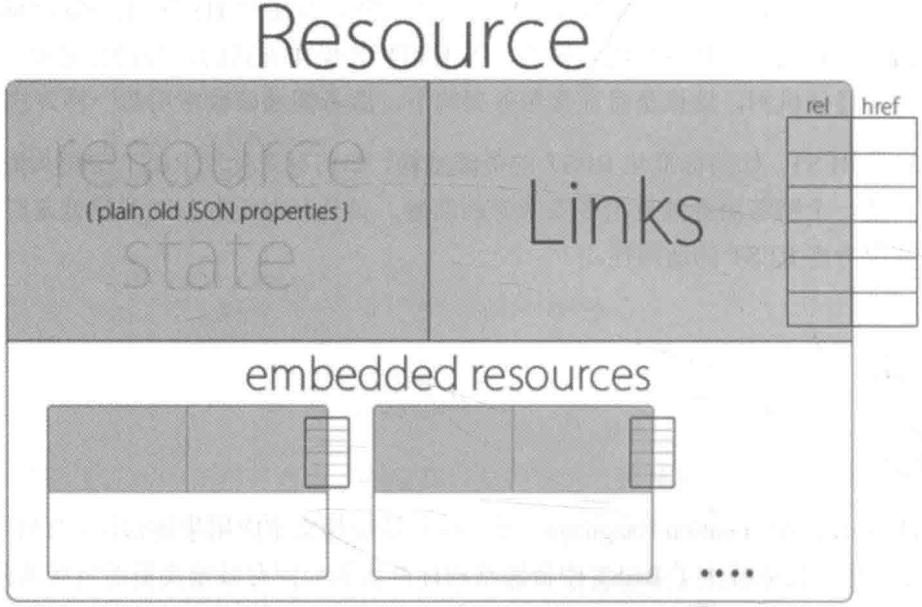


图 12-4 HAL 的三个标准部分

状态

状态（State）通常是指资源本身固有的属性。譬如，对于商品列表资源的响应而言，其状态的定义可能如下所示：

```
{
  ......
  "count": 20,
  "total": 498,
  ......
}
```

如上代码所示，其状态主要包括：

- 每页的数量（count）
- 所有商品的数量（total）

对于服务器端商品明细资源的响应而言，其状态的定义可能如下所示：

```
{
  ......
  "id":   "9ff3c836",
  "name":  "Building microservice",
  "price" : 99.00,
  "created_at": "2015-06-01 19:00:00",
  ......
}
```

如上代码所示，该资源的状态主要包括：

- 编号（id）
- 名称（name）
- 价格（price）
- 创建时间（created_at）

链接

链接（Links）定义了与当前资源相关的一组资源的链接的集合。主要由以下几部分组成：

- 链接名称

- 目标 URI
- 访问 URI 的参数

譬如，对商品列表资源的响应而言，其链接的定义如下所示：

```
{
  "_links": {
     "self": {
        "href": "http://example.org/api/products?page=3"
     },
     "first": {
        "href": "http://example.org/api/products"
     },
     "prev": {
        "href": "http://example.org/api/products?page=2"
     },
     "next": {
        "href": "http://example.org/api/products?page=4"
     },
     "last": {
        "href": "http://example.org/api/products?page=25"
     }
    }
  }
}
```

如上代码所示，通常列表页返回的响应都是分页的结果，因此需要在资源中定义上一页、下一页、第一页以及最后一页等相关链接。

类似的，对于商品明细资源的响应而言，其链接的定义可能如下所示：

```
{
  "_links": {
    "self": {
      "href": "http://example.org/api/products/9ff3c836"
    }
  },
  ......
}
```

子资源

子资源（Embedded Resource）描述在当前资源的内部，其嵌套资源的定义。

譬如，对商品列表资源的响应而言，其子资源的定义可能如下所示：

```
{
  ......
  "_embedded": {
    "products": [
    {
      "_links": {......},
      state:   {......},
    },
    {
      "_links": {......},
      state:   {......},
    },
    ......
    ]
  }
}
```

如上代码所示，在列表页返回的资源中，通常将具体的列表中的每一项定义在_embedded 中。类似的，对于商品明细资源的响应而言，其子资源的定义可能如下所示：

```
{
  "_links": {......},
  "state": {......},

  "_embedded": {
    "category": {
      "_links": {......},
      "state":  {......}
    },
  }
}
```

如上代码所示，在商品明细页返回的资源中，将分类的相关信息作为子资源显示出来。

12.4.3 HAL 浏览器

使用 HAL，还有一个很大的优势，那就是可以使用 HAL 浏览器（HAL Browser）来可视化资源的信息。我们知道，HAL 将资源分成三个基本的部分：状态、链接和子资源。基于这个标准的结构，HAL 浏览器能够将资源的每一部分，通过可视化的方式显示出来，如图 12-5 所示。

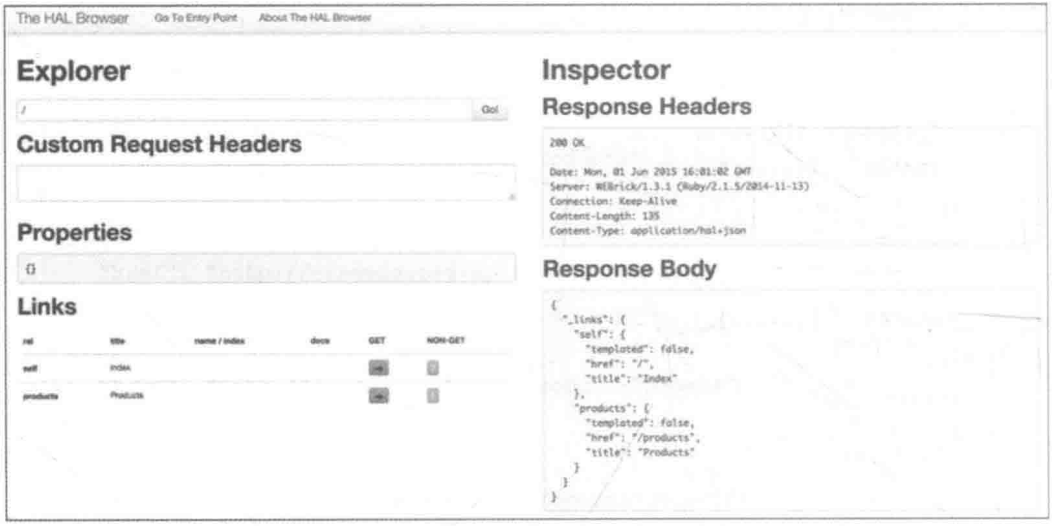


图 12-5　使用 HAL 浏览器可视化资源的信息

如图 12-5 所示，左边的可视化部分为 HAL Browser 的展示，其主要包括：

- 请求的 URL
- 请求的 Header
- 属性部分（状态部分）
- 链接部分

右边为对应的响应内容，其主要包括：

- 响应 Header
- 响应内容

在笔者参与的微服务项目中，大部分都会使用 HAL Browser 为调用者提供 API 功能的展示。当调用者使用浏览器访问该服务的 URL 时，显示的第一个页面就是 HAL Browser 展示的关于资源部分的显示。

在实践中，我们将对 HAL Browser 的使用封装成一个 Ruby 的 Gem，代码类似如下所示：

```
def call(env)
  request = Rack::Request.new(env)
  if match?(request)
    return [303, {'Location' => hal_browser_url_with_request(request)}, []]
  end
  ......
end
```

当 Rack 请求进来时，Gem 会先检查该请求是否期望被重定向到 HAL Browser 的显示页面。match 匹配的逻辑如下所示：

```
def match?(request)
  request.get? && is_html?(request) && include?(request)
end
```

如果匹配，则请求被重定向到 HAL Browser 的显示页面：

```
def hal_browser_url_with_request(request)
  url = URI.parse('/hal-browser/browser.html')
  url.fragment = url_path_with_query_string(request)
  url.to_s
end
```

在代码中使用该 Gem 时，只需要配置 include 部分，当访问相应的 URL 时会被重定向到 HAL Browser 的页面。

```
use HalBrowser::Forward, :include => ['/api']
```

更多关于 HAL Browser 的信息以及使用，请参考 HAL Browser 的 Github 代码库（https://github.com/mikekelly/hal-browser）。

通过使用 HAL，我们在享受 REST 带来的简洁性、便利性的同时，也标准化了服务器端响应的结构，并能有效处理相关资源的链接集合；同时，通过使用 HAL Browser，能用可视化的方式帮助客户端了解服务所提供的接口。

12.5 消息队列

消息队列（Message Queue）是一种处理节点之间异步通信的实现方式。发送消息的一

端称为发布者，接收消息的一端称为消费者。通过消息队列，发布者将消息放入队列中保存，然后消费者会在将来某个时间点获取消息并处理。消息队列使消息的发布者和消费者不需要同时交互，从而达到异步通信的效果。

12.5.1 核心部分

通常，消息队列会提供如下的核心部分，来保障消息的传递，如图 12-6 所示。

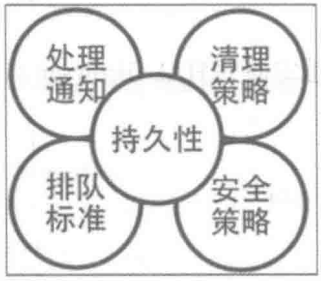


图 12-6 消息队列的核心特征

- 持久性

消息可能被保存在内存中、写入到磁盘，或者提交到数据库。

- 排队标准

消息队列会指定相应的标准和算法，保障消息进入或者出队列的优先级。譬如最常见的 FIFO（先进先出）算法。

- 安全策略

消息队列提供特定的安全策略，决定哪些接收者能够访问或者获取消息。

- 清理策略

消息队列会为处理过的消息提供清理策略，保障消息的有效清理机制。

- 处理通知

消息队列提供某种通知机制，帮助消息发布者知道何时部分或全部接收者收到了消息。

消息队列的存在，提供了可靠的消息传递机制以及清理策略，帮助发布者和消费者能够借助消息队列，完成异步通信。

12.5.2 访问方式

对于消息队列的访问，一般存在如下两种方式。

- 拉模式

拉模式（Pull Mode）要求消费者定期检查队列上的消息。如果找到一条匹配的消息，消费者会从队列中获取该消息，并进行处理。如果没有找到，则消费者会在指定的时间段内再次尝试。通常在拉模式下，一般存在一个发布者和一个消费者，如图 12-7 所示。

- 推模式

推模式（Push Mode）是每当发布者将消息添加到队列中时，会通过某种机制通知消费者。因此消费者可知道消息队列的变化情况，这意味着数据的处理更及时，但同时也意味着发布者、消费者以及队列必须依赖语言或者平台的某些特性，做更多的开发和配置工作。通常在推模式下，一般存在多个消费者，也称他们为订阅者，如图 12-7 所示。

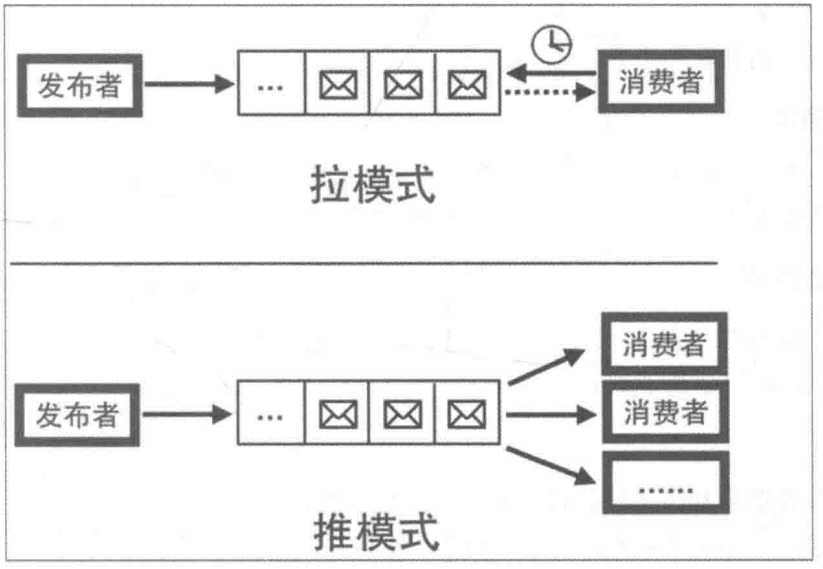


图 12-7 消息队列的模式

消息队列解耦了消息发布者和消费者之间的关系，使得二者可以借助于队列的方式完成异步通信。业界常用的消息队列有 RabbitMQ（https://www.rabbitmq.com/）、ActiveMQ（http://activemq.apache.org/）以及 ZeroMQ（http://zeromq.org/）等。

12.5.3 消息队列的优缺点

对于微服务架构的系统而言，使用消息队列的优点非常明显，主要包括：

- 服务间解耦

消息队列使服务之间并不需要直接通信，所有的通信都基于队列中的消息完成，服务和服务之间的耦合度降低。

- 异步通信

通信的双方并不需要同时在线，发布者可以将消息保存在队列中，消费者可以在合适的时间段内获取消息。

- 消息的持久化以及恢复支持

对于消息队列中保存的消息，队列可以将其持久化。同时，如果出现异常，队列一般也支持消息的恢复机制。

除了如上的优点，消息队列也存在一些缺点，主要包括：

- 实现复杂度增加

消息队列实际上是一种异步通信方式。因此，从开发、测试、调试以及问题的定位上，都会引入异步通信所带来的复杂度。

- 平台或者协议依赖

对于某些功能，譬如实时推送、路由等，对语言、平台的依赖性较强。譬如 RabbitMQ 基于 AMQP 协议，ActiveMQ 基于 JMS 协议等。

- 维护成本高

使用了消息队列就意味着该消息队列成为整个系统基础设施重要的一部分。需要考虑随着系统复杂度的增加，如何保障消息队列本身的可用性以及可扩展性。

通过使用消息队列，不同服务之间不仅能有效地完成异步通信，同时也大大降低了服务间的耦合度。不过，在使用消息队列实现异步通信的同时，还应该考虑其维护成本以及对某些平台或者语言的依赖性。

12.6 后台任务处理系统

当构建复杂的分布式系统时，消息队列是常用的异步通信方式。

但实际上，消息队列的存在会增加整个系统的开发及维护成本，原因主要有：

- 学习曲线陡

对于目前成熟的消息队列产品而言，譬如 ActiveMQ、RabbitMQ 等，其提供的功能较为全面，解决问题的场景较多，因此需要花费的学习成本较高。

- 维护成本高

消息队列作为分布式系统的一部分，是独立存在的。也就意味着，消息队列需要被独立安装和配置；同时，消息队列也需要被部署和监控，以保证其能够水平伸缩并保持高可用性。

因此，在一些相对简单的异步场景中，使用消息队列可能显得略微复杂。譬如，对一个用户注册的场景，当用户单击注册按钮后，希望后台服务进行处理并发送欢迎电子邮件，但同时要求该操作不会影响用户在页面上的其他操作。对于这个场景，如果需要安装 ActiveMQ、RabbitMQ，定义消息的发布者和消费者，显然太过复杂，因此可以使用后台任务处理系统。常用的后台任务处理系统有 Resque、Sidekiq 以及 Delayed_job 等。

12.6.1 核心部分

通常，后台任务处理系统主要包括如下几部分，如图 12-8 所示。

- 任务
- 队列
- 执行器
- 定时器

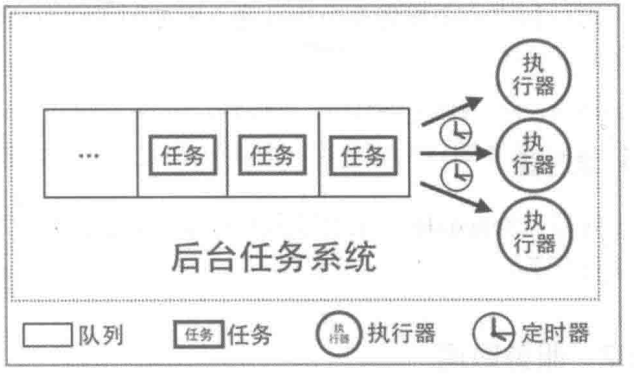


图 12-8 后台任务处理系统

- 任务

任务（Task）是指后台任务处理系统中可执行的最小单元。通常，后台任务处理系统会对任务有一个定义，如实现某规定的执行函数、定义任务存储的队列等；然后，任务会被发布者通过接口或者 SDK 的方式存入队列中，等待执行。譬如，在 Resque 中，每个任务需要实现 perform 方法，并且通过给@queue 赋值，定义存储任务的队列名称，类似代码如下所示：

```
class EmailSendingTask
  @queue = "email_queue"

  def self.perform(service)
    # sending email
  end
end
```

- 队列

队列（Queue）主要用于存储任务，并提供任务执行失败后的错误处理机制，譬如失败重试、任务清理等，通常我们也称队列为任务队列。每个任务队列都有一个全局唯一的名称作为标识。目前大多后台任务处理系统通常都采用 Redis 作为队列的实现机制，譬如像 Resque、Sidekiq 等。

- 执行器

执行器（Worker）主要负责从队列中获取任务，并执行任务。对于后台任务系统而言，运行时可以指定一个或者多个执行器，譬如如下代码启动 Resque，并配置了两个执行器执行任务。

```
resque: env TERM_CHILD=1 COUNT=2 bundle exec rake resque:workers
```

- 定时器

定时器（Scheduler）主要设置执行器运行的周期，譬如每 1 分钟或者 3 分钟运行执行器并执行任务。

12.6.2 服务回调

在笔者的实践中，为了保持轻量级的通信机制，使不同的服务之间能通过后台任务系

统执行异步通信，通常会保持任务的轻量级，不会在任务中做过多的逻辑，而是尽量做到由任务回调具体的服务来完成交互，类似流程如图 12.9 所示。

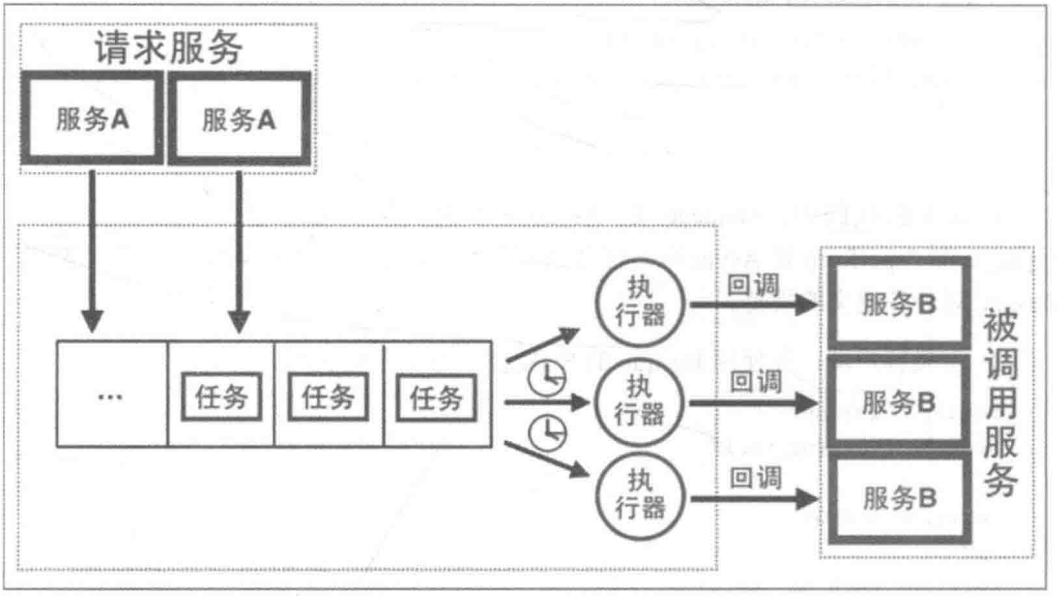


图 12-9 后台任务处理系统与服务回调

- 将服务 A（请求者）与服务 B（响应者）交互的部分，定义成任务。
- 服务 A 调用后台任务处理系统的 API 或者 SDK，将任务存入队列。
- 执行器根据调度器设置的周期，定期取出任务并执行。
- 任务执行时，回调服务 B 提供的接口。

12.6.3 一个例子

接下来，笔者就以 Resque 作为后台任务系统，实现一个异步回调服务的例子。

1. 首先，新建 resque-demo 目录并创建 Gemfile，添加如下代码：

```
source 'https://rubygems.org'
gem 'resque'
gem 'rake'
```

2. 接下来，定义任务 AsyncTask，代码如下所示：

```
class AsyncTask
  @queue = "async_tasks"
  def self.perform(service)
    sleep 3 # fake analysis here
    puts "Finished communicate with service #{service}"
  end
end
```

在如上的代码中，@queue 是 AsyncTask 的实例变量，定义了任务队列的名称为 async_tasks；perform 是 AsyncTask 任务的执行函数，定义了该任务需要完成的逻辑，这是 Resque 对任务定义的规范。

3. 定义客户端，并使用 Resque 的 SDK 将任务保存进队列，代码如下所示：

```
require "resque"
require "./async_task"

services = ARGV

services.each do |service|
   # This is where we would enqueue
   Resque.enqueue(AsyncTask, service)
end
```

4. 修改 Rakefile 引用 resque，代码如下所示：

```
require "./async_task"
require "resque/tasks"
```

接下来，就可以启动 Resque，代码如下所示：

```
$> bundle exec rake resque:work QUEUE=async_tasks
$> ruby run_async_handler.rb products customers contracts
$> resque-web
```

在上面的代码中，第一行启动 Resque，并指定执行器监听队列的名称为 async_tasks。

第二行运行客户端，并传入参数 products、customers 以及 contracts，表明任务执行时，会回调这三个服务。

最后一行启动 resque 的 Web 管理界面，帮助团队通过可视化的界面了解当前后台任务

的执行情况，执行结果如图 12-10 所示。

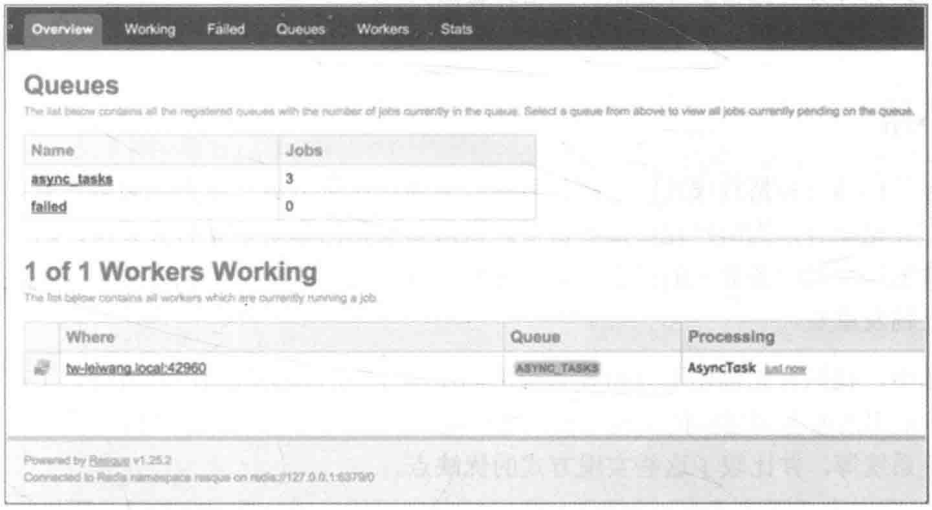


图 12-10　使用 resque

12.6.4　后台任务与微服务

对于微服务架构的系统而言，使用后台任务系统的优点很明显，主要包括如下几项。

- 轻量级通信机制

请求服务通过定义任务，将交互的请求存在任务队列中，执行器根据定时器的配置，从任务队列中取出任务并执行，再回调相关的服务。在整个交互过程中，后台任务系统只是承担了定时触发回调请求的责任，具体的实现均由被调用服务完成，从开发、测试以及问题定位和调试的角度而言，可以更集中在被调用服务本身。因此，相比消息队列，后台任务系统提供了更轻量级的方式，完成服务间的异步通信。

- 维护成本低

通常，后台任务系统的开发和维护成本比起消息队列要低得多。譬如 Resque、Sidekiq 等，通过 Gem 安装的方式就能快速安装后台任务系统的环境。

- SDK 及 API 支持

通常，后台任务系统会提供不同语言版本的 SDK 及 API，供请求服务方以低成本的方式将任务存储到任务队列中。

通过使用后台任务系统，使不同服务之间能够有效地完成业务异步通信。相比消息队列的方式，后台任务系统更简洁易用，也更轻量级。

12.7 小结

微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。微服务架构的本质是分布式系统，随着系统复杂度的增加以及微服务数量的增多，如何选择轻量级通信机制、完成服务和服务之间的协作和交互越发重要。

在本章中，我们首先描述了分布式系统中同步与异步的通信机制，然后讨论了在笔者的实践中，常用的服务间通信的实现方式，包括远程过程调用、REST、HAL、消息队列以及后台任务系统等，并比较了这些实现方式的优缺点。

最后，笔者制作了表 12-1，希望能为读者在选择服务间的通信机制时，提供一定的参考。

表 12-1 服务间的通信机制

	远程方法调用	REST	HAL	消息队列	后台任务系统
通信方式	同步通信	同步或异步通信	同步或异步通信	异步通信	异步通信
平台依赖性	强	平台无关	平台无关	强	弱
语言支持	好	语言无关	语言无关	好	中
学习成本	高	低	低	高	低
维护成本	高	低	低	高	低

第 13 章

微服务与测试

在微服务架构中，单一的应用程序被划分成一组服务，服务之间互相协调、互相配合，为用户提供最终的价值。在微服务带来灵活性的同时，随着服务数量的增多，不仅服务自身测试的复杂度增加，而且服务之间协作的测试成本也随之增加。如何才能做到让团队充满信心，既保障服务自身能有效工作，又保障服务之间能够正确地协同工作。

本章中，我们将讨论微服务的测试。

13.1 微服务的结构

当一个服务接收到请求时，通常需要在内部经过多个部分相互协调处理之后，形成一个最终响应并返回给请求者。

譬如在线电商系统，当用户查看某件商品时，商品服务会从数据库或者缓存中获取相关的详细信息，同时会从库存服务中获取库存信息，并且会从定价服务中获取最新定价信息等，然后根据相关的业务逻辑进行处理，并最终返回响应。

通常，对一个微服务而言，其内部的构成可能包括如下几个部分，如图 13-1 所示。

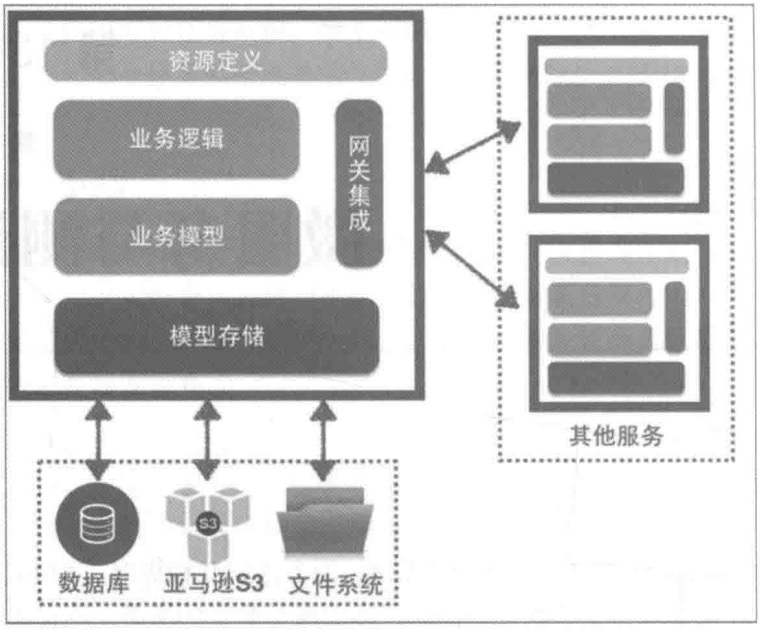


图 13-1　微服务的结构

- **业务模型**

业务模型（Domain）是对业务领域实体的抽象，譬如电商系统中的订单，社交系统中的用户以及 CRM 中的产品等。业界已经存在一些成熟的方法论，譬如面向对象设计、SOLID 原则、领域驱动设计等，能帮助我们有效地将业务领域的实体抽象成业务模型。

- **业务逻辑**

业务逻辑（Service/Business Logic）是对业务行为的抽象，也是如何基于业务模型，实现核心业务价值的部分。譬如，电商系统中用户如何下单，下单后的支付、配送，或者社交系统中用户状态的更新、信息分享等，都属于业务逻辑部分。

- **模型存储**

模型存储（Repository）关注如何存储或者获取业务数据。应用场景不同，存储或者获取数据的类型也存在很大差异。譬如关系型数据库适合存储结构化信息，NoSQL 适合存储非结构化信息，文件系统适合存储文本信息，云存储（譬如亚马逊的 S3 等）适合存放静态信息，如图片、CSS、JavaScript 等。当然，也有可能是调用其他服务来存储或者获取业务数据。

模型存储模式最早由埃里克·伊文在他的《领域驱动设计》一书中提到，其最大优势在于屏蔽了数据存储和获取的实现细节，让调用者更关注接口而非实现。更多关于领域驱动设计的细节，请参考《领域驱动设计》一书。

- 资源定义

资源定义（Resource）是指当请求端与服务端交互时，服务端响应对业务模型的表现形式。通常包括两部分。

  - 表述内容
    表述内容，指将业务模型转化成资源的内容部分。譬如在业务模型中，当订单表述为资源后，其表述内容包括订单编号、订单价格、创建时间等。类似的，业务模型中的用户表述为资源后，其表述内容包括用户姓名、联系方式以及用户当前状态等。
  - 表述格式
    表述格式，指表述内容的展现形式。譬如 XML 格式、JSON 格式，或者其他格式等。

当确定了表述内容与表述格式，业务模型就能够被清晰地定义成消费者使用的资源。

- 网关集成

网关集成（Gateway/Http Client）部分负责与其他服务间的协作，通过其他服务暴露的接口，获取数据或者提交数据。从某种意义上来看，其功能与模型存储的功能有相似的地方，都涉及业务数据的存取。因此，在笔者的实践中，有时也会将该部分合并到模型存储部分。

13.2 微服务的测试策略

与传统的测试类型相似，微服务架构中的测试类型主要包括：

- 单元测试
- 接口测试
- 集成测试
- 组件测试
- 端到端测试

由于微服务架构本质是基于分布式的系统，同传统的单块架构系统有较大的区别，因此测试策略上也与传统的单块架构有一定区别。另外，同传统的单块架构相比，微服务在模块划分和业务隔离上更加细致，对自动化测试、自动化运维的要求较高。因此，需要根据微服务本身的特点，合理地定义测试策略，找到有效的测试方式。

在测试金字塔理论中，对于一个系统的各种测试，其侧重点不一样，付出的成本以及带来的收益也不一样，如图 13-2 所示。

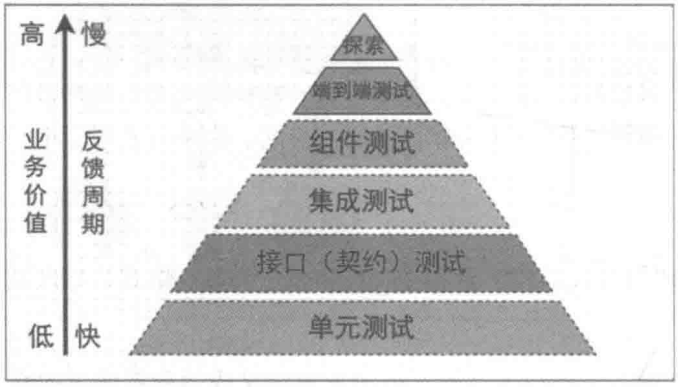


图 13-2　测试金字塔

在金字塔的顶端，是探索性测试。探索性测试并没有具体的测试方法，通常是团队成员基于对系统的理解以及基于现有测试无法有效覆盖的部分，做出的系统性的验证。譬如跨浏览器的测试，或者一些视觉效果的测试。因为这类功能变更较频繁，而且全部实现自动化成本较高。因此，小范围的探索性测试还是比较有效的。实际上，探索性测试强调测试人员的主观能动性，其通常不太容易通过自动化的方式实现，更多的是由团队成员手动完成。因此，探索性测试的成本最高，自动化难度大，测试以手动执行为主，同时还需要根据具体的场景改变策略。另外，探索性测试的反馈周期也是最慢的。

在金字塔的底部，是单元测试。单元测试是针对程序单元进行正确性的检验。理论上，程序单元是指应用中最小的可测部分，譬如函数、代码片段等。通常，单元测试都能通过自动化的方式运行。常用的单元测试框架有 Java 语言的 JUnit，Ruby 语言的 minitest 或者 RSpec，以及其他语言的 XUnit 系列。单元测试的实现成本低，运行效率高，能够使用工具或脚本完全自动化运行。另外，单元测试的反馈周期快，当单元测试失败后，能够很快发现，并且能较容易地找到出错的地方并修正。

在金字塔的中间部分，自底向上还包括接口（契约）测试、集成测试、组件测试以及端到端测试。这些测试各自的侧重点不同，使用的方法、技术和工具也不尽相同。譬如接口测试通常使用 Pact、Pacto 等框架完成，其运行所耗费的时间非常短。端到端测试通常采用 BDD 的方式描述测试用例，这类工具包括 JBehave、Cucumber 等，而组件测试则需要考虑是进程内的组件测试还是进程外的组件测试。

总体而言，在测试金字塔中，从底部往顶部移动，业务价值的比重逐渐增加，换句话说，越往顶部移动，测试越能体现业务价值，但相应的测试成本也在增加。而从顶部往底部移动，测试的效率逐渐提高，反馈周期逐渐缩短，执行效率更高。

通过测试金字塔，能帮助组织或者团队有效地制定测试策略。通过组合不同类型的测试，能提高测试覆盖率和测试效率并降低测试成本。

本节主要描述了常见的测试类型以及测试金字塔模型。通过测试金字塔模型，能帮助团队根据不同的场景，找到测试的平衡点，并制定合适的测试策略。接下来的几节，我们将详细了解微服务架构中各种类型的测试的侧重点。

13.3 微服务的单元测试

13.3.1 单元测试综述

单元测试，又称 Unit Testing，是针对程序单元的正确性进行的检验。理论上，程序单元是指应用中最小的可测试部件。在面向对象的设计与实现中，最小的可测试部件通常指的是类。但实际上，单元可以是类本身，也可以是一组相关的类的集合。不过，被测试的单元粒度越小，越容易测试。

对于被测单元而言，根据其复杂度、职责（以状态为主还是行为为主），或者其依赖部分的构建成本、协作成本等因素的不同，通常存在两种不同的单元测试，如图 13-3 所示。

图 13-3　单元测试及依赖

- 被测单元依赖于模拟部分

在这种情况下，当对被测单元进行单元测试时，通常会使用模拟的机制，模拟被测单元与其相关的依赖，使测试的焦点关注在被测单元的行为上。譬如，被测单元是否调用了其依赖类的某个特定方法，调用的次数是否符合期望。常用的模拟方法有：Mock 或者 Stub（打桩）。业界已经有很多优秀的框架支持这类模拟，譬如 Java 的 Mockito 或者 Ruby 的 RSpec 等。更多关于 Mock 或者 Stub 的介绍，请参考 ThoughtWorks 首席科学家马丁 •福勒的 *Mocks Aren't Stubs* 这篇文章（http:// martinfowler.com/ articles/mocksArentStubs.html）。

- 被测单元依赖于真实部分

在这种情况下，当构建单元测试时，被测单元会与其关联类相互协作，完成具体的业务逻辑。此时，测试的焦点关注在被测单元的数据变化或者状态变化上，而不关心内部的行为调用。譬如，当给定某输入，期望的输出结果是什么。另外，除了如上所说的被测单元以数据为主，另外一种情况是当被测单元的依赖类构建成本非常低，并且被测单元同其依赖类交互的协作成本也非常低，这时可以考虑使用真实的依赖而非模拟。

13.3.2　单元测试的内容

之前已经了解了微服务架构的组成部分，接下来让我们看看针对微服务的这几部分，如何实施单元测试。

基本上，对于微服务的资源定义、业务逻辑、模型存储以及网关集成这几部分而言，其逻辑以行为为主，通常我们会隔离不同部分之间的行为的交互，并使被测单元依赖于模

拟部分。

而对于业务模型部分而言，其行为简单，通常以数据的状态聚合或者状态变化为主，所以我们会使被测单元依赖于真实部分，而尽量不去模拟。

综上所述，微服务下的单元测试实施，通常情况如图 13-4 所示。

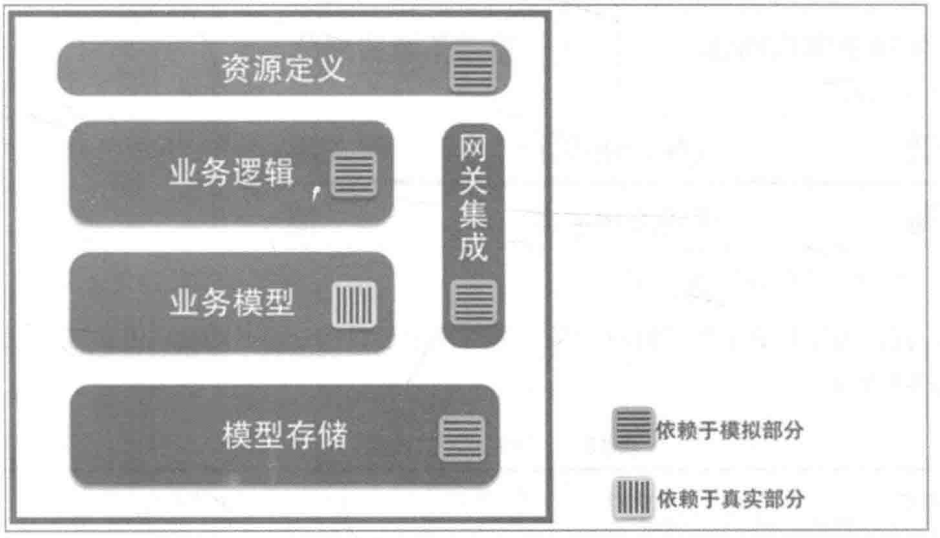


图 13-4　微服务的单元测试

不过，在笔者的实践中，当被测单元的依赖类构建成本较低时，也会倾向于使用真实而非模拟的策略。实际上，对于被测单元而言，使用真实依赖的优点在于不需要写 Mock 或者 Stub 代码，测试代码容易维护。而缺点在于需要构建真实依赖。使用模拟的方式，优点在于依赖类构建成本低，被测单元同依赖类的协作成本低，但随着 Mock 或者 Stub 代码的增多，测试代码维护成本增高。

因此，具体使用哪种策略来实施单元测试，团队需要根据具体的场景找到合适的平衡点，如图 13-5 所示。

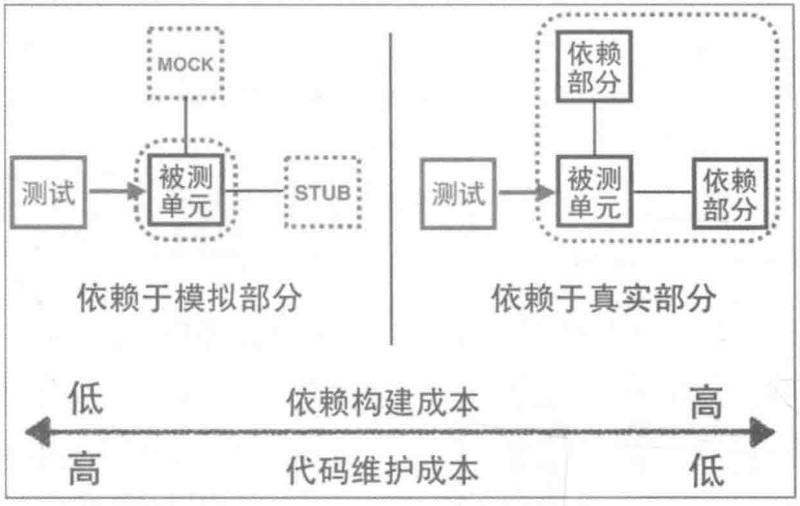


图 13-5　微服务的单元测试成本比较

另外，我们还需要清楚地了解对于微服务的几个组成部分，其功能以及对应的测试内容，如表 13-1 所示。

表 13-1　功能及测试内容

被测单元	功能	测试内容
业务模型	对业务领域实体的抽象	验证数据变化或者状态变化
业务逻辑	对业务行为的抽象。基于业务模型，实现核心业务价值	验证被测单元如何同依赖部分协作，完成核心业务价值
模型存储	对业务数据的存储以及获取	验证能否有效存储业务模型，或者从不同存储媒介获取数据并转换成业务模型
资源定义	业务模型的对外表现形式	验证能否基于业务模型，构建出期望的资源定义
网关集成	负责与其他服务间的协作	验证从其他服务获取数据后的校验、解析以及错误处理

综上所述，对于微服务而言，单元测试主要验证服务内部业务状态的变化以及业务的处理逻辑。

13.4 微服务的集成测试

13.4.1 集成测试综述

集成测试（Integration Testing），是将不同的单元按照期望组合起来，对其接口进行正确性检验的测试工作。通常，集成测试在单元测试之后、端到端测试之前进行。

集成测试的本质是确保当多个单元组合在一起时，能够按照期望协作运行。实际上，集成测试关注三个基本点：

- 当将多个单元组合起来后，能否达到预期的功能要求。
- 当一个单元的功能发生变化，是否会影响相关联部分的功能。
- 当一个单元的功能出现错误，是否会放大到整个系统。

除此之外，还有一些非功能性的验证也可以考虑在集成测试里进行。

在传统瀑布模型的工程实践中，集成测试开始得较晚，通常是大部分的子系统、模块或者单元开发完成后才开始。但随着敏捷、持续集成的深入人心，这部分工作逐渐伴随着开发的过程，持续进行着。

13.4.2 集成测试的实施方法

通常，集成测试的实施方法有两种，如下所示。

- **自顶向下集成**

自顶向下集成（Top-Down Integration）方式是一个递增的组合方法。它从应用的入口开始，把不同的部分组合起来，进行验证，并依次向底层移动，如图 13-6 所示。

对于自顶向下的集成测试而言，它的优点在于从价值高的行为开始，和最近几年大家谈的 BDD（Behavior Driven Development）测试方式有点类似。关于更多 BDD 的信息，请参考 Behavior-Driven Development（https://en.wikipedia.org/wiki/Behavior- driven_development）。

- **自底向上集成**

自底向上（Bottom-Up Integration）集成测试方案是传统瀑布模型的工程实践中较常用的测试方法。它从应用的底层开始，先将不同的模块组合起来，再依次往上移动。

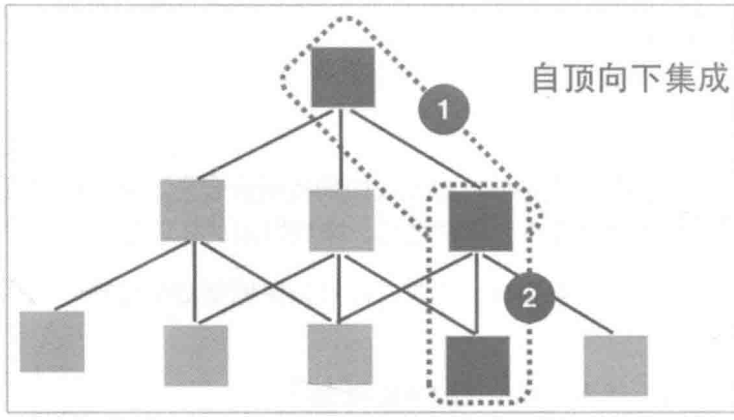


图 13-6　自顶向下集成

对于自底向上的集成测试方式而言，它的优点是方便管理且较易定位问题。但是，它并不太适用于敏捷开发的实践方法，它要求全部单元或者模块实现后再开始集成测试。

13.4.3　集成测试的内容

对于一个服务而言，其本身是高内聚、低耦合，具有独立业务逻辑的单元。同时，它也是整个应用或者系统的一个组成部分。因此，对于上一节提到的微服务以及其内部结构，其集成测试通常是验证微服务与外部存在依赖的部分，通常主要包括如下部分。

- **网关集成**

网关集成部分的测试主要关注它同其他服务的交互。包括但不限于交互过程中的协议处理，发生错误后的异常处理，譬如缺少 HTTP 头，不正确的参数请求等。大部分情况下，对于网关集成而言，其交互的部分可能存在于不同的分布式节点中，因此需要考虑网络超时、网络故障等原因。

- **模型存储**

模型存储部分的测试主要关注它同外部数据存储媒介的交互，包括但不限于数据库、云存储、文件系统等。同时，该部分测试也关注发生错误后的异常处理，譬如连接超时、数据持久性等问题。

对于数据库、云存储以及文件系统这类存储介质而言，其变化主要发生在 Schema 以及数据结构上，因此频率相对较低。测试的难度比起网关集成部分，要相对容易些。

综上所述，对于微服务而言，其集成测试的部分如图 13-7 所示。

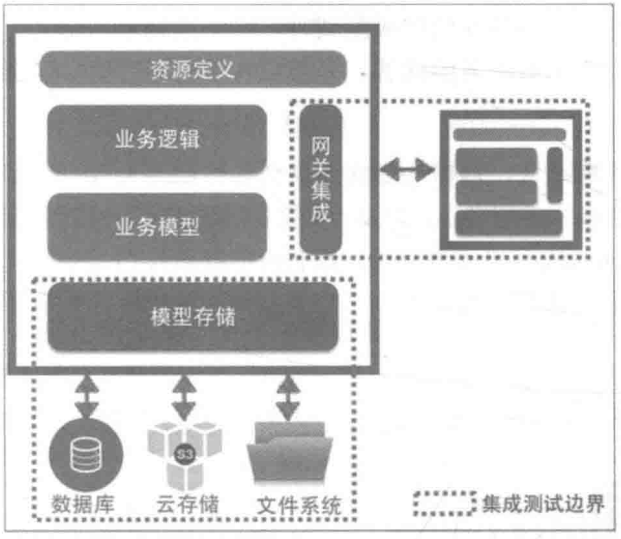


图 13-7　微服务的集成测试

对于微服务而言，集成测试更关注的是服务同外部资源的交互行为是否正确，譬如与其他的服务、第三方服务，或者与数据库、缓存等存储媒介的交互。

13.5　基于消费者驱动的契约测试

13.5.1　集成测试存在的弊端

JB Rainsberger 曾经说过，“集成测试是一个陷阱，它像一个自我扩散的病毒，无情地威胁着代码库、项目和团队。”

实际上，随着微服务系统复杂度的增加，集成测试所带来的弊端愈发明显。

- 运行效率低

由于微服务本身是基于分布式的系统，因此进行集成测试时，每个服务会同运行在其他节点的服务交互，而这类交互通常都是跨网络的。因此，相比进程内的交互或者同一节点内的交互，运行效率较低。尤其是当服务之间依赖关系复杂时，结果更明显。

- 运行结果不稳定

对于分布式的系统而言，当进行不同服务间的集成测试时，网络的延迟、带宽可能会影响到测试运行的结果。除此之外，所依赖服务的部署、升级或者临时不可用，也会导致测试运行的结果不稳定。

另外一方面，随着依赖服务的不断迭代，接口的变化通常会导致结果失败。譬如，被测服务依赖于 V1 版本的服务 A，V2 版本的服务 B，以及 V3 版本的服务 C。也就意味着，对服务 A、服务 B 或者服务 C 的任何一次升级或者降级，都有可能导致接口发生变化，从而使服务 A 的功能受影响。而随着服务依赖的数量增多，复杂度增加的同时，更容易导致运行结果不稳定。

譬如，当服务 C 的接口版本从 V3 升级到 V4 后，将导致测试失败，如图 13-8 所示。

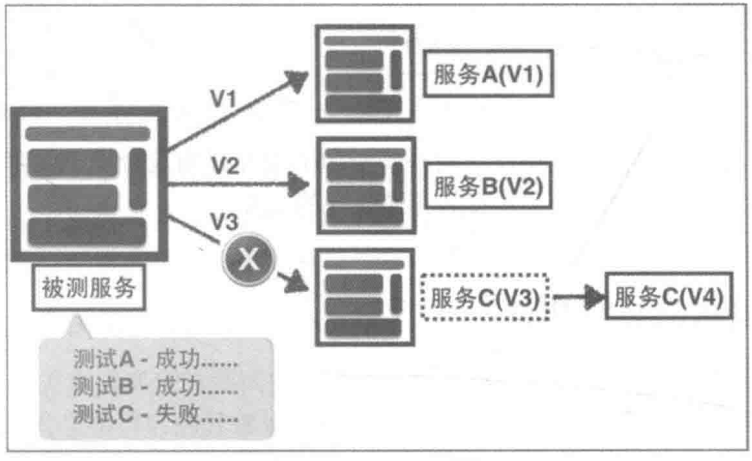


图 13-8　接口变化导致测试失败

- 反馈周期慢

从经典的测试金字塔理论来看，集成测试所关注的粒度比单元测试粗。通常，集成测试会将几个不同的服务组合起来进行验证。这也意味着，随着系统的复杂度逐渐增加，服务的数量越多，影响错误的因素就越多，反馈周期也就越长，开发人员需要花更多的时间去定位错误的原因。

譬如，如果集成测试包括 3 个部分，每个部分存在 2 个分支，则组合后就变成 2×2×2=8 种测试场景。这也意味着，当测试失败后，我们需要花费更多的时间定位问题。

13.5.2 什么是契约

之前已经提到，对于服务而言，当其依赖服务的接口升级、接口不匹配，都会导致集成测试失败。

那什么是接口？接口其实就是服务间的通信方式和协议，也可以称为契约（Contract）。契约的两端分别是消费者（Consumer）和提供者（Provider）。使用契约的一方称为消费者，提供契约的一方称为提供者。

当契约发生改变后，那么所有契约的消费者都需要针对这个变化做出调整。

在微服务架构下，这就意味着即便是简单的契约变化，也会使多个使用契约的消费者受到影响。如果测试不及时，可能导致很多服务在测试环境，甚至是生产环境中出现错误，如图 13-9 所示。

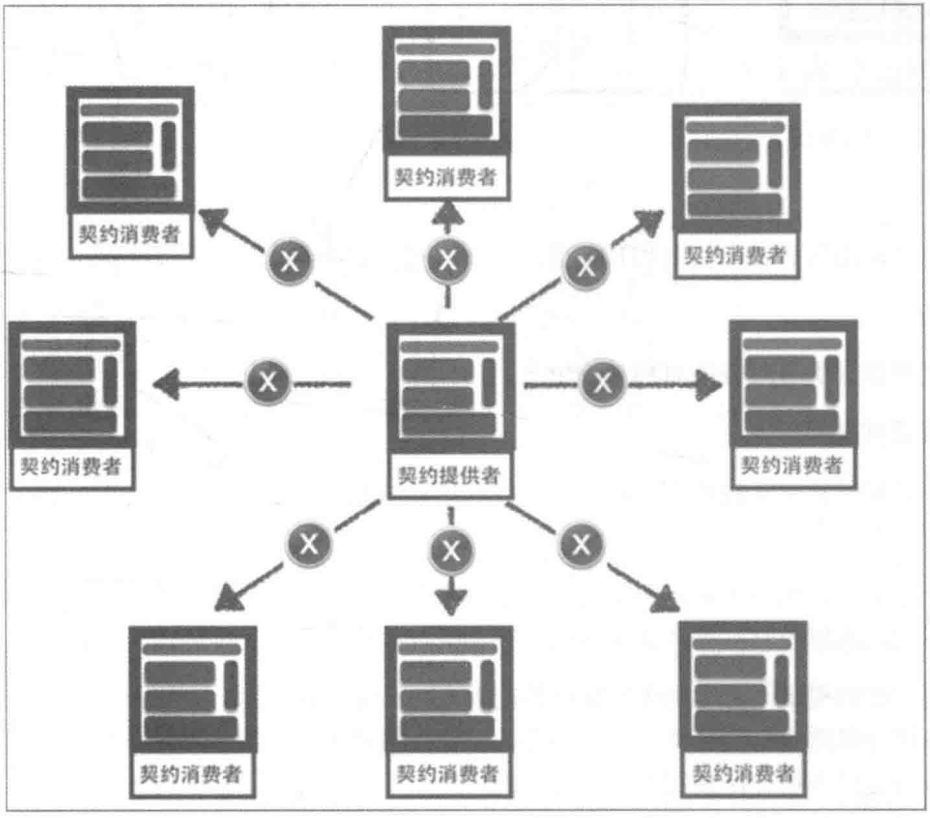


图 13-9 接口变化导致的失败

因此，如何找到一种有效的方式，及时发现服务之间契约的变化是否会导致消费者的逻辑出错，显得尤为重要。

13.5.3 什么是契约测试

契约测试是针对服务接口进行的测试，它能验证提供者提供的契约是否满足消费者的期望。在契约测试中，契约主要表现为三部分，如图 13-10 所示。

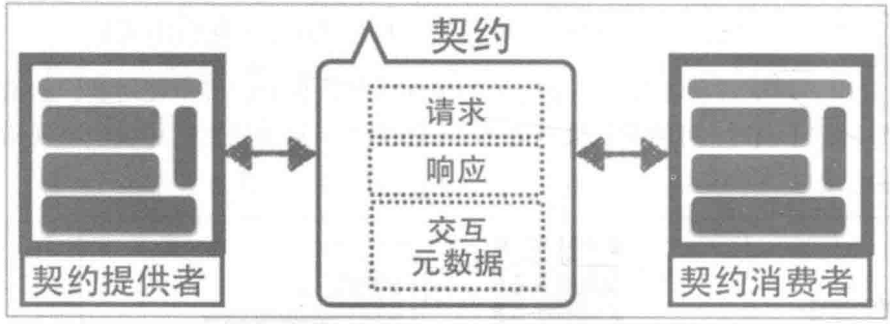


图 13-10　契约测试

- **请求**

请求主要指消费者发出的 HTTP 请求，包括 URL、请求的参数以及 HTTP 动词等。

- **响应**

响应主要指提供者应答的 HTTP 响应。

- **交互元数据**

交互元数据是指交互过程中，描述交互过程中的数据。譬如提供者的信息、消费者的信息、当前的上下文等。

对于一个服务提供者而言，每个消费者会根据与其交互的不同，生成不同的契约。当这个提供者被频繁修改时，那么就应该保证每个消费者依然能够正确地消费契约。

契约测试能够提供一个机制去验证组件能否始终满足契约。当涉及的组件是微服务时，接口便是各服务暴露出来的公共 API。每个消费方服务的维护者只需要编写独立的测试套件来验证服务提供方中他所使用的那一部分就可以了。这些测试不是组件测试，它不需要深入地去测试服务的行为，只需要去测服务的输入输出包含了必需的属性，响应的等待时

间以及吞吐量。

契约测试提供了一种机制，帮助验证消费者和提供者在协作过程中接口是否发生变化。譬如，每当提供者提供的接口发生变化，契约测试就能检测到提供者提供的当前接口是否和契约要求的保持一致。

譬如，如图 13-11 所示，对于 V0 版本的契约，契约提供者和契约消费者都能够相互协作，完成业务价值的实现。但是，当提供者提供的接口发生变化，譬如从 V0 升级到 V1 后，契约测试就能够立刻验证出，提供者当前发布的接口与之前消费者期待的契约不匹配，从而及早发现错误。

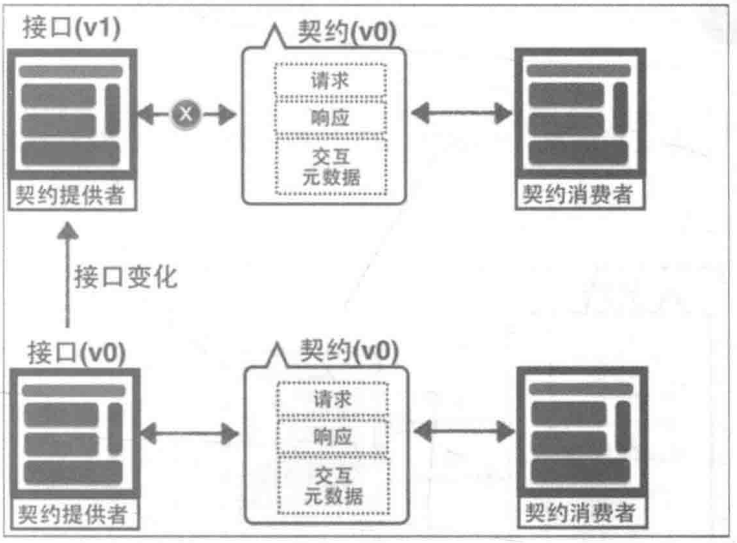


图 13-11　接口变化导致的契约测试失败

通过契约测试，能帮助我们有效地降低微服务接口变化所带来的消费者端的风险。

13.5.4　契约测试的方法

我们知道，对于契约而言，消费者消费契约，提供者提供契约。也就是说，当提供者提供的契约能够被消费者消费后，其对应的价值才被实现。对于那些消费者无法消费的契约，并无法真正有效实现价值。换句话说，消费者其实是价值流中离价值实现最近的。

如果我们从消费者出发，以先定义满足消费者的契约作为第一步，就能优先保障价值的实现，如图 13-12 所示。

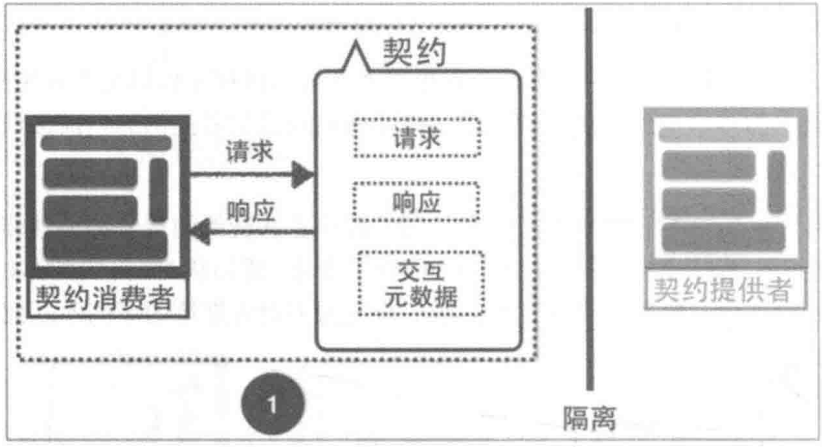


图 13-12　满足消费者的契约

再通过如上所示的契约，验证提供者，使其提供的契约和消费者需要的契约一致，如图 13-13 所示。

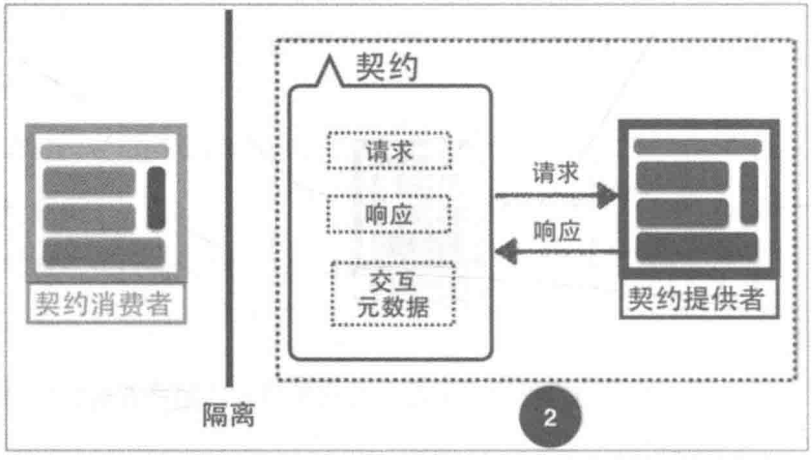


图 13-13　验证消费者的契约

通过如上两个分开的步骤，就能有效地保障消费者和提供者协同工作，这就是我们提到的消费者驱动的契约测试。

在消费者驱动的契约测试中，通常的实现方式是这样的：

1. 选择合适的场景，定义消费者的请求和期望的响应。譬如获取产品列表，获取产品

明细等。

2. 使用 Mock 机制，为消费者定义模拟的提供者以及期望的响应。

3. 记录消费者发送的请求、提供者提供的响应以及关于场景的其他元数据，并将其记录为当前场景的契约。

4. 模拟消费者，向真正的提供者模拟发送请求。

5. 验证提供者提供的契约是否和之前记录的契约一样。

目前，业界比较著名的消费者驱动的契约测试框架主要有 Janus（https://github.com/gga/janus）、Pact（https://github.com/realestate-com-au/pact）和 Pacto（http://thoughtworks.github.io/pacto/）等。

在微服务架构下，基于消费者驱动的契约测试，有两个非常明显的优势。

- **从价值实现的角度定义契约**
  如上描述所示，基于消费者驱动的契约测试，从消费者使用契约的角度出发，首先保证消费者基于此契约是可以实现价值的。有了这个前提，再使用契约来验证提供者，如果提供者提供的契约同定义的契约一致，则证明提供者提供的契约是能够服务消费者的。通过这种方式，能让团队或者组织更聚焦于如何从价值实现出发。

- **隔离消费者和提供者的测试**
  有了消费者驱动的契约测试，能够首先保证消费者基于契约是可以实现价值的，从而再通过契约验证提供者提供的契约是符合要求的。在这种情况下，我们可以看出，对于契约的消费者和提供者而言，二者可以分开独立测试。因此，基于消费者驱动的契约测试，能有效解决之前提到的传统集成测试在微服务架构下存在的弊端，将微服务接口的测试成本降到最低。

对于微服务而言，基于消费者驱动的契约测试，能够从消费者使用契约的角度出发，有效验证消费者和提供者之间的协作是否正常工作。同时，契约的存在，使得消费者和提供者之间，能独立进行测试，极大降低了测试成本。

13.5.5 Pact 实现契约测试

Pact 是一个基于消费者驱动的契约测试框架。它最早起源于 Realestate.com.au 的一个内部项目，目前已开源。起初，Pact 只支持 Ruby，后来开始支持 JVM 和.NET 平台。更多

关于 Pact 的信息，请访问 Pact（https://github.com/realestate-com-au/pact）。

Pact 的使用主要包括两步：消费者端生成契约与提供者端完成校验。

下面，我们将分别介绍这两部分。

1. 消费者端生成契约

如图 13-14 所示，在消费者端生成契约的过程中，主要的步骤如下所示。

（1）消费者端使用 Pact 框架提供的 DSL 以及相关的配置文件，运行一个本地的模拟服务，模拟服务的提供者端。

（2）消费者端发送 HTTP 请求。

（3）提供者端返回响应，供消费者使用。

（4）消费者端和提供者端完成一次交互。Pact 框架将当前的请求和响应记录下来，并生成 Pact 文件，也叫契约。

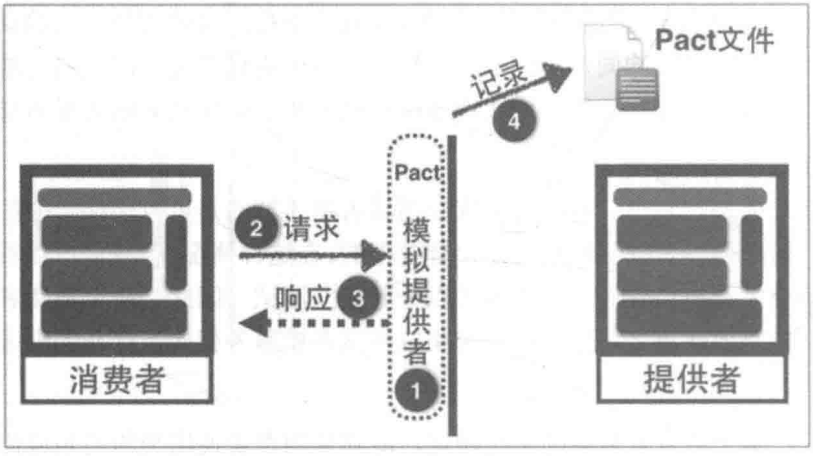


图 13-14　消费者端生成契约

Pact 文件的内容是 JSON 格式的，主要记录消费者以及提供者之间的交互过程，包括消费者的请求、参数以及提供者的响应等。譬如，如下的 Pact 文件记录了消费者端以及提供者端之间的交互过程。

```
{
  "consumer": {
```

```
    "name": "Products Consumer"
  },
  "provider": {
    "name": "Products Service"
  },
  "interactions": [
    {
      "description": "a request for all products",
      "provider_state": "products exist",
      "request": {
        "method": "get",
        "path": "/products"
      },
      "response": {
        "status": 200,
        "headers": {
          "Content-Type": "application/json"
        },
        "body": [
          {
            "name": "Programming Ruby",
            "price": 89.0
          },
          {
            "name": "Building Microservice",
            "price": 99.0
          }
        ]
      }
    }
  ],
  "metadata": {
    "pactSpecificationVersion": "1.0.0"
  }
}
```

如上代码所示，Pact 文件中的第一部分描述了消费者和提供者的相关信息：

```
{
  "consumer": {
```

```
    "name": "Products Consumer"
  },
  "provider": {
    "name": "Products Service"
  }
}
```

其中，消费者为 Products Consumer，提供者为 Products Service。

第二部分描述了消费者和提供者之间的交互，包括消费者的请求（request）和提供者的响应（response），如下所示：

```
"interactions": [
  {
    "description": "a request for all products",
    "request": {
      "method": "get",
      "path": "/products"
    },
    "response": {
      "status": 200,
      "headers": {
        "Content-Type": "application/hal+json"
      },
      "body": [
        {
          "name": "Programming Ruby",
          "price": 89.0
        },
        {
          "name": "Building Microservice",
          "price": 99.0
        }
      ]
    }
  }
]
```

如上所述，在当前的例子中，消费者端发送的 HTTP 请求的路径为/products，HTTP 动词为 get。

类似的，提供者端提供的 HTTP 响应包括如下几部分：

- 返回的状态码为 200。
- 返回的 HTTP 头信息：Content-Type: application/hal+json
- 返回的 HTTP 内容体为：

```
[
  {
    "name": "Programming Ruby",
    "price": 89.0
  },
  {
    "name": "Building Microservice",
    "price": 99.0
  }
]
```

2. 提供者端完成校验

有了消费者端生成的 Pact 文件，提供者端就可以通过 Pact 框架提供的 DSL 以及相关的配置文件，模拟消费者端发送请求，并验证提供者端返回的响应是否和 Pact 文件中描述的响应一致，其主要的步骤如图 13-15 所示。

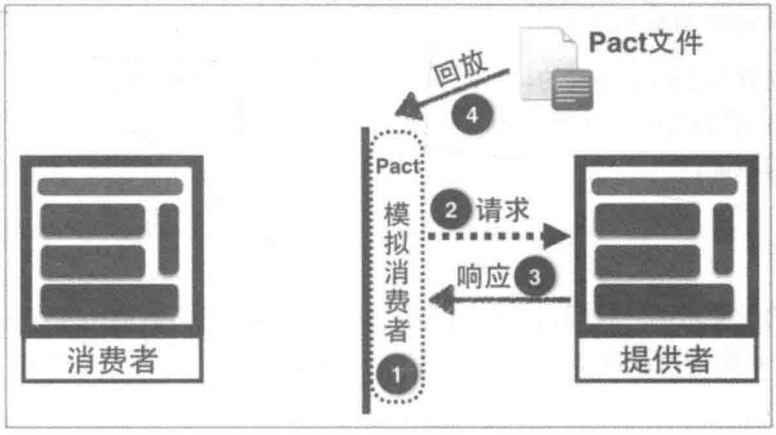


图 13-15　提供者端完成校验

（1）提供者端使用 Pact 框架提供的 DSL 以及相关的配置文件，配置 Pact。

（2）Pact 框架将模拟消费者向提供者端发送请求。

（3）提供者端返回响应。

（4）Pact 框架验证提供者端提供的响应是否与 Pact 文件匹配。

综上所述，Pact 框架从消费者端出发，先帮助消费者端获取预期的响应，记录交互过程并生成契约，再通过契约反向验证提供者端。基于这种方式，每当提供者端提供的响应发生变化时，我们就能通过 Pact 框架以及相关的 Pact 测试立刻体现出来，帮助我们及早发现服务间接口不一致的问题。这就是我们所说的消费者驱动的契约测试。

如何访问契约文件

我们知道，Pact 文件是由消费者端生成的，提供者端要访问 Pact 文件的话，通常有两种途径：

- 提供者端从消费者端获取 Pact 文件，并保存在其代码库里，作为本地测试集的一部分。
- 提供者端从 Pact Broker（https://github.com/bethesque/pact_broker）上获取 Pact 文件。

Pact Broker 作为 Pact 文件的发布和存储平台，可以方便地帮助消费者端与提供者端共享 Pact 文件。

在消费者端，当运行完 Pact 测试后，可以通过 Pact 框架提供的 Rake 任务将 Pact 文件发布到 Pact Broker 平台上；在提供者端，通过指定 Pact Broker 的访问地址，使其可以在运行 Pact 测试时从 Pact Broker 直接获取 Pact 文件。

另外，由于 Pact 文件是 JSON 格式的，这就保证了它的语言无关性。譬如，我们可以用 Ruby 语言完成消费者端的功能，并使用 Pact 框架生成 Pact 文件，而在提供者端用 Java 语言来完成 Pact 文件的验证。

13.5.6 一个例子

如下是使用 Pact 的一个示例。我们将采用测试驱动的方式，从消费者端出发，先驱动出消费者端获取响应，再驱动出提供者端提供响应。

- **场景分析**

假定存在合同服务（Contracts-service），当用户签署合同的时候，它同产品服务（Products-service）交互，获取产品的列表信息，计算完折扣后显示给用户。换句话说，合同服务是数据的消费者端，产品服务是数据的提供者端。接下来，我们分别实现消费者端

与提供者端，如图 13-16 所示。

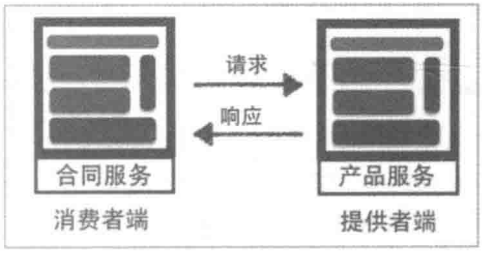


图 13-16　消费者与提供者

- 消费者端实现

在消费者端，主要实现步骤如图 13-17 所示。

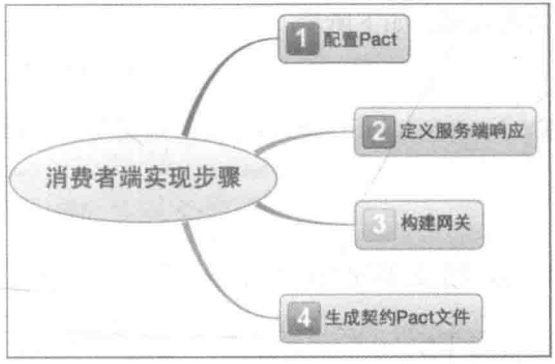


图 13-17　消费者端实现步骤

1. 配置 Pact

首先，创建 contracts-service：

```
mkdir contracts-service
```

在 contracts-service 中创建 Gemfile，并添加如下内容：

```
source 'https://rubygems.org'
gem 'httparty'

group :development, :test do
  gem 'rspec'
  gem 'pact'
end
```

创建 pact_helper，配置 Pact 并定义提供者端的访问端口，如下所示：

```
require "pact/consumer/rspec"

Pact.service_consumer 'Contracts Service' do
  has_pact_with 'Products Service' do
    mock_service :products_service do
      port 8181
    end
  end
end
```

如上代码所示，首先引入 Pact 框架配置文件 require "pact/consumer/rspec"，然后使用 Pact 框架提供的 DSL 声明消费者端和提供者端：

（1）使用 Pact.service_consumer 定义消费者端，如下所示：

```
Pact.service_consumer 'Contracts Service'
```

（2）使用 has_pact_with 定义提供者端。has_pact_with，顾名思义，定义与当前消费者相关的提供者，如下所示：

```
has_pact_with 'Products Service'
```

（3）使用 mock_service 定义提供者端的访问别名及端口，如下所示：

```
mock_service :products_service do
   port 8181
 end
```

2. 定义服务端响应

接下来，驱动出合同服务同外部的交互逻辑（如之前部分描述，同外部服务交互的逻辑通常放在服务的网关集成中，也可以放在模型存储部分。在当前的例子中，笔者将其放在网关集成中。）

首先，创建测试文件 contracts_service_gateway_spec.rb，并期望存在 ContractServiceGateway 类中负责同 product-service 的交互，获取产品列表的信息。如下所示：

```
require 'service_providers/pact_helper'
describe ContractServiceGateway, pact: true do
  before do
    ContractServiceGateway.base_uri 'localhost:8181'
```

```
  end

  subject { ContractServiceGateway.new }

  describe 'get_products' do
    before do
      products_service.given('products exist').
          upon_receiving('a request for all products').
          with(method: :get, path: '/products').
          will_respond_with(
              status: 200,
              headers: {'Content-Type' => 'application/json'},
              body:    [{name: 'Programming Ruby', price: 89.0},
                        {name: 'Building Microservice', price: 99.0}])
    end

    it "returns products" do
      expect(subject.get_products).to match_array(
              [{'name' => ' Programming Ruby', 'price' => 89.0},
               {'name' => 'Building Microservice',  'price' => 99.0}])
    end
  end
end
```

如上代码所示，首先引用 pact_helper：

```
require 'service_providers/pact_helper'
```

接下来，使用 RSpec 的语法描述 ContractServiceGateway，并使用 pact:true 标记当前测试为 Pact 测试。同时，设置消费者端待访问的 uri 为 localhost:8181，与之前定义的消费者端访问端口一致，如下所示：

```
describe ContractServiceGateway, pact: true do
  before do
    ContractServiceGateway.base_uri 'localhost:8181'
  end
```

随后，测试 ContractServiceGateway 向 product-service 发送请求，获取响应结果，并返回产品列表信息，如下代码所示：

```
subject { ContractServiceGateway.new }
```

```
describe '#get_products' do
  before do
    products_service.given('products exist').
      upon_receiving('a request for all products').
      with(method: :get, path: '/products').
      will_respond_with(
         status: 200,
         headers: {'Content-Type' => 'application/json'},
         body:    [{name: 'Programming Ruby', price: 89.0},
                   {name: 'Building Microservice', price: 99.0}])
  end

  it "returns products" do
    expect(subject.get_products).to match_array(
          [{'name' => 'Programming Ruby', 'price' => 89.0},
           {'name' => ' Building Microservice',  'price' => 99.0}])
  end
end
```

如上所示，products_service.given 定义当前上下文为 products exist，upon_receiving 是对该上下文 products exist 的描述。with(method: :get, path: '/products')表明消费者端将使用 get 的方式请求/products。

will_respond_with 表明当消费者发送请求后，期望的响应。

其实，Pact 消费者端测试的结构和 BDD 中的测试结构非常类似，也是类似于

```
Given......
When......
Then......
```

熟悉 BDD 的读者，可以将 products_service.given 理解为 BDD 中的 Given，with(method: :get, path: '/products')对应 BDD 中的 When，而 will_respond_with 则对应 BDD 中的 Then。

同时，如上代码使用 RSpec 对 ContractServiceGateway 的 get_products 结果进行验证，并期望返回产品信息列表：

```
it "returns products" do
  expect(subject.get_products).to match_array(
     [{'name' => 'Programming Ruby', 'price' => 89.0},
```

```
        {'name' => 'Building Microservice', 'price' => 99.0}])
    end
```

接下来，运行测试：

```
 $> rspec
```

这时，得到的结果是失败的，原因是我们并没有定义 ContractsServiceGateway，错误的结果类似如下所示：

```
    contracts-service/spec/contracts_service_gateway_spec.rb:2:in 'require_relative':
cannot load such file -- contracts-service/app/contracts_service_gateway (LoadError)
    from /Users/leiwang/Work/microservice/gitbook/Micro-Service-In-Analysis-Code/
contracts-service/spec/contracts_service_gateway_spec.rb:2:in '<top (required)>'
    ......
```

3. 构建网关

接下来，我们开始实现 ContractsServiceGateway，代码如下所示：

```
require 'httparty'

class ContractServiceGateway
  include HTTParty

  base_uri 'http://products-service.prod.com'

  def get_products
    JSON.parse(self.class.get("/products").body)
  end
end
```

如上代码所示，主要的逻辑为发送请求到/products，并解析获取到的响应：

```
def get_products
  JSON.parse(self.class.get("/products").body)
end
```

4. 生成契约 Pact 文件

此时再次运行 rspec，测试就通过了。

如果留意当前 contracts-service 的代码库，细心的读者会发现多了两个目录，分别是 log 和 spec/pacts，如下所示：

```
├── log
│   ├── pact.log
│   └── products_mock_service.log
└── spec
    |.....
    ├── pacts
    |   └── products_consumer-products_service.json
```

这两个目录是测试运行完成后，Pact 自动生成的。生成的文件主要包括：

- pact.log 记录了 Pact 框架运行的日志。
- products_mock_service.log 记录了提供者端接受请求和应答响应的过程。
- spec/pacts/products_consumer-products_service.json 就是之前我们提到的 Pact 文件，也称为契约，其内容如下所示：

```
{
  "consumer": {
    "name": "Products Consumer"
  },
  "provider": {
    "name": "Products Service"
  },
  "interactions": [
    {
      "description": "a request for all products",
      "provider_state": "products exist",
      "request": {
        "method": "get",
        "path": "/products"
      },
      "response": {
        "status": 200,
        "headers": {
          "Content-Type": "application/json"
        },
        "body": [
          {
            "name": "Docker In Action",
            "price": 59.0
```

```
        },
        {
          "name": "Agile In Action",
          "price": 55.0
        }
      ]
    }
  }
 ],
 "metadata": {
   "pactSpecificationVersion": "1.0.0"
 }
}
```

这和我们之前描述的例子非常类似，主要记录消费者端和提供者端的交互过程。至此，在 contract-service 端，其代码已经能够发送请求，获取数据，并将响应的结果转换成产品的数组了。有兴趣的读者可以基于本书前面讲述的内容，将 contracts-service 对响应数据的处理进行扩展，笔者在这里就不赘述了。

- 提供者端实现

在提供者端，实现步骤如图 13-18 所示。

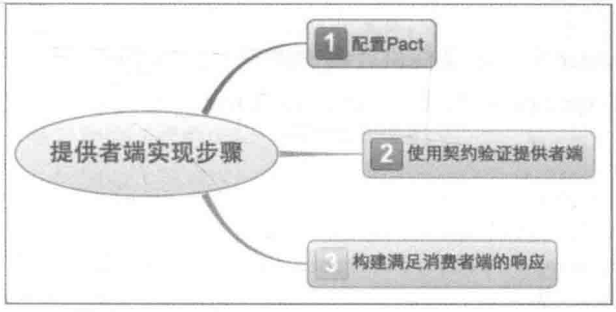


图 13-18　提供者端实现步骤

1. 配置 Pact

接下来，我们看看如何基于 Pact 文件，驱动出 products-service 对消费者的请求所做的响应。如第 3 章所述，我们已经有了基本的 products-service 代码库，后续的例子将基于 products-service。

在 products-service 的 Gemfile 中，添加如下内容：

```
group :development, :test do
  gem 'pact'
end
```

在 products-service 中使用 Pact 框架提供的 DSL 配置 Pact，同时将 contracts-service 生成的 Pact 文件复制到当前代码库中：

```
$> mkdir -p spec/pacts
$> cp contracts-service/spec/pact/contracts_service-products_service.json spec/pacts/
```

定义 Pact 测试文件 pact_with_products_service.rb，并添加如下内容：

```
Pact.service_provider 'Products Service' do
  honours_pact_with 'Contracts Service' do
    pact_uri '../products-service/spec/pacts/contracts_service-products_
service.json'
  end
end

Pact.provider_states_for 'Contracts Service' do
  provider_state 'products exist' do
  end
end
```

如上代码所示，Pact.service_provider 主要定义提供者端的配置并指明 pact_uri。在当前例子中，我们已经将 Pact 文件复制到 spec/pacts/目录下，因此设置 pact_uri 为本地的文件路径。

```
Pact.service_provider 'Products Service' do
  honours_pact_with 'Contracts Service' do
    pact_uri '../products-service/spec/pacts/contracts_service-products_
service.json'
  end
end
```

Pact.provider_states 则主要用于定义交互场景，即上下文。我们知道，对于交互的双方而言，场景不同，对应的请求和响应都不一样。实际上，在之前消费者驱动契约的部分，我们已经定义了上下文，代码如下所示：

```
before do
```

```
  products_service.given('products exist').
    with(method: :get, path: '/products').
    will_respond_with ......)
end
```

只有当消费者端和提供者端保持同样的上下文，对应的 Pact 测试才会通过。

接下来，定义 Pact 的配置文件 pact_helper，并设置其加载 Pact 的测试文件 pact_with_products_service.rb，代码如下所示：

```
project_root = File.dirname(__FILE__)
$LOAD_PATH << "#{project_root}"
require 'pact_with_products_service'
```

2. 使用契约反向验证

在 products-service 的 Rakefile 中，添加如下内容：

```
require 'pact/tasks'
```

并使用如下命令查看 Rake 任务，如下所示：

```
$> bundle exec rake -T
```

发现多了三个关于 Pact 的 Rake 任务，如下所示：

```
rake pact:verify
# 基于pact_helper配置的Pact文件，验证服务提供者

rake pact:verify:at[pact_uri]
# 基于Pact文件的URL，验证服务提供者

rake pact:verify:help[reports_dir]
# 调试Pact运行时的信息
```

如上所示，rake pact: verify 表示使用代码中默认的配置，运行 Pact 测试。

rake pact:verify:at[pact_uri]表示使用指定的 Pact Broker URI 运行 Pact 测试。

rake pact:verify:help[reports_dir]帮助调试 Pact 测试运行时的错误。

至此，我们还没有实现任何提供者端的响应代码，因此 Pact 测试会失败。

接下来的工作是如何实现提供者端的响应代码，使其满足 Pact 测试。

3. 构建 products-service 的响应

在之前的 products-service 的代码库中，我们已经使用 Grape 完成了 Hello World API 的实现。接下来，我们就使用 Grape 实现 products-service 对消费者请求的响应。具体测试驱动的细节，笔者就不赘述了，这里主要列出实现的几个部分。

- **业务模型定义**

首先，定义 products-service 的业务模型，如下所示：

```
class Product
  include Virtus.model

  attribute :id,              Integer
  attribute :name,            String
  attribute :price,           BigDecimal

end
```

- **模型存储实现**

接下来，实现模型存储。在本例中，笔者忽略了具体的数据库存储，假定 ProductRepository 内部直接返回产品列表数据。感兴趣的读者可以使用 ActiveRecord、ROM 或者其他 ORM 框架完成这部分逻辑，代码如下所示：

```
class ProductRepository
  @@products = [
     Product.new({id: 1,
            name: 'Programming Ruby',
            price: 89.0}),

     Product.new({id: 2,
            name: 'Building Microservice',
            price: 99.0})
  ]

  def self.list
    @@products
  end

  def self.find(id)
```

```
    @@products.select do |product|
      product.id == id
    end.first
  end
end
```

- **资源定义实现**

接下来，实现资源的定义。

在本例中，资源定义包括两部分，产品资源和产品列表资源。

产品资源的代码如下所示：

```
require 'roar/decorator'
require 'roar/json/hal'

class ProductPresenter < Roar::Decorator
  include Roar::JSON::HAL

  property :id
  property :name
  property :price, getter: lambda { |_| '%.2f' % price }

end
```

类似的，产品列表资源的代码如下所示：

```
require 'roar/decorator'
require 'roar/json'
require 'roar/json/hal'

class ProductsPresenter < Roar::Decorator
  include Roar::JSON
  include Roar::JSON::HAL

  collection :entries,
             as: :products,
             embedded: false,
             extend: ProductPresenter
end
```

- API 实现

最后，使用 Grape 实现 API 部分，代码如下所示：

```
require 'grape-roar'
require 'helpers/params_helper'

class ProductApi < Grape::API
  rescue_from :all do |e|
    error_response(message: {
                     type: 'http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html',
                     title: e.message
                   },
                   headers: { 'Content-Type' => 'application/problem+json' })
  end

  format :json
  formatter :json, Grape::Formatter::Roar
  content_type :json, 'application/hal+json'

  helpers ParamsHelper

  desc 'return all products'
  get '/' do
    ProductsPresenter.new(ProductRepository.list)
  end

  route_param :id do

    params do
      requires :id,      type: Integer, allow_blank: false
    end

    desc 'return one product by id'
    get do
      ProductPresenter.new(ProductRepository.find(params[:id]))
    end
  end
end
```

最后，我们再次运行 rspec，可以看到 Pact 测试已经通过了，如下所示：

```
SPEC_OPTS='' /Users/leiwang/.rvm/rubies/ruby-2.1.5/bin/ruby -S pact verify
--pact-helper /Users/leiwang/Work/microservice/gitbook/Micro-Service-In-Analysis-
Code/products-service/spec/pacts/pact_helper.rb
Reading pact at ../products-service/spec/pacts/contracts_service-products_
service.json

Verifying a pact between Contracts Service and Products Service
  Given products exist
    a request for all products
      with GET /products
        returns a response which
          has status code 200
          has a matching body
          includes headers
            "Content-Type" with value "application/hal+json"

1 interaction, 0 failures
```

13.5.7 本节小结

在本节中，我们首先介绍了什么是 Pact，并使用 Pact 框架完成了一个消费者端与提供者端交互的验证。

Pact 框架从消费者端出发，先帮助消费者端获取预期的响应，记录交互过程并生成契约，再通过契约反向验证提供者端。基于这种方式，每当提供者端提供的响应发生变化时，我们就能通过 Pact 框架以及相关的 Pact 测试立刻体现出来，帮助我们及早发现服务间接口不一致的问题。

13.6 微服务的组件测试

13.6.1 组件测试概述

组件测试，又称 Component Testing，是指对组件提供的功能进行正确性检验的测试工作。组件的概念大家一定不陌生，当将庞大的系统拆分成不同的部分时，每个独立的部分就是一个组件。在微服务架构中，每个服务也可以看作是一个组件。

之前我们已经提到，单元测试覆盖服务内部构成的主要部分，譬如业务模型、模型存储、网关集成等。集成测试验证服务同外部的交互，包括数据库、其他服务或者第三方服务。消费者驱动契约测试关注服务和服务之间接口的一致性验证。相比这些测试而言，组件测试是从外部功能的角度对微服务进行验证，它通常是一种黑盒测试。当对某服务进行组件测试时，会发送 HTTP 请求到指定的端口，并验证返回的响应是否满足期望。

13.6.2 组件测试的方法

在微服务中，组件的测试主要分为两类，进程内测试和进程外测试。

- **进程内测试**

进程内组件测试是指将服务的依赖部分，譬如外部数据库、外部服务等进行模拟，使用进程内的一些服务替换。譬如，使用内存数据库替换外部数据库，使用模拟网关集成部分代替真正的网关集成部分。替换后，这些依赖与被测服务运行在同一个进程中，从而达到既测试了当前服务的功能，又提高了测试执行效率。

以网关集成部分为例，为了隔离外部的服务接口，会使用模拟的方式替换网关集成部分。于是，当业务逻辑调用模拟网关集成时，会立刻得到期望的响应，而实际上网关集成部分并没有真正发送请求。

类似的，使用内存数据库替换数据库后，模型存储会直接访问内存数据库，更高效地加载或者存储数据，如图 13-19 所示。

另外，这种模拟的方式也可以使组件测试不仅仅只关注正常的业务逻辑，还能够测试某些特殊异常的处理。譬如网络异常、数据库异常等。

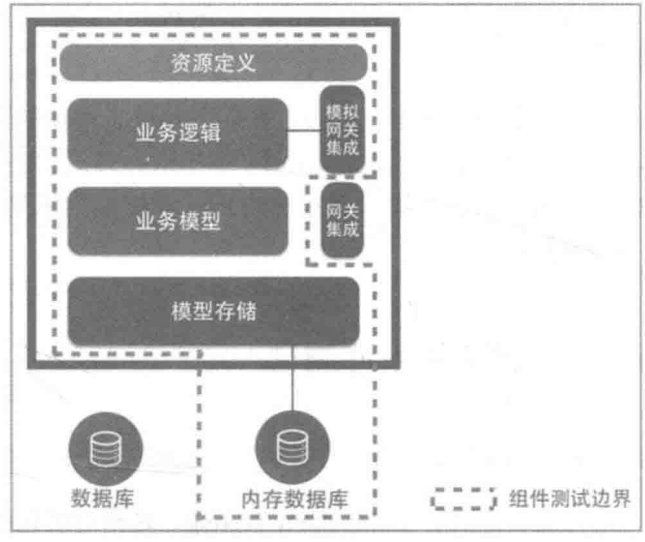


图 13-19 进程内组件测试

- **进程外测试**

进程外组件测试是指将被

测服务的外部依赖环境设置好，使被测服务与其依赖能够交互并完成真正的功能。这些依赖都运行在不同节点上，譬如真实的数据库或者外部的服务等。

进程外组件测试的优点在于能够真实地使用外部依赖，对当前服务进行功能验证，也意味着该测试更接近真实的生产环境。不过，其缺点在于管理被测服务的同时，还要管理其相关的外部资源。譬如，当启动被测试服务时，应该也能够启动其外部依赖的服务。这样才能保证测试过程是有效的。

同时，由于在进程外组件测试中，当前被测服务将同外部的依赖进行交互，因此，也保证了当前被测服务的参数配置、网络配置等是正确的。

虽然在进程外测试过程中，被测服务的内部单元不会有任何的模拟或者打桩，不过我们可以采取模拟外部依赖的方式来提高测试效率。譬如，可以对当前服务依赖的外部服务进行打桩，当它们接收到被测服务的请求时，直接返回期望的响应。目前，这类工具主要包括 moco、stubby4j 和 mountebank 等。

譬如，典型的进程外组件测试如图 13-20 所示。

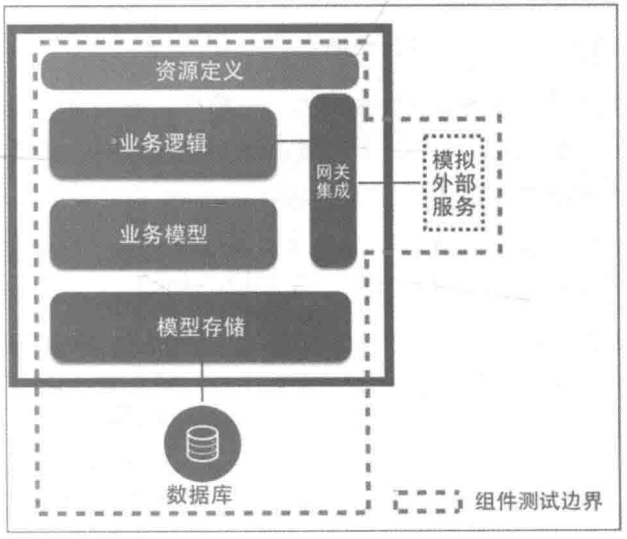


图 13-20　进程外组件测试

13.6.3　本节小结

本节中，我们讨论了微服务的组件测试。

对于微服务的组件测试而言，它关注的是当指定输入时，被测服务是否能提供期望的输出。实际上，组件测试并不会对服务内部的逻辑做任何模拟或者打桩。因此，通过组件测试，我们能有效保证当前被测服务的功能是正常的。

同时，需要注意一点，在被测服务接收并处理请求的过程中，它可能需要和其他服务、数据库或者缓存等依赖部分相互协作，因此我们可以根据具体的情况，决定采用进程内还是进程外的组件测试。

13.7 微服务的端到端测试

13.7.1 端到端测试概述

端到端测试，又称 End-To-End Testing 或者 System Testing，是从用户使用系统的角度出发，对系统的行为进行正确性验证的测试。

在端到端测试中，测试过程准备的所有数据都是为了模拟实际用户使用系统。同时，被测试系统也是经过多个部分集成以后的功能较完整的系统。

相比其他类型的测试，端到端测试的目的是验证系统作为一个整体，能否满足外部对其的期望。端到端测试的方式基本上以黑盒测试为主，不需要关注系统内部的组成结构、实现语言等。为系统提供输入后，验证是否能得到期望的输出。

13.7.2 端到端测试的内容

对于微服务架构的系统而言，系统通常会由多个不同的服务组成，因此，从某种意义上讲，微服务架构的端到端测试，会对系统内多个服务之间的协作进行较全面的覆盖，这同之前我们描述的集成测试、消费者驱动的契约测试相比，对服务间协作的测试更加充分和全面。

另外，端到端测试会涉及系统内部的多个服务间的协作。因此在测试过程中，对系统中的每个服务的可用性，包括功能、参数配置、网络、路由等要求较高，是较接近生产环境的验证，如图 13-21 所示。

同时，随着微服务架构的不断演进，端到端测试能有效地从用户使用的角度，保障系

统的功能不被破坏。

在前面的章节中，笔者已经提到，在经典的测试金字塔中，越往顶端，测试的业务价值越大，但同时测试所花费的成本也越高，所谓成本包括测试环境的搭建、测试数据的准备、测试依赖的安装配置以及发现缺陷的反馈时间等。端到端测试就属于典型的测试三角形顶部的部分。相反，越往底端，测试的成本越低，但其所带来的业务价值越小。单元测试就属于典型的这类测试。

因此，如何有效地定义测试策略，既能有效地覆盖系统的功能，又要避免测试导致的高成本或者长反馈周期问题，是组织或者团队需要不断努力找到的一个平衡点。

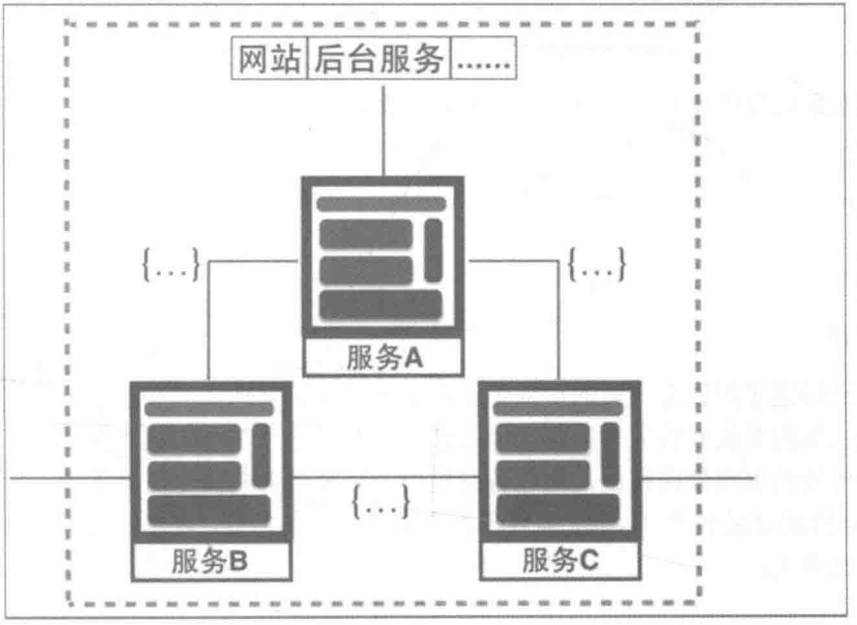


图 13-21　微服务的端到端测试

13.7.3　本节小结

本节中，我们讨论了微服务的端到端测试策略。

对于微服务而言，端到端测试关注的是从用户使用的角度出发，从业务价值的角度出发，对系统功能进行的验证。因为端到端测试的成本较高，因此如何有效地定义端到端测试，既能有效地覆盖系统的功能，又能避免测试导致的高成本或者长反馈周期问题，是组

织或者团队需要不断努力找到的一个平衡点。

13.8 小结

本章中，笔者首先描述了微服务内部的结构，主要包括如下几部分：

- 业务模型
- 业务逻辑
- 模型存储
- 资源定义
- 网关集成

接着从测试实践的角度出发，对微服务的测试策略进行了分析，主要包括：

- 单元测试
- 集成测试
- 接口测试
- 组件测试
- 端到端测试

另外，本章还描述了测试金字塔的价值理论，由于微服务架构本质是基于分布式的系统，同传统的单块架构系统有较大的区别，因此在测试策略上也与传统的单块架构有一定区别。另外，同传统的单块架构相比，微服务在模块划分和业务隔离上更加细致，对自动化测试、自动化运维的要求较高。因此，需要根据微服务本身的特点，合理定义测试策略，找到最有效的测试方式。

第 14 章

使用微服务架构改造遗留系统

本章中，笔者将向大家介绍一个使用微服务改造遗留系统的真实案例。在改造的过程中，服务的构建其实也是遵循第 3 章中的实践，一步一步不断重构而来。

14.1 背景与挑战

随着公司国际化战略的推行以及本土业务的高速发展，后台支撑系统已经不堪重负。在吞吐量、稳定性以及可扩展性上都无法满足日益增长的业务需求。对于每 10 万元额度的合同，从销售团队准备材料、与客户签单、递交给合同部门，再到合同生效大概需要 3.5 人天。随着业务量的快速增长，签订合同的成本急剧增加。

合同管理系统是后台支撑系统中重要的一部分。当前的合同系统是 5 年前使用.NET，基于 SAGE CRM（http://www.sagecrm.com/）二次开发的产品。一方面，系统架构过于陈旧，性能、可靠性无法满足现有的需求。另一方面，功能繁杂，结构混乱，定制的代码与 SAGE CRM 系统耦合度极高。

由于是遗留系统，熟悉该代码的人早已离职多时，新团队对其望而却步，只能做些周边的修补工作。同时，还要承担着边修补测试，边整理逻辑的工作。

14.2 改造策略

在无法中断业务处理的情况下，为了解决当前面临的问题，团队制定了如下策略。

- 最小修改
- 功能剥离
- 数据解耦
- 数据同步
- 迭代替换

14.2.1 最小修改

同当前系统的负责人、用户、产品经理等进行协商，制定对现有系统的修改保护策略：对现有的系统进行更改，譬如对增加新特性、修复缺陷等进行评审与分析，对于大部分非紧急的、非重要的任务，都禁止在原有的系统上进行修改。这样做的目的是用最小的成本保障当前系统能够正常运行，同时将工作重心从现有系统逐渐迁移到新的服务改造中，如图 14-1 所示。

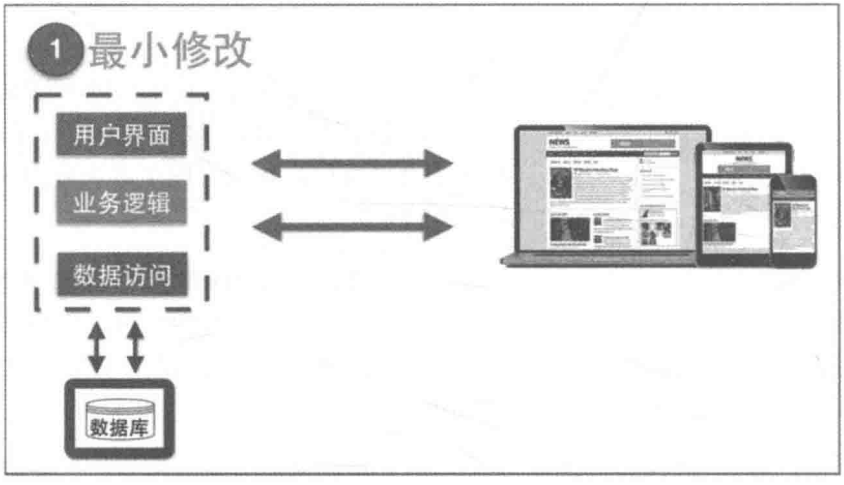


图 14-1 最小修改

14.2.2 功能剥离

在现有合同管理系统的外围，逐步构建功能服务接口，将系统核心的功能分离出来。

同时，使用代理机制，将用户对原有系统的访问，转发到新的服务中，从而解耦原合同系统与用户之间的依赖，如图 14-2 所示。

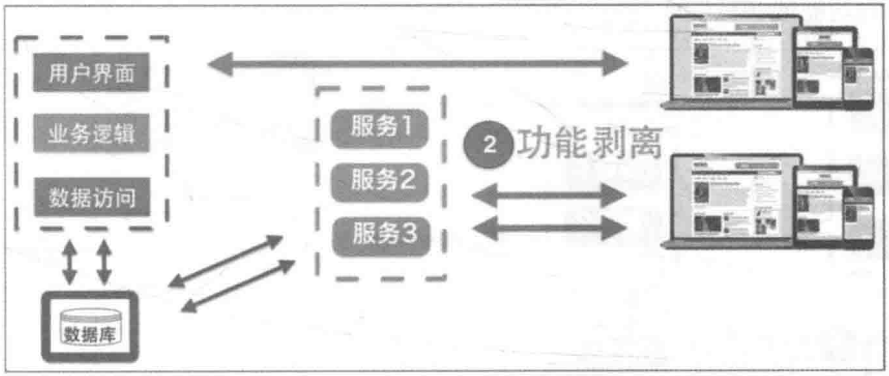


图 14-2 功能剥离

14.2.3 数据解耦

随着部分功能的解耦，已经存在一些微服务能够独立为用户提供功能。这时，逐渐考虑将数据解耦，从原有的单块架构数据库中剥离相关的业务数据。尽量满足对于每个服务，有独立的、隔离的业务数据系统，如图 14-3 所示。

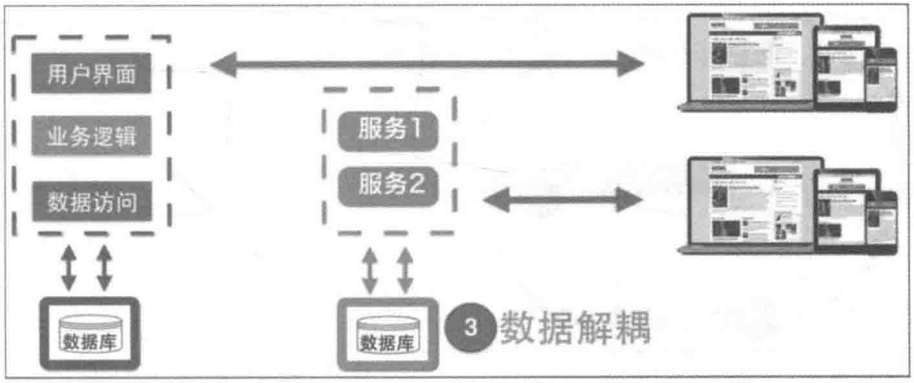


图 14-3 数据解耦

14.2.4 数据同步

通常，对某些遗留的单块架构系统而言，存在着较复杂的业务逻辑，因此改造所需要花费的时间和成本非常大。可能在很长一段时间内，由于新的服务（业务逻辑、数据）独

立出来，导致无法同现有系统协作。在这种情况下，为了持续交付价值，笔者通常会采用ETL（Extract、Transform、Load）的机制，将服务中的业务数据同步到单块架构的数据库中，保障原有的功能能够被继续使用，如图 14-4 所示。

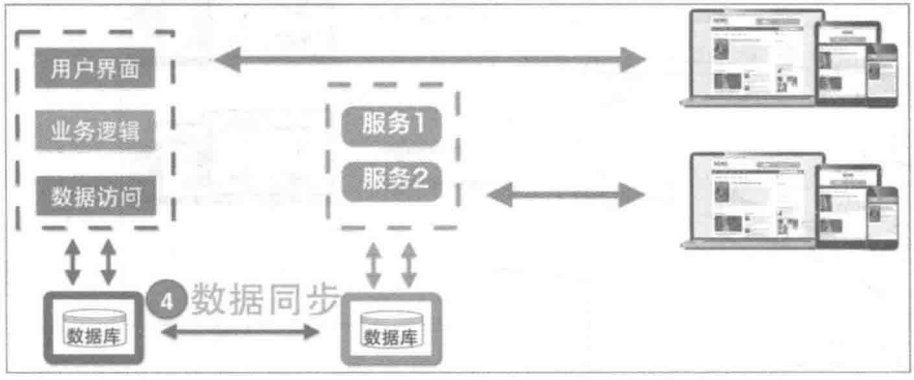


图 14-4　数据同步

14.2.5　迭代替换

通过如上所述的方式不断迭代，逐渐将功能解耦成独立的服务（包括业务逻辑以及业务数据等），如图 14-5 所示。

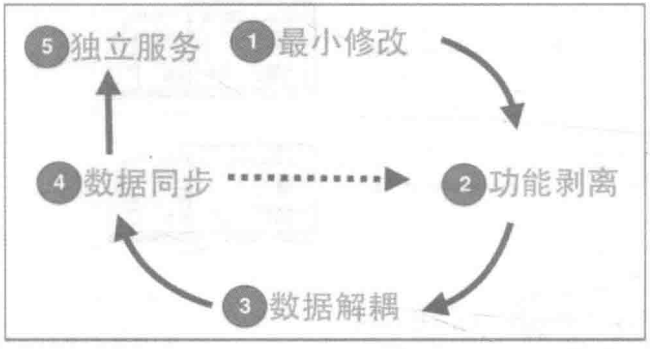


图 14-5　迭代替换

通过不断地迭代替换，最终将原有的合同管理系统替换成使用微服务架构解耦的新的合同管理系统。

14.3 快速开发实践

随着团队对业务的理解加深和对微服务实践的尝试，数个微服务程序已经成功构建出来。不过，问题同时也出现了：对于这些不同的微服务程序而言，虽然具体实现的代码细节不同，但其结构、开发方式、持续集成环境、测试策略、部署机制以及监控和告警等，都有着类似的实现方式。那么如何满足 DRY 原则并消除浪费呢？

在微服务的开发实践中，团队的微服务快速开发模板（Microservice Template）诞生了。

该模板是一个帮助快速构建 Ruby 微服务应用的开发框架，主要包括 4 部分：快速开发模板、代码生成工具、持续集成模板以及一键部署工具，如图 14-6 所示。

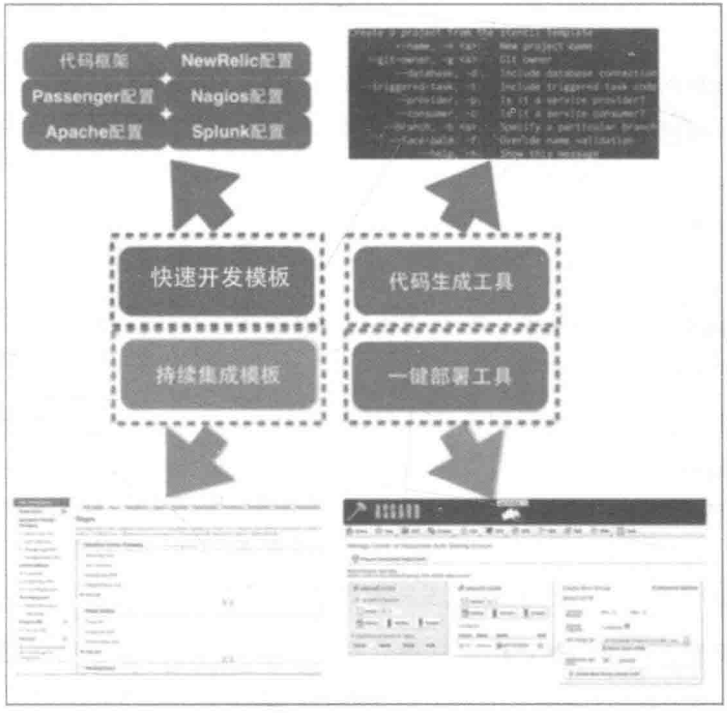


图 14-6 微服务开发模板

14.3.1 快速开发模板

微服务快速开发模板是一个独立的 Ruby 代码工程库，主要包括代码模板以及一组配置文件模板。

代码模板使用 Webmachine 或者 Grape 作为 Web 框架，RESTful 和 JSON 构建服务之间的通信方式，RSpec 作为测试框架。同时，代码模板还定义了一组 Rake 任务，譬如运行测试，查看测试报告，将当前的微服务生成 RPM/Docker 包等。

除此之外，该模板也提供了一组通用的 URL，帮助使用者查看微服务的当前版本、配置信息以及检测该微服务程序是否健康运行等。

```
[
  {
    rel: "index",
     path: "/diagnostic/"
    },
    {
     rel: "version",
     path: "/diagnostic/version"
    },
    {
        rel: "config",
        path: "/diagnostic/config"
     },
     {
        rel: "hostname",
        path: "/diagnostic/hostname"
     },
     {
        rel: "heartbeat",
        path: "/diagnostic/status/heartbeat"
     },
     {
        rel: "nagios",
        path: "/diagnostic/status/nagios"
     }
  ]
```

配置文件模板主要包括 NewRelic 配置、Passenger 配置、Nagios 配置、Apache 配置以及 Splunk 配置。通过定义这些配置文件模板，当把新的微服务程序部署到验收环境或者产品环境时，立刻就可以使用 Nagios、NewRelic 以及 Splunk 等第三方服务提供的功能，帮助我们有效地监控微服务，并在超过初始阈值时获得告警。

14.3.2 代码生成工具

借助代码生成工具，我们能在很短的时间内就构建出一个可以立即运行的微服务应用程序。随着系统越来越复杂，微服务程序的不断增多，开发模板和代码生成工具帮助我们大大简化了创建微服务的流程，让开发人员更关注如何实现业务逻辑并快速验证。

```
Create a project from the template
   --name,       -n:  New project name. eg. things-and-stuff
   --owner,      -o:  Git owner (default: which team or owner)
   --database,   -d:  Include database connection code
   --provider,   -p:  Is it a service provider?
   --consumer,   -c:  Is it a service consumer?
   --branch,     -b:  Specify a particular branch
   --help,       -h:  Show this message
```

如上所示，通过指定不同参数，能创建具有数据库访问能力的微服务程序或者是包含异步队列处理的微服务程序。同时，也可以标记该服务是数据消费者还是数据生产者，能帮助理解多个微服务之间的联系。

14.3.3 持续集成模板

基于持续集成服务器 Bamboo，团队创建了针对微服务模板的持续集成模板工程，并定义了 4 个主要阶段。

- 提交阶段：主要完成代码编译、静态检查以及单元测试等任务。
- 验证阶段：主要完成集成测试、用户行为测试、组件测试或者性能测试等任务。
- 构建阶段：构建 AWS 映像，并为 AWS 映像或者 RPM 包打上不同的版本标签，并存储在相应的服务器上。
- 发布阶段：将 AWS 映像部署到生产环境、测试环境或者类生产环境中。

利用持续集成模板工程，团队仅需花费很少的时间，就可以针对新建的微服务应用程序，在 Bamboo 上快速定义其对应的持续集成环境。

14.3.4 一键部署工具

关于部署，我们使用 Asgard 完成对 AWS 云环境中的资源创建、部署和管理。Asgard 是一套功能强大的基于 Web 的 AWS 云部署和管理工具，由 Netflix 采用 Groovy on Grails

开发，其主要优点有：

- 基于 B/S 的 AWS 部署及管理工具，使用户能通过浏览器直接访问 AWS 云资源，无须设置 Secret Key 和 Access Key。
- 定义了 Application 以及 Cluster 等逻辑概念，清晰、有效地描述了应用程序在 AWS 云环境中对应的部署拓扑结构。
- 在对应用的部署操作中，集成了 AWS Elastic Load Balancer、AWS EC2 以及 AWS Autoscaling Group，并将这些资源自动关联起来。
- 提供 RESTful 接口，能够方便地与其他系统集成。
- 简洁易用的用户接口，提供可视化的方式完成一键部署以及流量切换。

由于 Asgard 对 RESTful 的良好支持，团队实现了一套基于 Asgard 的命令行部署工具，只需如下一条命令，提供应用程序的名称以及版本号，就可自动完成资源的创建、部署、流量切换、删除旧的应用等操作。

```
asgard-deploy [AppName] [AppVersion]
```

同时，基于命令行的部署工具，也可以很容易地将自动化部署集成到 Bamboo 持续集成环境。

通过使用微服务开发模板，大大缩短了团队开发微服务的周期。同时，基于模板，我们定义了一套团队内部的开发流程，可帮助团队的每一位成员理解并快速构建微服务。

14.4 微服务架构下的新系统

经过数个月的努力，团队重新构建了合同管理系统，将之前的产品、价格、销售人员、合同签署、合同审查以及 PDF 生成都定义成了独立的服务接口。相比之前大而全、难以维护的合同管理系统，新的系统由不同功能的微服务组成，每个微服务程序只关注单一的功能。每个微服务应用都有相关的负责人，通过使用 PagerDuty 建立告警通知。每当有监控出现告警的时候，责任人能立即收到消息并快速做出响应。

重建后的系统如图 14-7 所示。

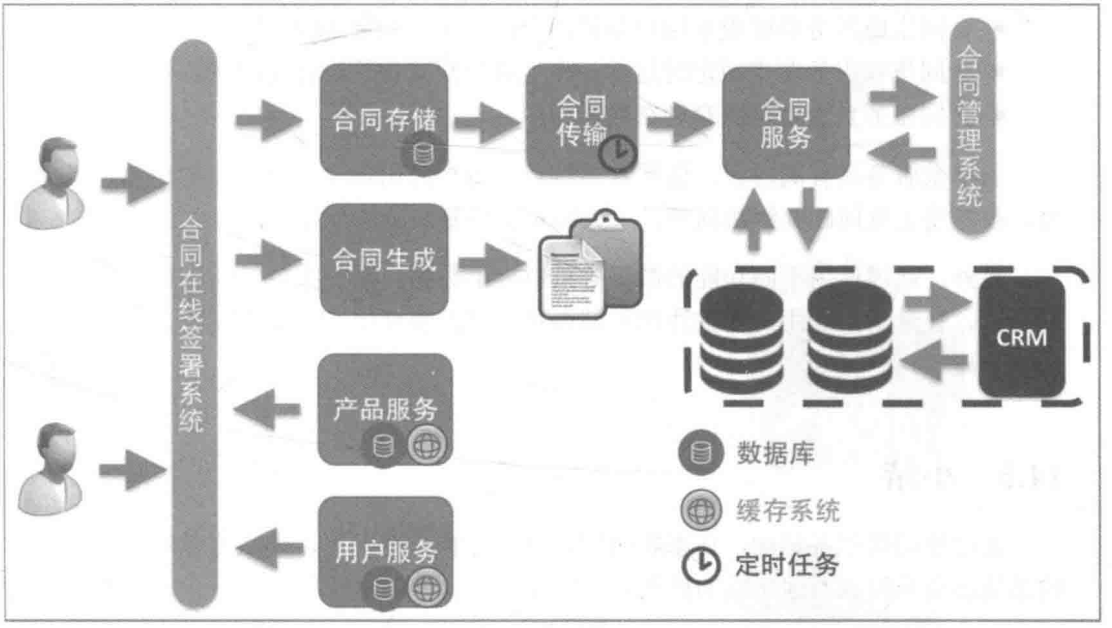


图 14-7　微服务架构下的合同管理系统

其中，主要部分的功能如表 14-1 所示。

表 14-1　合同管理系统各部分介绍

服务/系统名称	主要功能
合同在线签署	提供合同签署的入口
产品服务	提供产品相关的数据，譬如产品分类、产品明细等
用户服务	提供用户相关的数据，譬如折扣信息、优惠信息等
合同存储服务	提供签署的合同存储
合同传输服务	提供合同的传输，将暂存的合同迁移到最终的合同服务数据库里
合同生成服务	生成合同的PDF文档
合同服务	提供合同相关的数据
合同管理系统	提供合同的管理界面

改造后的流程大致如下所示：

- 用户通过访问在线签署系统，从产品服务获取相关产品信息。
- 从用户服务获取用户相关的折扣信息。
- 用户通过合同在线签署系统，在线完成合同的签署。

- 合同生成服务能够根据用户签署的合同，生成 PDF 版本。
- 合同传输服务在夜间定时运行，将生成的合同传输给合同服务。
- 合同服务负责将合同存储在数据库中。

由于微服务具有高内聚，低耦合的特点，每个应用都是一个独立的个体，当出现问题时，很容易定位问题并解决问题，大大缩短了修复缺陷的周期。

另外，通过使用不同功能的微服务接口提供数据，用户接口（UI）部分变成了一个非常简洁、轻量级的应用，更关注如何渲染页面以及表单提交等交互功能，做到了真正的前后端分离。

14.5 小结

通过使用微服务架构，在不影响现有业务运转的情况下，团队有效地将遗留的单块架构系统逐渐分解成不同功能的微服务应用。

同时，通过微服务开发框架，团队能够快速构建不同功能的微服务接口，并能方便地将其部署到验收环境或者生产环境。

最后，得益于微服务架构的灵活性以及扩展性，使得团队能够快速构建低耦合、易扩展、易伸缩性的应用系统。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

在本书中，作者通过把微服务架构和当下最热门的 Docker 容器技术、AWS自动化部署相结合，向读者介绍了具有前瞻性的微服务自动化运维最佳实践。同时详细阐述了微服务化应用的持续交付流程和设计要领，不乏独到见解和技术细节，相信企业CIO、软件设计师、架构师们读完这本书一定会受益匪浅。

郭峰
DaoCloud 联合创始人

本书基于作者在微服务实践方面的经验，总结了微服务架构的诞生、构建、部署以及运维的持续交付过程，理论结合实际、内容丰富，值得一读。

李思
神州专车首席架构师，前Google资深工程师

全面、透彻地讲述了微服务架构与实践，内容丰富、通俗易懂，推荐一读。

孙超
小米互娱资深架构师，前腾讯Qzone技术总监

这是一本从实践出发，用心积累、总结的微服务架构的书籍，读后受益匪浅，推荐阅读。

付重钦
腾讯互娱天美工作室群J1 技术总监

微服务已经成为当下最热门的话题之一，它涉及组织架构、设计、交付、运维等方面的变革，虽然有很多优势，但挑战也不少。之前王磊兄就在InfoQ开设了相关专栏来系统介绍微服务，深受读者喜欢，本书更是专栏的一个有力补充，推荐阅读。

郭蕾
InfoQ主编

王磊作为微服务架构方面的资深专家，在工作期间积累了非常丰富的微服务架构一线实践经验，在本书中，他把自己的经验与理论相结合，深入浅出地讲解了如何实践微服务架构，推荐对此领域感兴趣的工程师仔细研读。

李鑫
京东云平台高级架构师

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

策划编辑：张春雨
责任编辑：刘　舫
封面设计：吴海燕

上架建议：架构/开发/运维

ISBN 978-7-121-27591-3
9 787121 275913 >
定价：65.00元

[General Information]
书名=微服务架构与实践
作者=王磊著
页数=220
SS号=13918691
DX号=
出版日期=2016.01
出版社=北京电子工业出版社