



Mastering Microservices with Java

Java微服务

掌握在生产环境下轻松实现微服务的艺术

[美]Sourabh Sharma 著
卢涛 译



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

Java微服务

随着云平台的采用，企业应用程序的开发从整体应用程序转移到小型、轻量和过程驱动的组件，这种组件称为微服务。微服务是设计可扩展、易于维护的应用程序的下一个重大事件。它们不但使应用程序开发起来更容易，而且还提供了极大的灵活性来以最佳方式利用各种资源。

本书帮助你构建供企业使用的微服务架构实现。从核心概念和框架开始介绍，然后着重讲述大型软件项目的高层次设计，逐渐进入开发环境的设置和前期配置，对微服务架构进行持续集成的部署。然后使用Spring Security实现微服务的安全性，利用REST Java客户端和其他工具有效地执行测试。最后，展示了微服务设计的最佳做法和一般原则，以及如何检测和调试开发过程出现的问题。

本书的受众

如果您是一位熟悉微服务架构的Java开发人员，并对微服务的核心要素和应用程序有合理的知识水平和理解，现在想要深入了解如何有效地实施企业级微服务，那么本书适合您。



本书内容提要

- 使用领域驱动设计方法来设计和实现微服务
- 使用Spring Security实现微服务的安全性
- 部署和测试微服务
- 检测和调试开发过程出现的问题
- 利用JavaScript的Web应用程序来使用微服务
- 学习关于微服务的最佳做法和一般原则

PACKT open source[®]
community experience distilled
PUBLISHING



责任编辑：张春雨
封面设计：李玲

上架建议：编程语言>Java

ISBN 978-7-121-30493-4



9 787121 304934 >

定价：69.00元

Mastering Microservices with Java

Java微服务

[美]Sourabh Sharma 著
卢涛 译

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

微服务是利用云平台开发企业应用程序的最新技术，它是小型、轻量和过程驱动的组件。微服务适合设计可扩展、易于维护的应用程序。它可以使开发更容易，还能使资源得到最佳利用。本书帮助你用 Java 构建供企业使用的微服务架构，内容包括微服务核心概念和框架、大型软件项目的高层次设计、开发环境设置和前期配置、对微服务架构持续集成的部署、实现微服务的安全性、有效地执行测试、微服务设计的最佳做法和一般原则，以及如何检测和调试问题。

本书适合想要了解微服务架构，以及想要深入了解如何有效地实施企业级微服务的 Java 开发人员。

Copyright © 2016 Packt Publishing. First published in the English language under the title 'Mastering Microservices with Java'.

本书简体中文版专有出版权由 Packt Publishing 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2016-7945

图书在版编目（CIP）数据

Java 微服务 / (美) 沙鲁巴·夏尔马 (Sourabh Sharma) 著；卢涛译。—北京：电子工业出版社，2017.1

书名原文：Mastering Microservices with Java

ISBN 978-7-121-30493-4

I . ①J… II . ①沙… ②卢… III . ①JAVA 语言—程序设计 IV . ①TP312.8

中国版本图书馆 CIP 数据核字（2016）第 288000 号

责任编辑：张春雨

印 刷：北京中新伟业印刷有限公司

装 订：北京中新伟业印刷有限公司

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：15.5 字数：322.4 千字

版 次：2017 年 1 月第 1 版

印 次：2017 年 1 月第 1 次印刷

定 价：69.00 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, 88258888。

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

本书咨询联系方式：(010) 51260888-819, faq@phei.com.cn。

作者简介

Sourabh Sharmahas 具有十年以上的产品/应用程序开发经验。他的专长是开发、部署和测试多层 web 应用程序。他喜欢解决复杂的问题，并寻找最佳的解决方案。

在他的职业生涯中，他已成功地为财富 500 强的客户开发和交付了各种独立应用程序和云应用程序，给他们带来很多收益。

Sourabh 还为他的总部设在美国的顶尖企业产品公司发起并开发了一种基于微服务的产品。他在大学时代，即 20 世纪 90 年代后期，开始编写 Java 程序，而且至今仍然热爱这项工作。

审阅者简介

Guido Grazioli 担任过种类繁多的业务应用程序的开发人员、软件架构师和系统集成人员，他的工作跨越多个领域。他是一位复合型软件工程师，对 Java 平台和工具，以及 Linux 系统管理都有深入了解；对 SOA、EIP、持续集成和交付，以及在云环境中的服务业务流程尤其感兴趣。

目录

前言	XV
1 一种解决方法	1
微服务的演变	2
整体式架构概述	3
整体式架构的局限性与它的微服务解决方案的对比	3
一维的可扩展性	6
在出故障时回滚版本	7
采用新技术时的问题	7
与敏捷实践的契合	8
减轻开发工作量——可以做得更好	9
微服务的构建管道	10
使用诸如 Docker 的容器部署	11
容器	11
Docker	12
Docker 的架构	13
Docker 容器	14
部署	14
小结	14
2 设置开发环境	17
Spring Boot 配置	18
Spring Boot 概述	18
把 Spring Boot 添加至 REST 示例	19

添加一个嵌入式 Jetty 服务器.....	21
示例 REST 程序.....	22
编写 REST 控制器类.....	24
@RestController	25
@RequestMapping.....	25
@RequestParam.....	25
@PathVariable	26
制作一个示例 REST 可执行应用程序.....	29
设置应用程序构建.....	30
运行 Maven 工具.....	30
用 Java 命令执行.....	31
使用 Postman Chrome 扩展测试 REST API	31
更多的正向测试场景	34
反向的测试场景	35
NetBeans IDE 安装和设置.....	37
参考资料.....	42
小结.....	42
3 领域驱动设计	43
领域驱动设计基本原理.....	44
组成部分.....	45
普遍存在的语言	45
多层架构.....	45
表示层	46
应用程序层.....	46
领域层	46
基础架构层.....	47
领域驱动设计的工件.....	47
实体	47

值对象	48
服务	49
聚合	50
存储库	52
工厂	53
模块	54
战略设计和原则	55
有界上下文	55
持续集成	56
上下文映射	57
共享内核模式	58
客户和供应商模式	58
顺从者模式	59
反腐层	59
独立方法	59
开放主机服务	60
精馏	60
示例领域服务	60
实体的实现	61
存储库的实现	63
服务的实现	66
小结	67
4 实现微服务	69
OTRS 概述	70
开发和实现微服务	71
餐馆微服务	72
控制器类	73
服务类	76

存储库类.....	79
实体类	82
预订和用户服务	85
注册和发现服务（Eureka 服务）	85
执行.....	87
测试.....	87
参考资料.....	92
小结.....	92
5 部署和测试	93
使用 Netflix OSS 的微服务架构概述	93
负载均衡.....	95
客户端的负载均衡	95
服务器端的负载均衡	98
电路断路器与监控.....	102
使用 Hystrix 的回退方法	102
监控.....	103
设置 Hystrix 仪表板	105
设置 Turbine	107
使用容器部署微服务	109
安装和配置.....	109
具有 4 GB 内存的 Docker 机器.....	110
使用 Maven 构建 Docker 映像	110
使用 Maven 运行 Docker	114
使用 Docker 执行集成测试	115
把映像推送到注册表.....	118
管理 Docker 容器.....	119
参考资料.....	121
小结.....	121

6 实现微服务的安全性	123
启用安全套接字层	123
身份验证和授权	127
OAuth 2.0	127
OAuth 的用法	128
OAuth 2.0 规范——简明详细信息	128
OAuth 2.0 角色	129
OAuth 2.0 客户端注册	131
OAuth 2.0 协议端点	135
OAuth 2.0 授权类型	137
使用 Spring Security 的 OAuth 实现	144
授权码许可	150
隐式许可	153
资源所有者密码凭据许可	154
客户端凭据许可	155
参考资料	155
小结	156
7 利用微服务 Web 应用程序来使用服务	157
AngularJS 框架概述	157
MVC	158
MVVM	158
模块	158
提供程序和服务	160
作用域	161
控制器	161
过滤器	161
指令	162
UI-Router	162

OTRS 功能的开发	163
主页/餐馆列表页	163
index.html	164
app.js	169
restaurants.js	172
restaurants.html	179
搜索餐馆	180
餐馆详细信息与预订选项	181
restaurant.html	181
登录页面	183
login.html	184
login.js	185
预订确认	186
设置 web 应用程序	187
小结	201
8 最佳做法和一般原则	203
概述和心态	203
最佳做法和原则	205
Nanoservice (不推荐)、规模和整体性	205
持续集成和部署	206
系统/端到端测试自动化	207
自我监控和记录	207
每个微服务都使用独立的数据存储区	209
事务边界	210
微服务框架和工具	210
Netflix 开放源码软件 (OSS)	210
构建——Nebula	211
部署和交付——Spinnaker 与 Aminator	211

服务注册和发现——Eureka.....	211
服务沟通——Ribbon	212
电路断路器——Hystrix.....	212
边缘（代理）服务器——Zuul	212
业务监控——Atlas	213
可靠性监控服务——Simian Army.....	213
AWS 资源监控——Edda	214
主机性能监控——Vector	215
分布式配置管理——Archaius.....	215
Apache Mesos 调度器——Fenzo.....	215
成本和云利用率——Ice	216
其他安全工具——Scumblr 和 FIDO	216
参考资料.....	217
小结.....	218
9 故障排除指南	219
日志记录和 ELK 环境	219
简要概述	221
Elasticsearch	221
Logstash	221
Kibana	222
ELK 环境安装	222
安装 Elasticsearch.....	223
安装 Logstash	224
安装 Kibana	225
服务调用关联 ID 的使用	226
让我们看看怎样解决这个问题	226
依赖项和版本	227
循环依赖关系及其影响	227

设计系统时需要分析它.....	227
维护不同版本.....	227
让我们了解更多.....	228
参考资料.....	228
小结.....	228

前言

微服务（**Microservices**）架构是软件架构风格的一种。随着云平台的采用，企业应用程序的开发从整体应用程序转移到小型、轻量和过程驱动的组件，这种组件称为微服务。顾名思义，微服务是指小型服务。它们是设计可扩展、易于维护的应用程序的下一个重大事件。它不但使应用程序开发起来更容易，而且还提供了极大的灵活性来以最佳方式利用各种资源。

本书是帮助你构建供企业使用的微服务实现的实践指南。它还解释了领域驱动设计及其在微服务中的采用。它讲述了怎样构建更小型、更轻量、更快速的服务，同时确保其可以很方便地在生产环境中实施。它也讲述了企业应用程序开发从设计与开发，到部署、测试和实现安全性的完整生命周期。

本书包含的内容

第 1 章，一种解决方法，涉及大型软件项目的高层次设计，在生产环境中所面临的共同问题和解决问题的方法。

第 2 章，设置开发环境，讲述了如何设置开发环境，包括 IDE 和其他开发工具，以及不同的库。本章涉及创建基本项目到设置 spring 引导配置，以建立和发展第一个微服务。

第 3 章，领域驱动设计，通过引用一个示例项目为其余的章节设定基调。它使用此示例项目来驱动服务或应用程序的不同功能和领域组合来解释领域驱动设计。

第 4 章，实现微服务，讲述示例项目从设计到实现的过程。本章不仅涉及编码，还涉及微服务的不同方面——构建、单元测试和包装。在本章末尾，将完成一个可用于部署和使用的示例微服务项目。

第 5 章，部署和测试，讲述了如何采用不同的形式，包括独立部署和使用诸如 Docker

的容器来部署微服务。本章还将演示如何用 Docker 把我们的示例项目部署到诸如 AWS 的云服务上面。你还将掌握使用 REST Java 客户端和其他工具来测试微服务的知识。

第 6 章，实现微服务的安全性，解释如何利用身份验证和授权来保证微服务的安全。身份验证将使用基本身份验证和身份验证令牌来讲述。同样，授权将使用 Spring Security 来解释。本章还将解释常见的安全问题及对策。

第 7 章，利用微服务 Web 应用程序来使用服务，解释了如何利用 Knockout、Require 和 Bootstrap JS 库开发 web 应用程序（UI），构建使用微服务来显示数据的 web 应用程序的原型和一个小型实用程序项目（示例项目）的流程。

第 8 章，最佳做法和一般原则，讲述微服务设计的最佳做法和一般原则。本章还提供了有关使用行业做法进行微服务开发的详细信息和范例。本章还包含微服务实现会产生的错误，以及如何才能避免这类问题的几个例子。

第 9 章，故障排除指南，解释了在微服务及其解决方案的开发过程中会遇到的常见问题。这将帮助你顺利地掌握本书内容，并使学习过程轻松。

学习本书需要具备的条件

为了学习本书，可以使用至少具备 2GB 内存的安装了任何操作系统（Linux、Windows 或 Mac）的计算机；还需要 NetBeans with Java、Maven、Spring Boot、Spring Cloud、Eureka Server、Docker 和 CI/CD 的应用程序。对于 Docker 容器，可能需要一个单独的虚拟机或一个云主机，最好拥有 16GB 或更大的内存。

本书的受众

本书面向熟悉微服务架构，并对核心要素和微服务应用程序有一个合理的知识水平和理解，但现在想要深入了解如何有效地实施企业级微服务的 Java 开发人员。

版式约定

你会发现，本书采用了大量的文本样式，用以区分不同种类的信息。下面是这些样式和解释它们的含义的一些例子。

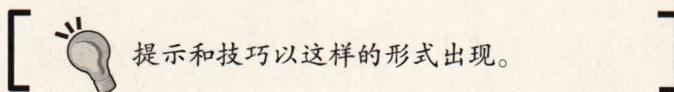
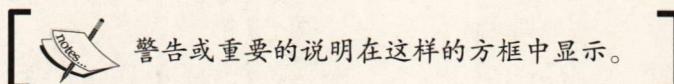
正文中的代码、数据库表名称、文件夹名称、文件名、文件扩展名、路径名、虚拟的 URL、用户输入和 Twitter 句柄如下所示：“可以使用下面的实现创建 Table 实体，并且可以根据自己的需要添加属性”。

代码块的设置，如下所示：

```
public class Table extends BaseEntity<BigInteger> {  
  
    private int capacity;  
  
    public Table(String name, BigInteger id, int capacity) {  
        super(id, name);  
        this.capacity = capacity;  
    }  
}
```

任何命令行输入或输出采用的格式如下：

```
docker push localhost:5000/sourabhh/restaurant-service:PACKT-SNAPSHOT  
docker-compose pull
```



下载示例代码

从 <http://www.broadview.com.cn> 下载所有已购买的博文视点书籍的示例代码文件。

勘误表

虽然我们已经尽力谨慎地确保内容的准确性，但错误仍然存在。如果你发现了书中的错误，包括正文和代码中的错误，请告诉我们，我们会非常感激。这样，你不仅帮助了其他读者，也帮助我们改进后续的出版。如发现任何勘误，可以在博文视点网站相应图书的

页面提交勘误信息。一旦你找到的错误被证实，你提交的信息就会被接受，我们的网站也会发布这些勘误信息。你可以随时浏览图书页面，查看已发布的勘误信息。

参与本书翻译的人员有卢涛、李颖、卢林、陈克非、李洪秋、张慧珍、李又及、卢晓瑶、李阳、陈克翠、刘雯、汤有四。

1

一种解决方法

作为先决条件，我希望你对微服务和软件架构能有一个基本了解。如果不是这样，我建议你用搜索引擎查一下，在解释并详细介绍它们的众多资源中锁定一个。这将帮助你彻底理解其概念和把握本书内容。

读完这本书，你就可以实现微服务，把它用于本地部署或云生产环境的部署，并学习完整的生命周期，包括设计、开发、测试和部署，以及持续集成和部署。本书是专门为实际使用而编写的，用来激发你作为解决方案架构师的智慧。你的学习将帮助你开发和交付任何类型的本地部署下的产品，包括 SaaS、PaaS，等等。我们将主要使用 Java 和基于 Java 的框架工具，如 Spring Boot 和 Jetty，我们还将使用 Docker 作为容器。

 从这里起，本书将用 μ Services 来表示 Microservices（微服务），除了用引号括起来的时候。（译者注：实际上只有少数章节用 μ Services，因此为了统一， μ Services 也译为微服务。）

在本章中，你将学习微服务的永存性及其演化过程。它强调了本地部署和基于云的产品面临的重大问题及微服务如何处理这些问题。本章还解释了在 SaaS、企业级或大型应用程序的开发过程中遇到的常见问题及其解决方案。

在这一章，我们将学习以下主题：

- 微服务和背景简介
- 整体式架构

- 整体式架构的限制
- 微服务提供的灵活性与效益
- 在诸如 Docker 的容器中部署微服务

微服务的演变

Martin Fowler 解释说：

“‘微服务’一词是 2011 年 5 月在威尼斯附近举办的软件架构师讲习班上讨论的，用来描述参与者所见到的作为一种通用的架构风格的东西，其中有许多已经在最近得到研究。在 2012 年 5 月，同一个小组决定把‘微服务’作为这种东西最适当的名称。”

让我们了解一下它在过去几年的发展的一些背景。企业架构更多地从历史悠久的大型机计算，经过客户端-服务器架构（2 层到 n 层）到面向服务的架构（**service-oriented architecture, SOA**）进化。

从 SOA 到微服务的转变不是一个由任何行业组织定义的标准，而是由许多组织实行的实用方法。SOA 最终进化成为微服务。

Netflix 架构师 Adrian Cockcroft，把它形容为：

“细粒度 SOA。所以微服务是强调小型短暂组件的 SOA。”

同样，以下引用 Mike Gancarz(X windows 系统的设计团队成员)的叙述，定义了 UNIX 哲学至高无上的一种感受，它同样适合微服务范式：

“小即是美。”

微服务与 SOA 有许多共同的特点，比如将重点放在服务上，以及如何把一个服务与另一个服务解耦。SOA 通过公开大多基于简单对象访问协议（SOAP）的 API，围绕整体应用程序集成展开进化。因此，中间件，比如企业服务总线（ESB），对 SOA 是非常重要的。微服务的复杂度更低，即使它可能使用消息总线，也仅把它用于消息传输并且不包含任何逻辑。

Tony Pujals 漂亮地定义了微服务：

“在我的心理模型中，我认为微服务是自包含（如在容器中）的轻量进程，它们通过 HTTP 进行通信，用相对较小的工作量与仪式来创建和部署，将只集中在有限领域的 API 提供给它们的使用者。”

整体式架构概述

微服务并不是新事物，它已经流传了很多年。其最近的崛起归功于其声望和知名度的提高。在微服务变得流行之前，被用于开发本地部署和云应用的主要的整体式架构。

整体式架构允许分别开发不同组件，如展示、应用程序逻辑、业务逻辑和数据访问对象（**data access objects, DAO**），然后你要么将它们一起打包在企业归档（**enterprise archive, EAR**）或 web 归档（**web archive, WAR**）文件中，要么将它们存储在某个单独的目录层次结构中（例如，Rails、NodeJS，等等）。

许多著名应用程序，如 Netflix，已经使用微服务架构进行开发。此外，eBay、亚马逊和 Groupon 也都已经从整体式架构发展到微服务架构。

现在，你已经深入了解了微服务的背景和历史，接下来让我们讨论一种传统的方法，即整体式应用程序开发的局限性，并将其与微服务解决局限性的方法做比较。

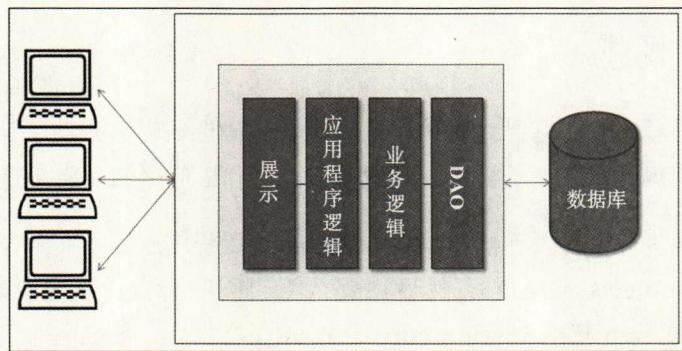
整体式架构的局限性与它的微服务解决方案的对比

正如我们所知，变化是永恒的。人类总是希望寻找更好的解决方案。这就是微服务如何发展成为了它今天的样子的原因，而且它可能会在未来进一步演变。今天，组织使用敏捷方法来开发应用程序，这是一种快节奏的开发环境，在云计算和分布式技术的发明以后，它的规模变得更大。许多人认为整体式架构也可能用于类似用途，并与敏捷方法契合，但微服务还为用于生产的应用程序的许多方面提供了更好的解决方案。

若要了解整体式设计和微服务的差别，让我们举一个餐馆订座应用程序的例子。这个应用程序可能包括很多个服务，如客户、订单、分析等服务，以及常规的组件，如展示和数据库组件。

我们将在这里探索三种不同的设计——传统的整体式设计、使用服务的整体式设计和微服务设计。

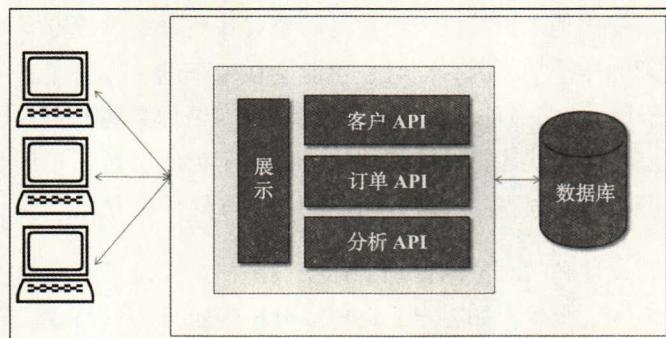
下面的关系图说明了传统的整体式应用程序设计。在 SOA 变得流行之前，这种设计一直被广泛应用：



传统的整体式设计

在传统的整体式设计中，一切都被打包在同一个归档文件中，包括展示代码、应用程序逻辑和业务逻辑代码、DAO 代码、与数据库文件或其他数据源进行交互有关的代码。

在 SOA 产生以后，应用程序开始基于服务来开发，在这种开发模式中，每个组件都为其他组件或外部实体提供服务。下面的关系图描述了提供不同服务的整体式应用程序，在这里，多个服务都被展示组件使用了。所有服务、展示组件或任何其他组件都被打包在一起：

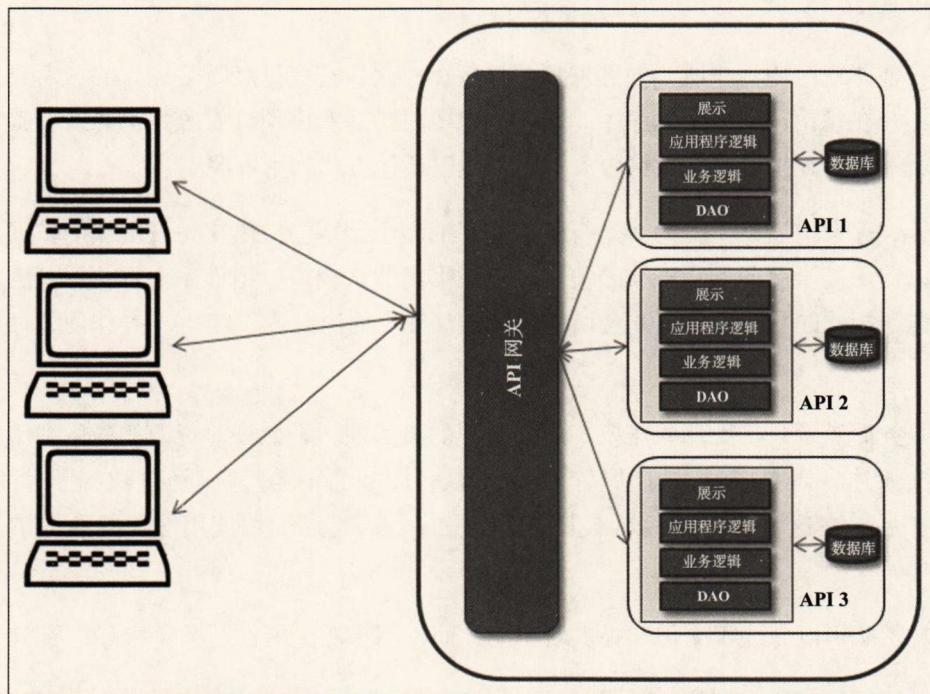


使用服务的整体式设计

下面的第三个设计描绘了微服务。在这里，每个组件都是自主的。每个组件都可以独立开发、构建、测试和部署。在这里，甚至应用程序 UI 组件也可以是一个使用微服务的客户端。在我们的示例中，为了举例说明，在微服务内部使用这个设计的层。

API 网关提供接口，不同的客户端可以通过它访问个别的服务，并解决以下问题：

- 如何让相同的服务给不同的客户端发送不同的响应。例如，预订服务可以发送如下不同的响应、向移动客户端发送最小化的信息、向桌面客户端发送详细信息，它们提供不同的细节，并向第三方客户端发送不同的东西。
- 某个响应可能需要从两个或多个服务中提取信息：



微服务设计

所有的示例设计关系图都是非常高层次的设计，观察了这些之后，你可能会发现，在整体式设计中，各个组件全被捆绑在一起，并且互相紧密耦合。

所有的服务都是相同的捆绑包的一部分。同样，在第二个设计的关系图中，可以看到

第一个关系图的变体，在这个设计中，所有服务都可以有其自己的层，并形成不同的 API，但如图所示，这些也全都捆绑在一起。

相反，在微服务中，设计组件并未捆绑在一起，并且具有松散的耦合。每个服务都有其自己的层，而 DB 被捆绑在一个单独的归档文件中。所有这些已部署的服务都提供它们特定的 API，例如客户、预订或分析。这些 API 都是准备就绪可供使用的，甚至连 UI 也是单独部署，并使用微服务设计的。因此，它比对应的整体式设计具备更多优点。尽管如此，我还是要提醒你，有一些特殊的情况下，整体式的应用程序开发模式是非常成功的，比如 Etsy 和对等电子商务 web 应用程序。

一维的可扩展性

随着整体式应用程序所有的部件都被打包在一起，它在扩展时是庞大的，需要扩展一切。例如，在餐馆订座应用程序中，即使你想要只扩展餐桌预订服务，也要扩展整个应用程序，而不能单独扩展餐桌预订服务。它未能以最佳方式利用资源。

此外，这个扩展是一维的。运行应用程序的多个副本会提供交易量增加的扩展。运营团队可以使用基于服务器集群或云负载的负载平衡器来调整应用程序副本的数量。每个副本都将访问相同的数据源，因此会增加内存的消耗，而由此产生的 I/O 操作使缓存不那么有效。

微服务提供了只扩展那些需要扩展的服务的灵活性，它允许对资源的最优利用。正如我们前文所述，当需要时，可以只扩展餐桌预订服务而不会影响任何其他组件。它还允许二维扩展，在这里我们能做到不但增加交易量，而且还增加使用缓存的数据量（平台扩展）。

开发团队可以专注于交付和提供新功能，而不用担心扩展性问题（产品扩展）。

正如我们先前所看到的，微服务可以帮助你扩展平台、人员和产品维度。在这里的人员扩展指的是，取决于微服务的具体开发和重点需要，扩大或缩小团队规模。

微服务开发使用 REST 式的 web 服务开发，REST 的服务器端是无状态的，使得在这个意义上，它是可扩展的，这意味着服务器之间没有太多的通信，从而它可以水平扩展。

在出故障时回滚版本

因为整体式应用程序是打包在相同的归档文件或包含在单个目录中的，所以这阻碍了代码的模块化部署。例如，很多人可能都曾遇到过由于一项功能的故障，致使整个版本延迟发布的痛苦。

要解决这种问题，微服务给我们提供仅回滚那些出故障的功能的灵活性。这是一种非常灵活和富有成效的方法。例如，假设你是在线购物门户开发团队的成员，想要开发一个基于微服务的应用程序。可以将你的应用程序基于不同的领域，如商品、付款、购物车等进行划分，并将所有这些组件都作为单独的程序包打包。一旦你已经分别部署所有这些软件包，这些都将作为单个组件，可以被单独开发、测试和部署，并调用微服务。

现在，让我们看看它是如何帮你解决问题的。假设在某个生产版本中发布新的功能、增强功能和 bug 修复后，你发现支付服务中有缺陷需要立即修复。由于你已经使用基于微服务的架构，可以仅回滚付款服务而不是回滚整个版本，如果你的应用程序架构允许，还可以只对微服务付款服务应用修补程序，而不会影响其他服务。这不仅允许你妥善处理故障，还有助于迅速向客户交付功能或修补程序。

采用新技术时的问题

整体式应用程序主要是基于在项目或产品开发过程最初主要使用的技术而开发和加强的。这使得很难在开发的后期或在产品处于成熟的状态后（例如，几年后）引进新技术。此外，同一个项目中的不同模块依赖于同一个库的不同版本，使这更具挑战性。

技术在逐年进步。例如，你的系统可能会使用 Java 设计，几年后，因为业务需要或为了利用新技术的优势，你又想要使用 Ruby on rails 或 NodeJS 开发新的服务。整体式的现有应用程序将很难利用新技术。

这不只是代码级集成，还与测试和部署相关。虽然可以通过重新编写整个应用程序来采用一项新技术，但这要耗费大量时间并且具有很高的风险。

另一方面，由于微服务是基于组件开发和设计的，它给我们提供了在开发中使用无论是新技术还是旧技术的任何技术的灵活性。它并不限制你使用特定的技术，这给你提供

了一种开发和工程活动的新范式。可以在任何时候使用 Ruby on Rails、NodeJS 或任何其他技术。

那么，它是如何实现的呢？嗯，非常简单。基于微服务的应用程序代码并不打包成一个单一的归档文件，也并不存在储在单个目录中。每个微服务都有其自己的文件，并且单独部署。一个新服务可以在隔离的环境中开发，并可以没有任何技术问题地被测试和部署。正如你所知，微服务还拥有自己独立的进程，它服务于它的目标时不存在任何冲突，如共享紧密耦合的资源，并且进程也保持独立。

因为根据定义，微服务是小型、自包含的功能，所以它提供了低风险地尝试一项新技术的机会。而对整体式系统而言，绝对不能这样。

还可以把你的微服务作为开放源码软件对外提供，所以它可由其他人使用。因此，如果需要，它能够与封闭源码的专有服务进行交互，而使用整体式应用程序，是不可能这么做的。

与敏捷实践的契合

使用敏捷实践开发整体式应用程序是没有问题的，而且这些应用程序正在被开发中。**持续集成（Continuous Integration, CI）** 和**持续部署（Continuous Deployment, CD）** 也可以使用，但是，问题是——它能有效地使用敏捷实践吗？让我们来研究以下几点：

- 例如，当事件互相依赖的可能性很大，并可能有不同的场景时，一个事件只有在它所依赖的事件都已完成后再发生。
- 随着代码量的增加，生成应用程序需要更多的时间。
- 大型整体式应用程序要实现频繁部署是一项困难的任务。
- 即使更新单个组件，也将不得不重新部署整个应用程序。
- 重新部署可能导致已经运行的组件出问题，例如作业调度器可能会更改，无论组件是否对它产生影响。
- 如果单个更改的组件不能正常工作，或如果它需要更多的修正，重新部署的风险可能会增加。
- UI 开发人员总是需要更多的重新部署，这对于整体式的大型应用程序是相当危险和费时的。

前面的问题都可以通过微服务很容易地解决。例如，UI 开发人员可能拥有他们自己的 UI 组件，这些组件能够单独地开发、构建、测试和部署。同样，其他的微服务也可能独立部署，并且因其自主的特点，降低了系统故障风险。它用于开发的另一个优势是，UI 开发人员可以使用 JSON 对象和模拟 Ajax 调用来开发 UI，这些能够以隔离的方式占用。开发完成后，开发人员可以使用实际的 API 并测试这些功能。总之，可以说，微服务开发是快捷的，它符合企业的增量需求。

减轻开发工作量——可以做得更好

一般来说，大型整体式应用程序的代码是开发人员最难理解的，并且新开发人员需要相当长的时间，才能成为生产力。甚至把大型整体式应用程序加载到 IDE 中都是麻烦的，它会降低 IDE 的运行速度，使开发人员的生产力降低。

大型整体式应用程序中的更改很难实施，并花费更多的时间。这是由于有大量的代码库，并且，如果未恰当并彻底地执行影响分析，存在 bug 的风险会很高。因此，执行全面影响分析成为开发人员实施更改之前的先决条件。

在整体式应用中，随着时间的推移，当所有组件都捆绑在一起时，依赖关系就建立了。因此，随着代码更改量（修改后的代码的行数）的增加，与代码更改相关联的风险呈指数级上升。

当一个代码库非常庞大并且有 100 多个开发人员正在它上面工作时，因为前面提到的原因，建立产品和实现新功能变得很困难。你需要确保一切都到位，并且一切都协调一致。在这种情况下，精心设计和记录的 API 的帮助很大。

Netflix，一家互联网流媒体按需提供商，在让约 100 人开发他们的应用程序时遇到了问题。然后，他们使用云平台，并将其应用程序分解成单独的小块。这些最终成为了微服务。提高速度和灵活性以便部署团队独立的愿望导致微服务不断壮大。

微型组件被制作成松耦合的，这归功于 API 的公开，这些 API 可以被持续地集成和测试。借助微服务的连续发布周期，变化被控制得很小，开发人员可以迅速利用它们执行回归测试，然后复审一遍并修复最终发现的瑕疵，以减少部署的风险。这会获得更快的速度与较低的相关风险。

由于功能分离和单一责任原则，微服务使团队非常有成效。可以在网上找到大量的大型项目由人数很少，如八至十个开发人员的团队开发完成的实例。

开发人员对于更少量的代码可以更集中注意力，由此产生的功能的更好实现，导致与产品的用户有更高的移情关系。这有助于功能的实现有更好的动机和明确性。与用户的移情关系会产生一个较短的反馈回路，以及更好、更迅速确定优先次序的功能管道。较短反馈回路使得缺陷检测也更快。

每个微服务团队都独立地工作，而无须与更多的观众协调就可以实现新功能或想法。在微服务设计中，端点故障处理也很容易实现。

最近，在一次会议中，一个团队展示了他们是如何在 10 周内开发出一个具有超级类型（Uber-type）跟踪功能，并包括 iOS 和 Android 应用程序基于微服务的传输-跟踪的应用程序。一家大型咨询公司针对相同的应用程序为他的客户给出 7 个月的估计开发时间。它表明了微服务与敏捷方法和 CI/CD 的契合方式。

微服务的构建管道

微服务也可以使用诸如 Jenkins、TeamCity 等流行的 CI/CD 工具建立和测试。它与整体式应用程序的构建方式是非常相似的。在多个微服务中，每个微服务都被当作一个小型应用程序。

例如，一旦把代码提交到存储库（SCM）中，CI/CD 工具就触发构建过程：

- 清理代码
- 代码编译
- 执行单元测试
- 建立应用程序归档文件
- 在开发、QA 等各种服务器上部署
- 执行功能和集成测试
- 创建映像容器
- 任何其他步骤

然后，更改 `pom.xml`（在 Maven 的情况下）中的快照（SNAPSHOT）或发行

(RELEASE) 版本的版本构建触发器，会构建工件，如同在正常构建触发器中所述的那样。将工件发布到 artifacts 存储库。在存储库中标记此版本。如果你使用容器映像，那么就会生成容器映像。

使用诸如 Docker 的容器部署

由于微服务的设计，需要有一个提供灵活性、敏捷性和平滑度的环境，以支持持续集成和部署，以及发布。微服务部署需要速度、隔离管理和敏捷的生命周期。

产品和软件也可以使用多式联运集装箱 (intermodal-container) 模型的概念来发布。多式联运集装箱是一个大型的标准化容器，用来执行多式联运货物运输。它允许货物使用各种交通工具——卡车、铁路或轮船运输，而不需要卸载并重新加载。这是储存和运输货物有效和安全的方式。它解决了航运的一个难题，这以前一直是一个耗时的劳动密集型过程，而重复的处理往往会打破易碎物品。

航运集装箱封装它们的内容。同样，软件容器也开始用于封装它们的内容（产品、应用程序、依赖关系，等等）。

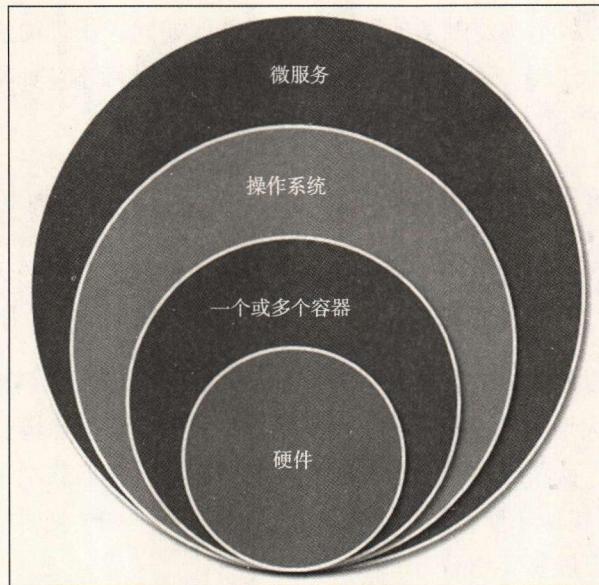
以前，虚拟机 (virtual machine, VM) 被用于创建可以在需要的地方部署的软件映像。后来，诸如 Docker 的容器变得更受欢迎，因为它们同时与传统虚拟站系统和云环境兼容。例如，在开发人员的笔记本电脑上部署超过两个 VM 不太现实。建立和启动一个 VM 机器通常是 I/O 密集型的操作，因此也相当缓慢。

容器

容器（例如，Linux 容器）提供轻量级运行时环境，该环境由虚拟机的核心功能和操作系统的隔离服务组成。这使得包装和执行微服务变得简单和顺利。

如下图所示，容器作为应用程序（微服务）在操作系统中运行。操作系统位于硬件的顶部，每个操作系统都可以有多个容器，使用一个容器运行应用程序。

容器能利用操作系统的内核接口，如 cname 和命名空间，它们允许共享相同内核的多个容器可以完全互相隔离地同时运行。这具有无须为每个用途而完成一个 OS 安装的好处，从而可以消除开销。这也使得硬件得到最佳的利用。



容器的层次关系图

Docker

容器技术是当今发展最迅速的技术之一，而 Docker 引领这一技术。Docker 是一个开放源码项目，2013 年发布。2013 年 8 月其交互式教程推出后，有 1 万个开发人员尝试使用它。1.0 版本在 2013 年 6 月发布的时候，被下载了 275 万次。许多大公司，如微软、RedHat、惠普、OpenStack，以及诸如亚马逊网络服务、IBM 和谷歌等服务提供商，都已与 Docker 签署了伙伴关系协定。

正如我们刚才提到的，Docker 也利用了 Linux 内核的功能，例如 cgroups 和命名空间，以确保资源隔离，并把应用程序及其依赖项打包在一起。这种依赖项的包装使应用程序能够跨不同 Linux 操作系统/发行版本正常运行，从而实现可移植性级别的支持。此外，这种可移植性允许开发人员用任何一种语言开发一个应用程序，然后轻松地把它从笔记本电脑部署到测试或生产服务器上。

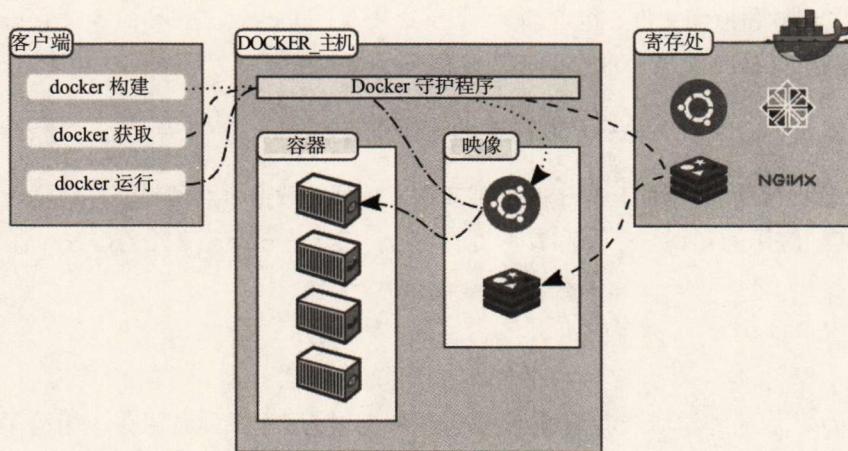
[ Docker 可原生地在 Linux 上运行。然而，利用 VirtualBox 和 boot2docker，Docker 也可以运行在 Windows 和 Mac OS 上。]

容器仅由应用程序及其依赖项组成，依赖项包括基本的操作系统。这使得它轻量，并在资源利用率方面保持高效。开发人员和系统管理员对容器的可移植性和资源的有效利用感兴趣。

Docker 容器中的所有程序都可原生地在主机上执行，并直接使用主机内核。每个容器都具有其自己的用户命名空间。

Docker 的架构

如 Docker 文档指出的，Docker 架构使用客户机-服务器架构。如下面的图例所示（源自 Docker 的网站），Docker 客户端主要是由最终用户使用的用户界面，客户端与 Docker 守护进程之间往复通信。Docker 守护程序承担构建、运行和分发你的 Docker 容器的繁重任务。Docker 客户端和守护进程既可以在同一个系统中，也可以在不同的机器上运行。Docker 客户端与守护进程通过套接字或通过基于 REST 的 API 进行通信。Docker 寄存处是公共或私有 Docker 映像存储库，你可以从中上传或下载映像，例如 Docker 枢纽（hub.docker.com）就是一个公共的 Docker 寄存处。



Docker的架构

Docker 的主要组件是一个 Docker 映像和一个 Docker 容器。

Docker 映像

Docker 映像是一个只读的模板。例如，映像可以包含安装了 Apache web 服务器和 web 应用程序的 Ubuntu 操作系统。Docker 映像是 Docker 生成组件。映像用于创建 Docker 容器。Docker 提供简单的方式来生成新的映像或更新现有的映像。你还可以使用由其他人创建的映像。

Docker 容器

Docker 容器是从某个 Docker 映像创建的。Docker 的工作是使得容器只能看到它自己的进程，并有分层到主机文件系统和网络栈的自己的文件系统，它通过管道连接到主机网络栈。Docker 容器可以被运行、启动、停止、移动或删除。

部署

使用 Docker 部署的微服务处理三个部分的问题：

1. 应用程序打包，例如 jar
2. 使用 jar 和依赖文件，包含 Docker 指令文件、Docker 文件和命令 `docker build` 来构建 Docker 映像。这有助于反复创建映像。
3. 使用 `docker run` 命令从新构建的映像执行 Docker 容器。

上述信息将帮助你了解 Docker 的基础知识。在第 5 章部署和测试，你将学习到更多关于 Docker 实际应用的知识。来源和参考资料：<https://docs.docker.com>。

小结

在这一章，你已经学会或者演练了从传统整体式应用程序到微服务应用程序的大型软件项目高层次设计。你也了解了微服务简史、整体式应用程序的限制和好处，及微服务提供的灵活性。我希望这一章除能帮助你理解整体式应用程序在生产环境中面临的共同问题，而微服务可以解决这些问题。你还了解了轻量级和高效的 Docker 容器，并看到容器化为何是一种很好的部署微服务的典型方法。

在下一章中，你会学习使用 IDE 和其他开发工具，针对不同的库来设置开发环境的相关知识。我们将面对创建基本项目和设置 Spring Boot 配置，建立和开发我们第一个微服务。在这里，我们将采用 Java 8 作为开发语言，并把 Spring Boot 用于我们的项目。

2

设置开发环境

本章着重介绍开发环境的设置和配置。如果你熟悉这些工具和库，那么你可以跳过这一章，继续阅读第 3 章，在第 3 章你可以研究领域驱动设计的课题。

这一章将涵盖以下主题：

- Spring Boot 配置
- 示例 REST 程序
- 生成安装程序
- 使用 Postman Chrome 扩展执行 REST API 测试
- NetBeans——安装和设置

本书将仅使用开放源代码工具和框架来举例和编码，也将使用 Java 8 作为其编程语言，而应用程序框架将基于 Spring 框架。本书使用 Spring Boot 来开发微服务。

NetBeans 提供最先进的、同时支持 Java 和 JavaScript 的集成开发环境（**NetBeans Integrated Development Environment, IDE**），足以满足我们的需求。它多年来演变得很多，并已内置支持大多数本书使用的技术，如 Maven、Spring Boot 等。因此，建议你使用 NetBeans IDE。然而，你可以随意使用任何 IDE。

我们将使用 Spring Boot 来开发 REST 服务和微服务。在本书中选择最流行的 Spring 框架——Spring Boot 或其子集 Spring Cloud 是有意为之。因为这样我们就不需要从头开始编

写应用程序，该框架对云应用程序大部分的东西都提供默认的配置。在 Spring Boot 配置一节中，我们对 Spring Boot 做了概述。如果你是 Spring Boot 初学者，那么这会对你有帮助。

我们将使用 Maven 作为我们的构建工具。与 IDE 的情况一样，可以随意使用任何构建工具，例如 Gradle 或 Ant，我们将使用嵌入式的 Jetty 作为 web 服务器，但另一种选择是使用嵌入式的 Tomcat web 服务器。我们还将使用 Chrome 的 Postman 扩展来测试 REST 服务。

我们将从 Spring Boot 配置开始介绍。如果你是 NetBeans 初学者或者正面临设置环境的问题，可以参考最后一节 NetBeans IDE 安装部分的解释，否则完全可以跳过该节。

Spring Boot 配置

为了开发最先进且可用于生产的特定于 Spring 的应用程序，Spring Boot 是一个明显的选择。其网站还说明了其真正的优势：

“需要坚持构建可用于生产的 Spring 应用程序的观念。Spring Boot 约定优先于配置，并且让你尽可能快地启动并运行。”

Spring Boot 概述

Spring Boot 是 Pivotal 创建的优异的 Spring 工具，它于 2014 年 4 月发布（GA）。它基于 SPR-9888 (<https://jira.spring.io/browse/SPR-9888>) 的请求被开发出来，该请求标题为“对无容器 web 应用程序架构的改进支持”。

你一定会想，为什么是无容器呢？因为，今天的云环境或 PaaS 都提供基于容器 web 架构的大多数功能，如可靠性、可管理性或可扩展性。因此，Spring Boot 侧重于使其本身成为超轻量的容器。

Spring Boot 预先配置使得开发可用于生产的 web 应用程序变得非常容易。Spring Initializer（初始值设定项）(<http://start.spring.io>) 是一个网页，可以在其中选择构建工具如 Maven 或 Gradle，还可以选择项目元数据，如组、工件和依赖项。当填充了所需的字段后，只需单击 **Generate Project**（生成项目）按钮，它就会提供一个可以用于生产

应用程序的 Spring Boot 项目。

在此页上，默认打包选项是 jar。我们也会在微服务开发中使用 jar 包装。原因非常简单：它使得微服务的开发更容易。只要想想，管理和创建每个微服务都运行在其自己的服务器实例上的基础设施会多么困难，就明白了。

Josh Long 在某个 Spring IOs 的谈话中分享了他的观点：

“最好生成 Jar，而不是 War。”

稍后，我们将使用 Spring Cloud，它是在 Spring Boot 之上的一个包装。

把 Spring Boot 添加至 REST 示例

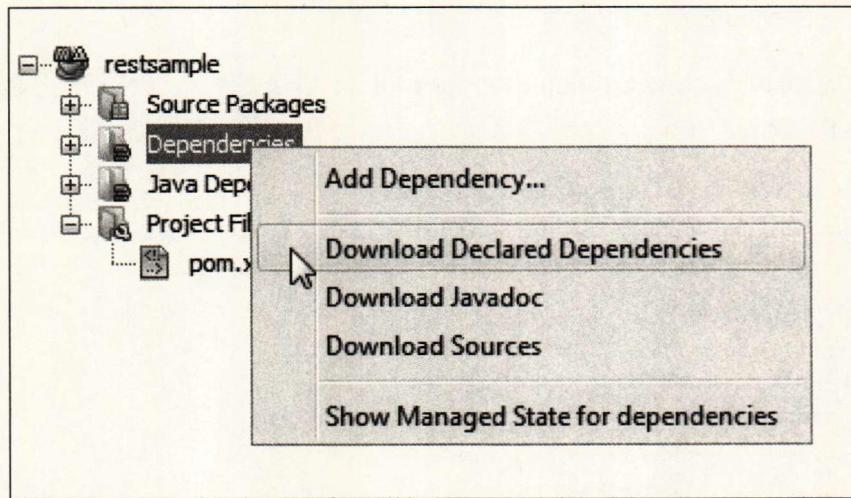
在编写本书的时候，Spring Boot 提供了 1.2.5 发布版本，你可以使用最新的发布版本。Spring Boot 使用 Spring 4 (4.1.7 版本)。

打开 pom.xml (可以从 **restsample | Project Files** (项目文件) 下找到)，把 Spring Boot 添加到 REST 示例项目中：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.packtpub.mmj</groupId>
    <artifactId>restsample</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.2.5.RELEASE</version>
    </parent>
    <properties>
```

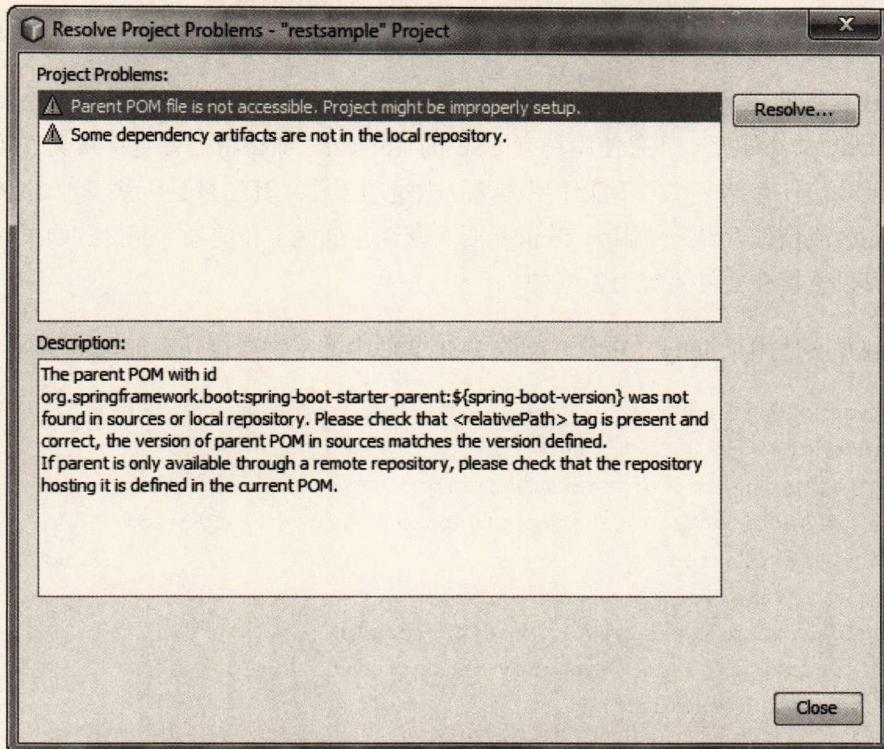
```
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<spring-boot-version>1.2.5.RELEASE</spring-boot-version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>${spring-boot-version}</version>
    </dependency>
</dependencies>
</project>
```

如果你是第一次添加这些依赖项，需要在如下所示的 **Projects** 窗格的 **restsample** 项目中通过鼠标右键单击 **Dependencies** 文件夹来下载依赖项。



下载NetBeans的Maven依赖项

同样，要解决项目问题，用鼠标右键在 NetBeans 项目 **restsample** 上单击，并选择 **Resolve Project Problems...**（解决项目问题），它将打开如下图所示的对话框。单击 **Resolve...** 按钮来解决问题。



解决项目问题对话框

[ 如果在代理服务器后面使用 Maven，那么需要更新<NetBeans 安装目录>\java\maven\conf\settings.xml 中的代理服务器设置。可能需要重新启动 NetBeans IDE 才能使更新的设置生效。]

如果在本地 Maven 存储库中未提供声明的依赖项和传递依赖项，那么前面的步骤将从远程 Maven 资源库中下载所有必需的依赖项。如果是第一次下载依赖项，那么可能需要花一些时间，这具体取决于你的 Internet 速度。

添加一个嵌入式 Jetty 服务器

Spring Boot 默认情况下提供 Apache Tomcat 作为嵌入式应用程序的容器。本书将在使用 Apache Tomcat 的地方使用嵌入式 Jetty 应用程序容器。因此，我们需要添加 Jetty 应用程

序容器依赖项，以支持 Jetty web 服务器。

Jetty 还允许使用 classpath（类路径）读取密钥或信任存储区，也就是说，不需要把这些存储在 JAR 文件外面。如果使用具有 SSL 的 Tomcat，那么将需要直接从文件系统访问密钥存储区或信任存储区，但不能使用类路径做这项工作。其结果是无法读取 JAR 文件中的密钥存储区或信任存储区，因为 Tomcat 要求密钥存储区（和信任存储区，如果你正在使用它）是可以直接在文件系统上访问的。

此限制并不适用于 Jetty，它允许读取 JAR 文件中的密钥或信任存储区：

```
<dependencies>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jetty</artifactId>
    </dependency>
</dependencies>
```

示例 REST 程序

我们将使用一个简单的方法来建立一个独立的应用程序。我们把这一切都打包成一个由 main() 方法驱动的单独的可执行 JAR 文件。在此过程中，使用 Spring 对嵌入 Jetty servlet 容器的支持作为 HTTP 运行时环境，而不是将它部署到一个外部的实例中。因此，我们将创建可执行的 JAR 文件来代替需要在外部 web 服务器上部署的 WAR 文件。

现在，当你在 NetBeans IDE 中准备好 Spring Boot 后，就可以创建你的示例 web 服务了。你将创建一个数学 API，它可以执行简单的计算并生成 JSON 格式的结果。

下面让我们讨论一下如何调用 REST 服务并获取其响应。

此服务将处理对/calculation/sqrt 或/calculation/power 等的 GET 请求。GET 请求应该返回 200 OK 响应，以及在正文中表示给定数字的平方根的 JSON。它看起来应该像下面这样：

```
{  
    "function": "sqrt",  
    "input": [  
        "144"  
    ],  
    "output": [  
        "12.0"  
    ]  
}
```

input 字段是用于平方根函数的输入参数，而内容是结果的文本表示形式。

可以采用普通旧式 Java 对象（Plain Old Java Object, POJO）创建一个资源表示类对表示法进行建模，这个资源表示类使用包含字段、构造函数、setter 和 getter 的普通旧式 Java 对象 Plain Old Java Object (POJO)，用于输入、输出和函数的数据。用它来对表示法进行建模：

```
package com.packtpub.mmj.restsample.model;  
  
import java.util.List;  
  
public class Calculation {  
  
    String function;  
    private List<String> input;  
    private List<String> output;  
  
    public Calculation(List<String> input, List<String> output, String  
function) {  
        this.function = function;  
        this.input = input;  
    }
```

```
        this.output = output;
    }

    public List<String> getInput() {
        return input;
    }

    public void setInput(List<String> input) {
        this.input = input;
    }

    public List<String> getOutput() {
        return output;
    }

    public void setOutput(List<String> output) {
        this.output = output;
    }

    public String getFunction() {
        return function;
    }

    public void setFunction(String function) {
        this.function = function;
    }
}
```

编写 REST 控制器类

Roy Fielding 在其博士论文中提出并定义了术语 **REST (Representational State Transfer, 具象状态转换)**。REST 是一种用于诸如 WWW 的分布式超媒体系统的软件架构风格，RESTful (REST 式) 是指那些符合 REST 架构属性、原则和约束的系统。

现在，你将创建 REST 控制器来处理计算资源。在 Spring REST 式的 web 服务实现中，控制器负责处理 HTTP 请求。

@RestController

`@RestController` 是 Spring 4 中引入的用于 `resource` 类的类级注解。它是 `@Controller` 与 `@ResponseBody` 的组合，因此类返回一个域对象，而不是视图。

在以下代码中，可以看到 `CalculationController` 类通过返回 `calculation` 类的一个新实例来处理对 `/calculation` 的 GET 请求。

我们将实现两个计算资源的 URL——将平方根 (`Math.sqrt()`) 函数实现为 `/calculation/sqrt` URL，以及将幂 (`Math.pow()`) 函数实现为 `/calculation/power` URL。

@RequestMapping

`@RequestMapping` 注解在类级别使用，它将 `/calculation` URI 映射到 `CalculationController` 类，它确保对 `/calculation` 的 HTTP 请求被映射到 `CalculationController` 类。基于使用 URI 的注解 `@RequestMapping` 定义的路径 (`/calculation` 后缀，例如 `/calculation/sqrt/144`)，会映射到各自的方法。在这里，映射到 `/calculation/sqrt` 的请求被映射到 `sqrt()` 方法，而映射到 `/calculation/power` 的请求被映射到 `pow()` 方法。

你可能也已观察到我们并没有定义这些方法会使用什么请求方法 (GET/POST/PUT 等等)。`@RequestMapping` 注解默认映射所有的 HTTP 请求方法。可以通过使用 `RequestMapping` 的方法属性来使用特定的方法。例如，可以通过以下方式使用 POST 方法来编写 `@RequestMethod` 注解：

```
@RequestMapping(value = "/power", method = POST)
```

为传递过程中的参数，此示例说明请求参数和路径参数都分别使用 `@RequestParam` 和 `@PathVariable` 注解。

@RequestParam

`@RequestParam` 负责将查询参数绑定到控制器方法的参数。例如，`QueryParam` 底数和指数分别绑定到 `CalculationController` 的 `pow()` 方法的参数 `b` 和参数 `e` 上。`pow()` 方法的两个查询参数都是必需的，因为我们未对它们使用任何默认值。可以使用

@RequestParam 的 defaultValue 属性来设置查询参数的默认值，例如 @RequestParam (value="base", defaultValue="2")，在这里，如果用户未传递查询参数底数，那么底数将使用默认值 2。

如果没有定义 defaultValue，并且用户未提供请求的参数，则 RestController 返回 HTTP 状态代码 400，以及 400 所需的字符串参数底数不存在（**Required String parameter base is not present**）的消息。如果缺少多个请求参数中的一个，它总是使用所需的第一个参数的引用：

```
{  
    "timestamp": 1464678493402,  
    "status": 400,  
    "error": "Bad Request",  
    "exception": "org.springframework.web.bind.  
MissingServletRequestParameterException",  
    "message": "Required String parameter 'base' is not present",  
    "path": "/calculation/power/"  
}
```

@PathVariable

@PathVariable 可以帮助你创建动态的 URI。@PathVariable 注解允许你将 Java 参数映射到一个路径参数。它与 @RequestMapping 配合工作，其中后者在 URI 中创建占位符，然后要么作为 PathVariable，要么作为方法的参数使用相同的占位符名称，正如可以在 CalculationController 类方法 sqrt() 中看到的。在这里，值占位符是在 @RequestMapping 内创建的，并且相同的值被赋予 @PathVariable 的值。

sqrt() 方法提取 URI 中的参数来代替请求的参数。例如，http://localhost:8080/calculation/sqrt/144。在这里，值 144 作为路径参数传递，而此 URL 应该返回 144 的算术平方根，也就是 12。

为了使用现成的基本检查，我们使用正则表达式 "^-?+\\d+\\.?+\\d*\$" 来只允许有效的数字作为参数。如果传递了非数值，那么每个方法都在 JSON 的输出键中添加一条错误消息。

CalculationController 也使用正则表达式，在 path 变量 (path 参数) 中的.+允许 /path/{variable:.+} 的数值中带小数点(.)。Spring 将忽略最后一个点号后面的任何东西。Spring 的默认行为把它当作文件扩展名。



还有其他替代办法，如在末尾添加一个正斜杠(/path/{variable}/) 或通过把 useRegisteredSuffixPatternMatch 设置为 true，使用 PathMatchConfigurer(在 Spring 4.0.1 及更高版本中可用) 重写 WebMvcConfigurerAdapter 的 configurePathMatch() 方法。

```
package com.packtpub.mmj.restsample.resources;

package com.packtpub.mmj.restsample.resources;

import com.packtpub.mmj.restsample.model.Calculation;
import java.util.ArrayList;
import java.util.List;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import static org.springframework.web.bind.annotation.RequestMethod.GET;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/calculation")
public class CalculationController {

    private static final String PATTERN = "^-?+\\d+\\.?+\\d*\$";

    @RequestMapping("/power")
    public Calculation pow(@RequestParam(value = "base") String b, @
    RequestParam(value = "exponent") String e) {
        List<String> input = new ArrayList();
        input.add(b);
        input.add(e);
    }
}
```

```

List<String> output = new ArrayList();
String powValue = "";
if (b != null && e != null && b.matches(PATTERN) && e.matches(PATTERN)) {
    powValue = String.valueOf(Math.pow(Double.valueOf(b), Double.
valueOf(e)));
} else {
    powValue = "Base or/and Exponent is/are not set to numeric value.";
}
output.add(powValue);
return new Calculation(input, output, "power");
}

@RequestMapping(value = "/sqrt/{value:.+}", method = GET)
public Calculation sqrt(@PathVariable(value = "value") String aValue)
{
    List<String> input = new ArrayList();
    input.add(aValue);
    List<String> output = new ArrayList();
    String sqrtValue = "";
    if (aValue != null && aValue.matches(PATTERN)) {
        sqrtValue = String.valueOf(Math.sqrt(Double.valueOf(aValue)));
    } else {
        sqrtValue = "Input value is not set to numeric value.";
    }
    output.add(sqrtValue);
    return new Calculation(input, output, "sqrt");
}
}

```

在这里，我们只使用 `URI/calculation/power` 和 `/calculation/sqrt` 来公开 `Calculation` 资源的 `power` 和 `sqrt` 函数。



在这里，我们使用 `sqrt` 和 `power` 作为我们 `URI` 的一部分，仅用于演示目的。理想情况下，这些应该已经被作为请求参数“`function`”的值来传递，或基于端点设计形成的类似东西。

一个有趣的事情是，由于 Spring 的 HTTP 消息转换器的支持，Calculation 对象被自动转换为 JSON，不需要手动执行此转换。如果 Jackson 2 位于类路径中，那么 Spring 的 MappingJackson2HttpMessageConverter 就会把 Calculation 对象转换为 JSON。

制作一个示例 REST 可执行应用程序

创建一个类 RestSampleApp 和 SpringApplication 注解。main() 方法使用 Spring Boot 的 SpringApplication.run() 方法来启动应用程序。

我们将使用 @SpringBootApplication 来注解 RestSampleApp 类，它隐式添加所有下列标记：

- @Configuration 注解把这个类标记为应用程序上下文定义的 Bean 源。
- @EnableAutoConfiguration 注解指示，Spring Boot 开始基于 classpath 设置，其他 Bean 类和各种属性设置来添加 bean。
- 如果 Spring Boot 在 classpath 上发现 springwebmvc，@EnableWebMvc 注解就被添加。它将此应用程序视为一个 web 应用程序，并激活设置 DispatcherServlet 等关键行为。
- @ComponentScan 注解告诉 Spring 在给定包中寻找其他的组件、配置和服务：

```
package com.packtpub.mmj.restsample;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.
SpringBootApplication;

@SpringBootApplication
public class RestSampleApp {

    public static void main(String[] args) {
        SpringApplication.run(RestSampleApp.class, args);
    }
}
```

这个 web 应用是 100% 纯 Java 的，你不必使用 XML 来处理任何管道或基础设施的配置，相反，它使用 Java 注解，Spring Boot 甚至使之变得更简单。因此，除了用于 Maven 的 pom.xml 外，没有一行 XML，甚至没有 web.xml 文件。

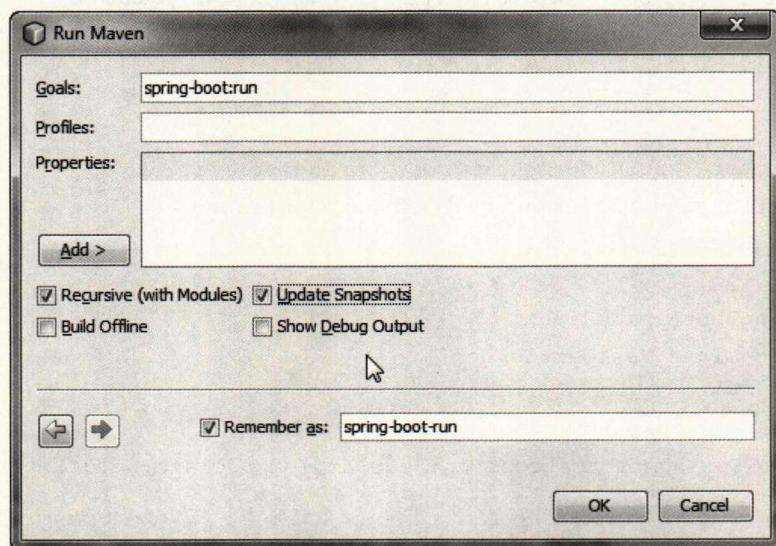
设置应用程序构建

直到现在，我们使用的 `pom.xml` 足以执行示例 REST 服务。这个服务的代码将打包进一个 JAR 文件里。要使这个 JAR 成为可执行文件，我们需要选择以下选项。

运行 Maven 工具

在这里，我们使用 Maven 工具来执行生成的 JAR，步骤如下：

1. 用鼠标右键单击 `pom.xml`。
2. 从弹出的快捷菜单中选择 **run-maven | Goals…**，它将打开对话框。在 **Goals** 字段中键入 `spring-boot:run`。我们已经在代码中使用了 Spring Boot 的正式发行版本。然而，如果你正在使用快照版本，可以选中 **Update Snapshots** 复选框。若要在将来使用它，请在 **Remember as** 字段中键入 `spring-boot-run`。
3. 下一次，可以直接单击 **run-maven | Goals | spring-boot-run** 执行该项目。



运行Maven对话框

4. 单击 **OK** 按钮执行该项目。

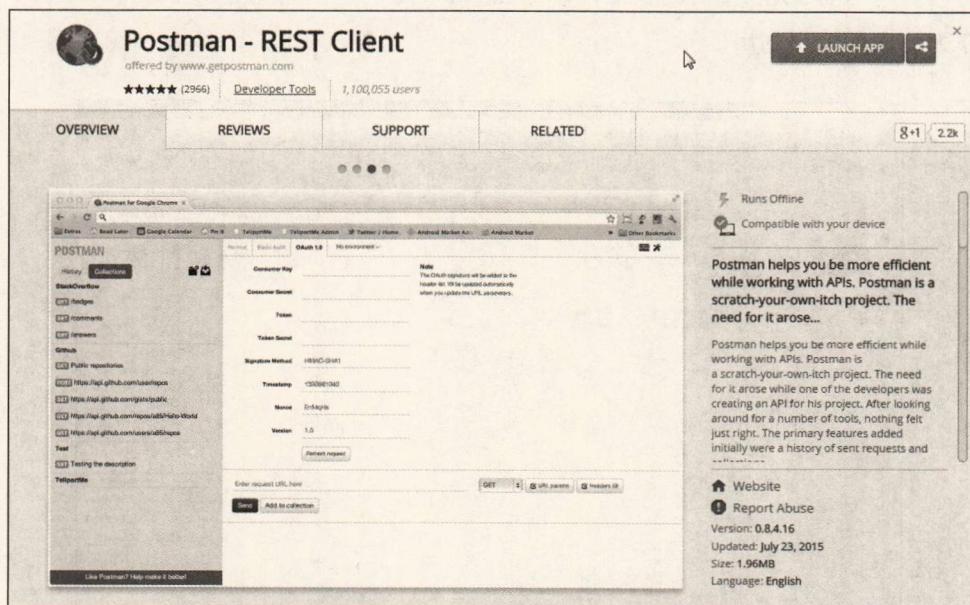
用 Java 命令执行

要生成 JAR, 请执行 mvn clean 来打包 Maven 目标。它在 target 目录中创建 JAR 文件, 然后可以使用如下命令执行 JAR:

```
java -jar target/restsample-1.0-SNAPSHOT.jar
```

使用 Postman Chrome 扩展测试 REST API

本书使用 Chrome 的 Postman-REST 客户端扩展来测试我们的 REST 服务。我们可以使用 <https://chrome.google.com/webstore/detail/postman-rest-client/fdmmgilgnpjigdojojojppooidkmcomcm> 下载的 0.8.4.16 版本。此扩展不再是可搜索的, 但可以使用给定的链接将其添加到 Chrome 中。此外可以使用 Postman Chrome 应用程序或任何其他 REST 客户端来测试示例 REST 应用程序。



Postman-REST 客户端 Chrome 扩展

一旦安装了 Postman-REST 客户端, 就可以测试第一个 REST 资源。我们从开始菜单或快捷方式启动 Postman-REST 客户端。



在默认情况下，嵌入式的 web 服务器在端口 8080 上启动。因此，我们需要使用 `http://localhost:8080/<resource>` URL 来访问示例 REST 应用程序。例如：`http://localhost:8080/calculation/sqrt/144`。

一旦 web 服务器启动，就可以在 Calculation REST URL 中键入 `sqrt` 和值 `144` 作为路径参数，可以在下图中看到它。在 Postman 扩展的 URL 输入字段（在这里输入请求 URL）中输入此 URL。默认情况下，请求方法是 GET。我们把此默认值作为请求方法使用，因为我们也编写了 REST 式的服务来为 GET 请求方法提供服务。

一旦准备好上面提到的作为输入的数据，就可以通过单击 **Send** 按钮提交该请求。可以在下图中看到，示例 REST 服务返回响应代码 200。你能在下图中找到 **Status** 标签以看到 **200 OK** 代码。成功的请求也返回 Calculation 资源的 JSON 数据，如屏幕截图中的 Pretty 页签所示。返回的 JSON 显示 `sqrt`，它是 `function` 键的值。它还显示 `144` 和 `12.0`，它们分别作为输入和输出列出。

The screenshot shows the Postman extension running in a browser window. The URL bar contains `http://localhost:8080/calculation/sqrt/144`. The main interface shows a **Normal** tab selected, with `GET` as the method. Below the URL, there are buttons for **Send**, **Preview**, and **Add to collection**. The **Status** section indicates **200 OK** and **TIME 62 ms**. The **Body** section has tabs for **Pretty**, **Raw**, **Preview**, **JSON**, and **XML**. The **Pretty** tab is selected, displaying the following JSON:

```
1 {
2     "function": "sqrt",
3     "input": [
4         "144"
5     ],
6     "output": [
7         "12.0"
8     ]
9 }
```

用Postman测试Calculation (sqrt)资源

同样，我们也可以测试示例 REST 服务的 power 函数。在 Postman 扩展中输入以下数据：

- **URL:** `http://localhost:8080/calculation/power?base=2&exponent=4`
- **Request method:** GET

在这里，我们分别传入值 2 和 4 作为底数和指数的请求参数，它将返回下面的屏幕截图中所示的 200 的响应状态和以下 JSON。

返回的 JSON：

```
{  
    "function": "power",  
    "input": [  
        "2",  
        "4"  
    ],  
    "output": [  
        "16.0"  
    ]  
}
```

The screenshot shows the Postman extension window. In the top bar, the URL is set to `http://localhost:8080/calculation/power?base=2&exponent=4` and the method is set to GET. Below the URL, there are input fields for 'base' (value 2) and 'exponent' (value 4). The 'Send' button is visible at the bottom left. The response section shows a status of 200 OK and a time of 47 ms. The 'Body' tab is selected, displaying the JSON response:

```
1 {  
2     "function": "power",  
3     "input": [  
4         "2",  
5         "4"  
6     ],  
7     "output": [  
8         "16.0"  
9     ]  
10 }
```

使用Postman测试Calculation (power) 资源

更多的正向测试场景

在下面的表中，所有 URL 都以 `http://localhost:8080` 开头。

URL	输出 JSON
<code>/calculation/sqrt/12344.234</code>	<pre>{ "function": "sqrt", "input": ["12344.234"], "output": ["111.1046083652699"] }</pre>
<code>/calculation/sqrt/-9344.34</code> Math.sqrt 函数的特别场景： • 如果参数是 NaN 或小于零，则结果是 NaN	<pre>{ "function": "sqrt", "input": ["-9344.34"], "output": ["NaN"] }</pre>
<code>/calculation/ power?base=2.09&exponent=4.5</code>	<pre>{ "function": "power", "input": ["2.09", "4.5"], "output": ["27.58406626826615"] }</pre>

续表

URL	输出 JSON
/calculation/power?base=-92.9&exponent=-4	{ "function": "power", "input": ["-92.9", "-4"], "output": ["1.3425706351762353E-8"] }

反向的测试场景

同样，可以还执行一些反向的场景，如下表所示。此表中的所有 URL 都以 `http://localhost:8080` 开头。

URL	输出 JSON
/calculation/power?base=2a&exponent=4	{ "function": "power", "input": ["2a", "4"], "output": ["Base or/and Exponent is/are not set to numeric value."] } //底数或指数未设置为数值

续表

URL	输出 JSON
/calculation/power?base=2&exponent=4b	{ "function": "power", "input": ["2", "4b"], "output": ["Base or/and Exponent is/are not set to numeric value."] } // 底数或指数未设置为数值
/calculation/power?base=2.0a&exponent=a4	{ "function": "power", "input": ["2.0a", "a4"], "output": ["Base or/and Exponent is/are not set to numeric value."] } // 底数或指数未设置为数值
/calculation/sqrt/144a	{ "function": "sqrt", "input": ["144a"], "output": ["Input value is not set to numeric value."] } // 输入值未设置为数值

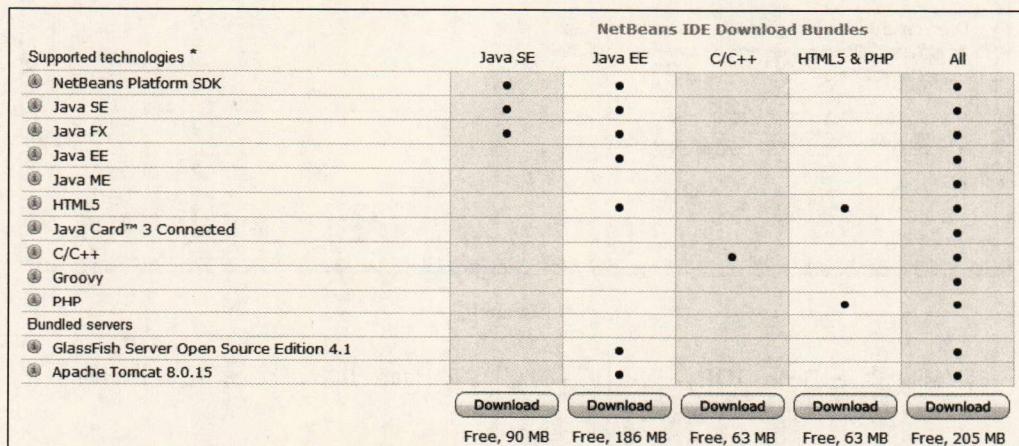
续表

URL	输出 JSON
/calculation/sqrt/144.33\$	{ "function": "sqrt", "input": ["144.33\$"], "output": ["Input value is not set to numeric value."] } //输入值未设置为数值

NetBeans IDE 安装和设置

NetBeans IDE 是免费并开放源码的，还有一个大的用户社区。可以从它的官方网站 <https://netbeans.org/downloads/> 下载 NetBeans IDE。

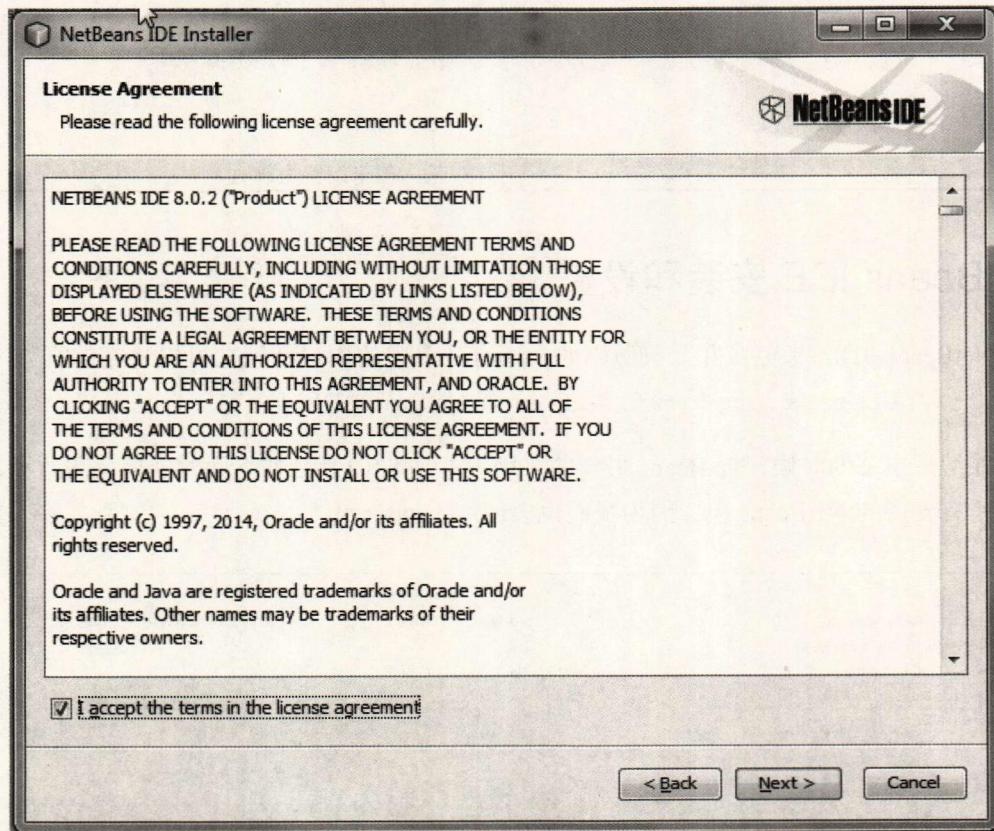
在编写本书的时候，NetBeans 最新的可用版本是 8.0.2 版。如下面的屏幕截图所示，请下载所有支持的 NetBeans 包，因为我们也会使用 Javascript。



NetBeans包

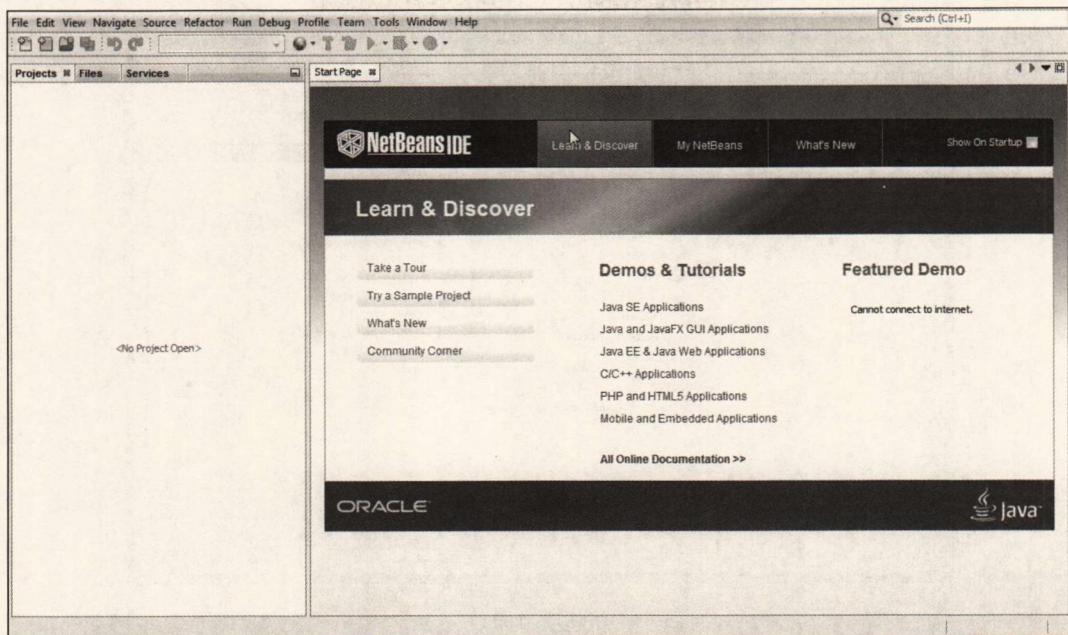
下载安装文件后，执行安装程序文件。接受使用协议，如下面的屏幕截图所示，并按照剩余步骤来安装 NetBeans IDE。Glassfish 服务器和 Apache Tomcat 是可选的。

 安装和运行所有的 NetBeans 包需要 JDK 7 或更高版本的 JDK。可以从 <http://www.oracle.com/technetwork/java/javase/downloads/index.html> 下载一个独立的 JDK 8。



NetBeans包

一旦安装了 NetBeans IDE，就可以启动它。NetBeans IDE 看起来应如下图所示。



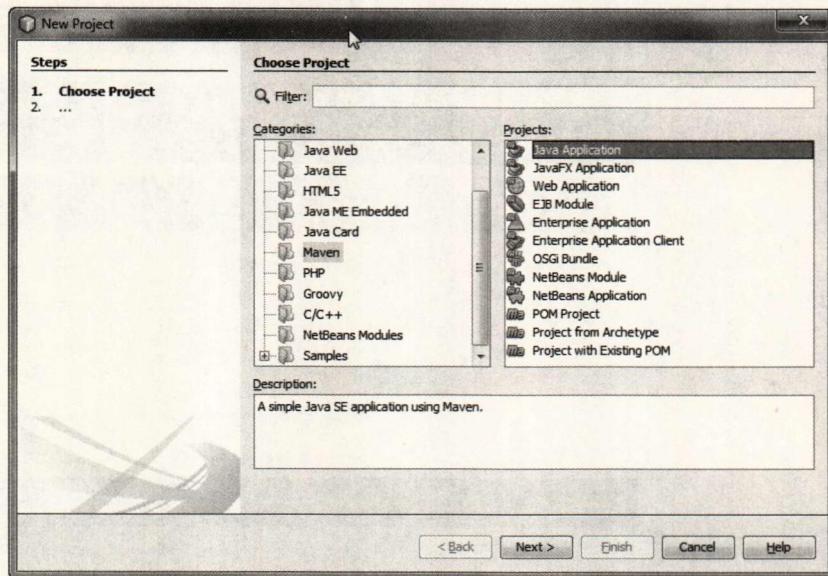
NetBeans 起始页

Maven 和 Gradle 两者都是 Java 构建工具，它们把依赖库添加到项目、编译你的代码、设置属性、建立归档文件，或执行多个相关的操作。Spring Boot 或 Spring Cloud 都同时支持 Maven 和 Gradle 构建工具。然而，在本书中我们将使用 Maven 构建工具。如果你喜欢，请随意使用 Gradle。

Maven 在 NetBeans IDE 中已经可用。现在，我们可以启动一个新的 Maven 项目来构建首个 REST 应用程序。

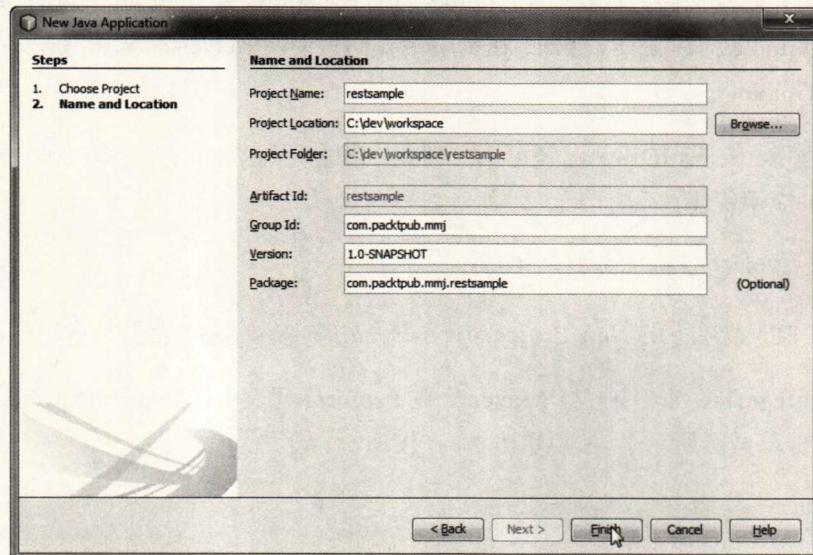
创建一个新的空 Maven 项目的步骤：

1. 单击 **File** 菜单下的 **New Project** (*Ctrl + Shift + N*)，它将会打开新建项目向导。
2. 在 **Categories** 列表中选择 **Maven**，在 **Projects** 列表中选择 **Java Application** (如下面的屏幕截图所示)，然后单击 **Next** 按钮。



新建项目向导

3. 现在，输入项目名称 `restsample`。此外，输入其他属性，如下面的屏幕截图所示。当所有必填字段都输入完成后，单击 **Finish** 按钮。



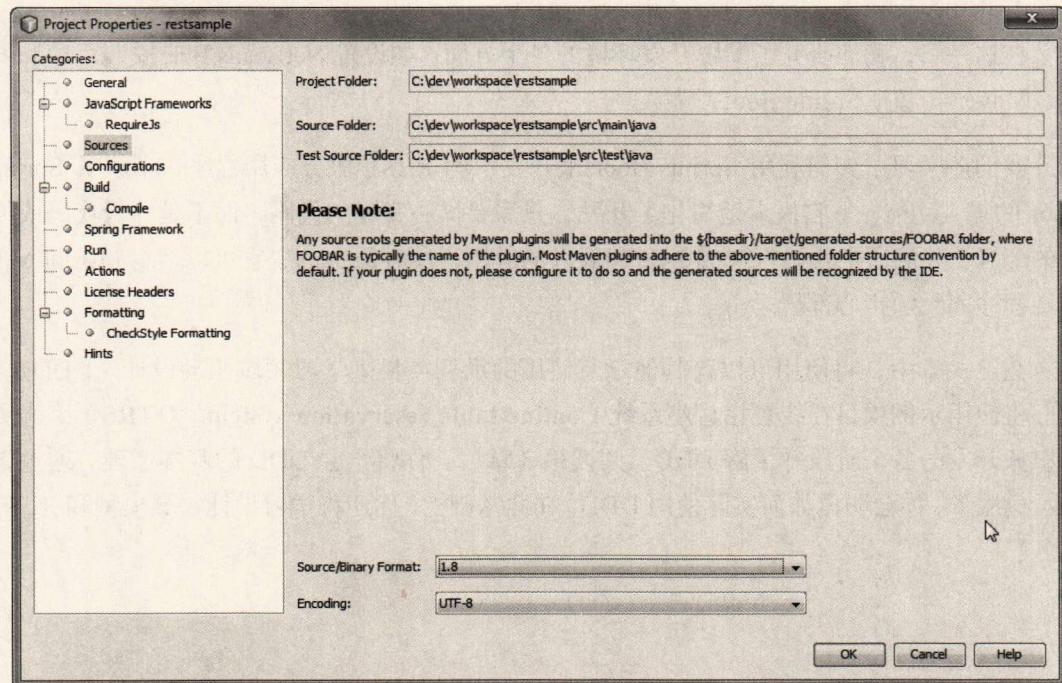
NetBeans Maven项目属性



Aggelos Karalias 已为 NetBeans IDE 开发了有用的插件，它为 Spring Boot 配置属性提供自动完成功能支持，可在 <https://github.com/keevosh/nbspringboot-configuration-support> 获取。可以从它在 <http://keevosh.github.io/nbspringboot-configuration-support/> 的项目主页下载它。还可以使用 Pivotal 的 Spring Tool Suite IDE (<https://spring.io/tools>) 来取代 NetBeans IDE。它是一个自定义的基于 Eclipse 的一体化版本，使得应用程序的开发变得容易。

完成上述所有步骤后，NetBeans 会显示新创建的 Maven 项目。你将使用此项目用于创建使用 Spring Boot 的示例 REST 应用程序。

若要使用 Java 8 作为源，请把 **Source/Binary Format** 设置为 **1.8**，如下面的屏幕截图所示。



NetBeans Maven 项目属性

参考资料

- **Spring Boot:** <http://projects.spring.io/spring-boot/>
- 下载 NetBeans: <https://netbeans.org/downloads>
- **Representational State Transfer (REST):** Roy Thomas Fielding 博士学位论文《Architectural Styles and the Design of Network-based Software Architectures》的第 5 章——<https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
- **REST:** https://en.wikipedia.org/wiki/Representational_state_transfer
- **Maven:** <https://www.apache.org/>
- **Gradle:** <http://gradle.org/>

小结

在这一章，我们研究了配置开发环境的各个方面，如设置 NetBeans IDE 安装程序和安装、Maven 配置、Spring Boot 配置。

我们还学习了如何利用 Spring Boot 来开发示例 REST 服务应用程序，了解了 Spring Boot 的强大功能——它极大地简化了开发，只需要操心实际的代码，而不是样板代码或你写的配置。我们还把代码打包成 JAR 和嵌入的应用程序容器 Jetty，它可以运行和访问 web 应用程序而无须担心部署。

在下一章中，将使用可以跨其他章节使用的示例项目来学习领域驱动设计（DDD）。我们将使用示例项目在线餐馆订座系统（**online table reservation system, OTRS**）来遍历微服务开发的各个阶段并了解 DDD。阅读第 3 章后，你将学会 DDD 的基本原理。通过设计示例服务，你会明白如何实际使用 DDD。在此基础上，你还将学习设计领域模型和 REST 服务。

3

领域驱动设计

本章通过引用一个示例项目为其余的章节定基调。从这里开始，将用这个示例项目来解释不同的微服务概念。本章使用此示例项目驱动不同的功能组合和领域服务或应用程序来解释领域驱动设计 (**domain driven design, DDD**)。它将帮助你了解 DDD 及其实际用法的基础知识。你还将使用 REST 服务来学习设计领域模型的概念。

本章包含以下主题：

- DDD 的基本要素
- 如何使用 DDD 设计应用程序
- 领域模型
- 一个基于 DDD 的领域模型设计示例

对于成功的产品或服务，良好的软件设计与它提供的功能同样关键，它们对产品取得成功有同等的重要性。例如，亚马逊提供了购物平台，但其架构设计使得它不同于其他类似的网站，并为它的成功做出了贡献。这表明软件或架构的设计对产品/服务的成功是多么重要。DDD 是软件设计实践活动之一，我们将使用各种理论和实际例子来研究它。

DDD 是一种关键的设计实践，它有助于设计你正在开发的产品的微服务。因此，在深入研究微服务的开发之前，我们将首先探讨 DDD。DDD 使用多层架构作为其组成部分之一，学习本章之后，你将理解 DDD 对于微服务开发的重要性。

领域驱动设计基本原理

企业或云应用程序是用来解决业务问题和其他现实世界的问题的。若没有所在领域的知识，就不能解决这些问题。例如，如果你不懂股票交易和其运作的原理，你就不能为金融系统，如网上股票交易提供一个软件解决方案。因此，拥有领域知识是解决问题的一个必备条件。现在，如果你想要提供一个使用软件或应用程序的解决方案，那么你需要领域知识的帮助来设计它。当我们把领域和软件设计相结合时，就产生了一种称为 DDD 的软件设计方法。

当我们开发软件来实现提供某个领域的功能的现实世界场景时，我们就创建了该领域的一个模型。模型是该领域的一种抽象或蓝图。



Eric Evans 在其于 2004 年出版的《领域驱动设计：处理位于软件核心的复杂性》（*Domain-Driven Design: Tackling Complexity in the Heart of Software*）这本书中首创了 DDD 这个术语。

虽然设计这种模型不像火箭科学那么艰难，但也需要付出大量的努力，需要领域专家的提炼和输入。它是软件设计师、领域专家和开发人员的集体工作。他们对信息进行组织，将其分成较小的部分，从逻辑上对它们进行分组并创建模块。每个模块可以单独处理，也可以使用类似的方法划分。我们可以遵循这一过程，直到到达单元级别或我们不能将它进一步划分为止。一个复杂的项目可能会有多个这样的迭代，而同样一个简单的项目可能只有单个的迭代。

一旦定义好了模型，并将其良好地记录在文档中，就可以转到下一个阶段——代码设计。所以，在这里我们有一个软件设计——领域模型和代码设计——和领域模型的代码实现。领域模型提供了高层次的解决方案（软件/应用程序）的架构，而代码实现作为一种能工作的模型给了领域模型生命。

DDD 使设计和开发人员团结合作。它持续不断地开发软件，同时根据从开发人员收到的反馈保持设计与时俱进的能力。它解决了敏捷和瀑布方法带来的局限性之一，即让软件包括设计和代码都具备可维护性，并使应用程序保持最低限度的可行性。

设计驱动的开发从初始阶段就涉及一个开发人员，他参加在建模过程中软件设计师与领域专家讨论领域的所有会议。这为开发人员提供合适的平台来了解领域，并提供机会来分享领域模型实现的早期反馈。它将消除在软件开发后期出现的股东等待可交付结果时的瓶颈。

组成部分

这部分描述 DDD 所采用的普遍存在的语言，以及之所以需要它的原因，用于模型驱动设计的不同模式和多层架构的重要性。

普遍存在的语言

正如我们所看到的，设计模型是软件设计师、领域专家和开发人员集体的工作，因此，它需要使用共同的语言来交流。DDD 使得有必要使用共同的语言，并使用普遍存在的语言。领域模型在其关系图、说明、陈述、演讲和会议中使用普遍存在的语言。这将消除误解、曲解和他们之间的沟通鸿沟。

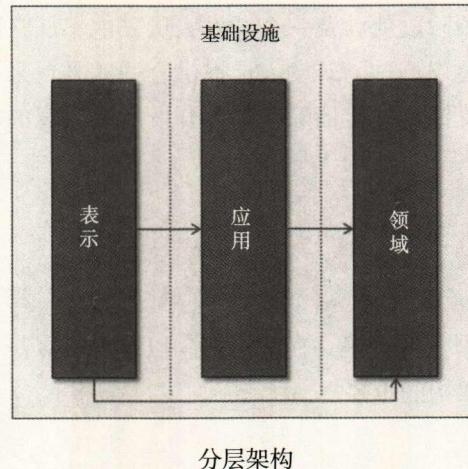
统一建模语言（**Unified Model Language, UML**）在创建模型时广泛使用，很流行。但这种语言也存在一些限制，例如，当你有一份绘制了成千上万个类的文件时，很难表示类之间的关系，也很难在理解它们的抽象性的同时理解它的含义。此外 UML 关系图也不会表示模型的概念以及对象都应该做些什么。

还有其他沟通领域模型的方法，如——文档、代码，等等。

多层架构

多层架构是 **DDD** 的一个通用解决方案。它包含四个层次：

1. 表示层或用户界面（User Interface, UI）。
2. 应用层。
3. 领域层。
4. 基础设施层。



在此可以看到只有领域层负责领域模型，而其他层都是与其他组件，例如用户界面、应用程序逻辑相关的。这种分层的架构非常重要，它使领域相关的代码与其他层分开。

在这种多层架构中，每层都包含其各自的代码，这有助于实现松耦合并避免把不同层的代码相混合，还有助于产品/服务的长期可维护性和增强功能的方便性。因为如果修改的目的只是针对各自的层，那么其中一层代码的修改不会对其他组件造成影响。使用多层架构，每一层都可以很容易地用另一个实现来切换。

表示层

此层表示 UI，并为交互和信息显示提供用户界面。这一层可能是一个 web 应用程序、移动应用或使用你的服务的第三方应用程序。

应用程序层

此层负责应用程序逻辑。它维护并协调产品/服务的整体工作流，它不包含业务逻辑或 UI，它可以保持应用程序对象的状态，如执行中的任务的状态。例如，你的产品 REST 服务将是应用层的一部分。

领域层

领域层是非常重要的一层，因为它包含领域信息和业务逻辑。它保持业务对象的状态，

它持久化业务对象的状态并将这些持久化的状态与基础设施层进行通信。

基础架构层

这一层向所有其他层提供支持，并负责其他层之间的通信。它包含由其他层使用的支持库，它还实现业务对象的持久化。

为了了解不同层的相互作用，让我们以一家餐馆的餐桌预订为例说明。最终用户使用 UI 做出预订一个餐桌的请求，用户界面将该请求传递到应用程序层。应用程序层从领域层提取领域对象，如餐馆、带有日期的餐桌，领域层从基础设施获取这些现有的持久化对象，并调用相应的方法进行预订并把它们持久化后返回给基础设施层。当领域对象被持久化后，应用层就给最终用户显示预订确认消息。

领域驱动设计的工件

在 DDD 中存在用于表示、创建和获取领域模型的不同工件。

实体

存在某些类别的可确定的对象，它们保持产品或服务的整体状态。这些对象不是(**NOT**)由其属性定义，而是由其身份和线程的连续性定义的。这些对象被称为实体。

这听起来很简单，但它承载着复杂性。你需要了解怎么定义实体。以餐馆订座系统为例，如果有 `restaurant` 类，包含餐馆名称、地址、电话号码以及建立数据等属性，可以取 `restaurant` 类的两个实例，它们不可使用餐馆名称来确定，因为可能还有其他餐馆具有相同名称。同样，如果通过任何其他单个属性去确定，也不会找到任何可以确定唯一的餐馆的属性。如果两个餐馆所有的属性值都相同，那么这两个餐馆是相同的，可以彼此互换。尽管如此，这些仍然是不同的实体，因为它们都具有不同的引用(内存地址)。

相反，让我们来看一个美国公民的类。每个公民都有他自己的社会安全号码。这个数字不但唯一，而且在公民的一生中不变，并保证连续性。这个公民对象将存在于内存中，会被序列化，并将会从内存中删除并且存储在数据库中。即使在该公民已去世后，它仍然存在。只要系统存在，它就将保留在系统中。公民社会安全号码保持不变，不论它的表示形式如何。

因此，在产品中创建实体是指创建标识符。所以，现在把标识符分配给前面示例中的任何一个餐馆，然后使用诸如餐馆名称、成立日期、街道等属性的组合或者添加标识符，如 `restaurant_id` 来标识它。基本规则是任何两个标识符都不能相同。因此，当我们为任何实体引入标识符时，我们需要确保这一点。

为对象创建一个唯一的标识符有不同的方式，如下所述：

- 使用表中的主键。
- 使用领域模块自动生成的 ID。领域程序生成标识符，并将它分配给在不同层间被持久化的对象。
- 少数现实生活中的对象本身就自带用户定义标识符。例如每个国家都有其自己的国际长途电话拨号国家/地区代码。
- 一个属性或属性的组合可用于创建一个标识符，正如前面对 `restaurant` 对象的解释。

实体对于领域模型是非常重要的，因此，应在建模过程的初始阶段就定义它们。当一个对象可以通过其标识符而不是它的属性来确定时，表示这些对象的类应该有一个简单的定义，并且应维护生命周期连续性和身份。对此类中具有相同属性值的对象进行标识势在必行。如果查询每个对象，一个既定的系统应返回唯一的结果。负责维护模型的设计师必须对同样的事物意味着什么加以定义。

值对象

实体具有一些特征，如身份、连续性的线程以及不能确定其身份的属性。**值对象（Value objects, VO）** 只是具有属性，而没有概念上的身份。最好的做法是把值对象保持为不可变的对象。如果可能，你甚至也应该保持实体对象不可变。

实体概念可能偏要把所有对象作为实体来保持，内存或数据库中的唯一可识别对象具有生命周期的持续性，但每个对象都必须有一个实例。现在，假设你正在把客户作为实体对象来创建。每个客户对象都表示餐馆的顾客，这个对象不能用于下其他顾客的订单。如果有数以百万计的客户使用此系统，这可能在内存中创建数以百万计的客户实体对象。在系统中，不止存在数以百万计的唯一可识别的对象，而且每个对象都被追踪。追踪和创建身份都是复杂的。创建和追踪这些对象需要高度可靠的系统，这个系统不但很复杂，而

且重度消耗资源。这可能会导致系统性能下降。因此，必须使用值对象而不是使用实体。原因将在接下来的几段中解释。

应用程序不总是需要有一个可识别的客户对象并加以追踪。有些情况下，只需要有领域元素的一些或全部属性。这些都是可以由应用程序使用值对象的情况，它使事情变得简单并提高了性能。

由于不存在身份，值对象可以被方便地创建和销毁。这简化了设计——它使得值对象在没有其他对象引用它们时可用于垃圾回收。

让我们讨论一下值对象的不变性。值对象应被设计和编码为不可变的。一旦创建了它们，它们绝不应在其生命周期中被修改。如果你需要此 VO 的一个不同的值，或其任何对象，那么只需创建一个新的值对象，但不要修改原始的值对象。在此，不变性从面向对象设计 (**object-oriented programming, OOP**) 继承了所有意义。当且仅当值对象是不可变的时，它才可以被共享和使用而不会影响其完整性。

常见的问题

1. 一个值对象可以包含另一个值对象吗？

是的，可以。

2. 一个值对象可以引用另一个值对象或实体吗？

是的，可以。

3. 可以使用不同的值对象或实体的属性来创建一个值对象吗？

是的，可以。

服务

创建领域模型时可能会遇到一些情况，其中的行为不能与任何对象明确关联。这些行为可以容纳在服务对象中。

在领域建模过程中，普遍存在的语言可帮助你确定具有不同属性和行为的不同对象、身份或者值对象。在创建领域模型的过程中，你可能会发现不同的不属于任何特定对象的

行为或方法。这些行为是重要的，所以不能忽视，也不能把它们添加到实体或值对象中。把行为添加到不属于它的对象中会破坏此对象。请记住，那些行为可能影响各种对象。使用面向对象编程可以将它们附加到一些对象上，这就是所谓的服务。

服务在技术框架中是常见的。这些也都用在 DDD 的领域层中。服务对象不具有任何内部状态，它的唯一目的是提供针对领域的行为。服务对象提供不能与特定实体或值对象相关联的行为，可能会为实体或值对象提供一个或多个相关的行为。这是在一个领域模型中显式定义的服务的做法。

在创建服务时，你需要勾选所有以下各点：

- 在实体和值对象上执行的服务对象的行为，但它不属于实体或值对象
- 服务对象的行为状态不会被保持，因此它们是无状态的
- 服务是领域模型的一部分

服务可能还存在于其他层中。保持领域层服务的隔离非常重要，它消除复杂性并保持设计解耦。

让我们看一个例子，一个餐馆老板想查看每月的餐桌预订的报告。在这种情况下，他将以管理员的身份登录，并提供所需的输入的字段，如时间段，然后单击 **Display Report**（显示报告）按钮。

应用程序层将请求传递到拥有的报告和模板对象的领域层，这种请求带有一些参数，如报表 ID 等。报告使用模板和从数据库或其他来源获取数据来创建。然后，应用程序层向业务层传递包括报告 ID 的所有参数。此时，需要从数据库或其他来源获取模板基于此 ID 生成报告。此操作不属于报表对象或模板对象。因此用一个服务对象来执行此操作，以便从数据库中提取所需的模板。

聚合

聚合域模式与对象的生命周期相关，并定义所有权和边界。

当你使用任何应用程序在网上预订你最喜欢的餐馆的餐桌时，你不需要操心处理预订的内部系统和过程，比如搜索可用的餐馆，然后搜索给定的日期和时间可用的餐桌，等等。因此，可以说一个预订应用程序是其他对象和作品的聚合，它充当一个餐馆订座系统的所

有其他对象的根。

这个根应该是将对象集合结合在一起的实体，它也称为聚合根。这个根对象不把对任何内部对象的引用传递到外部世界，并保护对内部对象执行的更改。

我们需要明白为什么需要聚合。领域模型可以包含大量的领域对象，应用程序的功能和规模越大，其设计越复杂，它就会存在越多的对象。这些对象之间存在关系，一些可能有多对多的关系，另一些可能有一对多的关系，而其他的可能有一对一的关系。这些关系是由代码或数据库中的模型实现来实施的，以确保这些对象之间的关系保持完整。关系不只是单向的，它们也可以是双向的。这也可能增加复杂性。

设计师的任务是简化这些模型中的关系。一些关系可能存在于一个现实的领域中，但可能在领域模型中不需要。设计师需要确保领域模型中不存在这种关系。同样，这些约束也可以减少多样性。在许多对象都满足某个关系的情况下，可以用一个约束来做这项工作。双向关系也可能转化为单向关系。

无论对输入进行多少简化，你仍然会面对模型中的关系。这些关系需要有维护它们的代码。当一个对象被删除时，代码应该从其他地方删除对此对象的所有引用。例如，从一个表中删除一行记录时，需要对它以外键形式引用的所有地方都加以处理，以保持数据的一致性并维护其完整性。在数据发生更改时，不变量（规则）也需要被实施和维护。

关系、约束和不变量带来的复杂性需要在代码中进行有效处理。我们利用由称为根的单一实体表示的聚合来解决这个问题，根与数据更改时用来保持一致性的一组对象相关联。

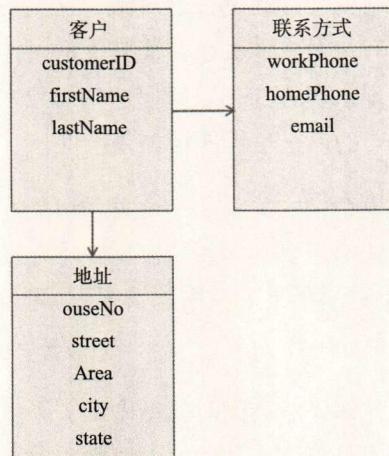
根是唯一可以从外部访问的对象，所以此根元素充当把内部对象与外部世界分离的边界的大门。根可以引用一个或更多的内部对象，并且这些对象内可以引用其他内部对象，而这些内部对象与根之间可能有关系，也可能没有关系。然而，外部对象也可以引用根，但不能引用任何内部对象。

聚合可确保数据的完整性并实施不变量。外部对象不能更改内部对象，它们只能改变根。然而，它们可以使用根，通过调用公开的操作，在此对象内部做出更改。根应该把所需的内部对象的值传递给外部对象。

如果一个聚合对象存储在数据库中，那么查询应该只返回此聚合对象。当对象内部链接到聚合根时，应该用遍历来返回它。这些内部的对象也可能会引用其他聚合。

聚合根实体保存其全局身份并保存其各实体内的本地身份。

在餐馆订座系统中，聚合的一个简单的例子就是客户。客户可以接触到外部对象，并且其根对象包含其内部对象的地址和联系信息。当发出请求时，就可以把诸如地址等内部对象的值对象传递给外部对象。



客户作为一个聚合

存储库

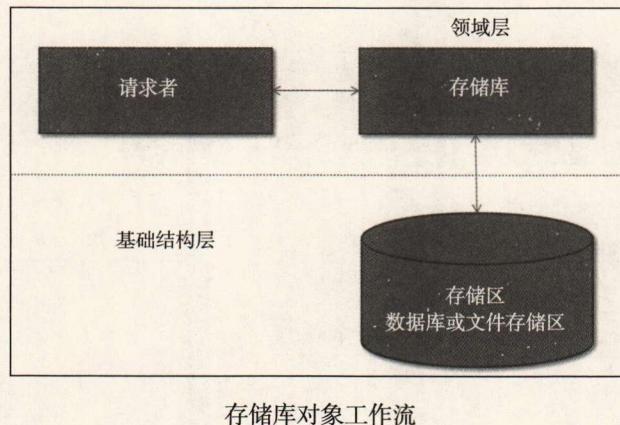
在一个领域模型中，在某一给定的时间，可能存在多个领域对象。每个对象都可能具有从创建到去除或持久的生命周期。每当任何领域操作需要一个领域对象时，它都应该有效地获取所请求的对象的引用。如果不把所有可用的领域对象都保存在一个中心的对象中，并由这个中心对象保存所有对象的引用并负责返回请求的对象的引用，这个任务将变得非常困难。这个中心的对象称为存储库 (repository)。

存储库是与基础设施如数据库或文件系统进行交互的一个地方。存储库对象是领域模型中与诸如数据库、外部来源等存储区(storage)，以检索持久化的对象进行交互的一部分。当存储库接收到对某个对象的引用请求时，它将返回现有对象的引用。如果存储库中不存在请求的对象，那么它就从存储区中获取对象。例如，如果你需要某个客户，你将查询存储库对象，请求它提供 ID 为 31 的客户。如果这个客户对象已经在存储库中，存储库就将提供请求的客户对象，如果不是这样，就将查询诸如数据库之类的持久化存储，把它取出

并提供它的引用。

使用存储库的主要优点是具有用来获取对象的一致方法，其中请求者不需要直接与诸如数据库的存储区进行交互。

存储库可以从各种存储区类型查询对象，这些存储区类型包括一个或多个数据库、文件系统或工厂资料库，等等。在这种情况下，存储库中可能也有指向不同的对象类型或类别的不同来源的策略。



如上图所示，存储库与基础设施层交互，此接口是领域层的一部分。请求者可能属于领域层或应用层。存储库帮助系统来管理领域对象的生命周期。

工厂

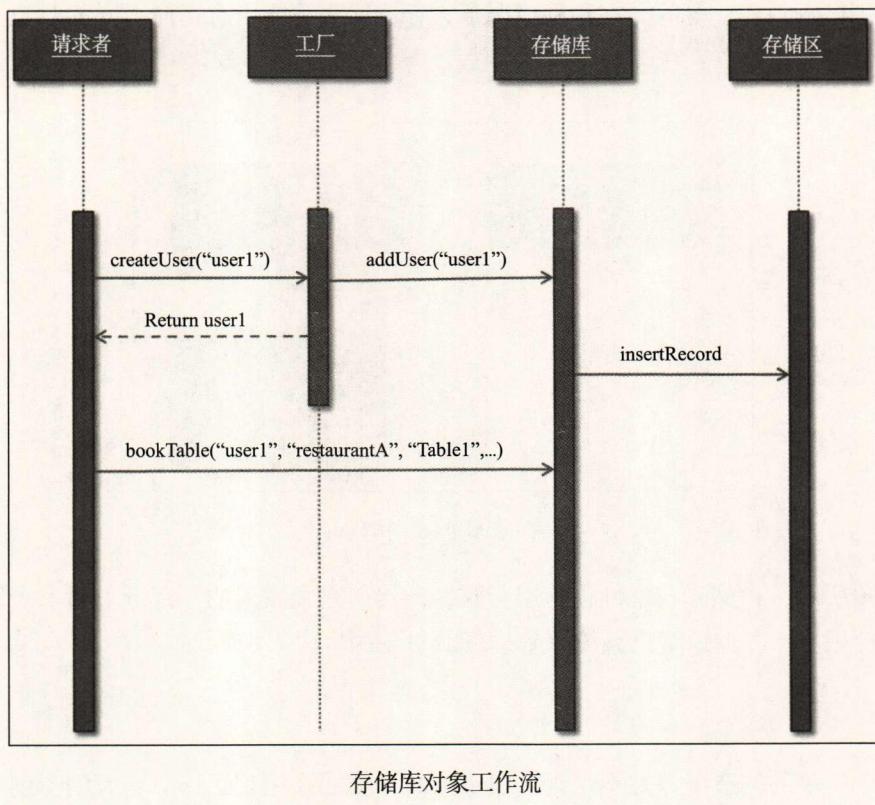
当一个简单的构造函数不足以创建对象时，就需要用到工厂。它有助于创建复杂的对象或一个涉及其他相关对象的创建的聚合。

工厂也是领域对象生命周期的一部分，因为它负责创建它们。工厂和存储库以某种方式彼此相关，因为两者都引用领域对象。工厂引用新创建的对象，而存储库从内存中或从外部存储器中返回已存在的对象。

让我们看看使用一个用户创建过程的应用程序的控制是如何流动的。假设一个用户注册用户名 user1。此用户创建过程首先与工厂进行交互，创建名称 user1，然后使用存储库将其缓存在领域中，存储库还将它存储在持久性存储区中。

当同一用户再次登录时，调用就转移到存储库中来获取一个引用。这使用存储区来加载此引用并将其传递给请求者。

然后请求者可能使用此 user1 对象在指定餐馆和指定的时间预订餐桌。这些值作为参数进行传递，并使用存储库在存储区中创建餐桌的一条预订记录。



工厂可以使用某种面向对象的编程模式，如工厂或抽象工厂模式之一来创建对象。

模块

模块是分离相关的业务对象的最好方法。这些都是最适合于领域对象的规模更大的大型项目。对于最终用户，把领域模型划分为模块并设置这些模块之间的关系是有道理的。一旦理解了模块和它们之间的关系，就会看到领域模型更宏观的画面，并且更容易进一步钻研和理解模型。

模块还有助于产生很高的内聚或保持低耦合的代码。普遍存在的语言可以用于命名这些模块。对于餐馆订座系统，我们可以拥有不同的模块，如用户管理、餐馆和餐桌、分析和报告，以及评论，等等。

战略设计和原则

企业模型通常都非常庞大和复杂。它可能分布在组织的不同部门之间，每个部门都可能有单独的领导团队，所以在一起工作和设计可能产生困难和协调问题。在这种情况下，维持完整的领域模型不是一项容易的任务。

在这种情况下，致力于一个统一的模型不能解决问题，大型企业模型需要被划分为不同的子模型。这些子模型包含预定义的准确关系和具有微小细节的合同。每个子模型都必须无一例外地保持既定的合同。

保持领域模型的完整性有各种可以遵循的原则，这些原则如下所示。

- 有界上下文
- 持续集成
- 上下文映射
 - 共享内核
 - 客户-供应商
 - 顺从者
 - 反腐层
 - 各自的方法
 - 开放主机服务
 - 精馏

有界上下文

当你有不同的子模型时，如果所有子模型都被结合在一起，代码是难以维护的。你需要有一个可以分配给单个团队的小型模型。你可能需要收集相关的元素，并将它们组合起来。上下文通过应用此组条件来保持并维护为其各自的子模型定义的领域术语的含义。

这些领域的术语定义创建上下文边界模型的范围。

有界上下文看起来非常类似于在上一节中了解到的模块。事实上，模块是有界的上下文的一部分，它定义了子模型产生并被开发的逻辑框架。而模块组织领域模型的元素，并可以在设计文档和代码中看见。

现在，作为一个设计师，你需要保持每个子模型的明确和一致性。以这种方式，可以在不影响其他子模型的情况下独立地重构每个模型。这使软件设计师可以在任何时点灵活地完善和改进它。

现在看看餐桌预订示例。当你开始设计系统时，你就会预见到客人将访问该应用程序，并将请求在所选的餐馆、日期和时间保留餐桌。然后在此基础上，还有把预订信息通知餐馆的后端系统。同样，餐馆会根据餐桌预订来更新他们的系统，因为餐桌可以也由餐馆自己预定。所以，当你更细致地查看这个系统时，可以看到两个领域模型：

- 在线餐馆订座系统
- 离线餐馆管理系统

每个领域模型都有其有界的上下文，需要确保它们之间的接口工作正常。

持续集成

当你进行开发时，代码散布在很多团队中，并使用各种技术，此代码可以被组织成不同的模块，并且各子模型都有适用的有界上下文。

这种开发可能会带来某种程度的重复代码、代码冲突或破坏有界上下文方面的复杂性。这种情况的发生不仅因为代码和领域模型的规模庞大，而且因为有其他因素的影响，如团队成员的变化，新成员或缺少有据可查的模型来命名其中的一小部分。

当使用 DDD 和敏捷方法来设计和开发系统时，在开始编码之前，领域模型不是完全设计好的，并且随着时间的推移，经历一段时间的不断改进和完善，领域模型及其元素都发生了进化。

因此，集成在持续进行，这是目前开发的关键原因之一，所以它发挥着非常重要的作用。在持续集成中，经常合并代码以避免造成任何破坏和领域模型的问题。合并的代码不

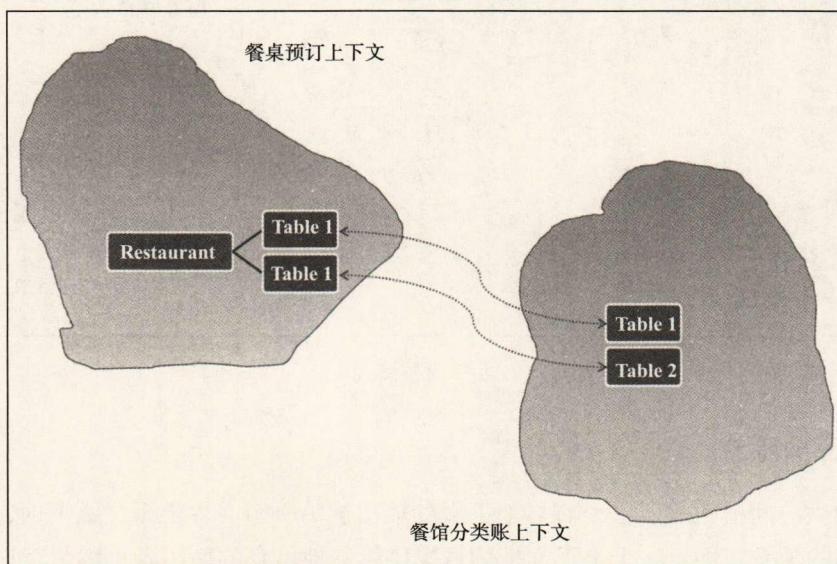
仅得到部署，而且还被定期测试。市场上有能够在预定的时间合并、构建和部署代码的各种持续集成工具。现在，各个组织更加注重持续集成的自动化。Hudson、TeamCity 和 Jenkins CI 是现在可用来进行持续集成的几个流行的工具。Hudson 和 Jenkins CI 是开放源代码工具，而 TeamCity 是一个专有的工具。

每个生成版本都具有附加的测试套件来验证模型的一致性和完整性。测试套件从物理的角度定义模型，而 UML 则在逻辑上定义模型。它告诉你有关任何错误或意外的结果，需要更改代码来改正它们，它还有助于及早发现领域模型中的错误和异常。

上下文映射

上下文映射帮助你理解一个大型企业应用程序的整体情况。它显示在企业模型中有多少有界上下文存在，以及它们如何相互关联。因此我们可以把任何解释了有界上下文和它们之间的关系的关系图或文档都称为上下文映射。

上下文映射能帮助所有团队成员，无论他们是在同一个团队还是在不同的团队，以各种部件（有界上下文或子模型）和关系的形式了解高层次企业模型。这使每个人都能清楚地知道某个人执行的任务，并允许他提出模型的完整性有关的任何关注点/问题。



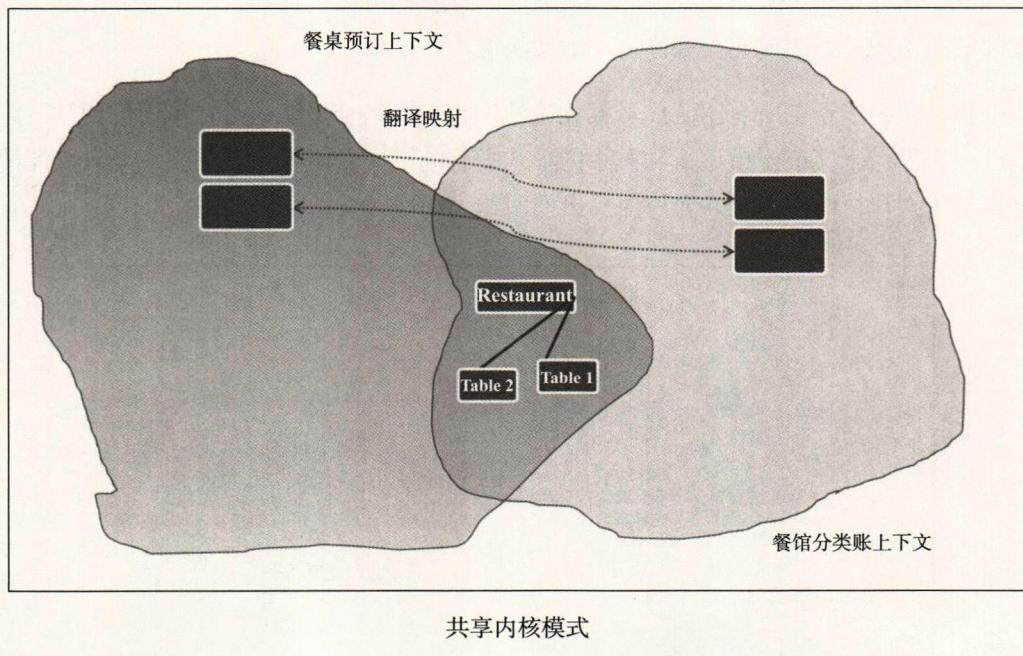
上下文映射示例

上下文映射的关系图是上下文映射的一个示例。在这里，**Table1** 和 **Table2** 二者都既出现在餐桌预订上下文中，又出现在餐馆分类账上下文中。有趣的事情是，**Table1** 和 **Table2** 在每个有界上下文中都有各自的概念。在这里，用普遍存在的语言把有界上下文命名为餐桌预订和餐馆分类账。

在下面的部分，我们将探索一些可以用来定义上下文映射中的不同上下文之间通信的模式。

共享内核模式

正如其名字所示，一个有界上下文与另一个有界上下文共享一个部分。正如你可以看到的，下面的 **Restaurant** 实体正在餐桌预订上下文和餐馆分类账上下文之间共享。



客户和供应商模式

客户和供应商模式表示一个有界上下文的输出被另一个有界上下文需要时，两个有界上下文之间的关系，即：一个上下文把信息提供给其他信息的使用者（称为客户）。

在现实的例子中，一个汽车经销商不能出售汽车，除非汽车制造商提供了这些汽车。因此，在此领域模型中，汽车制造商是供应商，而经销商是客户。这种关系建立了一种客户和供应商的关系，因为一个有界上下文（汽车制造商）的输出（汽车）是另一个有界上下文（经销商）所必需的。

在这里，客户和供应商团队应定期开会，建立一份合同，并形成正确的互相交流的协议。

顺从者模式

这种模式类似于客户和供应商模式，一方需要提供合同和信息，而另一方需要使用它。在这里，不同于有界上下文，实际参与的团队有上游下游的关系。

此外，上游团队因为缺乏动机而不对下游团队提供需要的信息。因此，下游团队可能需要计划和处理绝不可能获得的项目。要解决这种情况，如果供应商提供的信息不足，任一客户团队都可以开发他们自己的模型。如果供应商提供的信息是真正有价值或部分有价值的，那么客户就可以利用可用来消费供应商提供的信息的接口或翻译器来处理客户自己的模型。

反腐层

反腐层仍然是领域的一部分，当系统需要从外部系统或从他们自己的遗留系统获取数据时，就会用到它。在这里，反腐层是与外部系统进行交互，并在领域模型中使用外部系统数据，而不会影响领域模型完整性和独创性的一个层。

大多数情况下，可以使用服务作为反腐层，它可能使用具备适配器和翻译器的外观模式在内部模型内使用外部领域数据。因此，你的系统始终会使用服务来获取数据。服务层可以使用外观模式设计。这将确保它将与领域模型配合工作，以按照给定的格式提供所需的数据。然后，服务还可以使用适配器和翻译器模式，这将确保无论由外部来源发送的数据格式和层次结构是怎样的，都将使用适配器和翻译器提供所需的格式和层次结构的服务。

独立方法

当你拥有一个大型企业应用程序和领域，其中不同的领域没有共同的元素，并且它由能独立开展工作的大型子模型组成时，对于最终用户，这仍然作为单个应用程序工作。

在这种情况下，设计师可以创建单独的没有相互关系的模型，并在它们上面开发一个小应用程序。这些小应用程序合并在一起时成为一个单独的应用程序。

工作单位内部网应用程序是这类应用程序中的一种，它提供如人力资源相关、问题跟踪器、运输或公司内部社交网络的各种小应用程序，设计师可以使用独立方法的模式来开发它。

若要集成使用独立模型开发的应用程序，这将具有非常大的挑战性和复杂性。因此，在实现这种模式之前，你应该三思而行。

开放主机服务

当两个子模型彼此交互时，就会使用翻译层。这个翻译层用于把模型与外部系统集成。当你有一个使用该外部系统的子模型时，这种模式工作正常。当这个外部系统被用于很多子模型，以删除多余和重复的代码时，因为你要为每个子模型的外部系统都编写一个翻译层，则需要开放主机服务。

开放主机服务使用对所有子模型的包装提供外部系统的服务。

精馏

正如你所知，精馏是净化液体的过程。同样，在 DDD 中，精馏是过滤掉不必要的信息，并只保留有意义的信息的过程。它可以帮助你识别核心领域和业务领域的基本概念，过滤掉一般的概念，直到你得到代码域的概念为止。

开发人员和设计师对核心领域进行设计、开发和实施时应对细节加以最高的关注，因为它是整个系统成功的关键。

在我们的餐馆订座系统示例中，它不是一个大型或复杂领域的应用程序，它的核心领域不难识别。核心领域在这里的存在是为了共享实时准确的在餐馆里的空置餐桌，并允许用户用不麻烦的过程来预订它们。

示例领域服务

让我们基于餐馆订座系统创建一个示例领域服务。如本章中所述，有效领域层的重

要性是成功的产品或服务的关键。基于领域层开发项目有更好的可维护性、高内聚性和解耦性。它们在业务需求变化时能够提供高可扩展性，并对其他各层的设计有较少的影响。

领域驱动开发基于领域，因此不推荐你使用首先开发 UI，紧接着开发其余的层，最后开发持久层的自顶向下的方法，也不推荐首先开发类似数据库的持久层，然后开发其余的层，最后开发 UI 的自底向上的方法。

首先使用本书描述的模式开发出一个领域模型，将使所有团队成员对功能都有清晰的认识，便于软件设计师来建立一个灵活的、可维护的、一致的系统，可以帮助组织以较低的维护成本来开发世界一流的产品。

在这里，你将创建提供添加和检索餐馆功能的一个餐馆服务。基于具体实现，你可以添加其他功能，如基于菜品或评级查找餐馆。

从实体开始。在这里，餐馆是我们的实体。因为每个餐馆都是唯一的，并有一个标识符。可以使用一个接口或一组接口在我们的餐馆订座系统中实现实体。理想情况下，如果采用接口隔离原则，你会使用一组接口，而不是一个单独的接口。

 接口隔离原则（Interface Segregation Principle, ISP）：客户不应该被强迫去依赖于他们不使用的接口。

实体的实现

对于第一个接口，你可以有一个所有实体所需的抽象类或接口。例如，如果我们考虑 ID 和名称，对于所有实体，属性都将是常见的。因此，可以在你的领域层使用抽象类 Entity 作为实体的抽象：

```
public abstract class Entity<T> {
    T id;
    String name;
}
```

基于此，你也可以有另外一个继承 Entity 的抽象类：

```
public abstract class BaseEntity<T> extends Entity<T> {  
  
    private T id;  
    public BaseEntity(T id, String name) {  
        super.id = id;  
        super.name = name;  
  
    }  
    ... (getter/setter 和其他相关的代码)  
}
```

基于前面的抽象，我们可以创建用于餐馆管理的 Restaurant 实体。

现在，因为我们正在开发的是餐馆订座系统，Table 是领域模型中另一个重要的实体。所以，如果我们采用聚合模式，restaurant 会作为一个根来工作，而 Table 将是 Restaurant 内部的实体。因此，Table 实体将始终可以使用 Restaurant 实体来访问。

可以用下面的实现创建 Table 实体，并且可以根据需要添加属性。只为了示范，下面使用基本的属性：

```
public class Table extends BaseEntity<BigInteger> {  
  
    private int capacity;  
  
    public Table(String name, BigInteger id, int capacity) {  
        super(id, name);  
        this.capacity = capacity;  
    }  
  
    public void setCapacity(int capacity) {  
        this.capacity = capacity;  
    }  
  
    public int getCapacity() {  
        return capacity;  
    }
```