# 1. 第 1 章 欢迎来到 Cloud 和 Spring

***本章内容***

- 理解微服务以及企业为什么使用它们
- 使用 Spring, Spring Boot 和 Spring Cloud 构建微服务
- 学习基于微服务的应用为什么与云和微服务相关
- 构建微服务不仅仅是构建服务代码
- 了解基于云的开发
- 使用 Spring Boot 和 Spring Cloud 开发微服务

软件开发领域中永恒不变就是我们作为软件开发人员坐在无限混乱和变化的世界中。随着新技术和新方法的突然出现，这使我们重新评估我们如何为客户建立和交付解决方案。这样的例子，比如使用微服务构建应用的方式被许多企业快速采用。微服务是分布的、松散耦合的、单一职责的软件服务。

这本书向你介绍了微服务架构和为什么你应该考虑使用它们构建你的应用程序。我们来看看如何使用 java，Spring Boot 和 Spring Cloud 构建微服务。如果你是一个 java 开发者，Spring Boot 和 Spring Cloud 将为你提供一个将传统的、庞大的 Spring 应用迁移到可以部署到云的微服务应用的简单迁移路径。

## 1.1. 什么是微服务？

在微服务概念形成之前，大多数基于 web 的应用程序都是使用单一的体系结构风格构建的。在单体架构中，应用程序作为一个可部署的软件工件交付。所有的 UI（用户界面）、业务逻辑和数据库访问逻辑，被一起打包成一个单一的应用程序，并部署到应用服务器。

虽然应用程序可能是作为单个工作单元部署的，但大多数情况下将有多个开发团队在应用程序上工作。每一个开发团队都有他们自己的独立应用程序，并且他们经常用他们的功能部件来服务特定的客户。例如，当我在一家大型金融服务公司工作时，我们有一个内部定制的客户关系管理（CRM）应用程序，它涉及多个团队的协调，包括 UI、客户经理、数据仓库和共同基金团队。图 1.1 显示了该应用程序的基本结构。

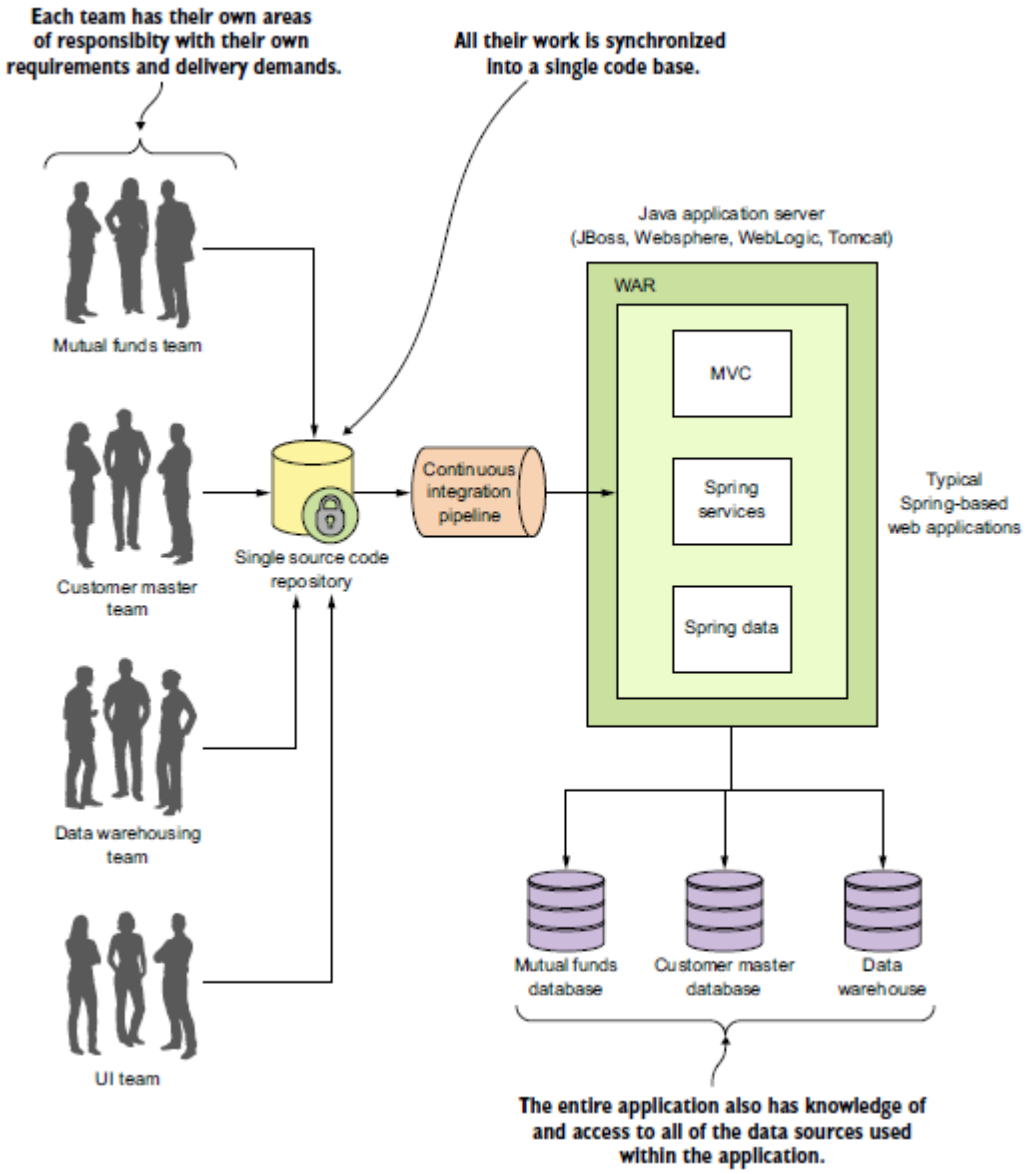


图 1.1 单体应用程序迫使多个开发团队人为地同步交付，因为它们的代码需要作为整个单元来构建、测试和部署。

*① Each team has their own areas of responsibity with their own requirements and*

*delivery demands.*

每个团队都有自己的工作要求和交货要求。

② *All their work is synchronized into a single code base.*

所有的资源都同步到一个单一的代码库。

③ *Mutual funds team*

共同基金团队

④ *Customer master team*

客户经理团队

⑤ *Data warehousing team*

数据仓库团队

⑥ *UI team*

UI 团队

⑦ *Single source code repository*

独立的源代码仓库

⑧ *Continuous integration pipeline*

持续集成管道

⑨ *Typical Spring-based web applications*

典型的基于 Spring 的 Web 应用程序

⑩ *The entire application also has knowledge of and access to all of the data*

sources used within the application.

整个应用程序了解和访问其内部使用的所有数据源。

这里的问题是，随着单体 CRM 应用规模和复杂性的增长，在应用程序上工作的各个团

队的交流和协调成本将无法控制。每当个别团队做出一点变动，整个应用程序必须被重新构建、重新测试和重新部署。

2014 年左右，微服务概念被引入到软件开发社区。它得到了许多试图在技术上和规模上挑战单体应用的大型组织的直接响应。微服务是一个小的，松散耦合的分布式服务。微服务允许你将一个大型的应用程序分解成易于管理和职责明确的组件。微服务通过将大型代码分解成小的、定义明确的块，帮助应对复杂性的传统问题。你需要接受一个关键概念：微服务可以将应用程序的功能分解和分拆，它们是完全独立的。如果我们把图 1.1 中看到的 CRM 应用分解为微服务，它可能如图 1.2 所示。

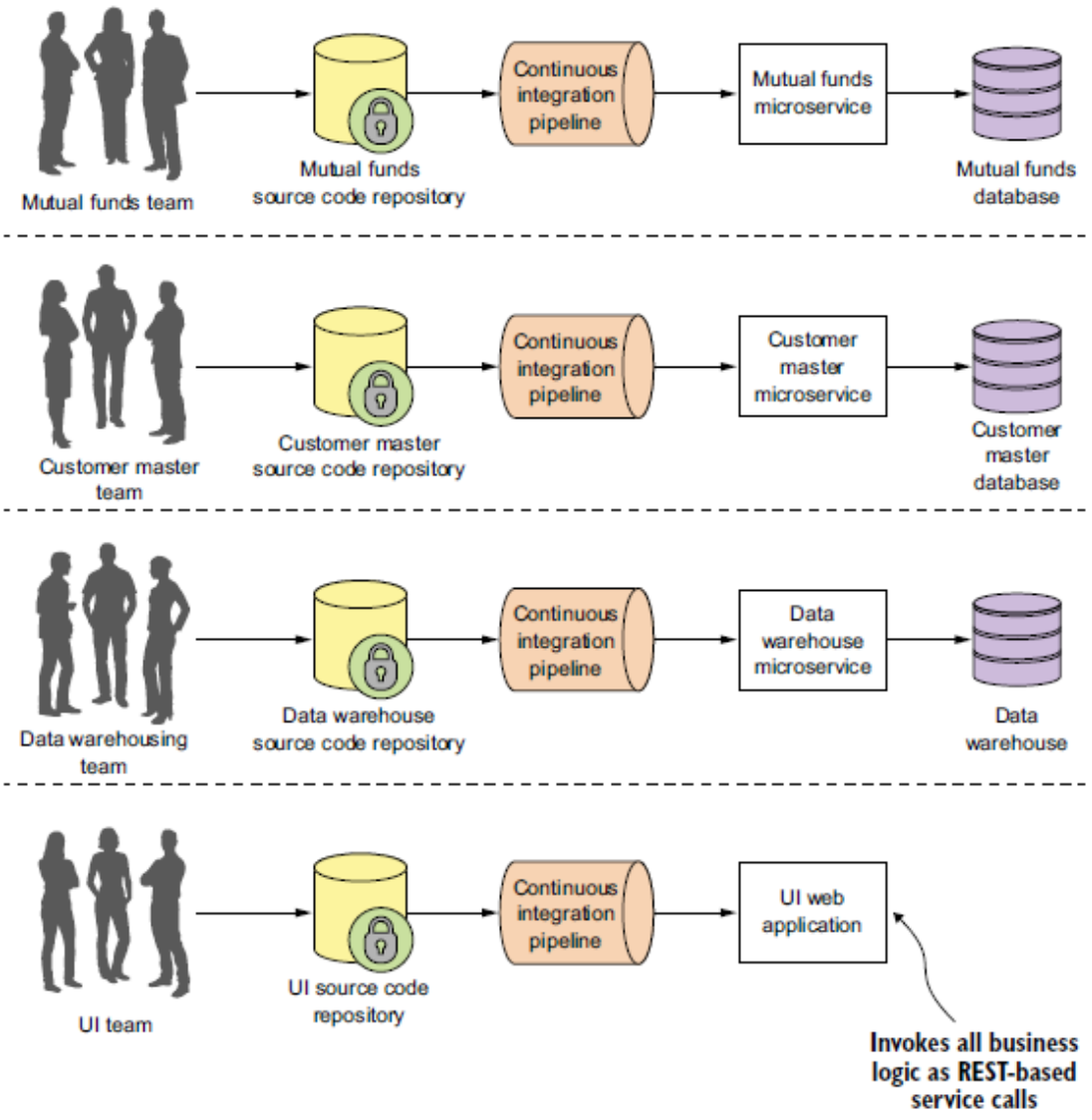

图 1.2 使用微服务架构，我们的 CRM 应用将被分解成一系列完全相互独立的微服务，让每个开发团队根据自己的要求管理和维护。

*① Invokes all business logic as REST-based service calls*

基于 REST 服务调用所有的业务逻辑

在图 1.2，你可以看到每个功能团队完全拥有他们的服务代码和服务基础设施。它们可以独立地构建、部署和测试，因为它们的代码、源代码版本控制仓库和基础设施（应用服务器和数据库）完全独立于应用程序的其他部分。

微服务架构具有以下特点：

- 应用逻辑被分解为小粒度的、责任界限明确的组件。
- 每个组件都有一个小的责任域，并且完全独立部署。微服务应该有一个业务领域的单一职责。同时，微服务应该跨多个应用程序可重用。
- 微服务通信基于几个基本原则（注意我说的是原则，没有标准）和采用轻量级的通信协议如 HTTP 和 JSON（JavaScript Object Notation）为服务消费者和服务提供者之间提供数据交换。
- 服务的底层实现与技术无关，因为应用程序总是与技术无关的协议通信（JSON 是最常见的）。这意味着使用微服务构建应程序，可以用多种语言和技术。
- 微服务具有小的，独立的，天然分布式的特点。允许企业拥有规模小、责任明确的开发团队。这些团队可能会朝着一个单一的目标，如发布一个应用程序，但是每个团队只负责与该服务相关工作的。

我经常和同事开玩笑：微服务是构建云应用的入门药物。你开始创建微服务，因为它们为你的开发团队提供高度的灵活性和自主性，但是你和你的团队很快会发现微服务体积小，天然独立的特点，使得它们更易于部署到云。一旦服务被部署在云上，体积小使得它们易于

启动大量相同服务的实例，使应用程序变得更具可扩展性，前瞻性和弹性。

## 1.2. 什么是 Spring，它和微服务有什么关联？

Spring 已经成为构建基于 java 应用程序事实上的开发框架。Spring 的核心是基于依赖注入的概念。在一个正常的 Java 应用里，应用被分解成各种各样的类。每个类经常被应用里其他的类明确的链接。链接就是一个类的构造函数中的代码直接调用。一旦代码被编译，这些链接点是不能改变的。

这在一个大型项目中是有问题的，因为这些外部链接是脆弱的，进行更改会导致其他代码的多个下游影响。依赖注入框架，如 Spring，让你通过在应用程序里使用约定（注释）定义外部对象之间的关系，更方便的管理大型的 java 项目，而不是在对象里硬编码对象之间的引用。在应用程序里不同的 java 类之间，Spring 作为中介管理它们的依赖。Spring 本质上是让你像一套扣合在一起的乐高积木一样，组装你的代码。

Spring 框架具有使人快速上手的优势，它迅速成为企业级应用 java 开发人员寻找替代使用 J2EE 技术栈构建应用的一种更轻量级的方式。J2EE 技术栈，虽然很强大，但是被许多人认为太臃肿了，有很多功能，并没有被应用开发团队使用。特别是，J2EE 应用强制要求你使用重量级的 java 应用服务器进行应用程序的部署。

Spring 框架令人惊奇的是，它的社区生态能够持续发展并坚持自身。Spring 开发团队很快发现许多开发团队正在摒弃单体应用程序，这些应用程序的展示、业务和数据访问逻辑被打包在一起，作为单个工件部署。相反，他们正在转移到高度分布式的模型中，这些服务能够构建为可以轻松部署到云上的小型分布式服务。作为回应，Spring 开发团队推出了两个项目：Spring Boot 和 Spring Cloud。

Spring Boot 是重新构想的 Spring 框架。然而它包含了 Spring 的核心特征，Spring

Boot 剥离了 Spring 里面许多的“企业级”特性，提供了一个基于 java，面向 REST（表述性状态转移）[1] 微服务的框架。使用简单的注释，一个 java 开发者就可以快速创建一个 REST 微服务，它可以被打包和部署而不需要外部的应用程序容器。

> ***注意：***我们将在第 2 章详细地讨论 REST，它的核心概念是你的服务应该使用 HTTP 谓词（GET, POST, PUT 和 DELETE）来表示服务的核心操作，并使用面向 Web 的轻量级数据序列化协议（如 JSON）来请求和接收来自服务的数据。

因为微服务已经成为构建基于云的应用程序的更常见的架构模式之一，Spring 社区为我们提供了 Spring Cloud。Spring Cloud 框架使得将微服务部署到私有云或公共云变得更简单。Spring Cloud 把几种流行的云管理微服务框架整合到共同的框架下，在代码里使用注释使得这些技术的使用和部署变得更容易。在本章后面，我将介绍 Spring Cloud 中的不同组件。

## 1.3. 你将从这本书里学到什么？

这本书介绍使用 Spring Boot 和 Spring Cloud 构建可以部署到你公司私有云或公共云（如 Amazon、谷歌或 Pivotal）中基于微服务应用程序。在本书，我们有亲身实践的例子：

- 什么是微服务和构建基于微服务应用的设计注意事项
- 什么时候不应该创建微服务应用
- 如何使用 Spring Boot 框架构建微服务
- 用来支持微服务应用，特别是基于云应用的核心业务模式
- 你如何使用 Spring Cloud 来实现这些业务模式
- 如何获取所学知识，并构建可用于将服务部署到私有的、内部管理的云或公有云的部署管道

[1] 我们在第 2 章介绍 REST，阅读 Roy Fielding 博士关于构建基于 REST 应用程序的博士论文是值得的 (http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm)。它仍然是目前能找到的关于 REST 的最好说明。

当读完本书的时候，你将学习到构建和部署基于 Spring Boot 微服务的相关知识。你也会了解到实施微服务需要到的关键设计决策。你将了解服务配置管理、服务发现、消息、日志和跟踪、与安全如何结合在一起，提供一个稳定的微服务环境。最后，你将会看到你的微服务可以部署在私有云或公共云。

## 1.4. 为什么这本书与你有关？

如果你已经阅读到第 1 章，我大胆的猜测：

- 你是一位 java 开发者。
- 你了解 Spring。
- 你有兴趣学习如何建立基于微服务应用。
- 你对如何使用微服务构建基于云的应用程序有兴趣。
- 你希望了解到构建微服务应用所使用的 Java 和 Spring 相关技术。
- 你对部署微服务应用到云上所涉及的内容有兴趣。

我选择写这本书有两个原因。首先，虽然我看到很多讲述微服务概念方面的书，但是我无法找到一本基于 java 实现微服务的好书。我一直认为自己是通晓多种编程语言的编程人员。Java 是我的核心编程语言，每当我创建新应用的时候，Spring 已经成为我最先想到的开发框架。当我接触到 Spring Boot 和 Spring Cloud 的时候，就被深深的打动了。我们使用 Spring Boot 和 Spring Cloud 构建运行在云上的微服务，大大简化了我们的开发周期。

其次，在我的整个职业生涯，我作为架构师和工程师一直工作，我发现很多时候我购买的技术书籍都倾向于两个极端。它们要么是没有具体的代码示例的概念，要么是对特定框架或编程语言的机械概述。我想要一本书，它将是架构和工程实施之间的桥梁和纽带。当你阅读本书的时候，我想给你真诚的介绍微服务开发模式和它们如何应用于现实世界的应用开发，

这些模式都是使用 Spring Boot 和 Spring Cloud 实现 ,经过实践的、易于理解的代码示例。

让我们改变一下，了解使用 Spring Boot 构建一个简单的微服务的过程。

## 1.5. 使用 Spring Boot 构建一个微服务

我一直认为，一个软件开发框架是深思熟虑的和易于使用，如果通过我亲切地称为“猴子身上测试”。如果像我这样的猴子能在 10 分钟以内找到一个框架，它就有希望。这是我第一次写 Spring Boot 服务示例时的感受。我希望你有同样的经历和快乐，所以让我们花一点时间看看如何使用 Spring Boot 编写一个简单的“Hello World”REST 服务。

在这一部分，我们不打算进行太多的代码展示。我们的目标是让你体验编写 Spring Boot 服务。我们会在第 2 章进行更详细介绍。

图 1.3 显示的是你的服务要做什么和 Spring Boot 微服务将如何处理用户请求的基本流程。

这个例子绝不是详尽或说明你应该如何建立一个生产水平的微服务 ,但它应该让你停一下，因为它能够教会你怎样编写更少的代码。在第 2 章之前，我们不打算讨论如何建立项目构建文件或代码的细节。如果你想看到 Maven pom.xml 文件和实际的代码，你可以在下载的代码的第 1 章中找到它。第 1 章所有的源代码可以从本书的 GitHub 库（https://github.com/carnellj/spmia-chapter1）获取。

***注意：***请务必先阅读附录 A，然后再尝试运行本书章节中的代码示例。附录 A 涵盖了本书中所有项目的总体工程布局，如何运行生成脚本，以及如何运行 Docker 环境。这一章中的示例代码都非常简单，它们被设计成不需要其他章节的信息就能运行在你本地的桌面环境。然而，在后面的章节中你会很快开始使用 Docker 运行本书用到的所有的服务和基础设施。不要过分阅读本书而不读附录 A 来设置桌面环境。

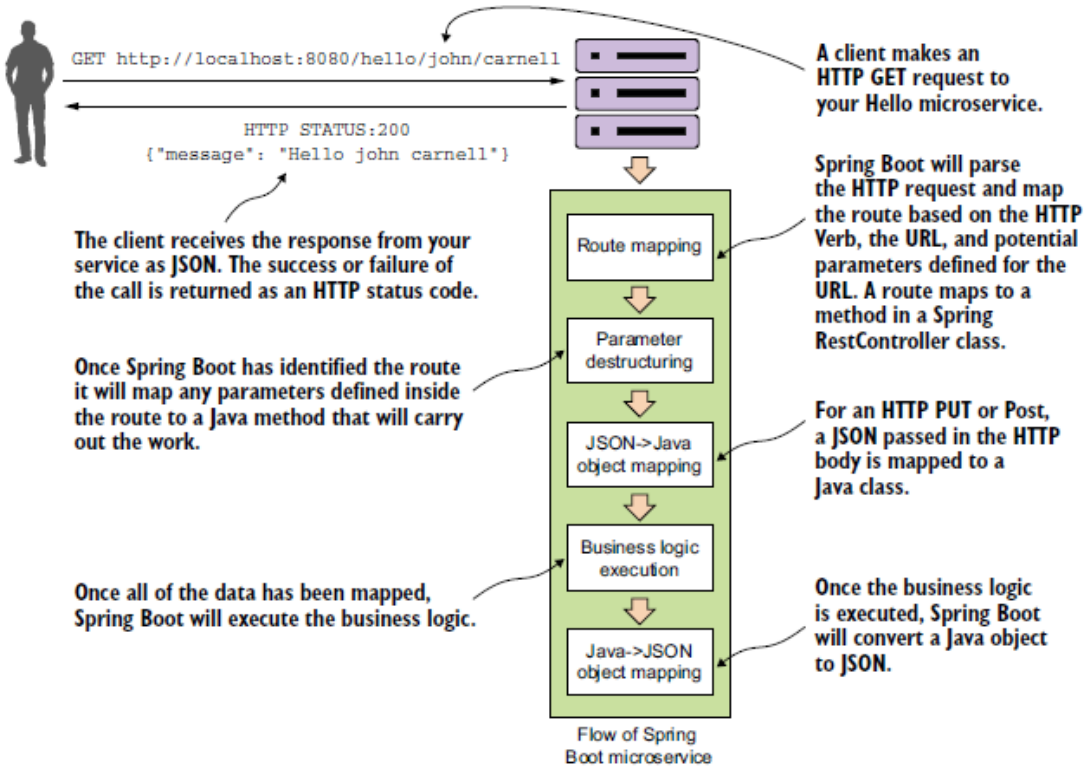


图 1.3 Spring Boot 抽象了 REST 微服务的共同功能（路由到业务逻辑，从 URL 解析 HTTP 参数，JSON / java 对象的数据转换），并让开发人员专注于服务的业务逻辑。

*① A client makes an HTTP GET request to your Hello microservice.*

客户端发出 HTTP GET 请求访问你的 Hello 微服务。

*② The client receives the response from your service as JSON. The success or failure of the call is returned as an HTTP status code.*

客户端接收到服务 JSON 格式的响应。调用的成功或失败被作为一个 HTTP 状态码返回。

*③ Once Spring Boot has identified the route it will map any parameters defined inside the route to a Java method that will carry out the work.*

一旦 Spring Boot 已经确定路由，它将映射里面定义的任何参数的路由到一个 java 方法，并进行工作。

*④ Once all of the data has been mapped, Spring Boot will execute the business logic.*

一旦所有的数据都被映射，Spring Boot 将执行业务逻辑。

⑤ *Route mapping*

路由映射

⑥ *Parameter destructuring*

参数解析

⑦ *JSON->Java object mapping*

JSON->Java 对象映射

⑧ *Business logic execution*

执行业务逻辑

⑨ *Flow of Spring Boot microservice*

Spring Boot 微服务流程

⑩ *Spring Boot will parse the HTTP request and map the route based on the HTTP Verb, the URL, and potential parameters defined for the URL.A route maps to a method in a Spring RestController class.*

Spring Boot 解析 HTTP 请求和根据 HTTP 谓词、URL 和 URL 定义的参数映射路由。在 Spring RestController 类中，有路由映射到的方法。

⑪ *For an HTTP PUT or Post, a JSON passed in the HTTP body is mapped to a Java class.*

通过 HTTP 的 PUT 或 Post 操作，HTTP 报文体内合法的 JSON 数据被映射到 java 类。

⑫ *Once the business logic is executed, Spring Boot will convert a Java object to JSON.*

执行业务逻辑之后，Spring Boot 将 Java 对象转换为 JSON。

这个例子，我们将有一个 Java 类 simpleservice/src/com/thoughtmechanix/application/simpleservice/Application.java，它将用于以访问路径为/hello 的 REST 端点暴露出来。

下面的清单显示了 Application.java 的代码。

**清单 1.1 Hello World with Spring Boot: 一个简单的 Spring 微服务**

```
package com.thoughtmechanix.simpleservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.PathVariable;

@SpringBootApplication
@RestController
@RequestMapping(value="hello")
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @RequestMapping(value="/{firstName}/{lastName}",
                    method = RequestMethod.GET)
    public String hello( @PathVariable("firstName") String firstName,
                         @PathVariable("lastName") String lastName) {
        return String.format("{\"message\":\"Hello %s %s\"}", firstName, lastName);
    }
}
```

①告诉 Spring Boot 框架，这是 Spring Boot 服务的入口点。

②告诉 Spring Boot，这个类将作为 Spring RestController 暴露。

③这个应用程序的所有的 URL 将以前缀/hello 进行暴露。

④Spring Boot 将端点暴露为基于 GET 的 REST 端点，它将接受两个参数：firstName 和 lastName。

⑤将 URL 传入的 firstName 和 lastName 参数映射到 hello 方法的两个变量

⑥返回一个你手动编写的简单 JSON 字符串。在第 2 章中，您不会创建任何 JSON。

在清单 1.1 中你主要暴露一个 HTTP GET 方式的端点，URL 传递两个参数（firstName 和 lastName），然后返回一个简单的 JSON 字符串，它的有效载荷包含消息“Hello firstName lastName”。如果你使用/hello/john/carnell 端点访问服务（我会很快展示），调用返回的将是{"message":"Hello john carnell"}。

让我们启动服务。进入命令提示符并发出以下命令：

**mvn spring-boot:run**

这个 mvn 命令 将使用一个 Spring Boot 插件 ,启动内嵌 Tomcat 服务器的应用程序。

**Java vs. Groovy and Maven vs. Gradle**

Spring Boot 框架为 java 和 Groovy 编程语言提供强大的支持。你可以使用 Groovy 构建微服务而无需项目设置。Spring Boot 还支持 Maven 和 Gradle 构建工具。本书的例子我使用 java 和 Maven。 作为一个长期对 Groovy 和 Gradle 酷爱的人，我对编程语言和构建工具更加关注。但要使本书可管理和受到更多的关注，我选择使用 java 和 Maven 以面向最大的读者群体。

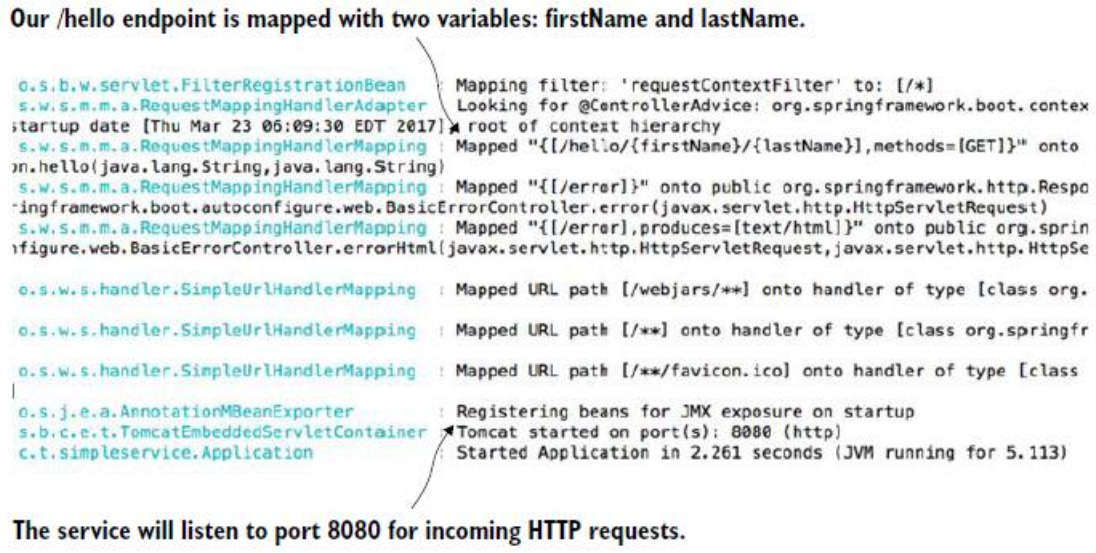


图 1.4 Spring Boot 服务将通过控制台的服务端口，与暴露的端点通信。

① *Our /hello endpoint is mapped with two variables: firstName and lastName.*

我们的/hello 端点被两个变量 firstName 和 lastName 映射。

② *The service will listen to port 8080 for incoming HTTP requests.*

该服务将侦听 HTTP 请求的 8080 端口。

如果一切都正常启动，你应该会从命令行窗口看到图 1.4 所示的内容。

如果仔细观察图 1.4，你会注意到两件事情。首先，一个 Tomcat 服务器使用端口 8080 启动。其次，/hello/{firstName}/{lastName}端点以 GET 方式暴露在服务器上。

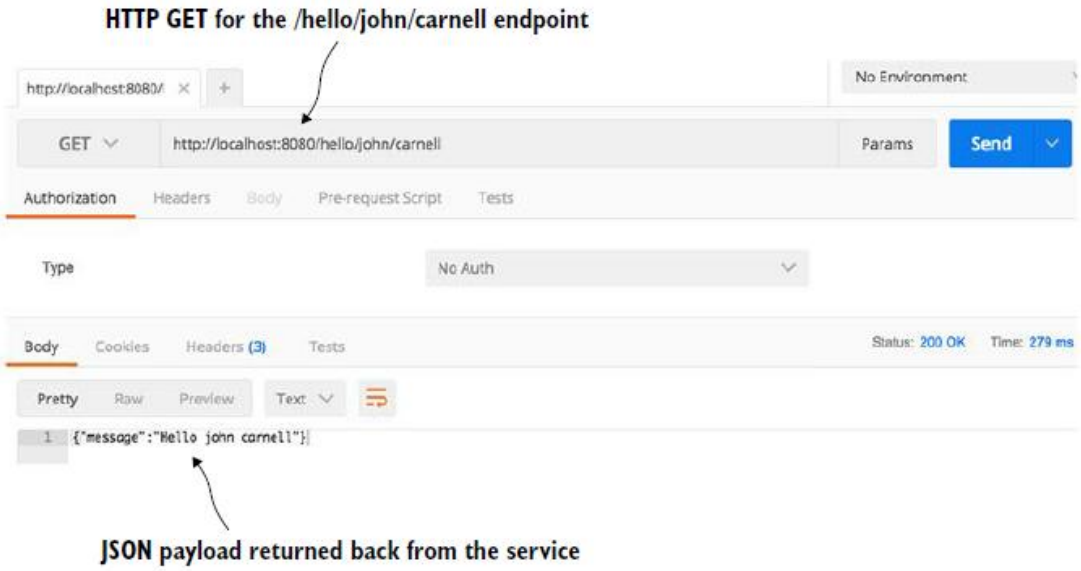


图 1.5 来自/hello 端点的响应显示了你请求的数据表示为一个 JSON 有效载荷。

① *HTTP GET for the /hello/john/carnell endpoint*

HTTP GET 方式的/hello/john/carnell 端点

② *JSON payload returned back from the service*

从服务返回的 JSON 有效载荷

你使用基于浏览器的 REST 工具 POSTMAN(https://www.getpostman.com/)调用你的服务。许多工具，包括图形和命令行，可供调用的基于 REST 的服务，但在本书所有的例子中我将使用 POSTMAN。图 1.5 显示了使用 POSTMAN 调用 http://localhost:8080/hello/john/carnell 端点和从服务返回的结果。

显然，这个简单的例子不证明 Spring Boot 的功能强大。但它所要表明的是，你可以在 java 中只要写 25 行代码 ,就能编写一个带 URL 参数路由映射 ,并基于 HTTP 协议 JSON 格式的 REST 服务。有经验的 java 开发人员会告诉你，在 java 中使用 25 行代码编写任何有意义的东西都是非常困难的。Java 是一门功能强大的语言 ,在与其它语言广泛的比较中，

已经获得良好的声誉。

我们完成了 Spring Boot 基本情况的介绍。我们现在必须问这个问题：这是否意味着我们应该着手开始使用微服务编写我们的应用？下一节 ,我们将介绍什么时候着手使用微服务构建应用程序是合理的原因。

## 1.6. 为什么要改变我们构建应用程序的方式？

我们正处在历史的转折点。现代社会的各个方面，通过互联网连接在一起。曾经为当地市场服务的公司突然发现他们可以接触到全球客户群。然而，一个更大的全球客户群也伴随着全球竞争。这些竞争压力意味着以下因素影响开发人员必须考虑构建应用程序的方式：

- 日益增加的复杂性：客户期望他们能够了解一个组织的所有部分。仅访问一个数据库、孤立的应用程序，不与其他应用程序集成不再是常态。当前的应用程序需要访问不只在公司数据中心内的多个服务和数据库 ,而且还要访问互联网上的提供商提供的外部服务。
- 客户希望更快的发布：客户不再希望等待下一个软件包的年度发布或版本。相反，他们希望将一个软件产品的功能分解开来，使新的功能可以在一星期（甚至几天）迅速发布而无需等待一个完整的产品发布。
- 性能和可伸缩性 :单体应用很难预测当交易量高峰期的时候应用程序能够处理多少交易量。应用程序需要能够跨多个服务器快速扩展，然后在需求量已经过去时缩小规模。
- 客户希望他们的应用程序是可用的：因为如果客户想比竞争者更成功，公司的应用程序就必须具有高度的弹性。应用程序某一部分的故障或问题不应导致整个应用程序失败。

为了满足这些期望，作为应用程序开发人员，我们不得不接受这样一个悖论：为了构建高可伸缩性和高度冗余的应用程序，我们需要将应用程序分解成可以独立构建和部署的小型服务。如果我们将单体应用程序拆分成小服务，我们能够使系统具有以下特性：

- 灵活的：解耦的服务可以由重新快速提供新的功能。使用的代码单元越小，更改代码的时间越少，测试部署代码所需的时间就越少。
- 有弹性的：解耦服务意味着应用程序不再是单一的“泥球”，其中应用程序的某一部分的恶化导致整个应用程序失败。故障可以定位到应用程序的一小部分，并在整个应用程序经历中断之前得到控制。这也使得应用程序能够完全降低发生不可恢复错误的几率。
- 可扩展的：解耦的服务可以很容易地跨多个服务器水平分布，从而可以适当地扩展功能/服务。在一个单体应用程序中，应用程序的所有逻辑相互交织在一起，整个应用程序若需要扩展，哪怕只是很小的一部分，都将成为应用的瓶颈。

为此，我们开始微服务的讨论，请牢记以下的内容：

小的，简单的，解耦的服务 = 可扩展的，有弹性的，灵活的应用

## 1.7. 云究竟是什么？

术语“云”已经被过度使用。每个软件供应商通过大肆宣传，宣称都提供云，每个平台都是基于云计算的。云计算中存在三种基本模型，它们是：

- Infrastructure as a Service (IaaS)
- Platform as a Service (PaaS)
- Software as a Service (SaaS)

为了更好地理解这些概念，让我们把每天做饭的任务映射到不同的云计算模型。

当你想要吃一顿饭，你有四个选择：

① 你可以在家做饭。

② 你可以去杂货店买预先做好饭，你自己加热。

③ 你可以通过快递订购一份送到家里的饭。

④ 你可以乘车到餐馆吃饭。

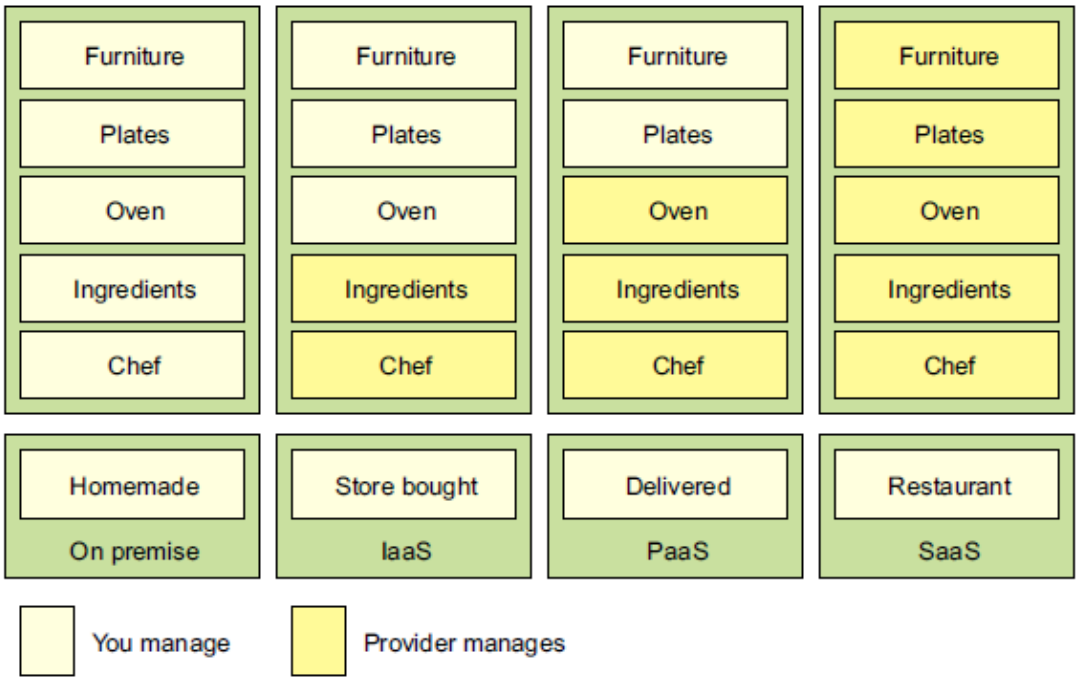


图 1.6 不同的云计算模型归结为谁负责什么：云供应商或你。

| 名称 | 说明 |
| --- | --- |
| Furniture | 家具，设备 |
| Plates | 盘子 |
| Oven | 烤箱，烤炉，灶 |
| Ingredients | （烹调的）原料 |
| Chef | 厨师，大师傅 |
| Homemade | （衣服、食品等）自家制的 |

| | |
|---|---|
| Store bought | 商店购买 |
| Delivered | 交付 |
| Restaurant | 餐馆 |
| On premise | 自行管理 |
| IaaS | 基础设施即服务 |
| PaaS | 平台即服务 |
| SaaS | 软件即服务 |
| You manage | 你管理 |
| Provider manages | 供应商管理 |

图 1.6 展示了每个模型。

这些选项的不同之处在于谁负责做饭和在哪里做饭。在先决条件模式下，在家吃一餐需要你做所有的工作，用你自己的烤箱和家里的配料。买饭就像使用基础设施即服务（IaaS）计算模型。 你正在使用商店的厨师和烤箱来烘烤这顿饭，但你仍然负责加热饭菜，然后在家里吃（然后洗碗）。

在平台即服务（PaaS）的模型，你仍然负责做饭，但是你更多的是依赖于供应商的服务来完成与做饭相关联的核心任务。例如，在 PaaS 模式，你提供盘子和家具，餐厅老板提供烤箱，配料和烹饪厨师。在软件即服务（SaaS）模式中，就好比你去一家餐馆，他们为你准备所有的食物。你在餐厅吃饭，然后你吃完之后买单。你无需准备或清洗餐具。

使用这些模型的关键因素是以下内容的决策 :谁负责维护基础设施？以及构建应用程序用什么技术？在 IaaS 模型，云供应商提供的基础设施，但你负责选择技术和提供最终解决方案。另一方面，使用 SaaS 模型，你是供应商提供的服务的被动使用者，对技术选择或维护应用程序的基础设施没有任何责任。

**新兴的云平台**

我有据可查，目前正在使用的有三个核心云平台类型（IaaS、PaaS、SaaS）。然而，新的云平台类型不断涌现。这些新的平台包括功能即服务（FaaS）和容器即服务（CaaS）。Amazon 的 Lambda 技术和 Google 云功能构建无需服务器就可部署的代码块使用的技术都是基于 FaaS(https://en.wikipedia.org/wiki/Function_as_a_Service)应用，它们完全运行在提供商的云计算平台中。使用 FaaS 平台，你不必管理任何服务器基础设施，只需为执行该函数所需的计算周期付费。你不必管理任何服务器基础设施，只需支付执行该函数所需的计算周期。

在容器即服务（CaaS)模型，开发者可以将他们的微服务作为云提供商的轻便虚拟容器（如 Docker）进行构建和部署。不像 IaaS 模型，你的开发人员需要考虑将虚拟机部署到那里。使用 CaaS，你在一个轻量级的虚拟容器中部署你的服务。云提供商运行容器的虚拟服务器上运行用于构建、部署、监视和伸缩容器的各种工具。Amazon 的弹性容器服务(ECS) 就是一个基于 CaaS 平台的例子。在本书的 10 章，我们将看到如何部署微服务到 Amazon 的 ECS。

需要注意的是，了解云计算的 FaaS 和 CaaS 模型的重要性，你才能继续构建基于微服务的架构。牢记，围绕微服务概念构建微小的、单一职责的服务，使用基于 HTTP 的接口与其通信。新型的云计算平台，如 FaaS 和 CaaS，是基础设施部署微服务的替代机制。

## 1.8. 为什么是云和微服务？

基于微服务架构的核心概念之一是每个服务都作为独立的组件被打包和部署。服务实例应该能够快速启动。服务的每个实例之间互不干扰，都应该与其他实例区分开来。

作为一位微服务的开发者，你将不得不决定是否将你的服务部署到以下的环境中：

- 物理服务器：虽然你可以构建和部署你的微服务到物理机，但很少有机构这样做是因为物理服务器的有限制。你不能快速提升一台物理服务器的处理能力。在多台物理服务器上水平伸缩你的微服务是非常昂贵的。
- 虚拟机镜像 :微服务一个关键的好处是它们具有快速启动和关闭服务实例以响应应用伸缩和服务故障事件处理的能力。虚拟机是主要的云计算提供商提供的核心服务。一个微服务可以被打包在一个虚拟机镜像。在任何一个 IaaS 私有云或公共云，服务的多个实例可以快速部署和启动。
- 虚拟容器：虚拟容器是在虚拟机镜像里部署微服务的自然延伸。与其将一个服务部署到一台完整的虚拟机，许多开发者将他们的服务部署到云上的 Docker 容器（或者等效的容器技术）。虚拟容器运行在一台虚拟机内部；使用虚拟容器，你可以将单个虚拟机分隔成一系列独立的、共享相同的虚拟机镜像的进程。

基于云的微服务的优势是围绕弹性的概念。云服务提供商能够为你提供在几分钟内快速地生成新的虚拟机和容器的能力。如果你的服务需求下降，你可以在不产生任何额外成本的情况下向下收缩虚拟服务器的数量。使用云提供商的设施部署你的微服务能够提高你应用横向扩展的能力（增加更多的服务器和服务实例）。服务器的伸缩性也意味着你的应用程序可以更具弹性。如果你的一个微服务有问题甚至宕机，生成新的服务实例，可以让你的应用程序处于更长的活动状态，使你的开发团队能够很好的解决问题。

在这本书中，所有的微服务和相应服务的基础设施将被部署到一个基于 IaaS 云提供商提供的 Docker 容器。

微服务传统的部署哲学：

- 简化基础设施管理：IaaS 云提供商给你提供对服务最大的控制能力。使用简单的 API 调用，新的服务可以启动和停止。一个 IaaS 云解决方案，你只需为你使用的

基础设施付费。

- 大规模的横向扩展能力：IaaS 云提供商能够让你快速地启动一个服务的一个或多个实例。这种能力意味着你可以快速伸缩服务和绕过有问题或已宕机的服务器。
- 通过地理分布实现高冗余：根据需要，IaaS 供应商有多个数据中心。通过使用 IaaS 云提供商部署你的微服务，你能获得一个比在数据中心中使用集群更高级别的冗余。

**为什么不是基于 Paas 的微服务？**

在本章前面，我们讨论了三种类型的云平台（基础设施即服务、平台即服务和软件即服务）。在这本书中，我选择特别关注使用基于 IaaS 的方法构建微服务。尽管某些云供应商会让你远离部署微服务的基础设施，但我选择保持独立于供应商，并亲自部署应用程序及相关资源（包括服务器）。

例如，Amazon, Cloud Foundry 和 Heroku 为你提供了不必知道底层应用程序容器就能部署服务的能力。它们提供了 Web 接口和 API，允许你将应用程序部署为 WAR 或 JAR 文件。你不需要设置和调整应用程序服务器和对应的 java 容器。虽然这样很方便，但每个云服务提供商的平台有不同的特性，这与个别的 PaaS 解决方案相关。

IaaS 的方式，尽管需要做更多的工作，但可以跨多个云提供商，让我们的资料适应更广泛的读者。就个人而言，我发现基于 PaaS 的解决方案将能够让你很快的开始你的开发工作，但是一旦你的应用达到足够的微服务，你开始需要云提供商提供灵活的 IaaS 方式。

在本章前面，我提过了一些新兴的云计算平台，如：功能即服务(FaaS)和容器即服务(CaaS)。如果你不小心，将 FaaS 平台方式可以将代码绑定到云供应商平台中，因为你的代码被部署到特定供应商的运行时引擎。使用基于 FaaS 的模型，你可以使用通用编程语言编写你的服务（java、python，JavaScript，等等），但你在一定程度上仍然将潜在供应商的 API 和程序部署的运行时环境绑定在一起。

在本书中的服务将被打包成 Docker 容器。我选择 Docker 作为一个容器技术其中的一个原因是，Docker 能够被部署到所有主要的云计算提供商。在后面的第 10 章，我将演示如何使用 Docker 打包微服务，并且怎样将这些容器部署到 Amazon 的云平台。

## 1.9. 微服务不仅仅是编写代码

虽然构建个别微服务的概念很容易理解，运行和支持一个强大的微服务应用（特别是运行在云中的）涉及服务更多的编写代码。写一个强大的服务需要考虑几个因素。图 1.7 突出显示了这些因素。

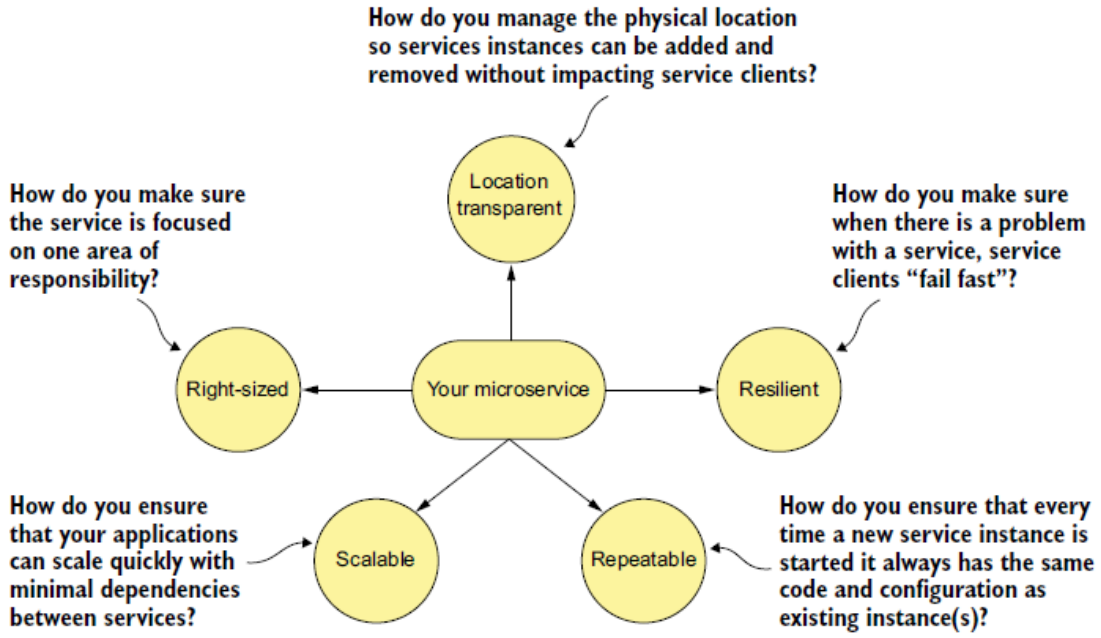


图 1.7 微服务不仅仅是业务逻辑。你需要考虑服务将要运行的环境以及服务将如何扩展并具有弹性。

① *Location transparent*

位置透明

② *How do you manage the physical location so services instances can be added and removed without impacting service clients?*

如何管理物理位置，以便在不影响客户端服务的情况下添加和删除服务实例？

*③ Right-sized*

合适的大小

*④ How do you make sure the service is focused on one area of responsibility?*

你如何确保服务集中在一个责任领域？

*⑤ Scalable*

可扩展的

*⑥ How do you ensure that your applications can scale quickly with minimal dependencies between services?*

你如何确保应用程序能够在服务之间的最小依赖关系下快速扩展？

*⑦ Repeatable*

可复用的

*⑧ How do you ensure that every time a new service instance is started it always has the same code and configuration as existing instance(s)?*

你如何确保每次启动新的服务实例时，它始终具有与现有实例相同的代码和配置？

*⑨ Resilient*

有弹性的

*⑩ How do you make sure when there is a problem with a service, service clients "fail fast"?*

当服务出现问题时，如何确保客户端服务“快速失败”？

让我们详细讨论图 1.7 显示的这些因素：

- 合适的大小：你如何确保你的微服务具有合适的大小，这样就不会让一个微服务承

担太多的职责？记住，一个合适大小的服务可以让你快速更改应用程序并降低整个应用程序的整个中断的风险。

- 位置透明：在一个微服务应用，多个服务实例可以快速启动和关闭，你如何管理服务调用的物理细节？
- 有弹性的：你如何保护你的微服务消费者和路由失败的服务应用程序的完整性，并确保你有一种“快速失败”的方法？
- 可复用的：你如何确保生成的服务的每个新实例都具有与生产中所有其他服务实例相同的配置和基础代码？
- 可扩展的：你如何使用异步处理和事件最小化服务之间的直接依赖并确保你能优雅的扩展你的微服务？

这本书以模式为基础的方法为我们回答这些问题。使用基于模式的方法，我们列出了可以在不同的技术实现中使用的通用设计。虽然我选择在本书中使用 Spring Boot 和 Spring Cloud 来实现这些模式，但没有任何东西可以阻止你使用此处介绍的概念，并将其与其他技术平台一起使用。具体来说，包括以下六类微服务模式：

- 核心开发模式
- 路由模式
- 客户端弹性模式
- 安全模式
- 日志记录和跟踪模式
- 构建和部署模式

让我们更详细地介绍这些模式。

### 1.9.1. 微服务核心开发模式

微服务核心开发模式是构建微服务最基础的部分。图 1.8 高亮显示的主题，我们将贯穿于基本的服务设计过程中。

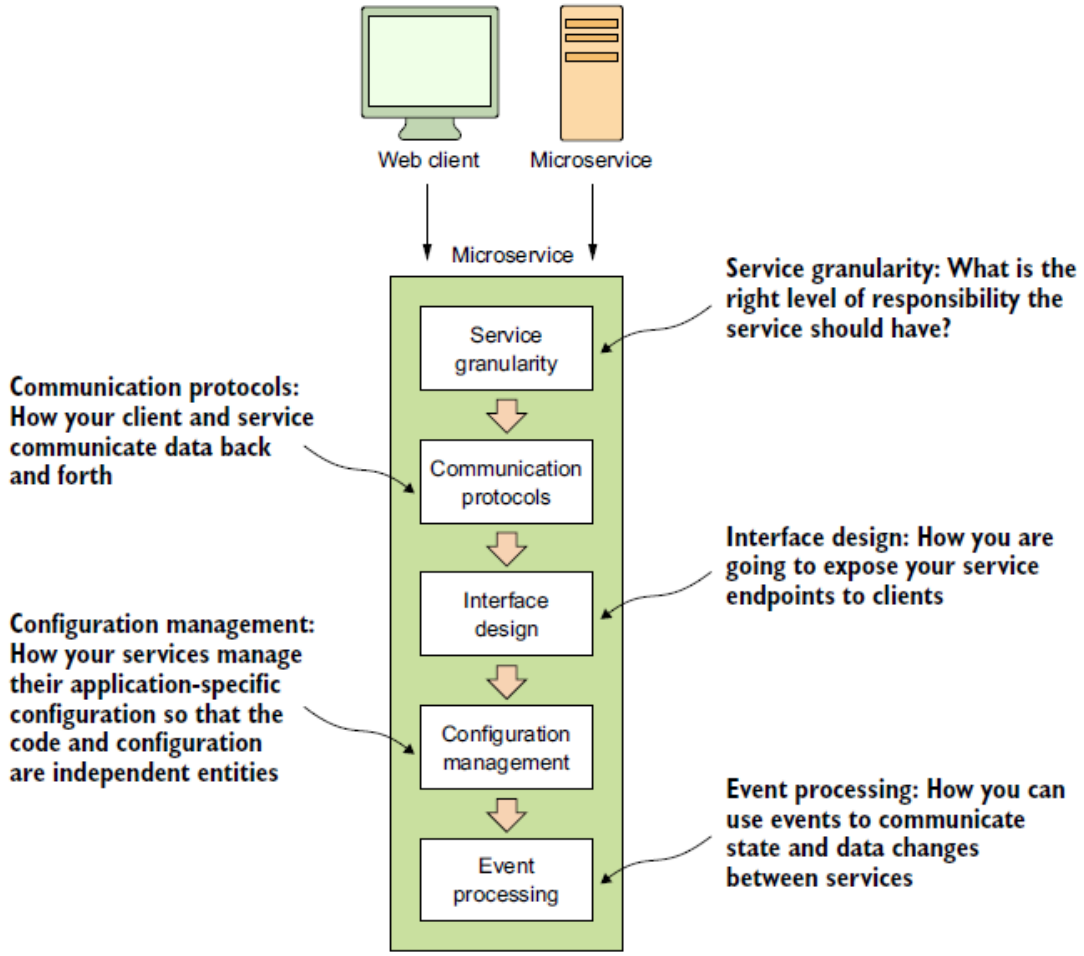


图 1.8 当你正在设计微服务时，你必须考虑服务将如何被消费和它如何与外部通信。

*① Service granularity:What is the right level of responsibility the service should have?*

服务粒度：服务具有什么正确层次的职责？

*② Communication protocols:How your client and service communicate data back and forth*

通信协议：客户端和服务端如何来回传输数据

*③ Interface design: How you are going to expose your service endpoints to clients*

接口设计：你计划如何向客户端暴露你的服务端点

*④ Configuration management: How your services manage their application-specific configuration so that the code and configuration are independent entities*

配置管理：你的服务如何管理其特定于应用程序的配置，从而使代码和配置是独立的实体

*⑤ Event processing: How you can use events to communicate state and data changes between services*

事件处理：如何使用事件在服务之间传递状态和更改数据

- 服务粒度 :你如何将业务领域分解成微服务以使每个微服务都具有正确层次的职责？服务粒度过粗 ,它的职责重叠到不同的业务问题域 ,使得服务随时间变化难以维护。服务粒度过细 ,它会增加应用程序的整体复杂性 ,并将服务变为 “哑” 数据抽象层，除了访问数据存储所需的逻辑之外，它没有逻辑。我在第 2 章讨论服务粒度。
- 通信协议：开发者将如何与你的服务通信？你会使用 XML（可扩展标记语言）、JSON（JavaScript 对象标记法），或一个二进制协议，如 Thrift 向你的微服务来回发送数据？我们将分析对于微服务为什么 JSON 是最理想的选择。同时，JSON 已经成为向微服务发送和接收数据最常见的选择。我在第 2 章讨论通信协议。
- 接口设计 :设计开发者计划用于调用你的服务的真实接口 ,最好的方法是什么？你如何构造能够表达服务含义的 URLs？你的服务如何进行版本控制？一个好的微服务接口设计让你的服务更直观。我在第 2 章讨论接口设计。
- 服务配置管理 :你如何管理你的微服务配置使它迁移到不同的云环境 ,你永远不需要改变应用程序核心代码或配置？我在第 3 章讨论服务配置管理。

- 服务间事件处理 :你如何让你的微服务使用事件以尽量减少微服务之间的硬编码依赖关系和增加应用的弹性？我在第 8 章讨论服务间事件处理。

### 1.9.2. 微服务路由模式

微服务路由模式用来处理客户端应用程序。该客户端应用程序要消费一个被发现服务位置和被路由到该位置的微服务。在一个基于云的应用程序，你可能有数百个正在运行的微服服务实例。你需要从这些物理 IP 地址抽离出来，且有一个单一的服务调用入口，使你可以始终为所有服务调用加强安全性和内容策略。

服务发现和路由回答这个问题，“我如何得到我的客户到一个指定服务实例的服务请求？”

- 服务发现 : 你如何使你的微服务是可发现的，以使客户端应用程序可以发现它们而不必要将服务的位置硬编码到应用程序？你如何确保失常的微服务实例能够被从可用服务实例池中移除？我在第 4 章讨论服务发现。
- 服务路由 : 你如何为你所有的服务提供一个单一的入口点，使安全策略和路由规则统一适用于多种服务和你微服务应用的服务实例？你如何确保团队中的每个开发人员不必拿出向他们的服务提供路由的解决方案？我在第 6 章讨论服务路由。

在图 1.9 中，服务发现和服务路由似乎在它们之间有一个硬编码的情况（首先是服务路由和服务发现）。然而，两种模式互不依赖。例如，我们可以实现服务发现而无需服务路由。你能实现服务路由而无需服务发现（虽然它的实现更加困难）。

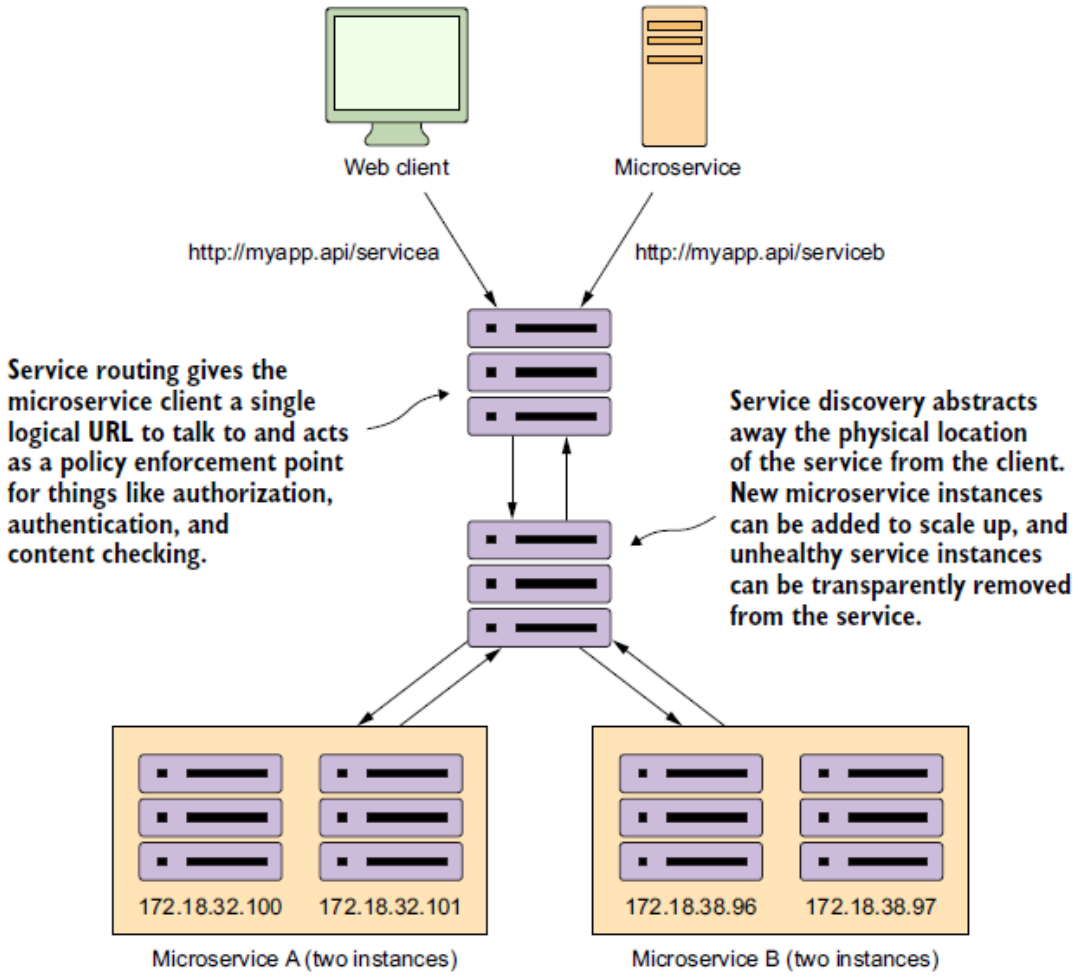


图 1.9 服务发现和路由是任何大规模微服务应用的关键部分。

① *Service routing gives the microservice client a single logical URL to talk to and acts as a policy enforcement point for things like authorization, authentication, and content checking.*

服务路由给微服务客户端一个可访问的逻辑 URL，作为一个策略执行点，如授权、认证和内容检查。

② *Service discovery abstracts away the physical location of the service from the client.New microservice instances can be added to scale up, and unhealthy service instances can be transparently removed from the service.*

服务发现使客户端从服务的物理地址抽离出来。新的微服务实例可以按比例的增加，不健康

的服务实例可以透明地从服务中删除。

### 1.9.3. 微服务客户端弹性模式

因为微服务架构是高度分散的，你需要重视如何预防一个这样的问题：一个单一的服务（或服务实例）对服务消费者上下级联。为此，我们将讨论四种客户端弹性模式：

- 客户端负载均衡：你如何在客户端服务缓存你服务实例的位置，使客户端调用一个微服务的多个实例时能够被负载均衡到所有健康的微服务实例？
- 断路器模式 :如何防止客户继续调用失败或遭遇性能问题的服务？当服务运行缓慢时，客户端调用它很消耗资源。你想失败的微服务调用能够快速失败，使调用客户端可以快速响应并采取适当的措施。
- 回退模式：当服务调用失败时，你如何提供一个“插件”机制，允许服务客户端尝试通过调用其他微服务的方式来执行其工作？
- 舱壁模式：微服务应用使用多个分布式资源来完成他们的工作。如何隔离这些调用使一个异常服务调用不会对应用程序的其余部分产生负面影响？

图 1.10 显示了这些模式如何保护服务消费者免受异常服务的冲击。我在第 5 章讨论这四个主题。

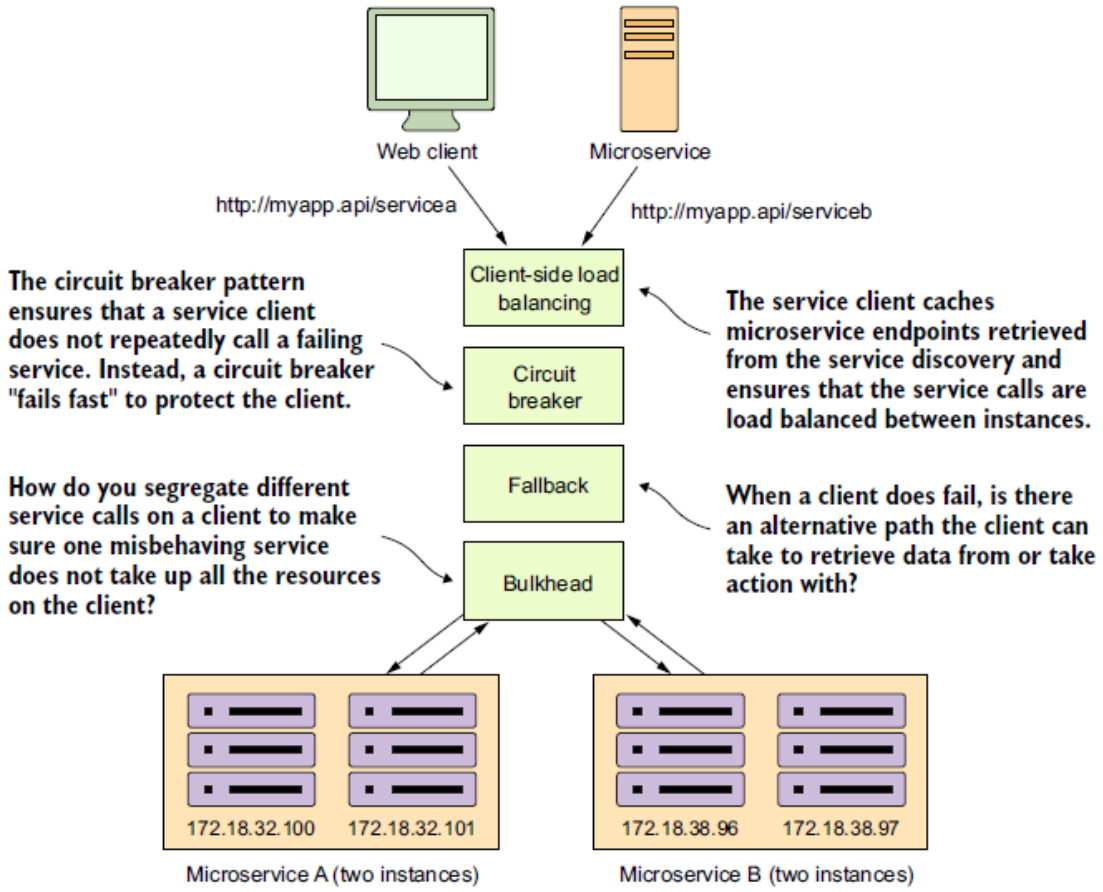


图 1.10 对于微服务，你必须保护在异常服务中的服务调用者。记住，慢服务或宕机的服务会造成即时服务之外的中断。

① *Client-side load balancing*

客户端负载均衡

② *The service client caches microservice endpoints retrieved from the service discovery and ensures that the service calls are load balanced between instances.*

客户端服务缓存从服务发现取回的微服务端点和确保服务调用实例之间的负载均衡。

③ *Circuit breaker*

断路器

④ *The circuit breaker pattern ensures that a service client does not repeatedly call a failing service. Instead, a circuit breaker "fails fast" to protect the client.*

断路器模式确保一个客户端服务不能重复调用失败的服务。相反，一个断路器能用“快速失败”来保护客户端。

⑤ *Fallback*

回退

⑥ *When a client does fail, is there an alternative path the client can take to retrieve data from or take action with?*

当客户端确实失败时，是否有另一种方式，客户端可以从中检索数据或采取措施？

⑦ *Bulkhead*

舱壁

⑧ *How do you segregate different service calls on a client to make sure one misbehaving service does not take up all the resources on the client?*

你如何隔离不同的客户端服务调用，使失败的服务不占用所有的客户端资源？

### 1.9.4. 微服务安全模式

我如果不谈微服务安全，就不能写一本有关微服务的书。在第 7 章，我们将讨论三个基本的安全模式。它们是：

- 认证：你如何确定调用服务的服务客户端是谁？
- 授权：你如何确定服务客户端调用微服务是被允许的试图采取行动的行为？
- 证书管理和传播 :如何防止服务客户端经常地为事务中的服务调用提供凭据？具体来说，我们将看看如何基于令牌的安全标准，如：OAuth2 和 JavaScript Web Tokens (JWT) 可以用来获得一个令牌，它被从服务调用传递给服务调用来为用户提供认证和授权。

图 1.11 显示了如何实现以前描述的三种模式来建立一个认证服务，让它保护你的微服务。

在这点，我不打算在图 1.10 中太深入的详细讨论。还有一个原因，就是安全需要一整章。（它本身就可以写一本书）

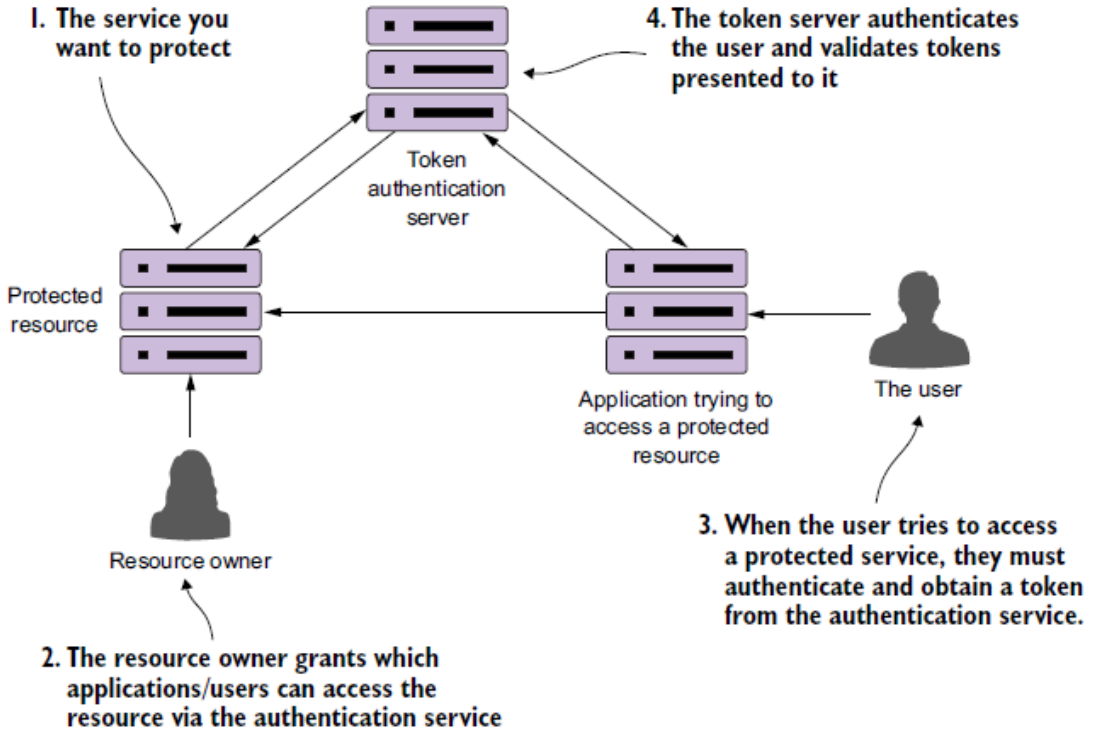


图 1.11 使用基于令牌的安全方案，可以在不传递客户凭证的情况下实现服务身份验证和授权。

### 1.9.5. 微服务日志记录和跟踪模式

微服务的优势是将单体应用程序分解成小的，可以独立部署的一个功能块。微服架构的不足之处是，在你的应用程序和服务它更难调试和跟踪。

为此，我们来看看三个核心日志和跟踪模式：

- 日志关联分析 :你是如何将单个用户事务服务之间所有产生的日志联系在一起的？用这种模式，我们将研究如何实现一个关联 ID，它是一个惟一标识符，它将在事务中的所有服务调用中进行传递，并可用于将来自每个服务的日志条目绑定在一起。

- 日志聚合：这种模式我们将看看如何将你的微服务（和它们个别的实例）产生的日志推送到一个可查询的数据库。我们也将看到如何使用关联 ID 协助搜索你的聚合日志记录。
- 微服务跟踪：最后，我们将探讨如何使所有涉及服务调用的客户端流程可视化，并了解事务中涉及的服务性能特征。

图 1.12 显示了如何将这些模式组合在一起。我们将会在第 9 章更详细的讨论日志和跟踪模式。

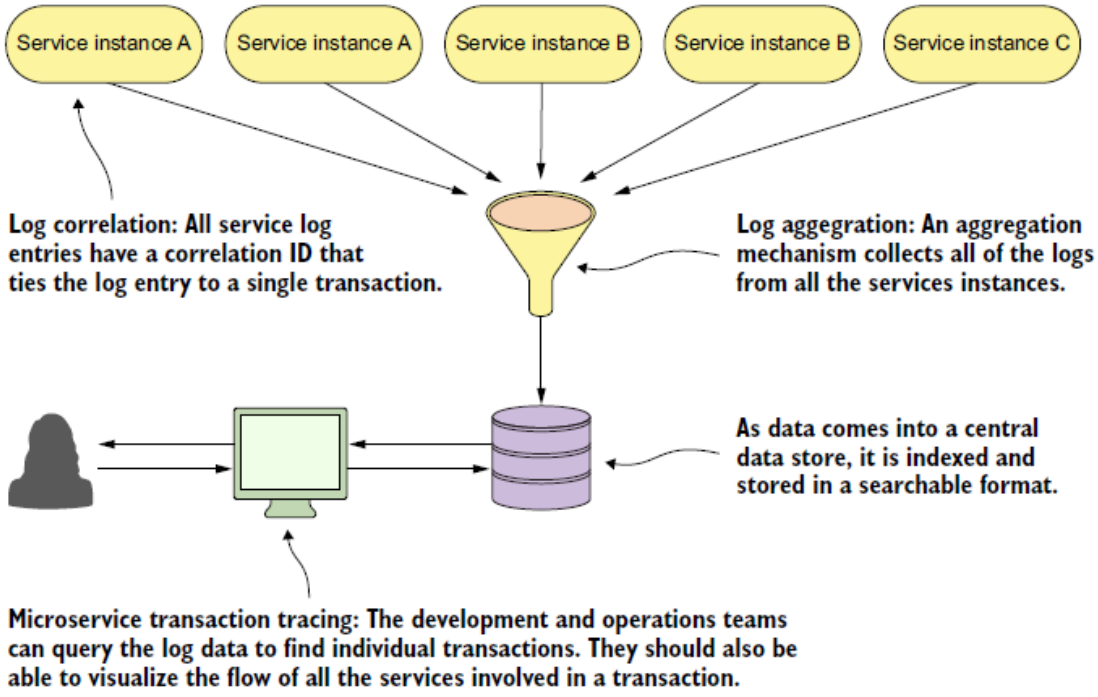


图 1.12 成熟的日志记录和跟踪策略使跨多个服务的调试处理可管理。

*① Log correlation: All service log entries have a correlation ID that ties the log entry to a single transaction.*

日志关联分析：所有服务日志条目都有一个相关 ID，将日志条目绑定到一个交易。

*② Log aggregration: An aggregation mechanism collects all of the logs from all the services instances.*

日志聚合：聚合机制收集来自所有服务实例的所有日志。

*③ Microservice transaction tracing: The development and operations teams can query the log data to find individual transactions. They should also be able to visualize the flow of all the services involved in a transaction.*

微服务交易跟踪：开发和运营团队可以查询日志数据中找出个别交易。他们也应该能够看到所有参与交易的服务流程。

*④ As data comes into a central data store, it is indexed and stored in a searchable format.*

当数据进入一个中央数据存储时，它被索引并存储在一个可搜索的格式中。

### 1.9.6. 微服务构建和部署模式

微服务架构核心部分之一是一个微服务的每个实例应该与它的其它所有实例是相同的。你不能允许“配置漂移”（服务器部署后发生的某些更改），因为这可以在你的应用程序引入不稳定性。

**常说的话**

多年来，我一直在研究关键情况团队，许多下行系统的解析通常是从开发人员或系统管理员的话开始的。工程师（和大多数人）具有良好的操作技能。他们不会在工作中犯错误或搞垮系统。相反，他们做他们能做的最好的，但他们忙或心不在焉。他们在服务器上调整一些东西，在所有环境中进行操作并打算回退。

稍后，停电发生了，每个人都在挠着头，想知道这与生产中较低级的环境有什么不同。我已经发现一个微服务规模小和有限的范围，使它有完美的机会引入“不可变的基础设置”为组织理念：一旦部署了服务，其运行的基础设施就不再被人类的双手所触及。

一个不可变的基础设施是利用微服架构成功的关键部分 ,因为你必须保证你在生产中对一个特定的微服务启动的每个微服务实例，它都和其兄弟实例是相同的。

为此，我们的目标是整合配置你的基础设施完全融入到你部署构建的过程，使你不再向一个已经运行的基础设施部署软件构件，如：Java WAR 或 EAR 包。相反，你要建立和编译你的微服务和虚拟机镜像，这将作为构建过程的一部分。相反，你要建立和编译你的市微服和虚拟服务器上运行的图像是作为构建过程的一部分。然后，当你的微服务被部署后，整个在服务器上运行的虚拟机镜像也被部署。

图 1.13 说明了这个过程。在书的最后，我们将看看如何改变你的构建和部署管道使你的微服务和服务器作为一个工作单元运行它们的部署。在第 10 章中，我们将介绍以下模式和主题：

- 构建和部署管道：你如何创建一个可重复的构建和部署的过程，突出一键生成和部署到你的组织任何环境中？
- 基础设施即代码：你如何看待将服务配置为可以在源代码管理下执行和管理的代码？
- 不可变的服务器：一旦微服务镜像被创建，你如何保证它从未在部署后改变？
- 凤凰服务器：服务器运行时间越长，配置漂移的机会越多。你如何确保运行微服务的服务器定期卸载和重建不可变的镜像？

我们使用这些模式和主题的目标是，在影响到你的上层环境（如阶段或生产）之前，尽可能快地暴露和消灭配置漂移。

**注意**：本书中的代码示例（除了第 10 章），一切将在你的桌面机器上本地运行。第 1 章和第 2 章的示例可以直接在命令行本地运行。从第 3 章开始，所有的代码将被编译和运行 Docker 容器。

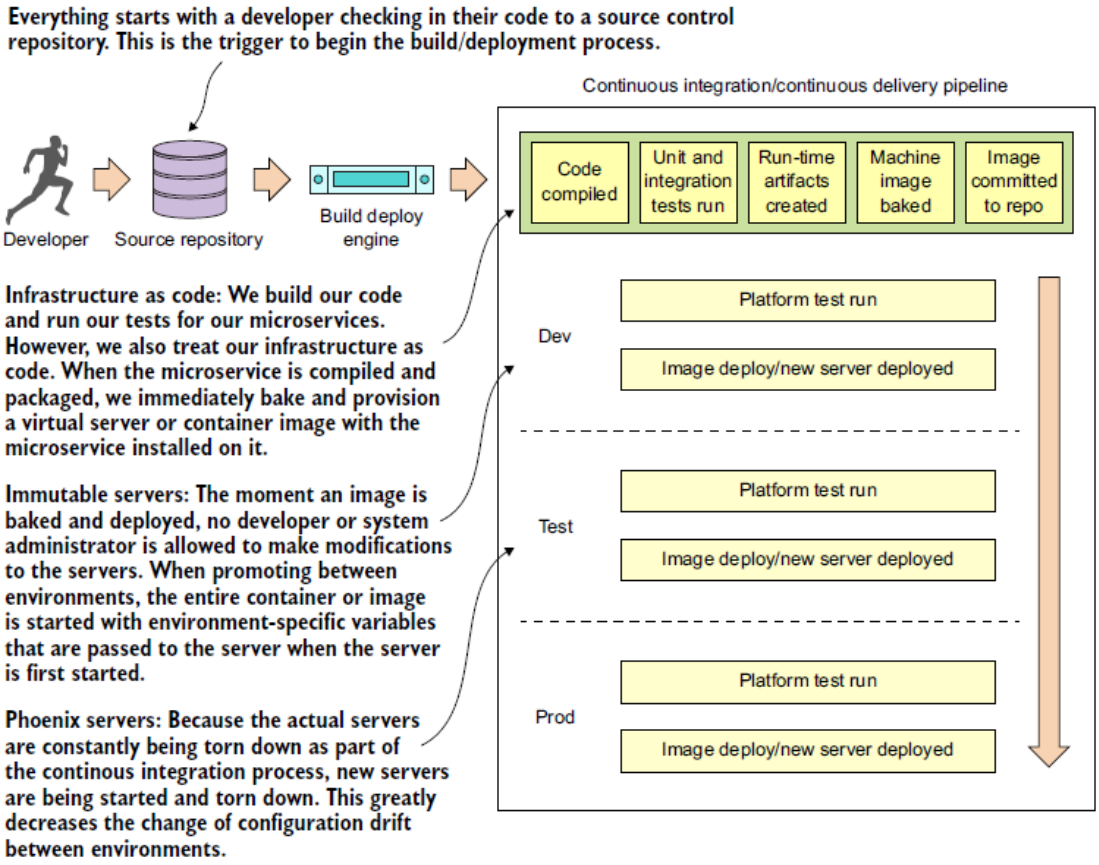


图 1.13 你想要的微服务部署和它运行的服务器是极小的构件，它们作为一个整体被部署。

① *Continuous integration/continuous delivery pipeline*

持续集成/连续的输送管道

② *Code compiled*：代码编译

*Unit and integration tests run*：运行单元和集成测试

*Run-time artifacts created*：创建运行时构件

*Machine image baked*：制作虚拟机镜像

*Image committed to repo*：提交镜像到仓库

③ *Everything starts with a developer checking in their code to a source control repository. This is the trigger to begin the build/deployment process.*

一切都开始于一个开发人员将他们的代码提交到源代码管理库。这是开始构建/部署过程的触发器。

*④ Infrastructure as code: We build our code and run our tests for our microservices.However, we also treat our infrastructure as code. When the microservice is compiled and packaged, we immediately bake and provision a virtual server or container image with the microservice installed on it.*

基础设施即代码：我们为微服务构建代码并运行测试。然而，我们也把我们的基础设施当做代码对待。当微服务被编译和打包，我们立即制作和设置虚拟服务器或已安装上微服务的容器镜像。

*⑤ Immutable servers: The moment an image is baked and deployed, no developer or system administrator is allowed to make modifications to the servers. When promoting between environments, the entire container or image is started with environment-specific variables that are passed to the server when the server is first started.*

不可变的服务器：当一个镜像被制作和部署时，开发者和系统管理员都不允许对服务器进行修改。在环境间进行升级时，整个容器或镜像都是使用服务器第一次启动时传递到服务器的特定于环境的变量启动的。

*⑥ Phoenix servers: Because the actual servers are constantly being torn down as part of the continous integration process, new servers are being started and torn down. This greatly decreases the change of configuration drift between environments.*

凤凰服务器：因为真实的服务器经常被作为连续集成过程的一部分被卸载，新服务器正在启动就被卸载。这大大降低了配置漂移之间的环境变化。

## 1.10. 使用 Spring Cloud 构建微服务

在这一部分中，我简单介绍 Spring Cloud 技术，你将使用它创建你的微服务。这是一个高层次的概述；在这本书里，当你使用各种技术，在你需要的时候我会教你每个细节。

从零开始实现所有这些模式将是一份惊人的工作量。幸运的是，Spring 团队集成了大量进过充分测试的开源项目为 Spring 子项目，统称为 Spring Cloud。(http://projects.spring.io/spring-cloud/)。

Spring Cloud 把开源公司 如 Pivotal ,HashiCorp 和 Netflix 交付的产品整合在一起。Spring Cloud 简化了设置和配置这些项目到你的 Spring 应用程序，使你可以专注于写代码，没有被掩埋在可以构建和部署一个微服务应用的所有基础设施的配置细节里。

图 1.14 将上一节列出的模式映射到实现它们的 SpringCloud 项目。

让我们更详细地浏览这些技术。

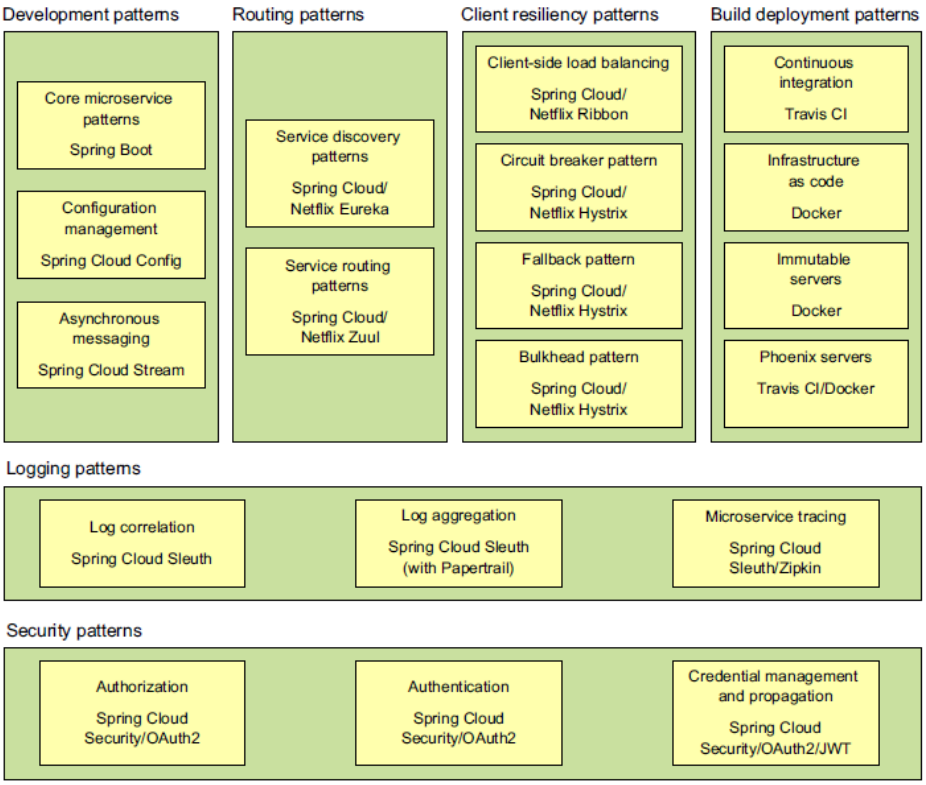


图 1.14 你可以绘制一张技术图谱，是我们迄今为止在本章中已经探讨的，正好打算要使用

的微服务模式。

### 1.10.1. Spring Boot

Spring Boot是用于我们微服务实现的核心技术。Spring Boot通过简化构建基于REST微服务的核心任务，极大的简化了微服务开发。Spring Boot 也极大的简化了 HTTP 方式谓词(GET, PUT, POST 和 DELETE)与 URLs 的映射，JSON 与 Java 对象互相转换的序列化协议，以及 java 异常返回与标准的 HTTP 错误代码的映射。

### 1.10.2. Spring Cloud Config

Spring Cloud Config 处理应用程序的配置数据的管理，通过一个集中的服务使你的应用程序配置数据（尤其是你的环境的具体配置数据）从你的部署微服务分离干净。这确保了无论你扩展多少微服务实例，它们总是会有相同的配置。Spring Cloud Config 有自己的属性管理仓库，还集成了开源项目，如以下：

- Git：Git（https://git-scm.com/）是一个开放源码的版本控制系统，可以管理和跟踪的任何类型的文本文件的变化。Spring Cloud Config 可以与 Git 支持的存储库集成，并从存储库中读取应用程序的配置数据。
- Consul：Consul（https://www.consul.io/）是一个开放源代码的服务发现工具，允许服务实例使用该服务注册自己。服务客户端可以向 Consul 询问服务实例位于何处。Consul 还包括关键的基于键值的数据库，可以使用 Spring Cloud Config 来存储应用程序配置数据。
- Eureka：Eureka（https://github.com/netflix/eureka）是一个开放源码的 Netflix 项目，如 Consul，提供类似的服务发现能力。Eureka 也有一个键值数据库，可以

与 Spring Cloud Config 一起使用。

### 1.10.3. Spring Cloud Service Discovery

通过 Spring Cloud 服务发现，你可以将消费服务的客户端从服务器部署的物理位置（IP 和/或服务器名称）抽离出来。服务消费者通过逻辑名称而不是物理位置调用服务器的业务逻辑。Spring Cloud 服务发现还负责注册和注销服务实例，当服务实例启动和停止的时候。Spring Cloud 服务发现可以使用 Consul（https://www.consul.io/）和 Eureka（https://github.com/netflix/eureka）作为服务发现引擎。

### 1.10.4. Spring Cloud/Netflix Hystrix and Ribbon

Spring Cloud 在很大程度上集成了 Netflix 开源项目。为微服务客户端弹性模式，Spring Cloud 将 Netflix Hystrix(https://github.com/Netflix/Hystrix)库和 Ribbon 项目(https://github.com/Netflix/Ribbon)集成在一起，在你的微服务实施中使用它们。

使用 Netflix Hystrix 库，你可以快速实现服务客户端的弹性模式，如断路器和舱壁模式。

虽然 Netflix Ribbon 项目简化了与服务发现代理（如 Eureka）的集成，它还提供了来自服务消费者的服务调用的客户端负载均衡。这使得即使服务发现代理暂时不可用，客户端也可以继续进行服务调用。

### 1.10.5. Spring Cloud/Netflix Zuul

Spring Cloud 使用 Netflix Zuul 项目（https://github.com/Netflix/zuul)为你的微服务应用提供服务路由能力。Zuul 是服务网关，它代理服务请求并确保在目标服务被调用之前，所有对你微服务的调用都通过一个单一的“前门”。有了这种集中的服务调用，你可以

执行标准的服务策略，如安全授权验证、内容过滤和路由规则。

### 1.10.6. Spring Cloud Stream

Spring Cloud Stream（https://cloud.spring.io/spring-cloud-stream/）是一门有利的技术，你可以很容易将轻量级消息处理整合到你的微服务。使用 Spring Cloud Stream，你可以构建智能的微服务，它可以使用在你的应用程序发生的异步事件。在 Spring Cloud Stream，你能快速将你的微服务与消息中间件集成，如：RabbitMQ (https://www.rabbitmq.com/) 和 Kafka (http://kafka.apache.org/)。

### 1.10.7. Spring Cloud Sleuth

Spring Cloud Sleuth（https://cloud.spring.io/spring-cloud-sleuth/）允许你将唯一的跟踪标识符集成到 HTTP 调用和在你的应用程序使用的消息通道（RabbitMQ，ApacheKafka）。这些跟踪号，有时称为关联或跟踪 ID，允许你跟踪一个交易，因为它在应用程序中的不同服务之间传递。在 Spring Cloud Sleuth，这些跟踪 ID 被自动添加到你在微服务中产生的任意日志语句。

Spring Cloud Sleuth 真正的优势是与日志聚合技术工具，如 Papertrail (http://papertrailapp.com)和跟踪工具如 Zipkin (http://zipkin.io)结合。Papertail 是一个基于云计算的日志平台，用于将不同微服务实时日志汇总到一个可查询的数据库。开源 Zipkin 使用 Spring Cloud Sleuth 产生的数据，允许你可视化一个单一交易的有关联的服务调用流程。

### 1.10.8. Spring Cloud Security

Spring Cloud Security（https://cloud.spring.io/spring-cloud-security/）是一个身

份验证和授权框架，它能控制谁可以访问你的服务和它们能为你的服务做些什么？Spring Cloud Security 是基于令牌的，它允许服务通过一个由身份验证服务器发出的令牌互相通信。每个服务接收到一个调用，可以检查在 HTTP 调用中提供的令牌，来验证用户的身份和它们访问服务的权限。另外，Spring Cloud Security 支持 JavaScript Web Token(https://jwt.io)。JavaScript Web Token (JWT)框架规范 OAuth2 令牌格式如何被创建和为数据签名创建的令牌提供标准。

#### 1.10.9.准备些什么？

为准备实现，我们要进行技术改造。Spring 框架是面向应用的开发。Spring 框架（包括 Spring Cloud）没有创建“构建和部署”管道的工具。实现“构建和部署“管道要使用下列工具：Travis CI（https://travis-ci.org），构建工具和 Docker（https://www.docker.com/），创建包含你的微服务、最终的服务器镜像。部署你构建的 Docker 容器，我们将使用一个如何在 Amazon 云部署贯穿本书的整个应用程序堆栈示例结束这本书。

### 1.11. 使用 Spring Cloud 举例

在最后一节中，我们讨论了所有不同的 Spring Cloud 技术，当你创建你的微服务的时候你将使用到它。因为每一种技术都是独立的服务，显然需要一个以上的章节来详细解释所有这些技术。然而，当我结束这一章，我想给你们一个小例子，再次证明将这些技术整合到自己的微服务开发工作是多么容易。

与清单 1.1 中的第一个代码示例不同，你不能运行此代码示例，因为需要设置和配置一些支持服务。不过别担心；这些 Spring Cloud 服务的安装成本（配置服务，服务发现）在

构建服务条款是一次性的成本。一旦它们安装，你个人的微服务可反复使用这些能力。我们不能满足所有美好的期望，在书的开始的一个单一的代码示例。

下面的清单中的代码很快演示了如何将服务发现、断路器、舱壁和远程服务的客户端负载均衡集成到我们的“Hello World”示例中。

**清单 1.2 Hello World Service using Spring Cloud**

```
package com.thoughtmechanix.simpleservice;

//为了简洁性，删除其他导入

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixProperty;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;

@SpringBootApplication
@RestController
@RequestMapping(value="hello")
@EnableCircuitBreaker <-- ①使服务能够使用 Hystrix 和 Ribbon 库
@EnableEurekaClient
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @HystrixCommand(threadPoolKey = "helloThreadPool") <-- ③使用一个 Hystrix 断路器包装对 helloRemoteServiceCall 方法的调用
    public String helloRemoteServiceCall(String firstName,
                                         String lastName){
        ResponseEntity<String> restExchange =
            restTemplate.exchange(
                "http://logical-service-id/name/
                [ca]{firstName}/{lastName}",
                HttpMethod.GET,
                null, String.class, firstName, lastName);
        return restExchange.getBody();
    }

    @RequestMapping(value="/{firstName}/{lastName}",
            method = RequestMethod.GET)
        public String hello( @PathVariable("firstName") String firstName,
                             @PathVariable("lastName") String lastName) {
```

②告诉服务，它应该注册一个 Eureka 服务发现代理，服务调用是使用服务发现来“查找”远程服务的位置。

④使用装饰类 RestTemplate 以一个“逻辑”的服务 ID，在 Eureka 范围里查找服务的物理位置。

```
            return helloRemoteServiceCall(firstName, lastName)
        }
}
```

此代码挤满了很多事情，所以让我们越过它。记住，这个清单只是一个例子，不包含在第 1 章 GitHub 库的源代码中。我在这里讨论它，是给你一个在本书以后会发生什么事的体验。

你应该注意的第一件事是：@EnableCircuitBreaker 和@EnableEurekaClient 注解。@EnableCircuitBreaker 注解告诉你 Spring 微服务，将在你的应用中使用 Netflix 的 Hystrix 库。@EnableEurekaClient 注解告诉你的微服务使用 Eureka 服务发现代理注册其本身，你要使用服务发现查找在你的代码里的远程 REST 服务端点。请注意，配置正在发生在一个属性文件中，它将告诉简单的服务要联系的 Eureka 服务器的位置和端口号。当你声明你的 hello 方法时，你将第一次看到 Hystrix 被使用。

```
@HystrixCommand(threadPoolKey = "helloThreadPool")
public String helloRemoteServiceCall(String firstName, String lastName)
```

@HystrixCommand 注解将做两件事情。首先，任何时候 helloremoteservicecall 方法被调用时，它不会被直接调用。相反，该方法将被委派到被 Hystrix 管理的一个线程池。如果调用时间太长（默认是 1 秒），将进入 Hystrix 并中断调用。这就是断路器模式的实现。其次，该注释的作用是创建一个称为 helloThreadPool 的线程池，它由 Hystrix 管理。所有到 helloRemoteServiceCall 方法的调用只会发生在这个线程池，并且由任何其他远程服务发起的调用将被隔离。

最后要注意的一件事是在 helloRemoteServiceCall 方法内发生了什么。@EnableEurekaClient 的存在告诉 Spring Boot，每当你做出一个 REST 服务调用，你要使用一个修改的 RestTemplate 类（这不是标准的 Spring RestTemplate 开箱即用）。RestTemplate 类将允许你为你试图调用的服务引入一个逻辑服务 ID：

```
ResponseEntity<String> restExchange = restTemplate.exchange
```

(http://logical-service-id/name/{firstName}/{lastName}

正如所论述的，RestTemplate 类将与 Eureka 服务联系，并查找一个或多个“名称”服务实例的物理位置。作为服务的消费者，你的代码永远不必知道该服务位于何处。

另外，RestTemplate 类使用 Netflix 的 Ribbon 库。Ribbon 将取回与服务相关联的所有物理端点的列表。每次服务被客户端调用时，它对客户端不同服务实例采用“round-robins”调用，而不必经过过一个集中的负载均衡器。通过消除集中式负载均衡器并将其移动到客户端，你将在应用程序基础设施中消除另一个故障点（负载均衡器停止）。我希望在这一点上你留下深刻的印象，因为你已经添加了相当多的微服务能力，而你只用很少的注释。

这才是 Spring Cloud 的真正优势。你作为一个开发者，从最先的云服务提供商获得身经百战的微服务能力，如 Netflix 和 Consul。这些功能，如果在 Spring Cloud 之外使用，可能会很复杂，很难建立。Spring Cloud 简化了它们的使用，只不过是一些简单的 Spring Cloud 注释和配置条目而已。

## 1.12. 了解与我们相关的例子

我想确保这本书提供的例子，可以与你的日常工作联系起来。为此，我制定这本书的章节结构及相应的代码示例，围绕一家称为 ThoughtMechanix 的虚构公司的奇遇（不幸的冒险）。

ThoughtMechanix 是一家软件开发公司，他的核心产品 EagleEye，提供一个企业级的软件软件资产管理应用。它提供了所有关键元素的覆盖：库存、软件交付、许可证管理、合规性、成本和资源管理。它的主要目标是使组织能够准确地了解其软件资产的时间点。

这家公司大约有 10 年历史。虽然他们已经经历了可靠的收入增长，它们内部正在讨论是否应重新以整体论为前提基础的应用构建平台核心产品或迁移到云应用。对于一家公司，

涉及 EagleEye 的平台重新调整将是一个影响“成败”的关键时刻。

该公司是在一个新的架构重建其核心产品 EagleEye。而对于应用程序的业务逻辑大多会保留，应用本身将被分解，从整体设计拆分成更小的微服务设计，它们每一部分都可以独立部署到云。在这本书中的例子将不会建立整个 ThoughtMechanix 应用。相反，你会从手头的问题域构建具体的微服务，并建立基础设施来支持这些服务使用各种各样的 Spring Cloud（和一些 Spring Cloud）技术。

成功地采用基于云的能力，微服务架构将影响技术组织的所有部分。这包括架构、工程、测试和运营团队。每一组的投入都是必要的，在最后，他们可能需要重组团队来重新评估在这个新的环境他们的责任。当你开始识别并建立了几个用于 EagleEye 的微服务的基础工作，让我们开始我们与 ThoughtMechanix 的旅程。这些微服务使用 Spring Boot 来构建。

## 1.13. 小结

- 微服务是负责一个特定区域的功能，范围非常小的块。
- 对于微服务没有行业标准。不同于其他早期的 Web 服务协议，微服务采取原则为基础的方法和结合 REST 和 JSON 的概念。
- 写微服务是容易的，但在生产完全投入运行它们，还需要更多的考虑。我们介绍了微服务开发模式的几个类别，包括核心开发，路由模式，客户端弹性，安全，日志，和建立/部署模式。
- 然而微服务是语言无关的，我们介绍了两种有助于建设微服务的 Spring 框架：Spring Boot 和 Spring Cloud。
- Spring Boot 被用来简化构建基于 REST/JSON 的微服务。它的目标是使你只不过使用很少的注解就能快速建立微服务。

- Spring Cloud 是一个来自开源技术公司的工具集，如 Netflix 和 HashiCorp，他们已经用 Spring 注解包装，大大简化这些服务的安装和配置。