

## 2. 第 2 章 使用 Spring Boot 构建微服务

### 本章内容

- 学习微服务关键特性
- 了解微服务如何适合云架构
- 分解一个业务领域成一组微服务
- 使用 Spring Boot 实现一个简单的微服务
- 理解构建基于微服务应用的观点
- 学习什么时候不使用微服务

软件开发的历史上充斥着大量开发项目的故事,这些项目是在投入了数百万美元和数十万软件开发人员的时间之后进行的,和许多业内最优秀和最聪明的人一起工作,不知何故,他们从来没有设法向顾客提供任何有价值的东西,而且在他们自己的复杂性和重任下崩溃了。

这些庞大的项目往往遵循大型传统的瀑布开发方法,坚持在项目开始时定义所有应用程序的需求和设计。太多的重点放在了让所有规格的软件“正确”的,没有什么余地来满足新的业务需求,或重构和学习在发展的初期阶段犯过的错误。

然而,实际情况是软件开发不是定义和执行的线性过程,但这是一个渐进的过程,在开发团队真正了解手头的问题之前,需要经过几次反复的交流、学习和交付给客户。

结合使用传统瀑布方法的挑战是很多次在这些项目中交付的软件产品的粒度是:

- 紧耦合:这大大增加了即使应用程序组件进行小的更改也可能破坏应用程序的其他部分并引入新 bug 的机会。
- 泄漏:大多数大型软件应用程序管理不同类型的数据。例如,客户关系管理(CRM)应用程序可以管理客户、销售和产品信息。在传统模型中,这些数据保存在同一个

数据模型中，并且存储在同一个数据存储中。尽管数据之间有明显的界限，但通常来自一个域的团队很容易直接访问属于另一个团队的数据。

这种易于访问的数据创建了隐藏的依赖关系，并允许一个组件的内部数据结构的实现细节泄漏到整个应用程序中。即使对单个数据库表进行小的更改也可能需要在整个应用程序中有大量代码更改和回归测试。

- 单体/庞大：因为一个传统的应用程序的大多数应用程序组件存在于一个单一的代码库，在多个团队之间分享，任何时间更改代码，整个程序必须重新编译，通过整个测试周期重新运行，并重新部署。即使是对应用程序代码库的小改动，不管是新的客户需求还是 bug 修复，都会变得昂贵和耗时，而且大的更改几乎是不可能及时完成的。

一个微服务架构采用了不同的方法来提供的功能。具体来说，微服务架构有以下特性：

- 有限的：微服务单一职责、范围有限。微服务拥抱 Unix 哲学：一个应用只不过是一个服务集合，每个服务做一件事，并且一件事情做得很好。
- 松耦合：一个微服务应用就是一个小服务的集合，仅仅通过使用非专有的调用协议（例如，HTTP 和 REST），不执行特定的接口相互作用。只要服务不改变接口，微服务的拥有者有比在传统的应用架构更多的自由来对服务进行修改。
- 分离的：微服务完全拥有自己的数据结构和数据来源。由微服务数据只能通过服务修改。对数据库的访问控制使微服务的数据被锁定，仅允许这个服务访问它。
- 独立的：在微服务应用程序中的每个微服务，可以独立于应用程序中的其他服务被编译和部署。这意味着与更为相互依赖的整体应用程序相比，更改可以更容易地隔离和测试。

为什么这些微服务架构属性对基于云计算的开发是重要的？基于云的应用程序一般有

以下几个特点：

- 庞大多样的用户群：不同的客户需要不同的特性，并且在开始使用这些特性之前，他们不需要等待长的应用程序发布周期。微服务允许功能很快交付，因为每个服务是小范围的，通过定义明确的接口访问。
- 极高的正常运行时间要求：因为微服务的分散性，微服务应用可以更容易地隔离应用程序的特定部分的故障和问题，而不必删除整个应用程序。这降低了整体应用的停机时间，使它们提升了抵御故障的能力。
- 不均匀的大量需求：随着时间的推移出现，在四面都是墙的企业数据中心部署的传统应用程序通常具有一致的使用模式。这使得这些类型的应用程序的容量规划变得简单。但在一个基于云的应用程序，在 Twitter 的一个简单鸣叫或者是在 Slashdot 上张贴消息，都可以从最高限度驱动云应用需求。

因为微服务应用程序被分解成小的组件，可以独立于其它的组件被部署，将重点放在组件的负载和在云中多台服务器上横向伸缩要容易得多。

本章为您提供了解实现目标所需的基础，在你的业务问题中识别微服务，建立一个微服务骨架，然后理解微服务在生产上被成功部署和管理所需的操作属性。

为了成功地设计和构建微服务，你需要像一个警察询问犯罪嫌疑人一样着手处理微服务。尽管每个证人都目睹了同样的事件发生，但他们对犯罪的解释是由他们的背景、对他们很重要的东西（例如，是什么诱发他们）形成的，以及在目睹事件发生的那一刻，施加了什么环境压力。参与者对自己认为重要的事物都有自己的观点和偏见。

像一个成功的警察试图获得真相，要构建一个成功的微服务架构之旅，包括你的软件开发组织多个人的观点。虽然交付一个完整的应用程序需要更多的技术人员，但我相信成功的微服务开发基础从三个关键角色的视角开始：

- 架构师：架构师的工作就是从顶层视图，了解应用程序如何可以被分解成单独的微服务和微服务如何互动，并提供一个解决方案。
- 软件开发人员：软件开发人员编写代码和详细理解用于提供微服务的开发语言和开发框架。
- 运维工程师：运维工程师带来如何部署和管理不仅仅生产，而且所有的非生产环境的智慧能力。运维工程师的目标是在每一个环境的一致性和可重复性。

在这一章中，我将演示如何从每一个角色的角度使用 Spring Boot 和 Java 设计和构建一套微服务。到本章结束时，您将有一个可以打包并部署到云上的服务。

## 2.1. 架构师的故事：设计微服务架构

架构师在软件项目中的角色是提供需要解决的问题的工作模型。架构师的工作是提供开发人员将构建代码的脚手架，以便应用程序的所有组件能够组合在一起。

在建立一个微服务架构的时候，一个项目的架构聚焦在三个关键任务：

- 分解业务问题
- 确定服务粒度
- 定义服务接口

### 2.1.1. 分解业务问题

面对复杂性，大多数人试图分解他们正在处理的问题。他们这样做，这样他们就不必试图把问题的所有细节都放在脑子里。相反，他们将问题抽象地分解成几个关键部分，然后寻找这些部分之间存在的关系。

在微服务架构，架构师分解业务问题成块，它表示活动离散域。这些块封装了业务规则

和与业务域特定部分相关联的数据逻辑。

虽然你想微服务封装进行一个交易的所有业务规则,但这并不总是可行的。你经常会遇到这种情况,你需要有一组微服务跨业务领域的不同的部分来完成整个交易。一个架构师梳理一套微服务的服务界限,看看哪里数据域不适合在一起。

例如,架构师可能会查看一个由代码执行的业务流,并意识到它们需要客户和产品信息。两个离散数据域的存在是一个很好的迹象,即多微服务在起作用。业务交易的两个不同部分如何相互作用通常成为微服务的服务接口。

分解业务领域是一种艺术形式,而不是一门黑白科学。使用下列准则确定和分解业务问题成为微服务候选者:

- **描述业务问题,依你用的名词来描述问题。**在描述问题时反复使用相同的名词,通常是一个核心业务领域和一个微服务契机的象征。从第 1 章的 EagleEye 域目标名词的例子,看起来如:合同、许可证和资产。
- **注意动词。**动词突出动作,通常表示问题域的自然轮廓。如果你发现自己在说“交易 X 需要从 A 和 B 类获得数据”,那通常意味着有多个服务在起作用。如果你运用 EagleEye 看动词的方法,你可能会寻找这样的语句,“当迈克使用桌面服务正在安装一台新电脑,他查看了可供软件 X 的可用许可证数量,如果许可证可用,安装软件。然后,他更新跟踪电子表格中已使用的许可证数量。”这里的关键动词是 looks 和 updates。
- **寻找数据内聚性。**当你把你的业务问题分解成离散的部分时,寻找那些彼此高度相关的数据块。如果突然间,在你人机对话的过程中,你正在阅读或更新与你之前讨论过的根本不同的数据,你可能会有另一个候选服务。微服务应该完全拥有自己的数据。

让我们把这些指导原则应用到现实世界的问题中去。第 1 章介绍了现有的软件产品称为 EagleEye，它用于管理软件资产，如软件许可证和安全套接字层（SSL）证书。这些项目部署到整个组织的各个服务器中。

EagleEye 是一个传统的单体 Web 应用程序，它被部署到客户数据中心的 J2EE 应用服务器。您的目标是将现有的单体应用程序分解成一组服务。

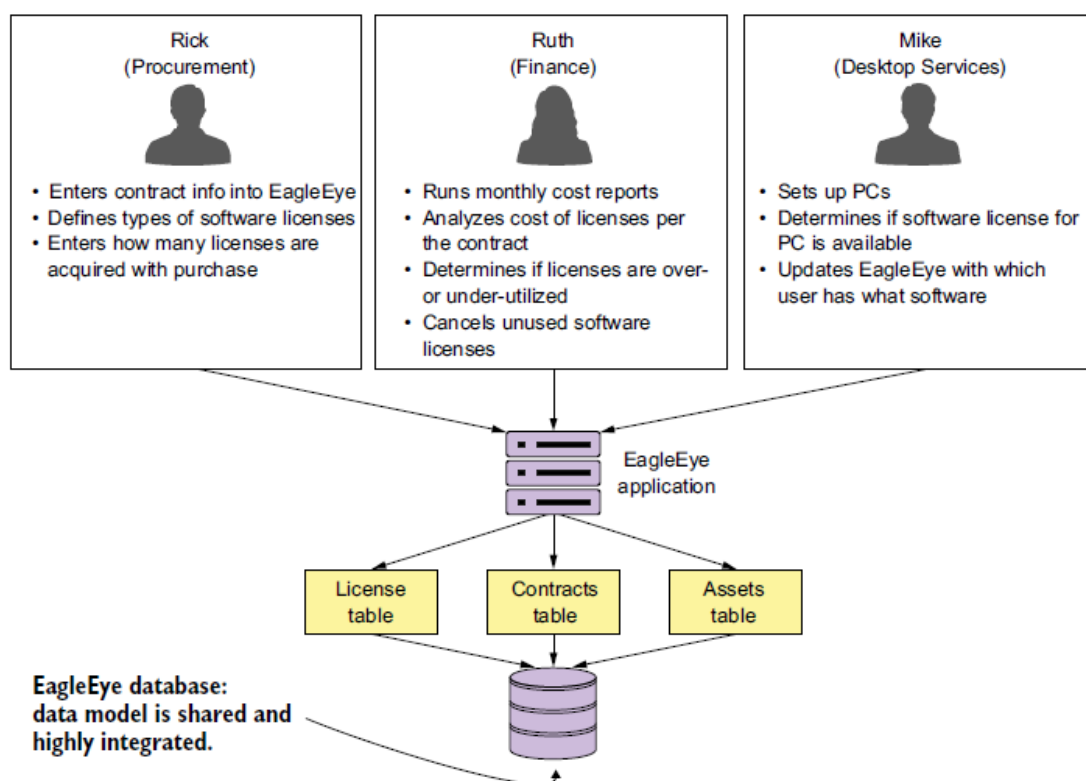


图 2.1 采访 EagleEye 的用户，并了解他们的日常工作。

① *EagleEye database: data model is shared and highly integrated.*

EagleEye 数据库：数据模型是共享和高度集成的。

你要开始与 EagleEye 应用程序的所有用户面谈和与他们讨论他们如何使用 EagleEye。

图 2.1 获取了你可能与不同业务客户进行的谈话摘要。通过看 EagleEye 的用户如何与应用程序交互和应用程序的数据模型如何被分解，你可以分解 EagleEye 的问题域为以下候选微服务。

在图中，我突出显示了与业务用户对话期间出现的一些名词和动词。因为这是一个已有的应用程序，您可以查看应用程序并将主要的名词映射到物理数据模型中的表。现有的应用程序可能有数百个表，但是每个表通常会映射到一组逻辑实体。

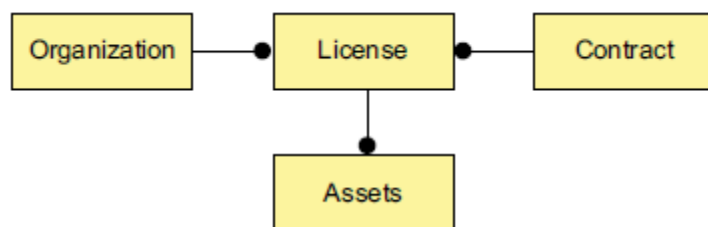


Figure 2.2 A simplified EagleEye data model

图 2.2 简化的 EagleEye 数据模型

图 2.2 显示了一个在与 EagleEye 客户交谈的基础上简化的数据模型。基于企业访谈和数据模型，候选微服务是组织、许可证、合同、资产服务。

### 2.1.2. 确定服务粒度

一旦你有了一个简化的数据模型，你可以开始定义过程，你将在应用中需要什么微服务。

根据图 2.2 中的数据模型，你基于以下要素可以看到潜在四个微服务：

- 资产
- 许可证
- 合同
- 组织

我们的目标是把这些主要功能部分提取出来，并将它们提取成完全独立的单元，这些单元可以独立地构建和部署。但从数据模型中提取服务不仅仅是重新包装成单独的项目代码。

它还涉及到梳理服务访问的实际数据库表，只允许每个单独的服务访问特定域中的表。图

2.3 显示了应用程序代码和数据模型如何将“块”变成独立的组件。

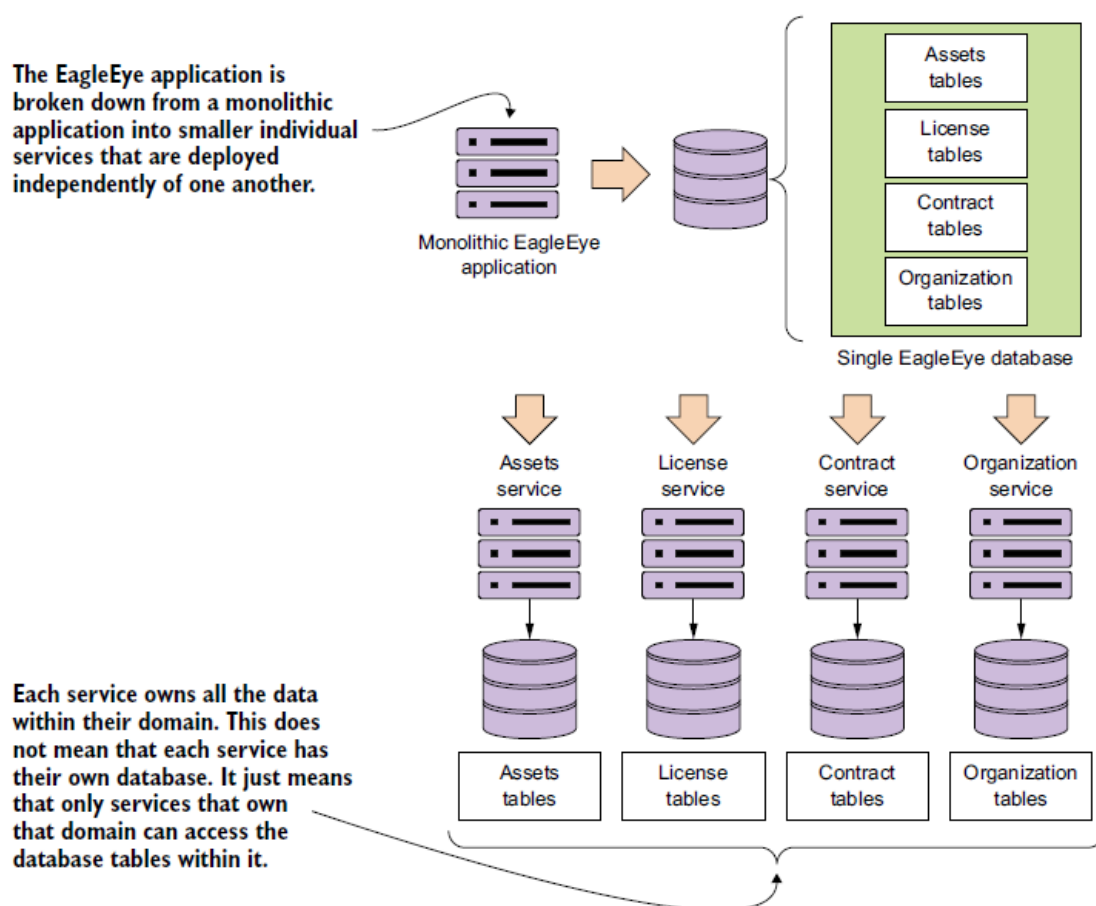


图 2.3 使用数据模型作为将单体应用分解到微服务的基础。

### ① Monolithic EagleEye application

单体 EagleEye 应用

### ② Single EagleEye database

单一的 EagleEye 数据库

③ The EagleEye application is broken down from a monolithic application into smaller individual services that are deployed independently of one another.

EagleEye 应用从单体应用中被分解成更小的个体服务，它们彼此独立部署。

④ Each service owns all the data within their domain. This does not mean that each service has their own database. It just means that only services that own that domain can access the database tables within it.



每个服务都拥有在其域内的所有数据。这并不意味着每个服务都有自己的数据库。它只是意味着只有拥有该域的服务才能访问它内部的数据库表。

在将一个问题域分解为离散块之后,您常常发现自己在努力确定是否已经为服务实现了正确的粒度级别。我们不久将讨论,一个微服务太粗或细会有一些明显的特征。

当你建立一个微服务架构、粒度的问题是重要的,但你可以用以下的概念来确定正确的解决方案:

- **更好开始广泛的使用微服务和重构为更小的服务。** 很容易走极端,当你开始你的微服务旅程,做一切事情都使用微服务。但分解问题域为小服务,往往导致过早的复杂性,因为微服务仅仅变成细粒度的数据服务。
- **首先关注您的服务如何与其它服务交互。** 这将有助于建立问题域的粗粒度接口。这使从太粗太细重构变得更容易。
- **随着您对问题域的理解不断增长,服务职责将随时间而变化。** 通常,当新的应用功能被要求的时候,一个微服务增加职责。一开始,一个单一的微服务可能成长为多个服务,原始的微服务作为对这些新的服务业务流程层和封装来自应用程序其它部分的功能。

### 有害微服务的气味

你如何知道你的微服务是否是合适的大小?如果一个微服务太粗,你可能会看到如下:

**服务太多的职责:** 服务中业务逻辑的一般流程是复杂的,并且似乎在执行一系列过于多样化的业务规则。

**服务通过大量的表管理数据:** 一个微服务是其管理数据的记录系统。如果你发现将数据持久保存到多个表,或者直接访问外部数据库的表,这说明服务太大了。我喜欢使用指南,微服务应该拥有不超过三至五表。而且,你的服务可能有太多的职责。

**测试用例太多**：随着时间的推移，服务的规模和职责也会增加。如果你有一个服务，它们开始于小数量小的测试用例，结束于成百上千的单元和集成测试用例，你可能需要重构。

一个微服务粒度太细会怎么样？

**在问题域的一部分微服务像兔子一样繁殖**：如果一切都变成一个微服务，构成业务逻辑的服务变得复杂和困难，因为要得到一块工作服务的数量的难度与日俱增。一个共同的气味，是你在应用程序中有很多的微服务和每一个服务只与一个单一的数据库表交互。

**微服务彼此严重依赖**：你发现的问题域的一部分微服务保持相互之间来回调用来完成一个用户请求。

**微服务成为一个简单的 CRUD（创建、替换、更新、删除）服务集**：微服务是一种业务逻辑表述，并不是你数据源的一个抽象层。如果微服务除了 CRUD 相关的逻辑，什么也不做，他们可能太细。

微服务架构将用进化的思维过程发展，你知道你不会在第一次就得到正确的设计。这就是为什么从第一组服务开始，粗粒度比细粒度更好。不要对自己的设计过于武断也是很重要的。你需要做一个将数据聚合在一起服务，可能在该服务上会遇到物理约束，因为两个单独的服务太多了，或者在服务域之间没有明确的界限。

最后，与其浪费时间试图使设计完美，然后在你的努力面前没有成绩可言，倒不如采取一种务实的态度交付。

### 2.1.3. 互相交谈：服务接口

架构投入的最后一部分，要谈一谈关于如何在您的应用程序定义微服务。使用微服务构建业务逻辑，服务接口应该是直观的，开发者通过学习应用中的一个或两个服务，将获得运行在应用中所有服务的规律。

在一般情况下，下列准则可作为服务接口的设计思想：

- **拥抱 REST 哲学**：服务的 REST 方法本质上是把 HTTP 作为服务的调用协议和使用标准 HTTP 动词(GET, PUT, POST, 和 DELETE)。围绕这些 HTTP 动词对你的基本行为建模。
- **使用 URI 来传达意图**：作为服务端点的 URI 应该描述您的问题域中的不同资源，并为您的问题域中的资源关系提供基本机制。
- **为请求和响应使用 JSON**：JavaScript Object Notation（换句话说，JSON）是一种非常轻量级的数据序列化协议，并且比 XML 更容易消费。
- **使用 HTTP 状态代码来传递结果**：HTTP 协议有大量的标准响应代码来表明服务的成功或失败。学习这些状态代码，最重要的是在所有的服务中始终如一地使用它们。

所有的基本原则都驱动着一件事，使您的服务接口易于理解和可消费。您希望开发人员坐下来查看服务接口并开始使用它们。如果一个微服务不易消费，开发人员将以自己的工作，颠覆架构的意图。

## 2.2. 什么时候不使用微服务

我们在本章已经讨论为什么微服务是构建应用程序的一个强大的架构模式。但我还没提及到，什么时候你不应该使用微服务来构建你的应用程序。来，让我们一起讨论它们：

- 分布式系统构建的复杂性
- 虚拟服务器/容器扩展
- 应用类型
- 数据转换和一致性

### 2.2.1. 分布式系统构建的复杂性

因为微服务是分布式的、细粒度的（小），他们将复杂程度引入到你的应用程序，就不会有更多的单体应用。微服务架构需要高度成熟的运维。不考虑使用微服务，除非你的组织愿意投资于自动化和高度分布式的应用能够取得成功的运维工作（监控、伸缩）。

### 2.2.2. 服务器扩展

微服务最常见的部署模型之一是将一个微服务实例部署在一台服务器上。在一个以大的微服务为基础的应用程序，你可能会有 50 到 100 的服务器或容器（通常是虚拟的），都必须在生产中被单独创建和维护。即使在云中运行这些服务的成本较低，管理和监视这些服务器的操作复杂性也可能是巨大的。

**注意：**微服务的灵活性需要针对运行所有这些服务器的成本进行权衡。

### 2.2.3. 应用类型

微服务是面向可复用，对构建需要高弹性和可扩展的大型应用程序非常有用。这就是为什么如此多的基于云计算的公司已经采用了微服务。如果你构建小型的，部门级的应用程序或一个小的用户群的应用程序，将它们构建在分布式模型（如微服务）的相关复杂性可能会导致花费更多的费用。

### 2.2.4. 数据转换和一致性

当你开始了解微服务，你需要思考你的服务的数据使用模式和服务消费者如何去使用它们。一个微服务包装和抽象了一小部分的表和作为执行“操作”任务的机制工作良好，如创建，添加，并进行简单的（非复杂）对存储查询。

如果你的应用程序需要在多个数据源做复杂的数据聚合或改造，微服务的分布式特性将使这项工作变得困难。你的微服务总是承担了太多职责，也容易成为性能问题。

也请记住，通过微服务执行事务不存在标准。如果您需要事务管理，则需要自己构建该逻辑。此外，你会在第 7 章看到，微服务通过使用消息传递可以在它们之间相互通信。后续，在数据更新中引入消息传递。应用程序需要处理最终一致性，而对于数据的更新可能不会立即出现。

### 2.3. 开发：使用 Spring Boot 和 Java 构建一个微服务

构建微服务时，从概念空间转变为实现空间需要换个角度思考。具体来说，作为一个开发者，你需要建立一个基本的模式，每个应用程序中的微服务将被如何实现。虽然每个服务将是独一无二的，你要确保你使用一个删除样板代码的框架，微服务的每一个部分都以一致的方法被规划。

在本节中，我们将探讨从你的 EagleEye 域模型创建授权微服务，是开发者优先考虑的事。您的许可服务将使用 Spring Boot 编写。Spring Boot 是标准 Spring 库上的抽象层，它允许开发者快速构建基于 Groovy 和 java 的 web 应用程序，微服务比成熟的 Spring 应用程序使用更少的配置。

以您的许可服务的例子，你可以使用 java 作为核心编程语言和 Apache Maven 作为构建工具。

在接下来的几个部分中，你将要：

- 创建微服务的基本骨架和构建该应用程序的 Maven 脚本
- 实现一个 Spring 引导类，它将为微服务启动 Spring 容器和开始剔除所有为类工作的初始化代码

- 实现一个 Spring Boot 控制器类，用于映射端点以暴露服务的端点

### 2.3.1. 从骨架项目快速入门

首先，您将为授权创建一个框架项目。你可以把源代码从

GitHub(<https://github.com/carnellj/spmia-chapter2>)拉下来或使用以下的目录结构创

建一个许可服务项目目录：

- licensing-service
- src/main/java/com/thoughtmechanix/licenses
- controllers
- model
- services
- resources

一旦你拉下来或创建目录结构，开始为项目写你的 Maven 脚本。pom.xml 文件将位于项

目目录的根目录。下面的清单显示许可服务的 maven pom 文件。

#### 清单 2.1 Maven pom file for the licensing service

```
<?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.thoughtmechanix</groupId>
  <artifactId>licensing-service</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>EagleEye Licensing Service</name>
  <description>Licensing Service</description>

  <parent>
```

```

<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>1.4.4.RELEASE</version>
<relativePath/>
</parent>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
</dependencies>

```

①告诉 Maven 包括 Spring Boot 启动工具包的依赖

②告诉 Maven 包括 Spring Boot web 依赖

③告诉 Maven 包括 Spring Actuator 依赖

<!--注意：一些构建属性和 Docker 构建插件不包括在这个 POM 的 pom.xml 文件（不在源代码在 GitHub 库）之内，因为它们与我们这里的讨论无关。-->

```

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
</project>

```

④告诉 Maven 包括 Spring 特有的 maven 构建和部署 Spring Boot 应用程序的插件

我们不会详细地讨论整个脚本，但是在开始时请注意几个关键的地方。Spring Boot 被分解成许多单独的项目。其原理是，如果你不打算在应用程序中使用不同的 Spring Boot，就不必“拉下世界”。这也允许各种 Spring Boot 项目独立地发布新版本的代码。为了帮助简化开发人员的工作，Spring Boot 团队已经将相关的项目集成到各种“starter”工具包中。在 Maven POM 文件的第 1 部分，你告诉 Maven 你需要拉下版本为 1.4.4 的 Spring Boot 框架。

在 Maven 文件的第 2 和第 3 部分，你确定你将拉下 Spring Web 和 Spring Actuator 的 starter 工具包。这两部分几乎是所有基于 Spring Boot REST 服务的核心。这两个项目

是几乎所有基于 Spring 的基于 REST 的服务的核心。你会发现，当你在服务中创建更多的功能时，这些依赖项目的列表就变得更长了。

另外，Spring 源代码提供了 Maven 插件，简化构建和部署 Spring Boot 应用程序。

第 4 步告诉你的 Maven 构建脚本安装最新的 Spring Boot Maven 插件。这个插件包含了一些附加的任务（如 `spring-boot:run`），简化了 Maven 和 Spring Boot 之间的操作。

最后，你会看到一个注释，Maven 文件的部分内容已被删除。为了简单起见，我在清单 2.1 中没有包括 Docker 插件。

**注意：**在这本书的每一章，都包括用于构建和部署应用程序作为 Docker 容器的 Docker 文件。你在每一章代码部分的 README.md 文件可以找到如何构建 Docker 镜像的详细说明。

### 2.3.2. 编写引导类,启动你的 Spring Boot 应用

你的目标是获得一个简单的微服务，它在 Spring Boot 运行良好，并且在其上可迭代的发布功能。为此，你需要在你的授权微服务项目中创建两个类：

- 一个 Spring 引导类，将用于 Spring Boot 启动并初始化应用程序
- 一个 Spring 控制器类，将暴露的 HTTP 端点，可以在微服务调用

你很快就会看到，Spring Boot 使用注解来简化服务的设置和配置。当您在下面的清单中查看引导类时，这就变得很明显了。这个引导类位于

`src/main/java/com/thoughtmechanix/licenses/Application.java` 文件。

#### 清单 2.2 介绍@SpringBootApplication 注解

```
package com.thoughtmechanix.licenses;
```

```
import org.springframework.boot.SpringApplication;
```

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

```
@SpringBootApplication
```

```
public class Application {
```

```
    public static void main(String[] args) {
```

①@SpringBootApplication 告诉 Spring Boot 框架，这是项目引导类



```

    SpringApplication.run(Application.class, args);
}
}

```

②调用以启动整个 Spring Boot 服务

第一件事是注意在这段代码中使用的 `@SpringBootApplication` 注解。Spring Boot 使用这个注释告诉 Spring 容器，这个类是在 Spring 中使用的 Bean 定义的源头。在 SpringBoot 应用程序中，您可以通过以下方式定义 Spring Bean：

- 在 Java 中使用 `@Component`, `@Service` 或 `@Repository` 注解标签注释。
- 用 `@Configuration` 注释一个类，然后为每一个你想用一个 `@Bean` 标签创建的 Spring Bean 定义一个构造函数方法。

正如所论述的，`@SpringBootApplication` 注解标记清单 2.2 中的应用程序的类作为一个配置类，然后开始自动扫描其它 Spring Bean 在 java 类路径的所有类。

要注意的第二点是应用程序类的 `main()` 方法。在 `main()` 方法，`SpringApplication.run(Application.class, args)`，调用启动 Spring 容器并返回一个 `Spring ApplicationContext` 对象。

最简单的事情是记住 `@SpringBootApplication` 注解和相应的应用程序类，它是整个微服务的引导类。服务的核心初始化逻辑应该放在这个类中。

### 2.3.3. 创建微服务的访问入口：Spring Boot 控制器

现在您已经获得了构建脚本，并实现了一个简单的 Spring Boot 引导类，您可以开始编写您的第一个代码，它将做一些事情。此代码将是控制器类。在一个 Spring Boot 应用，一个控制器类暴露服务的端点，从一个传入的 HTTP 请求映射到一个将处理请求数据的 Java 方法。

#### 尝试一下 REST

这本书中所有的微服务采用 REST 方法来构建你的微服务。对 REST 的深入讨论超出了本书

的范围，<sup>a</sup>但为了能够满足你的要求，你所构建的所有服务都具有以下特性：

**使用 HTTP 作为服务的调用协议：**该服务将通过 HTTP 端点暴露，并将使用 HTTP 协议将数据在服务间来回传送。

**将服务的行为映射到标准 HTTP 动词：**REST 强调服务将其行为映射到 HTTP 动词（POST, GET, PUT, 和 DELETE）中。这些动词在大多数服务中映射到增删改查功能。

**将 JSON 作为服务间传递的所有数据的序列化格式：**这不是一个基于 REST 的微服务的硬性原则，但 JSON 已经成为数据序列化的通用语，数据将通过微服务被提交和返回。可以使用 XML，但许多基于 REST 的应用程序大量使用 JavaScript 和 JSON（JavaScript Object Notation）。JSON 是序列化和反序列化基于 JavaScript 的 Web 前端和服务消费的数据的原生格式。

**使用 HTTP 状态代码来传递服务调用的状态：**HTTP 协议开发了一组丰富的状态代码来表明服务的成败。基于 REST 的服务利用这些 HTTP 状态码和其他基于网络的基础设施，如反向代理服务器和缓存，可以相对容易地与您的微服务集成。

HTTP 是 Web 的语言，使用 HTTP 作为构建服务的哲学框架是构建云服务的关键。

<sup>a</sup>可能最全面的讨论 REST 服务设计的书是 Ian Robinson 等人编写的《REST 实践》（O'Reilly，2010）

你的第一个控制器类位于 `src/main/java/com/thoughtmechanix/licenses/controllers/LicenseServiceController.java`。该类将暴露四个 HTTP 端点，这些端点将映射到动词 POST, GET, PUT, 和 DELETE。

让我们浏览一下控制器类，看看 Spring Boot 是如何提供一组注解的，这些注解可以不断努力将服务端点暴露到最低限度，并允许您集中精力构建服务的业务逻辑。我们将从基本控制器类定义开始，而不使用任何类方法。

下面的清单显示了为您的许可服务构建的控制器类。

### 清单 2.3 标记 LicenseServiceController 作为一个 Spring RestController

```
package com.thoughtmechanix.licenses.controllers;
```

```
@RestController
@RequestMapping(value="/v1/organizations/{organizationId}/licenses")
public class LicenseServiceController {

    //为了简洁而除去类的主体
}
```

①@RestController 告诉 Spring Boot，这是一个基于 REST 的服务，会自动序列化/反序列化服务请求/响应为 JSON。

②以前缀/v1/organizations/{organizationId}/licenses 的形式暴露该类中的所有 HTTP 端点

我们将从@RestController注解开始我们的探索。@RestController是一个类级的Java注解，它告诉 Spring 容器，这个 Java 类将被用于一个基于 REST 的服务。这个注释自动处理成了为 JSON 或 XML 的服务数据的序列化（默认情况下，@RestController 类将序列化返回的数据为 JSON）。不同于传统的 Spring @Controller 注解，@RestController 注解不需要你作为开发者从你的控制器类返回一个 ResponseBody 类。这是有 @RestController 注解存在的情况下所有的处理，它包括@ResponseBody 注解。

#### 为什么微服务使用 JSON ？

多协议可以用来发送基于 HTTP 的微服务来回之间的数据。由于几个原因，JSON 已经成为事实上的标准。

首先，与其他协议（如基于 XML 的 SOAP（简单对象访问协议））相比，它非常轻量级，您可以在没有大量文本开销的情况下传送数据。

其次，它很容易被人阅读和消费。这是一个选择序列化协议被低估的优势。当出现问题时，开发人员查看 JSON 块并快速、直观地处理其中的内容是至关重要的。协议的简单性使这项工作非常简单。

第三，JSON 是 JavaScript 中使用的默认序列化协议。由于 JavaScript 的关注度急剧上升，

作为一种编程语言和单页面的互联网应用的关注度同样急剧上升 ( SPJA ), 它们在很大程度上依赖于 JavaScript ,JSON 已经成为构建基于 REST 应用的一种天然契合 ,因为前端 Web 客户端使用它调用服务。

其他机制和协议比 JSON 更有效地用于服务间通信。Apache 的 Thrift

( <http://thrift.apache.org> ) 框架允许你使用二进制协议构建能够互相通信的多语言服务。

Apache 的 Avro(<http://avro.apache.org>)协议是一个数据序列化协议 ,它将客户端和服务端之间的来回传递的数据转换成二进制格式。

如果你需要最小化通过电缆发送的数据的大小 ,我建议你看看这些协议。但据我的经验 ,在你的微服务中直接使用 JSON 有效工作 ,不干预另一层服务消费者和服务客户端之间的通信调试。

清单 2.3 中显示的第二个注解是 @RequestMapping。你可以使用

@RequestMapping 作为类级或者方法级的注解。@RequestMapping 注解用来告诉 Spring 容器服务的 HTTP 端点将向外界暴露。当你使用类级的 @RequestMapping 注解 ,你将建立通过控制器暴露的所有其它端点的根 URL。

在清单 2.3 中 , @RequestMapping(value="/v1/organizations/{organizationId}/licenses")使用 value 属性创建控制器类中暴露的所有端点的根 URL。在这个控制器中暴露的所有服务端点都将以/v1/organizations/{organizationId}/licenses 开始 ,作为其端点的根。{organizationId}是一个占位符 ,表示你期待的 URL 在每次调用传递一个 organizationId 参数。对 URL 中的使用 organizationId ,允许你在不同客户之间谁会使用你的服务。

现在 ,你将向控制器添加第一个方法。此方法将实现 REST 调用中使用的 GET 谓词并返回单个许可证类实例 ,如下面的清单所示。( 讨论的目的是实例化一个称为 License 的 Java

类。)

#### 清单 2.4 暴露单个 GET HTTP 端点

```
@RequestMapping(value="/{licenseId}",method = RequestMethod.GET)
public License getLicenses(
    @PathVariable("organizationId") String organizationId,
    @PathVariable("licenseId") String licenseId) {
    return new License()
        .withId(licenseId)
        .withProductName("Teleco")
        .withLicenseType("Seat")
        .withOrganizationId("TestOrg");
}
```

①创建一个值为 v1/organizations /{organizationId}/licences {licenseId} 的 GET 端点

②将 URL 中的两个参数 (organizationId 和 licenseId) 映射到方法参数

你在这个清单中做的第一件事是：使用方法级注解 `@RequestMapping` 注释

`getLicenses()` 方法，通过在两参数注释：`value` 和 `method`。使用一个方法级

`@RequestMapping` 注解，你将在顶级类中创建指定的根级注释来匹配所有传入的 HTTP

请求到端点为 `/v1/organizations/{organizationId}/licences/{licenseId}` 的控制器。注解

的第二个参数 `method`，指定该方法将匹配的 HTTP 谓词。在前面的例子中，你通过

`RequestMethod.GET` 枚举匹配到 GET 方法。

第二件事是注意清单 2.4 中，你在 `getLicenses()` 方法体中使用 `@PathVariable` 注解的

参数。`@PathVariable` 注解被用于映射传入 URL 的参数值（如用 `{parameterName}` 语法）

到你的方法的参数。在你的代码清单 2.4 中，你将从 URL 映射两个参数，`organizationId`

和 `licenseId`，到方法里两个参数级变量：

```
@PathVariable("organizationId") String organizationId,
@PathVariable("licenseId") String licenseId)
```

#### 端点的名称问题

在你沿着编写微服务的道路走得太远之前，确保你（在你的组织潜在的其他团队）为通过你的服务暴露的端点建立标准。微服务的 URL（统一资源定位器）应该能够清楚的表达服务

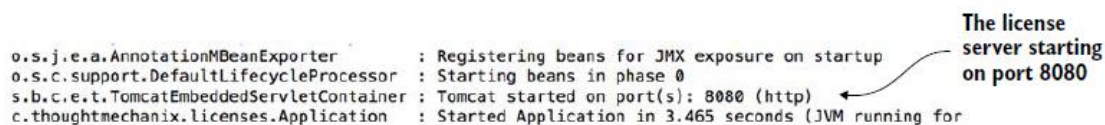
的目的，服务资源管理和在服务之间存在的资源管理关系。我发现下面的准则对于命名服务端点很有用：

- **使用明确的 URL 名称建立服务所代表的资源**：定义 URL 的规范格式将使你的 API 更直观，更容易使用。在命名约定中保持一致。
- **使用 URL 建立资源之间的关系**：通常在你的微服务资源之间有父子关系，哪里孩子不会在父上下文以外存在（因此你不可能有一个子微服务）。使用 URL 来表示这些关系。但如果你发现你的 URL 往往是太长和嵌套，那么你的微服务可能试图做太多的事情。
- **为 URL 建立早期版本控制方案**：URL 及其相应端点表示服务所有者和服务消费者之间的契约。一个常见的模式是在所有端点之前使用一个版本号。及早建立你的版本控制计划并坚持下去。在已经有好几个用户使用它们之后，为 URL 升级版本是非常困难的。

此时，你可以将其作为一个服务进行调用。从一个命令行窗口，切换到你已经下载示例代码的项目目录，执行以下的 Maven 命令：

**mvn spring-boot:run**

当您键入回车键时，您应该看到 Spring Boot 启动一个嵌入式 Tomcat 服务器，并开始监听端口 8080。



```
o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup
o.s.c.support.DefaultLifecycleProcessor : Starting beans in phase 0
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
c.thoughtmechanix.licenses.Application : Started Application in 3.465 seconds (JVM running for
```

图 2.4 许可服务启动成功

一旦服务启动，您就可以直接命中暴露的端点。因为第一个方法以 GET 调用暴露，所以可以使用许多种方法来调用服务。我的首选方法是使用像 POSTMAN 基于 Chrome 这样的工具或 CURL 来调用服务。图 2.5 显示了一个 GET 方式

<http://localhost:8080/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/>

licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a 端点。



图 2.5 许可服务被使用 POSTMAN 调用

① *When the GET endpoint is called, a JSON payload containing licensing data is returned.*

当 GET 端点被调用时，返回包含许可数据的 JSON 有效载荷。

此时，您有一个运行的服务框架。但是从开发的角度来看，这个服务并不完整。一个好的微服务设计并不回避将服务分离为定义明确的业务逻辑和数据访问层。当您在以后的章节中取得进展时，您将继续迭代这个服务并深入研究如何构造它。

让我们转换到最终的视角：探索 DevOps 工程师将如何实施服务并打包部署到云。

## 2.4. 运维：严格的运行时构建

对于 DevOps 工程师，微服务设计就是所有关于投产后服务的管理。编写代码通常是容易的部分。保持它的运行是最困难的部分。

而 DevOps 是一个丰富多彩的和新兴的 IT 领域，你将使用四项原则开始你的微服务开发工作，在本书的后面建立在这些原则。这些原则是：

- 微服务应该是独立和单独可部署的，多个服务实例作为单独的软件构件启动和停止。
- 微服务是可配置的。当服务实例启动时，它应该读取它需要从中心位置配置自己的



数据，或者将其配置信息传递为环境变量。配置服务不需要人为干预。

- 微服务实例需要对客户端是透明的。客户端永远不应该知道服务的确切位置。相反，一个微服务客户端应该与服务发现代理交互，将允许应用程序找到一个微服务实例，而无需知道它的物理位置。
- 微服务应该显示其健康状况。这是云架构的关键部分。微服务实例失败，客户端需要路由绕开坏的服务实例。

这四个原则揭示了矛盾与微服务开发共存。微服务规模和范围较小，但它们的使用在一个应用中引入了更多的活动部件，特别是因为微服务是分布式的，并且在自己的分布式容器彼此独立运行。这将为应用程序中的故障点提供高度的协调和更多的机会。

从一个 DevOps 的视角，你必须解决一个微服务预先的业务需求，将这四个原则转变为一套标准的生命周期事件（每次微服务发生构建和部署到一个环境）。这四条原则可以映射到以下操作生命周期步骤：

- 服务装配：你如何打包和部署服务以保证可复用和一致性，以便以同样的方式部署相同的服务代码和运行时？
- 服务引导：你如何将应用程序与运行时代码的特定环境下的配置代码分离，这样你就可以开始在任何环境中快速部署微服务实例而不需要人的干预来配置微服务？
- 服务注册/发现：当一个新的微服务实例被部署，如何使新的服务实例被其他应用程序的客户端发现？
- 服务监控：在微服务环境是极为常见的，由于高可用性需求而运行相同服务的多个实例。从一个 DevOps 的角度来看，你需要监控微服务实例和保证微服务任何故障被绕过，境况不佳的服务实例都被取下。

图 2.6 显示了这四个步骤是如何组合在一起的。



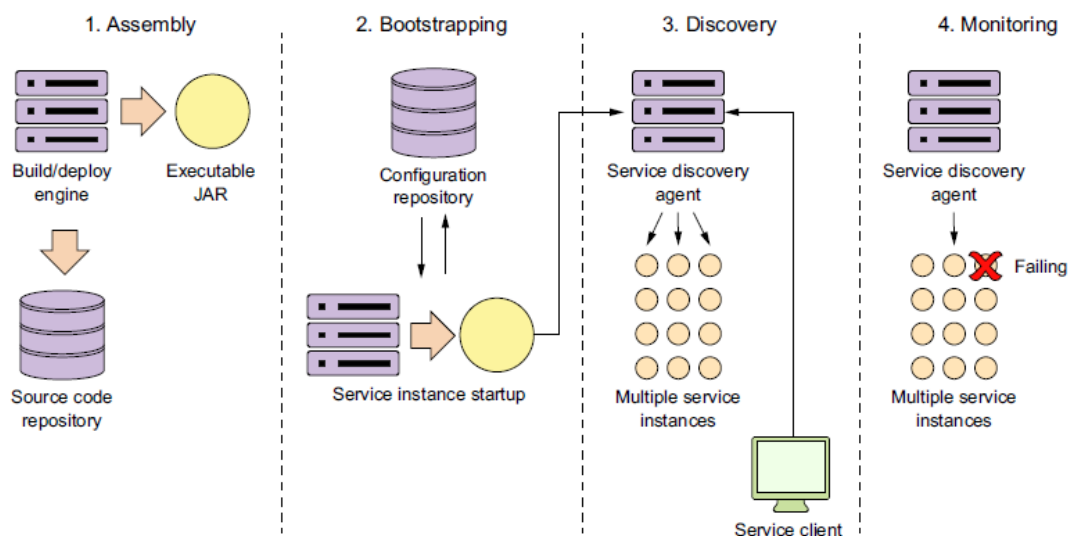


图 2.6 当微服务启动时，它完成在其生命周期中的多个步骤。

### 构建微服务服务应用的十二要素

我在这本书的一个最大的希望是，你意识到一个成功的微服务架构需要强大的应用开发和 DevOps 的实践。这一做法的最简洁的总结可以在 Heroku 的十二要素应用宣言 (<https://12factor.net/>) 发现。这个文档提供了 12 个最佳实践，在构建微服务的时候，你应该始终谨记在脑海里。当你阅读这本书时，你会看到这些实践贯穿于示例中。我总结如下：

**基准代码：**所有应用程序代码和服务器配置信息都应处于版本控制中。每个微服务在源码控制系统内有自己独立的代码库。

**依赖：**通过使用构建工具如 Maven (java) 显式声明你的应用程序的依赖。第三方 JAR 依赖性应该使用其特定版本号声明。这允许你的微服务总是使用相同版本的 lib 库进行构建。

**配置：**独立于代码存储应用程序配置（特别是特定于环境的配置）。你的应用程序配置不应该与源代码在同一个存储库中。

**后端服务：**你的微服务往往会通过网络与一个数据库或消息系统进行通信。当它这样做时，你应该确保在任何时候，你都可以将数据库的实现从内部管理的服务转换为第三方服务。在第 10 章中，我们演示了这一点：把你的服务从本地管理的 Postgres 数据库改变成由亚马

逊管理的。

**构建, 发布, 运行:** 将你的构建, 发布, 和运行部署的应用程序各部分完全分离。一旦代码被构建, 开发人员就永远不应该在运行时对代码进行更改。任何修改都需要重新执行构建过程和重新部署。构建的服务是不可变的, 不能更改。

**进程:** 你的微服务应该是无状态的。它们在任何时候都可以被杀死和取代, 而不必害怕一个受损的服务实例将导致数据丢失。

**端口绑定:** 一个微服务对服务的运行时引擎 (打包在服务的可执行文件) 是完全独立的。你将能运行该服务而不需要分离的 Web 或应用服务器。服务应该在命令行中独立启动, 并通过暴露的 HTTP 端口立即被访问。

**并发:** 当您需要伸缩时, 不要依赖于单个服务中的线程模型。相反, 加载更多的微服务实例和在水平方向扩展。这并不排除在你的微服务使用线程, 但别指望把它作为你伸缩的唯一途径。规模扩大, 而不是上升。

**通用性:** 微服务是一次性的, 一经要求就可以启动和停止。启动时间应尽量减少, 当接收到操作系统的一个终止信号时, 进程应该优雅地关闭。

**开发环境与线上环境等价:** 最小化服务运行的所有环境之间的差距 (包括开发人员的桌面)。开发人员应该在本地使用与实际服务将运行的相同的基础设施来进行服务开发。它还意味着服务在环境之间部署的时间应该是几个小时, 而不是几个星期。一旦代码被提交, 它应该已经被测试和从开发一直到生产尽快升级。

**日志:** 日志是一个事件流。一旦日志被输出, 它们对工具将是可流化的, 如 Splunk (<http://splunk.com>) 或 Fluentd (<http://fluentd.org>), 它们将聚合日志并将其写入到中心存储设备。微服务不应该关心这项技术是如何出现的, 开发人员应该在视觉上通过 STDOUT 看日志, 当日志正在被输出的时候。

**管理进程：**开发人员常常不得不针对他们的服务（数据迁移或转换）执行管理任务。这些任务不应该是临时的，而应该通过源代码存储库管理和维护的脚本来完成。这些脚本应该是可重复的，并且在它们运行的每个环境中都不改变（在每个环境中都不修改脚本代码）。

### 2.4.1. 服务装配：打包和部署微服务

从一个 DevOps 角度，微服务架构背后的关键概念是，一个微服务多实例可以被快速部署以响应应用环境的变化（例如，用户请求的突然涌入、基础设施中的问题等）。

要做到这一点，一个微服务需要作为一个单独的构件与它定义的所有的依赖被打包和安装。这个构件可以被部署到任意一台已安装 Java JDK 的服务器上。这些依赖关系，还包括将运行微服务的运行时引擎（例如，HTTP 服务器或应用程序容器）。

这个持续构建、打包和部署的过程是服务装配（图 2.6 中的步骤 1）。图 2.7 显示了关于服务装配步骤的其他详细信息。

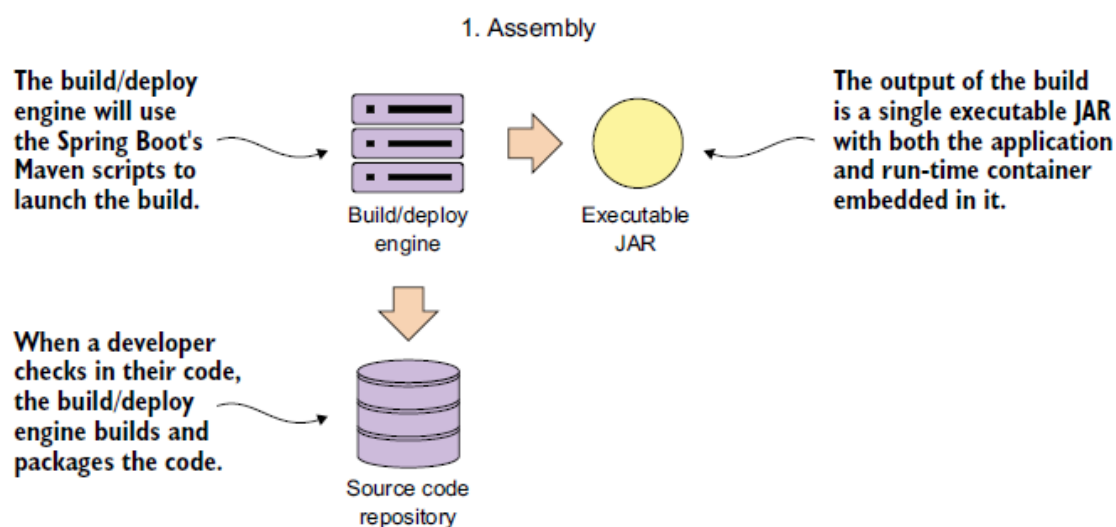


图 2.7 在服务装配步骤中，源代码在它的运行时引擎环境下被编译并打包。

① *The build/deploy engine will use the Spring Boot's Maven scripts to launch the build.*

构建/部署引擎将使用 Spring Boot 的 Maven 脚本来启动和构建。

② *When a developer checks in their code, the build/deploy engine builds and packages the code.*

当开发人员检出代码时，构建/部署引擎构建并打包代码。

③ The output of the build is a single executable JAR with both the application and run-time container embedded in it.

构建的输出是一个单独的可执行 JAR，其中嵌入了应用程序和运行时容器。

幸运的是，几乎所有的 Java 微服务框架将包括一个运行时引擎，它可以打包和部署代码。例如，在图 2.7 的 Spring Boot 示例中，你可以使用 Maven 和 Spring Boot 构建一个可执行的 Java jar 文件，它有一个嵌入式 Tomcat 引擎内置到 JAR 里。在下面的命令行示例中，你将许可服务构建为可执行 JAR，然后从命令行启动 JAR 文件：

```
mvn clean package && java -jar target/licensing-service-0.0.1-SNAPSHOT.jar
```

对于某些运维团队，在 JAR 文件中嵌入运行时环境的概念是他们考虑部署应用程序的方式的重大转变。在传统 J2EE 企业组织中，应用程序部署到应用服务器。这个模型意味着应用服务器本身就是一个实体，通常由一个系统管理员来管理，这些系统管理员独立于其部署的应用程序之外，管理服务器的配置。

应用程序服务器配置与应用程序的分离引入了部署过程中的故障点，因为在许多组织中，应用服务器的配置并不保留在源代码控制之下，而是通过用户界面和自己编写的管理脚本的组合来管理的。配置漂移很容易潜入应用服务器环境，并突然导致表面上出现的随机中断。

使用嵌入在构件中的运行时引擎的单个可部署构件消除了许多配置漂移的机会。它还允许你将整个构件放在源代码控制之下，并允许应用程序团队通过如何构建和部署应用程序来更好地解释原因。

### 2.4.2. 服务引导：管理微服务的配置

服务引导(在图 2.6 步骤 2 中)在微服务第一次启动并需要加载应用程序的配置信息时发生。图 2.8 为引导处理提供了更多的上下文。

正如任何应用程序开发人员都知道的那样，有时需要使用应用程序的运行时行为可配置。通常这涉及从应用程序部署的属性文件中读取应用程序配置数据，或者从数据存储(如关系数据库)中读取数据。

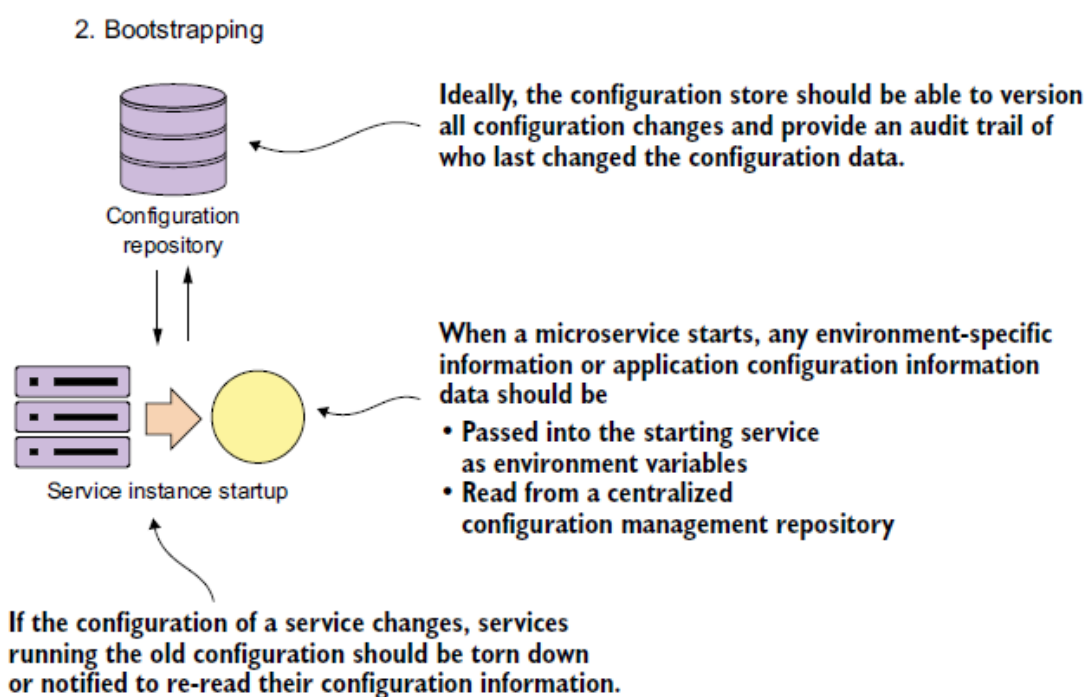


图 2.8 当服务启动(引导), 它从中央存储库读取配置。

① *If the configuration of a service changes, services running the old configuration should be torn down or notified to re-read their configuration information.*

如果服务的配置发生变化, 则应销毁正在运行得服务的旧配置或通知重新读取它们的配置信息。

② *Ideally, the configuration store should be able to version all configuration changes and provide an audit trail of who last changed the configuration data.*

理想情况下，配置存储应该能够对所有配置更改进行版本化，并提供对上次更改配置数据的审核跟踪。

③ *When a microservice starts, any environment-specific information or application configuration information data should be*

- *Passed into the starting service as environment variables*
- *Read from a centralized configuration management repository*

当一个微服务启动，任何环境的具体信息或应用程序的配置数据信息应该

- 作为环境变量传递到启动服务
- 从中央配置管理库中读取

微服务经常遇到同一类型的配置要求。不同的是，微服务应用运行在云中，你可能有数百甚至数千微正在运行的服务实例。更复杂的是，这些服务可能会遍布全球。大量的地理位置分散的服务，这将使使用新的配置数据重复部署你的服务变得难以实施。

将数据存储在服务的外部数据存储中解决了这一问题，但为云端微服务提供了一堆独特的挑战：

- 配置数据往往结构简单，通常是经常读和很少写的。在这种情况下，关系型数据库被过度使用，因为它们是被设计来管理更为复杂的数据模型，而不仅仅是一个简单的键值对集合。
- 因为数据是被有规律的访问的，但很少变化，所以数据必须具有低延迟的可读性。
- 数据存储必须高度可用，并且与读取数据的服务关系密切。配置数据存储不能完全关闭，因为它将成为应用程序的单一故障点。

在第 3 章中，我将展示如何使用像一个简单的 key-value 数据存储管理你的微服务应用程序配置数据。

### 2.4.3. 服务注册与发现：客户端如何与微服务通信

从微服务消费者的角度，微服务应该是位置透明的，因为在基于云的环境，服务器是短暂的。短时间意味着服务器承载的服务通常比在企业数据中心运行的服务寿命更短。基于云的服务器可以很快地启动和卸载，并使用一个全新的 IP 地址分配给正在运行的服务器。

坚持将服务器视为短暂的一次性对象，微服务架构通过一个服务运行多个实例，可以实现高度的可扩展性和可用性。服务需求和弹性可以在情况允许的情况下迅速得到管理。每个服务都有一个唯一的和永久的 IP 地址分配给它。临时服务器的缺点是，随着服务器的不断增加，人为或手工管理大量的临时服务器池将导致中断缺陷。

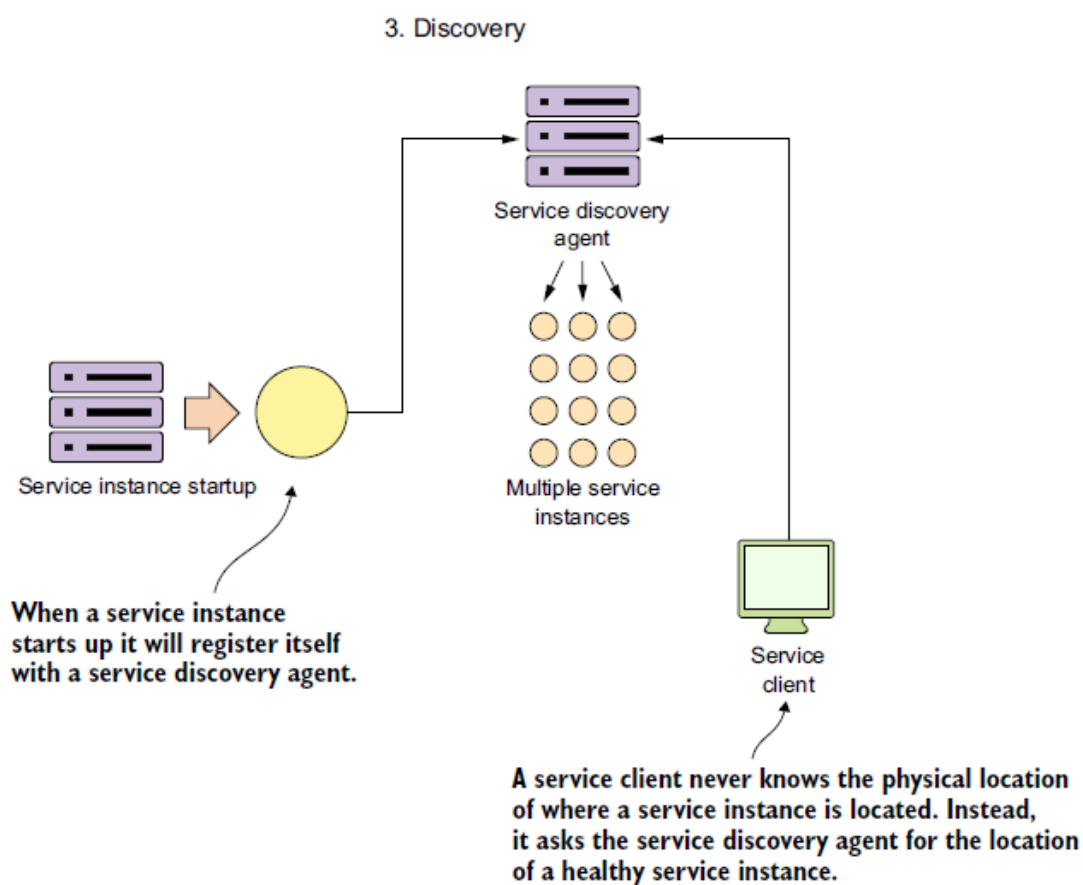


图 2.9 服务发现代理抽象出服务的物理位置。

① *When a service instance starts up it will register itself with a service discovery agent.*

当服务实例启动时，它将向服务发现代理注册自身。

② *A service client never knows the physical location of where a service instance is located. Instead, it asks the service discovery agent for the location of a healthy service instance.*

服务客户端永远不知道服务实例所在的物理位置。相反，它向服务发现代理询问一个健康服务实例的位置。

一个微服务实例需要在第三方代理注册本身。这个注册过程称为服务发现（参见如图 2.6 所示服务发现步骤 3；参见图 2.9，了解此过程的详细信息。）。当一个微服务实例在服务发现代理注册，它将告诉发现代理的两件事：该服务实例的物理 IP 地址或域名地址，和一个逻辑名称，即应用程序可以使用它查找服务。某些服务发现代理还需要一个 URL 返回到注册服务，服务发现代理可以使用它来执行健康检查。

然后，服务客户端与发现代理进行通信，查找服务的位置。

#### 2.4.4. 监控微服务健康状况

服务发现代理不只是充当一个将客户端引导到服务位置的流量管理器。在一个基于云的微服务应用，你通常有一个服务运行多个实例的情况。迟早有一天，其中的一个服务实例会失败。服务发现代理监视注册的每个服务实例的健康情况，并从路由表中删除任意服务实例，以确保客户端不向已失败的服务实例发送请求。

微服务恢复以后，服务发现代理将继续监控和 ping 健康检查接口来确保服务可用。这是图 2.6 中的步骤 4。图 2.10 提供了此步骤的上下文。

通过构建一个一致的健康检查接口，你可以使用基于云的监控工具来发现问题并对其作出适当的响应。



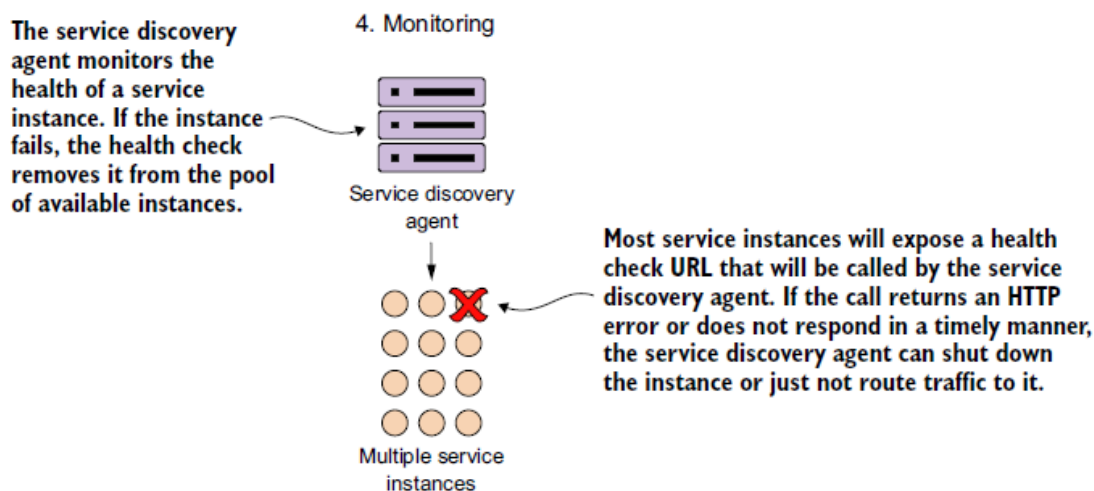


图 2.10 服务发现代理使用暴露的健康状况 URL 来检查微服务的健康状况。

① *The service discovery agent monitors the health of a service instance. If the instance fails, the health check removes it from the pool of available instances.*

服务发现代理监视服务实例的健康情况。如果实例失败，健康检查将从可用实例池中删除它。

② *Most service instances will expose a health check URL that will be called by the service discovery agent. If the call returns an HTTP error or does not respond in a timely manner, the service discovery agent can shut down the instance or just not route traffic to it.*

大多数服务实例将暴露一个将由服务发现代理调用的健康检查 URL。如果调用返回一个 HTTP 错误或没有及时响应，则服务发现代理可以关闭该实例或不将路由路由到它。

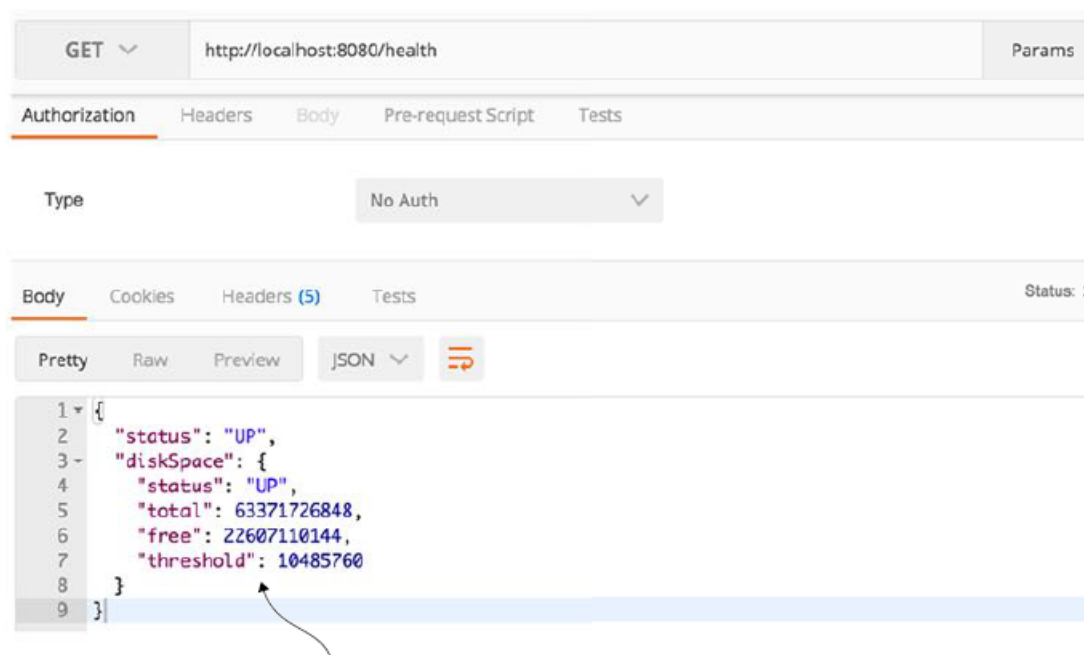
如果服务发现代理发现了服务实例的问题，它可以采取纠正措施，例如关闭有问题的实例或启动额外的服务实例。

在一个使用 REST 的微服务环境，创建一个健康检查接口最简单的方法是暴露一个 HTTP 端点，它可以返回一个 JSON 有效负载和 HTTP 状态代码。在一个没有基于 Spring Boot 的微服务，编写一个返回服务健康状况的端点，这通常是开发人员的责任。

在 Spring Boot，暴露一个端点是微不足道的，只不过是修改你的 Maven 构建文件包

括 Spring Actuator 模块。Spring Actuator 提供开箱即用的操作端点，这将帮助你了解和  
管理服务的健康状况。为了使用 Spring Actuator，你需要确保在你的 Maven 构建文件中  
包括以下依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```



The "out of the box" Spring Boot health check will return whether the service is up and some basic information like how much disk space is left on the server.

图 2.11 每个服务实例的健康检查允许监视工具确定服务实例是否正在运行。

① The "out of the box" Spring Boot health check will return whether the service is up and some basic information like how much disk space is left on the server.

“开箱即用”的 Spring Boot 健康检查将返回服务是否已启动以及一些基本信息，如服务器上留下了多少磁盘空间。

如果你访问了在许可服务上的 http://localhost:8080/health 端点，你应该看到返回的健康数据。图 2.11 提供了返回数据的示例。

正如您在图 2.11 中所看到的，健康检查可能不仅仅是一个启动和停止的指标。它还可

以提供有关正在运行微服务实例的服务器状态。这使得监控体验更加丰富。<sup>1</sup>

## 2.5. 观点汇总

云端微服务看起来似乎简单。但要成功的话，你需要有一个综合的观点，把架构师，开发人员和 DevOps 工程师合在一起的综合视角。对于这些观点的要点是：

- 架构师：关注业务问题的自然轮廓。描述你的业务问题域，听你讲的故事。目标候选微服务将出现。记住，那就是从一个“粗”微服务和可重构为小的服务比从一大群小服务开始要好。微服务架构像大部分优秀的架构一样，都是突然出现而未经事先计划到分钟的。
- 软件工程师：事实上，服务小并不意味着良好的设计原则被抛出窗外。专注于构建分层服务，其中服务中的每一层都有独立的职责。避免在你的代码建立框架的诱惑，努力使每个微服务完全独立。过早的框架设计和采用可能会在应用程序的生命周期后期产生大量的维护成本。
- DevOps 工程师：服务不存在于真空之中。尽早建立服务的生命周期。DevOps 的视角不仅要关注如何自动构建和部署服务，同时也关注如何监控服务的健康状况和在出错时做出反应。服务运维往往比写业务逻辑需要更多的工作和远见卓识。

## 2.6. 小结

- 要使微服务成功，你需要将架构师，软件开发人员，DevOps 的观点结合在一起。
- 微服务，虽然是强大的架构范式，但也有自己的利益和权衡。不是所有的应用程序都应是微服务应用。
- 从架构师的视角，微服务是小的，独立的，分布式。微服务应该有有限的范围和管理一

<sup>1</sup>Spring Boot 提供了大量定制你的健康检查的选项。欲了解更多详情，请查阅优秀书籍《Spring Boot 实战》(Manning Publications, 2015)。作者 Craig Walls 详尽地介绍了配置 Spring Boot Actuators 的各种不同机制。

个小的数据集。

- 从开发者的视角，微服务通常使用 REST 样式设计构建，JSON 作为发送和从服务接收数据的有效载荷。
- Spring Boot 是构建微服务的理想框架，因为它让你使用很少的简单注释，就可以创建一个基于 REST 的 JSON 服务。
- 从 DevOp 的视角，微服务如何打包、部署和监控是至关重要的。
- 开箱即用，Spring Boot 允许你将服务作为一个独立的可执行 JAR 文件交付。生产者 JAR 文件中的嵌入式 Tomcat 服务器承载服务。
- Spring Actuator，它包含在 Spring Boot 框架中，其暴露了关于服务运行健康的信息以及服务运行时的信息。