3. 第 3 章 使用 Spring Cloud Configuration Server 控制配置

本章内容

- 将服务配置与服务代码分离
- 配置一个 Spring Cloud 配置服务器
- 集成一个 Spring Boot 微服务
- 加密敏感特性

在某种程度上，开发人员将被迫从代码中分离配置信息。毕竟，从学校开始，他们就不应该硬编码代码到应用程序代码中。许多开发人员在其应用程序中使用常量类文件，以帮助将所有配置集中在一个地方。应用程序配置数据直接被写入代码的问题往往是因为每次更改配置了应用程序必须重新编译或重新部署。为了避免这种情况，开发人员将配置信息从应用程序代码中完全分离出来。这使得它易于更改配置无需经过重新编译的过程，同时也引入了复杂性，因为你现在有了另一个构件，它需要应用程序管理和部署。

许多开发人员将转向使用低性能文件（或 YAML、JSON 或 XML）来存储配置信息。此属性文件将放在服务器上，该服务器通常包含数据库和中间件连接信息和应用程序元数据，这些元数据将驱动应用程序的行为。分离你的应用程序在一个属性文件是容易的，大多数开发人员不会对应用程序配置做任何更多的操作，然后把配置文件放置在源代码管理（如果这样），它将作为应用程序的一部分部署。

这种方法可与少量的应用程序工作，但很快分崩离析，当处理基于云的应用程序可能包含数百个微服务，在把每个微服务可能有多个运行时服务实例。

突然，配置管理成了一个大问题，因为云环境中的应用程序和运维团队必须全力应付配置文件到哪去。基于云的微服务开发强调：

- 将应用程序的配置完全从正在部署的实际代码分离
- 创建服务器和应用程序以及一个不可变的镜像，它在你的环境被升级时不会改变
- 将任何应用程序配置信息在服务器启动时通过环境变量注入或在启动时应用程序的微服务从一个中央仓库读取

本章将向你介绍管理一个基于云的微服务应用程序的应用配置数据的核心原则和模式。

3.1. 管理配置（和复杂性）

管理应用程序的配置对于运行在云端的微服务是至关重要的，因为微服务实例需要以最小的人为干预，迅速启动。每当一个人需要手动配置或获取某个服务来部署它时，配置漂移、意外中断和响应时间延迟对于应用程序的可伸缩性带来挑战。

让我们通过建立我们想要遵循的四个原则来开始关于应用程序配置管理的讨论：

- 隔离：我们希望将服务配置信息与服务的实际物理部署完全分离。应用程序配置不应该部署到服务实例中。相反，在服务启动时，配置信息应该作为环境变量传递到正在启动的服务或从中央仓库中读取。
- 抽象：抽象服务接口背后的配置数据的访问。与其编写直接访问服务存储库的代码（即，读取文件的数据或使用 JDBC 访问数据库），不如让应用程序使用基于 REST 的 JSON 服务检索配置数据。
- 集中：因为基于云的应用程序可能有数百个服务，因此最小化保存配置信息的不同存储库的数量是至关重要的。将应用程序配置集中到尽可能少的存储库中。
- 稳定：因为你的应用程序的配置信息将与你部署的服务完全隔离和中心化，至关重要的是，无论你使用何种解决方案，都可以实现高可用和冗余。

要记住的关键一点是，当将配置信息分离到实际代码之外时，你将创建一个需要管理和

版本控制的外部依赖项。我再次强调，应用程序配置数据需要跟踪和版本控制，因为管理不善的应用配置是不易察觉的错误和意外中断一个肥沃的滋生地。

意外的复杂性

我亲身体验过没有策略来管理应用程序配置数据的危险。当我在财富 500 强金融服务公司工作时，我被要求帮助把一个大型的 WebSphere 升级项目重新引入正轨。公司的问题是在 WebSphere 上有超过 120 个应用程序，需要在整个应用程序环境在供应商维护的基础上结束之前，将其基础设施从 WebSphere 6 升级到 WebSphere 7。

该项目已经进行了一年，只有 120 个应用程序中的一个在部署。该项目在人员和硬件成本上花费了一百万美元，而且按目前的进度还需要两年时间才能完成升级。

当我开始与应用程序团队合作时，我发现的一个主要问题是，应用程序团队使用他们的数据库管理所有配置，并在属性文件内部管理他们的服务端点。这些属性文件是手动管理的，不在源代码管理之下。有 120 个应用程序分布在四个环境和每个应用程序的多个 WebSphere 节点上，这个硕大的配置文件巢导致团队试图迁移 12000 个配置文件，这些文件分布在服务器上运行的数百个服务器和应用程序上。（你读到的数字是正确的：12000）这些文件仅用于应用程序配置，甚至不包括应用程序服务器配置。

我说服项目发起人花两个月的时间将所有应用程序信息合并到一个具有 20 个配置文件的集中式、版本控制的配置存储库中。当我问框架团队他们如何到了有 12000 个配置文件程度的时候，团队的领导说他们最初围绕一个小应用程序设计了他们的配置策略。然而，构建和部署的 Web 应用程序的数量在五年内爆炸了，尽管他们要求金钱和时间来重新配置配置管理方法，但他们的业务伙伴和 IT 领导从未认为这是优先事项。

如果不花时间来考虑如何进行配置管理，则会产生真正（且代价高昂）的下游影响。

3.1.1. 配置管理架构

你会记得第 2 章，在引导一个微服务的阶段发生一个微服务配置管理的加载。作为一个提醒，图 3.1 显示了微服务生命周期。

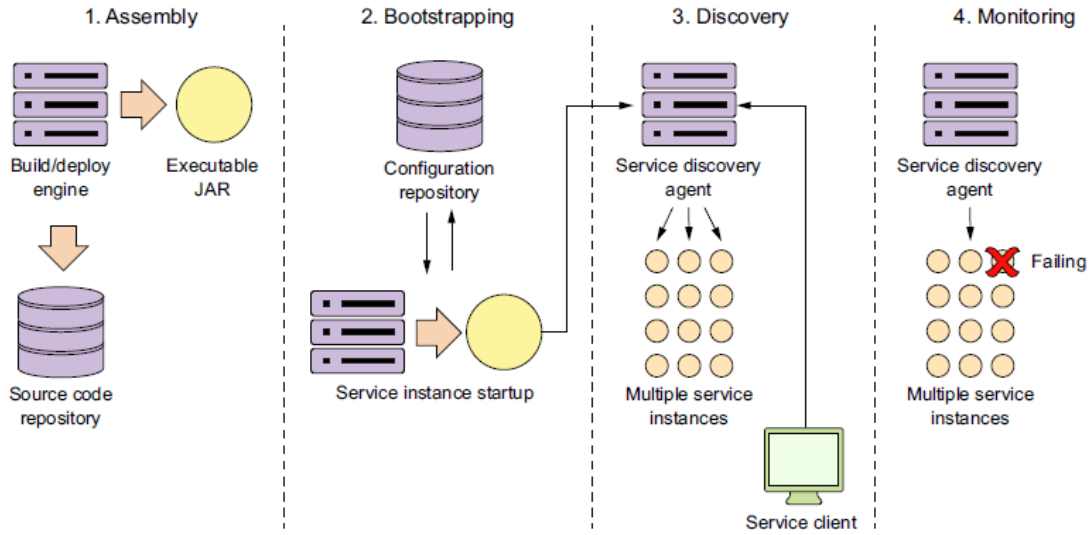


图 3.1 在服务引导阶段读取应用程序配置数据。

让我们来看看我们在第 3.1 节前面所阐述的四个原则（隔离、抽象、集中和稳定），看看这四个原则在服务引导时是如何应用的。图 3.2 更详细地探讨了引导过程，并展示了配置服务在这一步中起着关键的作用。

① Microservice instance starts up and obtains configuration information.

微服务实例启动并获得配置信息。

② Actual configuration resides in a repository

实际配置驻留在存储库中

③ Changes from developers are pushed through the build and deployment pipeline to the configuration repository.

来自开发人员的更改被通过构建和部署管道推送到配置存储库。

④ Applications with a configuration change are notified to refresh themselves.

配置更改的应用程序被通知刷新自己。

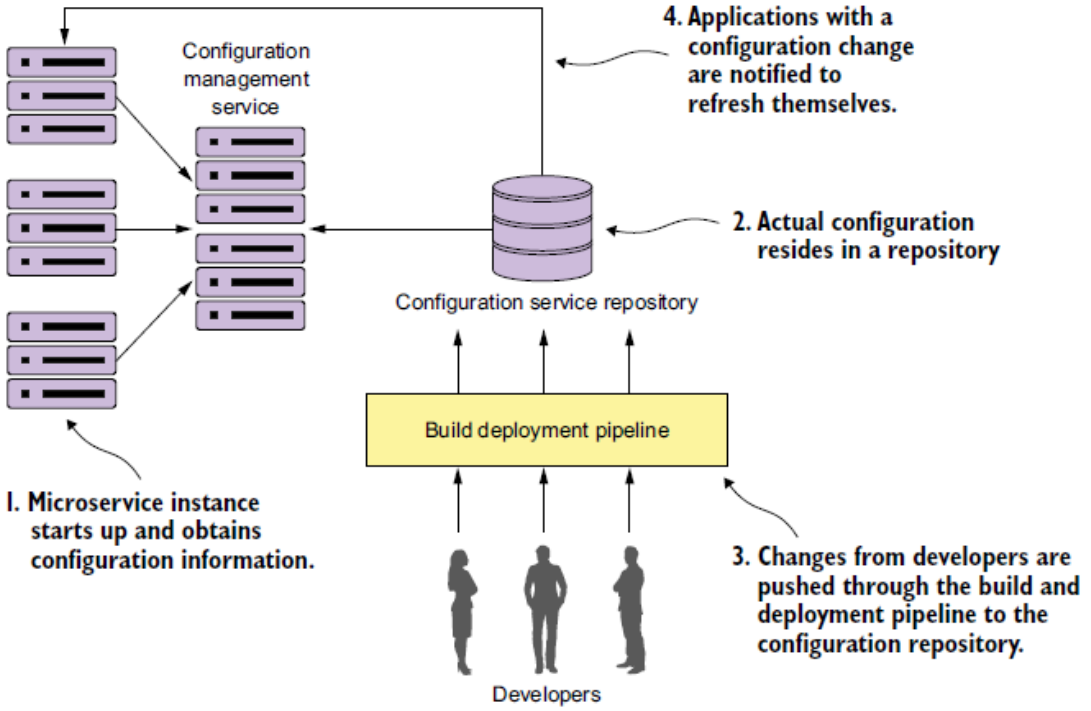


图 3.2 配置管理概念体系结构

在图 3.2 中，您看到了几个活动正在发生：

- 当微服务实例启动时，它会调用一个服务端点读取其操作的特定环境中的配置信息。在微服务启动时，配置管理的连接信息（连接凭据，服务端点，等等）会被传递到微服务。
- 实际配置将驻留在存储库中。基于配置存储库的实现，您可以选择使用不同的实现来保存配置数据。实现选项可以包括源代码控制的文件、关系数据库或键值数据存储。
- 应用程序配置数据的实际管理独立于应用程序部署的方式。配置管理的更改通常通过构建和部署管道来处理，其中配置的更改可以通过版本信息进行标记，并通过不同的环境进行部署。
- 当进行配置管理更改时，必须通知使用该应用程序配置数据的服务，并刷新应用程

序数据的副本。

在这一点上，我们已经完成了概念架构，它演示了配置管理模式的不同部分，以及这些组件是如何组合在一起的。 现在我们将继续为该模式研究不同解决方案，然后看看具体的实现。

3.1.2. 实现选择

让我们看看几种不同的选择，并比较它们。表 3.1 列出了这些选择。

表 3.1 实现配置管理系统的开源项目

项目名称	描述	特性
Etcd	编写于 Go 语言的开源项目。用于服务发现和键值管理。采用 raft（https://raft.github.io/）协议的分布式计算模型。	非常快速和可伸缩 分布式的 命令行驱动 易于使用和设置
Eureka	由 Netflix 撰写。经过充分的测试。用于服务发现和键值管理。	分布式键值存储。 柔性；通过配置提供动态客户端刷新的开箱即用功能
Consul	由 Hashicorp 编写。 特性与 Etcd 和 Eureka 类似，但使用不同的算法的分布式计算模型(SWIM protocol; https://www.cs.cornell.edu/~asdas/research/dsn02-swim.pdf)。	快速 提供直接与 DNS 集成的本地服务发现 不提供客户端动态刷新权限 开箱即用
ZooKeeper	提供分布式锁定功能的 Apache 项目。通	最古老，经过充分测试的解

	常用作访问键值数据的配置管理解决方案。	决方案 使用最复杂的 可用于配置管理，但应该考虑除非你已经使用 ZooKeeper 作为你架构的一部分
Spring Cloud configuration server	提供不同后端的通用配置管理解决方案的开放源代码项目。它可以与作为后端的 Git、Eureka 和 Consul 整合。	非分布式键值存储 为 Spring 和非 Spring 服务提供紧密集成 可以使用多个后端包括一个共享的文件系统，Eureka，Consul 和 Git 存储配置数据

表 3.1 中的所有解决方案都可以很容易地用于构建配置管理解决方案。对于本章和本书其余部分的示例，您将使用 Spring Cloud 配置服务器。我之所以选择这个解决方案有几个原因，包括以下几点：

- Spring Cloud 配置服务器很容易设置和使用。
- Spring Cloud 配置与 Spring Boot 紧密集成。您可以通过几个简单的注解来读取应用程序的所有配置数据。
- Spring Cloud 配置服务器提供多个后端来存储配置数据。如果你已经使用了像 Eureka 和 Consul 这样的工具 ,你可以把它们整合到 Spring Cloud 配置服务器中。
- 在表 3.1 中的所有解决方案中，Spring Cloud 服务器可以直接与 Git 源代码控制平台集成。Spring Cloud 配置与 Git 的集成消除了解决方案中的额外依赖性，并

使应用程序配置数据的版本化成为可能。

其他工具（Etcd, Consul, Eureka)不提供任何一种原生的版本，如果你想要，你必须自己建立它。如果你使用 Git，Spring Cloud 配置服务器的使用是一个诱人的选择。

在本章的剩余部分，你将要：

- 配置一个 Spring Cloud 配置服务器，并演示两种不同的机制：一种是使用文件系统，另一种是使用 Git 存储库来为应用程序配置数据提供服务。
- 继续构建许可服务以从数据库检索数据
- 将 Spring Cloud 配置服务集成到许可服务中，以提供应用程序配置数据。

3.2. 建立 Spring Cloud Config Server

Spring Cloud 配置服务器是构建在 Spring Boot 之上基于 REST 的应用程序。它不是一个独立的服务器。相反，您可以选择将其嵌入到现有的 Spring Boot 应用程序中，或者使用服务器启动一个新的 Spring Boot 项目的时候嵌入它。

你需要做的第一件事是建立一个新的项目，目录称为 confsvr。在 consvr 目录你会创建一个新的 Maven 文件，它将用来拉下启动你的 Spring Cloud 配置服务器需要的 JARs。不会讨论整个 Maven 文件，我会在下面列出关键部分。

清单 3.1 创建 Spring Cloud 配置服务器的 pom.xml 文件

```
<?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>configurationserver</artifactId>
<version>0.0.1-SNAPSHOT</version>
```

```
<packaging>jar</packaging>

<name>Config Server</name>
<description>Config Server demo project</description>

<parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>1.4.4.RELEASE</version>
</parent>
<dependencyManagement>
   <dependencies>
      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-dependencies</artifactId>
         <version>Camden.SR5</version>
         <type>pom</type>
         <scope>import</scope>
      </dependency>
   </dependencies>
</dependencyManagement>

<properties>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   <start-class>com.thoughtmechanix.confsvr.
   ➥ConfigServerApplication </start-class>
   <java.version>1.8</java.version>
   <docker.image.name>johncarnell/tmx-confsvr</docker.image.name>
   <docker.image.tag>chapter3</docker.image.tag>
</properties>

<dependencies>
   <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-config</artifactId>
   </dependency>

   <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-config-server</artifactId>
   </dependency>
</dependencies>
<!--Docker build Config Not Displayed -->
</project>
```

①你将使用的 Spring Boot 版本

②将要使用的 Spring Cloud 版本

③用于配置服务器的引导类

④你将在这个特定服务中使用的 Spring Cloud 项目

在前面列表的 Maven 文件里，你从声明你将为你的微服务使用的 Spring Boot 版本（version 1.4.4）。Maven 定义下的重要部分是你要用的 Spring Cloud 配置父 BOM（物料清单）。Spring Cloud 是大量独立项目的集合，它们都随自己的版本移动。这个父 BOM 包含了云项目中使用的所有第三方库和依赖项，以及组成该版本的各个项目的版本号。在这个例子中，你使用版本 camden.sr5 的 Spring Cloud。利用 BOM 的定义，你能保证在 Spring Cloud 项目中使用兼容版本的子项目。它还意味着您不必为子依赖项声明版本号。清单 3.1 中的其他示例涉及声明你将在服务中使用的特定 Spring Cloud 依赖项。第一个依赖项是所有 Spring Cloud 项目使用的 spring-cloud-starter-config 依赖项。第二个依赖项是 spring-cloud-config-server 启动项目。它包含了 spring-cloud-config-server 的核心库。

来吧，坐列车，发布列车

Spring Cloud 使用非传统机制标记 Maven 项目。Spring Cloud 是一个独立的子项目集。Spring Cloud 团队通过所谓的“发布列车”发布他们的版本。所有的子项目组成 Spring Cloud，它们使用 Maven 文件（BOM）打包并作为一个整体发布。Spring Cloud 团队已经利用伦敦地铁站的名字作为他们发布的名称，每个增量主要发布给伦敦地铁站，有下一个最高字母时停止。已经有三个版本：Angel，Brixton 和 Camden. Camden 是迄今为止最新的版本，但在子项目的分支内仍有多个候选版本。

需要注意的一点是，Spring Boot 是独立于 Spring Cloud 发布列车发布的。因此，不同版本的 Spring Boot 与 Spring Cloud 的不同版本是不兼容的。你可以看到 Spring Boot 和 Spring Cloud 之间的版本依赖关系，以及包含在发布培训项目的不同子项目版本，请参照 Spring Cloud 网站(http://projects.spring.io/spring-cloud/)。

你仍然需要设置一个文件以获取核心配置服务器并运行。这个文件是 application.yml，它在 confsvr/src/main/resources 目录。application.yml 文件会告诉 Spring Cloud 配置

服务所监听的端口和定位的后端，后端将提供配置数据。

你几乎准备好启动 Spring Cloud 配置服务了。你需要将服务器指向保存配置数据的后端存储库。在本章中，你将使用第 2 章中开始构建的许可服务作为如何使用 Spring Cloud Config 的一个示例。为了使事情简单，你将为三个环境设置应用程序配置数据：本地运行服务时的默认环境、开发环境和生产环境。

在 Spring Cloud 配置中，一切都脱离层次结构。应用程序配置由应用程序的名称来表示，然后为每个环境设置一个属性文件，以便为其配置信息。在每个环境中，你将设置两个配置属性：

- 将由你的许可服务直接使用的示例属性
- 用于存储许可服务数据的 Postgres 数据库的数据库配置

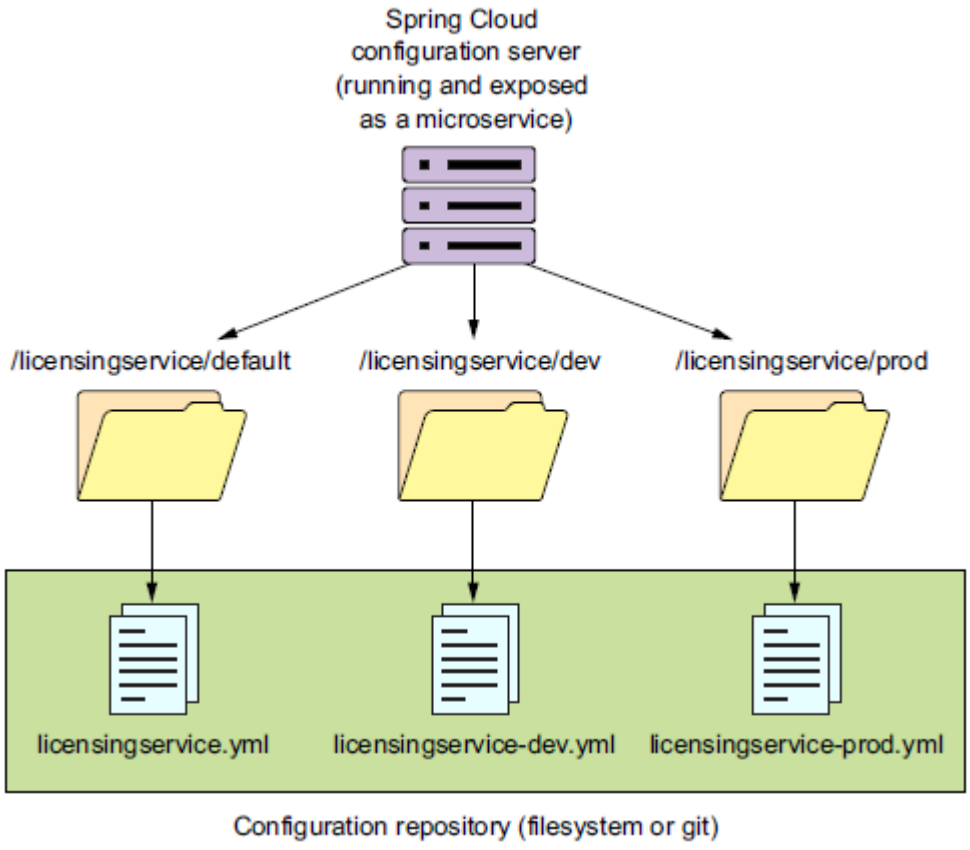


图 3.3 Spring Cloud 配置将特定环境的属性暴露为基于 HTTP 的端点。

① Spring Cloud configuration server(running and exposed as a microservice)

Spring Cloud 配置服务器（作为微服务运行和暴露）

图 3.3 演示了如何设置和使用 Spring Cloud 配置服务。有一点要注意的是，当你建立你的配置服务，这将是另一个运行在你的环境中的微服务。一旦设置完毕，服务的内容就可以通过基于 HTTP 的 REST 端点访问。

应用程序配置文件命名约定为 appnameenv.yml。从图 3.3 中的图表中可以看到，环境名称直接转换为将访问配置信息的 URL。 然后，当你开始许可微服务例子时，你想运行服务的环境反对通过你进入命令行服务启动的 Spring Boot 概要文件指定。如果概要文件在命令行中没有通过，Spring Boot 会默认使用打包在应用程序内的 application.yml 文件包含的配置数据。

下面是你将为许可服务提供的一些应用程序配置数据的一个示例。这些数据将包含在 confsvr/src/main/resources/config/licensingservice/licensingservice.yml 文件（在图 3.3 被提及）。这是这个文件的一部分内容：

```
tracer.property: "I AM THE DEFAULT"
spring.jpa.database: "POSTGRESQL"
spring.datasource.platform: "postgres"
spring.jpa.show-sql: "true"
spring.database.driverClassName: "org.postgresql.Driver"
spring.datasource.url: "jdbc:postgresql://database:5432/eagle_eye_local"
spring.datasource.username: "postgres"
spring.datasource.password: "p0stgr@s"
spring.datasource.testWhileIdle: "true"
spring.datasource.validationQuery: "SELECT 1"
spring.jpa.properties.hibernate.dialect:
    "org.hibernate.dialect.PostgreSQLDialect"
```

在实现之前先想一想

我建议不要使用基于文件系统的解决方案来解决中型到大型云应用程序。使用文件系统方法意味着需要为希望访问应用程序配置数据的所有云配置服务器实现共享文件挂载点。在

云中设置共享文件系统服务器是可行的，但它将维护此环境的责任放在了你身上。

我展示了文件系统方法，当使用 Spring Cloud 配置服务器时，它是最容易使用的示例。在后面的章节中，我将展示如何配置 Spring Cloud 配置服务器使用基于云的 Git 提供商，如 Bitbucket 或 GitHub 存储应用程序配置。

3.2.1. 创建 Spring Cloud 配置引导类

本书所涵盖的每一个 Spring Cloud 服务都需要一个引导类来启动服务。这个引导类将包含两点：一个 Java main()方法，作为服务启动的切入点，和一套 Spring Cloud 注解，它们告诉正在启动的服务，它们将为服务提供什么样的 Spring Cloud 行为。

下面列表显示的 confsvr/src/main/java/com/thoughtmechanix/confsvr /Application.java 类，被用作配置服务的引导类。

清单 3.2 Spring Cloud 配置服务器引导类

```
package com.thoughtmechanix.confsvr;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
    }
}
```

①Spring Cloud 配置服务是一个 Spring Boot 应用程序，所以你把它标记为@SpringBootApplication。

②@EnableConfigServer 注解使服务的 main()方法将服务作为 Spring Cloud 配置服务启动

③main 方法启动服务并启动 Spring 容器。

接下来，你将用我们最简单的示例建立 Spring Cloud 配置服务器：文件系统。

3.2.2. 通过配置文件使用 Spring Cloud Config Server

Spring Cloud 配置服务器使用在 confsvr/src/main/resources/application.yml 文件

的一个指向仓库的入口点，仓库将存储应用程序的配置数据。建立基于文件系统的存储库是实现这一目标的最简单方法。

为此，添加以下信息来配置服务器的 application.yml 文件。下面的列表显示了 Spring Cloud 配置服务器的 application.yml 文件的内容。

清单 3.3 Spring Cloud 配置的 application.yml 文件

```
server:
    port: 8888 ←——— ①Spring Cloud 配置服务器将监听的端口
spring:
  profiles:
    active: native ←——— ②将用于存储配置的后端存储库（文件系统）。
  cloud:
     config:
        server:                                          ③存储配置文件的
            native:                                      路径。
              searchLocations: file:///Users/johncarnell1/book/
              native_cloud_apps/ch4-config-managment/confsvr/src/main/
              resources/config/licensingservice
```

在清单中的配置文件中，您首先告诉配置服务器它应该为所有配置请求监听哪个端口号：

```
server:
    port: 8888
```

因为你正在使用文件系统来存储应用程序配置信息，所以你需要告诉 Spring Cloud 配置服务器以“native”配置文件运行：

```
profiles:
     active: native
```

在 application.yml 文件最后一部分，Spring Cloud 配置提供了应用数据所在的目录：

```
server:
    native:
        searchLocations: file:///Users/johncarnell1/book/spmia_code/chapter3-
            code/confsvr/src/main/resources/config
```

配置项的重要参数是 searchLocations 属性。这个属性为每个应用程序提供一个逗号分隔的目录列表，每个应用程序都有由配置服务器管理的属性。在前面的示例中，你只配置了许可服务。

> ***注意***：注意，如果你使用 Spring Cloud 配置本地文件系统的版本，你需要修改 spring.cloud.config.server.native.searchlocations 属性来反映当在本地运行你的代码时你的本地文件路径。

你现在已经完成了足够的工作来启动配置服务器。继续使用 mvn spring-boot:run 命令启动配置服务器。在命令行上，服务器现在应该使用 Spring Boot 启动画面启动。如果你将你的浏览器指向到 http://localhost:8888/licensingservice/default ，你会看到 JSON 负载返回所有包含在 licensingservice.yml 文件中的属性。图 3.4 显示了调用此端点的结果。

图 3.4 检索许可服务的默认配置信息

① The source file containing the properties in the config repository

在配置存储库中包含属性的源文件

如果你想看到许可服务开发环境的配置信息，点击 http://localhost:8888/licensingservice/dev 端点，以 GET 方式获取。图 3.5 显示了调用此端点的结果。

如果仔细查看，你会发现，当你点击 dev 端点时，将返回许可服务的默认配置属性和许可服务的开发环境配置。Spring Cloud 配置返回两组配置信息的原因是 Spring 框架实

现了一个分层的解析属性机制。当 Spring 框架执行属性解析时，它总是首先查找默认属性中的属性。然后，如果存在一个特定于环境的值，则覆盖默认值。

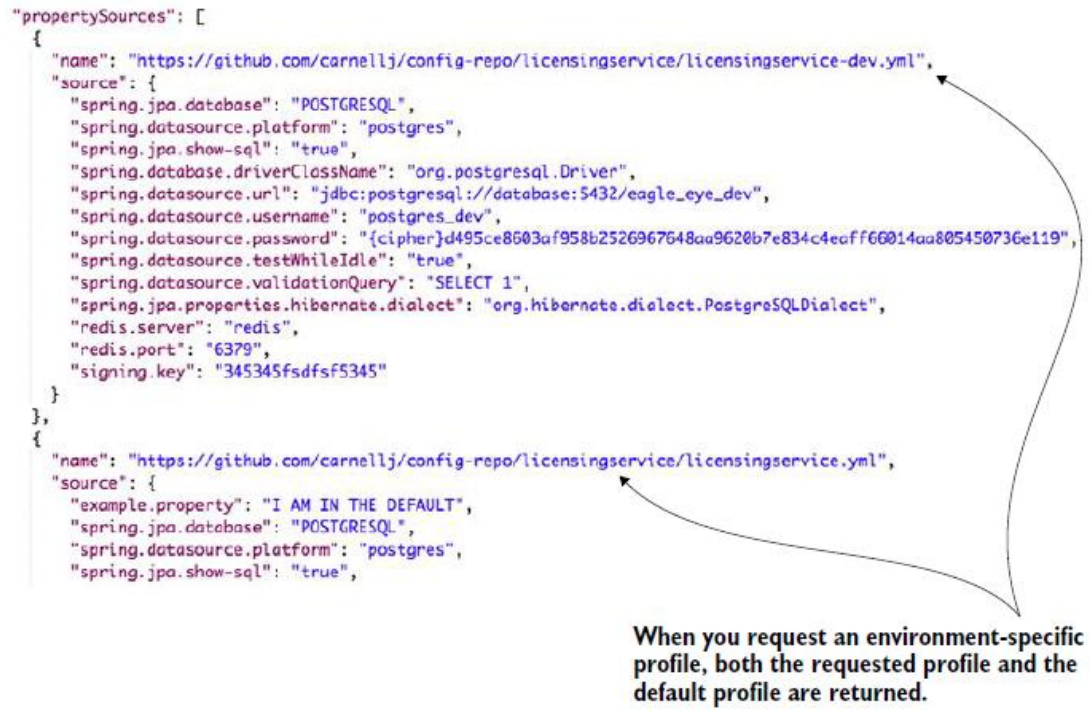


图 3.5 检索许可服务使用开发配置文件的配置信息

① *When you request an environment-specific profile, both the requested profile and the default profile are returned.*

当你请求一个特定环境的概要文件时，概要文件和默认配置文件都返回。

具体而言，如果你在 licensingservice.yml 文件定义一个属性和不能确定它在任何其他环境的配置文件（例如，在 licensingservice-dev.yml），Spring 框架将使用默认值。

> ***注意***：这不是你通过直接调用 Spring Cloud 配置 REST 端点来看到的行为。REST 端点被调用时，将返回所有配置的默认值和这些配置特定于环境的值。

让我们来看看你如何将 Spring Cloud 配置服务器集成到你的许可微服务。

3.3. 将 Spring Cloud 配置与 Spring Boot 客户端集成

在前面的章节中，你为许可服务建立了一个简单的骨架，只返回一个硬编码的 Java 对象，这个对象代表从数据库获取的一条许可记录。在下面的例子中，你将增建许可服务并与存储你的许可数据的一个 Postgres 数据库进行交互。

你将使用 Spring Data 与数据库通信，并将从许可表中获取的数据映射为一个存储数据的 POJO 对象。你的数据库连接和一个简单属性将从 Spring Cloud 配置服务器读取。图 3.6 显示了许可服务和 Spring Cloud 配置服务之间会发生什么。

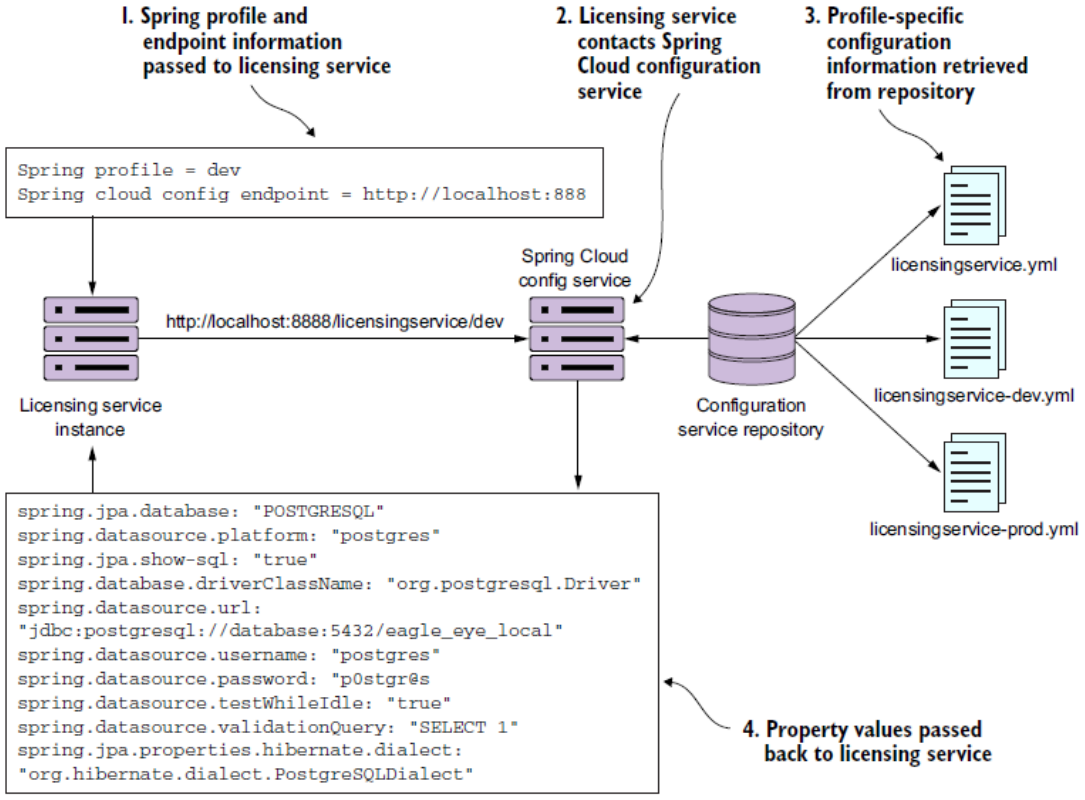


图 3.6 使用 dev 概要文件检索配置信息

① *Spring profile and endpoint information passed to licensing service*

传递给许可服务的 Spring 概要文件和端点信息

② *Licensing service contacts Spring Cloud configuration service*

许可服务与 Spring Cloud 配置服务联系

③ *Profile-specific configuration information retrieved from repository*

从存储库中检索的特定概要文件的配置信息

④ *Property values passed back to licensing service*

返回到许可服务的属性值。

当许可服务第一次启动时，你将通过命令行传递两条信息：Spring 概要文件和许可服务与 Spring Cloud 配置服务进行通信应该使用的端点。Spring 概要值映射为 Spring 服务检索的环境属性。当许可服务第一次启动时，它将通过一个通过 Spring 概要文件传入和构建的端点与 Spring Cloud 配置服务联系。Spring Cloud 配置服务将使用配置后端配置存储库（文件系统、Git、Consul、Eureka）检索 URI 中传递的 Spring 概要文件值的特定配置信息。然后将适当的属性值传递回许可服务。然后，Spring Boot 框架将这些值注入应用程序的适当部分。

3.3.1. 配置 Spring Cloud Config Server 的依赖

让我们将焦点从配置服务器转为许可服务。你需要做的第一件事是在你的许可服务 Maven 文件中添加几项。需要添加的条目如下面的列表所示。

清单 3.4 许可服务需要的额外 Maven 依赖

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

<dependency>
   <groupId>postgresql</groupId>
   <artifactId>postgresql</artifactId>
   <version>9.1-901.jdbc4</version>
</dependency>

<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-config-client</artifactId>
```

①告诉 Spring Boot 你要在服务使用 java 持久化 API（JPA）

②告诉 Spring Boot 拉下 Postgres 的 JDBC 驱动程序

③告诉 Spring Boot，你应该拉下 Spring Cloud Config 客户端的依赖项。

</dependency>

第一和第二个依赖，spring-boot-starter-data-jpa 和 PostgreSQL,引入 Spring Data Java 持久化 API（JPA）和 Postgres 的 JDBC 驱动程序。最后一个依赖项，spring-cloud-config-client，它包含与 Spring Cloud 配置服务器交互所需的所有类。

3.3.2. Spring Cloud Config 使用配置

在 Maven 的依赖已被定义，你需要告诉许可服务联系 Spring Cloud 配置服务器的地址。在一个使用 Spring Cloud Config 的 Spring Boot 服务中，配置信息可以设置在其中一个配置文件：bootstrap.yml 和 application.yml。

在任何其他配置信息使用之前，bootstrap.yml 文件读取应用程序的属性。总的来说，bootstrap.yml 文件包含服务的应用程序的名称，应用程序概要文件，和连接到 Spring Cloud 配置服务器的 URI。任何你想保留在本地的服务配置信息（而不是存储在 Spring Cloud Config）可以设置在本地服务的 application.yml 文件。通常，你存储在 application.yml 文件的配置数据，是你可能想即使 Spring Cloud 配置服务不可用也有一个可用的服务。bootstrap.yml 和 application.yml 文件都存储在一个项目 src/main/resources 目录。

为了使许可服务与 Spring Cloud 配置服务通信，你需要添加一个 licensing-service/src/main/resources/bootstrap.yml 文件和设置三个属性：spring.application.name， spring.profiles.active， 和 spring.cloud.config.uri。

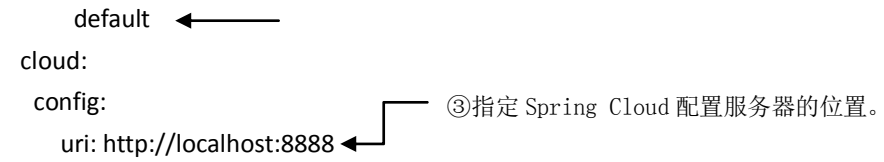
许可服务项目的 bootstrap.yml 文件如下面清单中所示。

清单 3.5 配置许可服务 bootstrap.yml

```
spring:
  application:
     name: licensingservice
  profiles:
     active:
```

①指定许可服务的名称，以便 Spring Cloud 配置客户端知道正在查找哪个服务。

②指定服务应该运行的默认配置文件。配置映射到环境。



注意：Spring Boot 应用程序支持两种机制来定义一个属性：YAML（Yet another Markup Language）和一个“.”分隔的属性名称。我们选择 YAML(Yet Another Markup Language)为手段配置应用程序。YAML 的属性值直接映射到 spring.application.name, spring.profiles.active, 和 spring.cloud.config.uri names 的分层格式。

spring.application.name 是应用程序的名称（例如，licensingservice）必须直接映射到在 Spring Cloud 配置服务器该目录的名称。以许可服务为例，在 Spring Cloud 配置服务器的目录名称为 licensingservice。

第二个属性，spring.profiles.active，用来告诉 Spring Boot 的应用程序应该运行什么文件。配置文件是一种机制，用来区分 Spring Boot 应用程序所消费的配置数据。对于许可服务的配置文件，你将支持将服务直接映射到云配置环境中的环境。例如，通过将 dev 作为我们的配置传递，Spring Cloud 配置服务器将使用 dev 属性。如果设置了配置文件，许可服务将使用默认配置文件。

第三个，即最后一个属性 spring.cloud.config.uri，这个位置信息是许可服务查找 Spring Cloud 配置服务器的端点。默认情况下，许可服务将在 http://localhost:8888 查找配置服务器。在本章后面，你将看到在应用启动时，如何覆盖定义在 boostrap.yml 和 application.yml 文件不同的属性。这将允许你告诉许可微服务应该运行在那个环境。

现在，如果你启动 Spring Cloud 配置服务，与相应的 Postgres 数据库在本地机器上运行，你可以使用它的默认配置文件启动许可服务。这是通过切换到 licensing-services 目录并发出以下命令完成的：

mvn spring-boot: run

通过运行这个命令而没有任何属性设置，许可服务器会自动尝试使用端点(http://localhost:8888)连接到 Spring Cloud 配置服务器和在许可服务的 bootstrap.yml 文件中定义的激活状态的概要文件（默认）。

如果你想覆盖这些默认值和指向到另一个环境，你可以通过编译 licensingservice 项目为 JAR 文件，然后用-D 系统属性覆盖 运行 JAR。下面的命令行演示如何使用非默认配置文件启动许可服务：

```
java -Dspring.cloud.config.uri=http://localhost:8888 \
     -Dspring.profiles.active=dev \
     -jar target/licensing-service-0.0.1-SNAPSHOT.jar
```

之前的命令行，最重要的两个参数：spring.cloud.config.uri 和 spring.profiles.active。系统属性-Dspring.cloud.config.uri=http://localhost:8888，指向一个脱离本地运行的配置服务器。

注意：如果你尝试从你的桌面使用之前的 Java 命令，运行从 GitHub 库（https://github.com/carnellj/spmia-chapter3）下载的许可服务，它会失败，因为你没有本地没有运行的一个 Postgres 服务器，并且在 GitHub 库的源代码在配置服务器上是使用加密的。我们将在本章稍后讨论使用加密。前面的示例演示如何通过命令行覆盖 Spring 属性。

使用-Dspring.profiles.active=dev 系统属性，你告诉许可服务使用 dev 配置文件（从配置服务器读取）连接到数据库的 dev 实例。

使用环境变量传递启动信息

在示例中，你硬编码值传递为-D 参数的值。在云中，你需要的大部分应用程序配置数据将在配置服务器中。然而，为了信息你需要启动你的服务（如配置服务器的数据），你会启动 VM 实例或 Docker 容器和环境变量传递。

每一章所有的代码例子，都可以完全运行在 Docker 容器。在 Docker，你模拟不同环

境通过特定环境下的 Docker-compose 文件，编排你所有的服务启动。容器所需的特定环境值作为环境变量传递到容器中。例如，在 dev 环境中启动许可服务，例如，在开发环境中启动许可证服务，docker/dev/docker-compose.yml 文件包含 licensing-service 的下列条目：

```
licensingservice:
    image: ch3-thoughtmechanix/licensing-service
    ports:
      - "8080:8080"
    environment:
      PROFILE: "dev"
      CONFIGSERVER_URI: http://configserver:8888
      CONFIGSERVER_PORT: "8888"
      DATABASESERVER_PORT: "5432"
```

①指定 licensing-service 服务容器的启动环境变量。

②概要文件环境变量传递到 Spring Boot 服务命令行，并告诉 Spring Boot 应该运行什么配置文件。

③配置服务的端点

文件中的环境条目包含两个变量概要文件的值，这就是许可服务将在下面运行的 Spring Boot 概要文件。CONFIGSERVER_URI 被传入许可服务和定义 Spring Cloud 配置服务器实例，该服务将从 Spring Cloud 配置服务器读取它的配置数据。

在启动脚本（它运行在容器中的）里，你传递这些环境变量作为-D 参数到 JVM 来启动应用程序。在每一个项目，你制作一个 Docker 容器，而 Docker 容器使用启动脚本启动容器内的软件。对于许可服务，制作到容器的启动脚本将被发现在 licensing-service/src/main/docker/run.sh。在 run.sh 脚本，以下进入启动你的 licensing-service JVM：

```
echo "********************************************************"
echo "Starting License Server with Configuration Service :
      $CONFIGSERVER_URI";
echo "********************************************************"
java    -Dspring.cloud.config.uri=$CONFIGSERVER_URI
-Dspring.profiles.active=$PROFILE    -jar /usr/local/licensingservice/
      licensing-service-0.0.1-SNAPSHOT.jar
```

因为通过 Spring Boot Actuator 增强了所有具有自省功能的服务，所以你可以通过点击 http://localhost:8080/env 来确认你正在运行的环境。/env 端点将提供关于服务的配置

信息的完整列表，包括服务启动的属性和端点，如图 3.7 所示。

关键还是注意图 3.7，许可服务的激活概要文件是 dev。通过检查返回的 JSON，你也可以看到，Postgres 数据库返回的是一个开发 URI：jdbc:postgresql://database:5432/eagle_eye_dev。


```
{
  "profiles": [
    "default"
  ],
  "server.ports": {
    "local.server.port": 8080
  },
  "decrypted": {
    "spring.datasource.password": "******"
  },
  "configService:configClient": {
    "config.client.version": "8907411ed638d7a66e2ae4142f83671425f4113f"
  },
  "configService:https://github.com/carnellj/config-repo/licensingservice/licensingservice.yml": {
    "example.property": "I AM IN THE DEFAULT",
    "spring.jpa.database": "POSTGRESQL",
    "spring.datasource.platform": "postgres",
    "spring.jpa.show-sql": "true",
    "spring.database.driverClassName": "org.postgresql.Driver",
    "spring.datasource.url": "jdbc:postgresql://database:5432/eagle_eye_local",
    "spring.datasource.username": "postgres",
    "spring.datasource.password": "******",
    "spring.datasource.testWhileIdle": "true",
    "spring.datasource.validationQuery": "SELECT 1",
    "spring.jpa.properties.hibernate.dialect": "org.hibernate.dialect.PostgreSQLDialect",
    "redis.server": "redis",
    "redis.port": "6379",
    "signing.key": "******"
  },
```


图 3.7 许可服务加载的配置可以通过调用/env 端点来检查。

暴露太多信息

每个组织对于如何在他们的服务上实现安全都有不同的规则。许多组织认为服务不应该播放关于他们自己的信息，并且不允许像/env 端点那样在服务上活动，因为他们相信（这是理所当然的），这将为潜在的黑客提供太多的信息。Spring Boot 提供了丰富的功能，说明如何配置 Spring Actuators 端点返回的信息，这些信息超出了本书的范围。Craig Walls 的优秀书籍，《Spring Boot 实战》，详细介绍了这一主题，我强烈建议你审查你的企业安全策略和 Walls 的书提供了你想通过 Spring Actuator 暴露的正确的细节层次。

3.3.3. 使用 Spring Cloud Configuration Server 配置数据源

在这一点上，你的数据库配置信息被直接注入到你的微服务。设置数据库的配置，配置许可微服务成为使用标准 Spring 组件构建，从 Postgres 数据库检索数据的练习。许可服务已被重构为不同的类，每个类有独立的职责。这些类显示在表 3.2。

表 3.2 许可服务所有类和位置

类名	位置
License	licensing-service/src/main/java/com/thoughtmechanix/licenses/model
LicenseRepository	licensing-service/src/main/java/com/thoughtmechanix/licenses/repository
LicenseService	licensing-service/src/main/java/com/thoughtmechanix/licenses/services

License 类是保留从 licensing 数据库检索的数据的模型类。下面的清单显示了 License 类的代码。

清单 3.6 单条 license 记录 JPA 模型代码

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

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity                                          ①@Entity 告诉 Spring 这是一个 JPA 类。
@Table(name = "licenses")                        ②@Table 映射到数据库表。
public class License{
  @Id                                            ③@Id 将此字段标记为主键。
  @Column(name = "license_id", nullable = false) ④@Column 将字段映射到特定的数据库表。
  private String licenseId;
```

```
    @Column(name = "organization_id", nullable = false)
    private String organizationId;

    @Column(name = "product_name", nullable = false)
    private String productName;
}
```

该类使用几个 Java 持久化注解（JPA）帮助 Spring Data 框架将数据从 Postgres 数据库 licenses 表映射到 Java 对象。@Entity 注解让 Spring 知道这个 Java POJO 将被从存储数据映射成对象。@Table 注解告诉 Spring/JPA 什么数据库表应该映射。@Id 注解标识数据库的主键。最后，数据库中的每一列将要映射到都使用@Column 属性进行标记的单个属性。

Spring Data 和 JPA 框架提供了用于访问数据库基本的 CRUD 方法。如果你想在此之外构建方法，可以使用 Spring Data 存储库接口和基本命名约定来构建这些方法。在启动时，Spring 将从存储库接口解析方法的名称，将它们转换成基于名称的 SQL 语句，然后生成动态代理类，并在其遮掩下来完成工作。许可服务的存储库如下所示。

清单 3.7 定义查询方法的 LicenseRepository 接口

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

import com.thoughtmechanix.licenses.model.License;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface LicenseRepository
    extends CrudRepository<License,String>
{
    public List<License> findByOrganizationId
    ➥(String organizationId);
    public License findByOrganizationIdAndLicenseId
    ➥(String organizationId,String licenseId);
}
```

①告诉 Spring Boot，这是一个 JPA 存储库类

②定义其继承自 Spring 的 CrudRepository 类

③单个查询方法由 Spring 解析为 SELECT…FROM 查询中。

LicenseRepository 仓库接口，标有@Repository 注解告诉 Spring，它应该把这个接

口作为一个存储库和为它生成一个动态代理。Spring 提供了不同类型的数据访问仓库类。你可以选择使用 Spring CrudRepository 基类来扩展你的 LicenseRepository 类。CrudRepository 基类包含有基本的 CRUD 方法。除了 CRUD 方法从 CrudRepository 扩展，你已经增加了两个自定义查询方法从 licensing 表中检索数据。Spring Data 框架将把方法的名称分开，以构建一个查询来访问底层数据。

> ***注意***：Spring Data 框架提供了各种数据库平台上的抽象层，而不仅仅局限于关系数据库。NoSQL 数据库，如 MongoDB 和 Cassandra 也支持。

与第 2 章许可服务的前身不同，现在你已经将 licensing 服务的业务和数据访问逻辑分开成 LicenseController 和一个类名叫做 LicenseService 的单独服务。

清单 3.8 用于执行数据库命令的 LicenseService 类

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

import com.thoughtmechanix.licenses.config.ServiceConfig;
import com.thoughtmechanix.licenses.model.License;
import com.thoughtmechanix.licenses.repository.LicenseRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.UUID;

@Service
public class LicenseService {

    @Autowired
    private LicenseRepository    licenseRepository;

    @Autowired
    ServiceConfig    config;

    public License getLicense(String organizationId,String licenseId) {
        License license = licenseRepository.findByOrganizationIdAndLicenseId(
organizationId, licenseId);
        return license.withComment(config.getExampleProperty());
    }
```

```
    public List<License> getLicensesByOrg(String organizationId){
        return licenseRepository.findByOrganizationId( organizationId );
    }

    public void saveLicense(License license){
        license.withId( UUID.randomUUID().toString());
        licenseRepository.save(license);
     }
}
```

控制器，服务和存储库类被使用标准的 Spring @Autowired 注解组合在一起。

3.3.4. 使用@Value 注解直接读取属性

在上一节中的 LicenseService 类，你可能会注意到在 getLicense()代码中使用从 config.getExampleProperty()获取的值来设置 license.withComment()的值。此处涉及的代码如下所示：

```
public License getLicense(String organizationId,String licenseId) {
        License license = licenseRepository.findByOrganizationIdAndLicenseId(organizationId, licenseId);
        return license.withComment(config.getExampleProperty());
}
```

如果你看看 licensing-service/src/main/java/com/thoughtmechanix/licenses/config /ServiceConfig.java 类，你将看到一个带有@Value 的属性注解。下面的清单显示正在使用的@Value 注解。

清单 3.9 ServiceConfig 用来集中应用属性

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

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class ServiceConfig{

   @Value("${example.property}")
   private String exampleProperty;

   public String getExampleProperty(){
```

```
        return exampleProperty;
    }
}
```

当 Spring Data“自动神奇”地将数据库的配置数据注入数据库连接对象时，所有其他属性都必须使用@Value 注解来注入。在之前的例子，@Value 注解从 Spring Cloud 配置服务器获取 example.property 值并其注入到 ServiceConfig 类的 example.property 属性。

> ***提示***：虽然可以直接将配置值注入到单个类的属性中，我发现将所有配置信息集中到一个配置类中很有用，然后将配置类注入到需要的地方。

3.3.5. 通过 Git 使用 Spring Cloud Config Server

如前所述，使用文件系统作为 Spring Cloud 配置服务器的后端存储库对于基于云的应用程序是不切实际的，因为开发团队必须建立并管理安装在云配置服务器所有实例上的共享文件系统。

Spring Cloud 配置服务器集成了不同的后端存储库，它可用于宿主应用程序配置属性。我成功使用的一个方法是使用带有一个 Git 源代码管理存储库的 Spring Cloud 配置服务器。

通过使用 Git，你可以获得将配置管理属性置于源代码控制之下的所有好处，并提供一种简单的机制来集成你的构建和部署管道中的属性配置文件的部署。

使用 Git，你把文件系统的后置配置换出到以下列出的服务的 bootstrap.yml 文件。

清单 3.10 Spring Cloud config bootstrap.yml

```
server:
  port: 8888
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/carnellj/config-repo/
          searchPaths: licensingservice,organizationservice
          username: native-cloud-apps
          password: 0ffended
```

①告诉 Spring Cloud 配置使用 Git 作为后端存储库

②告诉 Spring Cloud 配置到 Git 服务器和 Git 仓库的 URL

③告诉 Spring Cloud 配置在 Git 中查找配置文件是那个路径

在前面的例子中配置的三个主要部分是 spring.cloud.config.server，spring.cloud.config.server.git.uri 和 spring.cloud.config.server.git.searchPaths 属性。spring.cloud.config.server 属性告诉 Spring Cloud 配置服务器使用非基于文件系统的后端存储仓库。在前面的例子中，你将连接到基于云的存储库，GitHub。

spring.cloud.config.server.git.uri 属性提供你正在连接的存储库的 URL。最后，spring.cloud.config.server.git.searchPaths 属性告诉 Spring Cloud 配置服务器在 Git 仓库上的相对路径，它在 Spring Cloud 配置服务器启动之后将可以搜索。与配置文件系统版本类似，spring.cloud.config.server.git.seachPaths 的属性值将为配置服务托管的每个服务提供一个逗号分隔的列表。

3.3.6. 通过 Spring Cloud Config Server 刷新属性

开发团队在使用 Spring Cloud 配置服务器时遇到的第一个问题是，当属性更改时，它们如何动态刷新应用程序。Spring Cloud 配置服务器将始终服务于最新版本的属性。通过底层存储库对属性做出的更改将是最新的。

然而，Spring Boot 应用程序只在启动时读取它们的属性，所以 Spring Cloud 配置服务器中所做的属性更改不会被 Spring Boot 应用程序自动拾取。Spring Boot Actuator 确实提供了一个@RefreshScope 注解，将允许开发团队访问/refresh 端点，将迫使 Spring Boot 应用程序重新读取它的应用配置。下面的清单显示在实战中使用@RefreshScope 注解。

清单 3.11 @RefreshScope 注解

```
package com.thoughtmechanix.licenses;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.context.config.annotation.RefreshScope;
```

```
@SpringBootApplication
@RefreshScope
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

注意一些关于@RefreshScope 注解的东西。首先，注解只会重新加载应用程序配置中的自定义 Spring 属性。如你的数据库配置，使用@RefreshScope 注解标注，Spring Data 不会重载的。执行刷新，你可以打开 http://<yourserver>:8080/refresh 端点。

刷新微服务

使用 Spring Cloud 配置服务微服务的时候，有一件事你需要在你的动态变化特性之前考虑，你可能有相同服务的多个实例运行，你需要使用它们新的应用程序配置刷新所有这些服务。有几种方法可以解决这个问题：

Spring Cloud 配置服务确实提供了一种基于“推”的机制，称为 Spring Cloud Bus，它允许 Spring Cloud 配置服务器使用发生更改的服务向所有客户端发布。Spring Cloud 配置需要一个额外的，运行的中间件（RabbitMQ）。这是一个非常有用的检测变化的手段，但不是所有的 Spring Cloud 配置的后端支持都“推”机制（即 Consul 服务器）。

在下一章中，您将使用 Spring 服务发现和 Eureka 来注册服务的所有实例。我用来处理应用程序配置刷新事件的一种技术是在 Spring Cloud 配置中刷新应用程序属性，然后编写一个简单的脚本来查询服务发现引擎，查找服务的所有实例，并直接调用/refresh 端点。

最后，您可以重新启动所有服务器或容器以获取新属性。这是一个微不足道的练习，尤其是如果在一个容器服务运行你的服务，如 Docker。重新启动 Docker 容器只需要几秒并将强制重读应用程序的配置。

记住，基于云的服务器是短暂的。不要害怕使用新配置启动新的服务实例，对新服务的

直接通信，然后删除旧的服务。

3.4. 保护敏感配置信息

默认情况下，Spring Cloud 配置服务器在应用程序的配置文件中以纯文本格式存储所有属性。这包括诸如数据库凭据之类的敏感信息。

将敏感凭据存储为源代码存储库中的纯文本是极其糟糕的做法。不幸的是，这种情况比你想象的要频繁得多。Spring Cloud Config 使您能够轻松地加密敏感属性。Spring Cloud Config 支持使用对称（共享秘密）和非对称加密（公共/私钥）。

我们将了解如何设置 Spring Cloud 配置服务器，以便使用对称密钥进行加密。要做到这一点，你需要：

- 下载并安装加密所需的 Oracle JCE JAR 包。
- 设置加密密钥。
- 加密和解密属性。
- 配置微服务使用加密的客户端。

3.4.1. 下载和安装加密所需的 Oracle JCE JAR 包

首先，你需要下载并安装 Oracle 的无长度限制的 Java 加密扩展（JCE）。这不能通过 Maven 必须 Oracle 公司下载。[1]一旦你已经下载了 ZIP 文件包含 JCE jars，你必须做到以下几点：

- 找到你的$JAVA_HOME/jre/lib/security 目录。
- 备份在$JAVA_HOME/jre/lib/security 目录的 local_policy.jar 和 US_export_policy.jar 文件到另一个位置。

[1] http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html. 此 URL 可能会更改。在 Google 快速搜索 Java 加密扩展，应该返回给你正确的值。

- 解压从 Oracle 下载 JCE zip 文件。
- 复制 local_policy.jar 和 US_export_policy.jar 到$JAVA_HOME/jre/lib/security 目录。
- 配置 Spring Cloud Config 以使用加密。

自动安装 Oracle 的 JCE 文件的过程

我讨论过你需要在你的笔记本上安装 JCE 的手动步骤。因为我们使用 Docker 创建我们的服务作为 Docker 容器，在 Spring Cloud Config Docker 容器我们已经编写下载和安装这些 JAR 文件的脚本。下面的 OS X 操作系统的 shell 脚本片段展示我如何自动使用 curl 命令行工具（https://curl.haxx.se/）：

```
cd    /tmp/
curl    –k-LO    "http://download.oracle.com/otn-pub/java/jce/8/jce_policy-8.zip"
         -H    'Cookie: oraclelicense=accept-securebackup-cookie'    && unzip jce_policy-8.zip
rm    jce_policy-8.zip
yes    |cp -v /tmp/UnlimitedJCEPolicyJDK8/*.jar /usr/lib/jvm/java-1.8-openjdk/jre/lib/security/
```

我不想讨论所有的细节，但基本上我使用 CURL 来下载 JCE 的 ZIP 文件（注意 Cookie 头参数是通过 curl 命令的-H 属性传递的），然后解压文件并将它们复制到 Docker 容器的 /usr/lib/jvm/java-1.8-openjdk/jre/lib/security 目录。

如果你看看这章的源代码 src/main/docker/Dockerfile 文件，你可以看到这个脚本实战的一个例子。

3.4.2. 配置加密密钥

一旦 JAR 文件就位，您就需要设置一个对称加密密钥。对称加密密钥无非是一个共享的秘密，被用作加密者来加密值和用作解密者来解密值。在 Spring Cloud 配置服务器，对称加密密钥是一个你选择的字符串，它通过操作系统环境变量 ENCRYPT_KEY 传递给服务。为了本书的目的，你将总是设置环境变量 ENCRYPT_KEY 为：

export ENCRYPT_KEY=IMSYMMETRIC

注意两个关于对称密钥的问题：

- 你的对称密钥应该是 12 个或更大长度的字符，最好是一组随机字符。
- 不要丢失对称密钥。一旦你已经使用加密密钥加密一些东西，你不能解密它。

加密密钥管理

为了这本书的目的，我做了两件我通常不会推荐在生产中部署的事情：

- 我将加密密钥设置为一个短语。我想保持密钥简单，这样我就能记住它，而且很适合作为文本阅读。在真实的部署中，我将为我部署的每个环境使用一个单独的加密密钥，并使用随机字符作为密钥。
- 在本书，我直接在 Docker 文件中硬编码 ENCRYPT_KEY 环境变量。我这样做是为了让读者可以下载这些文件并在不需要记住设置环境变量的情况下启动这些文件。在实际的运行环境 ,我会在我的 Dockerfile 里面引用 ENCRYPT_KEY 作为操作系统环境变量。要知道不要在你的 Dockerfiles 里面硬编码你的加密密钥。记住，你的 Dockerfiles 应该在源代码下控制。

3.4.3. 加密和解密属性

现在可以开始加密属性以便在 Spring Cloud Config 中使用了。你会加密许可服务 Postgres 数据库密码，你已经使用它访问 EagleEye 数据。这个属性称为 spring.datasource.password，目前设置为纯文本的值 p0stgr@s。

当你启动 Spring Cloud Config 实例，Spring Cloud Config 检测到 ENCRYPT_KEY 环境变量和自动添加了两个新的端点(/encrypt 和 /decrypt)到 Spring Cloud 配置服务。你将使用/encrypt 端点加密 p0stgr@s 值。

① *The value we want to encypt*

我们想加密的值

② *The encrypted result*

加密的结果

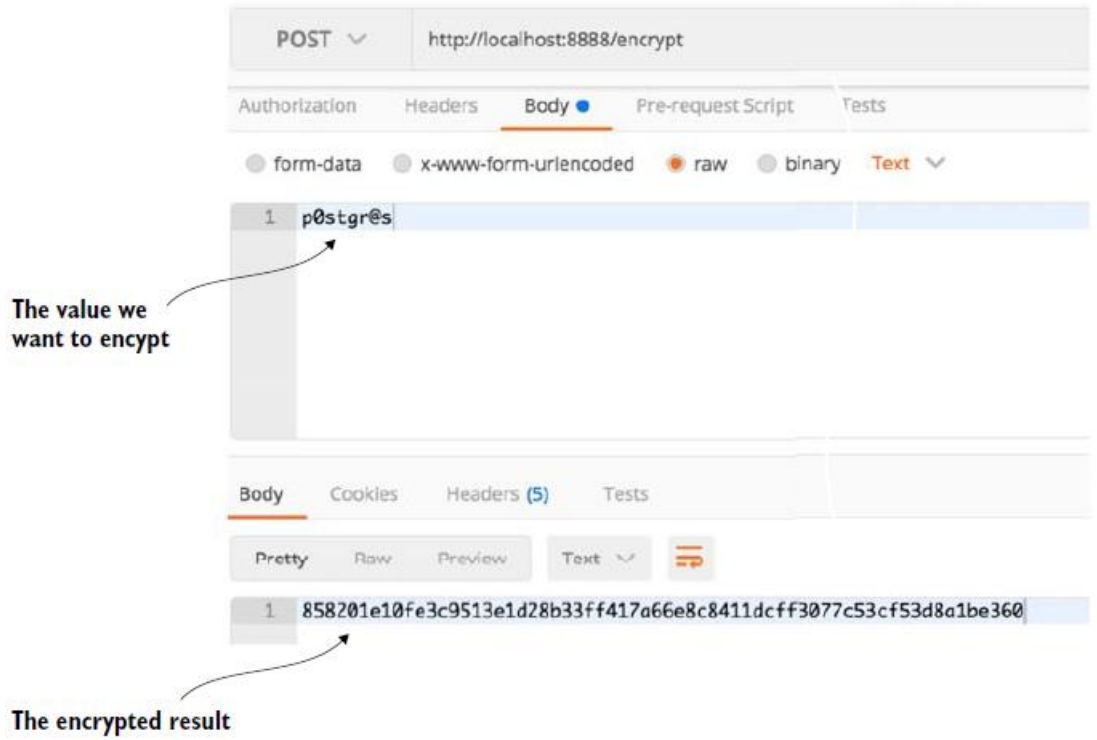


图 3.8 使用/encrypt 端点你可以加密值。

图 3.8 显示了如何使用/encrypt 和 POSTMAN 加密 p0stgr@s 值。请注意，无论什么时候调用/encrypt 或/decrypt 端点，都需要确保你对这些端点执行了一个 POST 请求。

如果你想解密该值，你将使用在调用中的加密字符串中传递的/decrypt 端点。

现在你可以许可服务使用以下语法，添加加密属性到你的 GitHub 活基于文件系统的配置文件：

```
spring.datasource.password:"{cipher}
    858201e10fe3c9513e1d28b33ff417a66e8c8411dcff3077c53cf53d8a1be360"
```

Spring Cloud 配置服务器要求所有的加密属性都使用{cipher}前缀。{cipher}值告诉 Spring Cloud 配置服务器，它将处理一个加密值。启动 Spring Cloud 配置服务器并以 GET

方式点击 http://localhost:8888/licensingservice/default 端点。

图 3.9 显示了这个调用的结果。

你已经通过使用加密属性使 spring.datasource.password 更安全，但你仍然有一个问题。当你点击 http://localhost:8888/licensingservice/default 端点时，数据库密码以纯文本暴露。

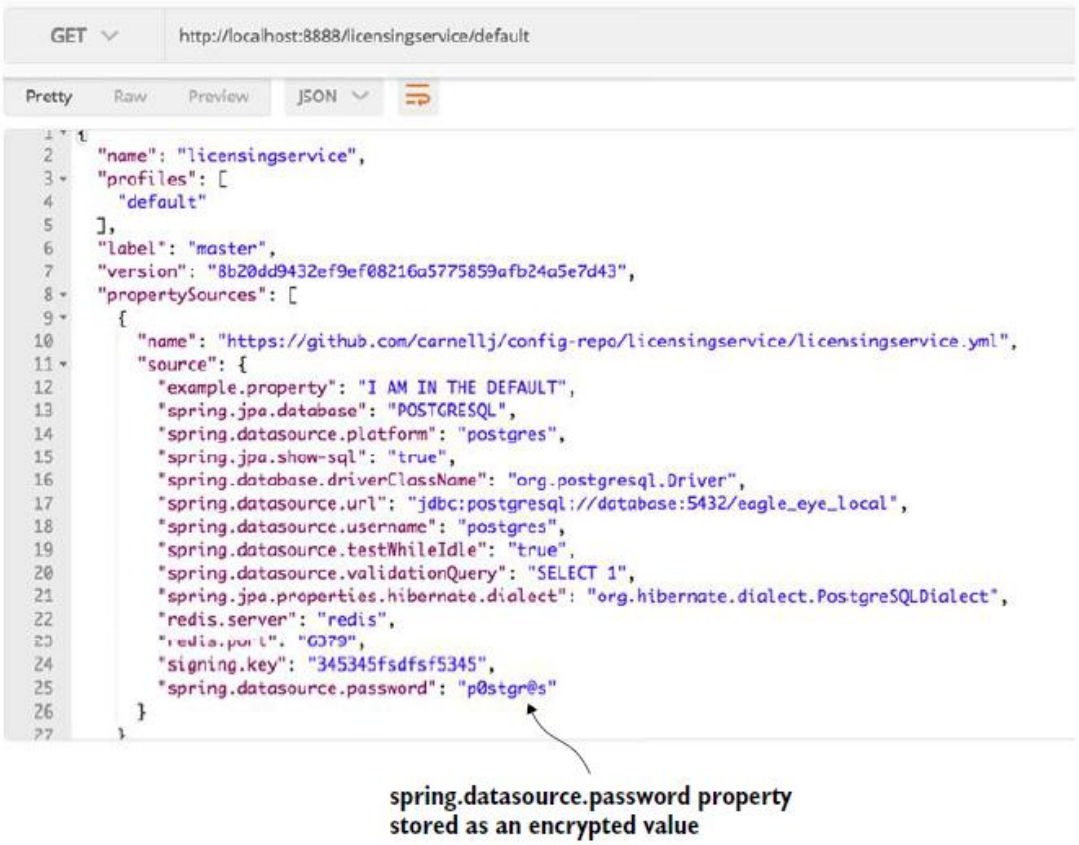


图 3.9 尽管 spring.datasource.password 在属性文件中被加密，但当许可服务检索配置时它被解密。这仍然是个问题。

① *spring.datasource.password property stored as an encrypted value*

spring.datasource.password 属性作为一个加密的值存储

默认情况下，Spring Cloud Config 将所有在服务器上的属性解密并把结果作为明文（未加密的文本）返回给应用程序消费属性。但是，你可以告诉 Spring Cloud Config 在服务器上不进行解密，使应用程序检索配置数据并解密加密属性由应用程序负责。

3.4.4. 配置微服务使用加密的客户端

要启用客户端对属性的解密，你需要做三件事：

- 配置 Spring Cloud Config 在服务器端不解密属性。
- 设置许可服务器的对称密钥。
- 在许可服务的 pom.xml 文件中添加与 spring-security-rsa 相关的 JAR 包。

你需要做的第一件事是禁用服务器端在 Spring Cloud Config 中解密的属性。通过配置 Spring Cloud Config 的 src/main/resources/application.yml 文件来设置属性 pring.cloud.config.server.encrypt.enabled: false。这就是你在 Spring Cloud 配置服务器上必须做的事情。

因为许可服务现在负责解密加密的属性，你首先需要在许可服务设置对称密钥，确保 ENCRYPT_KEY 环境变量被设置为具有相同的对称密钥(例如, IMSYMMETRIC)，对称密钥与你在你的 Spring Cloud 服务器上使用的相同。

接下来，你需要在许可服务中包含 spring-security-rsa JAR 依赖项：

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-rsa</artifactId>
</dependency>
```

这些 JAR 文件包含解密从 Spring Cloud Config 取回的加密属性所需的 Spring 代码。有了这些更改，您就可以启动 Spring Cloud Config 和许可服务了。如果你点击 http://localhost:8888/licensingservice/default 端点，你将看到 spring.datasource.password 以加密形式返回。图 3.10 显示了调用的输出。

① The spring.datasource.password property is encrypted.

spring.datasource.password 属性被加密。

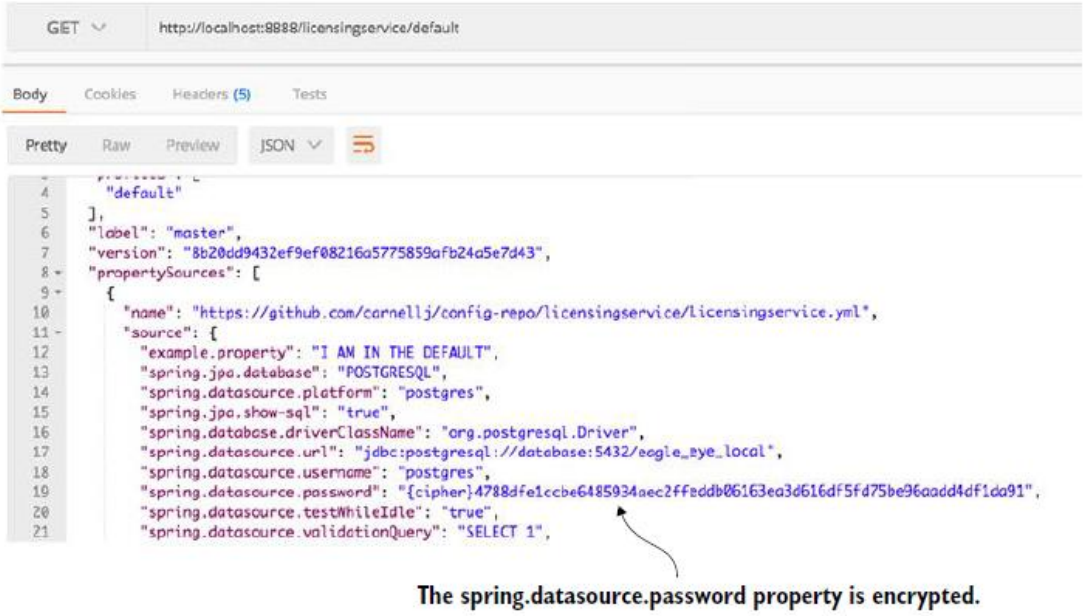


图 3.10 启用了客户端解密，敏感属性将不再从 Spring Cloud Config REST 调用以纯文本文本格式返回。相反，当调用服务从 Spring Cloud Config 加载其属性时，该属性将被解密。

3.5. 最后的思考

应用程序配置管理似乎是一个平凡的话题，但它在基于云的环境中非常重要。正如我们在后面的章节中更详细地讨论的，关键是你的应用程序和它们运行的服务器是不可变的，并且正在提升的整个服务器从不在环境之间手动配置。这与传统部署模型，你部署一个应用构件（如 JAR 或 WAR 文件）以及其属性文件部署到“固定”环境相冲突。

使用基于云的模型，应用程序配置数据应该与应用程序完全隔离，在运行时需要注入适当的配置数据，以便在所有环境中一致地升级相同的服务器/应用程序构件。

3.6. 小结

- Spring Cloud 配置服务器允许你设置具有特定于环境的应用程序属性值。

- Spring 使用 Spring 概要文件启动一个服务，以确定从 Spring Cloud 配置服务中检索到哪些环境属性。
- Spring Cloud 配置服务可以使用基于文件或基于 Git 的应用程序配置存储库来存储应用程序属性。
- Spring Cloud 配置服务允许你使用对称和非对称加密方式加密敏感属性文件。