4. 第 4 章 服务发现

本章内容

- 解释为什么服务发现对任何基于云的应用环境是重要的
- 与传统的负载均衡器方法比较，理解服务发现的优点和缺点
- 创建一个 Spring Netflix Eureka 服务器
- 使用 Eureka 注册一个基于 Spring Boot 的微服务
- 配置 Spring Cloud 和 Netflix 的 Ribbon 库来使用客户端负载均衡

在任何分布式架构中，我们都需要找到机器所在位置的物理地址。自从分布式计算开始以来，这个概念就一直存在，并被正式地称为服务发现。服务发现可以像维护应用程序所使用的所有远程服务的地址的属性文件一样简单，或者作为形式化（和复杂化）的一个 UDDI（通用描述、发现和集成）存储库。[1]

服务发现对微服务和基于云应用是极重要的。这有两个关键原因。首先，它为应用程序团队提供了快速地在环境中运行的服务实例数量按比例增加和减少的水平扩展能力。服务消费者通过服务发现从服务的物理位置抽离出来。由于服务消费者不知道实际服务实例的物理位置，所以可以从可用服务池中添加或删除新的服务实例。

这种快速扩展服务而不中断服务消费者的访问的能力是一个非常强大的概念，因为它将用于一个开发团队以构建整体的、单租户（例如，一个客户）的应用，使其远离只有添加更大、更好的硬件（垂直扩展）来添加更多的服务器（横向扩展）的更强大的方法的思维。

一个整体的方法通常是驱动开发团队走上超出他们容量需要的道路。容量以块状和尖峰的形式增加，很少是平稳的路径。微服务让我们扩展/缩小新服务实例。服务发现帮助抽象出这些部署正在远离服务消费者。

[1]https://en.wikipedia.org/wiki/Web_Services_Discovery#Universal_Description_Discovery_and_Integration

服务发现的第二个好处是它有助于增加应用程序的弹性。当微服务实例变得不健康或不可用，大多数服务发现引擎将从其内部可用服务列表中删除该实例。由于服务发现引擎将路由绕过不可用的服务，所以有问题的服务造成的损害将最小化。

我们已经了解了服务发现的好处，但是有什么大不了的呢？毕竟，我们不能使用诸如 DNS（域名服务）或负载均衡器之类的经过检验而可靠的方法来帮助服务发现吗？让我们讨论为什么它们不会与微服务应用一起工作，特别是在云计算环境中运行的微服务。

4.1. 我的服务在哪里？

每当有一个调用资源跨越多个服务器的应用程序时，它需要定位这些资源的物理位置。在非云世界中，这种服务位置解析通常通过 DNS 和网络负载均衡器的组合来解决。图 4.1 演示了这个模型。

应用程序需要调用位于组织另一部分的服务。它试图使用通用 DNS 名称伴随唯一表示应用程序试图调用的服务的路径调用服务。DNS 名称解析为一个商业的负载均衡器，如流行的 F5 负载平衡器（http://f5.com）或一个开源的负载均衡器如 HAProxy（http://haproxy.org）。

① *Application uses generic DNS and service-specific path to invoke the service*

应用程序使用通用 DNS 和特定服务的路径来调用服务

② *Load balancer locates physical address of servers hosting the service*

负载均衡器定位承载服务的服务器的物理地址

③ *Services deployed to application container running on a persistent server*

服务被部署到运行在持久服务器的应用程序容器

④ *Secondary load balancer checks on primary load balancer, and takes over if*

necessary

如有必要则在主负载均衡器基础上进行次级负载均衡器检查。

⑤ *Applications consuming services*：应用程序消费服务

Services resolution layer：服务解析层

Services layer：服务层

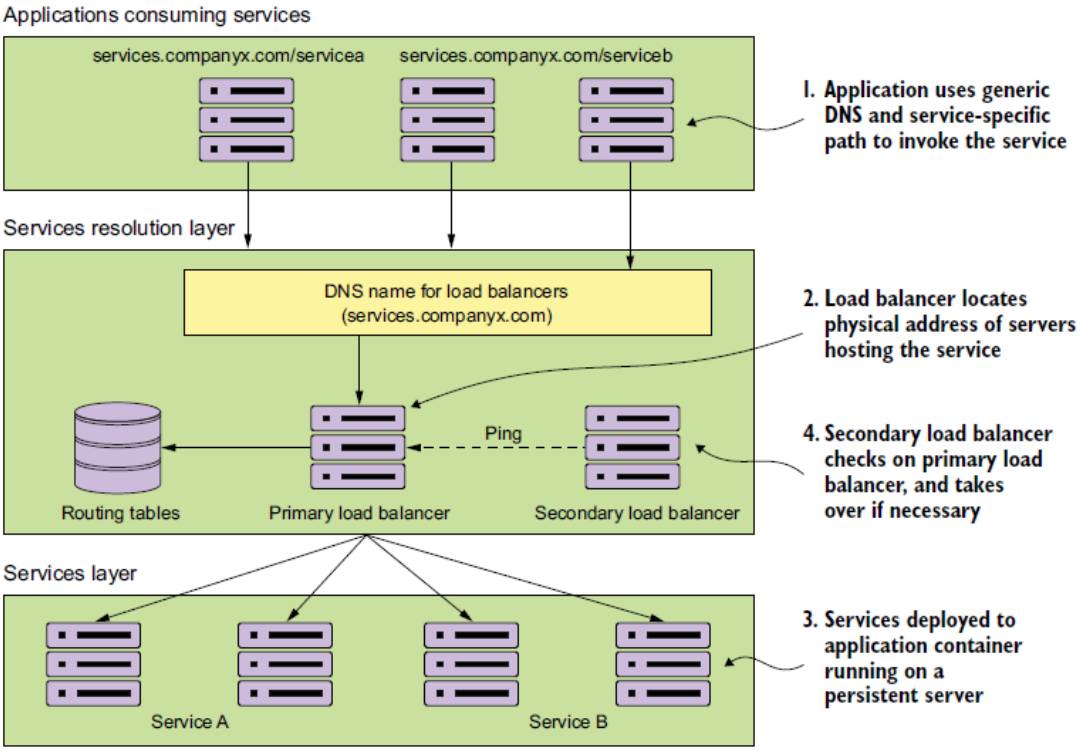


图 4.1 使用 DNS 和负载均衡器的传统服务位置解析模型

负载均衡器在接收来自服务消费者的请求后，根据用户试图访问的路径定位路由表中的物理地址条目。此路由表项包含一个或多个承载该服务的服务器的列表。然后，负载均衡器选择列表中的一个服务器，并将请求转发到该服务器。

服务的每个实例部署到一个或多个应用服务器。

这些应用程序服务器的数量往往是静态的（例如，应用程序服务器托管服务的数量没有上升和下降）和持久性（例如，如果一个运行在应用程序服务器的服务崩溃，它将恢复到崩溃时相同的状态，和之前同一 IP 和配置。）

为了实现高可用性，次级负载平衡器被设置成空闲的和 Ping 主要负载均衡器看它是否活着。如果它不存在，则次级负载均衡器变得活跃，接管主负载均衡器的 IP 地址和开始为请求提供服务。

虽然这种类型的模型适用于运行在企业数据中心的应用程序，并且在一组静态服务器上运行的服务数量相对较少，它不能很好的在基于云的微服务应用程序下工作。原因包括：

- 单点故障：虽然负载均衡器可以高度可用，但它是整个基础设施的单点故障。如果负载均衡器处理能力下降，依赖它的所有应用程序也会下降。尽管你可以使一个负载均衡器高可用，但是负载平衡器会在你的应用程序基础设施集中阻塞点而成为瓶颈。
- 有限的水平可伸缩性：通过将你的服务集成到一个单一的负载均衡集群，你获得跨越多个服务器负载均衡基础设施有限的水平扩展能力。许多商业负载均衡器由两件事情约束：冗余模型和许可证费用。大多数商业负载均衡器使用冗余的热交换模型，所以你只有一个单一的服务器来处理负载，而次级负载均衡器只有在主负载均衡器宕机的情况下才投入使用。本质上，你受制于硬件。其次，商业负载均衡器也有严格的许可模式，面向固定的容量而不是一个可变的模型。
- 静态管理：最传统的负载均衡器不是被设计为快速注册和注销服务的。它们使用集中的数据库来存储规则的路由，而添加新路由的唯一途径通常是通过供应商专有的 API（Application Programming Interface）。
- 复杂：因为负载均衡器充当服务的代理，所以服务消费者请求必须将它们的请求映射到物理服务。这个解析层经常给你的服务基础设施增加一层复杂性，因为服务的映射规则必须手工定义和部署。在传统的负载均衡器场景中，新服务实例的注册是手工完成的，而不是在新服务实例启动时进行的。

这四个原因，不是对一般的负载均衡器的控诉。它们在企业环境中工作得很好，大多数应用程序的大小和规模可以通过集中式网络基础设施处理。此外，负载平衡器仍然有在集中的 SSL 卸载和管理服务端口安全方面发挥作用。负载均衡器可以锁定入站（入口）和出站（出口）端口访问所有在它后面的服务器。在满足行业标准认证要求，如 PCI（支付卡行业）的合规性时，这种最少网络访问的概念通常是一个关键的组成部分。

然而，在云中需要处理大量交易和冗余，集中式的网络基础设施最终不能正常工作，因为它没有有效地伸缩，也没有成本效益。现在让我们看看如何为基于云的应用程序实现健壮的服务发现机制。

4.2. 关于云端服务发现的研究

一个基于云的微服务环境的解决方案是使用一个服务发现机制，即：

- 高可用性：服务发现需要能够支持“热”集群环境，在服务发现集群里服务查找可以跨多个节点共享。如果某个节点不可用，则集群中的其他节点应该能够接管。
- 对等的：服务发现集群中的每个节点共享服务实例的状态。
- 负载均衡：服务发现需要动态负载均衡请求到所有服务实例，来确保服务调用遍布所有受管的服务实例。在许多方面，服务发现取代静态，手工管理的负载均衡器使用在许多早期的 Web 应用程序的实现。
- 有弹性的：服务发现客户端应该在本地“缓存”服务信息。本地缓存允许逐步降低服务发现特性，这样，如果服务发现服务变得不可用，应用程序仍然可以基于本地缓存中维护的信息来对服务保留原始功能和进行定位。
- 容错性：服务发现需要检测服务实例不健康的情况，并将实例从可接受客户请求的可用服务列表中删除。它应该用服务来检测这些故障，并在没有人工干预的情况下

采取行动。

在下面的部分中，我们将要：

- 了解云服务发现代理如何工作的概念架构
- 显示客户端缓存和负载均衡允许服务在服务发现代理不可用时如何继续运行
- 了解使用 Spring Cloud 和 Netflix 的 Eureka 服务发现代理如何实现服务发现

4.2.1. 服务发现架构

为了展开围绕服务发现架构的讨论，我们需要理解四个概念。这些一般概念在所有服务发现实现中共享：

- 服务注册：服务如何向服务发现代理注册？
- 客户端查找服务地址：服务客户端查找服务信息的方法是什么？
- 信息共享：服务信息如何跨节点共享？
- 健康监控：服务如何将其健康状态返回到服务发现代理？

① A services location can be looked up by a logical name from the service discovery agent.

服务位置可以由服务发现代理的逻辑名称查找。

② When a service comes online it registers its IP address with a service discovery agent.

当一个服务上线时，它会用一个服务发现代理注册它的 IP 地址。

③ Service discovery nodes share service instance health information among each other.

服务发现节点彼此共享服务实例健康信息。

④ Services send a heartbeat to the service discovery agent. If a service dies, the service discovery layer removes the IP of the "dead" instance.

服务向服务发现代理发送一个心跳。如果服务死亡，服务发现层将删除“死”实例的 IP。

⑤ Client applications never have direct knowledge of the IP address of a service. Instead they get it from a service discovery agent.

客户端应用程序从来不直接知道服务的 IP 地址。相反，它们从一个服务发现代理获取。

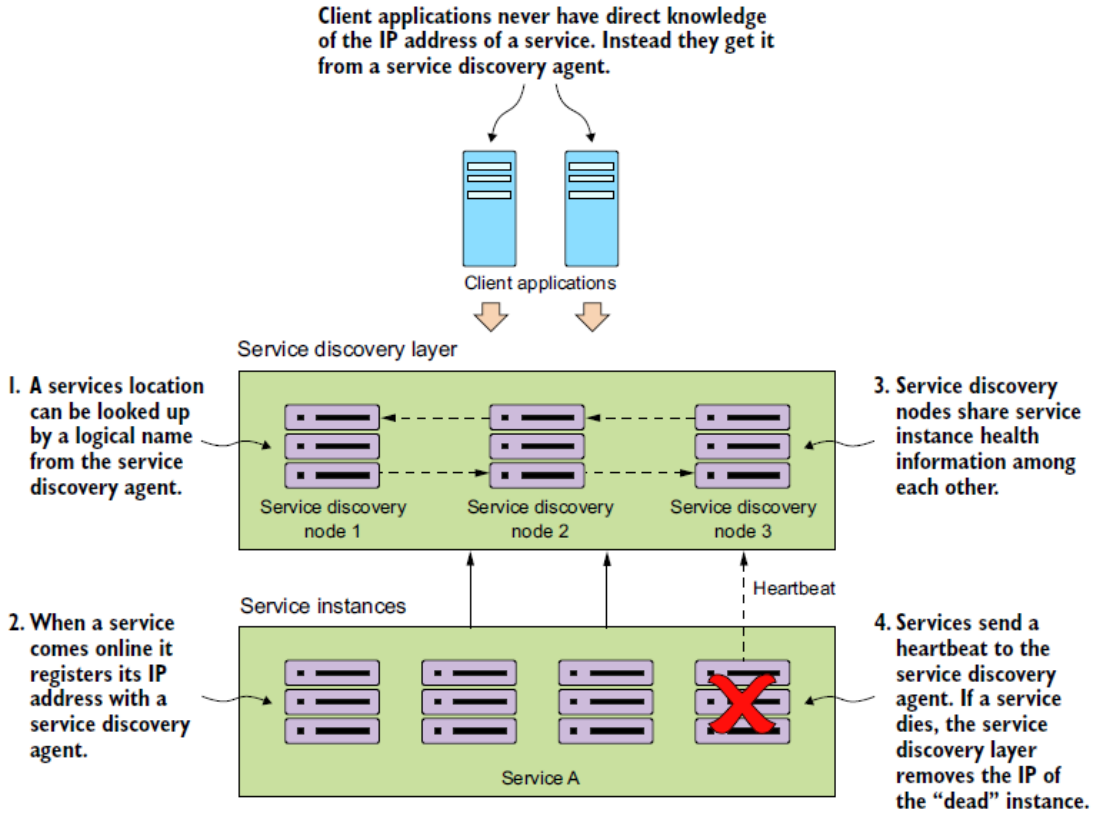


图 4.2 添加/删除服务实例时，它们将更新服务发现代理，并可用以处理用户请求。

图 4.2 显示了这四个部分的流程，以及服务发现模式实现中通常发生的情况。

在图 4.2 中，已经启动了一个或多个服务发现节点。这些服务发现实例通常是唯一的，并且没有负载均衡器位于它们前面。

当服务实例启动时，它们将通过一个或多个服务发现实例来注册它们的物理位置、路径和端口。虽然服务的每个实例都有唯一的 IP 地址和端口，但是每次启动的服务实例都会在

同一个服务 ID 下注册。服务 ID 不过是唯一标识同一个服务实例组的键。

一个服务通常只注册到一个服务发现服务实例。大多数服务发现实现使用数据传播的对等模型，其中每个服务实例中的数据被传送到集群中的所有其他节点。

根据服务发现实现，传播机制可能使用一个硬编码的服务列表来传播或使用诸如“gossip”[2]或“infection-style”[3]协议的多播协议，以允许其他节点“发现”集群中的更改。

最后，每个服务实例将通过服务发现服务推送或拉出其状态。任何未能返回健康检查的服务将从可用服务实例池中删除。

一旦服务注册到服务发现服务，它就可以由需要使用其功能的应用程序或服务使用。不同的模型存在于客户端“发现”服务中。客户端可以只依赖于服务发现引擎在每次调用服务时解析服务位置。通过这种方法，服务发现引擎将被调用，每次一个调用由已注册的微服务实例发起。不幸的是，这种方法很脆弱，因为服务客户端完全依赖于正在运行的服务发现引擎来查找和调用服务。

更健壮的方法是使用所谓的客户端负载均衡。[4]图 4.3 说明了这种方法。

在这个模型中，当一个消费参与者需要调用一个服务时：

- 它将与服务发现服务（服务消费者要求的所有服务实例）通信，然后在服务消费者的机器上本地缓存数据。
- 每次客户端要调用服务时，服务消费者都会从缓存中查找服务的位置信息。通常，客户端缓存将使用一种简单的负载均衡算法，如“循环轮询”负载均衡算法，以确保服务调用在多个服务实例之间传播。
- 然后，客户端将定期与服务发现服务通信，并刷新服务实例的缓存。客户端缓存最终是一致的，但总有这样的风险：当客户端与服务发现实例进行刷新和调用时，调

2 https://en.wikipedia.org/wiki/Gossip_protocol

3 https://www.cs.cornell.edu/~asdas/research/dsn02-swim.pdf

4 https://en.wikipedia.org/wiki/Load_balancing_(computing)#Client-Side_Random_Load_Balancing

用可能指向不健康的服务实例。

如果在调用服务时，服务调用失败，本地服务发现缓存无效，服务发现客户端将试图从服务发现代理刷新其条目。

现在让我们把通用的服务发现模式运用到 EagleEye 的问题域。

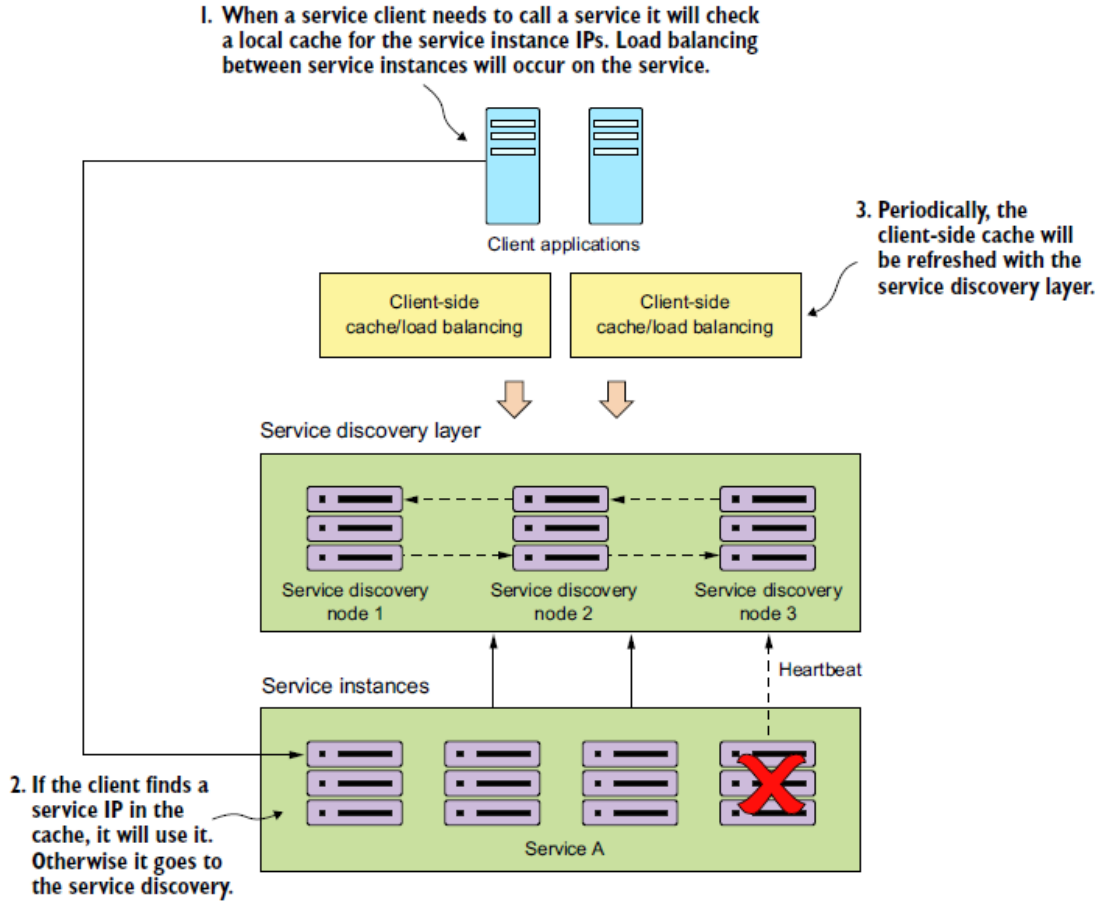


图 4.3 客户端负载平衡缓存服务的位置，以便服务客户机在每次调用时不必与服务发现通信。

① When a service client needs to call a service it will check a local cache for the service instance IPs. Load balancing between service instances will occur on the service.

当服务客户机需要调用服务时，它将检查服务实例 IPs 的本地缓存。服务实例之间的负载均衡将发生在服务上。

② If the client finds a service IP in the cache, it will use it.Otherwise it goes to the

service discovery.

如果客户端在缓存中找到服务 IP，它将使用它。否则，它将定位到服务发现。

③ Periodically, the client-side cache will be refreshed with the service discovery layer.

客户端缓存将周期性地被服务发现层刷新。

4.2.2. 使用 Spring 和 Netflix Eureka 实现服务发现

现在你将通过设置服务发现代理来实现服务发现，然后通过代理注册两个服务。然后，通过使用服务发现取回的信息，将有一个服务调用另一个服务。Spring Cloud 提供了从服务发现代理查找信息的多种方法。我们还将了解每种方法的优点和缺点。

再者，Spring Cloud 项目使这种类型的安装变得微不足道。你将使用 Spring Cloud 和 Netflix 的 Eureka 服务发现引擎来实现服务发现模式。对于客户端均载平衡，你将使用 Spring Cloud 和 Netflix 的 Ribbon 库。

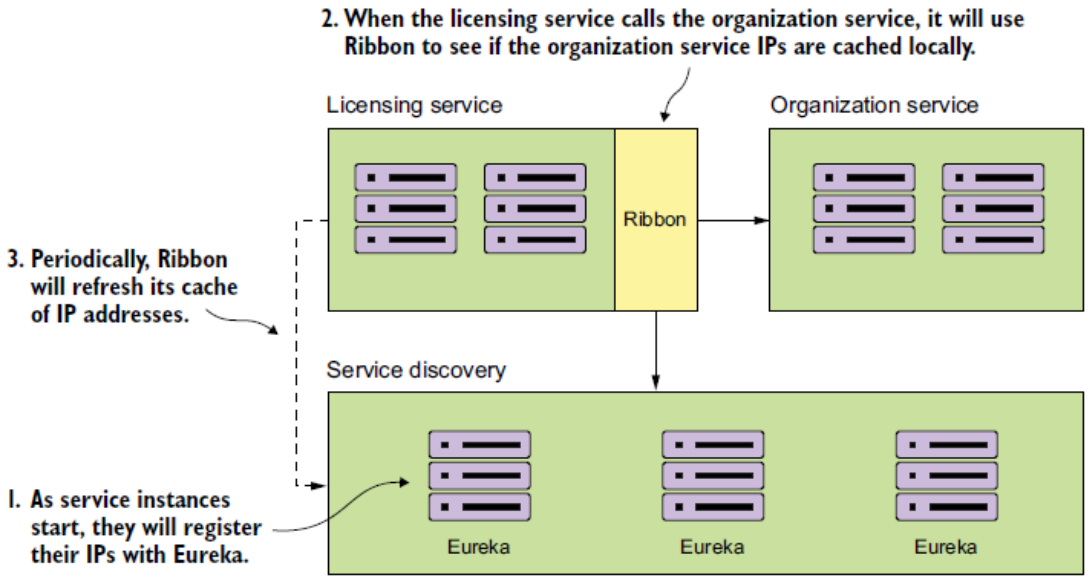


图 4.4 通过实现客户端缓存和使用 Eureka 的许可和组织服务，如果 Eureka 不可用的话，你可以减少 Eureka 服务器上的负载，并提高客户端的稳定性。

① As service instances start, they will register their IPs with Eureka.

当服务实例开始时，他们将用 Eureka 注册它们的 IP。

② When the licensing service calls the organization service, it will use Ribbon to see if the organization service IPs are cached locally.

当许可服务调用组织服务时，它将使用 Ribbon 查找组织服务 IP 是否在本地缓存。

③ Periodically, Ribbon will refresh its cache of IP addresses.

Ribbon 将周期性地刷新其 IP 地址缓存。

在前两章中，你简化了许可服务，并将许可证的组织名称包含在许可证数据中。在本章中，你将将组织信息分解为自己的服务。

当调用许可服务时，它将调用组织服务来检索与指定的组织 ID 相关联的组织信息。组织服务的实际位置将保存在服务发现注册信息中。对于本例，你将使用服务发现注册信息注册组织服务的两个实例，然后使用客户端负载均衡查找和缓存每个服务实例中的注册信息。图 4.4 显示了这种安排：

- 当服务启动的时候，许可服务和组织服务也将把它们自己注册为 Eureka 服务。这个注册过程将告诉 Eureka 每个服务实例的物理位置和端口号以及启动服务的服务 ID。
- 当许可服务调用组织服务时，它将使用 Netflix 的 Ribbon 库提供客户端负载均衡。Ribbon 将与 Eureka 服务通信，检索服务位置信息，然后在本地缓存。
- Netflix 的 Ribbon 库将周期性地 ping Eureka 服务并刷新其本地缓存服务位置。

任何新的组织服务实例现在都可以对本地许可服务可见，而任何非健康实例都将从本地缓存中删除。

接下来，您将通过设置 Spring Cloud Eureka 服务来实现此设计。

4.3. 创建 Spring Eureka Service

在本节中，你将使用 Spring Boot 设置我们的 Eureka 服务。与 Spring Cloud 配置服务一样，设置 Spring Cloud Eureka 服务首先要构建一个新的 Spring Boot 项目，并应用注释和配置。让我们从 Maven pom.xml 文件开始。[5] 下列清单显示了 Spring Boot 项目需要设置的 Eureka 服务依赖。

清单 4.1 向 pom.xml 文件添加依赖

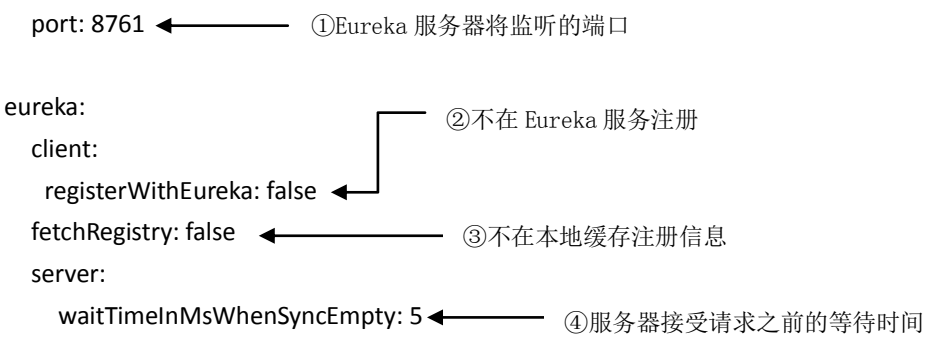
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
        www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
        maven.apache.org/xsd/maven-4.0.0.xsd">

   <modelVersion>4.0.0</modelVersion>

   <groupId>com.thoughtmechanix</groupId>
   <artifactId>eurekasvr</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   <packaging>jar</packaging>

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

   <!--没有显示使用 Spring Cloud Parent 的 maven 定义-->
   <dependencies>
      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-starter-eureka-server</artifactId>
      </dependency>
   </dependencies>
....
</project>
```

①告诉你的 Maven 构建包括 Eureka 库（其中包括 Ribbon）

然后你就需要设置 src/main/resources/application.yml 文件，它使用需要的配置来设置在独立模式下运行的 Eureka 服务（例如，集群中没有其他节点），如下一个清单所示。

清单 4.2 在 application.yml 文件设置 Eureka 配置

```
<?xml version="1.0" encoding="UTF-8"?>
server:
```



关键属性被设置为 server.port 属性，它设置用于该服务的默认端口。

eureka.client.registerWithEureka 属性告诉服务当 Spring Boot Eureka 应用启动的时候，不在 Eureka 服务注册，因为它是 Eureka 服务。eureka.client.fetchRegistry 属性被设置为 false，Eureka 服务启动时，它不会尝试本地化缓存它的注册信息。当运行一个 Eureka 客户端时，你需要为要在 Eureka 注册的 Spring Boot 服务更改此值。

你会发现最后一个属性 ,eureka.server.waitTimeInMsWhenSync 为空 ,被注释掉了。当你测试你的本地服务时 ,你应该取消注释 ,因为 Eureka 不会立即通告任何已注册的服务。默认情况下，它会等待五分钟，在广播之前给所有的服务一个注册的机会。为本地测试取消注释，将有助于将对大大加快 Eureka 服务启动时间和显示注册服务。

单个服务注册并显示在 Eureka 服务需要 30 秒，因为在回复服务准备就绪之前，需要连续三次心跳，每次从服务 ping 间隔 10 秒。在部署和测试自己的服务时，请记住这一点。在设置 Eureka 服务时，你要做的最后一件工作是向应用程序引导类添加一个注解，用于启动 Eureka 服务。

对于 Eureka 服务，应用引导类可以在 src/main/java/com/thoughtmechanix/eurekasvr/EurekaServer-Application.java 类看到。下面的清单显示了在哪里添加注解。

清单 4.3 注释引导类以启用 Eureka 服务器

```
package com.thoughtmechanix.eurekasvr;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
```

```
@SpringBootApplication
@EnableEurekaServer ◄──────── ①在 Spring 服务中启用 Eureka 服务器
public class EurekaServerApplication {
   public static void main(String[] args) {
      SpringApplication.run(EurekaServerApplication.class, args);
   }
}
```

你只使用一个新的注解 ,告诉你的服务是一个 Eureka 服务 ,即@EnableEurekaServer。此刻，你可以通过 mvn spring-boot:run 运行启动 Eureka 服务或 docker-compose（见附录一）启动服务。一旦运行了这个命令，你将有一个正在运行的 Eureka 服务，其中没有注册服务。接下来，你将创建组织服务并将其注册到 Eureka 服务中。

4.4. 使用 Spring Eureka 注册服务

此刻，你有一个基于 Spring 的 Eureka 服务器运行。在本节中，你将配置您的组织和许可服务，并将它们自己注册到 Eureka 服务器。这项准备工作是为了让服务客户端从你的 Eureka 注册信息中查找服务。到你完成的本章的时候，,你应该了解如何将注册一个 Spring Boot 微服务到 Eureka。

用 Eureka 注册一个基于 Spring Boot 微服务是一个非常简单的练习。本章的目的，我们不需要了解编写服务的所有的 Java 代码（我们特意保留小量的代码）,而是专注于在上一节中你创建的 Eureka 服务中注册服务。

你需要做的第一件事是把 Spring Eureka 依赖增加到你的组织服务的 pom.xml 文件：

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-eureka</artifactId> ◄── ①包括 Eureka 库以便服务可以注册到 Eureka
</dependency>
```

spring-cloud-starter-eureka 库是正在使用的唯一新库。Spring Cloud 将使用 spring-cloud-starter-eureka 构件与 Eureka 服务进行交互。

当你创建 pom.xml 文件，你需要告诉 Spring Boot 使用 Eureka 注册组织服务。这个

注册是通过增加组织服务的 src/main/java/resources/application.yml 文件的配置，如下所列。

清单 4.4 修改组织服务的 application.yml，使其与 Eureka 通信

```
spring:
  application:
    name: organizationservice          ← ①将在 Eureka 注册的服务的逻辑名称
  profiles:
    active:
      default
  cloud:
    config:
      enabled: true

eureka:                                ← ②注册服务的 IP 而不是服务器名称。
  instance:
    preferIpAddress: true              ← ③用 Eureka 注册服务。
  client:
    registerWithEureka: true           ← ④下拉注册信息并复制一份到本地。
    fetchRegistry: true
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/   ← ⑤Eureka 服务的位置
```

在 Eureka 中注册的每个服务都有两个要素：应用程序 ID 和实例 ID。应用程序 ID 用于表示一组服务实例。在一个基于 Spring Boot 的微服务，应用程序 ID 将总是由 spring.application.name 属性设置值。对于组织服务，spring.application.name 被命名为 organizationservice。实例 ID 将是一个随机数，用于表示单个服务实例。

> ***注意***：记住，通常 spring.application.name 属性在 bootstrap.yml 文件配置。我已经将它包括在 application.yml 来说明用途。该代码将与 spring.application.name 一起工作，但这个属性长期合适的地方为是 bootstrap.yml 文件。

你的配置的第二部分提供了服务应该如何和在哪里注册到 Eureka 服务。eureka.instance.preferIpAddress 属性告诉 Eureka 要注册到 Eureka 的是服务 IP 地址，而不是主机名。

为什么更喜欢 IP 地址？

默认情况下，Eureka 将会尝试注册与其通信服务的主机名。这在基于服务器的环境中运行良好，在该环境中，服务被指派了一个 DNS 支持的主机名。然而，在一个基于容器的部署（例如，Docker 容器），将使用随机生成的主机名启动而不是容器的 DNS 条目。

如果你不设置 eureka.instance.preferIpAddress 为 true ,你的客户端应用程序将不能妥善解决主机名的位置，因为没有这个容器的 DNS 条目。设置 preferIpAddress 属性将通知 Eureka 服务，客户端通过 IP 地址广播。

就个人而言，我们总是将这个属性设置为 true。基于云的微服务应该是短暂的和无状态的。它们可以启动并随时关闭。IP 地址更适合这些类型的服务。

eureka.client.registerWithEureka 属性是触发器，告诉组织服务在 Eureka 注册它本身。eureka.client.fetchRegistry 属性用来 Spring Eureka 客户端读取注册信息的本地副本。将此属性设置为 true 将在本地缓存注册信息，而不是每次查找都调用 Eureka 服务。每隔 30 秒，客户端软件将重新与 Eureka 服务通信，以便对注册信息进行任何更改。

最后一个属性，eureka.serviceUrl.defaultZone 属性，使用一个逗号分隔的 Eureka 服务列表，客户端将使用解析服务位置。对于我们而言，你只需要一个 Eureka 服务。

Eureka 的高可用性

设置多个 URL 服务对于高可用性是不够的。eureka.serviceurl.defaultZone 属性只提供了一个与客户端通信的 Eureka 服务列表。你还需要设置 Eureka 服务来复制它们彼此之间注册信息的内容。

一组 Eureka 注册中心使用对等通信模型进行通信，为了解集群中的其他节点，其中每个 Eureka 服务都必须配置。建立一个 Eureka 集群超出了这本书的范围。如果你有兴趣建立一个 Eureka 集群，请访问 Spring Cloud 项目的网站了解更多信息。[a]

[a] http://projects.spring.io/spring-cloud/spring-cloud.html

此时，你将在你的 Eureka 服务中注册一个服务。

你可以使用 Eureka 的 REST API 来查看注册信息的内容。要查看服务的所有实例，请单击以下 GET 端点：

http://<eureka service>:8761/eureka/apps/<APPID>

例如，为了在注册信息中查看组织服务，你可以调用

http://localhost:8761/eureka/apps/organizationservice。

图 4.5 调用 Eureka 的 REST API 来查看组织服务，将显示在 Eureka 注册的服务实例的 IP 地址，以及服务状态。

① Lookup key for the service

服务的查找键

② IP address of the organization service instance

组织服务实例的 IP 地址

③ The service is currently up and functioning.

该服务目前正在运行。

Eureka 服务返回的默认格式是 XML。Eureka 还可以将图 4.5 中的数据作为 JSON 有

效载荷返回，但必须将 HTTP 头的 Accept 属性设置为 application/json。JSON 有效负载的一个例子如图 4.6 所示。

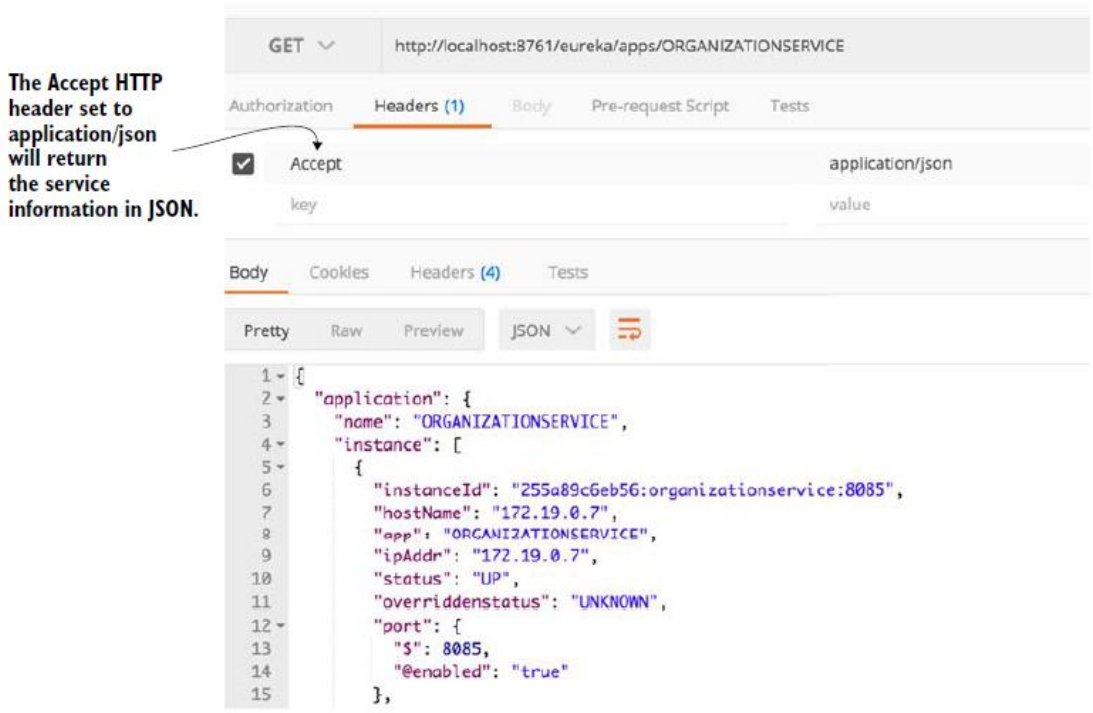


图 4.6 调用 Eureka 的 REST API，结果是 JSON

① The Accept HTTP header set to application/json will return the service information in JSON.

设置 HTTP 头 Accept 属性为 application/json，将以 JSON 返回服务信息。

在 Eureka 和服务启动时，不要心急

当一个服务在 Eureka 注册，服务通过 Eureka 成为可用之前，Eureka 将在 30 秒的过程中等待三个连续的健康检查。这一暖机时间将抛开开发人员，因为它们认为 Eureka 如果他们在服务启动后立即尝试调用它们的服务，它们就不会注册他们的服务。我们的代码示例在 Docker 环境下运行，因为 Eureka 服务和应用程序服务（许可和组织服务）都在同一时间启动。请注意，在应用程序启动后，即使服务本身已经启动，你可能会收到关于未找到服务的 404 错误。等待 30 秒钟后再调用你的服务。

在生产环境中，你的 Eureka 服务已经在运行，如果部署了一个已存在的服务，旧的服务仍然可以接受请求。

4.5. 使用服务发现查找服务

组织服务已注册到 Eureka。你还可以在不直接知道组织服务位置的情况下，使用许可服务调用组织服务。许可服务将使用 Eureka 查找组织服务的物理位置。

为了便于理解，我们将研究三种不同的 Spring/Netflix 客户端库，其中服务消费者可以与 Ribbon 交互。这些库将从最底下的抽象层转移到与最高级别的 Ribbon 交互。我们将探究的库包括：

- Spring Discovery client
- Spring Discovery client enabled RestTemplate
- Netflix Feign client

让我们了解一下这些客户端，并查看它们在许可服务中的使用情况。在开始讨论客户端的细节之前，我在代码中编写了一些便利类和方法，这样你就可以使用相同的服务端点与不同的客户端类型进行交互。

首先，我修改了 src/main/java/com/thoughtmechanix/licenses/controllers/LicenseServiceController.java 包括许可服务的新路由。这个新路由将允许你指定要调用服务的客户端类型。这是一个辅助路由，因此在我们探究通过 Ribbon 调用组织服务的每一种不同方法时，你可以通过一条路由尝试每个机制。

下面的清单显示 LicenseServiceController 类中新路由的代码。

清单 4.5 用不同的 REST 客户端调用许可服务

```
@RequestMapping(value="/{licenseId}/{clientType}",
        method = RequestMethod.GET)
public License getLicensesWithClient(
    @PathVariable("organizationId") String    organizationId,
```

①客户端类型决定了 Spring REST 客户端使用的类型。

```
    @PathVariable("licenseId") String    licenseId,
    @PathVariable("clientType") String    clientType) {
    return licenseService.getLicense(organizationId, licenseId, clientType);
}
```

这段代码，由路由传递的 clientType 参数将限制我们将要在代码示例中使用的客户端类型。在路由这部分内容中，你可以了解的具体类型包括：

- 发现：使用发现客户端和一个标准的 Spring RestTemplate 类调用组织服务
- Rest：使用增强 Spring RestTemplate 调用基于 Ribbon 的服务
- Feign：使用 Netflix 的 Feign 客户端库通过 Ribbon 调用服务

> ***注意***：因为我对所有这三种类型的客户端都使用相同的代码，所以你可能会看到某些客户端的注解，即使它们看起来不需要。例如，你会在代码中看到@EnableDiscoveryClient 和@EnableFeignClients 注解，即使这个版本只能解释其中一个客户类型。因此我可以用一个代码库作为例子。每当遇到这些冗余和代码时，我都会调用它们。

在 src/main/java/com/thoughtmechanix/licenses/services/LicenseService.java 类，我添加了一个简单的方法称为 retrieveorginfo()，它将解析基于 clientType 传递到路由的客户端类型，该方法将被用于查找一个组织服务实例。LicenseService 类的 getLicense() 方法，将使用 retrieveOrgInfo()从 Postgres 数据库检索组织的数据。

清单 4.6 getLicense()方法将使用多种方法进行 REST 调用

```
public License getLicense(String organizationId, String licenseId, String
      clientType) {
    License license = licenseRepository.findByOrganizationIdAndLicenseId(organizationId, licenseId);

    Organization org = retrieveOrgInfo(organizationId, clientType);

    return license
        .withOrganizationName( org.getName())
        .withContactName( org.getContactName())
        .withContactEmail( org.getContactEmail() )
        .withContactPhone( org.getContactPhone() )
        .withComment(config.getExampleProperty());
}
```

你可以在 licensing-service 源代码 src/main/java/com/thoughtmechanix/licenses/clients 包中找到我们使用 Spring DiscoveryClient，Spring RestTemplate 或者 Feign 库创建的每一个客户端。

4.5.1. 使用 Spring DiscoveryClient 查找服务实例

Spring DiscoveryClient 提供了对 Ribbon 及其内部注册的服务的最低访问级别。使用 DiscoveryClient，可以用 Ribbon 客户端查询已注册的所有服务和及其相应的 URLs。

接下来，你将构建一个使用 DiscoveryClient 从 Ribbon 取回一个组织服务的 URL 然后使用标准 RestTemplate 类调用该服务的简单例子。开始使用 DiscoveryClient，你首先需要使用@EnableDiscoveryClient 注解注释 src/main/java/com/thoughtmechanix/licenses/Application.java 类，如下一个清单所示。

清单 4.7 使用 Eureka Discovery Client 设置引导类

```
@SpringBootApplication
@EnableDiscoveryClient ←——— ①激活 Spring DiscoveryClient，使其可用
@EnableFeignClients
public class Application {          ②现在忽略这一点，我们将在本章后面介绍。
   public static void main(String[] args) {
      SpringApplication.run(Application.class, args);
   }
}
```

@EnableDiscoveryClient 注解是 Spring Cloud 让应用程序使用 DiscoveryClient 和 Ribbon 库的触发器。@EnableFeignClients 注解现在可以忽略，我们不久将讨论它。

现在，让我们看看你通过 Spring DiscoveryClient 调用组织服务的代码的实现，如下清单所示。你可以在 src/main/java/com/thoughtmechanix/licenses/OrganizationDiscoveryClient.java 类中找到以上代码。

清单 4.8 使用 DiscoveryClient 查找信息

```
@Component
public class OrganizationDiscoveryClient {
```

```
    @Autowired
    private DiscoveryClient discoveryClient;  <-- ①DiscoveryClient 自动注入类。

    public Organization getOrganization(String organizationId) {
      RestTemplate restTemplate = new RestTemplate();
      List<ServiceInstance> instances =                    <-- ②获取组织服务的所有实例的列表。
        discoveryClient.getInstances("organizationservice");
      if (instances.size()==0) return null;
      String serviceUri = String.format("%s/v1/organizations/%s",
        instances.get(0).getUri().toString(),
        organizationId);  <-- ③检索我们要调用的服务端点。
      ResponseEntity< Organization > restExchange =
            restTemplate.exchange(                         <-- ④使用一个标准的 Spring REST Template 类调用服务
                serviceUri,
                HttpMethod.GET,
                null, Organization.class, organizationId);
      return restExchange.getBody();
    }
}
```

代码中感兴趣的第一项是 DiscoveryClient。你将使用这个类与 Ribbon 交互。检索已经在 Eureka 注册的组织服务的所有实例，你可以使用 getinstances()方法，通过你要找的服务的键检索 ServiceInstance 对象列表。

ServiceInstance 类被用于存储一个包含主机名，端口和 URI 指定服务实例的信息。

在清单 4.8 中，你首先使用 serviceinstance 类列表中建立一个目标 URL，然后可以用来调用你的服务。一旦你有一个目标的 URL，你可以使用一个标准的 Spring RestTemplate 调用你的组织服务和检索数据。

现实中的 DiscoveryClient

我将通过介绍 DiscoveryClient 来完成我们用 Ribbon 创建服务消费者的讨论。现实的情况是，当你的服务需要查询 Ribbon 来了解什么服务和服务实例被注册时，你应该直接使用 DiscoveryClient。这段代码有几个问题，包括以下几点：

你没有利用 Ribbon 的客户端负载平衡。通过直接调用 DiscoveryClient，你得到一个服务

列表，但它会成为你的责任，选择那一个服务实例返回是你要调用的。

你做的工作太多了。现在，您必须构建将用于调用服务的 URL。这是一件小事，但是你可以避免编写的每一段代码都是你不得不调试的一部分代码。

细心的 Spring 开发人员可能已经注意到，你在代码直接实例化 RestTemplate 类。这是与正常的 Spring REST 调用是对立的，通常你会在 Spring 框架通过使用@Autowired 注解注入 RestTemplate 类。

你在清单 4.8 中实例化 RestTemplate 类，因为一旦你通过@EnableDiscoveryClient 注解在应用类中启用 Spring DiscoveryClient，通过 Spring 框架管理所有的 RestTemplates 将有一个启用 Ribbon 的拦截器注入到框架中，它将改变 URLs 在 RestTemplate 类如何被创建。直接实例化 RestTemplate 类允许你避免这种行为。

总之，有更好的机制来调用带 Ribbon 的服务。

4.5.2. 使用 Ribbon-aware Spring RestTemplate 调用服务

接下来，我们要去看一个例子，如何使用 RestTemplate，即 Ribbon-aware。这是通过 Spring 与 Ribbon 交互的更常见的机制之一。为了使用一个 Ribbon-aware RestTemplate 类，你需要定义一个 RestTemplate bean，并使用 Spring Cloud 的 @LoadBalanced 注解注释构造方法。对于许可服务，这种方法被用于创建 RestTemplate bean，在 src/main/java/com/thoughtmechanix/licenses/Application.java 可看到其源代码。

下列的清单显示 getRestTemplate()方法，它将创建支持 Ribbon 的 Spring RestTemplate bean。

清单 4.9 注解和定义一个 RestTemplate 构造方法

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

```
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class Application {

   @LoadBalanced
   @Bean
   public RestTemplate getRestTemplate(){
         return new RestTemplate();
   }

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

①因为我们在示例中使用多个客户端类型，所以我将在代码中包含它们。然而，应用程序不需要@EnableDiscoveryClient 和 @EnableFeignClients ， 当 使 用 支 持 Ribbon 的 RestTemplate 时可以删除。

②@LoadBalanced 注解告诉 Spring Cloud 创建一个支持 Ribbon 的 RestTemplate 类

注意：在 Spring Cloud 的早期版本，RestTemplate 类自动支持 Ribbon。这是默认行为。然而，从 Spring Cloud 的 Angel 版本，在 Spring Cloud 的 RestTemplate 不再支持 Ribbon。如果你想在 RestTemplate 使用 Ribbon，你必须明确使用@LoadBalanced 注解来注释。

现在，支持 Ribbon 的 RestTemplate 被定义为 bean,任何时候你想用 RestTemplate bean 调用一个服务，你只需要将其自动注入类就可使用它。

使用支持 Ribbon 的 RestTemplate 类几乎非常像一个标准的 Spring RestTemplate 类，除了一个小的差异，即目标服务的 URL 如何被定义。RestTemplate 调用不是使用服务的物理位置，你要使用你想调用服务在 Eureka 注册的服务 ID 创建目标 URL。

让我们看看下面的清单，看看这个差异。源代码在 src/main/java/com/thoughtmechanix/licenses/clients/OrganizationRestTemplate.java 类中可看到。

清单 4.10 使用支持 Ribbon 的 RestTemplate 调用服务

```
@Component
public class OrganizationRestTemplateClient {
```

```
  @Autowired
  RestTemplate restTemplate;

  public Organization getOrganization(String organizationId){
    ResponseEntity<Organization> restExchange =
        restTemplate.exchange(
            "http://organizationservice/v1/organizations/{organizationId}",
            HttpMethod.GET,
            null, Organization.class, organizationId);
    return restExchange.getBody();
  }
}
```

① 当使用支持 Ribbon 的 RestTemplate 时，你创建使用 Eureka 的服务 ID 的目标 URL。

除了两个关键区别外，这段代码应该与前一个示例有点类似。首先，Spring Cloud DiscoveryClient 不见了。第二，URL 在 restTemplate.exchange()调用被使用，看起来有些奇怪：

```
restTemplate.exchange(
            "http://organizationservice/v1/organizations/{organizationId}",
            HttpMethod.GET,
            null, Organization.class, organizationId);
```

URL 中的服务器名称相匹配应用程序 ID，这是你在 Eureka 注册组织服务使用的 organizationservice 键：

```
http://{applicationid}/v1/organizations/{organizationId}
```

启用 Ribbon 的 RestTemplate 将解析传递进来的 URL，通过传入的服务名称作为键用 Ribbon 查询一个服务实例。实际的服务位置和端口完全从开发人员那里抽象出来。

此外，使用 RestTemplate 类，Ribbon 将轮询 robin，负载均衡所有服务实例之间的所有请求。

4.5.3. 使用 Netflix Feign client 调用服务

Netflix 的 Feign 客户端库是一种替代 Spring Ribbon-enabled RestTemplate 类的选择。Feign 库采用一种不同的方法，通过开发人员首先定义一个 Java 接口并使用 Spring Cloud 注解注释接口来映射 Ribbon 将调用基于 Eureka 的服务， 调用一个 REST 服务。

Spring Cloud 框架动态生成一个代理类，它将被用于调用目标 REST 服务。除了接口定义之外，没有编写调用服务的代码。

为了使许可服务启用 Feign 客户端，你需要在许可服务的 src/main/java/com/thoughtmechanix/licenses/Application.java 类添加一个新注解，即@EnableFeignClients。下面的清单显示了此代码。

清单 4.11 在许可服务启用 Spring Cloud/Netflix Feign client

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

①因为我们只使用 FeignClient，你可以在你自己的代码删除@EnableDiscoveryClient 注解。

②在你的代码中，使用 FeignClient 需要@EnableFeignClients 注解。

现在你已经在你的许可服务中启用 Feign 客户端，让我们看看一个 Feign client 接口定义，它可以被用于调用组织服务的一个端点。下面的清单显示了一个示例。清单中的代码可以在 src/main/java/com/thoughtmechanix/licenses/clients/OrganizationFeignClient.java 类中找到。

清单 4.12 定义一个调用组织服务的 Feign 接口

```
@FeignClient("organizationservice")
public interface OrganizationFeignClient {
   @RequestMapping(
            method= RequestMethod.GET,
            value="/v1/organizations/{organizationId}",
            consumes="application/json")
   Organization getOrganization(
    @PathVariable("organizationId") String organizationId);
}
```

①确定你的服务使用 FeignClient 注解定义 Feign

②使用@RequestMapping 注解定义端点的 path 和 action

③传入端点的参数使用@PathVariable 注解定义。

你用@FeignClient 注解开始 Feign 示例，并定义你想表示的接口的服务应用程序 ID。下一步，在你的接口，你将定义一个方法，getOrganization()，在客户端调用组织服务时被调用。

你如何定义 getOrganization()方法看起来就像你如何在一个 Spring 控制器类暴露一

个端点。首先，你将使用@RequestMapping 注解定义 getOrganization()方法，该注解映射 HTTP 动词和暴露在组织服务调用的端点。其次，你使用@PathVariable 注解，将在 URL 中传递的组织 ID 映射到方法调用的 organizationId 参数。调用组织服务的返回值将被自动映射到 Organization 类，该类被定义为 getOrganization()方法的返回值。

使用 OrganizationFeignClient 类，所有你需要做的是自动装配和使用它。Feign 客户端代码会为你处理所有的编码工作。

对错误的处理

当你使用标准的 Spring RestTemplate 类，所有服务调用的 HTTP 状态码将通过 ResponseEntity 类的 getStatusCode()方法返回。对于 Feign 客户端，任何 HTTP 4xx–5xx 状态码返回通过被调用的服务将被映射到一个 FeignException。FeignException 将包含一个 JSON 报文体，它将被解析为具体的错误信息。

Feign 为你提供编写错误解码器类的能力，它将返回的错误映射为一个自定义异常类。编写这个解码器超出了本书的范围，但你可以在 Feign GitHub 仓库(https://github.com/Netflix/feign/wiki/Custom-error-handling)找到示例。

4.6. 小结

- 服务发现模式用于抽象服务的物理位置。
- 像 Eureka 这样的服务发现引擎可以无缝地添加和删除环境中的服务实例，而不影响服务客户端。
- 客户端负载均衡可以通过在服务调用的客户端上缓存服务的物理位置来提供额外的性能和弹性。
- Eureka 是一个 Netflix 项目，当与 Spring Cloud 一起使用时，很容易设置和配置。

- 你在 Spring Cloud，Netflix Eureka 和 Netflix Ribbon 中使用三种不同的机制来调用服务。这些机制包括：
  - 使用一个 Spring Cloud 服务 DiscoveryClient
  - 使用 Spring Cloud 和支持 Ribbon 的 RestTemplate
  - 使用 Spring Cloud 和 Netflix 的 Feign 客户端