6. 第 6 章 使用 Spring Cloud 和 Zuul 进行服务路由

本章内容

- 在你的微服务中使用服务网关
- 使用 Spring Cloud 和 Netflix Zuul 实现一个服务网关
- 在 Zuul 里映射微服务路由
- 使用关联 ID 和跟踪构建过滤器
- 使用 Zuul 实现动态路由

在一个像微服务的分布式架构，你需要确保关键行为，如：安全，日志记录和跟踪跨用户多个服务调用的发生。为了实现此功能，你希望这些属性在所有的服务中得到一致地执行，而不需要每个开发团队构建自己的解决方案。虽然可以使用公共库或框架来帮助在单个服务中直接构建这些能力，但这样做有三个含义。

首先，要在正在构建的每个服务中始终如一地实现这些能力是很困难的。开发人员专注于交付功能，在日常活动的旋风中，他们很容易忘记实现服务日志记录或跟踪。（我个人对此感到内疚。）不幸的是，对于那些在监管很严的行业，如金融服务业或医疗保健公司，在你的系统中显示一致和记录的行为常常是遵守政府规章的关键要求。

其次，正确实现这些能力是一个挑战。像微服务安全建立和为每个服务实现配置一定是件痛苦的事情。把像安全这样的横切关注点实现责任推到个人开发团队，这会大大增加一个人没有正确执行或忘记执行它的几率。

第三，现在你已经在所有服务上创建了一个硬依赖项。更多的能力，你构建一个通用框架共享你所有的服务，更困难的是在你的公共代码中改变或添加的行为而无需重新编译和重新部署你的所有服务。这可能似乎不是什么大不了的，在你的应用中你有 6 个微服务，但

它是一个大问题，当你有大量的服务，也许是 30 个或更多的。突然，构建到共享库中的核心能力的升级成为一个长达数月的迁移过程。

为了解决这个问题，你需要抽象这些横切关注点为服务， 该服务能独立作为应用程序里所有的微服务调用的过滤器和路由器。这种横切关注点称为服务网关。你的服务客户端不再直接调用服务。相反，所有调用都通过服务网关路由，该网关充当一个单一的策略执行点（PEP），然后路由到最终目的地。

在这一章中，我们将看到如何使用 Spring Cloud 和 Netflix 的 Zuul 实现服务网关。Zuul 是 Netflix 的开源服务网关的实现。具体来说，我们来看看如何使用 Spring Cloud 和 Zuul 来：

- 将所有服务调用放在一个 URL 后面，并使用服务发现将这些调用映射到实际的服务实例中。
- 将关联 ID 注入到通过服务网关的每个服务调用中
- 从客户端发回的 HTTP 响应中注入关联 ID
- 构建一个动态路由机制，将特定的单个组织路由到与其他人使用不同的服务实例端点

让我们深入到更详细的关于服务网关如何适合于本书中正在构建的整个微服务。

6.1. 服务网关

直到现在，你已经在前几章构建微服务，你通过 Web 客户端直接调用单独的服务或通过服务发现引擎的编程方式调用它们，如 Eureka。

① When a service client invokes a service directly, there’s no way you can easily implement cross-cutting concerns such as security or logging without having each

service implement this logic directly in the service.

当服务客户端直接调用服务时，你不可能轻松地实现横切关注点，如安全或日志记录，且不必让每个服务直接在服务中实现此逻辑。

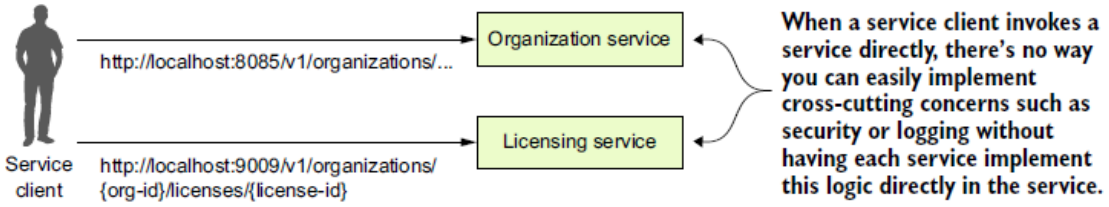

图 6.1 没有服务网关，服务客户端将为每个服务调用不同的端点。

服务网关充当服务客户端和正在调用的服务之间的中介。服务客户端只对由服务网关管理的单个 URL 进行访问。服务网关将从服务客户端调用中提取分离出来的路径，并确定服务客户端试图调用的服务。图 6.2 说明了如何像一个交通警察指挥交通，服务网关直接将用户路由到目标微服务和相应的实例。服务网关充当在应用程序中所有到达微服务调用的守门人。有了服务网关，你的服务客户端不会直接调用单个服务的 URL，而是将所有调用都放置到服务网关。

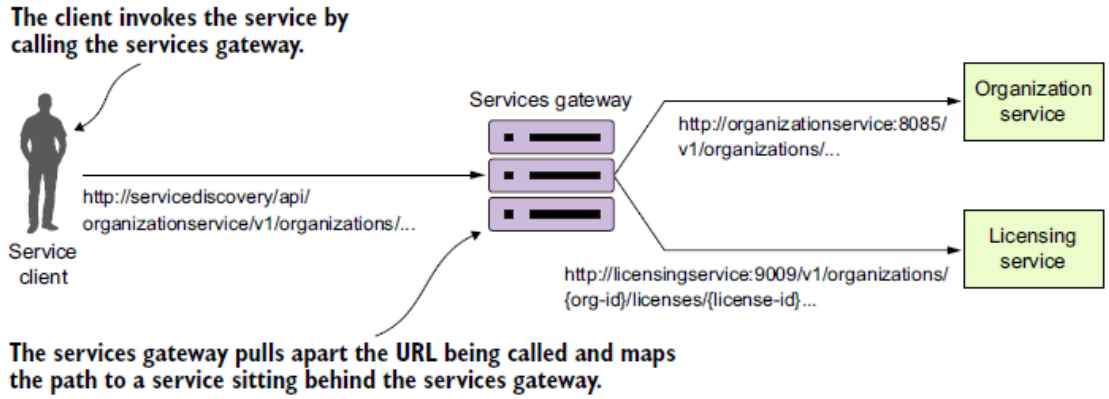

图 6.2 服务网关位于服务客户端和相应的服务实例之间。所有服务调用（内部和外部）都应该流经服务网关。

① *The client invokes the service by calling the services gateway.*

客户端通过调用服务网关调用服务。

② The services gateway pulls apart the URL being called and maps the path to a service sitting behind the services gateway.

服务网关将被调用的 URL 分离，并将路径映射到服务网关后面的服务。

由于服务网关位于从客户端到各个服务的所有调用之间，它也充当服务调用的中央策略执行点（PEP）。使用集中式 PEP 意味着横切服务关注点可以在一个地方实现，而不需要单个开发团队实现这些关注点。可以在服务网关中实现的横切关注点示例包括：

- 静态路由：服务网关将所有服务调用放在一个 URL 和 API 路由后面。这简化了开发，因为开发人员只需要知道所有服务的一个服务端点。
- 动态路由：服务网关可以检查传入的服务请求，并根据来自传入请求的数据，执行基于服务调用者的智能路由。例如，参与 beta 测试程序的客户可能对所有路由到特定服务集群的服务进行调用，这些服务组正在运行与其他人使用的不同版本的代码。
- 认证和授权：由于所有服务调用都通过服务网关路由，因此服务网关是检查服务调用方是否已进行身份验证并授权进行服务调用的一个自然场所。
- 度量收集和日志记录：当服务调用通过服务网关时，服务网关可用于收集度量和日志信息。你还可以使用服务网关来确保用户请求上的关键信息已经到位，以确保日志记录是一致的。这并不意味着你不应该还从单个服务中收集度量标准，而是服务网关允许你集中收集许多基本度量，例如服务调用的次数和服务响应时间。

等等，服务网关不是一个单一的故障点和潜在的瓶颈吗？

早在第 4 章我介绍 Eureka 时，我谈到了集中负载均衡器如何成为你服务一个单一的故障点和瓶颈。如果服务网关没有正确实现，它可能承担同样的风险。在构建服务网关实现时，请记住以下几点。

负载均衡器在一组单独的服务前面时，仍然是有用的。在这种情况下，位于多个服务网关实例前面的负载均衡器是一个适当的设计，并确保服务网关可以实现伸缩。在所有服务实例前面都有一个负载均衡器，这不是一个好主意，因为它成为瓶颈。

保持为服务网关编写的所有代码都是无状态的。不要将任何信息存储在服务网关的内存中。如果你不注意，你将限制网关的可伸缩性，并必须确保在所有服务网关实例中复制数据。

保持为服务网关编写的代码是轻量级的。服务网关是你的服务调用的“瓶颈”。具有多个数据库调用的复杂代码可能是跟踪服务网关性能问题的根源。

现在让我们来看看如何使用 Spring Cloud 和 Netflix Zuul 实现服务网关。

6.2. Spring Cloud 和 Netflix Zuul

Spring Cloud 集成了 Netflix 的开源项目 Zuul。Zuul 是服务网关，通过 Spring Cloud 注解非常易于设置和使用。Zuul 提供了许多功能，包括：

- *将应用程序中所有服务的路由映射到一个 URL。*Zuul 并不局限于单一的 URL。在 Zuul，你可以定义多个路由条目，使路径映射极细粒度（每个服务端点都有自己的路由映射）。然而，对于 Zuul 第一和最常见的情况是建立一个单一的入口点，所有的服务客户端调用都流经它。
- *构建能够检查和响应通过网关的请求的过滤器。*这些过滤器允许您在代码中注入策略执行点，并以一致的方式在所有服务调用上执行大量操作。

开始使用 Zuul 之前，你要做的三件事：

- 创建一个 Zuul Spring Boot 项目并配置适当的 Maven 的依赖。
- 修改你的 Spring Boot 项目与 Spring Cloud 注解，告诉它，它将是一个 Zuul 服务。

- 配置 Zuul 与 Eureka 通信（可选）。

6.2.1. 配置 Spring Boot 工程引用 Zuul 依赖

如果你在本书中按顺序阅读这些章节，你对要做的工作应该是熟悉的。创建一个 Zuul 服务器，你需要建立一个新的 Spring Boot 启动服务并定义相应的 Maven 依赖。你可以找到本章项目的源代码，在这本书的 GitHub 库 (https://github.com/carnellj/spmia-chapter6)。幸运的是，在 Maven 中很少需要设置 Zuul。你只需要在你的 zuulsvr/pom.xml 文件中定义一个依赖：

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```

这个依赖告诉 Spring Cloud 框架，这个服务将运行 Zuul 和适当初始化 Zuul。

6.2.2. 使用 SpringCloud 注解配置 Zuul 服务

你定义 Maven 依赖之后，你需要注解 Zuul 服务的引导类。Zuul 服务引导类实现可以在 zuulsvr/src/main/java/com/thoughtmechanix/zuulsvr/Application.java 类找到。

清单 6.1 创建 Zuul 服务器引导类

```
package com.thoughtmechanix.zuulsvr;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
@EnableZuulProxy  <-------------------  ①使服务成为一个 Zuul 服务器
public class ZuulServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ZuulServerApplication.class, args);
    }
}
```

就是这样。只有一个注解需要到位：@EnableZuulProxy。

> ***注意***：如果你查看文档或自动完成打开，你可能会注意到一个名为@EnableZuulServer 的注解。使用这个注解将创建一个 Zuul 服务器，它不加载任何的 Zuul 反向代理过滤器或使用 Netflix Eureka 服务发现。（我们不久将进入 Zuul 和 Eureka 集成的主题。）当你想建立你自己的路由服务和不使用任何 Zuul 预置功能时，@EnableZuulServer 被使用。举个例子，如果你想使用 Zuul 与服务发现引擎集成，除了 Eureka(例如，Consul)。我们在本书中只使用@EnableZuulProxy 注解。

6.2.3. 配置 Zuul 与 Eureka 通信

Zuul 代理服务器被设计为默认情况下在 Spring 产品下工作。因此，Zuul 将自动使用 Eureka 通过服务 ID 查找服务，然后使用 Netflix 的 Ribbon 为 Zuul 请求做客户端的负载均衡。

> ***注意***：我经常不按顺序读一本书中的章节，而是跳到我最感兴趣的主题。如果你也做同样的事情，不知道 Netflix 的 Eureka 和 Ribbon 是什么，我建议你在进入下一步之前先读第 4 章。Zuul 使用那些技术大量开展工作，所以了解服务发现能力，Eureka 和 Ribbon 带来的东西会使理解 Zuul 更容易。

在配置过程的最后一步是修改你的 Zuul 服务器的 zuulsvr/src/main/resources/application.yml 文件指向你的 Eureka 服务器。下列清单显示的是 Zuul 与 Eureka 通信的配置。清单中的配置应该看起来很熟悉，因为它与我们在第 4 章中看到的配置相同。

清单 6.2 配置 Zuul 服务器和 Eureka 交互

```
eureka:
  instance:
    preferIpAddress: true
  client:
    registerWithEureka: true
    fetchRegistry: true
```

```
serviceUrl:
  defaultZone: http://localhost:8761/eureka/
```

6.3. 在 Zuul 中配置路由

Zuul 本质上是一个反向代理。反向代理是介于客户端试图到达资源和资源本身之间的中间服务器。客户端不知道它甚至与代理以外的服务器通信。反向代理负责捕获客户端的请求，然后以客户端的名义调用远程资源。

在一个微服务架构的情况下，Zuul（反向代理）以微服务调用从客户端转发到下游服务。服务客户端认为它只与 Zuul 通信。对于 Zuul 与下游客户端通信，Zuul 已经知道如何将传入的调用映射到下游路由。Zuul 有这样几种机制，包括：

- 通过服务发现自动映射路由
- 通过服务发现手动映射路由
- 使用静态 URL 手动映射路由

6.3.1. 通过服务发现自动映射路由

Zuul 所有路由映射通过 zuulsvr/src/main/resources/application.yml 文件定义。然而，Zuul 可以基于零配置服务 ID 自动路由请求。如果你不指定任何路由，Zuul 将自动使用被调用服务的 Eureka 服务 ID 和将它映射到下游服务实例。例如，如果你想调用你的组织服务和通过 Zuul 使用自动路由，你将会有你的客户端调用 Zuul 服务实例，使用下面的 URL 作为端点：

http://localhost:5555/organizationservice/v1/organizations/e254f8c-c442-4ebea82a-e2fc1d1ff78a

通过 http://localhost:5555 访问你的 Zuul 服务器。你试图调用的服务

(organizationservice)由服务中端点路径的第一部分表示。

① *The service name acts as the key for the service gateway to lookup the physical location of the service.*

服务名称充当服务网关查找服务的物理位置的键。

② *The rest of the path is the actual url endpoint that will be invoked.*

路径的其余部分是将调用的实际 URL 端点。

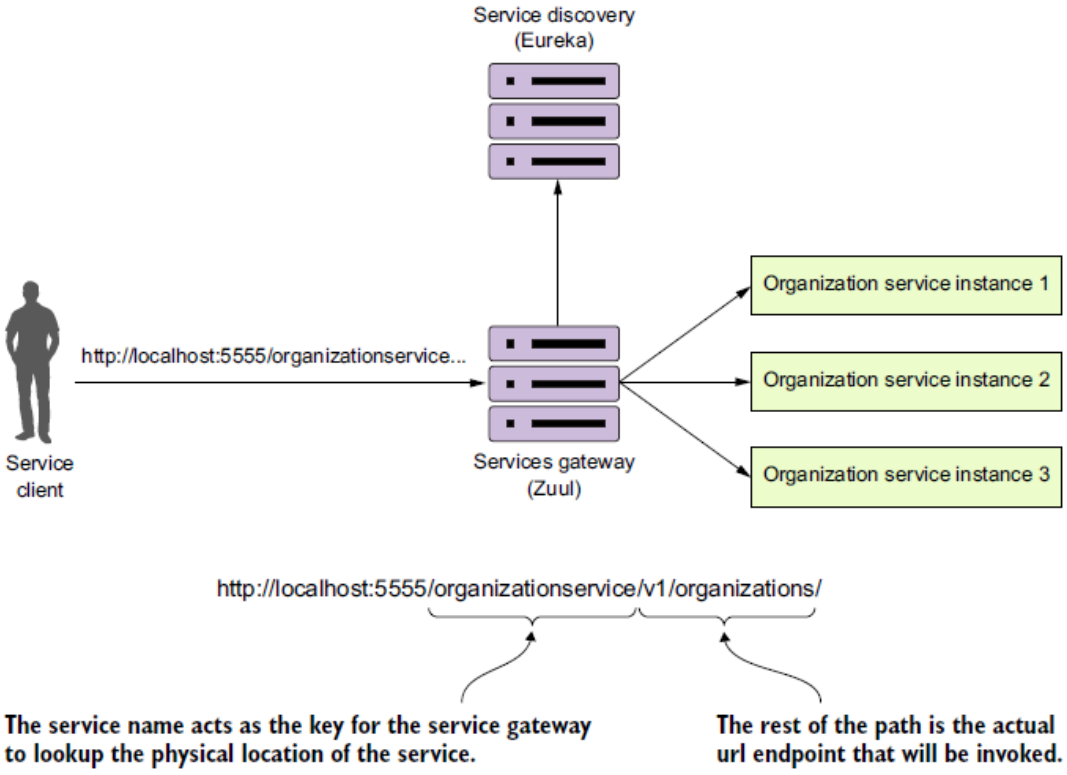

图 6.3 Zuul 将使用 organizationservice 应用程序名称来映射到组织服务实例的请求。

图 6.3 演示了映射的实践。

使用带 Eureka 的 Zuul 的优点是，你不仅现在有一个单一的，你可以调用的端点，但使用 Eureka，你也可以添加和删除一个服务实例而不必修改 Zuul。例如，你可以添加一个新的服务到 Eureka，和 Zuul 将自动路由到它因为它与 Eureka 通信，Eureka 知道实际物理服务端点位于何处。

如果你想看到路由被 Zuul 服务器管理 ,你可以通过在 Zuul 服务器上的/routes 端点访问该路由。这将返回服务上所有映射的列表。图 6.4 显示点击 http://localhost:5555/route 的输出。

图 6.4 中将注册在 Zuul 的服务映射显示在/route 调用返回 JSON 报文体的左侧。路由的实际 Eureka 服务 ID 显示在右侧。

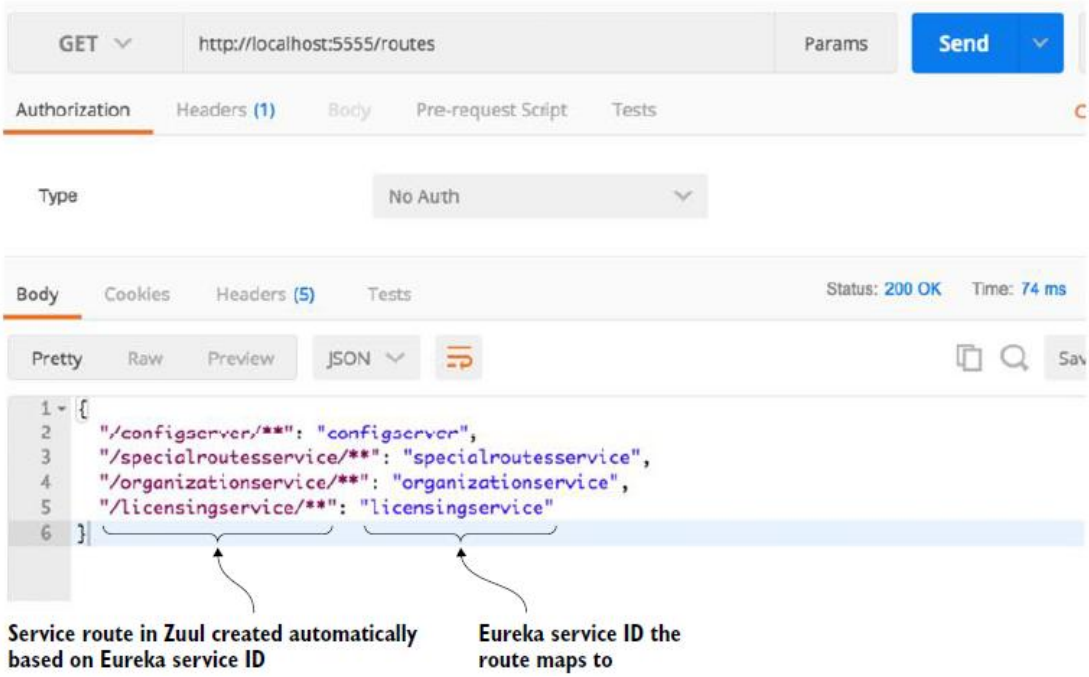

图 6.4 映射在 Eureka 的每个服务现在将被映射为一个 Zuul 路由。

① *Service route in Zuul created automatically based on Eureka service ID*

基于 Eureka 服务 ID 自动创建 Zuul 服务路由

② *Eureka service ID the route maps to*

路由映射到 Eureka 服务 ID

6.3.2. 通过服务发现手动映射路由

Zuul 允许你通过明确的定义路由映射来定义更细粒度的路由，而不仅仅是依靠自动路由服务创建服务的 Eureka 服务 ID。假如你想通过缩短组织的名字而不是你的组织服务通过

默认路由/organizationservice/v1/organizations/{organizationid}访问 Zuul 来简化路由。你可以通过在 zuulsvr/src/main/resources/application.yml 文件中手动定义路由映射：

```
zuul:
  routes:
    organizationservice: /organization/**
```

通过添加此配置 ,你现在可以通过点击/organization/v1/organizations/{organization-id}路由来访问组织服务。如果你再检查 Zuul 服务器的端点 ,你应该看到如图 6.5 所示的结果。

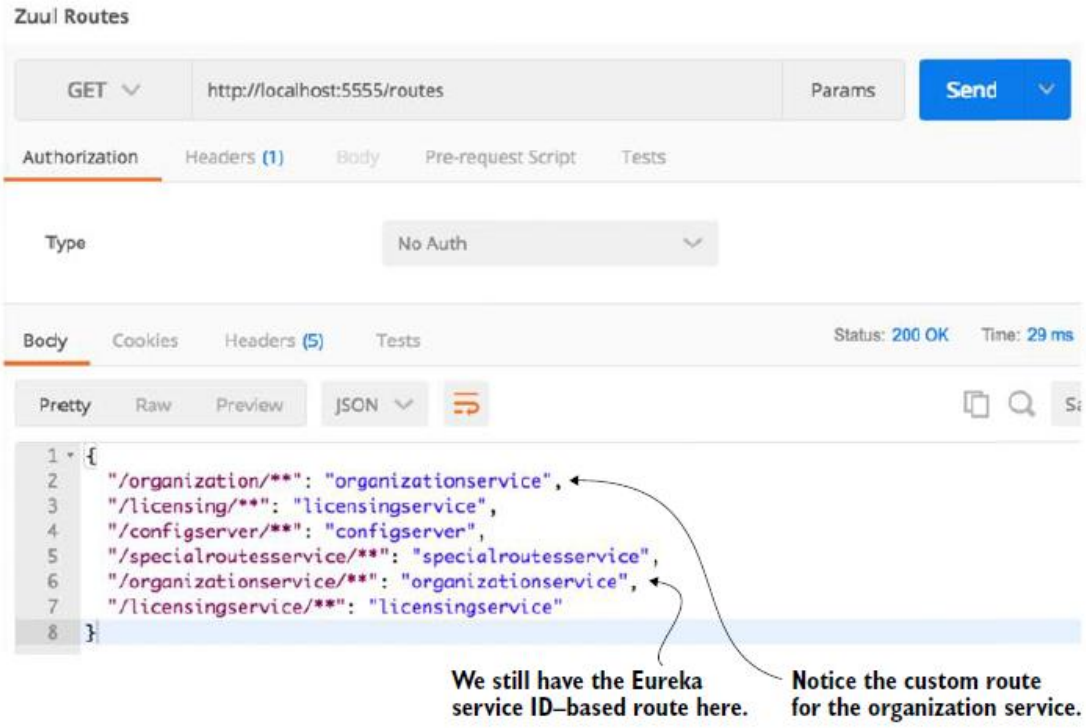

图 6.5 使用组织服务的手动映射 Zuul /routes 的调用结果

① *We still have the Eureka service ID–based route here.*

我们在这里仍然有以 Eureka 服务 ID 为基础的路由。

② *Notice the custom route for the organization service.*

请注意组织服务的自定义路由。

如果仔细查看图 6.5，你会发现组织服务中有两个条目。第一个服务入口是你定义的 application.yml 文件的映射：“organization/**” ： “organizationservice”。第二服务入

口是通过 Zuul 创建基于组织服务的 Eureka ID 的自动映射："/organizationservice/**"："organizationservice"。

> ***注意***：当你使用自动路由映射在 Zuul 暴露完全基于 Eureka 服务 ID 的服务，如果没有服务实例在运行，Zuul 不会暴露服务路由。然而，如果你手动映射路由到服务发现 ID 且没有实例在 Eureka 注册，Zuul 将仍然显示该路由。如果你尝试调用不存在的服务路由，Zuul 会返回一个 500 错误。

如果你想排除 Eureka 服务 ID 路由的自动映射和你已定义的可用的组织服务路由，你可以添加一个额外的 Zuul 参数到你的 application.yml 文件，称为 ignored-services。下面的代码片段显示了 ignored-services 属性可以用来排除通过 Zuul 自动映射的 Eureka 服务 ID "organizationservice"。

```
zuul:
  ignored-services: 'organizationservice'
  routes:
    organizationservice: /organization/**
```

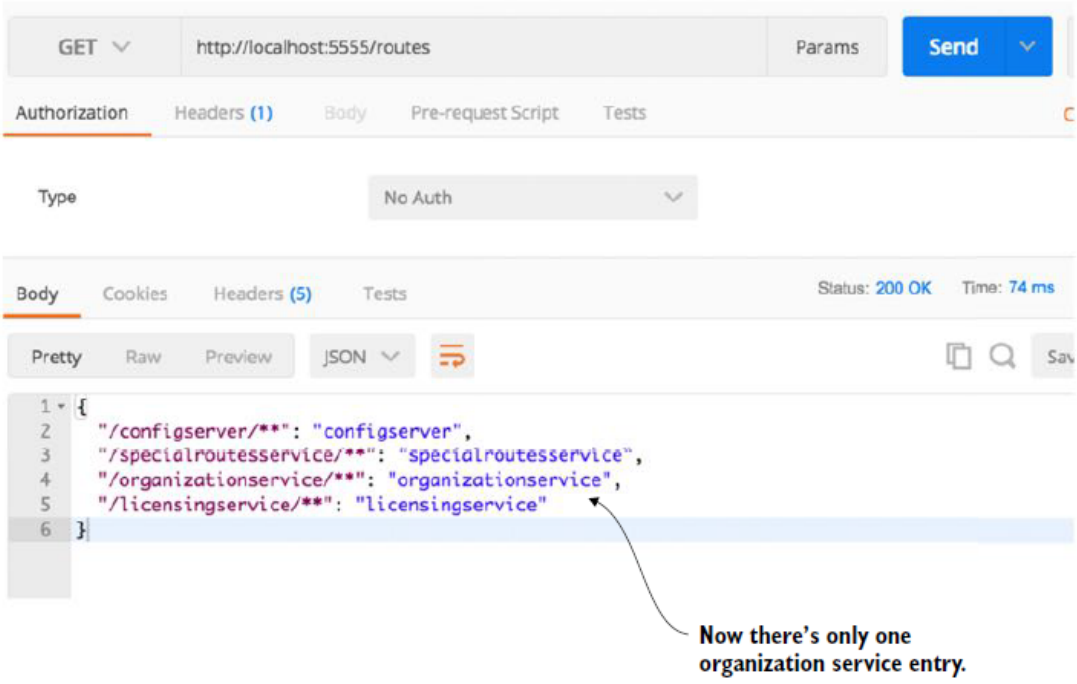

图 6.6 现在仅有一个组织服务定义在 Zuul。

① *Now there's only one organization service entry.*

现在只有一个组织服务条目。

ignored-services 属性允许你定义一个以逗号分隔的 Eureka 服务 ID 列表，你希望将其排除在注册之外。现在，当你调用 Zuul 的/routes 端点，你只能看到你定义的组织服务映射。图 6.6 显示了这个映射的结果。

如果你想排除所有基于 Eureka 的路由，可以将 ignored-services 属性设置为“*”。

区分服务网关 API 路由与内容路由的一个常见的模式是通过使用一类标签的所有服务调用前缀，如：/api，来区别。Zuul 通过在 Zuul 配置中使用前缀属性支持上述情况。图 6.7 概念地给出了这个映射前缀看起来像什么样的。

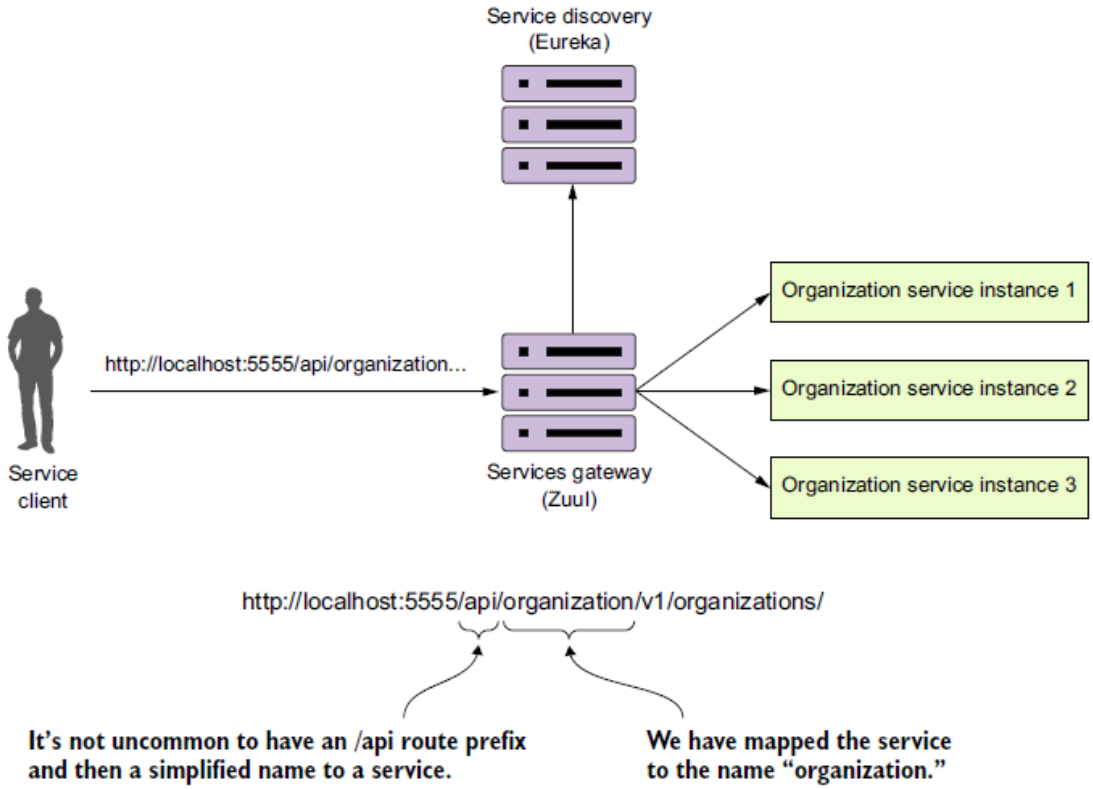

图 6.7 使用前缀，Zuul 将映射/api 前缀到每个它管理的服务。

① *It’s not uncommon to have an /api route prefix and then a simplified name to a service.*

有一个/api 路由前缀，然后是一个简化的服务名称，这并不少见。

② We have mapped the service to the name "organization."

我们已经将服务映射到“organization”这个名称。

在下面的清单中，我们将看到如何为你的组织服务和许可服务设置特定的路由，排除所有 Eureka 生成的服务，并用/api 前缀作为服务的前缀。

清单 6.3 使用前缀设置自定义路由

```
zuul:
  ignored-services: '*'
  prefix: /api
  routes:
    organizationservice: /organization/**
    licensingservice: /licensing/**
```

①ignored-services 属性设置为*，以排除所有基于 eureka 服务 ID 路由的注册。

②所有定义的服务将以/api 为前缀。

③你的 organizationservice 和 licensingservice 分别映射到 organization 和 licensing 端点。

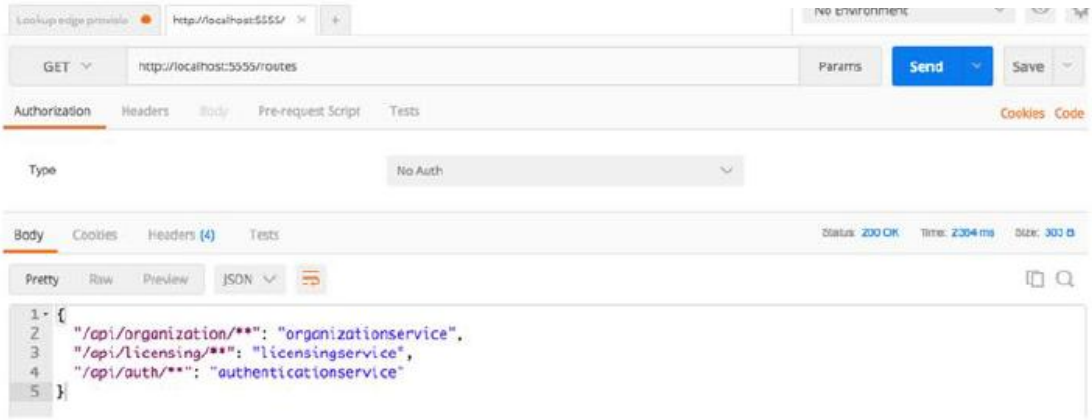

图 6.8 现在，你在 Zuul 的路由有/api 前缀。

一旦配置完成，Zuul 服务已经重新加载，当点击/route 端点时你应该看到以下两项：/api/organization 和/api/licensing。图 6.8 显示了这些条目。

现在让我们来看看如何使用 Zuul 映射到静态 URL。静态 URL 是指向服务但没有在 Eureka 服务发现引擎注册的 URL。

6.3.3. 使用静态 URL 手动映射路由

Zuul 可用于路由服务，但它不受 Eureka 管理。在这些情况下，Zuul 可建立直接路由到一个静态定义的 URL。例如，假设你的许可服务是用 Python 写的，你还想通过 Zuul 代

理。你会在下面的清单使用 Zuul 配置来实现这一需求。

清单 6.4 映射许可服务到静态路由

```
zuul:
  routes:
    licensestatic:
      path: /licensestatic/**
      url: http://licenseservice-static:8081
```

①Zuul 用来识别内部服务键名

②许可服务的静态路由

③你已经建立了一个许可服务的静态实例，它将直接被调用，而不是由 Zuul 通过 Eureka。

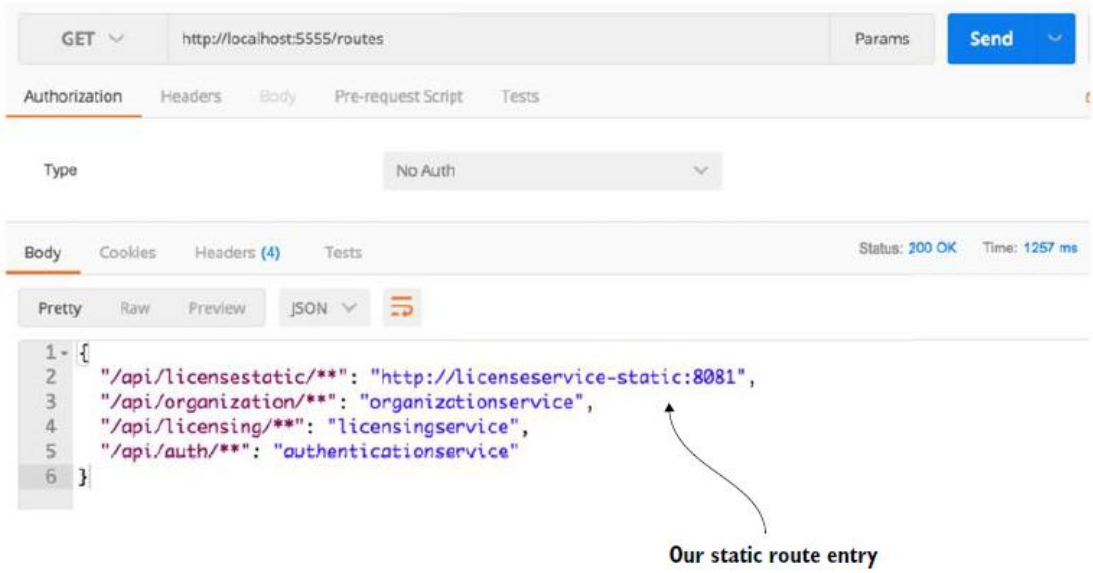

图 6.9 你现在已经映射一个静态路由到许可服务。

① *Our static route entry*

静态路由条目

一旦这配置变化了，你可以点击/routes 端点，并看到已增加到 Zuul 的静态路由。图 6.10 显示了/routes 列表的结果。

在这一点上，licensestatic 端点不使用 Eureka 并且直接路由请求到 http://licenseservice-static:8081 端点。问题是通过绕过 Eureka，你只能有一个指向请求的路由。幸运的是，你可以手动配置 Zuul 禁用 Ribbon 与 Eureka 整合，然后列出 ribbon 将负载均衡每个服务实例。下面的清单显示了这一点。

清单 6.5 将许可服务静态地映射到多个路径

```
zuul:
```

```
routes:
   licensestatic:
     path: /licensestatic/**
     serviceId: licensestatic
ribbon:
 eureka:
   enabled: false
licensestatic:
 ribbon:
   listOfServers: http://licenseservice-static1:8081,
    http://licenseservice-static2:8082
```

①定义一个服务 ID，用于在 Ribbon 中查找服务。

②禁用 Ribbon 中的 Eureka 支持

③请求路由到的服务器列表

一旦配置完成，对/routes 端点的调用现在显示/api/licensestatic 路由已经被映射到一个称为 licensestatic 的服务 ID。图 6.10 显示了这一点。

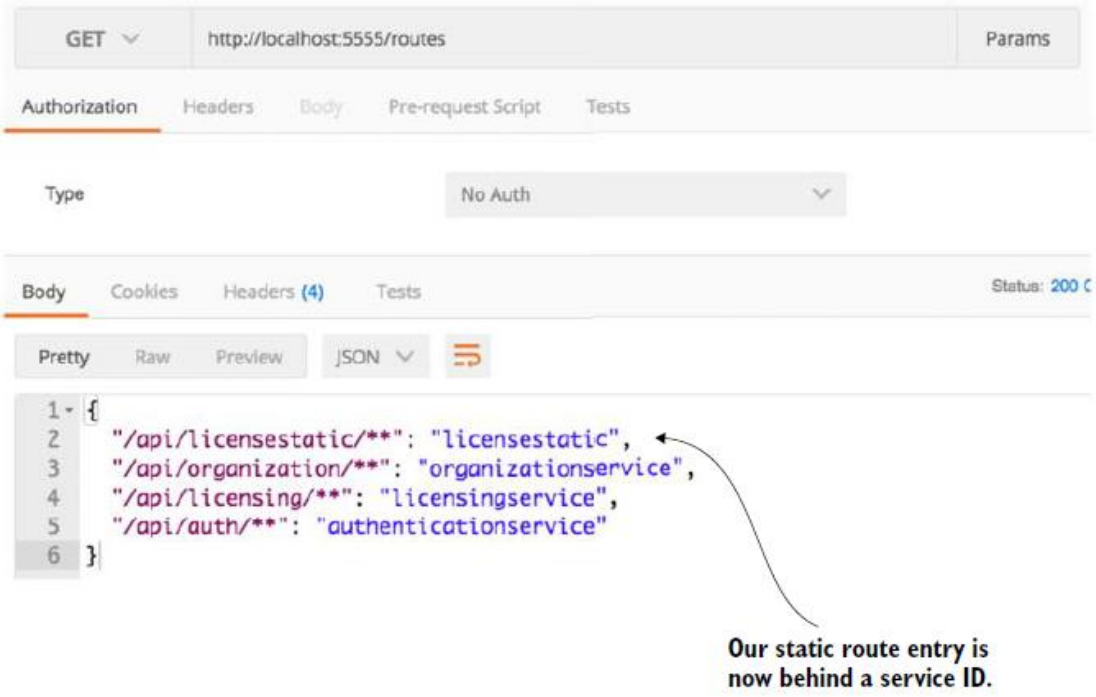

图 6.10 你现在看到/api/licensestatic 映射到一个称为 licensestatic 的服务 ID

① *Our static route entry is now behind a service ID.*

我们的静态路由条目现在位于服务 ID 后面。

处理非 JVM 服务

静态路由映射的问题和在 Ribbon 中禁用 Eureka 支持，将禁用通过 Zuul 服务网关的

所有服务的 Ribbon 支持。这意味着更多的负载将被放置在你的 Eureka 服务器，因为 Zuul 不能使用 Ribbon 缓存查找服务。记住，Ribbon 不会每次都调用 Eureka。相反，它在本地缓存服务实例的位置，然后定期对 Eureka 的更改进行检查。在 Ribbon 缺席的情况下，Zuul 将每次调用 Eureka 来解析服务的位置。

在本章的前面，我谈到了如何使用多个服务网关，根据所调用的服务类型，将执行不同的路由规则和策略。非 JVM 的应用程序，你可以建立一个单独的 Zuul 服务器来处理这些路由。然而，我发现，与非基于 JVM 的语言，你最好设置一个 Spring Cloud "Sidecar" 实例。Spring Cloud sidecar 让你使用 Eureka 实例注册非 JVM 服务，然后通过 Zuul 代理它们。我在本书中不涵盖 Spring Sidecar，因为你没有写任何非 JVM 服务，但是设置 sidecar 实例非常容易。关于如何做到这一点，可以在 Spring Cloud 网站上找到 (http://cloud.spring.io/spring-cloud-netflix/spring-cloud-netflix.html#spring-cloud-ribbonwithout-eureka)。

6.3.4. 动态重新加载路由配置

我们来看看在在 Zuul 配置路由的下一件事是如何动态重新加载路由。动态重新加载路由的能力非常有用，因为它允许你改变路由的映射而不必回收 Zuul 服务器。现有的路由可以被快速修改，并通过你的环境中的每个 Zuul 服务器的回收行为添加新的路由。在第 3 章中，我们讨论了如何使用 Spring Cloud 配置服务来呈现微服务配置数据。你可以使用 Spring Cloud 配置呈现 Zuul 路由。在 EagleEye 的例子，你可以在你的配置仓库 (http://github.com/carnellj/config-repo)创建一个称为 zuulservice 的新的应用程序文件夹。像你的组织服务和许可服务，你将创建三个文件：zuulservice.yml，zuulservicedev.yml 和 zuulservice-prod.yml，它们将保存你的路由配置。

为了与第 3 章配置中的示例一致，我已经更改了路由格式，将其从分层格式调整到“.”格式。初始路由配置中只有一个条目：

```
zuul.prefix=/api
```

如果你点击/routes 端点，你应该看到当前显示在 Zuul，用/api 作为前缀的所有基于 Eureka 的服务。现在，如果你想立即添加新的路由映射，你只需对配置文件进行更改，然后将它们提交给 Git 仓库，其中 Spring Cloud 配置正在从配置数据库中提取配置数据。例如，如果你想禁用所有基于 Eureka 服务注册并且只暴露两个路由（组织服务和许可服务），你可以像这样修改 zuulservice-*.yml 文件：

```
zuul.ignored-services: '*'
zuul.prefix: /api
zuul.routes.organizationservice: /organization/**
zuul.routes.organizationservice: /licensing/**
```

然后你可以向 GitHub 提交修改。Zuul 以 POST 方式暴露/refresh 端点路由，该端点会重新加载其路由的配置。一旦这个/refresh 被点击，如果你再点击/routes 端点，你将会看到两个新的路由被暴露，并且所有基于 Eureka 的路由都消失了。

6.3.5. Zuul 和服务超时

Zuul 使用 Netflix 的 Hystrix 和 Ribbon 库来帮助防止长时间运行的服务调用影响服务网关的性能。默认情况下，如果任何调用处理一次请求需要消耗超过 1 秒（这是 Hystrix 的默认值。），Zuul 将中断调用并返回 HTTP 500 错误。幸运的是，你可以通过在 Zuul 服务器的配置中设置 Hystrix 超时属性来配置这样的行为。

为经过 Zuul 的所有服务设置 Hystrix 超时时间，你可以使用 hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds 属性。例如，如果你想设置默认 Hystrix 时间为 2.5 秒，你可以在你的 Zuul 的 Spring Cloud 配置文件使用以下配置：

```
zuul.prefix: /api
zuul.routes.organizationservice: /organization/**
zuul.routes.licensingservice: /licensing/**
zuul.debug.request: true
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 2500
```

如果需要为特定服务设置 Hystrix 超时，则可以使用你想要覆盖的超时时间，替换服务的 Eureka 服务 ID 名称属性的默认部分。例如，如果你想修改仅有 licensingservice 的超时时间为 3 秒，并且让其它剩余的服务使用默认 Hystrix 超时时间，你可以使用这样的配置：

```
hystrix.command.licensingservice.execution.isolation.thread.timeoutInMilliseconds: 3000
```

最后，你需要知道另一个超时属性。虽然你已经覆盖了 Hystrix 超时时间，但 Netflix 的 Ribbon 对任何耗时超过 5 秒的调用也会超时。我强烈建议你重新审视任何耗时超过 5 秒的调用的设计，你可以通过设置以下属性覆盖 Ribbon 超时时间：servicename.ribbon.ReadTimeout。例如，如果你想覆盖 licensingservice 的超时时间为 7 秒，你会使用以下配置：

```
hystrix.command.licensingservice.execution.isolation.thread.timeoutInMilliseconds:
7000
licensingservice.ribbon.ReadTimeout: 7000
```

> ***注意***：配置的时间超过 5 秒，你必须同时设置 Hystrix 和 Ribbon 的超时时间。

6.4. Zuul 过滤器

同时能够代理所有请求通过 Zuul 网关允许你简化你的服务调用，当你想写可以应用于流经网关的所有的服务调用的定制逻辑，Zuul 的真正优势就体现出来了。通常，这种定制逻辑用于强制一致的应用程序策略集，如：安全、日志记录和服务跟踪。

这些应用程序策略被认为是横切关注点，因为你希望将它们应用到应用程序中的所有服务，而不必修改每个服务来实现它们。在这种方式中，Zuul 过滤器可以使用同样的方式作

为一个 J2EE Servlet 过滤器或 Spring 切面，它可以拦截一个广泛的行为和装饰或修改调用行为而没有原始编码器知道该变化。当一个 Servlet 过滤器或 Spring 切面被定位到一个特定的服务，使用 Zuul 和 Zuul 过滤器允许你实现通过 Zuul 路由的所有服务的横切关注点。

Zuul 允许你在 Zuul 网关使用过滤器创建自定义的逻辑。过滤器允许你实现每个服务请求在实现时通过的业务逻辑链。

Zuul 支持三种类型的过滤器：

- 前置过滤器：在 Zuul，实际请求到达目标发生之前，前置过滤器被调用。前置过滤器通常执行确保服务有一个一致的消息格式的任务（例如，HTTP 头部的 key 已就位）或作为一个看门人确保用户调用服务已认证（他们说他们是谁）和授权（他们可以做他们请求要做的）。
- 后置过滤器：在调用目标服务之后，调用一个后置过滤器，并将响应发送回客户端。通常会实现一个后置过滤器，将从目标服务返回的响应记录下来，处理错误，或者对响应的敏感信息进行审核。
- 路由过滤器：路由过滤器被用于在调用目标服务之前拦截该调用。通常使用路由过滤器来确定是否需要进行某种级别的动态路由。例如，在本章后面，你将使用路由级过滤器，它将在同一服务的两个不同版本之间进行路由，以便将一小部分对服务的调用路由到一个新版本的服务，而不是现有的服务。这将允许你将新功能暴露给少量用户，而不必让每个人都使用新服务。

图 6.11 显示了在处理服务客户端的请求时，前置、后置和路由过滤器是如何组合在一起的。

① Pre-route filters are executed as the incoming request comes into Zuul.

当传入的请求进入 Zuul，前置过滤器被执行。

② Route filters allow you to override Zuul's default routing logic and will route a

user to where they need to go.

路由过滤器允许你覆盖 Zuul 的默认路由逻辑和将用户路由到他们需要去的地方。

③ A route filter may dynamically route services outside Zuul.

一个路由过滤器可以在 Zuul 动态路由服务。

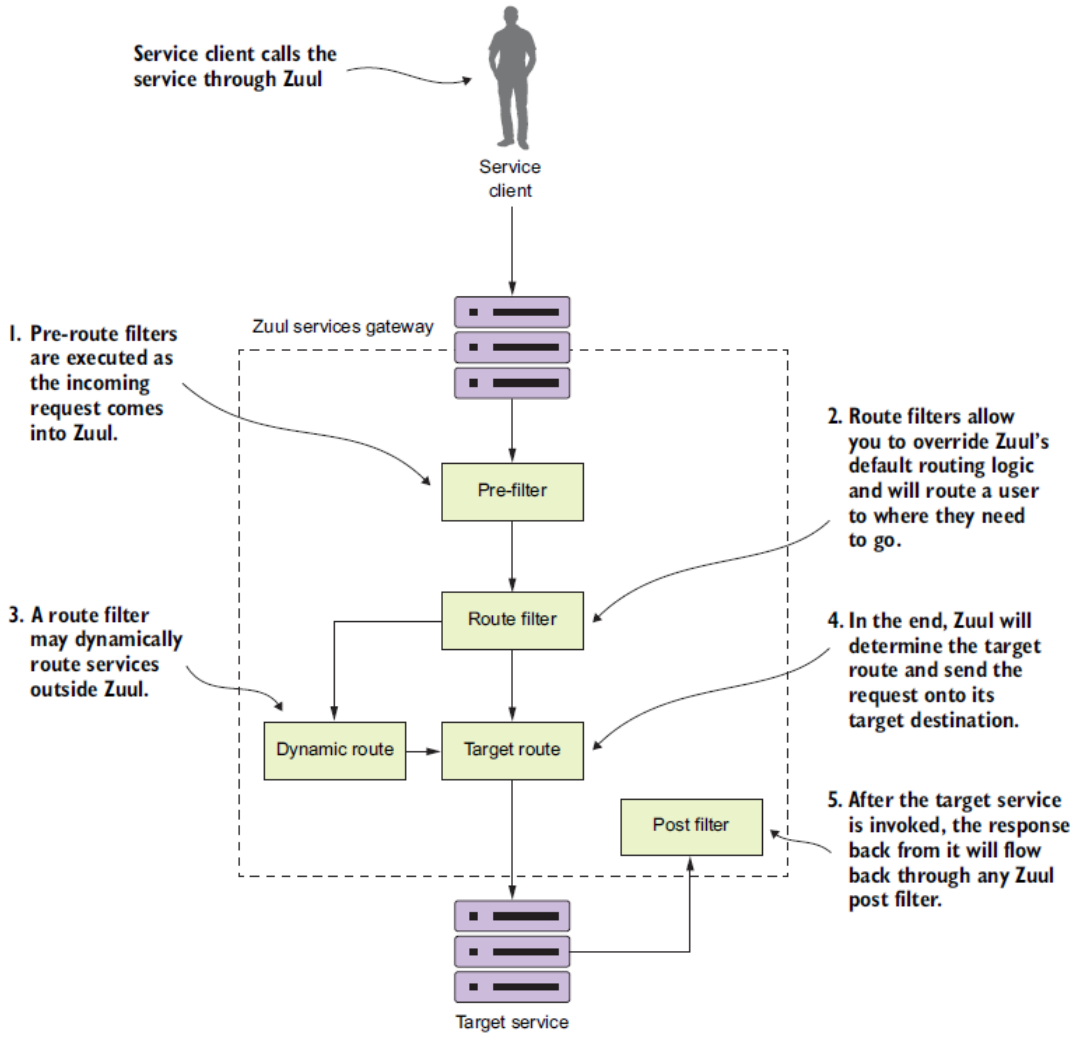

图 6.11 前置、路由和后置过滤器形成一个客户端请求流通过的管道。作为一个请求进入 Zuul，这些过滤器可以处理传入的请求。

④ In the end, Zuul will determine the target route and send the request onto its target destination.

最后，Zuul 将确定目标路由并发送请求到目标。

⑤ After the target service is invoked, the response back from it will flow back

through any Zuul post filter.

在目标服务被调用之后，返回响应将回流通过任何 Zuul 后置过滤器。

⑥ Service client calls the service through Zuul

服务客户端通过 Zuul 调用服务

如果你遵循图 6.11 中所示的流程，你将看到一切都始于服务客户端通过服务网关暴露的服务发出的调用。在那里进行下列活动：

- 任何定义在 Zuul 网关的前置过滤器通过 Zuul 作为请求进入 Zuul 网关被调用。前置过滤器可以在到达实际服务之前检查和修改 HTTP 请求。前置过滤器不能将用户重定向到不同的端点或服务。
- 前置过滤器通过 Zuul 传入的请求被执行之后 ,Zuul 将执行任何定义的路由过滤器。路由过滤器可以更改服务所在的目的地。
- 如果一个路由过滤想重定向服务调用到另外的地方，那里是 Zuul 服务器被配置为发送路由的地方 ,它可以这样做。然而 ,一个 Zuul 路由过滤器不做 HTTP 重定向，而是终止传入的 HTTP 请求 ,然后调用代表原始调用者的路由。这意味着路由过滤器必须完全拥有动态路由的调用，不能执行 HTTP 重定向。
- 如果路由过滤器不动态重定向调用者对一个新的路由，Zuul 服务器将发送路由到原来的目标服务。
- 目标服务被调用后，Zuul 后置过滤器将被调用。后置过滤器可以检查和修改从调用的服务返回的响应。

了解如何实现 Zuul 过滤器的最好方式是在使用中领会。为此，在接下来的几节中，你将构建一个前置、路由和后置过滤器，然后通过它们运行服务客户端请求。

图 6.12 显示了在处理你的 EagleEye 服务的请求时如何将这些过滤器组合在一起。

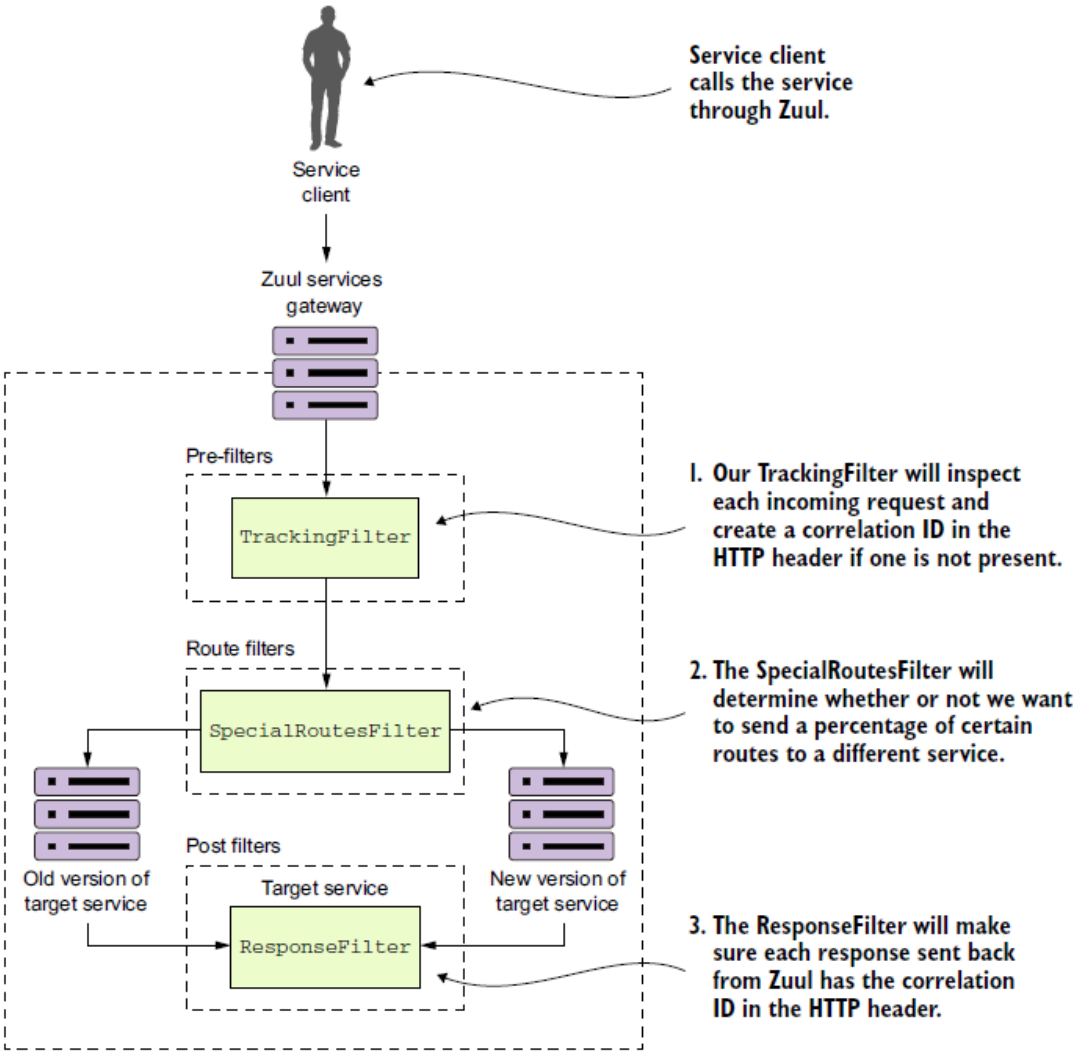

图 6.12 Zuul 过滤器提供集中的服务调用跟踪、日志记录和动态路由。Zuul 过滤器允许你在微服务调用之前执行自定义的规则和策略。

① *Our TrackingFilter will inspect each incoming request and create a correlation ID in the HTTP header if one is not present.*

我们的 TrackingFilter（跟踪过滤器）将检查每个传入的请求，若在 HTTP 头不存在关联 ID，则创建一个关联 ID。

② *The SpecialRoutesFilter will determine whether or not we want to send a percentage of certain routes to a different service.*

SpecialRoutesFilter（特定的路由过滤器）将决定我们是否要将一定数量的路由发送到不同

的服务。

③ The ResponseFilter will make sure each response sent back from Zuul has the correlation ID in the HTTP header.

ResponseFilter(响应过滤器)将确保从 Zuul 返回的每个响应在 HTTP 头已经包含关联 ID。

④ Service client calls the service through Zuul.

服务客户端通过 Zuul 调用服务。

⑤ Old version of target service

旧版本的目标服务

⑥ New version of target service

新版本的目标服务

在图 6.12 的流程之后，你将看到正在使用的过滤器：

- TrackingFilter：TrackingFilter 将是一个前置过滤器，它确保从 Zuul 流入的每个请求都有一个与它相关的关联 ID。关联 ID 是一个唯一的 ID，它在所有当执行一个客户请求时被执行的微服务间传输。关联 ID 允许你跟踪事件链， 该事件链出现在一个调用通过一系列的微服务调用时。
- SpecialRoutesFilter：SpecialRoutesFilter 是一个 Zuul 路由过滤器，它将检查传入的路由并确定你是否想在路由上进行 A/B 测试。A/B 测试是一种技术，其中一个用户(在这种情况下是一个服务)随机地使用同一服务提供两种不同版本的服务。A/B 测试背后的想法是 ,在将新功能发布到整个用户群体之前 ,可以对其进行测试。在我们的示例中，你将拥有同一组织服务的两个不同版本。少数用户将被路由到较新版本的服务，而大多数用户将被路由到旧版本的服务。
- ResponseFilter：ResponseFilter 是一个后置过滤器，它将与服务调用相关的关

联 ID 注入到发送给客户端的 HTTP 响应头中。通过这种方式，客户端将访问与它们请求相关的关联 ID。

6.5. 创建 pre 类型 Zuul 过滤器生成关联 ID

在 Zuul 构建过滤器是一个非常简单的活动。首先，你要创建一个 Zuul 预过滤器，称为 TrackingFilter，它将检查所有传入到网关的请求并确定在请求里是否有一个称为 tmx-correlationid 的存在于 HTTP 头。tmx-correlation-id 头将包含一个独特的 GUID(全球通用的 ID)，它被用于跟踪跨多个微服务的一个用户请求。

> ***注意***：我们在第 5 章中讨论了关联 ID 的概念。在这里，我们要对如何使用 Zuul 产生关联 ID 进行比较详细的介绍。如果你在书中跳过了这些章节，我强烈推荐你看看第 5 章和读一读关于 Hystrix 和线程上下文的部分。关联 ID 将使用 ThreadLocal 变量实现 ,ThreadLocal 变量与 Hystrix 一起工作会有一些额外的工作要做。

如果在 HTTP 头不存在 tmx-correlation-id，TrackingFilter 过滤器将生成和设置关联 ID。如果关联 ID 已经存在，Zuul 不会为关联 ID 做任何事情。关联 ID 的存在意味着这个特定服务调用是执行用户请求的服务调用链的一部分。在这种情况下，TrackingFilter 类将什么也不做。

让我们继续看看下面清单中 TrackingFilter 的实现。这个代码也可以在本书的示例 zuulsvr/src/main/java/com/thoughtmechanix/zuulsvr/filters/TrackingFilter.java 中找到。

清单 6.6 Zuul 前置过滤器生成关联 ID

```
package com.thoughtmechanix.zuulsvr.filters;

import com.netflix.zuul.ZuulFilter;
import org.springframework.beans.factory.annotation.Autowired;

@Component
```

```
public class TrackingFilter extends ZuulFilter{
   private static final int FILTER_ORDER = 1;
   private static final boolean SHOULD_FILTER=true;
   private static final Logger logger =
        LoggerFactory.getLogger(TrackingFilter.class);

   @Autowired
   FilterUtils filterUtils;

   @Override
   public String filterType() {
      return FilterUtils.PRE_FILTER_TYPE;
   }

   @Override
   public int filterOrder() {
      return FILTER_ORDER;
   }

   public boolean shouldFilter() {
      return SHOULD_FILTER;
   }

   private boolean isCorrelationIdPresent(){
    if (filterUtils.getCorrelationId() !=null){
       return true;
    }
    return false;
   }

   private String generateCorrelationId(){
      return java.util.UUID.randomUUID().toString();
   }

   public Object run() {
      if (isCorrelationIdPresent()) {
        logger.debug("tmx-correlation-id found in tracking filter: {}.",
        filterUtils.getCorrelationId());
      }
      else{
        filterUtils.setCorrelationId(generateCorrelationId());
        logger.debug("tmx-correlation-id generated in tracking filter: {}.", filterUtils.getCorrelationId());
      }
      RequestContext ctx = RequestContext.getCurrentContext();
```

①所有 Zuul 过滤器必须扩展 ZuulFilter 类和重写的四种方法：filterType()，filterOrder()，shouldFilter() 和 run()。

②你所有的过滤器常用的功能都被封装在 FilterUtils 类。

③filterType()方法告诉 Zuul 过滤器是一个前置，路由或后置过滤器。

③filterOrder()方法返回一个整型值表示 Zuul 应该通过不同的过滤器类型发送请求的顺序。

④shouldFilter()方法返回一个布尔值，表示过滤器是否应该处于活动状态。

⑤它是实际检查 tmx-correlation-id 是否存在的辅助方法，并且也可以使用 GUIID 值生成一个关联 ID

⑥run()方法的代码，每一次服务通过过滤器被执行。在 run()方法中，你看看 tmxcorrelation-id 是否存在，如果不存在，你生成一个相关值并将其设置在 HTTP 的 tmx-correlation-id

```
        logger.debug("Processing incoming request for {}.", ctx.getRequest().getRequestURI());
        return null;
    }
}
```

在 Zuul 实现一个过滤器，你必须继承 ZuulFilter 类，然后覆盖四个方法：filterType()，filterOrder()，shouldFilter()和 run()。此清单中的前三个方法描述你将在 Zuul 构建什么类型的过滤器，它与其它类型的过滤器相比应该以什么顺序运行，以及它是否应该是活动的。最后一种方法，run()，包含过滤器将实现的业务逻辑。

你已经实现了一个称为 FilterUtils 的类。这个类用于封装所有过滤器所使用的公共功能。FilterUtils 类位于 zuulsvr/src/main/java/com/thoughtmechanix/zuulsvr/FilterUtils.java。我们不讨论整个 FilterUtils 类，但我们在这里讨论关键方法：getCorrelationId() 和 setCorrelationId()。下面的清单显示 FilterUtils 类的 getCorrelationId()方法的代码。

清单 6.7 从 HTTP 头检索 tmx-correlation-id

```
public String getCorrelationId(){
    RequestContext ctx = RequestContext.getCurrentContext();

    if (ctx.getRequest().getHeader(CORRELATION_ID) !=null) {
        return ctx.getRequest().getHeader(CORRELATION_ID);
    }
    else{
        return ctx.getZuulRequestHeaders().get(CORRELATION_ID);
    }
}
```

在清单 6.7 中注意的关键点是，你首先查看 tmx-correlation-id 是否已在传入请求的 HTTP 头中设置。你使用 ctx.getRequest().getHeader(CORRELATION_ID)调用来完成此操作。

注意：在传统的 Spring MVC 或 Spring Boot 服务，RequestContext 应该是 org.springframework.web.servletsupport.RequestContext 类型。然而，Zuul 提供了一个专用的 RequestContext，它有几个用于访问 Zuul 特定值的额外方法。这个请求上下文

是 com.netflix.zuul.context 包的一部分。

如果它不存在，你再检查 ZuulRequestHeaders。Zuul 不允许你直接添加或修改一个传入的请求的 HTTP 请求头。如果我们添加了 tmx-correlation-id，然后尝试在过滤器后面再次访问它，它作为 ctx.getRequestHeader()调用部分将不可用。你可能还记得，在前面 TrackingFilter 类的 run()方法里，你使用以下代码片段完成了这一工作。

```
else{
   filterUtils.setCorrelationId(generateCorrelationId());
   logger.debug("tmx-correlation-id generated in tracking filter: {}.",    filterUtils.getCorrelationId());
}
```

使用 FilterUtils 的 setCorrelationId()方法设置 tmx-correlation-id。

```
public void setCorrelationId(String correlationId){
   RequestContext ctx = RequestContext.getCurrentContext();
   ctx.addZuulRequestHeader(CORRELATION_ID, correlationId);
}
```

在 FilterUtils 类的 setCorrelationId()方法，当你想对 HTTP 请求头添加值的时候，你用 RequestContext 的 addZuulRequestHeader()方法。该方法将为 HTTP 头维护一个单独的 map，当请求在 Zuul 服务器流经过滤器时被添加。当目标服务由你的 Zuul 服务器调用时，在 ZuulRequestHeader 的 map 里面包含的数据将被合并。

6.5.1. 在服务调用中使用关联 ID

现在你已经保证关联 ID 已被添加到流经 Zuul 的每个微服务调用，你如何确保：

- 被调用的微服务易于访问关联 ID
- 任何下游服务调用微服务也可能将关联 ID 传播到下游调用

为了实现这一点，你要在你的每个微服务中创建三个类。这些类将一起从传入的 HTTP 请求中读取关联 ID（以及你之后添加的其他信息），将它映射到一个类以易于理解，且通过应用程序中的业务逻辑方便使用，并确保关联 ID 传播到下游的任何服务调用。

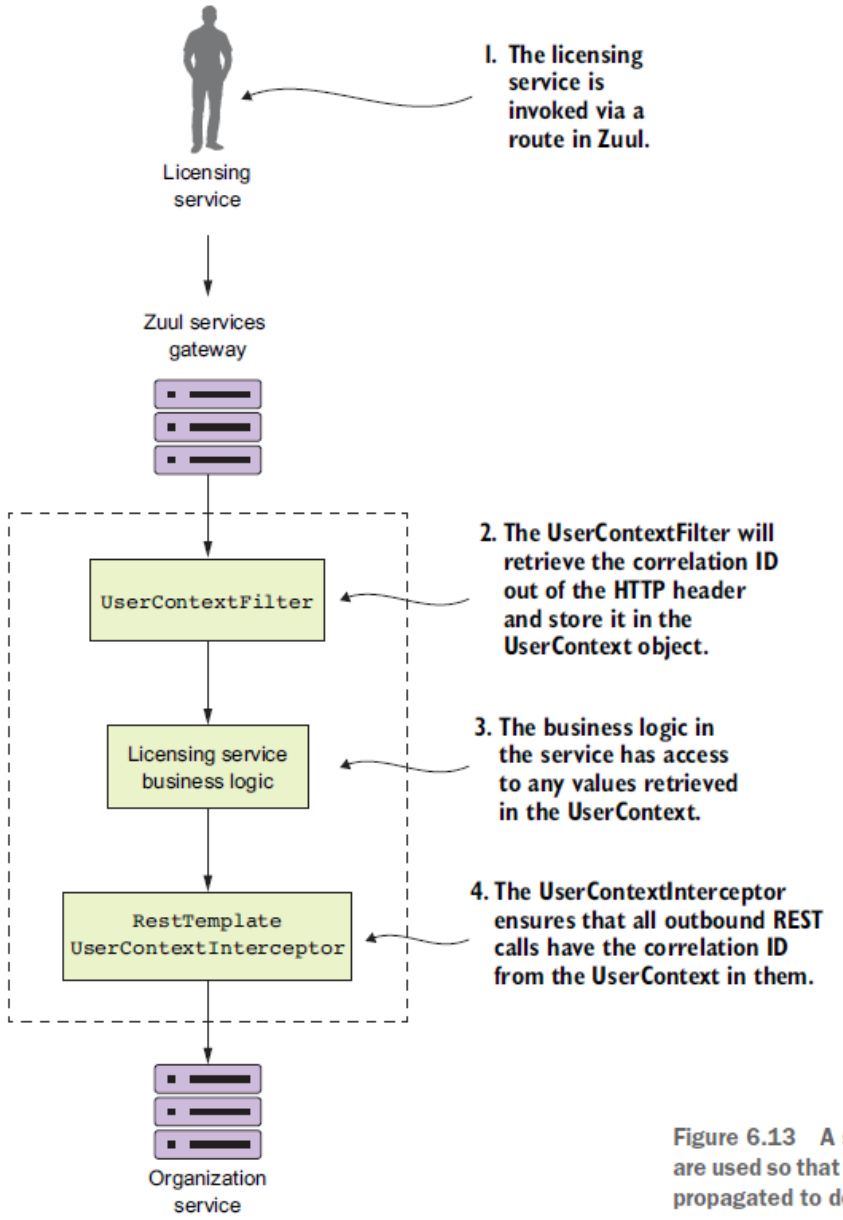

Figure 6.13 A set of common classes are used so that the correlation ID can be propagated to downstream service calls.

图 6.13 使用了一组公共类，以便将关联 ID 传播到下游服务调用。

① *The licensing service is invoked via a route in Zuul.*

许可服务通过 Zuul 里的路由被调用。

② *The UserContextFilter will retrieve the correlation ID out of the HTTP header and store it in the UserContext object.*

UserContextFilter 将检索 HTTP 头信息的关联 ID 并将其存储在 UserContext 对象。

③ *The business logic in the service has access to any values retrieved in the*

UserContext.

服务中的业务逻辑访问从 UserContext 取回的任意值。

④ The UserContextInterceptor ensures that all outbound REST calls have the correlation ID from the UserContext in them.

UserContextInterceptor 确保所有对外的 REST 调用在它们的 UserContext 对象中存在关联 ID。

图 6.13 演示了如何使用许可服务来构建这些不同的部分。

让我们看看图 6.13 中所发生的事情：

- 当对许可服务的调用通过 Zuul 网关时，TrackingFilter 将为进入 Zuul 的任何调用在传入的 HTTP 头中注入一个关联 ID。
- UserContextFilter 类是一个自定义的 HTTP Servlet 过滤器。它映射一个关联 ID 到 UserContext 类。UserContext 类被用于存储调用里后续使用的本地线程存储的值。
- 许可服务业务逻辑需要执行对组织服务的调用。
- RestTemplate 被用于调用组织服务。RestTemplate 将使用自定义的 Spring 拦截器类（UserContextInterceptor）注入关联 ID 到外部调用，作为一个 HTTP 头。

重复的代码 vs. 共享库

你是否应该在微服务使用公共库的主题是微服务设计是一个很难界定的问题。微服务纯粹主义者会告诉你，你不应该在你的服务使用一个自定义的框架，因为它在你的服务中引入了人为的依赖。业务逻辑的修改或 bug 可能导致对所有服务进行大规模的重构。另一方面，其他的微服务从业者会说，一个纯粹的方法是不切实际的，某些情况下存在是有意义的（想前面的 UserContextFilter 示例），创建一个公共库并在服务间共享它。

我认为这里有妥协。在处理基础设施风格的任务时，公共库很好。如果你开始共享面向业务的类，那么你就是在自找麻烦，因为你正在打破服务之间的界限。

我似乎在用本章中的代码示例打破我自己的建议，因为如果你查看本章中的所有服务，它们（UserContextFilter，UserContext 和 UserContextInterceptor 类）都有自己的副本。我在这里使用一个非共享方法的原因是我不想通过创建一个共享库而使在本书中的代码示例变得复杂，因为共享库将需要发布到第三方的 Maven 仓库。因此，在服务的 utils 包中的所有类在所有服务中共享。

USERCONTEXTFILTER: 拦截传入 HTTP 请求

你将要创建的第一个类是 UserContextFilter 类。这个类是一个 HTTP Servlet 过滤器，用来拦截进入服务的所有 HTTP 请求，并从 HTTP 请求映射关联 ID（和一些其它的值）到 UserContext 类。下面的清单显示 UserContext 类的代码。这个类的源代码可以在 licensing-service/src/main/java/com/thoughtmechanix/licenses/utils/UserContextFilter.java 中找到。

清单 6.8 映射关联 ID 到 UserContext 类

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

@Component
public class UserContextFilter implements Filter {
  private static final Logger logger = LoggerFactory.getLogger(UserContextFilter.class);

  @Override
  public void doFilter(ServletRequest servletRequest,
                       ServletResponse servletResponse,
                       FilterChain filterChain)
    throws IOException, ServletException {
    HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;

    UserContextHolder.getContext()
                     .setCorrelationId(
                     ➥httpServletRequest
                     .getHeader(UserContext.CORRELATION_ID));
```

①由 Spring 通过使用 Spring 的@Component 注解和实现 javax.servler.Filter 接口，过滤器被注册并接收。

②你的过滤器从头部检索关联 ID 并在 UserContext 类设置该值。

```
        UserContextHolder.getContext().setUserId(
            httpServletRequest
              .getHeader(UserContext.USER_ID));

        UserContextHolder
              .getContext()
              .setAuthToken(
                    httpServletRequest
                      .getHeader(UserContext.AUTH_TOKEN) );

        UserContextHolder
              .getContext()
              .setOrgId(
                   httpServletRequest
                     .getHeader(UserContext.ORG_ID) );

        filterChain.doFilter(httpServletRequest, servletResponse);
}
```

③如果你使用了代码的 README 文件定义的认证服务示例，那么从 HTTP 报头中删除其它的值将起作用。

最后，UserContextFilter 被用于映射 HTTP 头你感兴趣的值到一个 Java 类（UserContext）。

USERCONTEXT：使服务易于访问 HTTP 头

UserContext 类被用于存储通过微服务正在处理的单独的服务客户端请求 HTTP 头的值。它由一个 getter 和 setter 方法组成，用来从 java.lang.ThreadLocal 检索和存储值。下面的清单显示 UserContext 类的代码。这个类的源代码可以在 licensing-service/src/main/java/com/thoughtmechanix/licenses/utils/UserContext.java 中找到。

清单 6.9 在 UserContext 里存储 HTTP 头的值

```
@Component
public class UserContext {
    public static final String CORRELATION_ID = "tmx-correlation-id";
    public static final String AUTH_TOKEN = "tmx-auth-token";
    public static final String USER_ID = "tmx-user-id";
    public static final String ORG_ID = "tmx-org-id";

    private String correlationId= new String();
    private String authToken= new String();
    private String userId = new String();
    private String orgId = new String();
```

```
    public String getCorrelationId() {
        return correlationId;
    }

    public void setCorrelationId(String correlationId) {
        this.correlationId = correlationId;
    }

    public String getAuthToken() {
        return authToken;
    }

    public void setAuthToken(String authToken) {
        this.authToken = authToken;
    }

    public String getUserId() {
        return userId;
    }

    public void setUserId(String userId) {
        this.userId = userId;
    }

    public String getOrgId() {
        return orgId;
    }

    public void setOrgId(String orgId) {
        this.orgId = orgId;
    }
}
```

现在 UserContext 类仅仅是一个存储从传入的 HTTP 请求获取的值的 POJO 对象。你使用一个称为 zuulsvr/src/main/java/com/thoughtmechanix/zuulsvr/filters/UserContextHolder.java 的类在 ThreadLocal 变量存储 UserContext 对象，通过线程处理用户的请求，调用任何方法可以访问该对象。UserContextHolder 的代码如下面清单所示。

清单 6.10 UserContextHolder 在一个 ThreadLocal 存储 UserContext

```
public class UserContextHolder {
```

```
    private static final ThreadLocal<UserContext> userContext
        = new ThreadLocal<UserContext>();

    public static final UserContext getContext(){
          UserContext context = userContext.get();

          if (context == null) {
             context = createEmptyContext();
             userContext.set(context);
          }

          return userContext.get();
     }

     public static final void setContext(UserContext context) {
        Assert.notNull(context,    "Only non-null UserContext instances are permitted");
        userContext.set(context);
     }

     public static final UserContext createEmptyContext(){
          return new UserContext();
     }
}
```

自定义 RESTTEMPLATE 和 USERCONTEXTINTECEPTOR：确保关联 ID 的获取和向前传播

最后的代码部分，我们将看的是 UserContextInterceptor 类。这个类是用来注入关联 ID 到任何输出的基于 HTTP 的服务请求，该请求从 RestTemplate 实例中执行。这样做是为了确保你可以在服务调用之间建立联系。

为此你要使用一个 Spring 拦截器，它被注入 RestTemplate 类。让我们看看下面清单中的 UserContextInterceptor 类。

清单 6.11 所有的外部微服务调用有关联 ID 注入

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

public class UserContextInterceptor
   implements ClientHttpRequestInterceptor {

   @Override
```

①UserContextIntercept 实现 Spring 框架的 ClientHttpRequestInterceptor。

```
    public ClientHttpResponse intercept(
        HttpRequest request, byte[] body,
        ClientHttpRequestExecution execution)
      throws IOException {

      HttpHeaders headers = request.getHeaders();
      headers.add(UserContext.CORRELATION_ID,
        UserContextHolder.getContext().getCorrelationId());

      headers.add(UserContext.AUTH_TOKEN,
        UserContextHolder.getContext().getAuthToken());

      return execution.execute(request, body);
    }
}
```

②intercept()方法在实际的 HTTP 服务调用通过 RestTemplate 发生之前被调用。

③你获取正在准备外部服务调用的 HTTP 请求头，并在 UserContext 中存储关联 ID。

为了使用 UserContextInterceptor，你需要定义一个 RestTemplate bean 并且将 UserContextInterceptor 添加给它。要做到这一点，你要在 licensing-service/src/main/java/com/thoughtmechanix/licenses/Application.java 类中添加你自己的 RestTemplate bean 定义。下面的清单显示了添加到该类的方法。

清单 6.12 添加 UserContextInterceptor 到 RestTemplate 类

```
@LoadBalanced
@Bean
public RestTemplate getRestTemplate(){
  RestTemplate template = new RestTemplate();
  List interceptors = template.getInterceptors();
   if (interceptors==null){
      template.setInterceptors(
       Collections.singletonList(
          new UserContextInterceptor()));
   }
   else{
      interceptors.add(new UserContextInterceptor());
      template.setInterceptors(interceptors);
   }

   return template;
}
```

①@LoadBalanced 注解表明 RestTemplate 对象将使用 Ribbon。

②添加 UserContextInterceptor 到已创建的 RestTemplate 实例

使用 bean 定义，任何时候你使用@Autowired 注解将为一个类注入 RestTemplate，你可以使用清单 6.11 中已创建的、与 UserContextInterceptor 绑定的 RestTemplate。

日志聚合和认证等

现在你已经将关联 ID 传递给每个服务，当它流经调用中所涉及的所有服务时，跟踪一个交易是可能的。要做到这一点，你需要确保每个服务日志记录到一个集中的日志聚合点，它从所有服务中捕获日志条目到一个点。在日志聚合服务中捕获的每个日志条目都将有一个与每个条目相关联的关联 ID。实现日志聚合的解决方案超出了本章的范围，但在第 9 章中，我们将看到如何使用 Spring Cloud Sleuth。Spring Cloud Sleuth 不会使用你在这里创建的 TrackingFilter，但它会使用与跟踪关联 ID 相同的概念和确保在每个调用中关联 ID 都被注入。

6.6. 创建 post 类型 Zuul 过滤器接收关联 ID

记住，Zuul 代表服务客户端执行实际的 HTTP 调用。Zuul 有机会检查从目标服务调用返回的响应并修改或使用附加的信息装饰响应。当加上使用前置过滤器采集数据时，一个 Zuul 后置过滤器是收集数据和完成与用户的交易相关的任何记录的理想地点。你要利用这个来注入关联 ID，你已经通过你的微服务返回到用户。

你打算通过使用一个 Zuul 后置过滤器注入关联 ID 到正在被传递回给服务调用者的 HTTP 响应头。通过这种方式，你可以将关联 ID 传递给调用者，而无需接触消息体。下面的清单显示了创建一个后置过滤器的代码。此代码可以在 zuulsvr/src/main/java/com/thoughtmechanix/zuulsvr/filters/ResponseFilter.java 中找到。

清单 6.13 将关联 ID 注入 HTTP 响应

```
package com.thoughtmechanix.zuulsvr.filters;

@Component
public class ResponseFilter extends ZuulFilter{
   private static final int FILTER_ORDER=1;
   private static final boolean SHOULD_FILTER=true;
```

```
    private static final Logger logger =LoggerFactory.getLogger(ResponseFilter.class);

    @Autowired
    FilterUtils filterUtils;

    @Override
    public String filterType() {          ①过滤器类型是 POST_FILTER_TYPE。
        return FilterUtils.POST_FILTER_TYPE;
    }

    @Override
    public int filterOrder() {
        return FILTER_ORDER;
    }

    @Override
    public boolean shouldFilter() {
        return SHOULD_FILTER;
    }

    @Override
    public Object run() {
        RequestContext ctx =RequestContext.getCurrentContext();
        logger.debug("Adding the correlation id to the outbound headers. {}",
        ➥filterUtils.getCorrelationId());

                                  ②获取在原始 HTTP 请求中传递的关联 ID 并将其注入响应中。
        ctx.getResponse()
          .addHeader(FilterUtils.CORRELATION_ID, filterUtils.getCorrelationId());

        logger.debug("Completing outgoing request for {}.", ctx.getRequest().getRequestURI());
        return null;
    }
                ③记录输出的请求 URI，你有“书挡”，将显示进入 Zuul 的用户的请求传入和传出的入口。
}
```

一旦 ResponseFilter 已经实现，你可以启动你的 Zuul 服务，并通过它调用 EagleEye 的许可服务。一旦服务完成后，你会在调用的 HTTP 响应头看到一个 tmx-correlation-id。图 6.14 显示的是从调用发送回的 tmx-correlation-id。

到目前为此，我们所有的过滤器示例都处理了在路由到目标目的地之前和之后的服务客户端调用。对于最后一个过滤器示例，让我们看看如何动态地更改你要将用户发送到的目标路由。

① The correlation ID returned in the HTTP response

HTTP 响应中返回的关联 ID

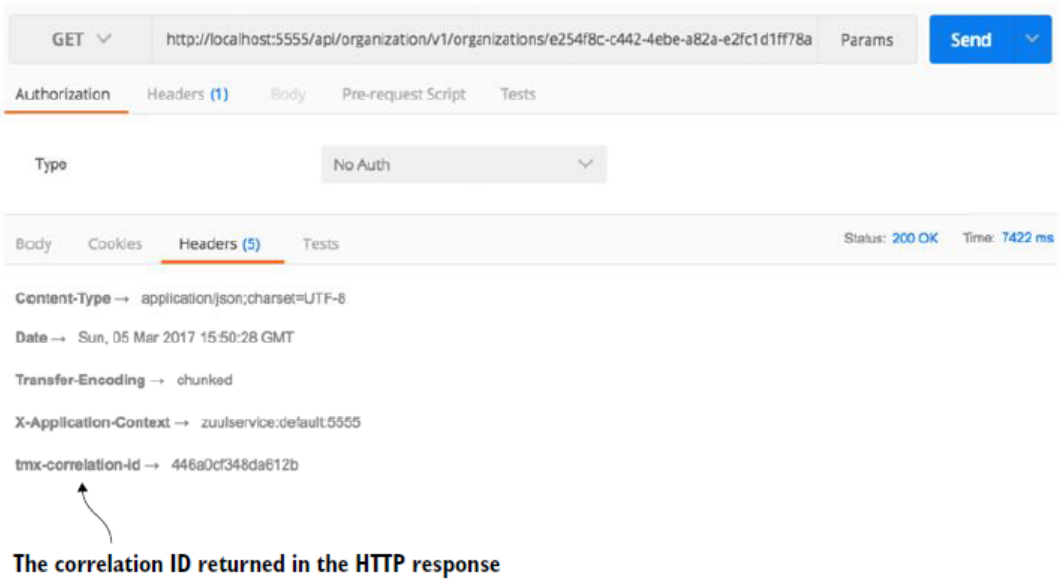

图 6.14 tmx-correlation-id 已被添加到发送回服务客户端的响应头。

6.7. 创建动态路由过滤器

最后的 Zuul 过滤器，我们要看的是 Zuul 路由过滤器。如果没有一个自定义的路由过滤器，Zuul 将你在本章前面看到的基于映射的定义来完成路由。然而，通过构建一个 Zuul 路由过滤器，你可以为服务客户端调用将如何路由增加智能。

在这一部分中，你将通过创建一个路由过滤器了解 Zuul 的路由过滤器，这将将允许你做服务新版本的 A/B 测试。A/B 测试是在你推出一个新功能，然后有一部分用户使用该特性。其余的用户仍然使用旧服务。在本例中，你将模拟出一个组织服务的新版本，你希望其中 50%的用户转到旧服务，50%的用户转到新服务。

为此你需要创建一个称为 SpecialRoutesZuulFilter 的 Zuul 路由过滤器，它将获取通过 Zuul 被调用的服务的 Eureka 服务 ID，并调用其它称为 SpecialRoutes 的微服务。SpecialRoutes 服务将检查一个内部数据库看服务名称是否存在。如果目标服务名称存在，

它将返回一个权重和服务的另一个位置的目标目的地。SpecialRoutesFilter 将获取返回的权重，并根据权重随机生成一个数，这个数将被用来确定用户的调用将被路由到其它的组织服务或在 Zuul 路由映射定义的组织服务。图 6.15 显示了当 SpecialRoutesFilter 被使用时发生的流程。

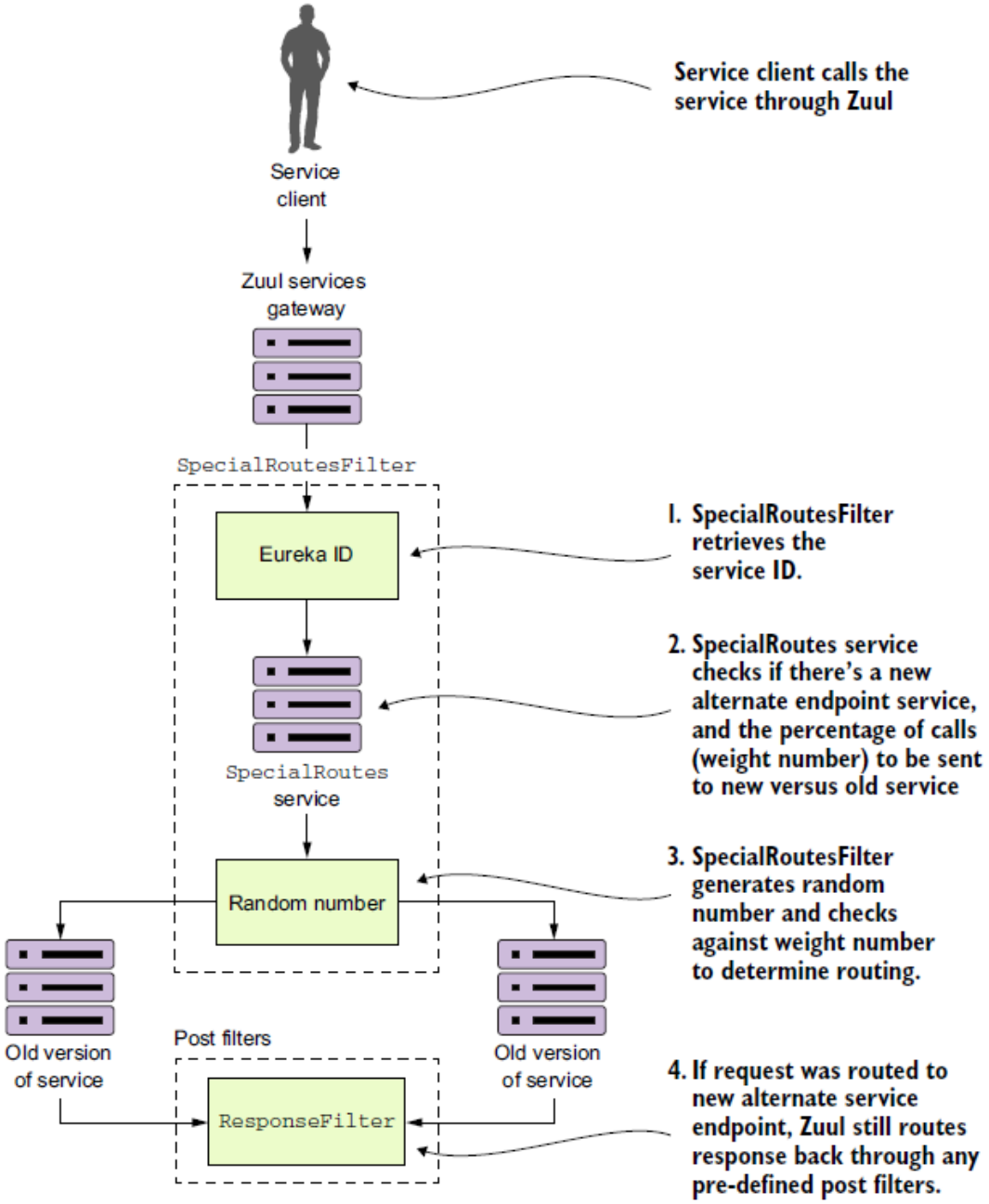

图 6.15 通过 SpecialRoutesFilter 调用组织服务的流程

① *SpecialRoutesFilter retrieves the service ID.*

SpecialRoutesFilter 检索服务 ID。

② SpecialRoutes service checks if there’s a new alternate endpoint service, and the percentage of calls (weight number) to be sent to new versus old service

SpecialRoutes 服务检查是否有新的替代端点服务，且一部分调用（权重）被分别指派到新或旧的服务。

③ SpecialRoutesFilter generates random number and checks against weight number to determine routing.

SpecialRoutesFilter 生成随机数并根据权重来确定路由。

④ If request was routed to new alternate service endpoint, Zuul still routes response back through any pre-defined post filters.

如果请求被路由到新的替代服务端点，Zuul 仍然通过任何预定义的后置过滤器路由返回响应。

在图 6.15 中，在服务客户端通过 Zuul 调用一个“前端”服务之后，SpecialRoutesFilter 采取以下行动：

- SpecialRoutesFilter 检索正在被调用的服务的服务 ID。
- SpecialRoutesFilter 调用 SpecialRoutes 服务。SpecialRoutes 服务检查是否有为目标端点替代的端点定义。如果记录被发现，它包含的权重将会告诉 Zuul，应按服务调用的百分比分别发送到旧服务和新服务。
- 然后，SpecialRoutesFilter 生成一个随机数，并将其与 SpecialRoutes 服务返回的权重进行比较。如果随机生成的数字小于替代端点的权重，SpecialRoutesFilter 发送请求到服务的新版本。
- 如果 SpecialRoutesFilter 将请求发送到服务的新版本，Zuul 保持原有预定义的管

道和通过任何定义的后置过滤器发送从替代服务端点返回的响应。

6.7.1. 构建路由过滤器的框架

我们要开始介绍你用来创建 SpecialRoutesFilter 的代码。到目前为止，我们看到的所有过滤器，实现一个 Zuul 路由过滤器需要最多的编码工作，因为由一个路由过滤器接管 Zuul 的核心部分功能，路由和使用你自己的功能替代它。我们不打算详细讨论整个类，而是仔细研究相关的细节。

SpecialRoutesFilter 遵循与其它 Zuul 过滤器一样的基本模式。它扩展了 ZuulFilter 类并设置 filterType()方法返回“route”值。我不会去考虑任何更多关于 filterOrder()和 shouldFilter()方法的解释，因为它们不同于在本章前面讨论的过滤器。下面的清单显示了路由过滤器框架。

清单 6.14 路由过滤器的框架

```
package com.thoughtmechanix.zuulsvr.filters;

@Component
public class SpecialRoutesFilter extends ZuulFilter {
   @Override
   public String filterType() {
      return filterUtils.ROUTE_FILTER_TYPE;
   }

   @Override
   public int filterOrder() {}

   @Override
   public boolean shouldFilter() {}

   @Override
   public Object run() {}
}
```

6.7.2. 实现 run 方法

SpecialRoutesFilter 的真正的工作开始在 run()方法的代码。下面的清单显示了此方法的代码。

清单 6.15 SpecialRoutesFilter 的 run()方法是工作开始的地方

```
public Object run() {
   RequestContext ctx = RequestContext.getCurrentContext();

   AbTestingRoute abTestRoute =
       getAbRoutingInfo( filterUtils.getServiceId() );

   if (abTestRoute!=null &&
     useSpecialRoute(abTestRoute)) {
    String route =
        buildRouteString(
          ctx.getRequest().getRequestURI(),
          abTestRoute.getEndpoint(),
          ctx.get("serviceId").toString());
    forwardToSpecialRoute(route);
    }

    return null;
}
```

①执行调用 SpecialRoutes 服务确定是否为本组织的路由记录

②useSpecialRoute()方法将获取路由的权重，生成一个随机数，并确定你是否要转发请求到替代服务。

③如果有一个路由记录，建立到服务位置完整的 URL（路径），该位置通过 SpecialRoutes 服务指定。

④forwardToSpecialRoute()方法完成转发到替代服务的工作。

清单 6.15 中的代码的一般流程是：当一个路由请求触发 SpecialRoutesFilter 的 run()方法，它将执行一个向 SpecialRoutes 服务的 REST 调用。此服务将执行查找，并确定被调用的目标服务的 Eureka 服务 ID 是否在路由记录中存在。在 getAbRoutingInfo()方法中调用 SpecialRoutes 服务。getAbRoutingInfo()方法如下面的清单所示。

清单 6.16 调用 SpecialRoutesService 看看路由记录是否存在

```
private AbTestingRoute getAbRoutingInfo(String serviceName){
   ResponseEntity<AbTestingRoute> restExchange = null;
   try {
    restExchange = restTemplate.exchange(
       "http://specialroutesservice/v1/route/abtesting/{serviceName}",
       HttpMethod.GET,null, AbTestingRoute.class, serviceName);
   }
   catch(HttpClientErrorException ex){
     if (ex.getStatusCode()== HttpStatus.NOT_FOUND){
       return null;
```

①调用 SpecialRoutesService 端点

②如果路由服务没有找到记录（它将返回 404 HTTP 状态代码），则该方法将返回 null。

```
            throw ex;
        }
        return restExchange.getBody();
}
```

一旦你已经确定有一个目标服务的路由记录，你需要确定你是否应该路由目标服务请求到替代服务的位置或由 Zuul 路由集合管理的静态默认服务位置。为了做这个决定，你调用 useSpecialRoute()方法。下面的清单显示了此方法。

清单 6.17 确定是否使用替代的服务路由

```
public boolean useSpecialRoute(AbTestingRoute testRoute){
   Random random = new Random();

   if (testRoute.getActive().equals("N"))  ←——— ①检查路由是否有效
      return false;

   int value =random.nextInt((10 - 1) + 1) + 1;  ←——— ②确定是否应使用替代的服务路由

   if (testRoute.getWeight()<value)
      return true;

   return false;
}
```

这个方法做了两件事。首先，该方法检查从 SpecialRoutes 服务返回的 AbTestingRoute 记录上的有效域。如果记录设置为“N”，useSpecialRoute()方法不做任何事情，因为在这一刻你不想做任何路由。第二，该方法生成一个在 1 到 10 之间的随机数。然后，该方法将检查返回路由的权重是否小于随机生成的数字。如果条件为真，则 useSpecialRoute 方法返回 true，表示你希望使用该路由。

一旦你确定要路由服务请求进入 SpecialRoutesFilter，你就要将请求转发到目标服务。

6.7.3. 转发路由

实际传递到下游服务的路由是大多数工作发生在 SpecialRoutesFilter 中的地方。而 Zuul 确实提供了辅助功能，使这项工作更容易，大部分的工作仍在开发者。

forwardToSpecialRoute()方法为你提供转发工作。此方法中的代码大量借鉴了 Spring Cloud SimpleHostRoutingFilter 类的源代码。虽然我们不打算讨论 forwardToSpecialRoute()方法中调用的所有辅助方法，我们将浏览这个方法中的代码，如下面的清单所示。

清单 6.18 forwardToSpecialRoute 调用替代的服务

```
private ProxyRequestHelper helper
    = new ProxyRequestHelper ();

private void forwardToSpecialRoute(String route) {
 RequestContext    context = RequestContext.getCurrentContext();
 HttpServletRequest    request = context.getRequest();

 MultiValueMap<String, String> headers =
 ➥helper.buildZuulRequestHeaders(request);

 MultiValueMap<String, String> params =
 ➥helper.buildZuulRequestQueryParams(request);

 String verb = getVerb(request);
 InputStream requestEntity = getRequestBody(request);
 if (request.getContentLength() < 0)
    context.setChunkedRequestBody();

 this.helper.addIgnoredHeaders();
 CloseableHttpClient httpClient = null;
 HttpResponse response = null;
 try {
  httpClient = HttpClients.createDefault();
  response = forward(
                httpClient,
               ➥verb,
                route,
                request,
                headers,
                params,
                requestEntity);
  setResponse(response);
 }
 catch (Exception ex ) {}
}
```

①辅助变量是一个类型为 ProxyRequestHelper 类的实例变量。这是 Spring Cloud 类为代理服务请求的辅助方法。

②创建将发送到服务的所有 HTTP 请求头的副本。

③创建所有 HTTP 请求参数的副本

④制作转发到替代服务的 HTTP 报文体的副本

⑤使用前面的辅助方法调用替代服务（未显示）

⑥服务调用的结果通过 setResponse()辅助方法保存到 Zuul 服务器。

清单 6.18 中的代码的关键之处在于，你将从传入的 HTTP 请求（头参数、HTTP 谓词和报文体）中复制所有值到在目标服务上被调用的新请求。forwardToSpecialRoute()方法获取从目标服务返回的响应并将其设置在用于 Zuul HTTP 请求上下文。这是通过 setResponse()辅助方法（未显示）完成的。Zuul 使用 HTTP 请求上下文返回从服务客户端调用返回的响应。

6.7.4. 把代码整合在一起

现在你已经实现了 SpecialRoutesFilter，你可以通过调用许可服务来看一看它的行为。你可能还记得前面的章节，许可服务调用组织服务来检索组织的联系人数据。

在代码示例中，specialroutesservice 有一条组织服务的数据库记录，将到组织服务的调用请求 50%路由到现有的组织服务（在 Zuul 映射）和 50%路由到另一个组织服务。从 SpecialRoutes 服务返回的替代的组织服务路由将是 http://orgservice-new，并不会直接从 Zuul 访问。为了区分两个服务，我已经修改了组织服务在组织服务返回的联系人名字值前面添加“OLD::”和“NEW::”文本。

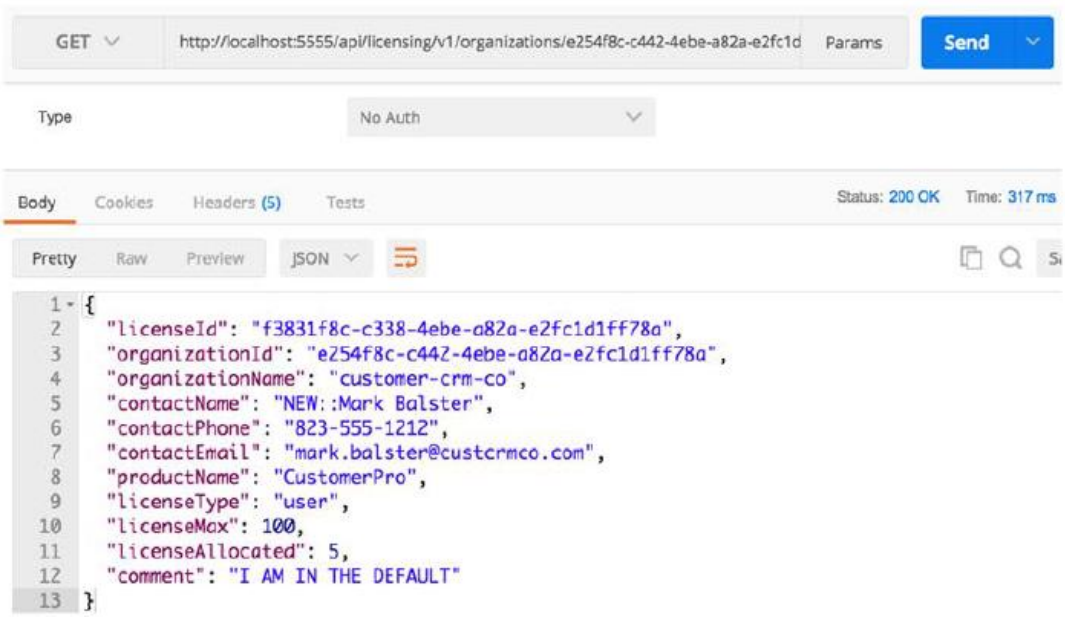

图 6.16 当你点击替代的组织服务，你会看到联系人名字值有 NEW 前缀。

如果你现在通过 Zuul 点击许可服务端点

http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82ae2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a

你应该看到从许可服务调用返回的 contactName 值在 OLD::和 NEW::之间翻转。图 6.16 显示了这一点。

一个 Zuul 路由过滤器的实现与前置过滤器或后置过滤器相比需要做更多的工作，但它也是 Zuul 最强大的一部分，因为你可以轻松地将智能添加到服务路由的方式中。

6.8. 小结

- Spring Cloud 使构建服务网关变得微不足道。
- Zuul 服务网关集成了 Netflix 的 Eureka 服务器，并且可以将在 Eureka 注册的服务自动映射到一个 Zuul 路由。
- Zuul 可以为受管理的所有路由添加前缀，所有你可以很容易为你的路由添加一些如 /api 的前缀。
- 使用 Zuul，你可以手动定义路由映射。这些路由映射是在应用程序配置文件中手动定义的。
- 通过使用 Spring Cloud 配置服务器，你可以动态的重新加载路由映射，而无需重新启动 Zuul 服务器。
- 你可以自定义 Zuul 的 Hystrix 和 Ribbon 超时时间为全局或单独的服务级别。
- Zuul 允许你通过 Zuul 过滤器实现自定义业务逻辑。Zuul 有三种类型的过滤器 :前置、后置和路由过滤器。
- Zuul 前置过滤器可用于生成一个关联 ID，它可以注入到流过 Zuul 的每一个服务。
- Zuul 后置过滤器可以为返回到服务客户端的每个 HTTP 服务响应注入一个关联 ID。

- 自定义 Zuul 路由过滤器可以执行基于 Eureka 服务 ID 的动态路由来为相同服务的不同版本之间做 A/B 测试。