5. 第 5 章 坏事发生时：使用 Spring Cloud 和 Netflix Hystrix 的客户端弹性模式

本章内容

- 实现断路器、回退和舱壁
- 使用断路器模式保护微服务客户端资源
- 当一个远程服务失败时使用 Hystrix
- 实现 Hystrix 的舱壁模式来隔离远程资源调用
- 调整 Hystrix 的断路器和舱壁的实现
- 自定义 Hystrix 的并发策略

所有系统，特别是分布式系统，都会经历失败。我们如何构建应用程序以应对失败，这是每个软件开发人员工作的一个关键部分。然而，当涉及到构建弹性系统时，大多数软件工程师只考虑到一个基础设施或关键服务的完全失败。他们着重于使用集群服务器、负载均衡、以及将基础设施分成多个位置等技术，将冗余构建到应用程序的每一层。

虽然这些方法考虑到系统组件的完整（通常是惊人的）损失，但它们只解决了构建弹性系统的一小部分。当一个服务崩溃时，很容易发现它已经不在了，应用程序可以绕过它。然而，当一个服务运行得很慢时，发现它的性能较差，并且绕过它是非常困难的，因为：

- *服务的恶化可以断断续续的开始并有扩大的势头*：恶化可能只在小范围内爆发。失败的第一个迹象可能是一小群用户抱怨一个问题，直到突然应用程序容器耗尽线程池并完全崩溃。
- *对远程服务的调用通常是同步的，并且不会缩短长时间调用*：服务的调用方没有超时的概念，以使服务调用永远中断。应用程序开发人员调用服务执行一个操作并等

待服务返回。

- *应用程序经常被设计来处理远程资源的完全失败，而不是局部恶化*：通常，只要服务没有完全失败，应用程序将继续调用服务，并且不会快速失败。应用程序将继续调用行为不良的服务。调用应用程序或服务可能会因资源耗尽而优雅降级，或者更可能崩溃。资源枯竭是一个有限的资源，如线程池和数据库连接超出最大连接数和调用客户端必须等待资源可用。

由于远程服务性能不佳所造成的问题是隐藏的，它们不仅难以检测，而且可能触发连锁反应，从而波及整个应用程序生态系统。如果没有保障措施，单个性能较差的服务可以很快地影响到多个应用程序。基于云，基于微服务的应用特别容易受到这些类型的冲击，因为这些应用程序由大量的细粒度分布式服务组成，其中包含了完成用户交易的不同基础设施。

5.1. 客户端的弹性模式

客户端的弹性软件模式都集中在当远程资源失败时从崩溃中保护远程资源的(另一个微服务调用或数据库查找) 客户端，因为远程服务抛出错误或表现不佳。这些模式的目标是允许客户端“快速失败”，不消耗宝贵的资源，如数据库连接和线程池，以及防止远程服务向客户端的消费者传播“上游”的问题。

有四种客户端弹性模式：

- 客户端负载均衡
- 断路器
- 回退
- 舱壁

① *The service client caches microservice endpoints retrieved during service*

discovery.

服务的客户端缓存微服务从服务发现取回的端点。

② The circuit breaker pattern ensures that a service client does not repeatedly call a failing service.

断路器模式确保服务客户端不重复调用失败的服务。

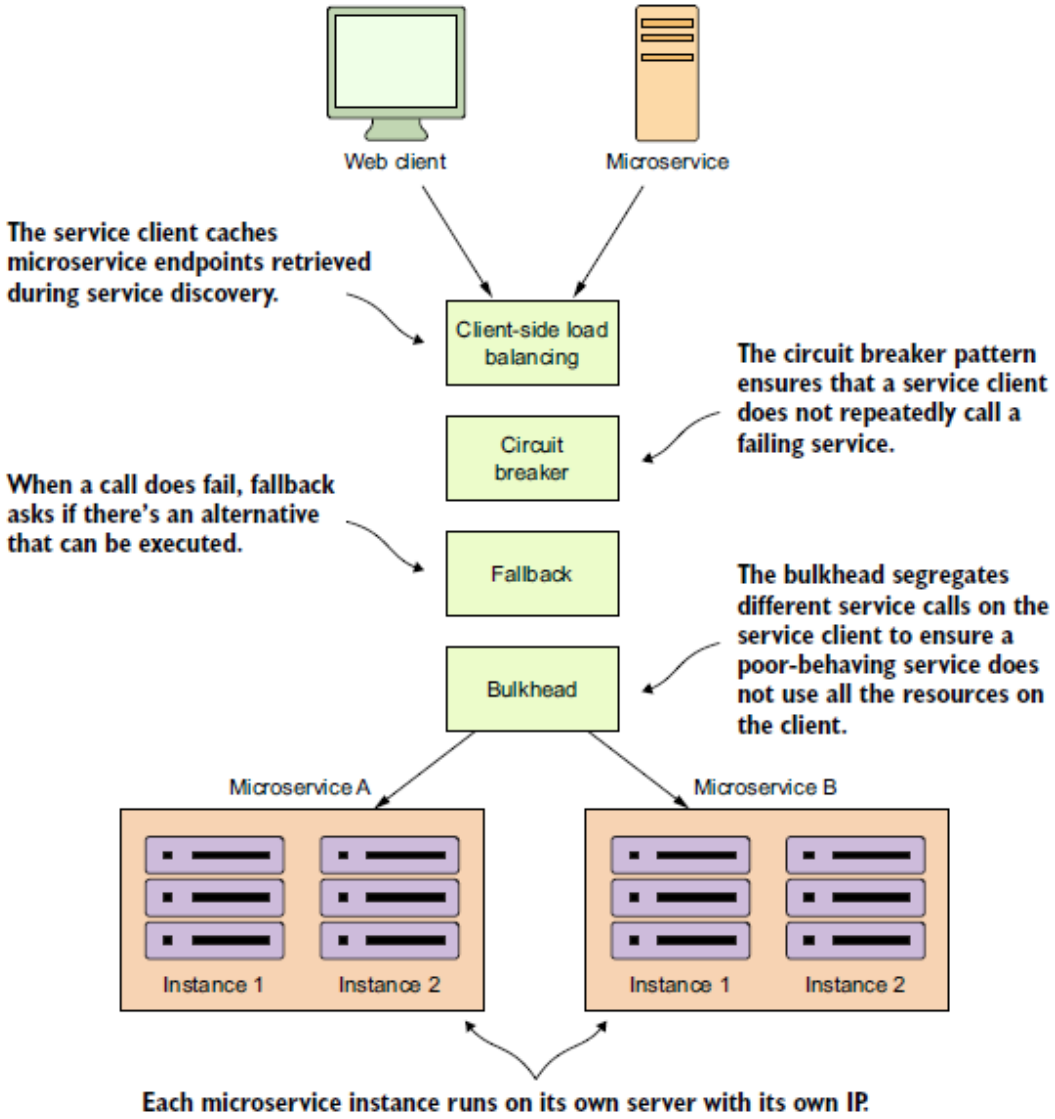

图 5.1 四个客户端弹性模式充当服务消费者和服务之间的保护缓冲区。

③ When a call does fail, fallback asks if there’s an alternative that can be executed.

当调用失败时，回退询问是否有可执行的替代方案。

④ The bulkhead segregates different service calls on the service client to ensure a poor-behaving service does not use all the resources on the client.

舱壁将服务客户端不同服务调隔离用来确保不良行为服务不使用客户端上的所有资源。

⑤ Each microservice instance runs on its own server with its own IP.

每个微服务实例使用它自己的 IP 运行在自己的服务器。

这些模式是在客户端中调用远程资源实现的。这些模式的实现逻辑位于客户端消费的远程资源和资源本身之间。

5.1.1. 客户端负载均衡

在最近一章（第 4 章）中，我们讨论服务发现时介绍了客户端负载均衡模式。客户端负载均衡包括让客户端从服务发现代理（如 Netflix Eureka）查找所有服务的单独实例，然后缓存所述服务实例的物理位置。每当服务消费者需要调用该服务实例时，客户端负载均衡器将从它维护的服务池中返回一个位置。

由于客户端负载均衡器位于服务客户端和服务消费者之间，因此负载均衡器可以检测服务实例是否抛出错误或表现不佳。如果客户端负载均衡器检测到问题，它可以从可用服务位置池中删除该服务实例，并防止后续的服务调用命中该服务实例。

这正是 Netflix 的 Ribbon 库在没有额外配置的情况下提供的行为。因为我们在第 4 章介绍了 Netflix Ribbon 的客户端负载均衡，所以我们不会在本章中更详细地讨论这个问题。

5.1.2. 断路器

断路器模式是在电路断路器之后模型化的客户端弹性模式。在电气系统中，断路器会检

测流过导线的电流是否过大。如果断路器检测到问题，它将中断与电气系统其余部分的连接，并使下游部件不被烧毁。

有了一个软件断路器，当一个远程服务被调用时，断路器将监听调用。如果调用需要花费很长时间，断路器将调解和终止调用。此外，断路器将监视对远程资源的所有调用，如果有足够多的调用失败，断路实现将启用，快速失败，并防止对失效的远程资源的后续调用。

5.1.3. 回退处理

在回退模式下，当远程服务调用失败时，服务消费者将执行另一个代码路径，并尝试通过另一种方式执行一个操作，而不是生成一个异常。这通常包括查找来自另一个数据源的数据或排队用户后续处理的请求。用户的调用不会显示说明问题的异常，但可能会通知他们的请求必须在以后完成。

例如，假设你有一个电子商务网站，监控用户的行为，并试着给他们一些他们可以购买的物品的建议。通常情况下，你可能会调用一个微服务来运行一个用户过去的行为分析和返回一个针对特定用户建议的列表。但是，如果首选项服务失败，你的回退可能是检索基于所有用户购买的更一般的首选项列表，并且更为通用。这些数据可能来自完全不同的服务和数据源。

5.1.4. 舱壁

舱壁模式是基于造船的概念的。关于舱壁的设计，船舶被分为完全隔离和密封的舱叫舱壁。即使船体被击穿，因为船被分隔为密封舱（舱壁），舱壁将保持水局限于击穿发生的地方和防止整个船装满了水而沉没。

同样的概念也可以应用于必须与多个远程资源交互的服务。通过使用舱壁模式，你可以

在自己的线程池中中断对远程资源的调用，并减少使用一个缓慢的远程资源调用会降低整个应用程序的风险。线程池为你服务的舱壁。每个远程资源被隔离并分配给线程池。如果一个服务响应缓慢，这种类型的服务调用的线程池将变得饱和，停止处理请求。对其他服务的服务调用不会被饱和，因为它们被分配给其他线程池。

5.2. 客户端弹性的重要性

我们已经在抽象中讨论了这些不同的模式；但是，让我们深入研究一个更具体的例子，说明这些模式可以应用于何处。让我们通过一个常见的场景，我遇到和看到为什么客户端弹性模式如断路器模式是实现一个基于服务的体系结构的关键，特别是在云上运行的微服务架构。

在图 5.2 中，我展示了一个典型的场景，它涉及到使用远程资源，比如数据库和远程服务。

在图 5.2 的场景中，三个应用程序以三种不同的服务以某种方式进行通信。应用 A 和 B 直接与服务 A 通信。服务 A 从数据库检索数据并调用服务 B 来为它工作。服务 B 从一个完全不同的数据库平台检索数据，并从第三方云提供商那里调用另一个服务 C，其服务严重依赖于内部网络区域存储（NAS）设备，将数据写入共享文件系统。此外，应用程序直接调用服务 C。

上周末，网络管理员对 NAS 上的配置做了一个小小的调整，如图 5.2 中的粗体所示。这种变化似乎工作得很好，但在星期一早上，任何对特定磁盘子系统的读取都开始非常缓慢地执行。

编写服务 B 的开发者从来没有预料到调用服务 C 会运行缓慢。他们编写代码，以便在同一事务中对数据库和服务的读写。当服务 C 开始运行缓慢，不仅要求服务 C 启动备份线

程池，在服务容器的连接池的数据库连接数枯竭，因为服务 C 的调用从未完成，这些连接被打开。

最后，服务 A 开始耗尽资源，因为它调用了服务 B，因为服务 C 运行得很慢。最终，所有三个应用程序都停止响应，因为它们在等待请求完成时耗尽了资源。

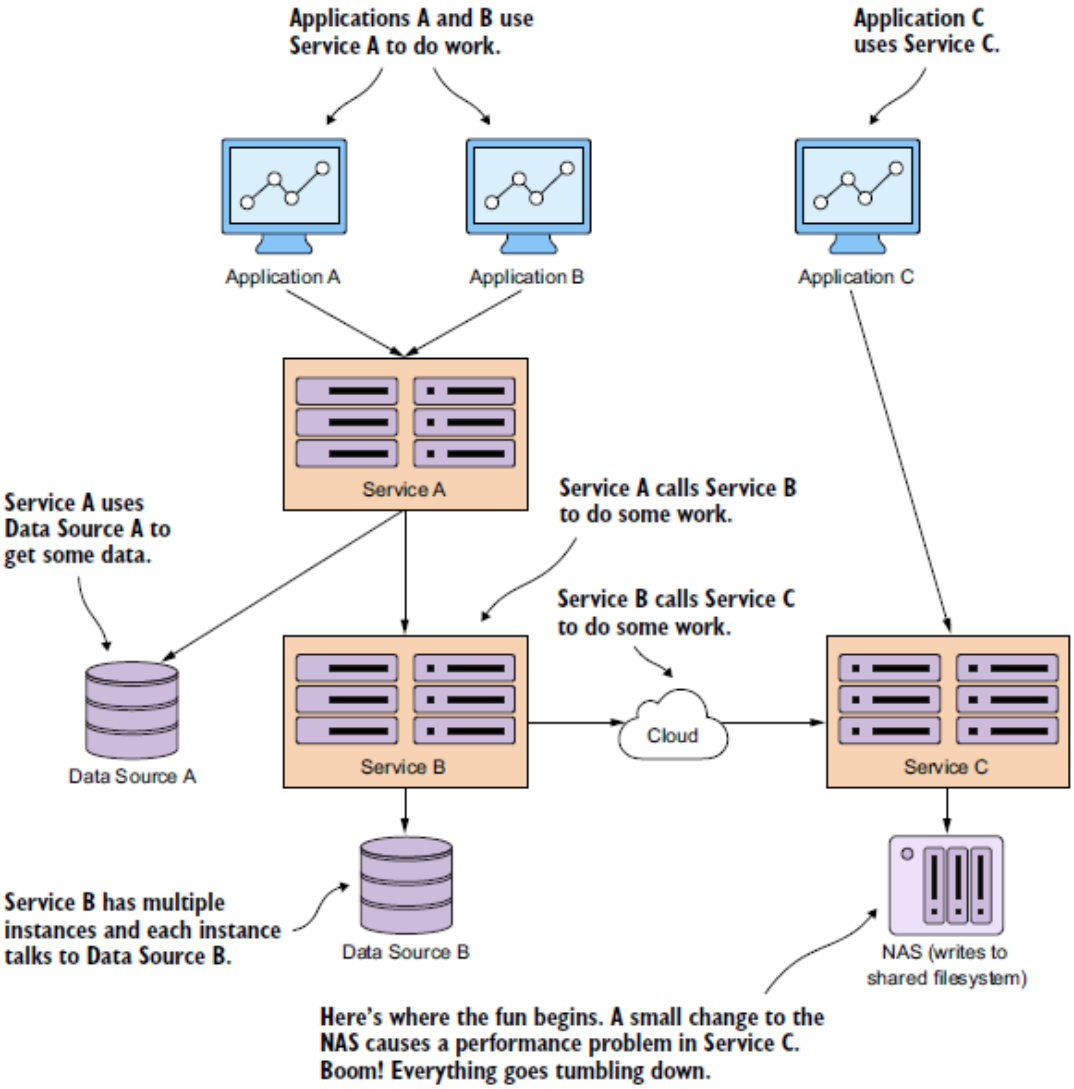

图 5.2 应用程序是一个相互依赖的关系图。如果不管理它们之间的远程调用，那么一个行为不好的远程资源就可以搞垮图中的所有服务。

① *Service A uses Data Source A to get some data.*

服务 A 使用的数据源 A 获得数据。

② *Service A calls Service B to do some work.*

服务 A 调用服务 B 来做一些工作。

③ Service B calls Service C to do some work.

服务 B 调用服务 C 来做一些工作。

④ Service B has multiple instances and each instance talks to Data Source B.

服务 B 有多个实例，每个实例都与数据源 B 交互。

⑤ Here’s where the fun begins. A small change to the NAS causes a performance problem in Service C.Boom! Everything goes tumbling down.

这是有趣的事开始的地方。对 NAS 的一个小更改会导致服务 C 中的性能问题！一切都在衰退。

如果在一个分布式资源被调用（无论是对数据库的调用还是对服务的调用）的每个点上实现了一个断路器模式，那么可以避免整个场景。在图 5.2 中，如果对服务 C 的调用是用一个断路器来实现的，那么当服务 C 开始表现不佳时，对服务 C 的特定调用的断路器将在不消耗线程的情况下被快速地中断和失败。如果服务 B 有多个端点，那么只有与服务 C 的特定调用交互的端点才会受到影响。服务 B 其余部分的功能仍然是完整的，能满足用户的要求。

断路器在应用程序和远程服务之间充当中间人。在前面的场景中，断路器实现可以保护应用程序 A、B 和 C 完全崩溃。

在图 5.3 中，服务 B（客户端）永远不会直接调用服务 C。相反，当调用时，服务 B 将服务的实际调用委托给断路器，它将调用并将其包装在一个线程中（通常由线程池管理），该线程独立于原始调用方。通过将调用包装在一个线程中，客户端不再直接等待调用完成。相反，断路器正在监视线程，如果线程运行时间过长，则可以终止调用。

图 5.3 显示了三个场景。在第一种情况下，happy 路径，断路器将维护一个计时器，

如果对远程服务的调用在计时器耗尽之前完成，一切都很好，服务 B 可以继续工作。在部分降级的方案，服务 B 将通过断路器调用服务 C。不过，这一次，服务 C 运行得很慢，如果断路器上的定时器超时，断路器将无法完成远程服务的连接。

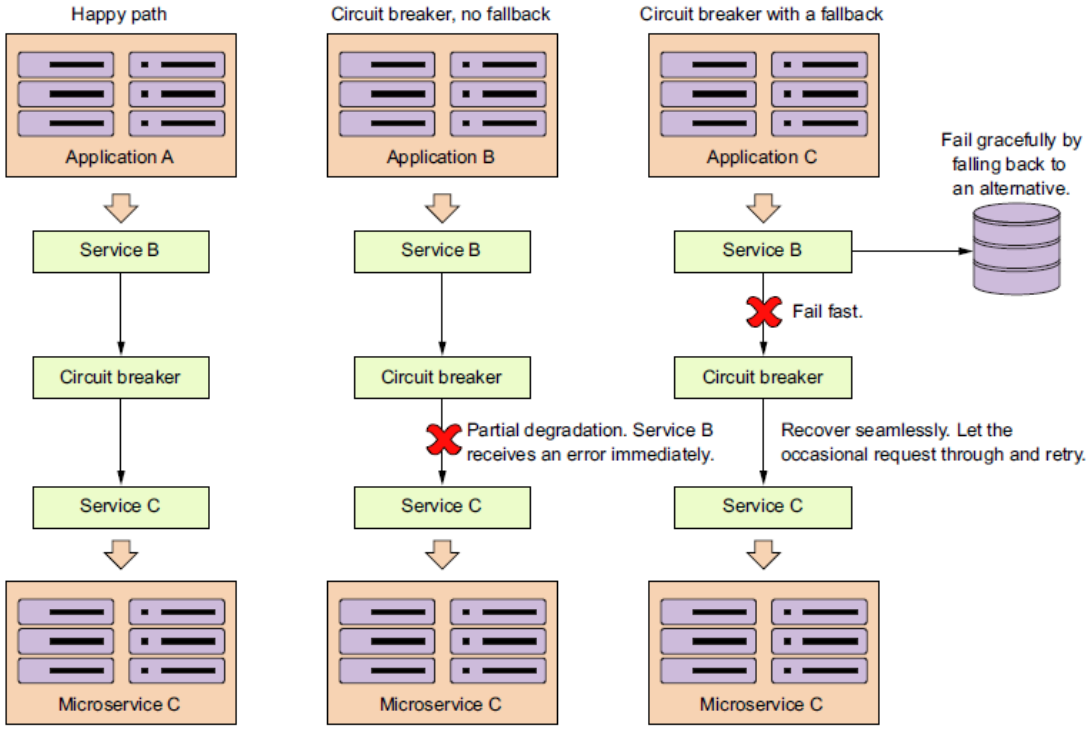

图 5.3 断路器跳闸和允许一个运行状况不良的服务调用快速和优雅地失败。

① *Partial degradation. Service B receives an error immediately.*

部分降级。服务 B 立即收到错误。

② *Recover seamlessly. Let the occasional request through and retry.*

无缝恢复。让偶尔的请求通过和重试。

③ *Fail gracefully by falling back to an alternative.*

通过回退到一个替代的方案实现优雅失败。

服务 B 将在调用时出错，但服务 B 将没有占用等待服务 C 完成的资源（也就是它自己的线程或连接池）。如果通过断路器对服务 C 的调用超时，断路器将开始跟踪发生故障的次数。

如果在某个时间段内发生了足够多的错误，那么断路器将“跳闸”，所有对服务 C 的调

用都将在不调用服务 C 的情况下失败。

断路器跳闸允许三件事情发生：

- 服务 B 现在立即知道有问题，而不必等待断路器超时。
- 服务 B 现在可以选择要么完全失败，要么使用另一组代码（回退）采取行动。
- 服务 C 将有机会恢复，因为当断路器已经跳闸的时候服务 B 没有调用它。这允许服务 C 有喘息空间，并帮助防止当服务降级发生时发生的级联崩溃。

最后，断路器偶尔会让调用通过降级的服务，如果这些调用成功连续次数足够，断路器将自动复位。

断路器模式关键是提供远程调用的能力：

- 快速失败：当远程服务正在经历降级时，应用程序将快速失败，并通常关闭整个应用程序阻止资源耗尽。在大多数停电情况下，最好是部分停机，而不是完全停机。
- 优雅的失败：通过超时和快速失败，断路器模式使应用程序开发人员能够优雅地失败，或者寻求替代机制来实现用户的意图。例如，如果用户试图从一个数据源检索数据，并且数据源正在经历服务降级，那么应用程序开发人员可以尝试从另一个位置检索该数据。
- 无缝恢复：断路器模式作为中介，断路器可以周期性地检查所请求的资源是否已恢复在线，并在没有人工干预的情况下重新启用所需的资源。

在一个有数以百计服务的基于大型云服务应用上，这个优雅的恢复是至关重要的因为它可以减少需要恢复服务的时间，显著减轻了操作员或应用工程师在服务的恢复时通过直接干预造成更大的风险问题（重新启动失败的服务）。

5.3. 熔断机制

建立断路器、回退和舱壁模式的实现需要对线程和线程管理有深入的了解。让我们面对现实吧，编写健壮的线程代码是一门艺术（我从来没有掌握过），正确地完成它是困难的。为了实现一套高质量的断路器、回退和隔板模式，需要大量的工作。幸运的是，你可以使用 Spring Cloud 和 Netflix 的 Hystrix 库来提供一个经过大量测试的库，它日常被应用在 Netflix 的微服务结构中。

在本章的后面几节中，我们将介绍如何：

- 配置许可服务的 Maven 构建文件（pom.xml）包括 Spring Cloud/Hystrix 依赖。
- 使用 Spring Cloud/Hystrix 注解和断路器模式包装远程调用。
- 在远程资源上自定义单独的断路器来使用每次调用的自定义超时时间。我还将演示如何配置断路器，以便控制在断路器“跳闸”之前发生多少次故障。
- 在断路器必须中断调用或调用失败时执行回退策略。
- 在你的服务中使用单个线程池隔离服务调用和建立不同的远程资源之间的舱壁。

5.4. 配置使用 Spring Cloud 和 Hystrix 的依赖

开始我们对 Hystrix 的研究 ,你需要设置你的项目的 pom.xml 文件导入 Spring Hystrix 依赖。你将获得你的许可服务，我们通过增加 Hystrix 的 Maven 依赖已经构建和修改它的 pom.xml 文件：

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
<dependency>
   <groupId>com.netflix.hystrix</groupId>
   <artifactId>hystrix-javanica</artifactId>
   <version>1.5.9</version>
```

```
</dependency>
```

第一个<dependency>标签（spring-cloud-starter-hystrix）告诉 Maven 拉取 Spring Cloud Hystrix 依赖。第二个<dependency>标签(hystrixjavanica)将拉取 Netflix Hystrix 核心库。设置 Maven 依赖，你就可以开始使用在前面章节构建的许可服务和组织服务的 Hystrix 实现。

> ***注意***：你不必直接在 pom.xml 文件包括 hystrix-javanica 依赖。默认情况下，spring-cloud-starter-hystrix 包括了一个版本的 hystrix-javanica 的依赖。本书中 Camden.SR5 版本使用 hystrix-javanica-1.5.6。hystrix-javanica 的版本有一个不一致的引入，导致 Hystrix 代码没有回退，抛出 java.lang.reflect.UndeclaredThrowableException 异常代替 com.netflix.hystrix.exception.HystrixRuntimeException 异常。对于许多使用旧版本的 Hystrix 的开发者来说，这是一个突然的变化。hystrix-javanica 库固定在以后的版本中，所以我特意使用一个较新的 hystrix-javanica 版本替换 Spring Cloud 内使用的默认版本。

最后一件事是，你能在你的应用程序代码里使用 Hystrix 断路器之前，使用 @EnableCircuitBreaker 注解注释你的服务的引导类。例如，对于许可服务，你将在 licensing-service/src/main/java/com/thoughtmechanix/licenses/Application.java 类中增加@EnableCircuitBreaker 注解。下面的清单显示了此代码。

清单 5.1 @EnableCircuitBreaker 注解被用于在服务里激活 Hystrix

```
package com.thoughtmechanix.licenses

import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;

@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker  <-------  ①告诉 Spring Cloud 你将为你的服务使用 Hystrix
public class Application {
    @LoadBalanced
    @Bean
```

```
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

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

注意：如果你忘记在引导类添加@EnableCircuitBreaker 注解，没有断路器将会被激活。当服务启动时，你将不会收到任何警告或错误消息。

5.5. 使用 Hystrix 实现断路器

我们来看看两大类型的 Hystrix 实现。在第一种类型中，你将使用 Hystrix 断路器包装许可服务和组织服务对数据库所有的调用。然后你在许可服务和组织服务之间使用 Hystrix 包装 inter-service 调用。虽然这是两个不同的类型调用，你会看到，Hystrix 的使用将完全相同。图 5.4 显示的是你将用一个 Hystrix 断路器包装远程资源。

① Implementing a circuit breaker using Hystrix

使用 Hystrix 实现一个断路器

② First category: All calls to database wrapped with Hystrix

第一类：用 Hystrix 包装所有对数据库的调用

③ Second category: Inter-service calls wrapped with Hystrix

第二类：使用 Hystrix 包装 Inter-service 的调用

让我们从通过展示如何使用一个同步的 Hystrix 断路器包装从许可数据库检索许可服务数据开始我们的 Hystrix 讨论。通过同步调用，许可服务将检索其数据，但在继续处理之前，将等待 SQL 语句完成或断路器超时。

Hystrix 和 Spring Cloud 使用@HystrixCommand 注解来标记 Java 类方法，通过一

个 Hystrix 断路器来管理。当 Spring 框架看到@HystrixCommand，它会动态生成一个代理，将包装方法和管理所有的调用，该方法通过专门预留处理远程调用线程的线程池。

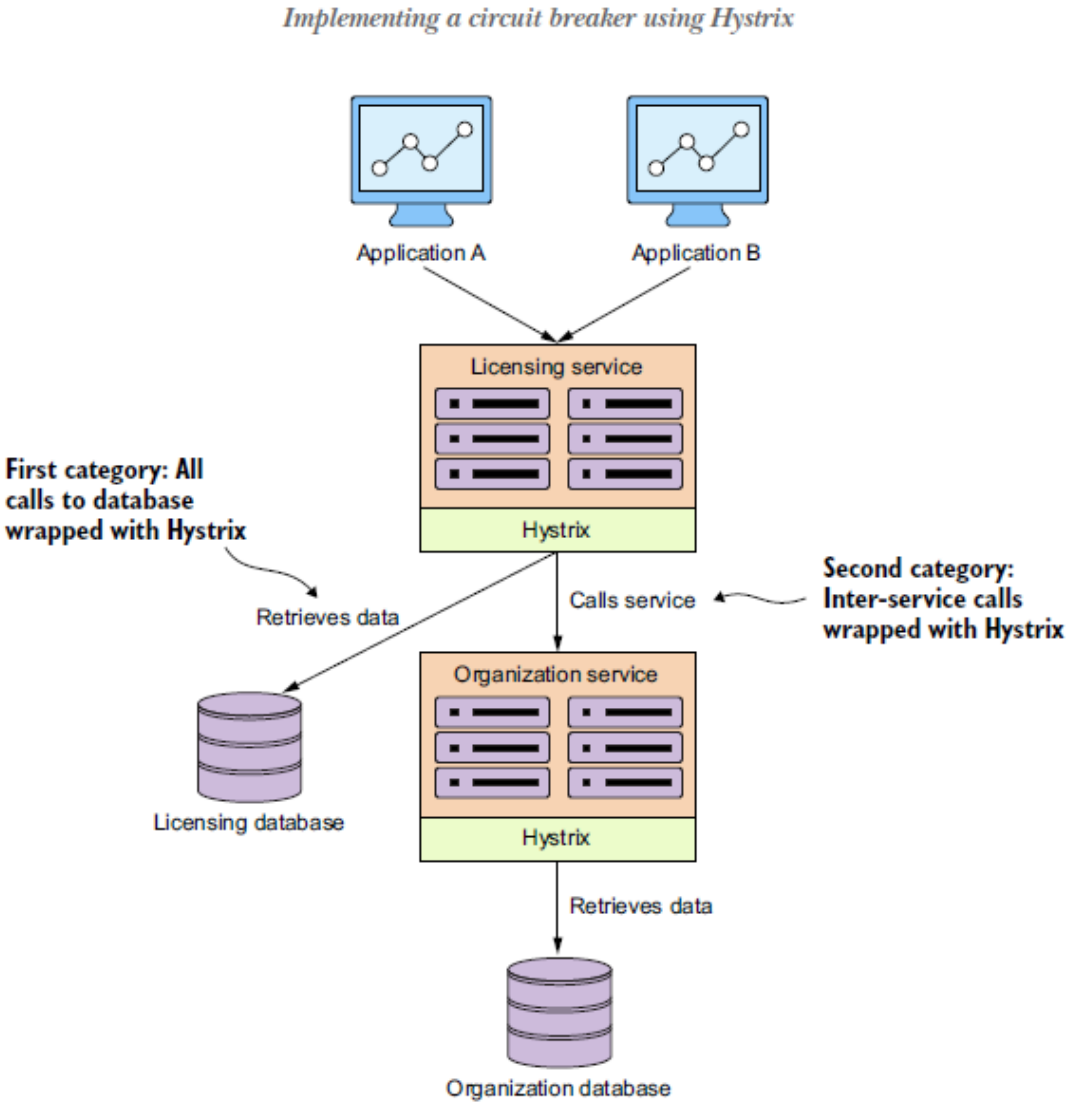

图 5.4 Hystrix 位于每个远程资源的调用之间，并保护客户端。远程资源调用是数据库调用还是基于 REST 的服务调用，这并不重要。

你将在 licensingservice/src/main/java/com/thoughtmechanix/licenses/services/LicenseService.java 类包装 getLicensesByOrg()方法，如下面的清单所示。

清单 5.2 使用一个断路器包装一个远程资源调用

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

①@HystrixCommand注解被用于使用 Hystrix 断路器包装 getLicenseByOrg()方法

```
}
```

> ***注意***：如果你看一下再源代码库中清单 5.2 的代码，你会看到比先前清单中显示的@HystrixCommand 注解多了几个参数。我们将在本章后面研究这些参数。清单 5.2 中的代码使用@HystrixCommand 注解，并且所有属性值均使用默认值。

这看起来不像是很多代码，但是这个注释里面有很多功能。使用@HystrixCommand 注解，任何时候 getLicensesByOrg()方法被调用，调用将被使用一个 Hystrix 断路器包装。断路器将中断任何对 getLicensesByOrg()方法任何时候调用时间长于 1000 毫秒的调用。

如果数据库正常工作，这个代码示例会很乏味。让我们模拟 getLicensesByOrg()方法通过调用运行一个缓慢的数据库查询，大约每三个调用消耗一秒钟多一点。下面的清单演示了这一点。

清单 5.3 随机超时调用许可服务数据库

```
private void randomlyRunLong(){
   Random rand = new Random();

   int randomNum = rand.nextInt((3 - 1) + 1) + 1;

   if (randomNum==3) sleep();
}

private void sleep(){
   try {
     Thread.sleep(11000);
   } catch (InterruptedException e) {
     e.printStackTrace();
   }
}

@HystrixCommand
public List<License> getLicensesByOrg(String organizationId){
   randomlyRunLong();

   return licenseRepository.findByOrganizationId(organizationId);
}
```

① randomlyRunLong() 方法给你三分之一的机会数据库调用长期运行。

②你休眠了 11000 毫秒（11 秒）。Hystrix 默认行为是 1 秒后调用。

如果你点击 http://localhost/v1/organizations/e254f8c-c442-4ebea82a-

e2fc1d1ff78a/licenses/端点足够多的次数，你应该会看到从许可服务返回的超时错误消息。图 5.5 显示了这个错误。

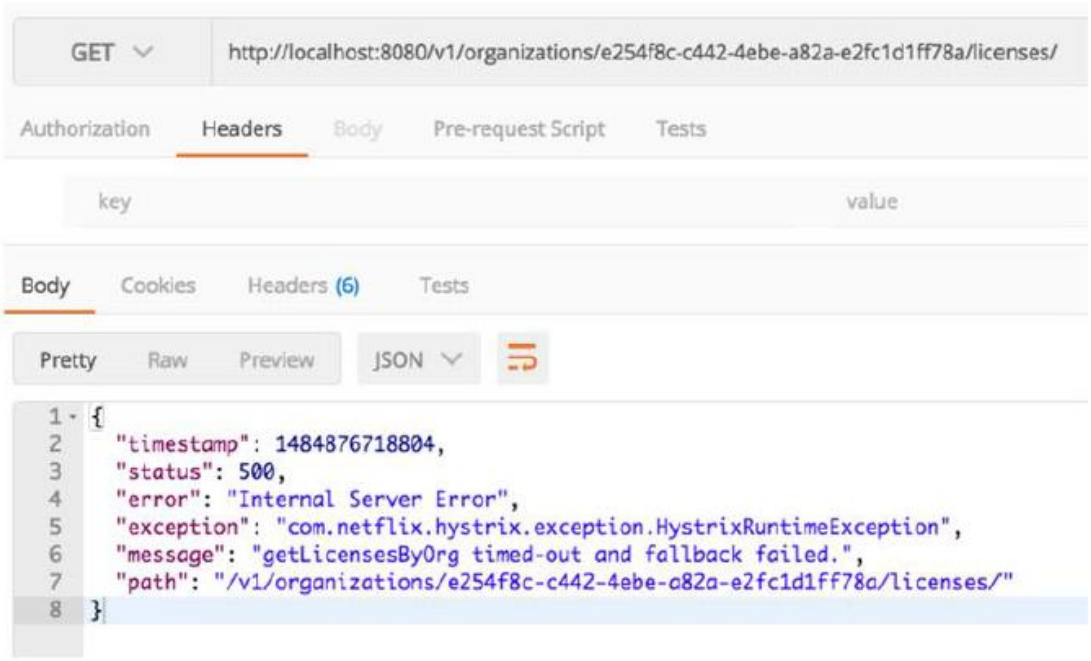

图 5.5 当远程调用消耗时间太长时，HystrixRuntimeException 异常被抛出。

现在，在@HystrixCommand 注解的地方，如果查询时间过长，许可服务将中断对数据库的调用。如果数据库调用需要花费超过 1000 毫秒将执行 Hystrix 代码包装，你的服务调用将抛出一个 com.nextflix.hystrix.exception.HystrixRuntimeException 异常。

5.5.1. 调用微服务超时

具有断路器行为的方法级注解标签调用的优点就是无论你访问一个数据库还是调用一个微服务都用相同的注解。

例如，在你的许可服务中，你需要查找与许可证相关联的组织的名称。如果你希望用断路器将你的调用包装到组织服务，这很简单，就是把 RestTemplate 调用分解成自己的方法和使用@HystrixCommand 注解注释它。

```
@HystrixCommand
private Organization getOrganization(String organizationId) {
```

```
    return organizationRestClient.getOrganization(organizationId);
}
```

注意：而使用@HystrixCommand 是容易实现的，你确实需要小心使用在注解里没有配置的默认@HystrixCommand 注解。默认情况下，当你指定一个没有属性的@HystrixCommand 注解时，注解会将所有远程服务调用放置在同一个线程池下。这会在应用程序中引入问题。在本章后面我们将讨论如何实现舱壁模式，我们将向您展示如何将这些远程服务调用隔离到自己的线程池中，并配置线程池的行为彼此独立。

5.5.2. 自定义断路器上的超时时间

当我与新的开发人员一起工作时，经常遇到的第一个问题是在 Hystrix 中断调用之前，它是如何自定义时间。通过向@HystrixCommand 注解中传递额外的参数，很容易做到这一点。下面的清单演示了在调用超时前，如何自定义 Hystrix 等待的时间。

清单 5.4 自定义断路器调用超时时间

```
@HystrixCommand(
    commandProperties=
        {@HystrixProperty(
     name="execution.isolation.thread.timeoutInMilliseconds",
     value="12000")})
public List<License> getLicensesByOrg(String organizationId){
    randomlyRunLong();
    return licenseRepository.findByOrganizationId(organizationId);
}
```

①commandProperties 属性让你提供额外的属性来自定义 Hystrix。

①execution.isolation.thread.timeoutInMilliseconds 用于设置断路器的超时时间长度（以毫秒为单位）。

Hystrix 允许你通过 commandProperties 属性自定义的断路器的行为。commandProperties 属性接受一个 HystrixProperty 对象数组，它可以通过自定义属性来配置 Hystrix 断路器。在清单 5.4 中，你使用 execution.isolation.thread.timeoutInMilliseconds 属性设置最大超时时间为在 Hystrix 调用失败前 12 秒。

现在，如果你重建并重新运行代码示例，你将永远不会得到超时错误，因为你调用时的

人工超时为 11 秒，而你的@HystrixCommand 注解现在被配置为仅在 12 秒后才超时

服务超时

很明显，我用断路器超时 12 秒作为一个教学例子。在分布式环境中，如果我开始听取开发团队的意见，我常常会感到紧张，因为远程服务调用的 1 秒超时太低，因为它们的服务 X 平均需要 5-6 秒。

这通常告诉我，被调用的服务存在未解决的性能问题。避免在 Hystrix 调用中增加默认超时，除非你完全不能解决缓慢运行的服务调用。

如果你确实存在这样的情况，你的服务调用中的一部分将消耗比其他服务调用更长的时间，一定要考虑将这些服务调用隔离到单独的线程池。

5.6. 回退处理

断路器模式的一个优点是，一个“中间人”位于远程资源的消费者和资源本身之间，有机会让开发人员拦截服务失败并选择一种替代的操作方案。

在 Hystrix，这是一个被称为回退策略和易于实现。让我们看看如何为你的许可数据库构建一个简单的回退策略，该策略只简单返回一个许可对象，该对象表示当前没有可用的许可信息。下面的清单演示了这一点。

清单 5.5 在 Hystrix 实现回退

```
@HystrixCommand(fallbackMethod = "buildFallbackLicenseList")
public List<License> getLicensesByOrg(String organizationId){
    randomlyRunLong();

    return licenseRepository.findByOrganizationId(organizationId);
}

private List<License> buildFallbackLicenseList(String organizationId){
    List<License> fallbackList = new ArrayList<>();
    License license = new License()
      .withId("0000000-00-00000")
      .withOrganizationId( organizationId )
```

①fallbackMethod 属性在你的类中定义一个单一的功能，如果 Hystrix 失败该类将被调用。

②在回退方法中，返回硬编码值。

```
    .withProductName(
    "Sorry no licensing information currently available");

    fallbackList.add(license);
    return fallbackList;
}
```

注意：在从 GitHub 库获取的源代码中，我注释掉 fallbackMethod，所以你可以看到服务调用随机失败。看看在清单 5.5 中实战的回退代码，你需要取消 fallbackMethod 属性的注释。否则，你将永远看不到实际调用的回退。

使用 Hystrix 实现一个回退策略你必须做两件事。首先，你需要给@HystrixCommand 注解添加一个叫做 fallbackMethod 的属性。此属性将包含一个方法的名称，该方法将在 Hystrix 必须中断调用时调用，因为它占用的时间太长。

你需要做的第二件事是定义要执行的回退方法。此回退方法必须在同一个类，被@HystrixCommand 保护的原始方法。回退方法必须与原始方法具有相同的方法参数定义，所有参数传递到由@HystrixCommand 保护的原始方法将传递给回退。

在清单 5.5 中的例子，buildFallbackLicenseList()回退方法就是简单构建一个单一的许可对象包含模拟的信息。你可以让回退方法从其他数据源中读取这些数据，但为了演示目的，你将构建一个列表，该列表将由原始方法调用返回。

回退

回退策略工作非常好的情况下，你的微服务正在检索数据且调用失败。在我工作的一个组织中，我们将客户信息存储在操作数据存储（ODS）中，并在数据仓库中进行汇总。

我们高兴的是总是能检索最新的数据，并根据它计算摘要信息。然而，一个特别讨厌的停电，一个缓慢的数据库连接了多个服务后，我们决定使用 Hystrix 回退实现保护检索和汇总客户信息的服务调用。如果由于性能问题或错误而对 ODS 的调用失败，我们使用回退从数据仓库表中检索汇总数据。

我们的业务团队决定，给客户旧的数据比让客户看到错误或整个应用程序崩溃要好得多。当选择是否使用回退策略时，关键是你的客户对他们的数据生命周期的容忍程度，以及从不让他们看到应用程序有问题的重要性。

在决定是否要实现回退策略时，需要记住以下几点：

- 回退是一种机制，当资源超时或失败时提供一种处理的方式。如果你发现自己使用回退捕获一个超时异常并且仅仅是记录错误日志，那么你应该在你的服务调用中使用一个标准的 try.. catch 块，捕获 HystrixRuntimeException 异常，然后在 try..catch 块中添加日志记录逻辑。
- 注意你的回退功能所采取的操作。如果在回退服务中调用另一个分布式服务，可能需要用@HystrixCommand 注解包装回退。记住，同样的失败也会影响你的次要选择。代码的防御。当使用回退时，我已经体会到我没有考虑到这一点。

现在你的回退方法已经就位，你可以继续再次调用你的端点。这一次当你点击它并遇到超时错误时（记住你有 3 的机会），你不应该从服务调用中获得异常，而是返回模拟许可证值。

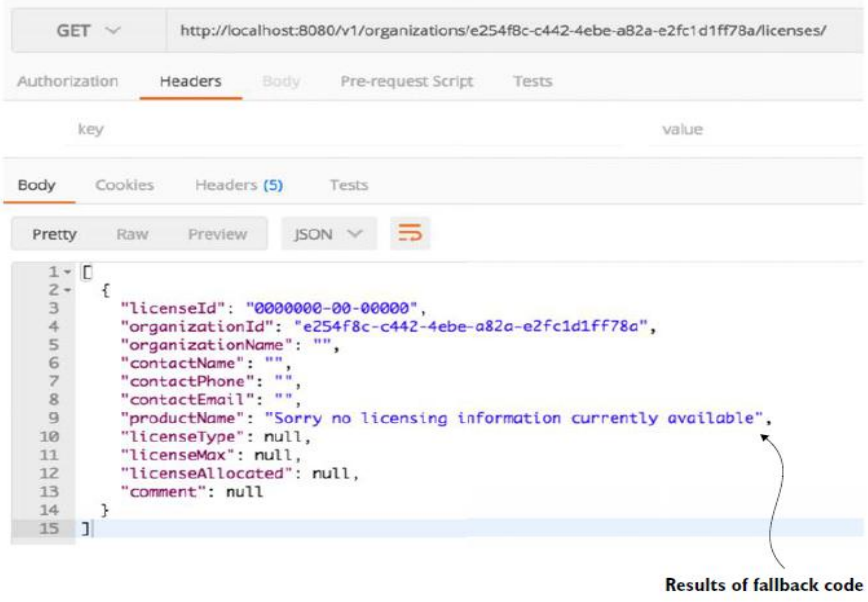

图 5.6 你的服务调用使用 Hystrix 回退

5.7. 舱壁模式的实现

在微服务应用中，你为了完成特定任务需要调用多个微服务。没有使用舱壁模式，这些调用的默认行为是使用预留处理整个 Java 容器请求的同一线程执行。大量地，一个服务的性能问题可以导致 Java 容器所有的线程被刷爆并等待处理 ,在此期间新的工作请求被堵塞。Java 容器最终会崩溃。舱壁模式将在自己的线程池中隔离远程资源调用，一个性能不好的服务可以包含在容器内而不会导致容器崩溃。

Hystrix 使用一个线程池代理远程服务请求。默认情况下，所有 Hystrix 命令将共享相同的线程池来处理请求。这个线程池将有 10 个线程用于处理远程服务调用，这些远程服务调用可以是任何东西，包括其 REST 服务调用，数据库调用，等等。图 5.7 说明了这一点。

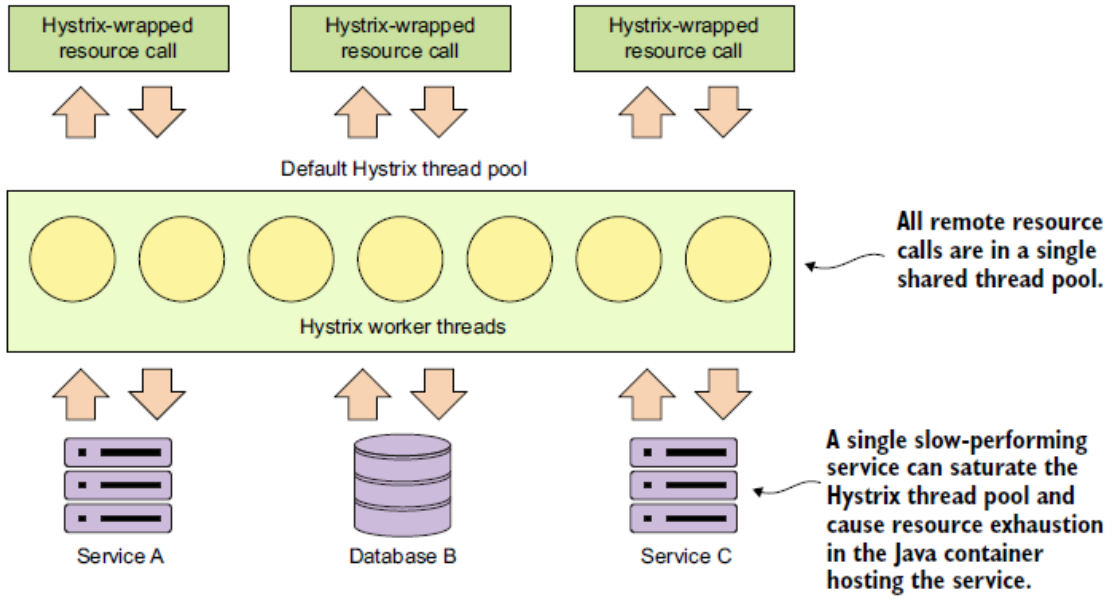

图 5.7 跨多个资源类型共享的默认 Hystrix 线程池

① *Hystrix-wrapped resource call*

包装 Hystrix 的资源调用

② *Default Hystrix thread pool*

默认的 Hystrix 线程池

③ *Hystrix worker threads*

Hystrix 工作线程

④ *All remote resource calls are in a single shared thread pool.*

所有远程资源调用都在一个共享线程池中。

⑤ *A single slow-performing service can saturate the Hystrix thread pool and cause resource exhaustion in the Java container hosting the service.*

一个单一的执行速度较慢的服务可以使 Hystrix 线程池饱和，并导致托管服务的 Java 容器资源枯竭。

当你在应用程序中访问了少量的远程资源，并且每个服务的调用量相对均匀分布时，此模型可以较好的工作。问题是如果你有更高的量或完成时间较长之后的其他服务，你可以最终引入线程耗尽你的 Hystrix 线程池，因为服务结束了对默认线程池中所有线程的控制。

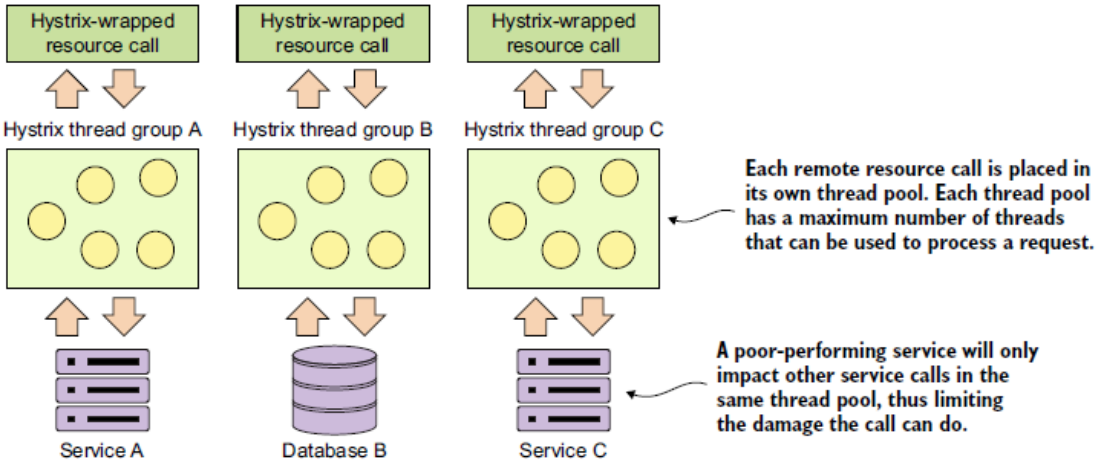

图 5.8 绑定到隔离线程池的 Hystrix 命令

① *Each remote resource call is placed in its own thread pool. Each thread pool has a maximum number of threads that can be used to process a request.*

每个远程资源调用都放在自己的线程池中。每个线程池的最大线程数可以用来处理请求。

② *A poor-performing service will only impact other service calls in the same thread*

pool, thus limiting the damage the call can do.

性能较差的服务只会影响同一线程池中的其他服务调用，从而限制调用所能造成的损害。

幸运的是，Hystrix 为创建舱壁提供了用于不同的远程资源之间的一种易于使用的机制。图 5.8 显示了 Hystrix 所管理的资源看起来像他们隔离自己的“隔壁”。

为了实现隔离线程池，你需要使用通过@HystrixCommand 注解公开的附加属性。让我们看看一些代码：

- 为 getLicensesByOrg()调用创建一个隔离的线程池
- 设置线程池中的线程数
- 如果单个线程繁忙，可以为队列请求的请求数量设置队列大小

下面的清单演示了如何为调用我们许可服务的许可数据的所有调用设置一个舱壁。

清单 5.6 环绕 getLicensesByOrg()方法创建一个舱壁

```
@HystrixCommand(fallbackMethod = "buildFallbackLicenseList",
                threadPoolKey = "licenseByOrgThreadPool",
                threadPoolProperties =
                        {@HystrixProperty(name = "coreSize",value="30"),
                        HystrixProperty(name="maxQueueSize", value="10")}
)
public List<License> getLicensesByOrg(String organizationId){
   return licenseRepository.findByOrganizationId(organizationId);
}
```

①threadPoolKey 属性定义了线程池唯一的名称。

② threadPoolProperties 属性允许你定义和定制的线程池的行为。

③coreSize 属性可以让你定义线程池中线程的最大数量。

④maxQueueSize 让你在线程池前面定义一个队列，它能使进入的请求排队。

你应该注意的第一件事是，我们在@HystrixCommand 注解引入了一个新的属性，threadPoolkey。这个 Hystrix 属性表明，你想建立一个新的线程池。如果你没有进一步的对线程池设置值，Hystrix 在 threadPoolKey 属性中设置线程池的主键名称，但将使用所有的默认值作为线程池的配置。

你在@HystrixCommand 注解中使用 threadPoolProperties 属性，自定义你的线程池。这个属性接受一系列 HystrixProperty 对象。这些 HystrixProperty 对象可以用来控制线程池的行为。你可以用 coreSize 属性设置线程池的大小。

你还可以在线程池前面设置一个队列，该线程控制线程池中的线程忙时会允许多少请求堵塞。这个队列的大小是由 maxQueueSize 属性设置。一旦请求数量超过队列大小，任何向线程池额外的请求都会失败，直到队列中有空间为止。

注意 maxQueueSize 属性的两件事情。首先，如果你设置为-1，一个 Java 同步队列用来存储所有传入的请求。同步队列本质上限制你在进程中不能有更多的请求，即线程池中可用的线程数。设置 maxQueueSize 值大于一个会导致 Hystrix 使用 Java 的 LinkedBlockingQueue 的值。一个 LinkedBlockingQueue 的使用允许开发者排队请求，即使所有的线程都忙着处理请求。

第二点需要注意的是，maxQueueSize 属性只能在线程池首次初始化时（例如，在应用程序启动）被设置。Hystrix 会允许你用 queueSizeRejectionThreshold 属性动态改变队列的大小，但是这个属性只能设置在 maxQueueSize 属性，且其值一个大于 0。

自定义线程池的正确大小是多少？Netflix 推荐以下公式：

（当服务正常时每秒的峰值请求 * 99%平均响应时间）+ 少量额外线程用于开销

你通常不知道服务的性能特性，直到它处于负载状态。线程池属性需要调整的一个关键指标是当服务调用超时时，即使目标远程资源是健康的。

5.8. 深入理解 Hystrix

在这一点上，我们已经看了使用 Hystrix 建立断路器和舱壁模式的基本概念。我们现在要去看看如何真正定制 Hystrix 断路器的行为。记住，Hystrix 不仅仅是时间长调用。Hystrix 还将监控调用失败的次数，如果有足够多次数的调用失败，Hystrix 将自动防止后续的请求在请求远程资源之前调用失败。

有两个原因。首先，如果远程资源存在性能问题，那么快速失败将阻止调用应用程序必

须等待调用超时。这大大降低了调用应用程序或服务将经历自身资源耗尽问题和崩溃的风险。第二，快速失败和阻止来自服务客户端的调用将帮助陷入困境的服务跟上其负载，而不是在负载下完全崩溃。快速失败给出系统经历性能下降的恢复时间。

了解如何在 Hystrix 配置断路器，你需要先了解 Hystrix 如何决定断路器什么时候跳闸的流程。图 5.9 显示了当远程资源调用失败时，使用 Hystrix 的决策过程。

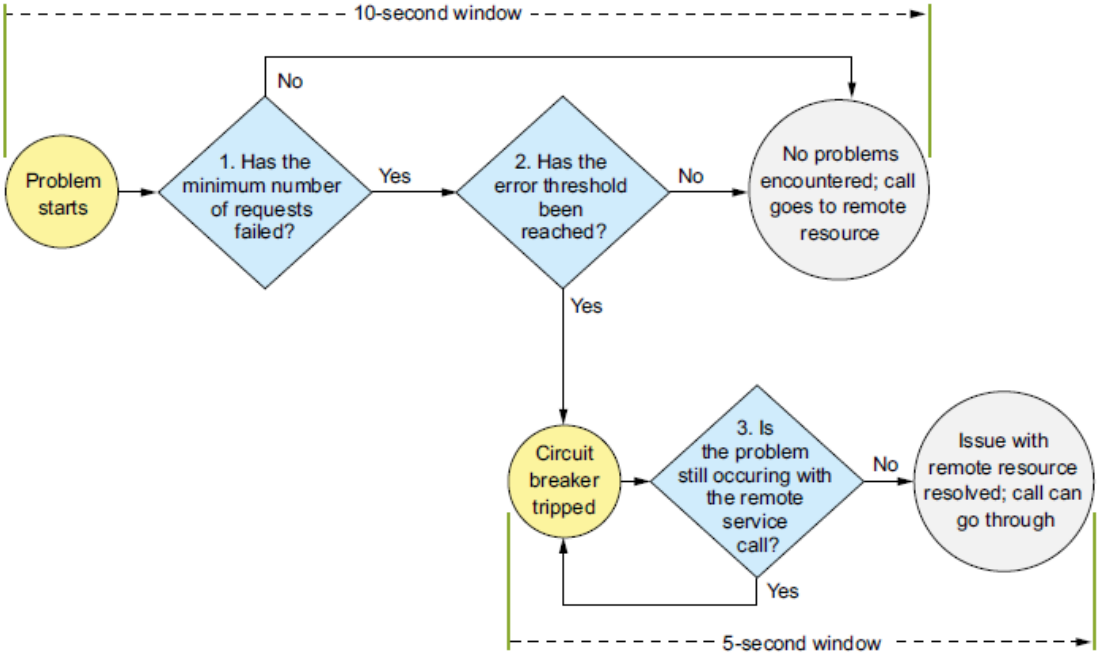

图 5.9 Hystrix 经过一系列的检查以确定是否断路器跳闸。

① *10-second window*：10 秒的窗口

5-second window：5 秒的窗口

② *Problem starts*

问题开始

③ *Has the minimum number of requests failed?*

请求失败次数达到最小数量了吗？

④ *Has the error threshold been reached?*

达到错误阈值了吗？

⑤ *No problems encountered; call goes to remote resource*

没有遇到问题；调用到远程资源

⑥ *Circuit breaker tripped*

断路器跳闸

⑦ *Is the problem still occuring with the remote service call?*

远程服务调用是否仍然存在问题？

⑧ *Issue with remote resource resolved; call can go through*

提供远程资源解析；调用可以通过

每当 Hystrix 命令遇到一个服务错误，它将启动一个 10 秒的定时器，计时器将被用来检查服务调用通常是如何失败的。这个 10 秒的窗口是可配置的。第一件事是 Hystrix 看在 10 秒的窗口内已经发生的调用次数。如果调用次数小于调用在窗口内需要发生的最小次数，然后 Hystrix 将不会采取行动，即使有些调用失败。例如，在 10 秒的窗口 Hystrix 将考虑行动之前，需要发生的调用次数缺省值是 20。如果在 10 秒的时间内调用失败有 15 次，没有足够的调用发生来使断路器跳闸，即使所有 15 次调用都失败。Hystrix 将继续让调用到达远程服务。

当远程资源调用最小次数在 10 秒的窗口内发生，Hystrix 将开始查看整体故障发生率。如果失败的总百分比超过阈值，Hystrix 会触发断路器和后续的调用绝大部分均失败。我们不久将讨论，Hystrix 将让部分调用通过来测试，看看服务是否恢复。错误阈值的默认值是 50%。

当在远程调用上，Hystrix 断路器已经跳闸时，它将尝试为活动启动新的窗口。每五秒（这个值是可配置的），Hystrix 会让调用到达挣扎中的服务。如果调用成功，Hystrix 将重置断路器并开始重新让调用通过。如果调用失败，Hystrix 将断路器闭合，在 5 秒之后再尝

试。

在此基础上，你可以看到有五个属性可以用来定制断路器的行为。@HystrixCommand 注解通过 commandPoolProperties 属性公开 5 个属性。然而 threadPoolProperties 属性允许你设置用于 Hystrix 命令基础线程池的行为，commandPoolProperties 属性允许你自定义与 Hystrix 命令相关的断路器行为。下面的清单显示属性的名称以及如何在每个属性中设置值。

清单 5.7 配置断路器的行为

```
@HystrixCommand(
    fallbackMethod = "buildFallbackLicenseList",
    threadPoolKey = "licenseByOrgThreadPool",
    threadPoolProperties ={
            @HystrixProperty(name = "coreSize",value="30"),
        @HystrixProperty(name="maxQueueSize"value="10"),
    },
    commandPoolProperties ={
      @HystrixProperty(
        name="circuitBreaker.requestVolumeThreshold",
        ➥ value="10"),
        ➥ @HystrixProperty(
        name="circuitBreaker.errorThresholdPercentage",
        ➥ value="75"),

            @HystrixProperty(
              name="circuitBreaker.sleepWindowInMilliseconds",
              value="7000"),
              ➥ @HystrixProperty(
                  name="metrics.rollingStats.timeInMilliseconds",
                  ➥ value="15000")
              @HystrixProperty(
                name="metrics.rollingStats.numBuckets",
                ➥ value="5")}
        )
public List<License> getLicensesByOrg(String organizationId){
➥ logger.debug("getLicensesByOrg Correlation id: {}",
➥ UserContextHolder
                .getContext()
                .getCorrelationId());
  randomlyRunLong();
```

```
    return licenseRepository.findByOrganizationId(organizationId);
}
```

第一个属性，circuitBreaker.requestVolumeTheshold，控制在 Hystrix 将考虑调用断路器跳闸之前，在一个 10 秒的窗口内必须出现的连续调用数量。第二个属性，circuitBreaker.errorThresholdPercentage，是 circuitBreaker.requestVolumeThreshold 值在断路器跳闸之前 circuitBreaker.requestVolumeThreshold 值已经通过之后，调用必须失败的百分比（由于超时，将抛出一个异常，或返回 HTTP 500）。在前面的代码示例中，最后一个属性 circuitBreaker.sleepWindowInMilliseconds，是一旦断路器跳闸，Hystrix 允许另外的调用通过来看服务是否恢复健康之前 Hystrix 将休眠的时间。

Hystrix 最后的两个属性 (metrics.rollingStats.timeInMilliseconds 和 metrics.rollingStats.numBuckets) 的命名与前面的属性有些不同，但它们仍然控制断路器的行为。第一个属性，metrics.rollingStats.timeInMilliseconds，用于控制窗口的大小，它将被 Hystrix 用于监控服务调用的问题。此值的默认值是 10000 毫秒（即 10 秒）。

第二个属性，metrics.rollingStats.numBuckets，控制你在窗口中定义并被收集的统计次数。Hystrix 收集在窗口中桶的度量并检查这些桶的统计数据来确定远程资源调用失败。桶数量的定义必须均匀分为以毫秒为单位的总体数量，用 rollingStatus.inMilliseconds 来设置统计。例如，在前面清单中你的自定义设置，Hystrix 将使用 15 秒窗口并将统计数据收集到五个长度为三秒的桶中。

注意：你进入的统计窗口越小，窗口中保存的桶数越大，在高容量服务上就可以提高 CPU 和内存利用率。要意识到这一点，并避免设置度量收集窗口和桶的细粒度，直到是你需要的可见级别。

5.8.1. 进一步理解 Hystrix 配置

Hystrix 库是可配置的，并让你可以紧紧的控制断路器的行为和你定义的舱壁模式。通过修改一个 Hystrix 断路器的配置，你可以在远程调用超时之前，控制 Hystrix 将等待的时间。当 Hystrix 断路器将跳闸和当 Hystrix 尝试重置断路器时，你也可以控制其行为。

用 Hystrix，你也可以通过定义每个远程服务调用单个线程组然后配置每一个线程组关联的线程数来微调你的舱壁实现。这允许你对远程服务调用进行微调，因为在其它远程资源调用将具有更高的量期间，某些调用比其它调用将具有更高的量。

关键是记住，当你看配置 Hystrix 环境时，在 Hystrix 你有三种配置级别：

- 整个应用程序的默认值
- 类默认值
- 类中定义的线程池级别

每个 Hystrix 属性通过默认值设置，将被应用程序里的每个@HystrixCommand 注解使用，除非他们设置为 Java 类级别或者在一个类中单个 Hystrix 线程池重写。

Hystrix 让你在类级别设置默认参数 以便在一个特定的类共享相同配置的所有 Hystrix 命令。类级属性通过一个叫做@DefaultProperties 的类级别注解进行设置。例如，如果你想让所有的资源在一个特定的类有一个 10 秒的超时时间，你可以按以下方式设置@DefaultProperties：

```
@DefaultProperties(
    commandProperties = {
     @HystrixProperty(
     ➥ name = "execution.isolation.thread.timeoutInMilliseconds",
     value = "10000")}
class MyService { ... }
```

除非在线程池级别显式重写 ,否则所有线程池将继承应用程序级别上的默认属性或类中定义的默认属性。Hystrix 的 threadPoolProperties 和 commandproperties 属性也绑上

定义的命令键。

> ***注意***：对编码的例子，我在应用程序代码硬编码了所有 Hystrix 值。在生产系统中，Hystrix 数据最可能需要修改（超时参数，线程池计数）为 Spring Cloud 配置。这样，如果你需要改变参数值，你可以改变值，然后重新启动服务实例，而无需重新编译和重新部署应用程序。

对于单个 Hystrix 池，我将尽可能保持代码的配置，并将线程池配置放在@HystrixCommand 注解中。表 5.1 总结了用于设置和配置@HystrixCommand 注解的所有配置值。

表 5.1 @HystrixCommand 注解配置值

属性名称	默认值	描述
fallbackMethod	None	标识如果远程调用超时调用的类里面的方法。回调方法必须在与@HystrixCommand 注解相同的类中，并且必须具有与调用类相同的方法参数。如果没有值，将通过 Hystrix 将抛出一个异常。
threadPoolKey	None	给@HystrixCommand 唯一的名称和创建一个线程池，它是独立的默认线程池。如果没有定义值，默认 Hystrix 线程池将被使用。
threadPoolProperties	None	用于配置线程池行为的核心 Hystrix 注解属性。
coreSize	10	设置线程池的大小。
maxQueueSize	-1	将在线程池前面设置的最大队列大小。如果设置为-1，没有采用队列而 Hystrix 将阻塞直到一个线程变为可用的处理。
circuitBreaker.request-	20	必须在滚动窗口内 Hystrix 将开始检查断路器是

VolumeThreshold		否会跳闸之前设置请求的最小数量。 注意：这个值只能用 commandPoolProperties 属性设置。
circuitBreaker.error-ThresholdPercentage	50	在断路器跳闸之前必须在滚动窗口内发生的故障百分比。 注意：这个值只能用 commandPoolProperties 属性设置。
circuitBreaker.sleep-WindowInMilliseconds	5,000	在服务调用前，断路器已跳闸之后 Hystrix 将等待的毫秒数。 注意：这个值只能用 commandPoolProperties 属性设置。
metricsRollingStats.timeInMilliseconds	10,000	Hystrix 在窗口中将收集和监控统计服务调用的毫秒数。
metricsRollingStats.numBuckets	10	Hystrix 将在它的监控窗口内维护的度量同数量。监视窗口中的桶越多，时间越低，Hystrix 将会监控窗口内的错误。

5.9. 线程上下文和 Hystrix

当一个@HystrixCommand 执行时，它可以采用两种不同的隔离策略 线程和信号量。默认情况下，Hystrix 运行在一个线程隔离。每个 Hystrix 命令用来保护运行在一个独立的线程池的调用，不与进行调用的父线程共享它的上下文。这意味着 Hystrix 可以中断控制的一个线程的执行，而不必担心中断任何与父线程做原始调用相关联的其他活动。

信号隔离，Hystrix 管理的分布式调用受@HystrixCommand 注解保护，如果调用超时没有开始一个新的线程和中断父线程。在同步容器服务器环境（Tomcat）中，中断父线程将导致无法由开发人员捕获的异常抛出。这可能会导致开发人员编写代码的意外后果，因为他们无法捕获抛出的异常或进行任何资源清理或错误处理。

设置一个命令池来控制隔离，你可以在 @HystrixCommand 注解设置 commandProperties 属性。例如，如果你想在 Hystrix 命令使用信号隔离设置隔离级别，你会用

```
@HystrixCommand(
commandProperties = {
  @HystrixProperty(
        name="execution.isolation.strategy", value="SEMAPHORE")})
```

注意：默认情况下，Hystrix 团队建议你为大部分命令使用默认的线程隔离策略。这将在你和父线程之间保持更高级别的隔离级别。线程隔离比使用信号量隔离要重。信号隔离模型是更轻量级的，当在你的服务中你有高容量时应该被使用，且运行在一个异步 I/O 编程模型（你使用异步 I/O 容器如 Netty）。

5.9.1. ThreadLocal 和 Hystrix

Hystrix，默认情况下，不会将父线程的上下文传播到由 Hystrix 命令管理的线程。例如，任何值设置为在父线程 ThreadLocal 的值对通过父线程调用的方法默认不可用，且通过@HystrixCommand 对象保护。（再次，这是假设你使用线程隔离级别）。

这可能有点不太清晰，所以让我们来看一个具体的例子。通常，在基于 REST 的环境中，你希望将上下文信息传递到服务调用，这将帮助你在操作上管理服务。例如，你可以在 REST 调用的 HTTP 头中传递关联 ID 或认证令牌，然后将其传播到任何下游服务调用。关联 ID 允许你拥有一个惟一标识符，该标识符可以跟踪单个交易中的多个服务调用。

为了使这个值在服务调用的任何地方都可用，你可以使用一个 Spring 过滤器类拦截 REST 服务的每一次调用，从传入的 HTTP 请求检索此信息，并在一个自定义的用户上下文对象存储上下文信息。然后，在你的代码需要在 REST 服务调用访问这个值，你的代码可以从 ThreadLocal 存储变量中检索用户上下文和读取该值。下面的清单显示了一个 Spring 过滤器示例，你可以在许可服务中使用。你可以在 licensingservice/src/main/java/com/thoughtmechanix/licenses/utils/UserContextFilter.java 找到代码。

清单 5.8 UserContextFilter 解析 HTTP 头和检索数据

```
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) );

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

①在 UserContext 中检索调用的 HTTP 头中设置的值，它被存储在 UserContextHolder

```
                .getContext()
                .setOrgId(
                    httpServletRequest.getHeader(UserContext.ORG_ID));
            filterChain.doFilter(httpServletRequest, servletResponse);
        }
}
```

UserContextHolder 类被用于将 UserContext 存储在 ThreadLocal 类中。一旦它被存储在 ThreadLocal 存储，请求任何代码的执行，将使用存储在 UserContextHolder 的 UserContext 对象。UserContextHolder 类显示在以下清单中。这类被发现在 licensing-service/src/main/java/com/thoughtmechanix/licenses/utils/UserContextHolder.java。

清单 5.9 所有的 UserContext 数据由 UserContextHolder 管理

```
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();
      }
}
```

①UserContext 存储在静态 ThreadLocal 变量中。

②getContext()方法将为消费检索 UserContext 对象。

此时，你可以向你的许可服务添加两条日志语句。你将向以下的许可服务类和方法添加日志记录：

- com/thoughtmechanix/licenses/utils/UserContextFilter.java

  doFilter()方法

- com/thoughtmechanix/licenses/controllers/LicenseServiceController.java

  getLicenses()方法

- com/thoughtmechanix/licenses/services/LicenseService.java

  getLicensesByOrg()方法。这个方法使用@HystrixCommand 注解。

接下来，你将调用你的服务，使用一个名为 tmx-correlation-id 的 HTTP 头传递一个关联 ID，其值为 TEST-CORRELATION-ID。图 5.10 显示了一个在 Postman 的 HTTP GET 调用 http://localhost:8080/v1/organizations/e254f8c-c442-4ebe-a82ae2fc1d1ff78a/licenses/。

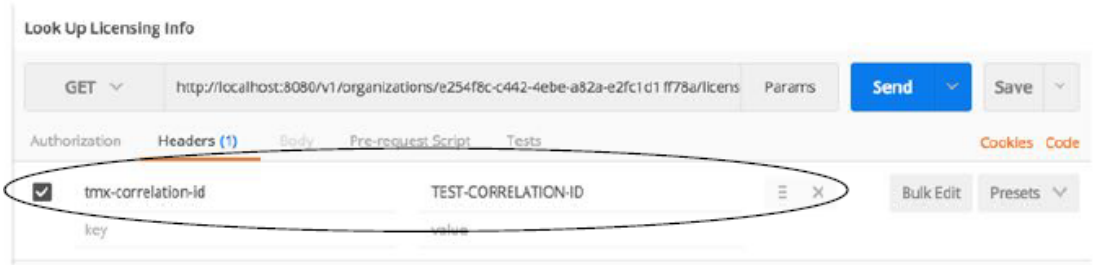

图 5.10 将关联 ID 添加到许可服务调用的 HTTP 头

一旦调用被提交，你应该看到当请求流经 UserContext，LicenseServiceController，LicenseServer 类时，通过关联 ID 输出三个日志信息：

```
UserContext Correlation id: TEST-CORRELATION-ID
LicenseServiceController Correlation id: TEST-CORRELATION-ID
LicenseService.getLicenseByOrg Correlation:
```

正如预期的那样，一旦调用命中 Hystrix 在 LicenseService.getLicensesByOrder()上保护的方法，你将不会获得关于关联 ID 的输出值。幸运的是，Hystrix 和 Spring Cloud 提供了一种机制，将父线程的上下文传播到由 Hystrix 线程池管理的线程。这种机制被称为 HystrixConcurrencyStrategy。

5.9.2. Hystrix 并发策略

Hystrix 允许你定义一个自定义并发策略，该策略将包装你的 Hystrix 调用，并允许你将任何附加的父线程上下文注入到由 Hystrix 命令管理的线程中。要实现一个自定义的 HystrixConcurrencyStrategy 你需要进行三个步骤：

- 定义你的自定义 Hystrix 并发策略类
- 定义一个 Java 调用类将 UserContext 注入到 Hystrix 命令
- 配置 Spring Cloud 使用自定义的 Hystrix 并发策略

所有的 HystrixConcurrencyStrategy 示例可以在 licensingservice/src/main/java/com/thoughtmechanix/licenses/hystrix 包中发现。

定义你的自定义 Hystrix 并发策略类

你需要做的第一件事就是定义你的 HystrixConcurrencyStrategy。默认情况下，Hystrix 只允许你为应用程序定义一个 HystrixConcurrencyStrategy。Spring Cloud 已经定义了用于处理扩展传播的 Spring 安全信息的并发策略。幸运的是，Spring Cloud 允许你将 Hystrix 并发策略链接在一起，这样你就可以定义并使用自己的并发策略，将其“插入”到 Hystrix 并发策略中。

我们 Hystrix 并发策略的实现可以在许可服务的 hystrix 包中发现，叫做 ThreadLocalAwareStrategy.java。下面的清单显示了该类的代码。

清单 5.10 定义你自己的 Hystrix 并发策略

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

public class ThreadLocalAwareStrategy extends HystrixConcurrencyStrategy{
    private HystrixConcurrencyStrategy existingConcurrencyStrategy;

    public ThreadLocalAwareStrategy(
                HystrixConcurrencyStrategy existingConcurrencyStrategy) {
        this.existingConcurrencyStrategy = existingConcurrencyStrategy;
    }
```

① 扩展自基础的 HystrixConcurrencyStrategy 类。

①Spring Cloud 已经定义了一个并发类。将现有的并发策略传递到你的 HystrixConcurrencyStrategy 的类构造方法

```
    @Override
    public BlockingQueue<Runnable> getBlockingQueue(int maxQueueSize){
          return existingConcurrencyStrategy != null
                  ? existingConcurrencyStrategy.getBlockingQueue(maxQueueSize)
                  : super.getBlockingQueue(maxQueueSize);
    }

    @Override
    public <T> HystrixRequestVariable<T> getRequestVariable(
            HystrixRequestVariableLifecycle<T> rv)
    {//Code removed for conciseness }

    @Override
    public ThreadPoolExecutor getThreadPool(
    ➥ HystrixThreadPoolKey threadPoolKey,
      HystrixProperty<Integer> corePoolSize,
      HystrixProperty<Integer> maximumPoolSize,
      HystrixProperty<Integer> keepAliveTime,
      TimeUnit unit,
      BlockingQueue<Runnable> workQueue)
    {//code removed for conciness}

    @Override
    public <T> Callable<T> wrapCallable(Callable<T> callable) {
    return existingConcurrencyStrategy != null
      ? existingConcurrencyStrategy.wrapCallable(
      ➥new DelegatingUserContextCallable<T>(
              callable,
            ➥UserContextHolder.getContext()))
      : super.wrapCallable(
          new DelegatingUserContextCallable<T>(
              callable,
              UserContextHolder.getContext()));
     }
}
```

②需要重写一些方法。无论是调用的 existingConcurrencyStrategy 方法实现或基于 HystrixConcurrencyStrategy 的调用。

③注入设置在 UserContext 的回调实现。

注意清单 5.10 中类实现中的一些事情。首先，因为 Spring Cloud 已经定义了一个 HystrixConcurrencyStrategy，可以覆盖的每种方法都需要检查是否存在现有的并发策略，然后调用现有的并发策略方法或基于 Hystrix 并发策略的方法。

你要做这一约定确保你调用已经存在的处理安全的 Spring Cloud 的 HystrixConcurrencyStrategy。否则，当试图在你的 Hystrix 保护代码中使用 Spring

security 上下文时，你会有令人令人不愉快的行为。

要注意的第二点是清单 5.11 中的 wrapCallable()方法。在这种方法中，你通过回调实现，DelegatingUserContextCallable，它将用于设置从父线程执行用户的 REST 服务调用的用户上下文到 Hystrix 命令线程的保护方法，该方法是在内部进行的工作。

定义一个 Java 回调类来将 UserContext 注入到 Hystrix 命令

将父线程的线程上下文传播到你的 Hystrix 命令的下一个步骤是实现将执行传播的回调类。在这个例子中，这调用在 hystrix 包，称为 delegatingusercontextcallable.java。下面的清单显示了该类的代码。

清单 5.11 使用 DelegatingUserContextCallable.java 传播 UserContext

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

public final class DelegatingUserContextCallable<V>
      implements Callable<V> {
   private final Callable<V> delegate;
   private UserContext originalUserContext;

   public DelegatingUserContextCallable(
       Callable<V> delegate,
                UserContext userContext) {
     this.delegate = delegate;
     this.originalUserContext = userContext;
   }

   public V call() throws Exception {
       UserContextHolder.setContext( originalUserContext );

       try {
           return delegate.call();
       }
       finally {
           this.originalUserContext = null;
       }
    }

    public static <V> Callable<V> create(Callable<V> delegate,
                                         UserContext userContext) {
```

①自定义回调类将通过原始的回调类传递，该类将调用你的 Hystrix 受保护的代码，UserContext 来自父线程。

②通过@HystrixCommand 注解保护方法之前 call()方法被调用。

③设置 UserContext。ThreadLocal 变量存储 UserContext，它与正在运行线程的 Hystrix 保护方法关联。

④在 Hystrix 保护方法上，一旦 UserContext 被设置调用 call()方法；例如，你的 LicenseServer.getLicenseByOrg()方法。

```
            return new DelegatingUserContextCallable<V>(delegate, userContext);
        }
}
```

当调用了一个受 Hystrix 保护的方法，Hystrix 和 Spring Cloud 将实例化 DelegatingUserContextCallable 类的一个实例，通过通常由 Hystrix 命令池管理的线程调用的回调类。在前面的清单，回调类存储在称为 delegate 的一个 Java 属性。从概念上讲，你可以将 delegate 属性看作是由@HystrixCommand 注解保护的方法的句柄。

除了委托回调的类，Spring Cloud 也使 UserContext 对象与初始化调用的父线程隔离。在创建 DelegatingUserContextCallable 实例时，这两个值被设置，实际操作会发生在你的类的 call()方法。

要做的第一件事情是通过 UserContextHolder.setContext()方法在 call()方法里设置 UserContext。记住，setContext()方法在正在运行的特定线程的一个 ThreadLocal 变量里存储一个 UserContext 对象。一旦 UserContext 被设置，然后你调用委托回调类的 call()方法。这个对 delegate.call()的调用调用了由@HystrixCommand 注解保护的方法。

配置 Spring Cloud 使用自定义的 Hystrix 并发策略

现在你已经有通过 ThreadLocalAwareStrategy 类和通过 DelegatingUserContextCallable 类定义了回调类的 HystrixConcurrencyStrategy，你需要将它们挂在 Spring Cloud 和 Hystrix 中。为此，你将定义一个新的配置类。这种配置，称为 ThreadLocalConfiguration，如下清单所示。

清单 5.12 将自定义的 HystrixConcurrencyStrategy 类挂入 Spring Cloud

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

@Configuration
public class ThreadLocalConfiguration {
   @Autowired(required = false)
   private HystrixConcurrencyStrategy existingConcurrencyStrategy;

   @PostConstruct
```

①当配置对象被构造时，它将在已存在的 HystrixConcurrencyStrategy 中自动装配。

```
  public void init() {
    // Keeps references of existing Hystrix plugins.
    HystrixEventNotifier eventNotifier =
    ➥ HystrixPlugins
          .getInstance()
          .getEventNotifier();
      HystrixMetricsPublisher metricsPublisher =
      ➥HystrixPlugins
          .getInstance()
          .getMetricsPublisher();
      HystrixPropertiesStrategy propertiesStrategy =
      ➥HystrixPlugins
          .getInstance()
          .getPropertiesStrategy();
      HystrixCommandExecutionHook commandExecutionHook =
      ➥HystrixPlugins
          .getInstance()
          .getCommandExecutionHook();
      HystrixPlugins.reset();

      HystrixPlugins.getInstance()
        .registerConcurrencyStrategy(
         new ThreadLocalAwareStrategy(existingConcurrencyStrategy));
      HystrixPlugins.getInstance()
        .registerEventNotifier(eventNotifier);
      HystrixPlugins.getInstance()
        .registerMetricsPublisher(metricsPublisher);
      HystrixPlugins.getInstance()
        .registerPropertiesStrategy(propertiesStrategy);
      HystrixPlugins.getInstance()
        .registerCommandExecutionHook(commandExecutionHook);
  }
}
```

②因为你注册一个新的并发策略，你要抢占所有其它的 Hystrix 组件，然后重置 Hystrix 插件。

③你用 Hystrix 插件注册你的 HystrixConcurrencyStrategy(ThreadLocalAwareStrategy)。

④然后使用 Hystrix 插件重新注册所有的 Hystrix 组件。

这个 Spring 配置类基本上重建了 Hystrix 插件，用于管理在你的服务中运行的所有不同组件。在 init() 方法，你通过插件抓住所有已用的 Hystrix 组件引用。然后你注册你的自定义 HystrixConcurrencyStrategy (ThreadLocalAwareStrategy)。

```
HystrixPlugins.getInstance().registerConcurrencyStrategy(
  new ThreadLocalAwareStrategy(existingConcurrencyStrategy));
```

记住，Hystrix 只允许一个 HystrixConcurrencyStrategy。Spring 将尝试自动装配任何现有的 HystrixConcurrencyStrategy (如果它存在的话)。最后，当你做完这一切，你重

新注册原始的 Hystrix 组件，你使用 Hystrix 插件在 init()方法开始后抓住这些组件。

通过这些片段，你现在可以重建并重新启动许可服务，并通过图 5.10 中所示的 GET(http://localhost:8080/v1/organizations/e254f8c-c442-4ebea82a-e2fc1d1ff78a/licenses/)调用它。现在，当这个调用完成时，你应该在控制台窗口中看到以下输出：

```
UserContext Correlation id: TEST-CORRELATION-ID
LicenseServiceController Correlation id: TEST-CORRELATION-ID
LicenseService.getLicenseByOrg Correlation: TEST-CORRELATION-ID
```

产生一个小的结果需要大量的工作，但不幸的是，当你使用 Hystrix 的线程级隔离时，这是必须的。

5.10. 小结

- 当设计高度分布式的应用如微服务应用，必须考虑客户端的弹性。
- 服务的彻底失败（例如，服务器崩溃）很容易检测和处理。
- 单个性能较差的服务可能会触发资源耗尽的级联效应 ,因为调用客户端中的线程被阻塞，等待服务完成。
- 三种核心客户端弹性模式是断路器模式、回退模式和舱壁模式。
- 断路器模式试图杀死运行缓慢和退化的系统调用，调用快速失败，防止资源枯竭。
- 回退模式允许开发人员在远程服务调用失败或调用的断路器失败时定义替代代码路径。
- 主体头模式将远程资源调用彼此远离，将远程服务的调用隔离在自己的线程池。如果一组服务调用失败，不允许它的失败吞噬应用程序容器中的所有资源。
- Spring Cloud 和 Netflix Hystrix 库提供了断路器、回退和舱壁模式的实现。
- Hystrix 库是高度可配置的，可以在全局、类和线程池级别设置。
- Hystrix 支持两种隔离模式：线程和信号量。
- Hystrix 的默认隔离模式线程，完全隔离了一个受 Hystrix 保护的调用，但不会将父线

程的上下文传播到受 Hystrix 管理的线程。

- Hystrix 的另一个隔离模式，信号量，不使用单独的线程进行一个 Hystrix 调用。虽然这是更有效的，这也暴露出如果 Hystrix 中断调用，服务不可预知的行为。
- Hystrix 允许通过自定义 HystrixConcurrencyStrategy 实现，将父线程上下文注入到一个受 Hystrix 管理的线程中。