异步图书
www.epubit.com.cn

深入理解 Spring Cloud 与微服务构建

方志朋◎著

由浅入深，全面讲解 Spring Cloud 基础组件。

一站式了解用 Spring Cloud 构建微服务，**实战案例，快速上手**。

详解 Spring Security OAuth2，为微服务系统的**安全保驾护航**。

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

深入理解 Spring Cloud 与微服务构建

方志朋◎著

人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

深入理解Spring Cloud与微服务构建 / 方志朋著
. -- 北京 : 人民邮电出版社, 2018.3
ISBN 978-7-115-47522-0

Ⅰ. ①深… Ⅱ. ①方… Ⅲ. ①互联网络－网络服务器
Ⅳ. ①TP368.5

中国版本图书馆CIP数据核字(2017)第315454号

内 容 提 要

本书共分 16 章，全面涵盖了 Spring Cloud 构建微服务相关的知识点。第 1、2 章详细介绍了微服务架构和 Spring Cloud。第 3、4 章讲解了用 Spring Cloud 构建微服务的准备工作。第 5～12 章以案例为切入点，讲解了 Spring Cloud 构建微服务的基础组件，包括 Eureka、Ribbon、Feign、Hystrix、Zuul、Config、Sleuth、Admint 等组件。第 13～15 章讲述了使用 Spring Cloud OAuth2 来保护微服务系统的相关知识。第 16 章用一个综合案例，全面讲解了如何使用 Spring Cloud 构建微服务，可以作为实际开发的样例工程。

本书既适合 Spring Cloud 初学者入门使用，又适合正在做微服务实践的架构师或打算实施微服务的团队作为参考用书，同时也可作为高等院校计算机相关专业的师生用书和培训学校的教材。

◆ 著　　方志朋
责任编辑　张　涛
执行编辑　张　爽
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京市艺辉印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：17.5
字数：412 千字　　2018 年 3 月第 1 版
印数：1 – 3 500 册　　2018 年 3 月北京第 1 次印刷

定价：69.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

序　一

一直以来，系统的架构设计是 IT 领域经久不衰的话题，也是构建每一个系统最核心且重要的部分之一。它决定了系统能否满足业务、技术、组织、灵活、可扩展性等多种要求，同时肩负起了解放程序员生产力的作用。

2016 年底，由于业务的不断发展，我所在公司维护的项目也越来越“臃肿”。随着无数个版本的迭代，以及开发人员的不断增加，开发效率越来越低，每次投产的人力成本和时间成本都逐渐增加，我们一直在思索如何能破局。评估了各种方案后，最终微服务进入了我们的视野。

谈到微服务，大家众说纷纭，但却很难有一个清晰的概念来描述。微服务不是“银弹”，我理解的微服务是一种文化，而我们要做的就是将微服务的理念运用到实际开发中。经过一系列的技术选型，最终 Spring Cloud 凭借其成熟的组件、完善的一站式解决方案，最终成为了我们落地微服务的选择。

此时的 Spring Cloud 相关资料在国内还是凤毛麟角，没有完整的中文书籍和教程可以参考，只有官方的英文文档以及网上零零散散的教程可以阅读。就是在这种情况下，本书的作者方志朋在公司技术选型以及后续的微服务落地过程中，逐渐有了自己的积累和理解，同时在博客中连载了“史上最简单的 Spring Cloud 教程”。此教程一出，就受到广大程序员的欢迎，因此最终整理为此书。

纵览全书，文字清晰明了，通过理论结合实践的方式介绍了 Spring Cloud 的每一个组件的实践，并解读了部分源代码。图文并茂，语言朴实，不愧为“简单”之名。本书融合了作者实施微服务的一线经验和心得，具体指导了 Spring Cloud 在落地方面的实践，非常值得参考。

深圳小安时代互联网金融服务有限公司，研发中心总经理
于际予

序　三

好像一夜之间，全世界都在讨论微服务，讨论如何使用 Spring Cloud 来构建自己的微服务体系。但是一切都像是“徒手抓泥鳅”，无处下手，很是难受，方兄在前期也遇到过这种情况。这在一般人看来就是“算了，还是换一种技术框架吧”的结果，但是方兄反其道而行之，不仅把这些东西一一梳理起来，编写了一系列的教程，而且作为 Spring Cloud 中国社区联合发起人，他还向 Spring Cloud 开源社区贡献了大量的代码。

从 Spring Cloud 的造诣上来说，方兄很是值得我们学习。此书循循善诱，从字里行间就能感受到方兄在理论和实践方面“两手抓、两手强”，造诣之深，着实难得。更难得的是方兄的分享精神，他毫无保留地把自己的项目、思考、总结和方法集结成书，与广大读者共同成长，可谓无私。

好了，闲言少叙，赶紧翻开这本书，你会喜欢上它的！

前平安普惠数据应用架构师，微信公众号“一名叫大蕉的程序员”出品人
杨钊

序　二

2015 年，Spring Cloud 项目最初在 GitHub 上出现时，我就一直在关注其项目发展。到 2016 年，国内关注 Spring Cloud 的人越来越多，但是相应学习交流的平台和材料却比较分散，不利于学习交流，机缘巧合之下 Spring Cloud 中国社区诞生并发展至今。

Spring Cloud 中国社区是国内首个 Spring Cloud 构建微服务架构的交流社区，也是致力于为 Spring Boot 和 Spring Cloud 技术人员提供分享和交流的平台，推动 Spring Cloud 在中国的普及和应用。因为技术交流，我和志朋相识。志朋所写的博客“史上最简单的 Spring Cloud 教程”系列，浏览量已经达到数百万，并获得了广大读者的一致好评，为 Spring Cloud 学习者和开发者答疑解惑。

《深入理解 Spring Cloud 与微服务构建》一书总结并延伸了博客中的精华内容，结合社区中的常见问题进行了方案梳理，抛砖引玉，通俗易懂，涵盖了 Spring Cloud 中常用的组件和项目案例实战。希望本书能够帮助开发者快速使用 Spring Cloud 对企业 IT 架构微服务进行开发和改造！

Spring Cloud 中国社区创始人
许进

序　四

本书对于想了解 Spring Cloud 但又无从下手的读者来说简直就是福音，看完前几章就可以对 Spring Cloud 有大致的了解。

本书对于那些已经了解了 Spring Cloud 的专业人士来说也是福音，因为这里可能有你没见过的技术点。

作为见证了本书从无到有的旁观者来说，本书不仅解决了我在技术上的疑惑，还让我明白了该如何面对挫折。

大概在开始写作一个多月的时间后，志朋感到很迷茫，害怕技术点写得不够深入，对购书的读者不负责任。大概有近一周的时间，他惶惶不可终日。所幸的是，他的博客中有很多的读者和网友发信息鼓励他。其中一个网友说“只要投入了最大的热情，专注于技术就会感到很满足”，这句话让志朋重新燃起了斗志，有了坚持写下去的动力和勇气。

为了保证本书的质量，将一手资料及时准确地传送给读者，他翻阅了大量的中英文资料，反复论证内容的可行性，举一反三。为了让读者能看的懂，并且有看下去的欲望，他反复打磨每一句话，尽可能使其简单化，通俗易懂。

世间万物，都有其精神，本书之精神乃艰苦奋斗、专注专业的工匠精神！

深圳捷顺科技架构师

鄂超

序　五

当今世界，互联网飞速发展，并融入生活的方方面面，人们对互联网产品提出了更严格的要求，重体验、重响应速度、关联性强、业务场景越来越复杂等是互联网产品的新特点。这对软件架构师提出了新的要求，如何使得系统架构能够轻松持续改进、快速部署、业务之间低耦合，成为软件架构师的新的思考方向。

微服务作为系统解决这些问题的一种思路应运而生，方兴未艾。本书作者对微服务的见解独到，实战经验丰富，为我们介绍了什么是微服务，微服务与传统软件架构的对比，为什么要使用微服务，微服务的设计原则和具体特点，微服务应该具备的功能，微服务的应用场景，需要使用的组件，以及如何综合运用这些组件构建整个微服务框架。

本书主要针对 Java 开发者构建微服务框架，作者比较青睐于 Java 语言的 Spring Cloud 微服务框架，究其原因是 Spring Cloud 有快速开发、持续交付和易于部署等特点，且开源社区比较活跃，同时有国际巨头公司的推动。本书在 Spring Cloud 框架范围内，介绍了服务注册和发现的 Eureka 组件、负载均衡 Ribbon 组件、熔断器 Hystrix 组件、路由网关 Zuul 组件、Spring Cloud 配置中心、服务链路追踪等内容，同时也与其他微服务框架做了对比，拓展了微服务知识的深度和广度。本书结构清晰，行文优美，每一个例子都经过作者斟酌再三，力求使用最简单的例子，将复杂的逻辑原理阐述清楚，让读者印象深刻。

本书是一本用于微服务入门和提高的好书，相信你读完本书后定能感受到作者的严谨与智慧，同时对微服务有一定的理解，并能够灵活运用。

平安集团
胡益雄

前　言

近年来随着互联网的飞速发展，各行各业都在拥抱互联网。互联网给人类生活带来了翻天覆地的变化，人们在享受互联网给生活带来便捷的同时，业务需求的发展也对互联网技术提出了更高的要求，传统的单体架构对越来越复杂的业务需求显得力不从心。此外，随着大数据、云计算和人工智能的飞速发展，软件的架构显得越来越重要。近几年来，“微服务”这一名词在各大网站、论坛、演讲中出现的频率足以让人们感觉到它对软件架构带来的影响。目前，各大公司都在纷纷采用微服务架构。

Spring Cloud 作为 Java 语言的微服务落地框架，在 Spring 开源社区和 Pivatol、Netflix 两大公司的推动下飞速发展，得到了众多开发者的认可，Spring Cloud 在未来很可能成为微服务框架的领导者和规范。和众多 Spring Cloud 开发者一样，我在工作和学习中对 Spring Cloud 系列框架、组件非常痴迷。我利用业余时间在 CSDN 博客上发表了一系列关于 Spring Cloud 的文章，受到广大开发人员的欢迎，在短短半年的时间里，Spring Cloud 系列文章的阅读量就超过 200 万。另外，作为 Spring Cloud 中国社区的联合发起人，我持续为社区贡献文章，得到了社区朋友们的认可。因为 Spring Cloud 是一个新技术，很多人对此还不是很了解，所以希望我的文章可以作为大家学习的资料，也欢迎读者关注我的博客 http://blog.csdn.net/forezp。

沿袭了博客的写作风格，我花费了半年的时间和大量的心血来完成本书。从 Spring Cloud 的基础组件开始讲解，对关键组件做了源码分析，力求帮助读者深入理解原理。同时也重点讲解了如何在 Spring Cloud 微服务系统中进行身份认证和权限安全的验证。在本书的最后，以一个综合案例来全面讲解 Spring Cloud 是如何构建微服务的，这个案例是我在学习和工作过程中使用 Spring Cloud 的提炼和总结，具有非常高的参考价值。

本书内容

本书共分为 16 章，各章主要内容如下。

第 1 章介绍了什么是微服务、为什么需要微服务、微服务的优缺点和挑战，并且将单体架构的系统和微服务架构的系统进行了比较。

第 2 章主要介绍微服务应该具备的功能，以及 Spring Cloud 的基本组件，最后介绍了 Spring Cloud 与 Dubbo、Kubernetes 之间的比较及优缺点。

第 3、4 章介绍了构建微服务的准备工作：开发环境的构建和 Spring Boot 的使用。其中，第 3 章介绍了开发环境的构建，包括 JDK 的安装、IDEA 和 Maven 的使用等；第 4 章介绍了 Spring Boot 的基本使用方法，包括 Spring Boot 的特点和优点、用 IDEA 创建一个 Spring Boot

项目、Spring Boot 配置文件详情、Spring Boot 的 Actuator 模块，以及 Spring Boot 集成 JPA、Redis、Swagger2 等。

第 5～9 章介绍了 Spring Cloud 框架的基础模块——Spring Cloud Netflix 模块，涵盖了 Spring Cloud 构建微服务的基础组件。例如 Eureka、Ribbon、Feign、Hystrix 和 Zuul 等，这些组件为微服务系统提供了基本的服务治理能力。以案例为切入点，由浅入深介绍这些组件，并从源码的角度分析这些组件的工作原理。

第 10 章介绍了分布式配置中心 Spring Cloud Config，详细讲解了 Config Server 如何从本地仓库和远程 Git 仓库读取配置文件，以及如何构建高可用的分布式配置中心和使用消息总线刷新配置文件。

第 11 章介绍了链路追踪组件 Spring Cloud Sleuth，包括微服务系统为什么需要链路追踪组件，并以案例的形式详细介绍了如何在 Spring Cloud 微服务系统中使用链路追踪，以及如何传输、存储和展示链路数据。

第 12 章以案例的形式介绍了 Spring Boot Admin，包括 Spring Boot Admin 在微服务系统中的应用、在 Spring Boot Admin 中集成安全登录组件。

第 13～15 章介绍了 Spring Cloud 微服务系统的安全验证模块，包括 Spring Boot Security 组件和 Spring Cloud OAuth2 模块。第 13 章详细介绍了如何在 Spring Boot 应用中使用 Spring Boot Security；第 14 章介绍了如何在 Spring Cloud 微服务系统中使用 Spring Cloud OAuth2 来保护微服务的系统安全；第 15 章介绍了如何在 Spring Cloud 微服务系统中使用 Spring Cloud OAuth2 和 JWT 来保护微服务的系统安全。

第 16 章以一个综合案例介绍了使用 Spring Cloud 构建微服务系统的全过程，该案例是对全书内容的总结和提炼。

本书特色

1. 案例丰富，通俗易懂

我的写作目标之一就是将复杂的事情简单化，从而让读者轻松地学习到技术。本书用丰富的案例循序渐进地讲解了如何使用 Spring Cloud 构建微服务。

2. 深入浅出，透析本质

以案例为切入点，对 Spring Cloud 关键组件进行源码解读，深入讲解原理，并在案例中使用大量的图解，包括展示图、架构图等，帮助读者深入理解。最后以一个综合案例完整讲解了如何使用 Spring Cloud 构建微服务，达到学以致用的目的。

3. 网络资源，技术支持

本书中所有的源码按章节划分，每一章节都有独立的源码，方便读者使用和理解。读者可以扫描下方二维码，到我的微信公众号（微信号 walkingstory）中下载源码。源码打开即用，

可以轻松运行。读者可以一边看书，一边看源码，易于快速学习和理解。

阅读建议

本书的读者对象既可以是 Spring Cloud 的初学者，也可以是经验丰富的架构师。建议循序渐进、从前往后对照源码通读本书。本书采用的 Spring Cloud 版本为 Dalston，Spring Boot 版本为 1.5.3。

建议和反馈

由于作者能力有限，虽然对书稿做了多次认真的检查和修改，但错漏之处在所难免，敬请读者批评指正。读者可以到我的微信公众号或者博客中留言反馈，也可以将意见或建议发至本书编辑的邮箱 zhangshuang@ptpress.com.cn，我会及时给出解答。

致谢

感谢我的家人在我写作本书过程中所给予的支持和鼓励。感谢大学时的导师王为民教授在精神上对我的鼓舞，使我如沐春风，终身受益。感谢我的同事提出的宝贵意见，和你们一起工作非常荣幸，也非常开心。感谢所有的技术朋友给我的帮助和建议，包括 Spring Cloud 中国社区的小伙伴们，以及各大社区的技术朋友们。感谢编辑张爽在本书写作和出版过程中所做的工作。感谢这么多良师益友……

方志朋

2017 年秋

目　录

第 1 章　微服务简介

随着互联网技术的飞速发展，目前全球超过一半的人口在使用互联网，人们的生活随着互联网的发展，发生了翻天覆地的变化。各行各业都在应用互联网，国家政策也在大力支持互联网的发展。随着越来越多的用户参与，业务场景越来越复杂，传统的单体架构已经很难满足互联网技术的发展要求。这主要体现在两方面，一是随着业务复杂度的提高，代码的可维护性、扩展性和可读性在降低；二是维护系统的成本、修改系统的成本在提高。所以，改变单体应用架构已经势在必行。另外，随着云计算、大数据、人工智能的飞速发展，对系统架构也提出了越来越高的要求。

微服务，是著名的 OO（面向对象，Object Oriented）专家 Martin Fowler 提出来的，它是用来描述将软件应用程序设计为独立部署的服务的一种特殊方式。最近两年，微服务在各大技术会议、文章、书籍上出现的频率已经让人们意识到它对于软件领域所带来的影响力。微服务架构的系统是一个分布式系统，按业务领域划分为独立的服务单元，有自动化运维、容错、快速演进的特点，它能够解决传统单体架构系统的痛点，同时也能满足越来越复杂的业务需求。

1.1　单体架构及其存在的不足

1.1.1　单体架构简介

在软件设计中，经常提及和使用经典的 3 层模型，即表示层、业务逻辑层和数据访问层。

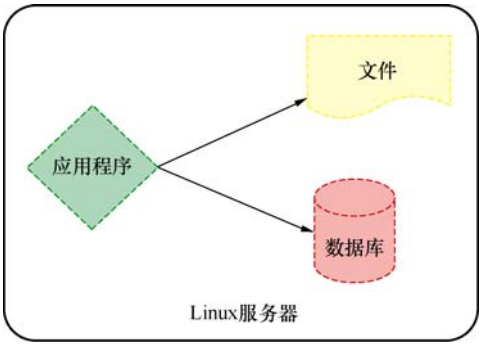
- 表示层：用于直接和用户交互，也称为交互层，通常是网页、UI 等。
- 业务逻辑层：即业务逻辑处理层，例如用户输入的信息要经过业务逻辑层的处理后，才能展现给用户。
- 数据访问层：用于操作数据库，用户在表示层会产生大量的数据，通过数据访问层对数据库进行读写操作。

虽然在软件设计中划分了经典的 3 层模型，但是对业务场景没有划分。一个典型的单体应用就是将所有的业务场景的表示层、业务逻辑层和数据访问层放在一个工程中，最终经过编译、

打包，部署在一台服务器上。例如典型的 J2EE 工程，它是将表示层的 JSP、业务逻辑层的 Service、Controller 和数据访问层的 Dao，打成 war 包，部署在 Tomcat、Jetty 或者其他 Servlet 容器中运行。经典的单体应用如图 1-1 所示。

在一个小型应用的初始阶段，访问量较小，应用只需要一台服务器就能够部署所有的资源，例如将应用程序、数据库、文件资源等部署在同一台服务器上。最典型的就是 LAMP 系统，即服务器采用 Linux 系统，开发应用程序的语言为 PHP，部署在 Apache 服务器上，采用 MySQL 数据库。在应用程序的初始阶段，采用这种架构的性价比是非常高的，开发速度快，开发成本低，只需要一台廉价的服务器。此时的服务器架构如图 1-2 所示。

▲图 1-1　经典的单体应用架构



▲图 1-2　LAMP 应用服务器示意图

1.1.2　单体架构存在的不足

在应用的初始阶段，单体架构无论是在开发速度、运维难度上，还是服务器的成本上都有着显著的优势。在一个产品的前景不明确的初始阶段，用单体架构是非常明智的选择。随着应用业务的发展和业务复杂度的提高，这种架构明显存在很多的不足，主要体现在以下 3 个方面。

- 业务越来越复杂，单体应用的代码量越来越大，代码的可读性、可维护性和可扩展性下降，新人接手代码所需的时间成倍增加，业务扩展带来的代价越来越大。
- 随着用户越来越多，程序承受的并发越来越高，单体应用的并发能力有限。
- 测试的难度越来越大，单体应用的业务都在同一个程序中，随着业务的扩张、复杂度的增加，单体应用修改业务或者增加业务或许会给其他业务带来一定的影响，导致测试难度增加。

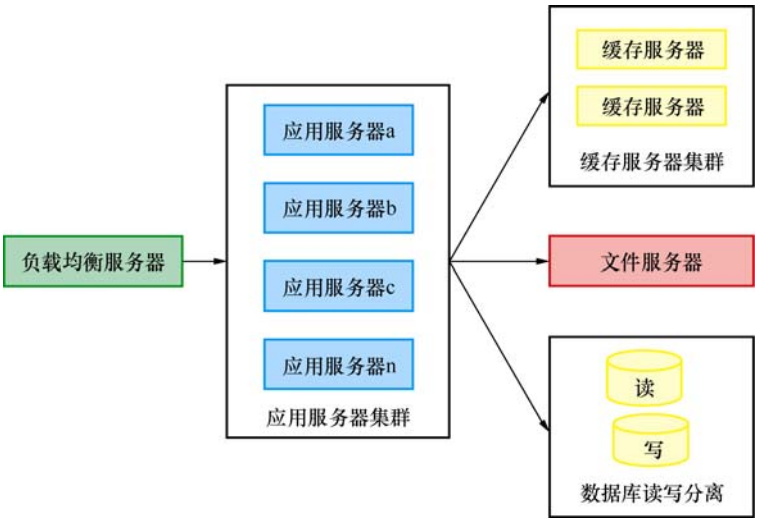
1.1.3　单体架构使用服务器集群及存在的不足

随着业务的发展，大多数公司会将单体应用进行集群部署，并增加负载均衡服务器（例如 Nginx 等）。另外，还需要增加集群部署的缓存服务器和文件服务器，并将数据库读写分离，以应对用户量的增加而带来的高并发访问量。此时的系统架构如图 1-3 所示。

用负载均衡服务器分发高并发的网络请求，用户的访问被分派到不同的应用服务器，应用服务器的负载不再成为瓶颈，用户量增加时，添加应用服务器即可。通过添加缓存服务器来缓

解数据库的数据以及数据库读取数据的压力。大多数的读取操作是由缓存完成的，但是仍然有少数读操作是从数据库读取的，例如缓存失效、实时数据等。当有大量的读写操作时，将数据库进行读写分离是一个不错的选择，例如 MySQL 的主从热备份，通过相关配置可以将主数据库服务器的数据同步到从数据库服务器，实现数据库的读写分离，读写分离能够改善数据库的负载能力。

这种架构有一定的处理高并发的能力，也能应对一定复杂的业务需求，改善了系统的性能，但是依然没有改变系统为单体架构的事实，此时存在的不足之处如下。



▲图 1-3　单体服务的集群化

- 系统仍然为单体应用，大量的业务必然会有大量的代码，代码的可读性和可维护性依然很差。
- 面对海量的用户，数据库将会成为瓶颈，解决方案将使用分布式数据库，也就是将数据库进行分库分表。
- 持续交付能力差，业务越复杂，代码越多，修改代码和添加代码所需的时间越长。新人熟悉代码的时间长、成本高。

由此看见，在应用初期，单体应用从成本、开发时间和运维等方面都有明显的优势。但是随着业务量和用户量的增加，它所暴露出来的缺点也显而易见。单体架构已经不能满足复杂的业务和海量的用户系统，改变单体架构势在必行。

1.2 微服务

微服务是最近一两年才出现的新名词，它在各大技术社区、博客、论坛和新闻报道中经常被提及，是程序员和架构师经常讨论的话题。的确，微服务已经是技术圈的热门话题，那么到

底什么是微服务呢？微服务产生的意义又是什么呢？微服务有哪些优势和缺点？另外，微服务与 SOA 架构有什么关系？下面让我来为你逐一阐述。

1.2.1　什么是微服务

“微服务”最初是由 Martin Fowler 在 2014 年写的一篇文章《MicroServices》中提出来的。关于 Martin Fowler 的介绍，维基百科上是这样描述的：

> Martin Fowler，软件工程师，也是一个软件开发方面的著作者和国际知名演说家，专注于面向对象分析与设计、统一建模语言、领域建模，以及敏捷软件开发方法，包括极限编程。主要著作有《可重用对象模型》《重构——改善既有代码的设计》《企业应用架构模式》《规划极限编程》等。
>
> （摘自网络）

对于微服务，业界没有一个严格统一的定义，但是作为“微服务”这一名词的发明人，Martin Fowler 对微服务的定义似乎更具有权威性和指导意义。他的理解如下：

> 简而言之，微服务架构的风格，就是将单一程序开发成一个微服务，每个微服务运行在自己的进程中，并使用轻量级机制通信，通常是 HTTP RESTFUL API。这些服务围绕业务能力来划分构建的，并通过完全自动化部署机制来独立部署。这些服务可以使用不同的编程语言，以及不同数据存储技术，以保证最低限度的集中式管理。

以我个人对这段话的理解，总结微服务具有如下特点。

- 按业务划分为一个独立运行的程序，即服务单元。
- 服务之间通过 HTTP 协议相互通信。
- 自动化部署。
- 可以用不同的编程语言。
- 可以用不同的存储技术。
- 服务集中化管理。
- 微服务是一个分布式系统。

根据这些特点，下面来进一步阐述微服务。

1. 微服务单元按业务来划分

微服务的“微”到底需要定义到什么样的程度，这是一个非常难以界定的概念，可以从以下 3 个方面来界定：一是根据代码量来定义，根据代码的多少来判断程序的大小；二是根据开发时间的长短来判断；三是根据业务的大小来划分。

根据 Martin Fowler 的定义，微服务的“微”是按照业务来划分的。一个大的业务可以拆分成若干小的业务，一个小的业务又可以拆分成若干更小的业务，业务到底怎么拆分才算合适，这需要开发人员自己去决定。例如微博最常见的功能是微博内容、关注和粉丝，而其中微博内容又有点赞、评论等，如何将微博这个复杂的程序划分为单个的服务，需要由开发团队去决定。

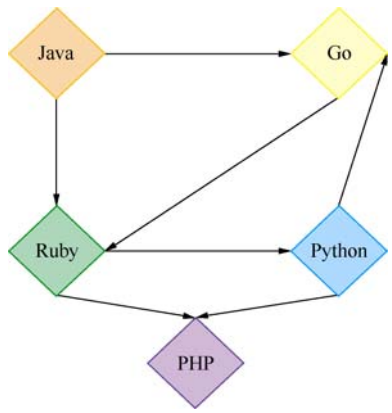
按业务划分的微服务单元独立部署，运行在独立的进程中。这些微服务单元是高度组件化

的模块，并提供了稳定的模块边界，服务与服务之间没有任何的耦合，有非常好的扩展性和复用性。

传统的软件开发模式通常由 UI 团队、服务端团队、数据库和运维团队构成，相应地将软件按照职能划分为 UI、服务端、数据库和运维等模块。通常这些开发人员各司其职，很少有人跨职能去工作。如果按照业务来划分服务，每个服务都需要独立的 UI、服务端、数据库和运维。也就是说，一个小的业务的微服务需要动用一个团队的人去协作，这显然增加了团队与团队之间交流协作的成本。所以产生了跨职能团队，这个团队负责一个服务的所有工作，包括 UI、服务端和数据库。当这个团队只有 1～2 个人的时候，就对开发人员提出了更高的要求。

2. 微服务通过 HTTP 来互相通信

按照业务划分的微服务单元独立部署，并运行在各自的进程中。微服务单元之间的通信方式一般倾向于使用 HTTP 这种简单的通信机制，更多的时候是使用 RESTful API 的。这种接受请求、处理业务逻辑、返回数据的 HTTP 模式非常高效，并且这种通信机制与平台和语言无关。例如用 Java 写的服务可以消费用 Go 语言写的服务，用 Go 写的服务又可以消费用 Ruby 写的服务。不同的服务采用不同的语言去实现，不同的平台去部署，它们之间使用 HTTP 进行通信，如图 1-4 所示。



▲图 1-4　不同语言、不同的平台的微服务相互调用

服务与服务之间也可以通过轻量级的消息总线来通信，例如 RabbitMQ、Kafaka 等。通过发送消息或者订阅消息来达到服务与服务之间通信的目的。

服务与服务通信的数据格式，一般为 JSON、XML，这两种数据格式与语言、平台、通信协议无关。一般来说，JSON 格式的数据比 XML 轻量，并且可读性也比 XML 要好。另外一种就是用 Protobuf 进行数据序列化，经过序列化的数据为二进制数据，它比 JSON 更轻量。用 Protobuf 序列化的数据为二进制数据，可读性非常差，需要反序列化才能够读懂。由于用 Protobuf 序列化的数据更为轻量，所以 Protobuf 在通信协议和数据存储上十分受欢迎。

服务与服务之间通过 HTTP 或者消息总线的方式进行通信，这种方式存在弊端，其通信机制是不可靠的，虽然成功率很高，但还是会有失败的时候。

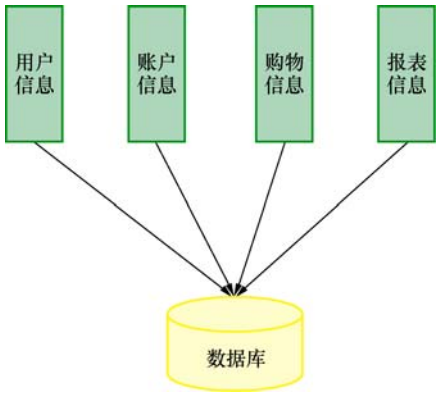
3. 微服务的数据库独立

在单体架构中，所有的业务都共用一个数据库。随着业务量的增加，数据库的表的数量越来越多，难以管理和维护，并且数据量的增加会导致查询速度越来越慢。例如，一个应用有这样几个业务：用户的信息、用户的账户、用户的购物车、数据报表服务等。典型的单体架构如

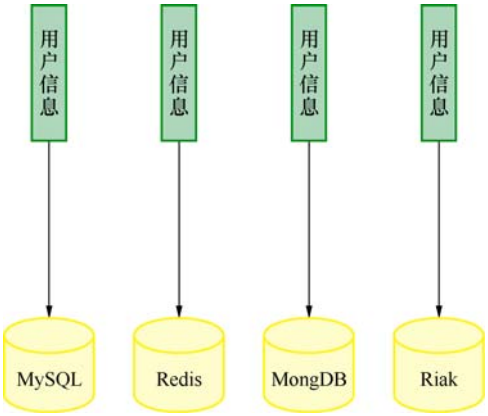
图 1-5 所示。

微服务的一个特点就是按业务划分服务，服务与服务之间无耦合，就连数据库也是独立的。一个典型的微服务的架构就是每个微服务都有自己独立的数据库，数据库之间没有任何联系。这样做的好处在于，随着业务的不断扩张，服务与服务不需要提供数据库集成，而是提供 API 接口相互调用；还有一个好处是数据库独立，单业务的数据量少，易于维护，数据库性能有着明显的优势，数据库的迁移也很方便。

另外，随着存储技术的发展，数据库的存储方式不再仅仅是关系型数据库，非关系数据库的应用也非常广泛，例如 MongDB、Redis，它们有着良好的读写性能，因此越来越受欢迎。一个典型的微服务的系统，可能每一个服务的数据库都不相同，每个服务所使用的数据存储技术需要根据业务需求来选择，如图 1-6 所示。



▲图 1-5　单体服务共享一个数据库



▲图 1-6　微服务的数据库独立

4. 微服务的自动化部署

在微服务架构中，系统会被拆分为若干个微服务，每个微服务又是一个独立的应用程序。单体架构的应用程序只需要部署一次，而微服务架构有多少个服务就需要部署多少次。随着服务数量的增加，如果微服务按照单体架构的部署方式，部署的难度会呈指数增加。业务的粒度划分得越细，微服务的数量就越多，这时需要更稳定的部署机制。随着技术的发展，尤其是 Docker 容器技术的推进，以及自动化部署工具（例如开源组件 Jenkins）的出现，自动化部署变得越来越简单。

自动化部署可以提高部署的效率，减少人为的控制，部署过程中出现错误的概率降低，部署过程的每一步自动化，提高软件的质量。构建一个自动化部署的系统，虽然在前期需要开发人员或者运维人员的学习，但是对于整个软件系统来说是一个全新的概念。在软件系统的整个生命周期之中，每一步是由程序控制的，而不是人为控制，软件的质量提高到了一个新的高度。随着 DevOps 这种全新概念的推进，自动化部署必然会成为微服务部署的一种方式。

5. 服务集中化管理

微服务系统是按业务单元来划分服务的，服务数量越多，管理起来就越复杂，因此微服务必须使用集中化管理。目前流行的微服务框架中，例如 Spring Cloud 采用 Eureka 来注册服务和发现服务，另外，Zookeeper、Consul 等都是非常优秀的服务集中化管理框架。

6. 分布式架构

分布式系统是集群部署的，由很多计算机相互协作共同构成，它能够处理海量的用户请求。当分布式系统对外提供服务时，用户是毫不知情的，还以为是一台服务器在提供服务。

分布式系统的复杂任务通过计算机之间的相互协作来完成，当然简单的任务也可以在一台计算机上完成。

分布式系统通过网络协议来通信，所以分布式系统在空间上没有任何限制，即分布式服务器可以部署不同的机房和不同的地区。

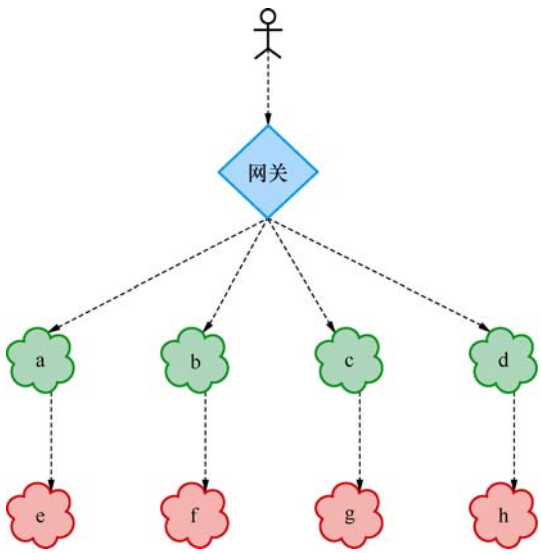
微服务架构是分布式架构，分布式系统比单体系统更加复杂，主要体现在服务的独立性和服务相互调用的可靠性，以及分布式事务、全局锁、全局 Id 等，而单体系统不需要考虑这些复杂性。

另外，分布式系统的应用都是集群化部署，会给数据一致性带来困难。分布式系统中的服务通信依赖于网络，网络不好，必然会对分布式系统带来很大的影响。在分布式系统中，服务之间相互依赖，如果一个服务出现了故障或者是网络延迟，在高并发的情况下，会导致线程阻塞，在很短的时间内该服务的线程资源会消耗殆尽，最终使得该服务不可用。由于服务的相互依赖，可能会导致整个系统的不可用，这就是“雪崩效应”。为了防止此类事件的发生，分布式系统必然要采取相应的措施，例如“熔断机制”。

7. 熔断机制

为了防止“雪崩效应”事件的发生，分布式系统采用了熔断机制。在用 Spring Cloud 构建的微服务系统中，采用了熔断器（即 Hystrix 组件的 Circuit Breaker）去做熔断。

例如在微服务系统中，有 a、b、c、d、e、f、g、h 等多个服务，用户的请求通过网关后，再到具体的服务，服务之间相互依赖，例如服务 b 依赖于服务 f，一个对外暴露的 API 接口需要服务 b 和服务 f 相互协作才能完成。服务之间相互依赖的架构图如图 1-7 所示。

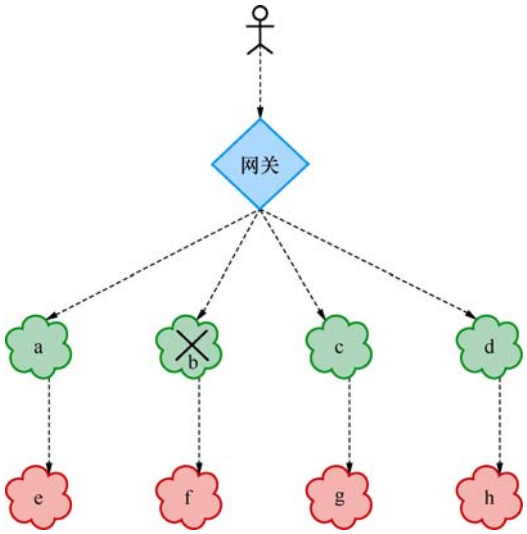


▲图 1-7 服务之间相互依赖

如果此时服务 b 出现故障或者网络延迟，在高并发的情况下，服务 b 会出现大量的线程阻塞，有可能在很短的时间内线程资源就被消耗完了，导致服务 b 的不可用。如果服务 b 为较

底层的服务，会影响到其他服务，导致其他服务会一直等待服务 b 的处理。如果服务 b 迟迟不处理，大量的网络请求不仅仅堆积在服务 b，而且会堆积到依赖于服务 b 的其他服务。而因服务 b 出现故障影响的服务，也会影响到依赖于因服务 b 出现故障影响的服务的其他服务，从而由 b 开始，影响到整个系统，导致整个系统的不可用。这是一件非常可怕的事，因为服务器运营商的不可靠，必然会导致服务的不可靠，而网络服务商的不可靠性，也会导致服务的不可靠。在高并发的场景下，稍微有点不可靠，由于故障的传播性，会导致大量的服务不可用，甚至导致整个系统崩溃。

为了解决这一难题，微服务架构引入了熔断机制。当服务 b 出现故障，请求失败次数超过设定的阀值之后，服务 b 就会开启熔断器，之后服务 b 不进行任何的业务逻辑操作，执行快速失败，直接返回请求失败的信息。其他依赖于 b 的服务就不会因为得不到响应而线程阻塞，这时除了服务 b 和依赖于服务 b 的部分功能不可用外，其他功能正常。熔断服务 b 如图 1-8 所示。



▲图 1-8　将服务 b 熔断

熔断器还有另外一个机制，即自我修复的机制。当服务 b 熔断后，经过一段时间，半打开熔断器。半打开的熔断器会检查一部分请求是否正常，其他请求执行快速失败，检查的请求如果响应成功，则可以判定服务 b 正常了，就会关闭服务 b 的熔断器；如果服务 b 还不正常，则继续打开熔断器。这种自我熔断机制和自我修复机制在微服务架构中有着重要的意义，一方面，它使程序更加健壮，另一方面，为开发和运维减少很多不必要的工作。

最后，熔断组件往往会提供一系列的监控，例如服务可用与否、熔断器是否被打开、目前的吞吐量、网络延迟状态的监控等，从而很容易让开发人员和运维人员实时地了解服务的状况。

1.2.2　微服务的优势

相对于单体服务来说，微服务具有很多的优势，主要体现在以下方面。

（1）将一个复杂的业务分解成若干小的业务，每个业务拆分成一个服务，服务的边界明确，将复杂的问题简单化。服务按照业务拆分，编码也是按照业务来拆分，代码的可读性和可扩展性增加。新人加入团队，不需要了解所有的业务代码，只需要了解他所接管的服务的代码，新人学习时间成本减少。

（2）由于微服务系统是分布式系统，服务与服务之间没有任何的耦合。随着业务的增加，可以根据业务再拆分服务，具有极强的横向扩展能力。随着应用的用户量的增加，并发量增加，可以将微服务集群化部署，从而增加系统的负载能力。简而言之，微服务系统的微服务单元具有很强的横向扩展能力。

（3）服务与服务之间通过 HTTP 网络通信协议来通信，单个微服务内部高度耦合，服务与服务之间完全独立，无耦合。这使得微服务可以采用任何的开发语言和技术来实现。开发人员不再被强迫使用公司以前的技术或者已经过时的技术，而是可以自由选择最适合业务场景的或者最适合自己的开发语言和技术，提高开发效率、降低开发成本。

（4）如果是一个单体的应用，由于业务的复杂性、代码的耦合性，以及可能存在的历史问题。在重写一个单体应用时，要求重写的应用的人员了解所有的业务，所以重写单体应用是非常困难的，并且重写风险也较高。如果是微服务系统，由于微服务系统是按照业务的进行拆分的，并且有坚实的服务边界，所以重写某个服务就相当于重写某一个业务的代码，非常简单。

（5）微服务的每个服务单元都是独立部署的，即独立运行在某个进程里。微服务的修改和部署对其他服务没有影响。试想，假设一个应用只有一个简单的修改，如果是单体架构，需要测试和部署整个应用；而如果采用微服务架构，只需要测试并部署被修改的那个服务，这就大大减少了测试和部署的时间。

（6）微服务在 CAP 理论中采用的是 AP 架构，即具有高可用和分区容错的特点。高可用主要体现在系统 7 × 24 小时不间断的服务，它要求系统有大量的服务器集群，从而提高了系统的负载能力。另外，分区容错也使得系统更加健壮。

1.3 微服务的不足

凡事都有两面性，微服务也不例外，微服务相对于单体应用来说具有很多的优势，当然也有它的不足，主要体现在如下方面。

- 微服务的复杂度。
- 分布式事务。
- 服务的划分。
- 服务的部署。

1.3.1 微服务的复杂度

构建一个微服务系统并不是一件容易的事，微服务系统是分布式系统，构建的复杂度远远超过单体系统，开发人员需要付出一定的学习成本去掌握更多的架构知识和框架知识。服务与服务之间通过 HTTP 协议或者消息传递机制通信，开发者需要选出最佳的通信机制，并解决网络服务较差时带来的风险。

另外服务与服务之间相互依赖，如果修改某一个服务，会对另外一个服务产生影响，如果掌控不好，会产生不必要的麻烦。由于服务的依赖性，测试也会变得复杂，比如修改一个比较基础的服务，可能需要重启所有的服务才能完成测试。

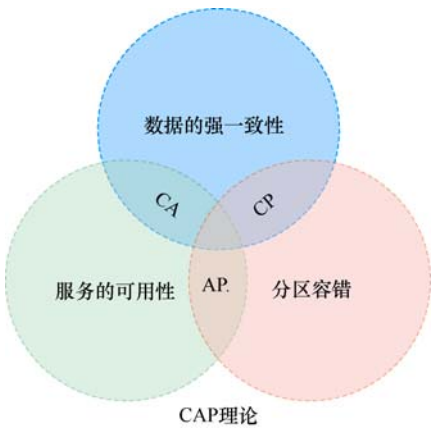
1.3.2 分布式事务

微服务架构所设计的系统是分布式系统。分布式系统有一个著名的 CAP 理论，即同

时满足“一致性”“可用性”和“分区容错”是一件不可能的事。CAP 理论是由 Eric Brewer 在 2000 年 PODC 会议上提出的，该理论在两年后被证明成立。CAP 理论告诉架构师不要妄想设计出同时满足三者的系统，应该有所取舍，设计出适合业务的系统。CAP 理论如图 1-9 所示。

- Consistency：指数据的强一致性。如果写入某个数据成功，之后读取，读到的都是新写入的数据；如果写入失败，之后读取的都不是写入失败的数据。
- Availability：指服务的可用性。
- Partition-tolerance：指分区容错。

在分布式系统中，P 是基本要求，而单体服务是 CA 系统。微服务系统通常是一个 AP 系统，即同时满足了可用性和分区容错。这就有了一个难题：在分布式系统中如何保证数据的一致性？这就是大家经常讨论的分布式事务。



▲图 1-9　CAP 理论示意图

在微服务系统中，每个服务都是独立的进程单元，每个服务都有自己的数据库。通常情况下，只有关系型数据库在特定的数据引擎下才支持事务，而大多数非关系型数据库是不支持事务的，例如 MongDB 是不支持事务的，而 Redis 是支持事务的。在微服务架构中，分布式事务一直都是一个难以解决的问题，业界给出的解决办法通常是两阶段提交。

网上购物在日常生活中是一个非常普通的场景，假设我在淘宝上购买了一部手机，需要从我的账户中扣除 1000 元钱，同时手机的库存数量需要减 1。当然需要在卖方的账户中加 1000 元钱，为了使案例简单化，暂时不用考虑。

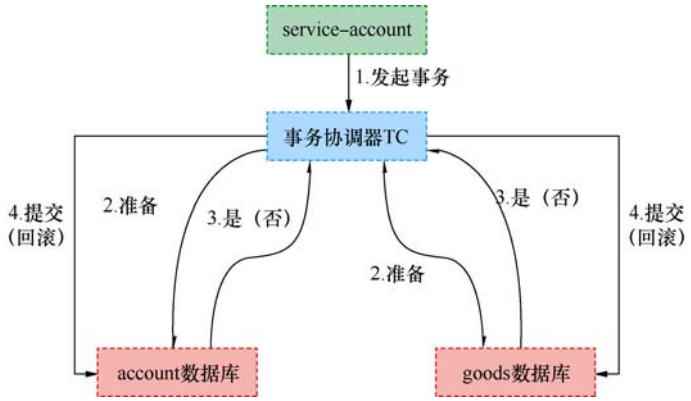
如果这是一个单体应用，并且使用支持事务的 MySQL 数据库（InnoDB 数据库引擎才支持事务），我们可能这样写代码：

```
@Transactional
public void update() throws RuntimeException{
    updateAccountTable(); //更新账户表
    updateGoodsTable(); //更新商品表
}
```

如果是微服务架构，账户是一个服务，而商品是一个服务，这时不能用数据库自带的事务，因为这两个数据表不在一个数据库中。因此常常用到两阶段提交，两阶段提交的过程如图 1-10 所示。

第一阶段，service-account 发起一个分布式事务，交给事务协调器 TC 处理，事务协调器 TC 向所有参与的事务的节点发送处理事务操作的准备操作。所有的参与节点执行准备操作，将 Undo 和 Redo 信息写进日志，并向事务管理器返回准备操作是否成功。

第二阶段，事务管理器收集所有节点的准备操作是否成功，如果都成功，则通知所有的节点执行提交操作；如果有一个失败，则执行回滚操作。



▲图 1-10 两阶段提交示意图

两阶段提交，将事务分成两部分能够大大提高分布式事务成功的概率。如果在第一阶段都成功了，而执行第二阶段的某一个节点失败，仍然导致数据的不准确，这时一般需要人工去处理，这就是当初在第一步记录日志的原因。另外，如果分布式事务涉及的节点很多，某一个节点的网络出现异常会导致整个事务处于阻塞状态，大大降低数据库的性能。所以一般情况下，尽量少用分布式事务。

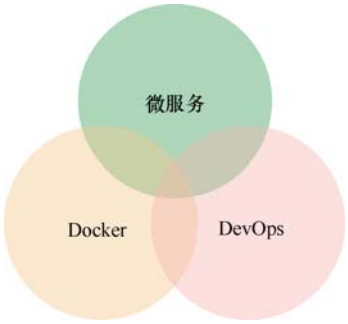
1.3.3 服务的划分

将一个完整的系统拆分成很多个服务，是一件非常困难的事，因为这涉及了具体的业务场景，比命名一个类更加困难。对于微服务的拆分原则，Martin Fowler 给出的建议是；服务是可以被替换和更新的。也就是服务和服务之间无耦合，任何一个服务都可以被替换，服务有自己严格的边界。当然这个原则很抽象，根据具体的业务场景来拆分服务，需要依靠团队人员对业务的熟悉程度和理解程度，并考虑与已有架构的冲突、业务的扩展性、开发的风险和未来业务的发展等诸多因素。

领域驱动设计是一个全新的概念，也是一个比较理想的微服务拆分的理念。领域驱动设计通过代码和数据分析找到合理的切分点，并通过数据分析来判断服务的划分边界和划分粒度。目前来说，在中国几乎没有公司去落地领域驱动设计这个理念，随着微服务的发展，这一理念在以后有可能会落地。

1.3.4 服务的部署

一个简单的单体系统可能只需要将程序集群部署并配置负载均衡服务器即可，而部署一个复杂的微服务架构的系统就复杂得多。因为每一个微服务可能还涉及比较底层的组件，例如数据库、消息中间件等。微服务系统往往由数量众多的服务构成，例如 Netflix 公司有大约 600



▲图 1-11　微服务、Docker、DevOps 之间的关系

个服务，而每个服务又有大量的实例。微服务系统需要对每个服务进行治理、监控和管理等，而每个服务有大量的配置，还需要考虑服务的启动顺序和启动时机等。

部署微服务系统，需要开发人员或者运维人员对微服务系统有足够强的控制力。随着云计算和云服务器的发展，部署微服务系统并不是一件难事，例如使用 PaaS 服务、使用 Docker 编排等。这就是人们往往提到微服务，就会想到 Docker、DevOps 的原因。其中，微服务是核心；Docker 为容器技术，是微服务最佳部署的容器；DevOps 是一种部署手段或理念。它们的关系如图 1-11 所示。

1.4 微服务和 SOA 的关系

SOA 即面向服务的架构，这种架构在 20 年前就已经被提出了。SOA 往往与企业服务总线（ESB）联系在一起，主要原因在于 SOA 的实施思路是根据 ESB 模式来整合集成大量单一庞大的系统，这是 SOA 主要的落地方式。然而，SOA 在过去 20 年并没有取得成功。在谈到微服务时，人们很容易联想到它是一个面向服务的架构。的确，微服务的概念提出者 Martin Fowler 没有否认这一层关系。

微服务相对于和 ESB 联系在一起的 SOA 显然轻便敏捷得多，微服务将复杂的业务组件化，实际也是一种面向服务思想的体现。对于微服务来说，它是 SOA 的一种实现，但是它比 ESB 实现的 SOA 更加轻便、敏捷和简单。

1.5 微服务的设计原则

软件设计就好比建筑设计。Architect 这个词在建筑学中是“建筑师”的意思，而在软件领域里则是“架构师”的意思，可见它们确实有相似之处。无论是建筑师还是架构师，他们都希望把作品设计出自己的特色，并且更愿意把创造出的东西被称为艺术品。然而现实却是，建筑设计和软件设计有非常大的区别。建筑师设计并建造出来的建筑往往很难有变化，除非拆了重建。而架构师设计出来的软件系统，为了满足产品的业务发展，在它的整个生命周期中，每一个版本都有很多的变化。

软件设计每一个版本都在变化，所以软件设计应该是渐进式发展。软件从一开始就不应该被设计成微服务架构，微服务架构固然有优势，但是它需要更多的资源，包括服务器资源、技术人员等。追求大公司所带来的技术解决方案，刻意地追求某个新技术，企图使用技术解决所有的问题，这些都是软件设计的误区。

技术应该是随着业务的发展而发展的，任何脱离业务的技术是不能产生价值的。在初创公司，业务很单一时，如果在 LAMP 单体构架够用的情况下，就应该用 LAMP，因为它开发速

度快，性价比高。随着业务的发展，用户量的增加，可以考虑将数据库读写分离、加缓存、加复杂均衡服务器、将应用程序集群化部署等。如果业务还在不断发展，这时可以考虑使用分布式系统，例如微服务架构的系统。不管使用什么样的架构，驱动架构的发展一定是业务的发展，只有当前架构不再适合当前业务的发展，才考虑更换架构。

在微服务架构中，有三大难题，那就是服务故障的传播性、服务的划分和分布式事务。在微服务设计时，一定要考虑清楚这三个难题，从而选择合适的框架。目前比较流行的微服务框架有 Spring 社区的 Spring Cloud、Google 公司的 Kubernetes 等。不管使用哪一种框架或者工具，都需要考虑这三大难题。为了解决服务故障的传播性，一般的微服务框架都有熔断机制组件。另外，服务的划分没有具体的划分方法，一般来说根据业务来划分服务，领域驱动设计具有指导作用。最后，分布式事务一般的解决办法就是两阶段提交或者三阶段提交，不管使用哪一种都存在事务失败，导致数据不一致的情况，关键时刻还得人工去恢复数据。总之，微服务的设计一定是渐进式的，并且是随着业务的发展而发展的。

第 2 章　Spring Cloud 简介

Spring Cloud 作为 Java 语言的微服务框架，它依赖于 Spring Boot，有快速开发、持续交付和容易部署等特点。Spring Cloud 的组件非常多，涉及微服务的方方面面，并在开源社区 Spring 和 Netflix、Pivotal 两大公司的推动下越来越完善。本章主要介绍 Spring Cloud，将从以下方面来讲解。

- 微服务应该具备的功能。
- Spring Cloud 介绍。
- Dubbo 介绍。
- Kubernetes 介绍。
- Spring Cloud 与 Dubbo 比较。
- Spring Cloud 与 Kubernetes 比较。

2.1 微服务应该具备的功能

微服务，可以拆分为“微”和“服务”二字。“微”即小的意思，那到底多小才算“微”呢？可能不同的团队有不同的答案。从参与微服务的人数来讲，单个微服务从架构设计、代码开发、测试、运维的人数加起来是 8~10 人才算“微”。那么何为“服务”呢？按照“微服务”概念提出者 Martin Fowler 给出的定义：“服务”是一个独立运行的单元组件，每个单元组件运行在独立的进程中，组件与组件之间通常使用 HTTP 这种轻量级的通信机制进行通信。

微服务具有以下的特点。

- 按照业务来划分服务，单个服务代码量小，业务单一，易于维护。
- 每个微服务都有自己独立的基础组件，例如数据库、缓存等，且运行在独立的进程中。
- 微服务之间的通信是通过 HTTP 协议或者消息组件，且具有容错能力。
- 微服务有一套服务治理的解决方案，服务之间不耦合，可以随时加入和剔除服务。
- 单个微服务能够集群化部署，并且有负载均衡的能力。
- 整个微服务系统应该有一个完整的安全机制，包括用户验证、权限验证、资源保护等。
- 整个微服务系统有链路追踪的能力。
- 有一套完整的实时日志系统。

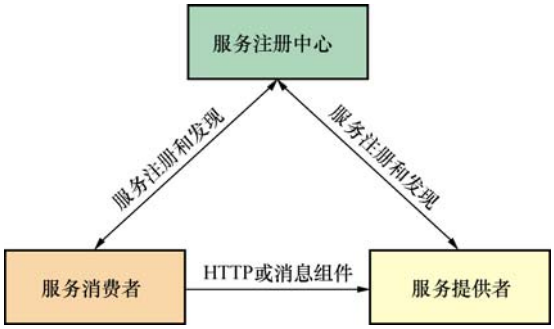
微服务具有以上这些特点，那么微服务需要具备一些什么样的功能呢？微服务的功能主要体现在以下几个方面。

- 服务的注册和发现。
- 服务的负载均衡。
- 服务的容错。
- 服务网关。
- 服务配置的统一管理。
- 链路追踪。
- 实时日志。

2.1.1 服务的注册与发现

微服务系统由很多个单一职责的服务单元组成，例如 Netflix 公司的系统是由 600 多个微服务构成的，而每一个微服务又有众多实例。由于该系统的服务粒度较小，服务数量众多，服务之间的相互依赖成网状，所以该系统需要服务注册中心来统一管理微服务实例，方便查看每一个微服务实例的健康状态。

服务注册是指向服务注册中心注册一个服务实例，服务提供者将自己的服务信息（如服务名、IP 地址等）告知服务注册中心。服务发现是指当服务消费者需要消费另外一个服务时，服务注册中心能够告知服务消费者它所要消费服务的实例信息（如服务名、IP 地址等）。通常情况下，一个服务既是服务提供者，也是服务消费者。服务消费者一般使用 HTTP 协议或者消息组件这种轻量级的通信机制来进行服务消费。服务的注册与发现如图 2-1 所示。



▲图 2-1 服务的治理—服务的注册和发现

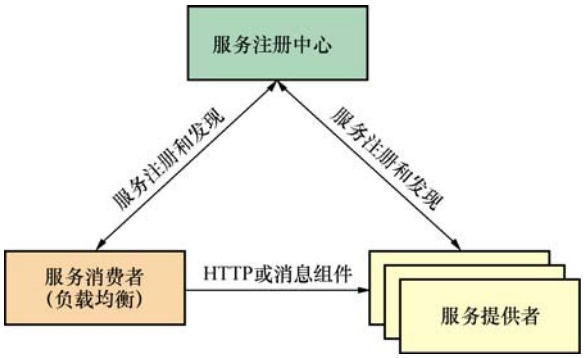
服务注册中心会提供服务的健康检查方案，检查被注册的服务是否可用。通常一个服务实例注册后，会定时向服务注册中心提供“心跳”，以表明自己还处于可用的状态。当一个服务实例停止向服务注册中心提供心跳一段时间后，服务注册中心会认为该服务实例不可用，会将该服务实例从服务注册列表中剔除。如果这个被剔除掉的服务实例过一段时间后继续向注册中心提供心跳，那么服务注册中心会将该服务实例重新加入服务注册中心的列表中。另外，微服务的服务注册组件都会提供服务的健康状况查看的 UI 界面，开发人员或者运维人员只需要登录相关的界面就可以知道服务的健康状态。

2.1.2 服务的负载均衡

在微服务架构中，服务之间的相互调用一般是通过 HTTP 通信协议来实现的。网络往往具

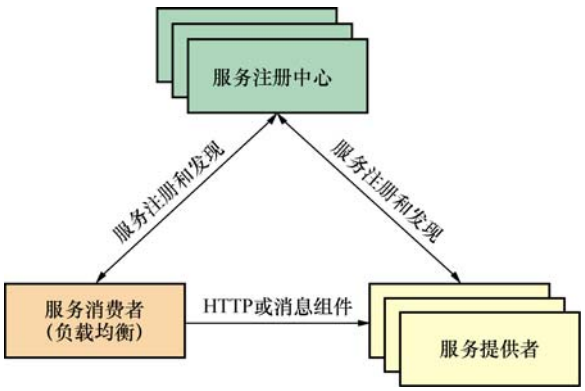
有不可靠性，为了保证服务的高可用（High Availability），服务单元往往是集群化部署。例如将服务提供者进行集群化部署，那么服务消费者该调用哪个服务提供者的实例呢？这就涉及到了服务的负载均衡。

服务的负载均衡一般最流行的做法如图 2-2 所示，所有的服务都向服务注册中心注册，服务注册中心持有每个服务的应用名和 IP 地址等信息，同时每个服务也会获取所有服务注册列表信息。服务消费者集成负载均衡组件，该组件会向服务消费者获取服务注册列表信息，并每隔一段时间重新刷新获取该列表。当服务消费者消费服务时，负载均衡组件获取服务提供者所有实例的注册信息，并通过一定的负载均衡策略（开发者可以配置），选择一个服务提供者的实例，向该实例进行服务消费，这样就实现了负载均衡。



▲图 2-2　服务的负载均衡

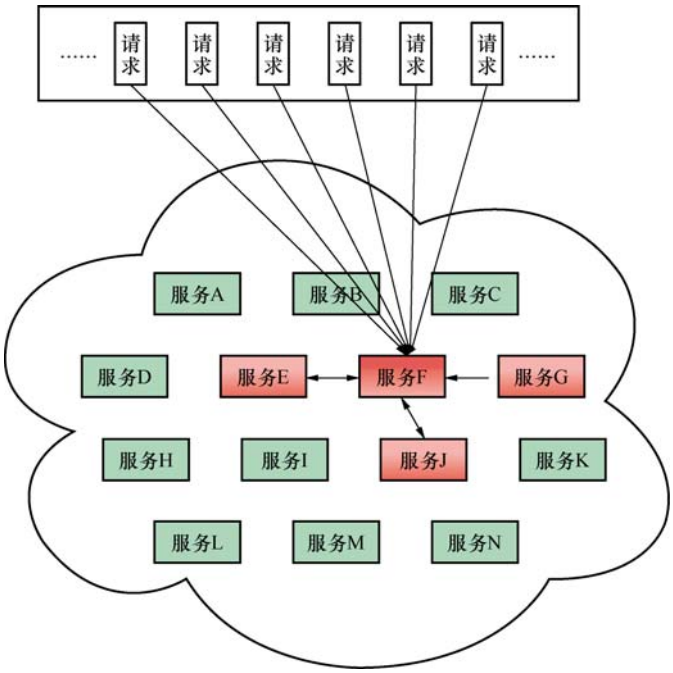
服务注册中心不但需要定时接收每个服务的心跳（用来检查服务是否可用），而且每个服务会定期获取服务注册列表的信息，当服务实例数量很多时，服务注册中心承担了非常大的负载。由于服务注册中心在微服务系统中起到了至关重要的作用，所以必须实现高可用。一般的做法是将服务注册中心集群化，每个服务注册中心的数据实时同步，如图 2-3 所示。



▲图 2-3　将服务注册中心高可用

2.1.3 服务的容错

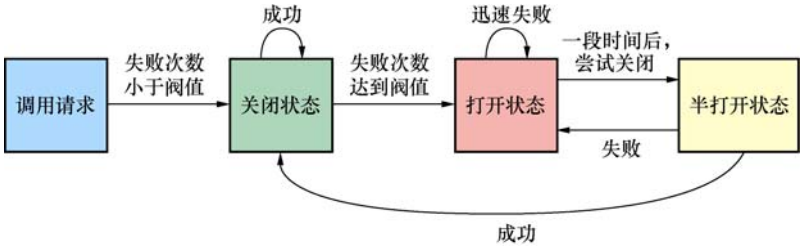
微服务落地到实际项目中，服务的数量往往非常多，服务之间的相互依赖性也是错综复杂的，一个网络请求通常需要调用多个服务才能完成。如果一个服务不可用，例如网络延迟或故障，会影响到依赖于这个不可用的服务的其他服务。如图 2-4 所示，一个微服务系统有很多个服务，当服务 F 因某些原因导致了服务的不可用，来自于用户的网络请求需要调用服务 F。由于服务 F 无响应，用户的请求都处于阻塞状态，在高并发的场景下，短时间内会导致服务器的线程资源消耗殆尽。另外，依赖于服务 F 的其他的服务，例如图中的服务 E、服务 G、服务 J，也会等待服务 F 的响应，处于阻塞状态，导致这些服务的线程资源消耗殆尽，进而导致它们的不可用，以及依赖于它们的服务的不可用，最后导致整个系统处于瘫痪的状态也就是 1.2.1 节中提到的雪崩效应。



▲图 2-4 服务的依赖性

为了解决分布式系统的雪崩效应，分布式系统引进了熔断器机制。熔断器（Circuit Breaker）一词来源于物理学中的电路知识，它的作用是当电路中出现故障时迅速切断电路，起到保护电路的作用，熔断器机制如图 2-5 所示。当一个服务的处理用户请求的失败次数在一定时间内小于设定的阀值时，熔断器处于关闭状态，服务正常；当服务处理用户请求的失败次数大于设定的阀值时，说明服务出现了故障，打开熔断器，这时所有的请求会执行快速失败，不执行业务逻辑。当处于打开状态的熔断器时，一段时间后会处于半打开状态，并执行一定数量的请求，剩余的请求会执行快速失败，若执行的请求失败了，则继续打开熔断器；若成功了，则将熔断

器关闭。



▲图 2-5　熔断器机制

这种机制有着非常重要的意义，它不仅能够防止系统的“雪崩”效应，还具有以下作用。

- 将资源进行隔离，如果某个服务里的某个 API 接口出现了故障，只会隔离该 API 接口。不会影响到其他 API 接口。被隔离的 API 接口会执行快速失败的逻辑，不会等待，请求不会阻塞。如果不进行这种隔离，请求会一直处于阻塞状态，直到超时。若有大量的请求同时涌入，都处于阻塞的状态，服务器的线程资源，迅速被消耗完。
- 服务降级的功能。当服务处于正常的状态时，大量的请求在短时间内同时涌入，超过了服务的处理能力，这时熔断器会被打开，将服务降级，以免服务器因负载过高而出现故障。
- 自我修复能力。当因某个微小的故障，例如网络服务商的问题，导致网络在短时间内不可用，熔断器被打开。如果不能自我监控、自我检测和自我修复，那么需要开发人员手动地去关闭熔断器，无疑会增加开发人员的工作量。

Netflix 的 Hystrix 熔断器开源组件功能非常强大，不仅有熔断器的功能，还有熔断器的状态监测，并提供界面友好的 UI，开发人员或者运维人员通过 UI 界面能够直观地看到熔断器的状态和各种性能指标。

2.1.4　服务网关

微服务系统通过将资源以 API 接口的形式暴露给外界来提供服务。在微服务系统中，API 接口资源通常是由服务网关（也称 API 网关）统一暴露，内部服务不直接对外提供 API 资源的暴露。这样做的好处是将内部服务隐藏起来，外界还以为是一个服务在提供服务，在一定程度上保护了微服务系统的安全。API 网关通常有请求转发的作用，另外它可能需要负责一定的安全验证，例如判断某个请求是否合法，该请求对某一个资源是否具有操作权限等。通常情况下，网关层以集群的形式存在。在服务网关层之前，有可能需要加上负载均衡层，通常为 Nginx 双机热备，通过一定的路由策略，将请求转发到网关层。到达网关层后，经过一系列的用户身份验证、权限判断，最终转发到具体的服务。具体的服务经过一系列的逻辑运算和数据操作，最终将结果返回给用户，此时的架构如图 2-6 所示。

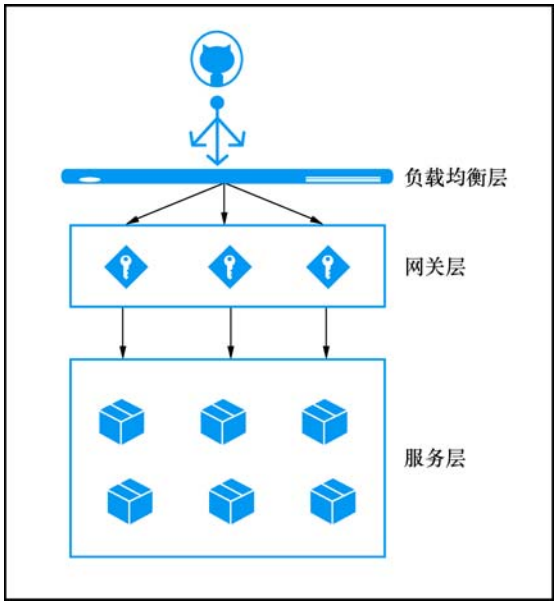
网关层具有很重要的意义，具体体现在以下方面。

- 网关将所有服务的 API 接口资源统一聚合，对外统一暴露，外界系统调用的 API 接

口都是网关对外暴露的 API 接口。外界系统不需要知道微服务架构中各服务相互调用的复杂性，微服务系统也保护了其内部微服务单元的 API 接口，防止被外界直接调用以及服务的敏感信息对外暴露。

- 网关可以做一些用户身份认证、权限认证，防止非法请求操作 API 接口，对内部服务起到保护作用。
- 网关可以实现监控功能，实时日志输出，对请求进行记录。
- 网关可以用来做流量监控，在高流量的情况下，对服务进行降级。
- API 接口从内部服务分离出来，方便做测试。

当然，网关实现这些功能，需要做高可用，否则网关很可能成为架构中的瓶颈。最常用的网关组件有 Zuul、Nginx 等。



▲图 2-6 服务网关架构图

2.1.5 服务配置的统一管理

在实际开发过程中，每个服务都有大量的配置文件，例如数据库的配置、日志输出级别的配置等，而往往这些配置在不同的环境中也是不一样的。随着服务数量的增加，配置文件的管理也是一件非常复杂的事。

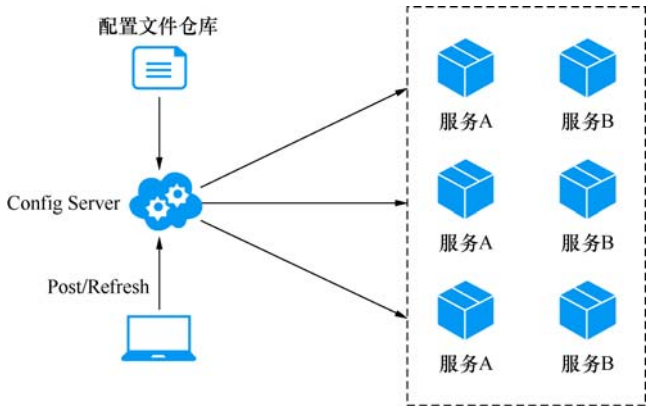
在微服务架构中，需要有统一管理配置文件的组件，例如 Spring Cloud 的 Spring Cloud Config 组件、阿里的 Diamond、百度的 Disconf、携程的 Apollo 等。这些配置组件所实现的功能大体相同，但是又有些差别，下面以 Spring Cloud Config 为例来阐述服务配置的统一管理。

如图 2-7 所示，大致过程如下。

- 首先，Config Server（配置服务）读取配置文件仓库的配置信息，其中配置文件仓库可以

存放在配置服务的本地仓库，也可以放在远程的 Git 仓库（例如 GitHub、Coding 等）。

- 配置服务启动后，读取配置文件信息，读取完成的配置信息存放在配置服务的内存中。
- 当启动服务 A、B 时，由于服务 A、B 指定了向配置服务读取配置信息，服务 A、B 向配置服务读取配置信息。
- 当服务的配置信息需要修改且修改完成后，向配置服务发送 Post 请求进行刷新，这时服务 A、B 会向配置服务重写读取配置文件。



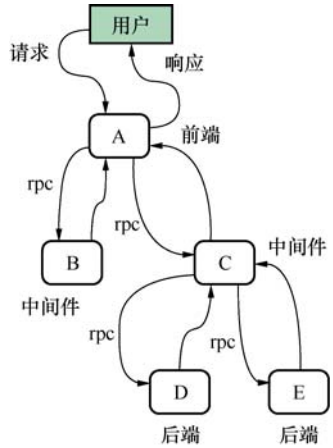
▲图 2-7　服务配置统一管理

对于集群化的服务，可以通过使用消息总线来刷新多个服务实例。如果服务数量较多，对配置中心需要考虑集群化部署，从而使配置中心高可用，做分布式集群。

2.1.6　服务链路追踪

微服务系统是一个分布式架构的系统，微服务系统按业务划分服务单元，一个微服务系统往往有很多个服务单元。由于服务单元数量很多且业务复杂，服务与服务之间的调用有可能非常复杂，一旦出现了异常和错误，就会很难去定位。所以在微服务架构中，必须实现分布式链路追踪，去跟进一个请求到底有哪些服务参与，参与的顺序又是怎样的，从而使每个请求链路清晰可见，出了问题很快就能定位。

举个例子，如图 2-8 所示，在微服务系统中，一个来自用户的请求先达到前端 A（如前端界面），然后通过远程调用，达到系统的中间件 B、C（如负载均衡、网关等），最后达到后端服务 D、E。后端经过一系列的业务逻辑计算，最后将数据返回给用户。对于这样一个请求，经历了这么多服务，怎么样将它的请求过程的数据记录下来呢？这就需要用到服务链路追踪。



▲图 2-8　请求通过 A、B、C、D、E

Google 开源了链路追踪组件 Dapper，并在 2010 年发表了

论文《Dapper, a Large-Scale Distributed Systems Tracing Infrastructure》，这篇文章是业内实现链路追踪的标杆和理论基础，具有非常高的参考价值。

目前，常见的链路追踪组件有 Google 的 Dapper、Twitter 的 Zipkin，以及阿里的 Eagleeye（鹰眼）等，都是非常优秀的链路追踪开源组件。

2.2 Spring Cloud

2.2.1 简介

Spring Cloud 是基于 Spring Boot 的。Spring Boot 是由 Pivotal 团队提供的全新 Web 框架，它主要的特点就是简化了开发和部署的过程，简化了 Spring 复杂的配置和依赖管理，通过起步依赖和内置 Servlet 容器能够使开发者迅速搭起一个 Web 工程。所以 Spring Cloud 在开发部署上继承了 Spring Boot 的一些优点，提高其在开发和部署上的效率。

Spring Cloud 的首要目标就是通过提供一系列开发组件和框架，帮助开发者迅速搭建一个分布式的微服务系统。Spring Cloud 是通过包装其他技术框架来实现的，例如包装开源的 Netflix OSS 组件，实现了一套通过基于注解、Java 配置和基于模版开发的微服务框架。Spring Cloud 框架来自于 Spring Resouces 社区，由 Pivotal 和 Netflix 两大公司和一些其他的开发者提供技术上的更新迭代。Spring Cloud 提供了开发分布式微服务系统的一些常用组件，例如服务注册和发现、配置中心、熔断器、智能路由、微代理、控制总线、全局锁、分布式会话等。

2.2.2 常用组件

（1）服务注册和发现组件 Eureka

利用 Eureka 组件可以很轻松地实现服务的注册和发现的功能。Eureka 组件提供了服务的健康监测，以及界面友好的 UI。通过 Eureka 组件提供的 UI，Eureka 组件可以让开发人员随时了解服务单元的运行情况。另外 Spring Cloud 也支持 Consul 和 Zookeeper，用于注册和发现服务。

（2）熔断组件 Hystrix

Hystrix 是一个熔断组件，它除了有一些基本的熔断器功能外，还能够实现服务降级、服务限流的功能。另外 Hystrix 提供了熔断器的健康监测，以及熔断器健康数据的 API 接口。Hystrix Dashboard 组件提供了单个服务熔断器的健康状态数据的界面展示功能，Hystrix Turbine 组件提供了多个服务的熔断器的健康状态数据的界面展示功能。

（3）负载均衡组件 Ribbon

Ribbon 是一个负载均衡组件，它通常和 Eureka、Zuul、RestTemplate、Feign 配合使用。Ribbon 和 Zuul 配合，很容易做到负载均衡，将请求根据负载均衡策略分配到不同的服务实例中。Ribbon 和 RestTemplate、Feign 配合，在消费服务时能够做到负载均衡。

（4）路由网关 Zuul

路由网关 Zuul 有智能路由和过滤的功能。内部服务的 API 接口通过 Zuul 网关统一对外暴露，内部服务的 API 接口不直接暴露，防止了内部服务敏感信息对外暴露。在默认的情况下，Zuul 和

Ribbon 相结合，能够做到负载均衡、智能路由。Zuul 的过滤功能是通过拦截请求来实现的，可以对一些用户的角色和权限进行判断，起到安全验证的作用，同时也可以用于输出实时的请求日志。

上述的 4 个组件都来自于 Netflix 的公司，统一称为 Spring Cloud Netflix。

（5）Spring Cloud Config

Spring Cloud Config 组件提供了配置文件统一管理的功能。Spring Cloud Config 包括 Server 端和 Client 端，Server 端读取本地仓库或者远程仓库的配置文件，所有的 Client 向 Server 读取配置信息，从而达到配置文件统一管理的目的。通常情况下，Spring Cloud Config 和 Spring Cloud Bus 相互配合刷新指定 Client 或所有 Client 的配置文件。

（6）Spring Cloud Security

Spring Cloud Security 是对 Spring Security 组件的封装，Spring Cloud Security 向服务单元提供了用户验证和权限认证。一般来说，单独在微服务系统中使用 Spring Cloud Security 是很少见的，一般它会配合 Spring Security OAuth2 组件一起使用，通过搭建授权服务，验证 Token 或者 JWT 这种形式对整个微服务系统进行安全验证。

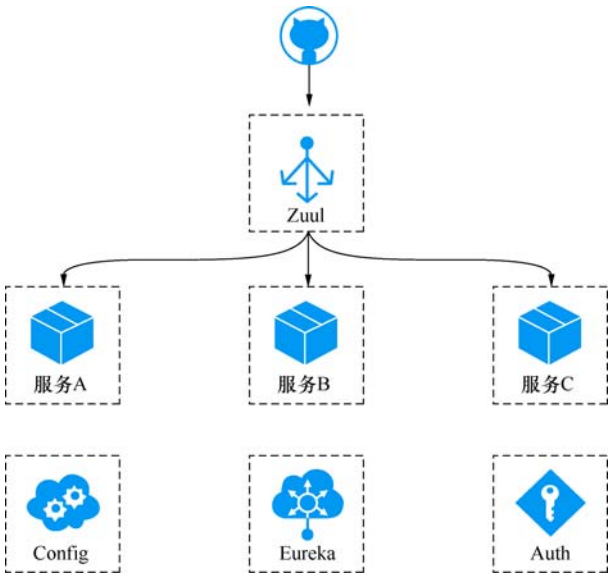
（7）Spring Cloud Sleuth

Spring Cloud Sleuth 是一个分布式链路追踪组件，它封装了 Dapper、Zipkin 和 Kibana 等组件，通过它可以知道服务之间的相互依赖关系，并实时观察链路的调用情况。

（8）Spring Cloud Stream

Spring Cloud Stream 是 Spring Cloud 框架的数据流操作包，可以封装 RabbitMq、ActiveMq、Kafka、Redis 等消息组件，利用 Spring Cloud Stream 可以实现消息的接收和发送。

上述列举了一些常用的 Spring Cloud 组件。一个简单的由 Spring Cloud 构建的微服务系统，通常由服务注册中心 Eureka、网关 Zuul、配置中心 Config 和授权服务 Auth 构成，架构如图 2-9 所示。



▲图 2-9　一个简单的由 Spring Cloud 构建的微服务系统

2.2.3 项目一览表

- Spring Cloud Config：服务配置中心，将所有的服务的配置文件放到本地仓库或者远程仓库，配置中心负责读取仓库的配置文件，其他服务向配置中心读取配置。Spring Cloud Config 使得服务的配置统一管理，并可以在不人为重启服务的情况下进行配置文件的刷新。
- Spring Cloud Netflix：它是通过包装了 Netflix 公司的微服务组件实现的，也是 Spring Cloud 核心的核心组件，包括 Eureka、Hystrix、Zuul、Archaius 等。
- Eureka：服务注册和发现组件。
- Hystrix：熔断器组件。Hystrix 通过控制服务的 API 接口的熔断来转移故障，防止微服务系统发生雪崩效应。另外，Hystrix 能够起到服务限流和服务降级的作用。使用 Hystrix Dashboard 组件监控单个服务的熔断器的状态，使用 Turbine 组件可以聚合多个服务的熔断器的状态。
- Zuul：智能路由网关组件。Netflix Zuul 能够起到智能路由和请求过滤的作用，是服务接口统一暴露的关键模块，也是安全验证、权限控制的一道门。
- Feign：声明式远程调度组件。
- Ribbon：负载均衡组件。
- Archaius：配置管理 API 的组件，一个基于 Java 的配置管理库，主要用于多配置的动态获取。
- Spring Cloud Bus：消息总线组件，常和 Spring Cloud Config 配合使用，用于动态刷新服务的配置。
- Spring Cloud Sleuth：服务链路追踪组件，封装了 Dapper、Zipkin、Kibina 等组件，可以实时监控服务的链路调用情况。
- Spring Cloud Data Flow：大数据操作组件，Spring Cloud Data Flow 是 Spring XD 的替代品，也是一个混合计算的模型，可以通过命令行的方式操作数据流。
- Spring Cloud Security：安全模块组件，是对 Spring Security 的封装，通常配合 OAuth2 使用来保护微服务系统的安全。
- Spring Cloud Consul：该组件是 Spring Cloud 对 Consul 的封装，和 Eureka 类似，它是另一个服务注册和发现组件。
- Spring Cloud Zookeeper：该组件是 Spring Cloud 对 Zookeeper 的封装，和 Eureka、Consul 类似，用于服务的注册和发现。
- Spring Cloud Stream：数据流操作组件，可以封装 Redis、RabbitMQ、Kafka 等组件，实现发送和接收消息等。
- Spring Cloud CLI：该组件是 Spring Cloud 对 Spring Boot CLI 的封装，可以让用户以命令行方式快速运行和搭建容器。
- Spring Cloud Task：该组件基于 Spring Task，提供了任务调度和任务管理的功能。

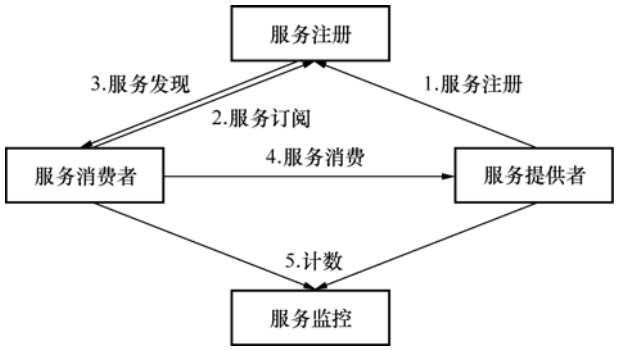
- Spring Cloud Connectors：用于 Paas 云平台连接到后端。

2.3 Dubbo 简介

Dubbo 是阿里巴巴开源的一个分布式服务框架，致力于提供高性能和透明化的 RPC 远程服务调用方案，以及 SOA 服务治理方案。Dubbo 广泛用于阿里巴巴的各大站点，有很多互联网公司也在使用这个框架，它包含如下核心内容。

- RPC 远程调用：封装了长连接 NIO 框架，如 Netty、Mina 等，采用的是多线程模式。
- 集群容错：提供了基于接口方法的远程调用的功能，并实现了负载均衡策略、失败容错等功能。
- 服务发现：集成了 Apache 的 Zookeeper 组件，用于服务的注册和发现。

Dubbo 框架的架构图如图 2-10 所示。



▲图 2-10　Dubbo 架构图

Dubbo 架构的流程如下。

（1）服务提供者向服务中心注册服务。

（2）服务消费者订阅服务。

（3）服务消费者发现服务。

（4）服务消费者远程调度服务提供者进行服务消费，在调度过程中，使用了负载均衡策略、失败容错的功能。

（5）服务消费者和提供者，在内存中记录服务的调用次数和调用时间，并定时每分钟发送一次统计数据到监控中心。

Dubbo 是一个非常优秀的服务治理框架，在国内互联网公司应用广泛。它具有以下特性。

- 连通性：注册中心负责服务的注册；监控中心负责收集调用次数、调用时间；注册中心、服务提供者、服务消费者为长连接。
- 健壮性：监控中心宕机不影响其他服务的使用；注册中心集群，任意一个实例宕机自动切换到另一个注册中心实例；服务实例集群，任意一个实例宕机，自动切换到另外

一个可用的实例。

- 伸缩性：可以动态增减注册中心和服务的实例数量。
- 升级性：服务集群升级，不会对现有架构造成压力。

2.4 Spring Cloud 与 Dubbo 比较

首先从微服务关注点来比较 Spring Cloud 和 Dubbo 两大服务框架，如表 2-1 所示。

表 2-1 从微服务关注点比较 Spring Cloud 和 Dubbo

微服务关注点	Spring Cloud	Dubbo
配置管理	Config	—
服务发现	Eureka、Consul、Zookeeper	Zookeeper
负载均衡	Ribbon	自带
网关	Zuul	—
分布式追踪	Spring Cloud Sleuth	—
容错	Hystrix	不完善
通信方式	HTTP、Message	RPC
安全模块	Spring Cloud Security	—

Spring Cloud 拥有很多的项目模块，包含了微服务系统的方方面面。Dubbo 是一个非常优秀的服务治理和服务调用框架，但缺少很多功能模块，例如网关、链路追踪等。在项目模块上，Spring Cloud 占据着更大的优势。

Spring Cloud 的更新速度非常块，Camden.SR5 版本发布于 2017 年 2 月 6 日，Camden.SR6 版本发布于 2017 年 3 月 10 日，Dalston 版本发布于 2017 年 4 月 12 日，基本每个月会发一次版本的迭代。从 GitHub 的代码仓库来看，Spring Cloud 几乎每天都有更新。阿里巴巴于 2011 年 10 月开源了 Dubbo，开源后的 Dubbo 发展迅速，大概每 2~3 个月有一次版本更新。然而，从在 2013 年 3 月开始，Dubbo 暂停了版本更新，并只在 2014 年 10 月发布了一个小版本，修复了一个 bug，之后长期处于版本停止更新的状态。直到 2017 年 9 月，阿里巴巴中间件部门重新组建了 Dubbo 团队，把 Dubbo 列为重点开源项目，并在 2017 年 9～11 月期间，一直保持每月一次版本更新的频率。

从学习成本上考虑，Dubbo 的版本趋于稳定，文档完善，可以即学即用，没有太大难度。Spring Cloud 基于 Spring Boot 开发，需要开发者先学会 Spring Boot。另外，Spring Cloud 版本迭代快，需要快速跟进学习。Spring Cloud 文档大多是英文的，要求学习者有一定的英文阅读能力。此外，Spring Cloud 文档很多，不容易快速找到相应的文档。

从开发风格上来讲，Dubbo 更倾向于 Spring Xml 的配置方式，Dubbo 官方也推荐这种方式。Spring Cloud 基于 Spring Boot，Spring Boot 采用的是基于注解和 JavaBean 配置方式的敏捷开发。

从开发速度上讲，Spring Cloud 具有更高的开发和部署速度。

最后，Spring Cloud 的通信方式大多数是基于 HTTP Restful 风格的，服务与服务之间完全无关、无耦合。由于采用的是 HTTP Rest，因此服务无关乎语言和平台，只需要提供相应 API 接口，就可以相互调用。Dubbo 的通信方式基于远程调用，对接口、平台和语言有强依赖性。如果需要实现跨平台调用服务，需要写额外的中间件，这也是 Dubbo 存在的原因。

Dubbo 和 Spring Cloud 拥有各自的优缺点。Dubbo 更易上手，并且广泛使用于阿里巴巴的各大站点，经历了“双 11”期间高并发、大流量的检验，Dubbo 框架非常成熟和稳定。Spring Cloud 服务框架严格遵守 Martin Fowler 提出的微服务规范，社区异常活跃，它很可能成为微服务架构的标准。

2.5 Kubernetes 简介

Kubernetes 是一个容器集群管理系统，为容器化的应用程序提供部署运行、维护、扩展、资源调度、服务发现等功能。

Kubernetes 是 Google 运行 Borg 大规模系统达 15 年之久的一个经验总结。Kubernetes 结合了社区的最佳创意和实践，旨在帮助开发人员将容器打包、动态编排，同时帮助各大公司向微服务方向进行技术演进。

它具有以下特点。

- Planet Scale（大容量）：使用 Kubernetes 的各大公司（包括 Google）每周运行了数十亿个容器，这些容器的平台采用同样的设计原则。这些平台在不增加 DevOps 团队成员的情况下，可以让容器数量增加，节省了人力成本，达到了复用性。
- Never Outgrow（永不过时）：无论容器是运行在一个小公司的测试环境中，还是运行在一个全球化企业的大型系统里，Kubernetes 都能灵活地满足复杂的需求。同时，无论业务多么复杂，Kubernetes 都能稳定地提供服务。
- Run Anywhere（随时随地运行）：Kubernetes 是开源的，可以自由地利用内部、混合或公共云的基础组件进行部署，让开发者可以将更多的时间和精力投入在业务上，而不是服务部署上。

Kubernetes 开源免费，是 Google 在过去 15 年时间里部署、管理微服务的经验结晶，所以目前 Kubernetes 在技术社区也是十分火热。下面来看它提供的功能。

- Automatic Binpacking（自动包装）：根据程序自身的资源需求和一些其他方面的需求自动配置容器。Kubernetes 能够最大化地利用机器的工作负载，提高资源的利用率。
- Self-healing（自我修复）：容器失败自动重启，当节点处于“死机”的状态时，它会被替代并重新编排；当容器达到用户设定的无响应的阀值时，它会被剔除，并且不让其他容器调用它，直到它恢复服务。
- Horizontal Scaling（横向扩展）：可以根据机器的 CPU 的使用率来调整容器的数量，

只需开发人员在管理界面上输入几个命令即可。

- Service Discovery and Load Balancing（服务发现和负载均衡）：在不需要修改现有的应用程序代码的情况下，便可使用服务的发现机制。Kubernetes 为容器提供了一个虚拟网络环境，每个容器拥有独立的 IP 地址和 DNS 名称，容器之间实现了负载均衡。
- Automated Rollouts and Rollbacks（自动部署或回滚）：Kubernetes 支撑滚动更新模式，能逐步替换掉当前环境的应用程序和配置，同时监视应用程序运行状况，以确保不会同时杀死所有实例。如果出现问题，Kubernetes 支持回滚更改。
- Secret and Configuration Management（配置管理）：部署和更新应用程序的配置，不需要重新打镜像，并且不需要在堆栈中暴露配置。
- Storage Orchestration（存储编排）：自动安装所选择的存储系统，无论是本地存储、公共云提供商（如 GCP 或 AWS），还是网络存储系统（如 NFS、iSCSI、Gluster、Ceph、Cinder 或 Flocker）。
- Batch execution（批量处理）：除了服务之外，Kubernetes 还可以管理批量处理和 CI 的工作负载，如果需要，可以替换容器，如 NFS、iSCSI、Gluster、Ceph、Cinder 或 Flocker 等。

从 Kubernetes 提供的功能来看，Kubernetes 完全可以成为构建和部署微服务的一个工具，它是从服务编排上实现的，而不是代码实现的。目前国外有很多知名的公司在使用 Kubernetes，如 Google、eBay、Pearson 等。由于它的开源免费，Microsoft、VMWare、Red Hat、CoreOS 等公司纷纷加入并贡献代码。Kubernetes 技术吸引了一大批公司和技术爱好者，它已经成为容器管理的领导者。

2.6 Spring Could 与 Kubernetes 比较

Spring Cloud 是一个构建微服务的框架，而 Kubernetes 是通过对运行的容器的编排来实现构建微服务的。两者从构建微服务的角度和实现方式有很大的不同，但它们提供了构建微服务所需的全部功能。从提供的微服务所需的功能上看，两者不分上下，如表 2-2 所示。

表 2-2 Spring Cloud 与 Kubernetes 比较

微服务关注点	Spring Cloud	Kubernetes
配置管理	Config	Kubernetes ConfigMap
服务发现	Eureka、Consul、Zookeeper	Kubernetes Services
负载均衡	Ribbon	Kubernetes Services
网关	Zuul	Kubernetes Services
分布式追踪	Spring Cloud Sleuth	Open tracing
容错	Hystrix	Kubernetes Health Check

续表

微服务关注点	Spring Cloud	Kubernetes
安全模块	Spring Cloud Security	—
分布式日志	ELK	EFK
任务管理	Spring Batch	Kubernetes Jobs

Spring Cloud 通过众多的类库来实现微服务系统所需的各个组件，同时不断集成优秀的组件，所以 Spring Cloud 组件是非常完善的。Spring Cloud 基于 Spring Boot 框架，有快速开发、快速部署的优点。对于 Java 开发者来说，学习 Spring Cloud 的成本不高。

Kubernetes 在编排上解决微服务的各个功能，例如服务发现、配置管理、负载均衡、容错等。Kubernetes 不局限于 Java 平台，也不局限于语言，开发者可以自由选择开发语言进行项目开发。

与 Kubernetes 相比，Spring Cloud 具有以下优点。

- 采用 Java 语言开发，基于 Spring 平台，继承了 Spring Boot 快速开发的优势，是 Java 程序员实现微服务的最佳实践。
- Spring Cloud 有大量的类库和资源，基本上能解决所有可能出现的问题。

与 Kubernetes 比较，Spring Cloud 具有以下缺点。

- 依赖于 Java 语言，不支持跨语言。
- Spring Cloud 需要在代码中关注微服务的功能点，例如服务发现、负载均衡等。Kubernetes 则不需要关注这些。

下面介绍 Kubernetes 的优点和缺点，优点如下。

- Kubernetes 支持多种语言，并且是一个容器管理平台。Kubernetes 使程序容器化，并在容器管理上提供了微服务的功能，例如配置管理、服务发现、负载均衡等。Kubernetes 能够被应用于多种场合，例如程序开发、测试环境、创建环境等。
- Kubernetes 除了提供基本的构建微服务的功能外，还提供了环境、资源限制、管理应用程序的生命周期的功能。Kubernetes 更像是一个平台，而 Spring Cloud 是一个框架。

Kubernetes 的缺点如下。

- Kubernetes 面向 DevOps 人员，普通的开发人员需要学习很多这方面的知识，学习成本非常高。
- Kubernetes 仍然是一个相对较新的平台，发展十分迅速。新特性更新得快，所以需要 DevOps 人员跟进，不断地学习。

Spring Cloud 尝试从 Java 类库来实现微服务的所有功能，而 Kubernetes 尝试从容器编排上实现所有的微服务功能，两者的实现角度和方式不一样。个人觉得，两者最终的实现功能和效果上不分胜负，但从实现的方式上来讲，Kubernetes 略胜一筹。Kubernetes 面向 DevOps 人员，学习成本高。Spring Cloud 有很多的类库，以 Spring 为基础，继承了 Spring Boot 快速开发的优点，为 Java 程序员开发微服务提供了很好的体验，学习成本也较低。所

以二者比较，各有优势。没有最好的框架，也没有最好的工具，关键是要适合业务需求和满足业务场景。

2.7 总结

本章首先介绍了微服务应该具备的功能，然后介绍了 Spring Cloud 和 Spring Cloud 的基本组件，最后介绍了 Spring Cloud 与 Dubbo、Kubernetes 之间的比较，以及它们的优缺点。Spring Cloud 作为 Java 语言的微服务落地框架，有很多的微服务组件。为了循序渐进地学习这些组件，第 3、4 章将介绍构建微服务前的准备工作，这是学习 Spring Cloud 组件的基本前提。

第 3 章　构建微服务的准备

子曰：“工欲善其事，必先利其器”。说的是做好一件事，准备工作是非常重要的。本章和下一章主要介绍构建微服务前的准备工作，本章介绍开发环境的搭建，下一章讲解开发框架 Spring Boot 的入门。搭建的环境包括 JDK 的安装、开发工具的安装，以及项目的构建工具。常见的开发 Spring Cloud 项目的工具包括 MyEclipse、IntellJ Idea（简称 IDEA），强烈推荐使用 IDEA 作为开发工具。IDEA 和 Spring Boot 一起使用，个人认为是开发 Java 程序的最佳体验。本书的案例代码都是在 IDEA 上开发的，所以本章介绍的开发工具也是 IDEA。项目的构建工具包括 Apache Maven 和 Gradle，Gradle 是一个基于 Apache Ant 和 Apache Maven 概念的项目自动化构建的工具。两个构建工具都非常方便，按个人习惯来选择，Apache Maven 的使用率要高一些，所以选择介绍的构建工具为 Apache Maven。

3.1 JDK 的安装

3.1.1 JDK 的下载和安装

由于 Spring Boot 在未来的版本 2.0 中要求最低的 JDK 版本为 1.8，所以选择安装 JDK1.8。Mac 系统已经安装了 JDK，所以不需要用户自行安装。Windows 用户需要从 Oracle 官网下载安装包，下载完成后，解压安装。

3.1.2 环境变量的配置

JDK 安装完成后，需要设置环境变量。在 Windows 操作系统中，打开“我的电脑”→“属性”→“高级”→“环境变量”，新建系统变量“JAVA_HOME”，变量值为“JDK”的安装目录，例如我的安装目录为“D:\Program Files\Java\ jdk1.8.0_121”。选择“系统变量”中变量名为“Path”的环境变量，双击该变量，把 JDK 安装路径中 bin 目录的绝对路径追加到环境变量 Path 尾部，注意用“;”来分隔前面的变量。例如我在 Path 尾部追加为“;%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;”。

JDK 的安装工作已经全部完成，现在来验证 JDK 是否安装成功。打开命令行窗口，输入“java –version”，如果 JDK 安装成功且环境变量设置成功，命令行窗口会显示如下信息：

```
java version "1.8.0_121"
Java(TM) SE Runtime Environment (build 1.8.0_121-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)
```

3.2 IDEA 的安装

对于习惯了使用 Eclipse 或者 MyEclipse 的开发者来说，可能不愿意换新的 IDEA，因为需要花时间去学习，还要去适应新的开发工具。个人觉得，IDEA 比 Eclipse 系列好用很多，它带来了不一样的开发体验，主要体现在以下 5 个方面。

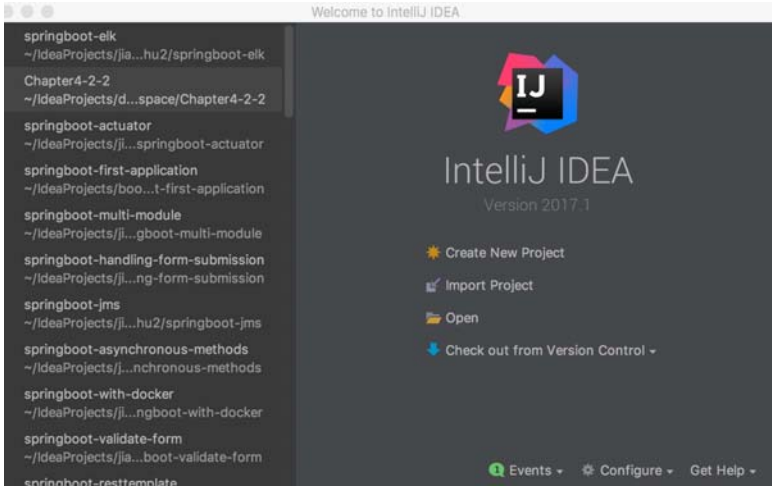
- 有对用户更加友好的界面，有更加护眼的黑色主题，感觉更高端大气。
- 比 Eclipse 更加智能，主要体现在代码的补全方面。
- 更加友好的代码提示功能。
- 内置 Maven、Gradle 等构建工具，并且下载依赖包非常智能和流畅。
- 更加强大的纠错能力。

虽然，Eclipse 和 IDEA 都能开发出 Java 项目，Eclipse 也非常好用，但两者的写代码体验不在一个级别上。IDEA 具有更友好的界面和更智能的代码提示，以及更强大的纠错能力，所以 IDEA 写代码体验更好、效率更高。建议读者用 IDEA 来开发 Java 项目，本书所有的代码都是用 IDEA 来写的。

3.2.1 IDEA 的下载

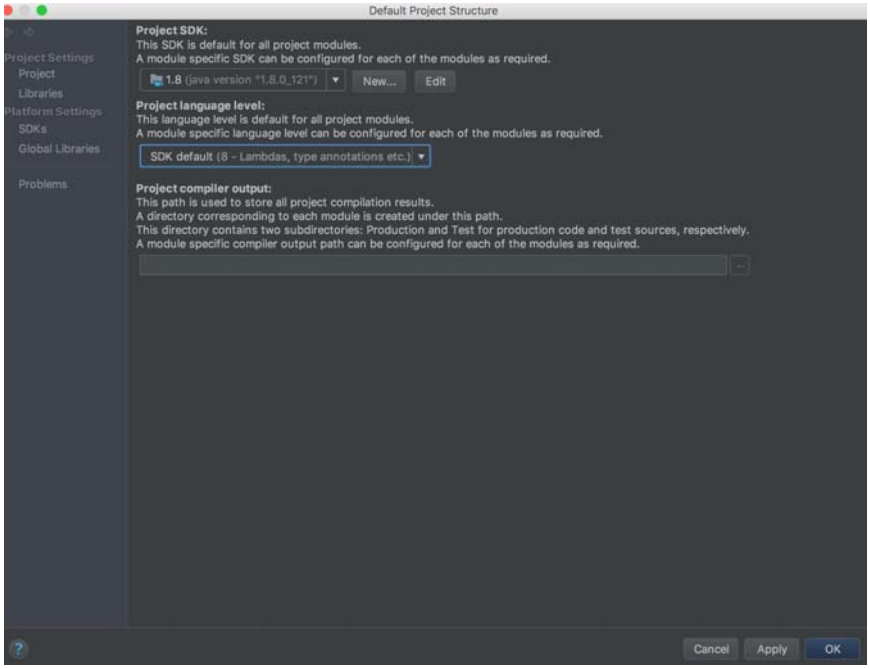
去官方网站 https://www.jetbrains.com/idea/下载 IDEA，IDEA 有免费版和商业版，免费版能做一些基本的 Java 开发，但是不能用来进行 J2EE 的开发，所以需要下载商业版。商业版对学生免费，你需要一个.edu 结尾的邮箱去申请获取免费版本，申请通过即可免费使用商业版本。

下载完成，按照提示的步骤安装即可。安装完成后，启动 IDEA，会进入如图 3-1 所示的界面。



▲图 3-1 IDEA 的启动界面

启动成功后，需要配置 JDK，单击图 3-1 右下方的“Configure”按钮，选择“Project Defaults”→“Project Structures”，进入配置界面，选择“Project SDK”为安装的 JDK1.8 即可，Project Language Level 为 SDK（default 8），如图 3-2 所示。



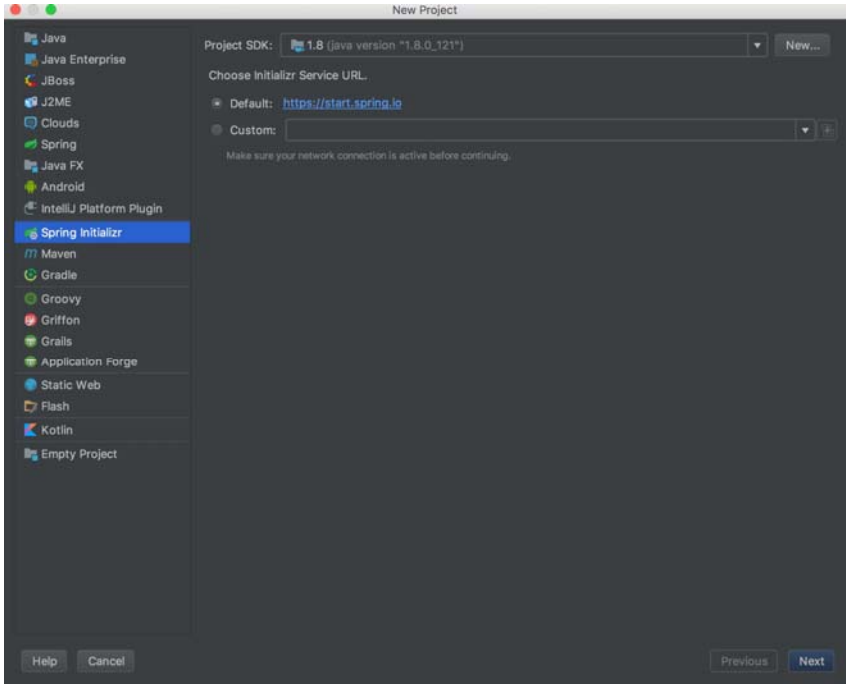
▲图 3-2　IDEA 配置 SDK

3.2.2　用 IDEA 创建一个 Spring Boot 工程

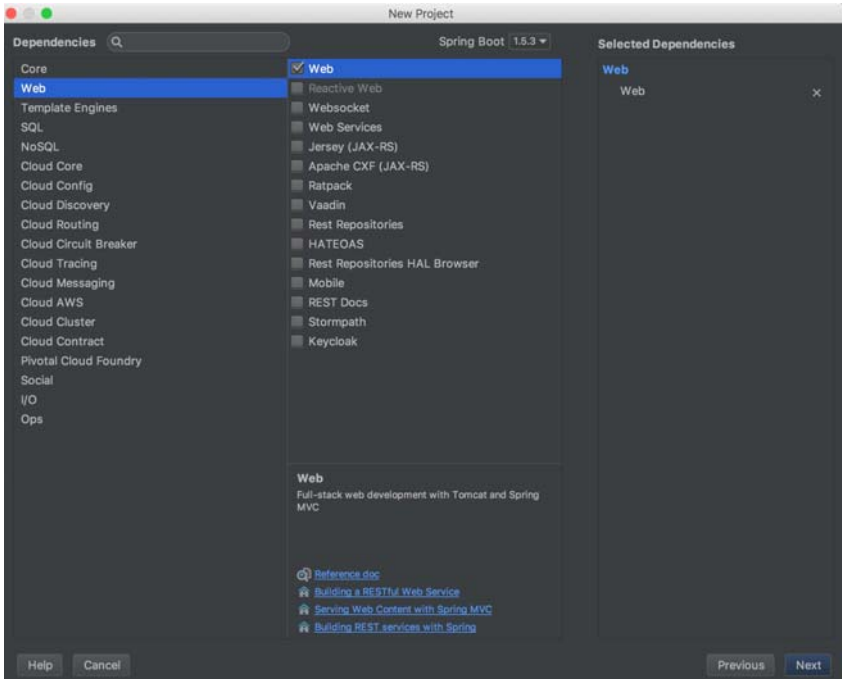
IDEA 提供了多种方式去创建工程，非常便捷。本节介绍采用 Spring Initializr 的方式来创建工程。在图 3-1 的界面选择“Create New Project”，创建新工程，选择“Spring Initalizr”的方式创建 Spring Boot 工程，如图 3-3 所示。

单击“Next”，填写 Group（例如“com.forezp”）和 Artifact（例如“hello-world”），选择默认的 Maven 工程，其他配置默认即可。单击“Next”，进入 Spring Initalizr 模块选择界面，如图 3-4 所示。Spring Initalizr 提供了很多可选的常见功能模块，大多数模块是与 Spring Boot 进行了整合的起步依赖的功能模块，例如 Core 提供了 AOP、Security、Cache、Session 等模块，Web 提供了 Web、Webservice、WebSocket 等模块，读者可以自行查看相关模块的相关功能。本例中选择 Web 模块的 Web 功能，单击“Next”，然后单击“Finish”。

单击“Finish”之后，IDEA 会从 spring.io 网站下载工程模板，下载完成后就是一个完整的 Spring Boot 工程。在工程的目录下有一个 HelloWorldApplication 类，该类为程序的启动类，在该类上添加@RestController 注解，开启 RestController 的功能，写一个接口“/hi”，使用@GetMappin 注解表明为 Get 类型的请求。具体代码如下：



▲图 3-3　采用 Spring Initalizr 创建 Spring Boot 工程



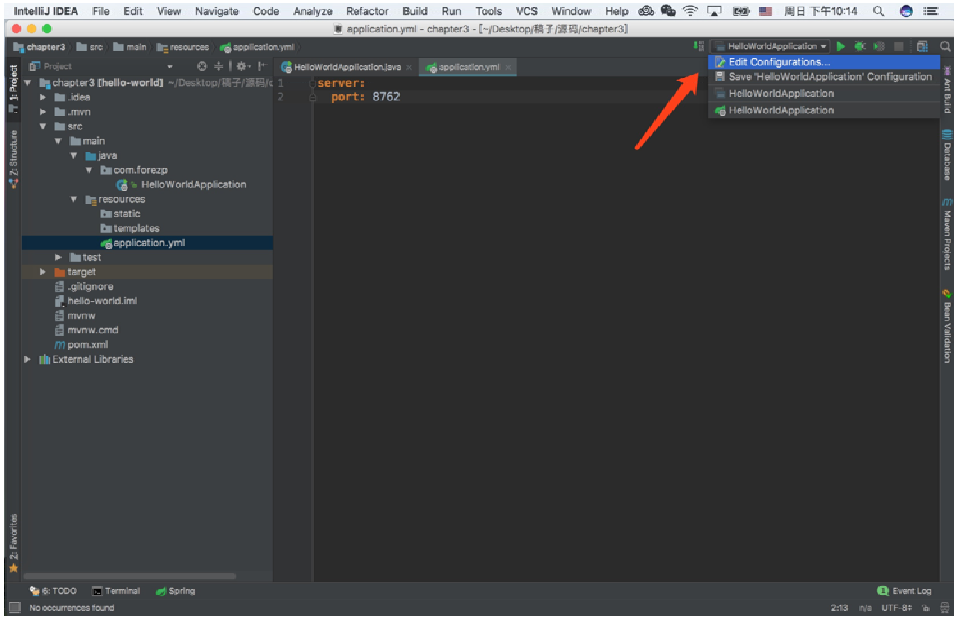
▲图 3-4　使用 Spring Initalizr 创建 Spring Boot 工程

```
@SpringBootApplication
@RestController
public class HelloWorldApplication {
        public static void main(String[] args) {
                SpringApplication.run(HelloWorldApplication.class, args);
        }
        @GetMapping("/hi")
        public String hi(){
                return "hi,I'm forezp";
        }
}
```

启动 HelloWorldApplication 类的 main 方法，程序启动。程序启动完成后，在浏览器上输入“http://localhost:8080 /hi”，浏览器会显示“hi,I'm forezp”。关于 Spring Boot，会在下一章中会作入门级的详细介绍。

3.2.3　用 IDEA 启动多个 Spring Boot 工程实例

在上述讲解的案例中，一个 Spring Boot 工程经常需要启动多个实例，分别占用不同的端口。在 IDEA 上单击 Application 右边的下三角，弹出选项后，单击“Edit Configuration”，如图 3-5 所示。



▲图 3-5　Edit Configuration 界面

打开配置界面后，将默认的“Single instance only”（单实例）前的对号去掉，如图 3-6 所示。通过修改配置文件 application.yml 的 server.port 端口，并启动。多个实例需要多个不同的

端口号，分别启动即可。

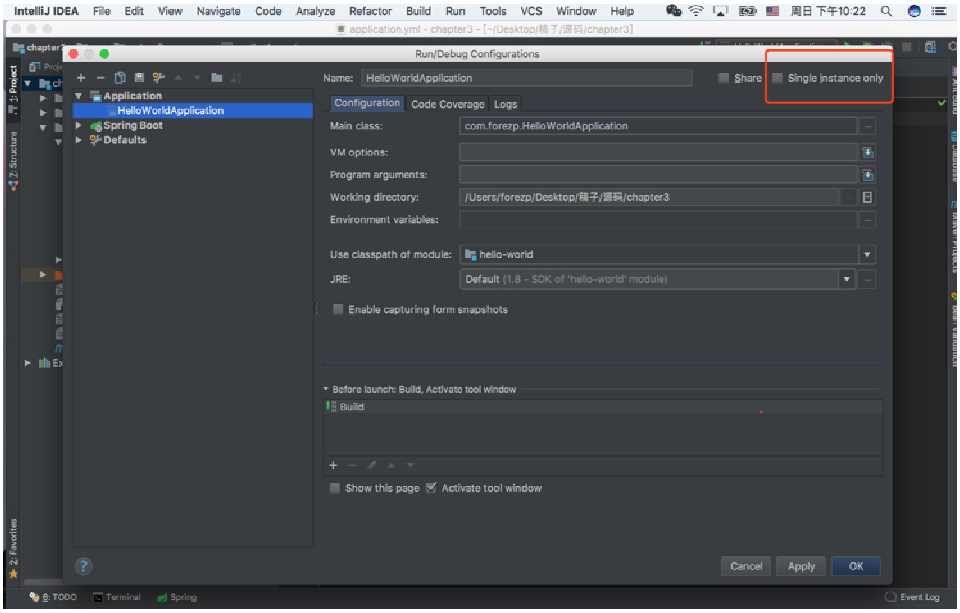


图 3-6　将 Single instance only 前的对号去掉

3.3 构建工具 Maven 的使用

3.3.1　Maven 简介

Apache Maven 是一款软件项目管理的开源工具，是基于工程对象模型（Pom）的概念而设计的。Maven 可以管理项目构建的整个生命周期，包括清理（Clean）、编译（Compile）、打包（Package）、测试（Test）等环节。同时 Maven 提供了非常丰富的插件，使得构建项目和管理项目变得简单。构建一个项目所需要的流程如下。

（1）生成源码。

（2）从源码中生成文档。

（3）编译源码。

（4）测试。

（5）将源码打包成 Jar，运行在服务器、仓库或者其他位置。

Apache Maven 已经实现了以上的全部功能，并且只需要相关的命令就可以完成相关的功能。

3.3.2　Maven 的安装

安装 Maven 前，需要保证 JDK 已经正确安装，并且环境变量已经配置正确，Maven 版本

为 3.0 之后的版本，需要的 JDK 版本至少为 1.6 版本。

从官网 http://maven.apache.org/download.cgi 下载完 Apache Maven，解压到任意目录，例如我解压到/usr/local 下：

```
tar xzvf apache-maven-3.5.0-bin.tar.gz          sudo mv apache-maven-3.5.0 /usr/local/
```

配置环境，先打开配置环境变量的文件，在终端运行如下命令：

```
$ vi ~/.bash_profile
```

在配置文件中需要配置一个 M2_HOME 变量，它的路径为 Maven 的安装目录路径，添加 M2_HOME 变量到环境变量 path 中。配置文件如下：

```
export M2_HOME=/usr/local/apache-maven-3.5.0
export PATH=$PATH:$M2_HOME/bin
```

检查 Maven 是否安装成功和 Maven 的环境变量是否配置正确，可以使用 Maven 命令“mvn -v”去检查。在终端输入“mvn –v”，如果终端界面显示如下信息，则证明 Maven 安装成功且环境变量配置正确。

```
Apache Maven 3.5.0 (ff8f5e7444045639af65f6095c62210b5713f426; 2017-04-04T03:39:06+08:00)
Maven home: /usr/local/apache-maven-3.5.0
Java version: 1.8.0_121, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/jre
Default locale: zh_CN, platform encoding: UTF-8
OS name: "mac os x", version: "10.12.3", arch: "x86_64", family: "mac"
```

设置 Maven 的本地仓库，在终端输入命令切换到 Maven 配置文件的目录，打开 Maven 的配置文件 settings.xml。在终端输入的命令如下：

```
cd /usr/local/apache-maven-3.5.0/
vim settings.xml
```

打开配置文件 settings.xml 后，在配置文件中修改本地仓库的路径，本案例的本地仓库路径为“/usr/local/mvn_repo”，配置如下：

```
<localRepository>/usr/local/mvn_repo</localRepository>
```

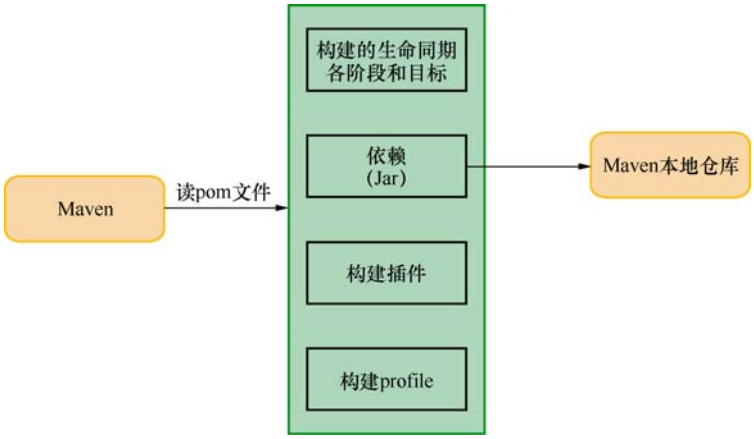
由于 Maven 远程服务器在国外，可以添加阿里云的镜像，这样下载 Jar 包的速度会大大增加，在配置文件 settings.xml 下添加如下内容：

```
<mirrors>
  <mirror>
      <id>alimaven</id>
            <name>aliyun maven</name>
      <url>http://maven.aliyun.com/nexus/content/groups/public/
```

```
        </url>
        <mirrorOf>central</mirrorOf>
    </mirror>
</mirrors>
```

3.3.3 Maven 的核心概念

Maven 的核心是 pom 文件，pom 文件以 xml 文件的形式来表示资源，包括一些依赖 Jar、插件、构建文件等。Maven 的工作过程如图 3-7 所示。



▲图 3-7 Maven 的工作过程

- 首先读取 pom 文件。pom 文件是 Maven 的核心，所有的项目依赖、插件都在 pom 文件中统一管理。
- 下载依赖 Jar 到本地仓库。Maven 命令执行时，首先会检查 pom 文件的依赖 Jar，当检测到本地没有安装依赖 Jar 时，会默认从 Maven 的中央仓库下载依赖 Jar，中央仓库地址为 http://repo1.maven.org/maven2/ 。依赖 Jar 下载成功后，会存放在本地仓库中，如果下载不成功，则该命令执行不会通过。
- 执行构建的生命周期。Maven 的构建过程会被分解成构建阶段和构建目标，它们共同构成了 Maven 的生命周期。
- 执行构建插件。插件可以更方便地执行构建的各个阶段，也可以用插件实现一些额外的功能。目前 Maven 有非常丰富的插件，如果需要，你也可以自己实现 Maven 插件。

3.3.4 编写 Pom 文件

pom 文件是一个 xml 文件，用于描述项目用到的资源、项目依赖、插件、代码位置等信息，是整个工程的核心。pom.xml 文件一般放在项目的根目录下。以 3.2.1 节案例中工程的 pom 文件为例来讲解，其代码如下：

```
<?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.forezp</groupId>
     <artifactId>hello-world</artifactId>
     <version>0.0.1-SNAPSHOT</version>
     <packaging>jar</packaging>

     <name>hello-world</name>
     <description>Demo project for Spring Boot</description>
 <parent>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-parent</artifactId>
          <version>1.5.3.RELEASE</version>
          <relativePath/> <!-- lookup parent from repository -->
     </parent>

     <properties>
     <project.build.sourceEncoding>UTF-8
  </project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8
  </project.reporting.outputEncoding>
         <java.version>1.8</java.version>
     </properties>

     <dependencies>
          <dependency>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-starter-web</artifactId>
          </dependency>

          <dependency>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-starter-test</artifactId>
              <scope>test</scope>
          </dependency>
      </dependencies>

      <build>
          <plugins>
              <plugin>
                  <groupId>org.springframework.boot</groupId>
```

```
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

pom.xml 的第一行指定了 xml 的版本和编码方式。project 的标签是该文件的根元素，它声明了 pom 相关的命名空间。modelVersion 指定了 pom 的版本，对于 Maven 3 来说，它的版本为 4.0.0。

groupId、artifactId 和 version 是 3 个最重要的标签，根据这 3 个标签，可以在 Maven 仓库中唯一确定该依赖 Jar。其中，groupId 代表了公司、组织的名称，一般为公司域名的倒写，如本例中的 com.forezp；artifactId 代表该项目的全局唯一 Id，如本例中的 hello-world；version 是指该项目的版本。将项目上传到 Maven 仓库中，有这 3 个标签才能准确无误地找到该 Jar 包。

parent 标签用于指定父 pom，本案例采用的父 pom 是版本号为 1.5.3.RELEASE 的 spring-boot-starter-parent 的 pom。

properties 标签用于声明一些常量，例如上述代码中的源码编码为 UTF-8，输出代码也为 UTF-8，Java 版本为 1.8。

dependencies 标签为依赖的根元素，里面可以包含多个 dependency 元素，dependency 里具体为各个依赖 Jar 的 3 个坐标，即 groupId、artifactId 和 version。其中 version 可以缺省，如果缺省，就会默认为最新发布的版本。

build 为构建标签，它可以包含 plugins（插件）标签，plugins 标签中可以包含若干个 plugin 标签，可以根据项目的需求添加相应的 plugin。本例中有 spring-boot-maven-plugin 插件，用此插件可以启动 Spring Boot 工程。

3.3.5 Maven 构建项目的生命周期

在 Maven 工程中，已经默认定义了构建工程的生命周期，不需要额外引用其他的插件，因为 Maven 本身就已经集成了这些插件。默认的生命周期包括了 23 个阶段，如表 3-1 所示。

表 3-1 Maven 构建工程默认的生命周期

阶　段	描　述
validate	验证工程的完整性
initialize	初始化
generate-sources	生成源码
process-sources	处理源码
generate-resources	生成所有源码
process-resources	处理所有源码
compile	编译

续表

阶　段	描　述
process-classes	处理 class 文件
generate-test-sources	生成测试源码
process-test-sources	处理测试源码
generate-test-resources	生成所有测试源码
process-test-resources	处理所有测试源码
test-compile	测试编译
process-test-classes	处理测试 class 文件
test	测试
prepare-package	预打包
package	打包（如 Jar、War）
pre-integration-test	预集成测试
integration-test	集成测试
post-integration-test	完成集成测试
verify	验证
install	安装到本地仓库
deploy	提交到远程仓库

3.3.6　常用的 Maven 命令

（1）mvn clean 删除工程的 target 目录下的所有文件。

（2）mvn package 将工程打为 Jar 包。

在终端上切换到 3.2.1 节中项目的根目录下，输入 mvn package 命令，终端最后会显示如下信息，证明工程打 Jar 包成功。

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ hello-world ---
[INFO] Building jar:
/Users/forezp/IdeaProjects/jianshu2/hello-world/target/hello-world-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.3.RELEASE:repackage (default) @ hello-world ---
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Final Memory: 32M/271M
[INFO] ------------------------------------------------------------
```

上述的 mvn package 打包命令不是一个简单的命令，它是由一系列有序的命令构成的，

mvn package 命令执行过程包含以下 6 个阶段。

- ❑ 验证。
- ❑ 编译代码。
- ❑ 处理代码。
- ❑ 生成资源文件。
- ❑ 生成 Jar 包。
- ❑ 测试。

（3）mvn package -Dmaven.test.skip=ture，打包时跳过测试。

（4）mvn compile 编译工程代码，不生成 Jar 包。

（5）mvn install 命令包含了 mvn package 的所有过程，并且将生成的 Jar 包安装到本地仓库。执行 mvn install 命令，可以看到终端输出的日志，经过了与 mvn package 相同的阶段，最后将 Jar 包安装到本地仓库。终端显示的日志如下：

```
[INFO] Installing
/Users/forezp/IdeaProjects/jianshu2/hello-world/target/hello-world-0.0.1-SNAPSHOT.jar
to
/Users/forezp/.m2/repository/com/forezp/hello-world/0.0.1-SNAPSHOT/hello-world-0.0.1-
SNAPSHOT.jar
[INFO] Installing /Users/forezp/IdeaProjects/jianshu2/hello-world/pom.xml to
/Users/forezp/.m2/repository/com/forezp/hello-world/0.0.1-SNAPSHOT/hello-world-0.0.1-
SNAPSHOT.pom
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 7.251 s
[INFO] Finished at: 2017-05-24T22:43:48+08:00
[INFO] Final Memory: 30M/272M
[INFO] ------------------------------------------------------------
```

（6）mvn spring-boot:run 使用 spring-boot 插件，启动 Spring Boot 工程。该命令执行时先检查 Spring Boot 工程源码是否编译，如果工程源码没有编译，则先编译；如果编译了；则启动工程，启动后工程日志如下：

```
[INFO] --- spring-boot-maven-plugin:1.5.3.RELEASE:run (default-cli) @ hello-world ---


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _' | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__' | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::       (v1.5.3.RELEASE)
```

```
2017-05-24 22:53:51.648  INFO 1549 --- [           main] com.forezp.HelloWorldAp
.............
2017-05-24 22:53:54.050  INFO 1549 --- [           main] com.forezp.HelloWorldApplica
tion         : Started HelloWorldApplication in 2.923 seconds (JVM running for 6.067)
```

（7）mvn test 测试。

（8）mvn idea:idea 生成 idea 项目。

（9）mvn jar:jar 只打 Jar 包。

（10）mvn validate 检验资源是否可用。

本章讲述了开发项目中开发环境的搭建和开发工具的使用，难免会有点枯燥，但却是开发微服务的基本前提，下一章将讲述 Spring Boot 的入门内容。

第 4 章　开发框架 Spring Boot

4.1 Spring Boot 简介

Spring Boot 是由 Pivotal 团队开发的 Spring 框架，采用了生产就绪的观点，旨在简化配置，致力于快速开发。Spring Boot 框架提供了自动装配和起步依赖，使开发人员不需要配置各种 xml 文件。通过这种方式，极大地提高了程序的开发速度。因此，Spring Boot 被认为是新一代的 Web 开发框架。

在过去的 Spring 开发中，需要引入大量的 xml 文件。Spring 2.5 引入了包扫描，消除了显式的配置 Bean。Spring 3.0 又引入了基于 JavaBean 的配置，这种方式可以取代 xml 文件。尽管如此，在实际的开发中还是需要配置 xml 文件，例如配置 SpringMVC、事务管理器、过滤器、切面等。

在项目的开发过程中，会引入大量的第三方依赖，选择依赖是一件不容易的事，解决依赖与依赖之间的冲突也很耗费精力。所以，在以前的 Spring 开发中，依赖管理也是一件棘手的事情。

Pivotal 团队提供的 Spring Boot 框架，解决了以前 Spring 应用程序开发的痛点。

4.1.1 Spring Boot 的特点

对比之前的 Spring，Spring Boot 有三大特点：自动配置、起步依赖和 Actuator 对运行状态的监控。

自动配置就是程序需要什么，Spring Boot 就会装配什么。例如，当程序的 pom 文件引入了 Feign 的起步依赖，Spring Boot 就会在程序中自动引入默认的 Feign 的配置 Bean。再例如配置 Feign 的 Decoder 时，如果开发人员配置了 Decoder Bean，Spring Boot 就不会引入默认的 Decoder Bean。自动装配使得程序开发变得非常便捷、智能化。

在以前开发过程中，向项目添加依赖是一件非常有挑战的事情。选择版本，解决版本冲突，十分耗费精力。例如，程序需要 Spring MVC 的功能，那么需要引入 spring-core、spring-web 和 spring-webmvc 等依赖，但是如果程序使用 Spring Boot 的起步依赖，只需要加入

spring-boot-starter-web 的依赖，它会自动引入 Spring MVC 功能的相关依赖。

Spring Boot 能够提供自动装配和起步依赖，解决了以前重量级的 xml 配置和依赖管理的各种问题。一切都显得那么敏捷、智能，但是却带来了一系列的其他问题：开发者该怎么知道应用程序中注入了哪些 Bean？应用程序的运行状态是怎么样的？为了解决这些问题，Spring Boot 提供了 Actuator 组件，并提供了对程序的运行状态的监控功能。

4.1.2　Spring Boot 的优点

Spring Boot 不仅提供了自动装配、起步依赖，还自带了不少非功能性的特性，例如安全、度量、健康检查、内嵌 Servlet 容器和外置配置。开发人员可以更加敏捷快速地开发 Spring 程序，专注于应用程序本身的业务开发，而不是在 Spring 的配置上花费大量的精力。

另外，Actuator 提供了运行时的 Spring Boot 程序中的监控端点，让开发人员和运维人员实时了解程序的运行状况。

4.2　用 IDEA 构建 Spring Boot 工程

打开“IDEA”→“new Project”→“Spring Initializr”→填写“group”和“artifact”→勾选“web”（开启 web 功能）→单击“下一步”。IDEA 会自动下载 Spring Boot 工程的模板。

4.2.1　项目结构

创建完工程后，工程的目录结构如下：

```
- src
     -main
          -java
               -package
                    -SpringbootApplication
          -resouces
               - statics
               - templates
               - application.yml
     -test
- pom
```

各目录含义如下。

- pom 文件为依赖管理文件。
- resouces 为资源文件夹。
- statics 为静态资源。
- templates 为模板资源。
- application.yml 为配置文件。
- SpringbootApplication 为程序的启动类。

4.2.2 在 Spring Boot 工程中构建 Web

打开用 IDEA 创建的项目，其依赖管理文件 pom.xml 有 spring-boot-starter-web 和 spring-boot-starter-test 的起步依赖。其中，spring-boot-starter-web 为 Web 功能的起步依赖，它会自动导入与 Web 相关的依赖。spring-boot-starter-test 为 Spring Boot 测试功能的起步依赖，它会导入与 Spring Boot 测试相关的依赖。代码如下：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
</dependency>
    <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

在工程的代码包的主目录下有一个 SpringbootFirstApplication 的类，该类是程序的启动类，代码如下：

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

其中，@ SpringBootApplication 注解包含了@SpringBootConfiguration、@EnableAuto-Configuration 和@ComponentScan ，开启了包扫描、配置和自动配置的功能。

这是一个完整的、具有 Web 功能的工程，为了演示 Web 效果，建一个 Web 层的 Controller，代码如下：

```
@RestController
public class HelloController {
   @RequestMapping("/hello")
   public String index() {
      return "Greetings from Spring Boot!";
   }
}
```

其中，@RestController 注解表明这个类是一个 RestController。@RestController 是 Spring 4.0 版本的一个注解，它的功能相当于@Controller 注解和@ResponseBody 注解之和。

@RequestMapping 注解是配置请求地址的 Url 映射的。

启动 SpringbootFirstApplication 的 main 方法，Spring Boot 程序启动。在控制台会打印启动的日志，程序的默认端口为 8080。

打开浏览器，在浏览器上输入“http://localhost:8080/hello”，浏览器会显示“Greetings from Spring Boot!”

你会不会觉得Spring Boot的确很神奇？Spring Boot的神奇之处在于，在程序中没有做web.xml的配置，也没有做 SpringMVC 的配置，甚至都不用部署在 Tomcat 上，就可以构建一个具备 Web 功能的工程。其实，Spring Boot 自动为你做了这些配置，并且它默认内嵌了 Tomcat 容器。

4.2.3　Spring Boot 的测试

Spring Boot 开启测试也非常简单，只需要加@RunWith(SpringRunner.class)和@ SpringBootTest 注解，在@SpringBootTest 注解加上 Web 测试环境的端口为随机端口的配置。TestRestTemplate 类为 RestTemplate 测试类，RestTemplate 用于远程调用 Http API 接口。测试代码如下：

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HelloControllerIT {
   @LocalServerPort
   private int port;
   private URL base;
   @Autowired
   private TestRestTemplate template;
   @Before
   public void setUp() throws Exception {
      this.base = new URL("http://localhost:" + port + "/hello");
}
   @Test
   public void getHello() throws Exception {
      ResponseEntity<String> response = template.getForEntity(base.toString(),
              String.class);
      assertThat(response.getBody(), equalTo("Greetings from Spring Boot!"));
   }
}
```

启动测试类的 getHello( ) 方法，通过控制台可以发现 Spring Boot 程序会先启动，然后运行测试代码，最后测试通过。

4.3　Spring Boot 配置文件详解

Spring Boot 采用了构建生产就绪 Spring 应用程序的观点，旨在让程序快速启动和运行。

在一般情况下，不需要做太多的配置就能够让 Spring Boot 程序正常运行。在一些特殊的情况下，我们需要修改一些配置，或者需要有自己的配置。

4.3.1 自定义属性

在用 IDEA 创建一个 Spring Boot 工程时，系统默认会在 src/main/java/resources 目录下创建一个配置文件 application.properties。它也支持 yml 格式的文件，下面以 yml 格式的文件为例来讲解如何自定义属性。

在工程的配置文件 application.yml 自定义一组属性，如下：

```
my:
 name: forezp
 age: 12
```

如果要读取配置文件 application.yml 的属性值，只需在变量上加 @Value("${属性名}")注解，就可以将配置文件 application.yml 的一个属性值赋给一个变量。新建一个 Controller，其代码清单如下：

```
@RestController
public class MiyaController {
    @Value("${my.name}")
    private String name;
    @Value("${my.age}")
    private int age;

    @RequestMapping(value = "/miya")
    public String miya(){
       return name+":"+age;
    }
}
```

启动工程 SpringBoot，打开浏览器访问“http://localhost:8080/miya”，浏览器显示如下：

```
forezp:12
```

这说明配置文件 application.yml 的属性 my.name 和 my.age 已经成功读入应用程序中。

4.3.2 将配置文件的属性赋给实体类

当有很多配置属性时，如果逐个地读取属性会非常麻烦。通常的做法会把这些属性名作为变量名来创建一个 JavaBean 的变量，并将属性值赋给 JavaBean 变量的值。

在配置文件 application.yml 中添加如下属性：

```
my:
 name: forezp
```

```
age: 12
number:  ${random.int}
uuid : ${random.uuid}
max: ${random.int(10)}
value: ${random.value}
greeting: hi,i'm  ${my.name}
```

其中，配置文件中用到了${random} ，它可以用来生成各种不同类型的随机值。random.int 随机生成一个 int 类型的值，random.uuid 随机生成一个 uuid，random.value 随机生成一个值，random.int(10)随机生成一个小于 10 的整数。

怎么将这些属性赋给一个 JavaBean 呢？创建一个 JavaBean，其代码清单如下：

```
@ConfigurationProperties(prefix = "my")
@Component
public class ConfigBean {
   private String name;
   private int age;
   private int number;
   private String uuid;
   private int max;
   private String value;
   private String greeting;
   //省略了 getter setter....
}
```

在上面的代码中，在 ConfigBean 类上加一个注解@ConfigurationProperties，表明该类为配置属性类，并加上配置的 prefix，例如本案例的“my”。另外需要在 ConfigBean 类上加@Component 注解，Spring Boot 在启动时通过包扫描将该类作为一个 Bean 注入 IoC 容器中。

创建一个 Controller，读取 ConfigBean 类的属性。在 Controller 类上，加@EnableConfigurationProperties 注解，并指明 ConfigBean 类，其代码清单如下：

```
@RestController
@EnableConfigurationProperties({ConfigBean.class})
public class LucyController {
   @Autowired
   ConfigBean configBean;
   @RequestMapping(value = "/lucy")
   public String miya(){
      return configBean.getGreeting()+"-"+configBean.getName()+"-"+
configBean.getUuid()+ "-"+configBean.getMax();
   }
}
```

启动工程，在浏览器上访问“http://localhost:8080/lucy”，浏览器会显示从配置文件读取的属性。

4.3.3 自定义配置文件

上面介绍了如何把配置属性写到 application.yml 配置文件中，并把配置属性读取到一个配置类中。有时属性太多，把所有的配置属性都写到 application.yml 配置文件中不太合适，这时需要自定义配置文件。例如在 src/main/resources 目录下自定义一个 test.properties 配置文件，其配置信息如下：

```
com.forezp.name=forezp
com.forezp.age=12
```

如何将这个配置文件 test.properties 的属性和属性值赋给一个 JavaBean 呢？需要在类名上加@Configuration、@PropertySource 和@ConfigurationProperties 这 3 个注解。需要注意的是，若 Spring Boot 版本为 1.4 或 1.4 之前，则需要在@PropertySource 注解上加 location，并指明该配置文件的路径。本案例采用的 Spring Boot 版本为 1.5，代码如下：

```
@Configuration
@PropertySource(value = "classpath:test.properties")
@ConfigurationProperties(prefix = "com.forezp")
public class User {
   private String name;
   private int age;
  //…省略 getter 、setter
}
```

写一个 LucyController 的类，在类的上方加上@RestController 注解，开启 RestControllerde 的功能；加上@EnableConfigurationProperties 注解，并指明需要引用的 JavaBean 的类，开启引用配置属性的功能，其代码清单如下：

```
@RestController
@EnableConfigurationProperties({ConfigBean.class,User.class})
public class LucyController {
   @Autowired
   ConfigBean configBean;

   @RequestMapping(value = "/lucy")
   public String miya(){
     return configBean.getGreeting()+configBean.getName()+
configBean.getUuid()+ configBean. getMax();
   }

   @Autowired
   User user;
   @RequestMapping(value = "/user")
   public String user(){
```

```
        return user.getName()+": "+user.getAge();
    }
}
```

启动工程，在浏览器上访问“http://localhost:8080/user”。浏览器会显示“forezp：12 ”，这说明自定义配置文件的属性被读取到了 JavaBean 中。

4.3.4　多个环境的配置文件

在实际的开发过程中，可能有多个不同环境的配置文件，例如：开发环境、测试环境、生产环境等。Spring Boot 支持程序启动时在配置文件 applicaition.yml 中指定环境的配置文件，配置文件的格式为 application-{profile}.properties，其中{profile}对应环境标识，例如：

- application-test.properties——测试环境；
- application-dev.properties——开发环境；
- application-prod.properties——生产环境。

如何指定这个环境配置文件呢？只需要在 application.yml 中加上 spring.profiles.active 的配置，该配置指定采用哪一个 profiles。例如使用 application-dev.properties，则配置代码如下：

```
spring:
  profiles:
    active: dev
```

其中，application-dev.yml 的配置文件中指定程序的启动端口，配置代码如下：

```
server:
  port: 8082
```

启动工程，查看控制台打印的日志，程序的启动端口为 8082，而不是默认的 8080，这说明配置文件生效了。

另外，我们也可以通过 java –jar 这种方式启动程序，并指定程序的配置文件，启动命令如下：

```
$ java –jar springbootdemo.jar -- spring.profiles.active=dev
```

4.4　运行状态监控 Actuator

Spring Boot 的 Actuator 提供了运行状态监控的功能，Actuator 的监控数据可以通过 REST、远程 shell 和 JMX 方式获得。我们首先来介绍通过 REST 方式查看 Actuator 的节点的方法，这种是最常见且简单的方法。

在工程的 pom 文件中引入 Actuator 的起步依赖 spring-boot-starter-actuator，代码清单如下：

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

```
</dependency>
```

在配置文件 application.yml 中配置 management.port 和 management.security.enabled，这两个配置分别配置了 Actuator 对外暴露 REST API 接口的端口号和 Actuator 采取非安全验证方式，其代码清单如下：

```
management:
  port: 9001
  security:
   enabled: false
```

在上述的配置代码中指定了 Actuator 对外暴露 REST API 接口的端口为 9001，如果不指定，端口为应用程序的启动端口，这样做的目的是将程序端口和程序的监控端口分开。Spring Boot 1.5x 版本默认开启了 Actuator 的安全验证，为了能够在浏览器上展示效果，不做安全验证，将 management.security. enables=false。启动工程，在控制台可以看到如下信息：

```
Mapped"{[/autoconfig||/autoconfig.json],methods=[GET],produces=[application/vnd.sprin
gboot.actuator.v1+json || application/json]}"
Mapped"{[/beans||/beans.json],methods=[GET],produces=[application/vnd.spring-boot.act
uator.v1+json||application/json]}"
Mapped"{[/trace||/trace.json],methods=[GET],produces=[application/vnd.spring-boot.act
uator.v1+json || application/json]}"
...
```

Spring Boot Actuator 的关键特性是在应用程序里提供众多的 Web 节点，通过这些节点可以实时地了解应用程序的运行状况。有了 Actuator，你可以知道 Bean 在 Spring 应用程序上下文里是如何组装在一起的，并且可以获取环境属性的信息和运行时度量信息等。

Actuator 提供了 13 个 API 接口，用于监控运行状态的 Spring Boot 的状况，具体如表 4-1 所示。

表 4-1 Actuator 端口信息

类 型	API 端口	描 述
GET	/autoconfig	提供了一份自动配置报告，记录哪些自动配置条件通过了，哪些没有通过
GET	/configprops	描述配置属性如何注入 Bean
GET	/beans	描述应用程序上下文里全部的 Bean，以及它们的关系
GET	/dump	获取线程活动的快照
GET	/env	获取全部环境属性
GET	/env/{name}	根据名称获取特定的环境属性值
GET	/health	应用程序的健康指标
GET	/info	获取应用程序的信息

续表

类　型	API 端口	描　述
GET	/mappings	描述全部的 URI 路径，以及它们和控制器（包含 Actuator 端点）的映射关系
GET	/metrics	获取应用程序度量信息，例如内存用量和 HTTP 请求计数
GET	/metrics/{name}	获取程序的指定名称的度量信息
POST	/shutdown	关闭应用程序，需要将 endpoints.shutdown.enabled 设置为 true
GET	/trace	提供基本的 HTTP 请求跟踪信息（时间戳、HTTP 头等）

4.4.1　查看运行程序的健康状态

打开浏览器访问“http://localhost:9091/heath”，可以查看应用程序的运行状态和磁盘状态等信息。浏览器显示的信息如下：

```
{
   "status": "UP",
   "diskSpace": {
      "status": "UP",
      "total": 392461021184,
      "free": 381625602048,
      "threshold": 10485760
   }
}
```

“/health”API 接口提供的信息是由一个或多个健康指示器提供的健康信息的组合结果。如表 4-2 所示，列出了 Spring Boot 自带的健康指示器。

表 4-2　　Spring Boot 自带的健康指示器

指　示　器	键	内　容
ApplicationHealthIndicator	none	永远为 UP
DataSourceHealthIndicator	db	如果数据库能连上，则为 UP；否则为 DOWN
DiskSpaceHealthIndicator	diskSpace	如果可用空间大于阈值，则为 UP 和可用磁盘空间；如果空间不足，则为 DOWN
JmsHealthIndicator	jms	如果能连上消息代理，则为 UP；否则为 DOWN
MailHealthIndicator	mail	如果能连上邮件服务器，则为 UP 和邮件服务器主机和端口；否则为 DOWN
MongoHealthIndicator	mongo	如果能连上 MongoDB 服务器，则为 UP 和 MongoDB 服务器版本；否则为 DOWN
RabbitHealthIndicator	rabbit	如果能连上 RabbitMQ 服务器，则为 UP 和版本号；否则为 DOWN
RedisHealthIndicator	redis	如果能连上服务器，则为 UP 和 Redis 服务器版本；否则为 DOWN
SolrHealthIndicator	solr	如果能连上 Solr 服务器，则为 UP；否则为 DOWN

4.4.2 查看运行程序的 Bean

如果需要了解 Spring Boot 上下文注入了哪些 Bean，这些 Bean 的类型、状态、生命周期等信息时，只需要发起一个 GET 类型的请求，请求 API 为“/beans”，在浏览器上访问“http://locahost:9091/beans”，浏览器会显示如下的信息：


```
 [
    {
        "context": "application:dev:8082",
        "parent": null,
        "beans": [
           {
              "bean": "springbootFirstApplication",
              "aliases": [

              ],
              "scope": "singleton",
              "type": "com.forezp.SpringbootFirstApplication$$EnhancerBySpringCGLIB$$
3efbe85a",
              "resource": "null",
              "dependencies": [

              ]
           },
           {
              "bean": "org.springframework.boot.autoconfigure.internalCachingMetadata
ReaderFactory",
              "aliases": [

              ],
              "scope": "singleton",
              "type": "org.springframework.core.type.classreading.CachingMetadataRead
erFactory",
              "resource": "null",
              "dependencies": [

              ]
           },
           {
              "bean": "configBean",
              "aliases": [

              ],
              "scope": "singleton",
              "type": "com.forezp.bean.ConfigBean",
```

```
            "resource": "file [F:/book/springboot-first-application/target/classes/
com/forezp/bean/ConfigBean.class]",
            "dependencies": [

            ]
         },
         {
            "bean": "user",
            "aliases": [

            ],
            "scope": "singleton",
            "type": "com.forezp.bean.User$$EnhancerBySpringCGLIB$$eb8cd986",
            "resource": "file [F:/book/springboot-first-application/target/classes/
com/forezp/bean/User.class]",
            "dependencies": [

            ]
         },
         {
            "bean": "helloController",
            "aliases": [

            ],
            "scope": "singleton",
            "type": "com.forezp.web.HelloController",
            "resource": "file [F:/book/springboot-first-application/target/classes/
com/forezp/web/HelloController.class]",
            "dependencies": [

            ]
         },
         {
            "bean": "lucyController",
            "aliases": [

            ],
            "scope": "singleton",
            "type": "com.forezp.web.LucyController",
            "resource": "file [F:/book/springboot-first-application/target/classes/
com/forezp/web/LucyController.class]",
            "dependencies": [
                "configBean",
                "user"
            ]
         }
```

```
        ...
    ]
```

在返回的信息中包含了 Bean 的以下 4 类信息。

- bean： Spring 应用程序上下文中的 Bean 名称或 Id。
- resource：class 文件的物理位置，通常是一个 Url，指向构建出的 Jar 文件的路径。
- scope：Bean 的作用域（通常是单例 singleton，也可以是 ptototype、request 和 session）。
- type:：Bean 的类型。

4.4.3 使用 Actuator 关闭应用程序

当需要关闭某个应用程序时，只需要通过 Actuator 发送一个 POST 请求“/shutdown”。很显然，关闭程序是一件非常危险的事，所以默认的情况下关闭应该程序的 API 接口没有开启的。通过 Curl 模拟关闭应用程序的请求，Curl 命令如下：

```
$ curl -X POST http://localhost:9091/shutdown
```

得到的响应信息如下：

```
{
  "timestamp": 1493092036024,
  "status": 404,
  "error": "Not Found",
  "message": "No message available",
  "path": "/shutdown"
}
```

上述信息显示找不到该请求路径，这是因为在默认的情况下这个节点是没有开启的，需要将 endpoints. shutdown. enabled 改为 true。在程序的配置文件 application.yml 中添加如下代码：

```
endpoints:
  shutdown:
    enabled: true
```

加上配置之后，重启 Spring Boot 程序，再发送一次 POST 请求，请求 API 接口地址为 http://localhost:9091/shutdown，得到的响应信息如下：

```
{
  "message": "Shutting down, bye..."
}
```

从得到的响应信息可以知道程序已经关闭。另外，Actuator 的其他 10 个 API 接口为 Spring Boot 程序的运行状态给开发人员或者运维人员提供了许多有用的信息，这些信息帮助我们更好地了解程序所处的状态，例如稳定性如何、故障点在哪里。在这里就不一一介绍了，有兴趣

的读者可以对每个 API 接口逐一了解。

4.4.4　使用 shell 连接 Actuator

通过 REST API 这种方式，开发人员通过 Actuator 可以很方便地了解运行中的程序的监控信息。Actuator 也可以通过 shell 的方式连接。通过 shell 连接 Actuator，需要在工程的 pom 文件加上 shell 的起步依赖 spring-boot-starter-remote-shell，代码如下：

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

在程序的 pom 文件加上 spring-boot-starter-remote-shell 起步依赖后，启动 Spring Boot 应用程序，在程序的控制台会输出连接 shell 的密码，密码是随机的，每次都不一样，大概格式如下：

```
Using default password for shell access: 45f17018-5839-478e-a1a1-06de4cc82d4f
```

与密码相对应的用户名是 user，可以通过远程 SSH 连接 shell，它的端口是 2000，这是固定的。如果你是用 Mac，这时可以用终端连接 shell，在终端输入连接命令，命令如下：

```
ssh user@localhost -p 2000
Password authentication
Password:
PTY allocation request failed on channel 0
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::       (v1.5.2.RELEASE)
```

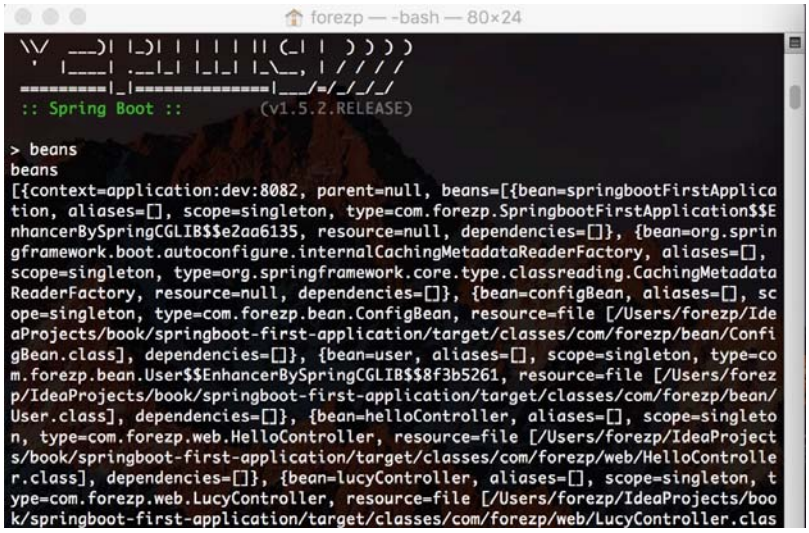
连接上 shell 后，这时可以通过终端查看 Actuator 的各个端点。Spring Boot 提供了 4 个特有的 shell 命令，如表 4-3 所示。

表 4-3　　Spring Boot 提供的 4 个特有的 shell 命令

命　令	说　明
beans	列出了 Spring Boot 程序应用上下文的 Bean
endpoint	调用 Actuator 端点
metrics	Spring Boot 的度量信息
autoconfig	生成自动配置说明报告

现在以第一个命令 beans 为例来做演示，连接上 shell 后，在终端输入 beans，终端显示了应用上下文的注册信息，如图 4-1 所示。

由图 4-1 可知，beans 命令和通过请求 REST API 接口“/beans”的结果相同，都是以 JSON 的数据格式输出了应用上下文的所有 bean 的信息。其他命令就不一一展示了。



▲图 4-1 beans 命令的输出信息

Actuator 是 Spring Boot 的一个非常重要的功能，Actuator 为开发人员提供了 Spring Boot 的运行状态信息，通过 Actuator 可以查看程序的运行状态的信息。

4.5 Spring Boot 整合 JPA

JPA 全称为 JAVA Persistence API，它是一个数据持久化的类和方法的集合。JPA 的目标是制定一个由很多数据库供应商实现的 API，开发人员可以通过编码实现该 API。目前，在 Java 项目开发中提到 JPA 一般是指用 Hibernate 的实现，因为在 Java 的 ORM 框架中，只有 Hibernate 实现得最好。本节以案例的形式来讲述如何在 Spring Boot 工程中使用 JPA。另外，案例使用的数据库为 MySQL 数据库，需要读者提前安装好。

（1）新建一个 Spring Boot 项目

在 Spring Boot 工程的 pom 文件依次引入 Web 功能的起步依赖 spring-boot-starter-web、JPA 的起步依赖 spring-boot-starter-data-jpa、MySQL 数据库的连接器的依赖 mysql-connector-java，其代码如下：

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

```
</dependency>
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-data-jpa
     </artifactId>
</dependency>
<dependency>
     <groupId>mysql</groupId>
     <artifactId>mysql-connector-java</artifactId>
     <scope>runtime</scope>
</dependency>
```

（2）配置数据源

在工程的配置文件 application.yml 文件中加上相应的配置，需要配置两个选项：DataSource 数据源的配置和 JPA 的配置。其中，数据源的配置包括连接 MySQL 的驱动类（例如 com.mysql.jdbc.Driver）、MySQL 数据库的地址 Url、MySQL 数据库的用户名 username 和密码 password。JPA 的配置包括 hibernate. ddl-auto 配置，配置为 create 时，程序启动时会在 MySQL 数据库中建表；配置为 update 时，在程序启动时不会在 MySQL 数据库中建表；jpa.show-sql 配置为在通过 JPA 操作数据库时是否显示操作的 SQL 语句。配置代码如下：

```
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/spring-cloud?useUnicode=true&characterEncoding=u
tf8&characterSetResults=utf8
    username: root
    password: 123456
  jpa:
    hibernate:
      ddl-auto: create  # 第一次简表 create  后面用 update
    show-sql: true
```

（3）创建实体对象

通过@Entity 注解表明该类是一个实体类，它和数据库的表名相对应；@Id 注解表明该变量对应于数据库中的 Id，@GeneratedValue 注解配置 Id 字段为自增长；@Column 表明该变量对应于数据库表中的字段，unique = true 表明该变量对应于数据库表中的字段为唯一约束。代码如下：

```
@Entity
public class User {
     @Id
     @GeneratedValue(strategy = GenerationType.IDENTITY)
     private Long id;
```

```
    @Column(nullable = false,  unique = true)
    private String username;

    @Column
    private String password;

  省略了 getter、setter 方法
}
```

（4）创建 DAO 层

数据访问层 DAO，通过编写一个 UserDao 类，该类继承 JpaRepository 的接口，继承之后就能对数据库进行读写操作，包含了基本的单表查询的方法，非常方便。在 UserDao 类写一个 findByUsername 的方法，传入参数 username，JPA 已经实现了根据某个字段去查找的方法，所以该方法可以根据 username 字段从数据库中获取 User 的数据，不需要做额外的编码。代码如下：

```
public interface UserDao extends JpaRepository<User, Long>{
    User findByUsername(String username);
}
```

（5）创建 Service 层

在 UserService 类中注入 UserDao，并写一个根据用户名获取用户的方法，代码如下：

```
@Service
public class UserService {
    @Autowired
    private UserDao userRepository;
    public User findUserByName(String username){
       return userRepository.findByUsername(username);
    }
}
```

（6）创建 Controller 层

在 UserController 类写一个 Get 类型的 API 接口，其中需要说明的是@PathVariable 注解，可以获取 RESTful 风格的 Url 路径上的参数。

```
@RequestMapping("/user")
@RestController
public class UserController {
    @Autowired
    UserService userService;
    @GetMapping("/{username}")
    public User getUser(@PathVariable("username")String username){
       return userService.findUserByName(username);
    }
}
```

启动运行程序，控制台输出的日志如下：

```
Hibernate: drop table if exists user
Hibernate: create table user (id bigint not null auto_increment, password varchar(255
), username varchar(255) not null, primary key (id))
```

可见，JPA 在启动程序时在数据中创建了 user 表。在终端上用命令行 show tables，确实发现 user 表被创建了。这时在数据库中插入一条数据：

```
insert into 'user'(username,password) VALUES('forezp','123456');
```

再打开浏览器，在浏览器中输入“localhost:8080/user/forezp”，可以从数据库读取 username 字段为 forezp 的用户对象，浏览器显示的数据如下：

```
{"id":1,"username":"forezp","password":"123456"}
```

4.6 Spring Boot 整合 Redis

4.6.1 Redis 简介

Redis 是一个开源的、先进的 key-value 存储系统，可用于构建高性能的存储系统。Redis 支持数据结构有字符串、哈希、列表、集合、排序集合、位图、超文本等。NoSQL（Not Only SQL）泛指非关系型的数据库。Redis 是一种 NoSQL，Redis 具有很多的优点，例如读写非常快速，支持丰富的数据类型，所有的操作都是原子的。

4.6.2 Redis 的安装

（1）Mac 下安装

通过 brew 命令安装，安装后启动 Redis 服务器和客户端，命令如下：

```
安装：  brew install redis
启动服务器：  redis-server
启动客户端：  redis-cli
```

（2）Windows 下安装

Redis 官方没有提供 Windows 版本，不过微软维护了一个版本，下载地址为 https://github.com/ MSOpenTech/redis/releases，下载.msi 版本，一直单击“下一步”完成安装即可。

4.6.3 在 Spring Boot 中使用 Redis

新建一个 Spring Boot 工程，在工程的 pom 文件中加入 Redis 的起步依赖 spring-boot-starter-data-redis，代码如下：

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

在工程的配置文件 application.yml 中加上 Redis 的数据源配置，例如 host、port、数据库配置信息等。如果 Redis 设置了密码，需要提供密码，选择序号为 1 的数据库，配置 Pool 的相关配置。配置代码如下：

```
spring:
  redis:
    host: localhost
    port: 6379
    password:
    database: 1
    pool:
      max-active: 8
      max-wait: -1
      max-idle: 500
```

数据操作层的 RedisDao 类通过@Repository 注解来注入 Spring IoC 容器中，该类是通过 RedisTemplate 来访问 Redis 的。通过注入 StringRedisTemplate 的 Bean 来对 Redis 数据库中的字符串类型的数据进行操作，写了两个方法，包括向 Redis 中设置 String 类型的数据和从 Redis 中读取 String 类型的数据，代码如下：

```
@Repository
public class RedisDao {
   @Autowired
   private StringRedisTemplate template;
   public void setKey(String key,String value){
     ValueOperations<String, String> ops = template.opsForValue();
     ops.set(key,value,1, TimeUnit.MINUTES);//1 分钟过期
   }
   public String getValue(String key){
      ValueOperations<String, String> ops = this.template.opsForValue();
      return ops.get(key);
   }
}
```

在 SpringBootTest 的测试类中注入 RedisDao，首先通过 RedisDao 向 Redis 设置两组字符串值，即 name 为 forezp，age 为 17 的两组字符串，然后分别通过 RedisDao 从 Redis 中读取这两个值，并打印出来，代码如下：

```
@RunWith(SpringRunner.class)
@SpringBootTest
```

```
public class SpringbootRedisApplicationTests {
   @Test
   public void contextLoads() {
   }
   @Autowired
   RedisDao redisDao;
   @Test
   public void testRedis(){
      redisDao.setKey("name","forezp");
      redisDao.setKey("age","17");
      logger.info(redisDao.getValue("name"));
      logger.info(redisDao.getValue("age"));
   }
}
```

启动单元测试，控制台打印了 forezp 和 17 的两个字符串值。可见，通过 RedisDao 首先向 Redis 数据库中写入了两个数据，然后又读取了这两个数据。

4.7 Spring Boot 整合 Swagger2，搭建 Restful API 在线文档

Swagger，中文“拽”的意思，它是一个功能强大的在线 API 文档的框架，目前它的版本为 2.x，所以称为 Swagger2。Swagger2 提供了在线文档的查阅和测试功能。利用 Swagger2 很容易构建 RESTful 风格的 API，在 Spring Boot 中集成 Swagger2，在本案例中需要以下 5 个步骤。

（1）引入依赖

在工程的 pom 文件中引入依赖，包括 springfox-swagger2 和 springfox-swagger-ui 的依赖，代码如下：

```
<dependency>
   <groupId>io.springfox</groupId>
   <artifactId>springfox-swagger2</artifactId>
    <version>2.6.1</version>
</dependency>
<dependency>
   <groupId>io.springfox</groupId>
   <artifactId>springfox-swagger-ui</artifactId>
   <version>2.6.1</version>
</dependency>
```

（2）配置 Swagger2

写一个配置类 Swagger2，在类的上方加上@Configuration 注解，表明是一个配置类，加上@EnableSwagger2 开启 Swagger2 的功能。在配置类 Swagger2 中需要注入一个 Docket 的 Bean，该 Bean 包含了 apiInfo，即基本 API 文档的描述信息，以及包扫描的基本包名等信息，代码如下：

```
@Configuration
@EnableSwagger2
```

```
public class Swagger2 {
@Bean
   public Docket createRestApi() {
     return new Docket(DocumentationType.SWAGGER_2)
     .apiInfo(apiInfo())
     .select()
     .apis(RequestHandlerSelectors.basePackage("com.forezp.controller"))
     .paths(PathSelectors.any())
     .build();
   }
  private ApiInfo apiInfo() {
     return new ApiInfoBuilder()
       .title("springboot 利用 swagger 构建 api 文档")
       .description("简单优雅的 restfun 风格，http://blog.csdn.net/forezp")
       .termsOfServiceUrl("http://blog.csdn.net/forezp")
       .version("1.0")
       .build();
   }
}
```

（3）写生成文档的注解

Swagger2 通过注解来生成 API 接口文档，文档信息包括接口名、请求方法、参数、返回信息等。通常情况下用于生成在线 API 文档，以下的注解能够满足基本需求，注解及其描述如下。

- @Api：修饰整个类，用于描述 Controller 类。
- @ApiOperation：描述类的方法，或者说一个接口。
- @ApiParam：单个参数描述。
- @ApiModel：用对象来接收参数。
- @ApiProperty：用对象接收参数时，描述对象的一个字段。
- @ApiResponse：HTTP 响应的一个描述。
- @ApiResponses：HTTP 响应的整体描述。
- @ApiIgnore：使用该注解，表示 Swagger2 忽略这个 API。
- @ApiError ：发生错误返回的信息。
- @ApiParamImplicit：一个请求参数。
- @ApiParamsImplicit：多个请求参数。

（4）编写 Service 层代码

本节案例在 4.5 节的案例基础之上进行改造，构建了一组 RESTful 风格的 API 接口，包括获取 User 列表、创建 User、修改 User、根据用户名获取 User、删除 User 的 API 接口，对应于 Service 层的代码如下：

```
@Service
public class UserService {
   @Autowired
```

```
    private UserDao userRepository;
    public User findUserByName(String username){

       return userRepository.findByUsername(username);
    }
    public List<User> findAll(){
       return userRepository.findAll();
    }
    public User saveUser(User user){
      return  userRepository.save(user);
    }
    public User findUserById(long id){
       return userRepository.findOne(id);
    }
    public User updateUser(User user){
       return userRepository.saveAndFlush(user);
    }
    public void deleteUser(long id){
        userRepository.delete(id);
    }
}
```

（5）Web 层

在 Web 层通过 Get、Post、Delete、Put 这 4 种 Http 方法，构建一组以资源为中心的 RESTful 风格的 API 接口，代码如下：

```
@RequestMapping("/user")
@RestController
public class UserController {
   @Autowired
   UserService userService;
   @ApiOperation(value="用户列表", notes="用户列表")
   @RequestMapping(value={""}, method= RequestMethod.GET)
   public List<User> getUsers() {
      List<User> users = userService.findAll();
      return users;
   }
   @ApiOperation(value="创建用户", notes="创建用户")
   @RequestMapping(value="", method=RequestMethod.POST)
   public User postUser(@RequestBody User user) {
     return   userService.saveUser(user);

   }
   @ApiOperation(value="获用户细信息", notes="根据 url 的 id 来获取详细信息")
   @RequestMapping(value="/{id}", method=RequestMethod.GET)
   public User getUser(@PathVariable Long id) {
      return userService.findUserById(id);
```

```
    }

    @ApiOperation(value="更新信息", notes="根据 url 的 id 来指定更新用户信息")
    @RequestMapping(value="/{id}", method= RequestMethod.PUT)
    public User putUser(@PathVariable Long id, @RequestBody User user) {
       User user1 = new User();
       user1.setUsername(user.getUsername());
       user1.setPassword(user.getPassword());
       user1.setId(id);
       return userService.updateUser(user1);
    }
    @ApiOperation(value="删除用户", notes="根据 url 的 id 来指定删除用户")
    @RequestMapping(value="/{id}", method=RequestMethod.DELETE)
    public String deleteUser(@PathVariable Long id) {
       userService.deleteUser(id);
       return "success";
    }
    @ApiIgnore//使用该注解忽略这个 API
    @RequestMapping(value = "/hi", method = RequestMethod.GET)
    public String  jsonTest() {
       return " hi you!";
    }
}
```

通过@ApiOperation 注解描述生成在线文档的具体 API 的说明，其中 value 值为该接口的名称，notes 值为该接口的详细文档说明。这样就可以让 Swagger2 生成在线的 API 接口文档了。如果不需要某接口生成文档，只需要再加@ApiIgnore 注解即可。启动工程，在浏览器上访问 http://localhost:8080/swagger-ui.html，浏览器显示 Swagger-UI 在线文档的界面，界面如图 4-2 所示。

▲图 4-2 Swagger-UI 在线文档的界面

第 5 章　服务注册和发现 Eureka

“Eureka”来源于古希腊词汇，意为“发现了”。在软件领域，Eureka 是 Netflix 在线影片公司开源的一个服务注册与发现的组件，和其他 Netflix 公司的服务组件（例如负载均衡、熔断器、网关等）一起，被 Spring Cloud 社区整合为 Spring Cloud Netflix 模块。

本章将从以下 4 个方面来讲解服务注册与发现模块 Eureka。

- Eureka 简介。
- 编写一个 Eureka 注册和发现的例子。
- 深入理解 Eureka。
- 编写高可用的 Eureka Server。

5.1 Eureka 简介

5.1.1　什么是 Eureka

和 Consul、Zookeeper 类似，Eureka 是一个用于服务注册和发现的组件，最开始主要应用于亚马逊公司旗下的云计算服务平台 AWS。Eureka 分为 Eureka Server 和 Eureka Client，Eureka Server 为 Eureka 服务注册中心，Eureka Client 为 Eureka 客户端。

5.1.2　为什么选择 Eureka

在 Spring Cloud 中，可选择 Consul、Zookeeper 和 Eureka 作为服务注册和发现的组件，那为什么要选择 Eureka 呢？

首先 Eureka 完全开源，是 Netflix 公司的开源产品，经历了 Netflix 公司的生产环境的考验，以及 3 年时间的不断迭代，在功能和性能上都非常稳定，可以放心使用。

其次，Eureka 是 Spring Cloud 首选推荐的服务注册与发现组件，与 Spring Cloud 其他组件可以无缝对接。

最后，Eureka 和其他组件，比如负载均衡组件 Ribbon、熔断器组件 Hystrix、熔断器监控组件 Hystrix Dashboard 组件、熔断器聚合监控 Turbine 组件，以及网关 Zuul 组件相互配合，能够很容易实现服务注册、负载均衡、熔断和智能路由等功能。这些组件都是由 Netflix 公司开源的，一起被称为 Netflix OSS 组件。Netflix OSS 组件由 Spring Cloud 整合为 Spring Cloud Netflix 组件，

它是 Spring Cloud 构架微服务的核心组件，也是基础组件。

5.1.3 Eureka 的基本架构

Eureka 的基本架构如图 2-1 所示，其中主要包括以下 3 种角色。

- Register Service：服务注册中心，它是一个 Eureka Server，提供服务注册和发现的功能。
- Provider Service：服务提供者，它是一个 Eureka Client，提供服务。
- Consumer Service：服务消费者，它是一个 Eureka Cient，消费服务。

服务消费的基本过程如下：首先需要一个服务注册中心 Eureka Server，服务提供者 Eureka Client 向服务注册中心 Eureka Server 注册，将自己的信息（比如服务名和服务的 IP 地址等）通过 REST API 的形式提交给服务注册中心 Eureka Server。同样，服务消费者 Eureka Client 也向服务注册中心 Eureka Server 注册，同时服务消费者获取一份服务注册列表的信息，该列表包含了所有向服务注册中心 Eureka Server 注册的服务信息。获取服务注册列表信息之后，服务消费者就知道服务提供者的 IP 地址，可以通过 Http 远程调度来消费服务提供者的服务。

5.2 编写 Eureka Server

由于本案例有多个 Spring Boot 工程，为了方便管理，采用 Maven 多 Module 的结构，所以需要创建一个 Maven 主工程。需要说明的是，本书中所有的案例都采用这种 Maven 多 Module 的结构。本案例最终完整项目的结构如下：

```
|_chapter5-2
  |_ eureka-client
  |_ eureka-server
  |_ pom.xml
```

创建完主 Maven 工程之后，在主 Maven 的 pom 文件下，引入 eureka-client 和 eureka-server 两个 Module 工程共同所需的依赖，包括版本为 1.5.3.RELEASE 的 Spring Boot 依赖，版本为 Dalston.SR1 的 Spring Cloud 依赖，指定 Java 版本为 1.8，编码为 UTF-8。主 Maven 的 pom 文件代码如下：

```
<?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.forezp</groupId>
    <artifactId>chapter5-2</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>
```

```
    <parent>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-parent</artifactId>
       <version>1.5.3.RELEASE</version>
       <relativePath/>
    </parent>

    <properties>
       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
       <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
       <java.version>1.8</java.version>
       <spring-cloud.version>Dalston.SR1</spring-cloud.version>
   </properties>

   <dependencyManagement>
       <dependencies>
           <dependency>
               <groupId>org.springframework.cloud</groupId>
               <artifactId>spring-cloud-dependencies</artifactId>
               <version>${spring-cloud.version}</version>
               <type>pom</type>
               <scope>import</scope>
           </dependency>
       </dependencies>
    </dependencyManagement>
</project>
```

创建完主 Maven 工程且配置完主工程的 pom 文件后，创建一个 Module 工程，命名为 eureka-server。采用 Spring Initializr 的方式构建，作为服务注册中心 Eureka Server 的工程，其工程目录结构如下：

```
|_eureka-server
   |_src
       |_main
      |_java
        |_com.forezp
            |_EurekaServerApplication
        |_resources
            |_application.yml
     |_test
   |_pom.xml
```

在 eureka-server 工程的 pom 文件引入相关的依赖，包括继承了主 Maven 工程的 pom 文件，引入了 Eureka Server 的起步依赖 spring-cloud-starter-eureka-server，以及 Spring Boot 测试的起步依赖 spring-boot-starter-test。最后还引入了 Spring Boot 的 Maven 插件 spring-boot-maven-plugin，有了

该插件，即可使用 Maven 插件的方式来启动 Spring Boot 工程。具体代码如下：

```
  <parent>
        <groupId>com.forezp</groupId>
        <artifactId>chapter5-2</artifactId>
        <version>1.0-SNAPSHOT</version>
        <relativePath/>
    </parent>

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

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

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

在工程的配置文件 application.yml 中做程序的相关配置，首先通过 server.port 指定 Eureka Server 的端口为 8761。在默认情况下，Eureka Server 会向自己注册，这时需要配置 eureka.client.registerWithEureka 和 eureka.client.fetchRegistry 为 false，防止自己注册自己。配置文件 application.yml 的代码如下：

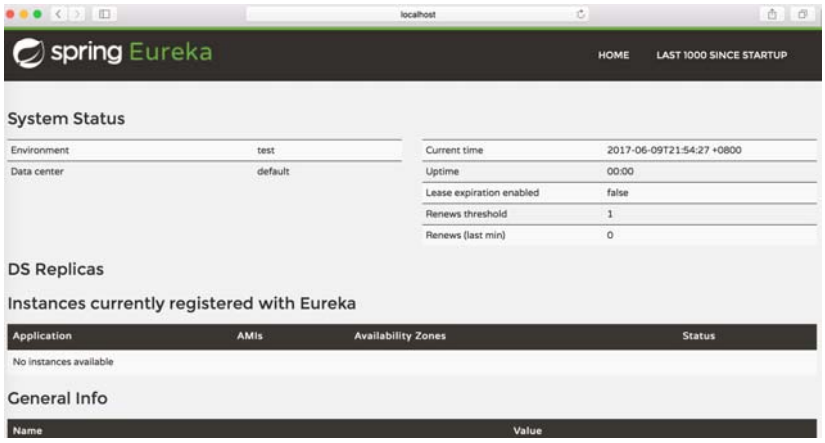
```
server:
  port: 8761

eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
```

```
      defaultZone:
         http://${eureka.instance.hostname}:${server.port}/eureka/
```

在工程的启动类 EurekaServerApplication 加上注解@EnableEurekaServer，开启 Eureka Server 的功能。代码如下：

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaServerApplication {

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

到目前为止，Eureka Server 的所有搭建工作已经完成。启动程序启动类 EurekaServerApplication 的 main 方法，启动程序。在浏览器上访问 Eureka Server 的主界面 http://localhost:8761，在界面上的 Instances currently registered with Eureka 这一项上没有任何注册的实例，没有是正常的，因为还没有 Eureka Client 客户端向注册中心 Eureka Server 注册实例。Eureka Server 的主界面如图 5-1 所示。



▲图 5-1　Eureka Server 的 UI 界面

5.3 编写 Eureka Client

在主 Maven 工程中创建一个新的 Module 工程，命名为 eureka-client，该工程作为 Eureka Client 的工程向服务注册中心 Eureka server 注册。eureka-client 工程创建完之后，在其 pom 文件中引入相关的依赖，其 pom 文件继承了主工程的 pom 文件，并且需要引入 Eureka Client 客户端所需的依赖 spring-cloud-starter-eureka，引入 Web 功能的起步依赖 spring-boot-starter-web，以及 Spring Boot 测试的起步依赖 spring-boot-starter-test，具体代码如下：

```
    <parent>
        <groupId>com.forezp</groupId>
        <artifactId>chapter5-2</artifactId>
        <version>1.0-SNAPSHOT</version>
        <relativePath/>
    </parent>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

在工程的配置文件 bootstrap.yml 做 Eureka Client 客户端的相关配置，配置了程序名为 eureka-client，程序端口为 8762，服务注册地址为 http://localhost:8761/eureka/，代码如下：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
server:
  port: 8762
spring:
  application:
    name: eureka-client
```

上述配置代码中，defaultZone 为默认的 Zone，来源于 AWS 的概念。区域（Region）和可用区（Availability Zone，AZ）是 AWS 的另外两个概念。区域是指服务器所在的区域，比如北美洲、南美洲、欧洲和亚洲等，每个区域一般由多个可用区组成。在本案例中 defaultZone 是指 Eureka Server 的注册地址。

在程序的启动类 EurekaClientApplication 加上注解@EnableEurekaClient 开启 Eureka Client 功能，其代码如下：

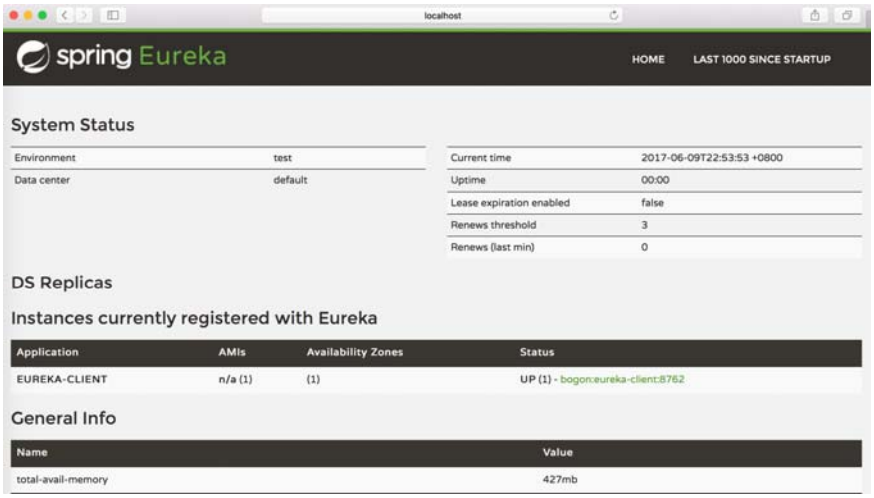
```
@SpringBootApplication
@EnableEurekaClient
public class EurekaClientApplication {
```

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

启动 Eureka Client 工程，启动成功之后，在控制台会打印出如下信息：

```
com.netflix.discovery.DiscoveryClient    : DiscoveryClient_EUREKA-CLIENT/bogon:eureka
-client:8762 - registration status: 204
```

以上的日志信息说明 Eureka Client 客户端已经向 Eureka Server 注册了。在浏览器上打开 Eureka Server 主页 http://localhost:8761，主页显示如图 5-2 所示。



▲图 5-2　Eureka Client 向 Eureka Server 注册

在上图中，在主页上的 Instances currently registered with Eureka 选项中已经有一个实例被注册，Application 为 EUREKA-CLIENT，Staus 为 UP（在线），端口为 8762。这就说明 Eureka Client 已成功向 Eureka Server 注册。

在 eureka-client 工程中写一个 API 接口。新建一个类 HiController，在 HiController 类加上 @RestController 注解，开启 RestController 的功能。@GetMapping 注解表明是一个 Get 请求，其请求地址映射为“/hi”，其中@Value("${server.port}")向配置文件读取配置的端口信息。其完整代码如下：

```
@RestController
public class HiController {

    @Value("${server.port}")
    String port;
    @GetMapping("/hi")
    public String home(@RequestParam String name) {
```

```
        return "hi "+name+",i am from port:" +port;
    }
}
```

在浏览器上访问 http://localhost:8762/hi?name=forezp，浏览器显示如下的信息：

```
hi forezp,i am from port:8762
```

5.4 源码解析 Eureka

5.4.1 Eureka 的一些概念

（1）Register——服务注册

当 Eureka Client 向 Eureka Server 注册时，Eureka Client 提供自身的元数据，比如 IP 地址、端口、运行状况指标的 Url、主页地址等信息。

（2）Renew——服务续约

Eureka Client 在默认的情况下会每隔 30 秒发送一次心跳来进行服务续约。通过服务续约来告知 Eureka Server 该 Eureka Client 仍然可用，没有出现故障。正常情况下，如果 Eureka Server 在 90 秒内没有收到 Eureka Client 的心跳，Eureka Server 会将 Eureka Client 实例从注册列表中删除。注意：官网建议不要更改服务续约的间隔时间。

（3）Fetch Registries——获取服务注册列表信息

Eureka Client 从 Eureka Server 获取服务注册表信息，并将其缓存在本地。Eureka Client 会使用服务注册列表信息查找其他服务的信息，从而进行远程调用。该注册列表信息定时（每 30 秒）更新一次，每次返回注册列表信息可能与 Eureka Client 的缓存信息不同，Eureka Client 会自己处理这些信息。如果由于某种原因导致注册列表信息不能及时匹配，Eureka Client 会重新获取整个注册表信息。Eureka Server 缓存了所有的服务注册列表信息，并将整个注册列表以及每个应用程序的信息进行了压缩，压缩内容和没有压缩的内容完全相同。Eureka Client 和 Eureka Server 可以使用 JSON 和 XML 数据格式进行通信。在默认的情况下，Eureka Client 使用 JSON 格式的方式来获取服务注册列表的信息。

（4）Cancel——服务下线

Eureka Client 在程序关闭时可以向 Eureka Server 发送下线请求。发送请求后，该客户端的实例信息将从 Eureka Server 的服务注册列表中删除。该下线请求不会自动完成，需要在程序关闭时调用以下代码：

```
DiscoveryManager.getInstance().shutdownComponent();
```

（5）Eviction——服务剔除

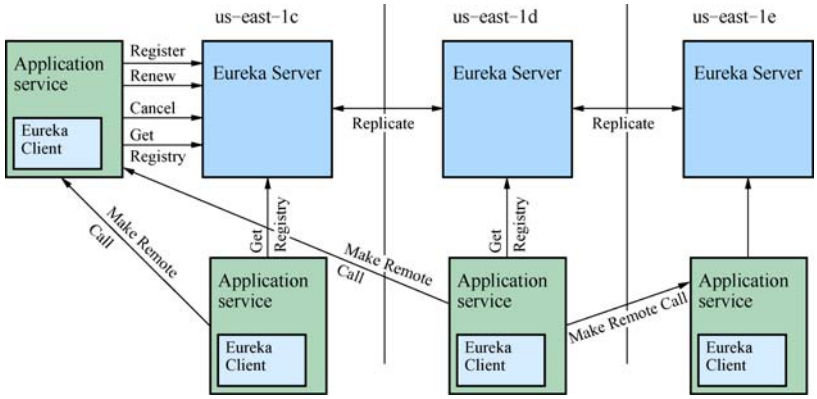
在默认情况下，当 Eureka Client 连续 90 秒没有向 Eureka Server 发送服务续约（即心跳）时，Eureka Server 会将该服务实例从服务注册列表删除，即服务剔除。

5.4.2　Eureka 的高可用架构

图 5-3 为 Eureka 的高可用架构图，该图片来自 Eureka 开源代码的文档，地址为 https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance。

从图 5-3 中可知，在这个架构中有两个角色，即 Eureka Server 和 Eureka Client。而 Eureka Client 又分为 Applicaton Service 和 Application Client，即服务提供者和服务消费者。每个区域有一个 Eureka 集群，并且每个区域至少有一个 Eureka Server 可以处理区域故障，以防服务器瘫痪。

Eureka Client 向 Eureka Server 注册，将自己的客户端信息提交给 Eureka Server。然后，Eureka Client 通过向 Eureka Server 发送心跳（每 30 秒一次）来续约服务。如果某个客户端不能持续续约，那么 Eureka Server 断定该客户端不可用，该不可用的客户端将在大约 90 秒后从 Eureka Serve 服务注册列表中删除。服务注册列表信息和服务续约信息会被复制到集群中的每个 Eureka Serve 节点。来自任何区域的 Eureka Client 都可以获取整个系统的服务注册列表信息。根据这些注册列表信息，Application Client 可以远程调用 Applicaton Service 来消费服务。

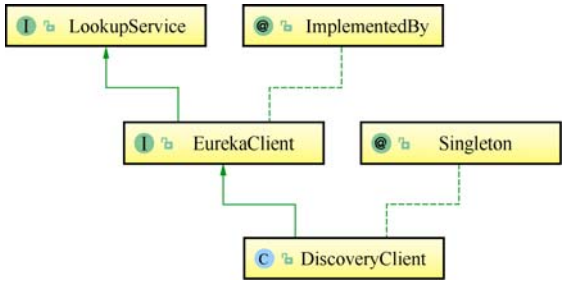


▲图 5-3　Eureka 的高可用架构图

5.4.3　Register 服务注册

服务注册，即 Eureka Client 向 Eureka Server 提交自己的服务信息，包括 IP 地址、端口、ServiceId 等信息。如果 Eureka Client 在配置文件中没有配置 ServiceId，则默认为配置文件中配置的服务名，即${spring.application.name}的值。

当 Eureka Client 启动时，会将自身的服务信息发送到 Eureka Server。这个过程其实非常简单，现在来从源码的角度分析服务注册的过程。在工程的 Maven 的依赖包下，找到 eureka-client-1.6.2.jar 包。在 com.netflix.discovery 包下有一个 DiscoveryClient 类，该类包含了 Eureka Client 向 Eureka Server 注册的相关方法。其中，DiscoveryClient 实现了 EurekaClient 接口，并且它是一个单例模式，而 EurekaClient 继承了 LookupService 接口。它们之间的关系如图 5-4 所示。



▲图 5-4 DiscoveryClient、EurekaClient 和 LookupService 的关系

在 DiscoveryClient 类中有一个服务注册的方法 register()，该方法通过 Http 请求向 Eureka Server 注册，其代码如下：

```
boolean register() throws Throwable {
       logger.info(PREFIX + appPathIdentifier + ": registering service...");
       EurekaHttpResponse<Void> httpResponse;
       try {
           httpResponse = eurekaTransport.registrationClient.register(instanceInfo);
       } catch (Exception e) {
           logger.warn("{} - registration failed {}", PREFIX + appPathIdentifier, e.g
etMessage(), e);
           throw e;
       }
       if (logger.isInfoEnabled()) {
           logger.info("{} - registration status: {}", PREFIX + appPathIdentifier, ht
tpResponse.getStatusCode());
       }
       return httpResponse.getStatusCode() == 204;
   }
```

在 DiscoveryClient 类下继续追踪 register()方法，这个方法被 InstanceInfoReplicator 类的 run()方法调用，其中 InstanceInfoReplicator 实现了 Runnable 接口，run()方法代码如下：

```
   public void run() {
       try {
           discoveryClient.refreshInstanceInfo();

           Long dirtyTimestamp = instanceInfo.isDirtyWithTime();
           if (dirtyTimestamp != null) {
               discoveryClient.register();
               instanceInfo.unsetIsDirty(dirtyTimestamp);
           }
       } catch (Throwable t) {
           logger.warn("There was a problem with the instance info replicator", t);
       } finally {
```

```
            Future next = scheduler.schedule(this, replicationIntervalSeconds, TimeUni
t.SECONDS);
            scheduledPeriodicRef.set(next);
        }
    }
```

而 InstanceInfoReplicator 类是在 DiscoveryClient 初始化过程中使用的，其中有一个 initScheduledTasks()方法，该方法主要开启了获取服务注册列表的信息。如果需要向 Eureka Server 注册，则开启注册，同时开启了定时任务向 Eureka Server 服务续约，具体代码如下：

```
private void initScheduledTasks() {
      ...//省略了任务调度获取注册列表的代码
       if (clientConfig.shouldRegisterWithEureka()) {
        ...
           // Heartbeat timer
           scheduler.schedule(
                  new TimedSupervisorTask(
                          "heartbeat",
                          scheduler,
                          heartbeatExecutor,
                          renewalIntervalInSecs,
                          TimeUnit.SECONDS,
                          expBackOffBound,
                          new HeartbeatThread()
                  ),
                  renewalIntervalInSecs, TimeUnit.SECONDS);

           // InstanceInfo replicator
           instanceInfoReplicator = new InstanceInfoReplicator(
                  this,
                  instanceInfo,
                  clientConfig.getInstanceInfoReplicationIntervalSeconds(),
                  2); // burstSize

           statusChangeListener = new ApplicationInfoManager.StatusChangeListener() {
               @Override
               public String getId() {
                   return "statusChangeListener";
               }

               @Override
               public void notify(StatusChangeEvent statusChangeEvent) {

                   instanceInfoReplicator.onDemandUpdate();
               }
           };
```

```
        ...
    }
```

再来跟踪 Eureka server 端的代码，在 Maven 的 eureka-core:1.6.2 的 jar 包下。打开 com.netflix.eureka 包，会发现有一个 EurekaBootStrap 的类，BootStrapContext 类在程序启动时具有最先初始化的权限，代码如下：

```
protected void initEurekaServerContext() throws Exception {

 ...//省略代码
  PeerAwareInstanceRegistry registry;
      if (isAws(applicationInfoManager.getInfo())) {
         ...//省略代码，如果是 AWS 的代码
      } else {
         registry = new PeerAwareInstanceRegistryImpl(
                eurekaServerConfig,
                eurekaClient.getEurekaClientConfig(),
                serverCodecs,
                eurekaClient
        );
    }

    PeerEurekaNodes peerEurekaNodes = getPeerEurekaNodes(
           registry,
           eurekaServerConfig,
           eurekaClient.getEurekaClientConfig(),
           serverCodecs,
           applicationInfoManager
    );
 }
```

其中，PeerAwareInstanceRegistryImpl 和 PeerEurekaNodes 两个类从其命名上看，应该和服务注册以及 Eureka Server 高可用有关。先追踪 PeerAwareInstanceRegistryImpl 类，在该类中有一个 register()方法，该方法提供了服务注册，并且将服务注册后的信息同步到其他的 Eureka Server 服务中。代码如下：

```
public void register(final InstanceInfo info, final boolean isReplication) {
      int leaseDuration = Lease.DEFAULT_DURATION_IN_SECS;
      if (info.getLeaseInfo() != null && info.getLeaseInfo().getDurationInSecs() > 0) {
        leaseDuration = info.getLeaseInfo().getDurationInSecs();
      }
      super.register(info, leaseDuration, isReplication);
      replicateToPeers(Action.Register, info.getAppName(), info.getId(), info, null,
isReplication);
    }
```

点击其中的 super.register(info, leaseDuration, isReplication)方法，进入其父类 AbstractInstanceRegistry 可以发现更多细节，注册列表的信息被保存在一个 Map 中。AbstractInstanceRegistry 类的 replicateToPeers()方法用于将注册列表信息同步到其他 Eureka Server 的其他 Peers 节点，追踪代码，发现该方法会循环遍历向所有的 Peers 节点注册，最终执行类 PeerEurekaNodes 的 register()方法，该方法通过执行一个任务向其他节点同步该注册信息，代码如下：

```
public void register(final InstanceInfo info) throws Exception {
     long expiryTime = System.currentTimeMillis() + getLeaseRenewalOf(info);
     batchingDispatcher.process(
            taskId("register", info),
            new InstanceReplicationTask(targetHost, Action.Register, info, null, true) {
               public EurekaHttpResponse<Void> execute() {
                    return replicationClient.register(info);
               }
           },
          expiryTime
      );
  }
```

经过一系列的源码追踪，可以发现 PeerAwareInstanceRegistryImpl 类的 register()方法实现了服务的注册，并且向其他 Eureka Server 的 Peer 节点同步了该注册信息，那么 register()方法被谁调用了呢？在前文中有关 Eureka Client 的分析中可以知道，Eureka Client 是通过 Http 来向 Eureka Server 注册的，那么 Eureka Server 肯定会提供一个服务注册的 API 接口给 Eureka Client 调用，PeerAwareInstanceRegistryImpl 的 register()方法最终肯定会被暴露的 Http 接口所调用。在 IDEA 开发工具中，同时按住“Alt”+鼠标左键（查看某个类被谁调用的快捷键），可以很快定位到 ApplicationResource 类的 addInstance ()方法，即服务注册的接口，其代码如下：

```
@POST
@Consumes({"application/json", "application/xml"})
public Response addInstance(InstanceInfo info,
@HeaderParam(PeerEurekaNode.HEADER_REPLICATION) String isReplication) {

    ...//省略代码
       registry.register(info, "true".equals(isReplication));
       return Response.status(204).build();  // 204 to be backwards compatible
    }
```

5.4.4　Renew 服务续约

服务续约和服务注册非常相似，通过前文中的分析可以知道，服务注册在 Eureka Client 程序启动之后开启，并同时开启服务续约的定时任务。在 eureka-client-1.6.2.jar 的 DiscoveryClient 的类下有 renew()方法，其代码如下：

```
    /**
```

```
     * Renew with the eureka service by making the appropriate REST call
     */
    boolean renew() {
        EurekaHttpResponse<InstanceInfo> httpResponse;
        try {
            httpResponse = eurekaTransport.registrationClient.sendHeartBeat(instanceI
nfo.getAppName(), instanceInfo.getId(), instanceInfo, null);
            logger.debug("{} - Heartbeat status: {}", PREFIX + appPathIdentifier, htt
pResponse.getStatusCode());
            if (httpResponse.getStatusCode() == 404) {
                REREGISTER_COUNTER.increment();
                logger.info("{} - Re-registering apps/{}", PREFIX + appPathIdentifier
, instanceInfo.getAppName());
                return register();
            }
            return httpResponse.getStatusCode() == 200;
        } catch (Throwable e) {
            logger.error("{} - was unable to send heartbeat!", PREFIX + appPathIdenti
fier, e);
            return false;
        }
    }
```

另外，Eureka Server 的续约接口在 eureka-core:1.6.2.jar 的 com.netflix.eureka 包下的 InstanceResource 类下，接口方法为 renewLease()，它是一个 RESTful API 接口。为了减少本章的篇幅，省略了大部分代码的展示。其中有一个 registry.renew()方法，即服务续约，代码如下：

```
@PUT
public Response renewLease(...参数省略）{
    ...  代码省略
    boolean isSuccess=registry.renew(app.getName(),id, isFromReplicaNode);
       ...  代码省略
 }
```

读者可以跟踪 registry.renew 的代码继续深入研究，和追踪服务注册的源码类似，在此不再赘述。另外服务续约有两个参数是可以配置的，即 Eureka Client 发送续约心跳的时间参数和 Eureka Server 在多长时间内没有收到心跳将实例剔除的时间参数。在默认情况下，这两个参数分别为 30 秒和 90 秒，官方的建议是不要修改，如果有特殊需求还是可以调整的，只需要分别在 Eureka Client 和 Eureka Server 的配置文件 application.yml 中加以下的配置：

```
eureka.instance.leaseRenewalIntervalInSeconds
eureka.instance.leaseExpirationDurationInSeconds
```

最后，有关服务注册列表的获取、服务下线和服务剔除的源码不在这里进行跟踪解读，因为与服务注册和续约类似，有兴趣的读者可以自行研究。总的来说，通过阅读源码可以发现，

整体架构与 5.4.2 节的 Eureka 的高可用架构图完全一致。

5.4.5　为什么 Eureka Client 获取服务实例这么慢

（1）Eureka Client 的注册延迟

Eureka Client 启动之后，不是立即向 Eureka Server 注册的，而是有一个延迟向服务端注册的时间。通过跟踪源码，可以发现默认的延迟时间为 40 秒，源码在 eureka-client-1.6.2.jar 的 DefaultEurekaClientConfig 类中，代码如下：

```
public int getInitialInstanceInfoReplicationIntervalSeconds() {
    return configInstance.getIntProperty(
        namespace + INITIAL_REGISTRATION_REPLICATION_DELAY_KEY, 40).get();
}
```

（2）Eureka Server 的响应缓存

Eureka Server 维护每 30 秒更新一次响应缓存，可通过更改配置 eureka.server.responseCacheUpdateIntervalMs 来修改。所以即使是刚刚注册的实例，也不会立即出现在服务注册列表中。

（3）Eureka Client 的缓存

Eureka Client 保留注册表信息的缓存。该缓存每 30 秒更新一次（如前所述）。因此，Eureka Client 刷新本地缓存并发现其他新注册的实例可能需要 30 秒。

（4）LoadBalancer 的缓存

Ribbon 的负载平衡器从本地的 Eureka Client 获取服务注册列表信息。Ribbon 本身还维护了缓存，以避免每个请求都需要从 Eureka Client 获取服务注册列表。此缓存每 30 秒刷新一次（可由 ribbon.ServerListRefreshInterval 配置），所以可能至少需要 30 秒的时间才能使用新注册的实例。

综上因素，一个新注册的实例，默认延迟 40 秒向服务注册中心注册，所以不能马上被 Eureka Server 发现。另外，刚注册的 Eureka Client 也不能立即被其他服务调用，原因是调用方由于各种缓存没有及时获取到最新的服务注册列表信息。

5.4.6　Eureka 的自我保护模式

当有一个新的 Eureka Server 出现时，它尝试从相邻 Peer 节点获取所有服务实例注册表信息。如果从相邻的 Peer 节点获取信息时出现了故障，Eureka Server 会尝试其他的 Peer 节点。如果 Eureka Serve 能够成功获取所有的服务实例信息，则根据配置信息设置服务续约的阈值。在任何时间，如果 Eureka Serve 接收到的服务续约低于为该值配置的百分比（默认为 15 分钟内低于 85%），则服务器开启自我保护模式，即不再剔除注册列表的信息。

这样做的好处在于，如果是 Eureka Server 自身的网络问题而导致 Eureka Client 无法续约，Eureka Client 的注册列表信息不再被删除，也就是 Eureka Client 还可以被其他服务消费。

在默认情况下，Eureka Server 的自我保护模式是开启的，如果需要关闭，则在配置文件添加以下代码：

```
eureka:
  server:
    enable-self-preservation: false
```

5.5 构建高可用的 Eureka Server 集群

在实际的项目中，可能有几十个或者几百个的微服务实例，这时 Eureka Server 承担了非常高的负载。由于 Eureka Server 在微服务架构中有着举足重轻的作用，所以需要对 Eureka Server 进行高可用集群。

本节以实战的方式介绍如何构建高可用的 Eureka Server 集群，本节的例子在 5.2 节案例的基础上进行改造。

首先更改 eureka-server 的配置文件 application.yml，在该配置文件中采用多 profile 的格式，具体代码如下：

```
---
spring:
   profiles: peer1
server:
   port: 8761
eureka:
   instance:
      hostname: peer1
   client:
      serviceUrl:
         defaultZone: http://peer2:8762/eureka/

---
spring:
   profiles: peer2
server:
   port: 8762
eureka:
   instance:
      hostname: peer2
   client:
      serviceUrl:
         defaultZone: http://peer1:8761/eureka/
```

在上述代码中，定义了两个 profile 文件，分别为 peer1 和 peer2，它们的 hostname 分别为 peer1 和 peer2。在实际开发中，可能是具体的服务器 IP 地址，它们的端口分别为 8761 和 8762。

因为是在本地搭建 Eureka Server 集群，所以需要修改本地的 host。Windows 系统的电脑在 C:/windows/systems/drivers/etc/hosts 中修改，Mac 系统的电脑通过终端 vim/etc/hosts 进行编辑修改，修改内容如下：

```
127.0.0.1 peer1
127.0.0.1 peer2
```

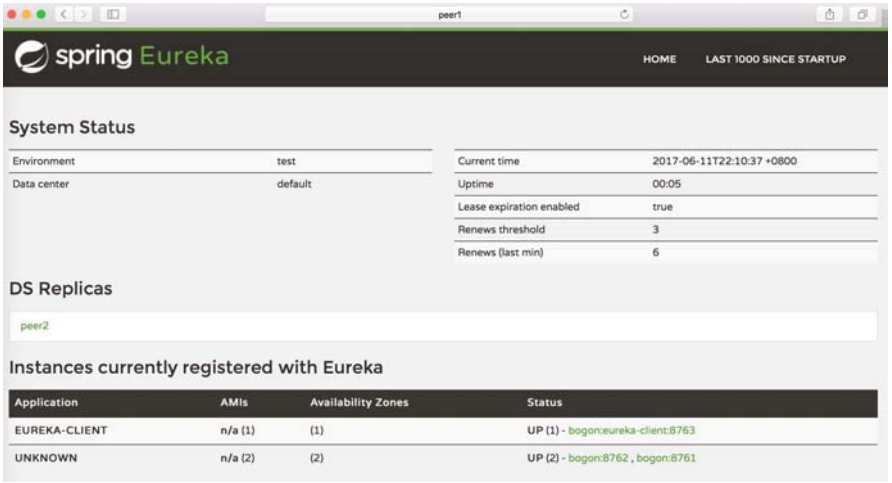
通过 Maven 编译工程，Maven 命令为 mvn clean package。编译完成后，在工程的目录下会有一个 target 文件夹，进入该文件夹，可以发现已经生成了一个 eureka-server-0.0.1-SNAPSHOT.jar 的 jar 包。通过 java–jar 的方式启动工程，并通过 spring.profiles.active 指定启动的配置文件。在本案例中需要启动两个 Eureka Server 实例，它们的 spring.profiles.active 配置文件分别为 peer1 和 pee2。启动实例的命令如下：

```
java –jar eureka-server-0.0.1-SNAPSHOT.jar
java –jar eureka-server-0.0.1-SNAPSHOT.jar
- -spring.profiles.active=peer1
- -spring.profiles.active=peer2
```

修改 eureka-client 的配置文件 bootstrap.yml，修改其端口为 8763，并且仅向 8761 的 Eureka Server 注册，代码如下：

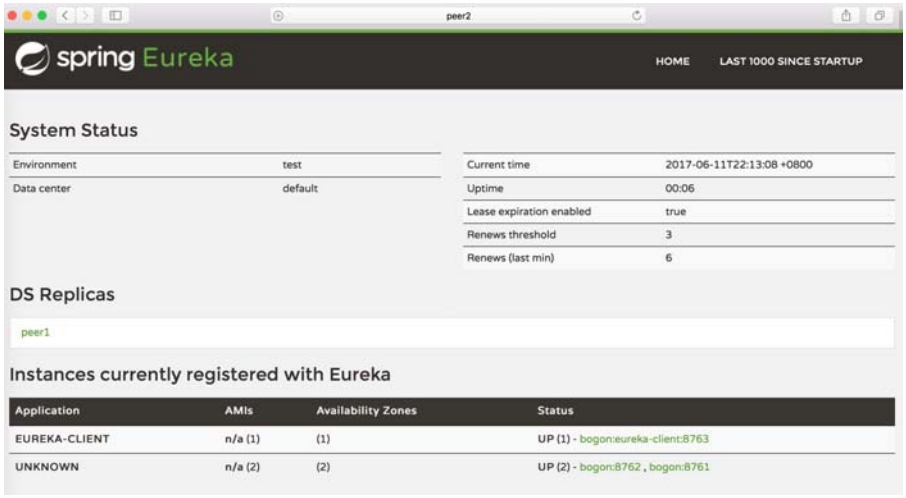
```
server:
  port: 8763
spring:
  application:
    name: eureka-client
eureka:
  client:
    serviceUrl:
      defaultZone: http://peer1:8761/eureka/
```

启动 eureka-client 工程，访问 http://peer1:8761/可以发现，eureka-client 已向 peer1 节点的 Eureka Server 注册了，并且在 DS Replicas 选项中显示了节点 peer2，界面如图 5-5 所示。



▲图 5-5　peer1 节点注册了 Eureka-client

这时 eureka-client 工程的配置文件中并没有指定向 peer2 的节点 Eureka Server 注册。访问 Eureka Server 的节点 peer2 的主界面，界面的 Url 地址为 http://peer2:8762/。节点 peer2 的主界面显示 Eureka Client 已经向 peer2 节点注册，可见 peer1 的注册列表信息已经同步到了 peer2 节点，节点 peer2 的主界面展示如图 5-6 所示。



▲图 5-6　peer1 节点的注册列表信息同步到了 peer2

5.6 总结

本章全面介绍了 Eureka 的相关内容，分析了为什么要选择 Eureka 作为服务注册和发现的组件。接着，以案例的形式讲述了 Eureka Client 是如何向 Eureka Server 注册的，并从源码的角度去深入理解 Eureka。最后，通过案例来讲解如何构建高可用的 Eureka Server。

第 6 章　负载均衡 Ribbon

上一章讲述了服务注册和发现组件 Eureka，同时追踪源码深入讲解了 Eureka 的机制，最后通过案例讲解了如何构建高可用的 Eureka Server。本章讲解如何使用 RestTemplate 和 Ribbon 相结合作为服务消费者去消费服务，同时从源码的角度来深入讲解 Ribbon。

6.1 RestTemplate 简介

RestTemplate 是 Spring Resources 中一个访问第三方 RESTful API 接口的网络请求框架。RestTemplate 的设计原则和其他 Spring Template（例如 JdbcTemplate、JmsTemplate）类似，都是为执行复杂任务提供了一个具有默认行为的简单方法。

RestTemplate 是用来消费 REST 服务的，所以 RestTemplate 的主要方法都与 REST 的 Http 协议的一些方法紧密相连，例如 HEAD、GET、POST、PUT、DELETE 和 OPTIONS 等方法，这些方法在 RestTemplate 类对应的方法为 headForHeaders()、getForObject()、postForObject()、put()和 delete()等。

举例说明，写一个 RestTestController 类，获取 https://www.baidu.com/的网页 Html 代码。首先在 RestTestController 类上加@RestController 注解，开启 RestController 的功能。通过 RestTemplate 的 getForObject()方法可以获取 https://www.baidu.com/的网页 Html 代码，并在 API 接口“/testRest”返回该网页的 Html 字符串。代码如下：

```
@RestController
public class RestTestController {
   @GetMapping("/testRest")
   public String testRest(){
      RestTemplate restTemplate=new RestTemplate();
      return
restTemplate.getForObject("https://www.baidu.com/",String.class);
   }
}
```

RestTemplate 支持常见的 Http 协议的请求方法，例如 Post、Put、Delete 等，所以用 RestTemplate 很容易构建 RESTful API。在上面的例子中，RestTemplate 用 Get 方法获取

https://www.baidu.com 网页的 Html 字符串。RestTemplate 的使用很简单，它支持 Xml、JSON 数据格式，默认实现了序列化，可以自动将 JOSN 字符串转换为实体。例如以下代码可以将返回的 JSON 字符串转换成一个 User 对象。

```
User user=restTemplate.getForObject("https://www.xxx.com/",User.class);
```

6.2 Ribbon 简介

负载均衡是指将负载分摊到多个执行单元上，常见的负载均衡有两种方式。一种是独立进程单元，通过负载均衡策略，将请求转发到不同的执行单元上，例如 Ngnix。另一种是将负载均衡逻辑以代码的形式封装到服务消费者的客户端上，服务消费者客户端维护了一份服务提供者的信息列表，有了信息列表，通过负载均衡策略将请求分摊给多个服务提供者，从而达到负载均衡的目的。

Ribbon 是 Netflix 公司开源的一个负载均衡的组件，它属于上述的第二种方式，是将负载均衡逻辑封装在客户端中，并且运行在客户端的进程里。Ribbon 是一个经过了云端测试的 IPC 库，可以很好地控制 HTTP 和 TCP 客户端的负载均衡行为。

在 Spring Cloud 构建的微服务系统中，Ribbon 作为服务消费者的负载均衡器，有两种使用方式，一种是和 RestTemplate 相结合，另一种是和 Feign 相结合。Feign 已经默认集成了 Ribbon，关于 Feign 的内容将会在下一章进行详细讲解。

Ribbon 有很多子模块，但很多模块没有用于生产环境，目前 Netflix 公司用于生产环境的 Ribbon 子模块如下。

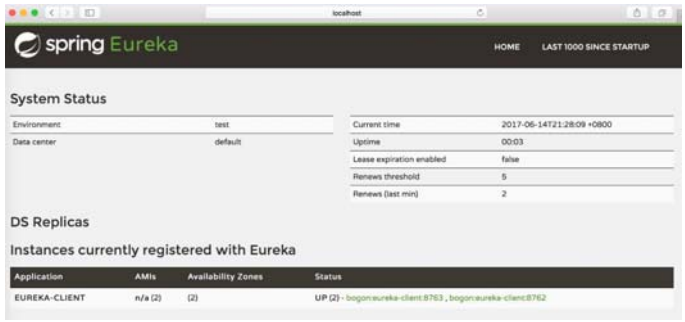
- ribbon-loadbalancer：可以独立使用或与其他模块一起使用的负载均衡器 API。
- ribbon-eureka：Ribbon 结合 Eureka 客户端的 API，为负载均衡器提供动态服务注册列表信息。
- ribbon-core：Ribbon 的核心 API。

6.3 使用 RestTemplate 和 Ribbon 来消费服务

本案例是在 5.2 节案例的基础上进行改造的，先回顾一下 5.2 节中的代码结构，它包括一个服务注册中心 eureka-server、一个服务提供者 eureka-client。eureka-client 向 eureka-server 注册服务，并且 eureka-client 提供了一个“/hi”API 接口，用于提供服务。

启动 eureka-server，端口为 8761。启动两个 eureka-client 实例，端口分别为 8762 和 8763。启动完成后，在浏览器上访问 http://localhost:8761/，浏览器显示 eureka-client 的两个实例已经成功向服务注册中心注册，它们的端口分别为 8762 和 8763，如图 6-1 所示。

在 5.2 节的工程基础之上，再创建一个 Module 工程，取名为 eureka-ribbon-client，其作为服务消费者，通过 RestTemplate 来远程调用 eureka-client 服务 API 接口的“/hi”，即消费服务。



▲图 6-1　http://localhost:8761/页面

创建完成 eureka-ribbon-client 的 Module 工程之后，在其 pom 文件中引入相关的依赖，包括继承了主 Maven 工程的 pom 文件，引入了 Eureka Client 的起步依赖 spring-cloud-starter-eureka、Ribbon 的起步依赖 spring-cloud-starter-ribbon，以及 Web 的起步依赖 spring-boot-starter- web，代码如下：

```
  <parent>
        <groupId>com.forezp</groupId>
        <artifactId>chapter5-2</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-ribbon</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
  </dependencies>
```

在工程的配置文件 application.yml 做程序的相关配置，包括指定程序名为 eureka-ribbon-client，程序的端口号为 8764，服务的注册地址为 http://localhost:8761/eureka/，代码如下：

```
spring:
  application:
    name: eureka-ribbon-client
server:
  port: 8764
eureka:
```

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

另外，作为 Eureka Client 需要在程序的入口类加上注解@EnableEurekaClient 开启 Eureka Client 功能，代码如下：

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

写一个 RESTful API 接口，在该 API 接口内部需要调用 eureka-client 的 API 接口“/hi”，即服务消费。由于 eureka-client 为两个实例，它们的端口为 8762 和 8763。在调用 eureka-client 的 API 接口“/hi”时希望做到轮流访问这两个实例，这时就需要将 RestTemplate 和 Ribbon 相结合，进行负载均衡。

通过查阅官方文档，可以知道如何将它们结合在一起，只需要在程序的 IoC 容器中注入一个 restTemplate 的 Bean，并在这个 Bean 上加上@LoadBalanced 注解，此时 RestTemplate 就结合了 Ribbon 开启了负载均衡功能，代码如下：

```
@Configuration
public class RibbonConfig {
   @Bean
   @LoadBalanced
   RestTemplate restTemplate() {
      return new RestTemplate();
   }
}
```

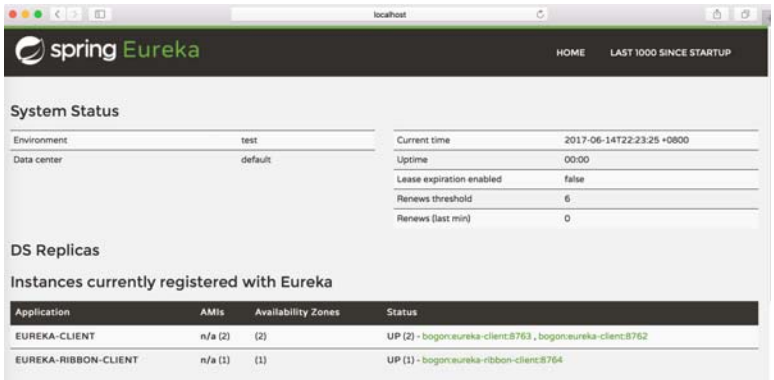
写一个 RibbonService 类，在该类的 hi()方法用 restTemplate 调用 eureka-client 的 API 接口，此时 Uri 上不需要使用硬编码（例如 IP 地址），只需要写服务名 eureka-client 即可，代码如下：

```
@Service
public class RibbonService {
   @Autowired
   RestTemplate restTemplate;
   public String hi(String name) {
      return restTemplate.getForObject("http://eureka-client/hi?name="+name,String.class);
   }
}
```

写一个 RibbonController 类，为该类加上@RestController 注解，开启 RestController 的功能，写一个“/hi”Get 方法的接口，调用 RibbonService 类的 hi()方法，代码如下：

```
@RestController
public class RibbonController {

   @Autowired
   RibbonService ribbonService;
   @GetMapping("/hi")
   public String hi(@RequestParam(required = false,defaultValue =
                "forezp") String name){
       return ribbonService.hi(name);
   } }
```

启动 eureka-ribbon-client 工程，在浏览器上访问 http://localhost:8761，显示的 Eureka Server 的主界面如图 6-2 所示。在主界面上发现有两个服务被注册，分别为 eureka-client 和 eureka-ribbon-client，其中 eureka-client 有两个实例，端口为 8762 和 8763，而 eureka-ribbon-client 的端口为 8764。



▲图 6-2　http://localhost:8761 界面

在浏览器上多次访问 http://localhost:8764/hi?name=forezp，浏览器会轮流显示如下内容：

```
hi forezp,i am from port:8762
hi forezp,i am from port:8763
```

这时可以发现，当访问 http://localhost:8764/hi?name=forezp 的 API 接口时，负载均衡器起了作用，负载均衡器会轮流地请求 eureka-client 的两个实例中的“/hi”API 接口。

6.4 LoadBalancerClient 简介

负载均衡器的核心类为 LoadBalancerClient，LoadBalancerClient 可以获取负载均衡的服务提供者的实例信息。为了演示，在 RibbonController 重新写一个接口“/testRibbon”，通过 LoadBalancerClient 去选择一个 eureka-client 的服务实例的信息，并将该信息返回，代码如下：

```
@RestController
public class RibbonController {
```

```
    ....//代码省略
    @Autowired
    private LoadBalancerClient loadBalancer;
    @GetMapping("/testRibbon")
    public String  testRibbon() {
       ServiceInstance instance = loadBalancer.choose("eureka-client");
        return instance.getHost()+":"+instance.getPort();
    }
}
```

重新启动工程，在浏览器上多次访问 http://localhost:8764/testRibbon，浏览器会轮流显示如下内容：

```
localhost:8762
localhost:8763
```

可见，LoadBalancerClient 的 choose("eureka-client")方法可以轮流得到 eureka-client 的两个服务实例的信息。

那么负载均衡器是怎么获取到这些客户端的信息的呢？查看官方文档可以知道，负载均衡器 LoadBalancerClient 是从 Eureka Client 获取服务注册列表信息的，并将服务注册列表信息缓存了一份。在 LoadBalancerClient 调用 choose()方法时，根据负载均衡策略选择一个服务实例的信息，从而进行了负载均衡。LoadBalancerClient 也可以不从 Eureka Client 获取注册列表信息，这时需要自己维护一份服务注册列表信息。

为了进一步讲解，在 6.3 节中工程的基础之上再创建一个工程，取名为 ribbon-client。ribbon-client 工程的 pom 文件同 eureka-ribbon-client 工程的类似，在 ribbon-client 工程的程序的启动类 RibbonClientApplication 加上@SpringBootApplication 的注解，开启 Spring Boot 的基本功能。

在 ribbon-client 工程的配置文件 application.yml 中，通过配置 ribbon.eureka.enable 为 false 来禁止调用 Eureka Client 获取注册列表。在配置文件 application.yml 中有一个程序名为 stores 的服务，有两个不同 Url 地址（例如 example.com 和 google.com）的服务实例，通过 stores.ribbon.listOfServers 来配置这些服务实例的 Url，代码如下：

```
stores:
  ribbon:
   listOfServers: example.com,google.com
ribbon:
  eureka:
   enabled: false
server:
  port: 8769
```

新建一个 RestController 类，创建一个 API 接口“/testRibbon”。在 RestController 类注入 LoadBalancerClient，通过 LoadBalancerClient 的 choose()方法获取服务实例的信息，代码如下：

```
@RestController
public class RibbonController {
```

```
    @Autowired
    private LoadBalancerClient loadBalancer;

    @GetMapping("/testRibbon")
    public String  testRibbon() {
       ServiceInstance instance = loadBalancer.choose("stores");
       return instance.getHost()+":"+instance.getPort();
    }
}
```

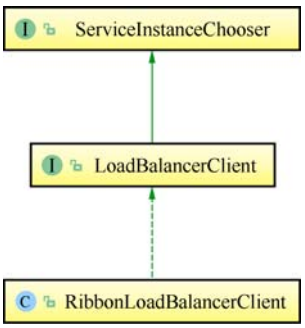
启动工程，在浏览器上多次访问 http://localhost:8769/testRibbon，浏览器会交替出现以下内容：

```
example.com:80
google.com:80
```

现在我们可以知道，在 Ribbon 中的负载均衡客户端为 LoadBalancerClient。在 Spring Cloud 项目中，负载均衡器 Ribbon 会默认从 Eureka Client 的服务注册列表中获取服务的信息，并缓存一份。根据缓存的服务注册列表信息，可以通过 LoadBalancerClient 来选择不同的服务实例，从而实现负载均衡。如果禁止 Ribbon 从 Eureka 获取注册列表信息，则需要自己去维护一份服务注册列表信息。根据自己维护服务注册列表的信息，Ribbon 也可以实现负载均衡。

6.5 源码解析 Ribbon

为了深入理解 Ribbon，现在从源码的角度来讲解 Ribbon，看它如何和 Eureka 相结合，并如何和 RestTemplate 相结合来做负载均衡。首先，跟踪 LoadBalancerClient 的源码，它是一个接口类，继承了 ServiceInstanceChooser，它的实现类为 RibbonLoadBalanceClient，它们之间的关系如图 6-3 所示。



▲图 6-3　LoadBalancerClient 的父类和实现类

LoadBalancerClient 是一个负载均衡的客户端，有如下 3 种方法。其中有 2 个 excute()方法，均用来执行请求，reconstructURI()用于重构 Url，代码如下：

```
public interface LoadBalancerClient extends ServiceInstanceChooser {

   <T> T execute(String serviceId, LoadBalancerRequest<T> request) throws IOException;
   <T> T execute(String serviceId, ServiceInstance serviceInstance, LoadBalancerRequest<T>
request) throws IOException;
   URI reconstructURI(ServiceInstance instance, URI original);
}
```

ServiceInstanceChooser 接口有一个方法用于根据 serviceId 获取 ServiceInstance，即通过服务名来选择服务实例，代码如下：

```
public interface ServiceInstanceChooser {
  ServiceInstance choose(String serviceId);
}
```

LoadBalancerClient 的实现类为 RibbonLoadBalancerClient。RibbonLoadBalancerClient 是一个非常重要的类，最终的负载均衡的请求处理由它来执行。RibbonLoadBalancerClient 的部分源码如下：

```
public class RibbonLoadBalancerClient implements LoadBalancerClient {
...//省略代码
@Override
    public ServiceInstance choose(String serviceId) {
        Server server = getServer(serviceId);
        if (server == null) {
            return null;
        }
        return new RibbonServer(serviceId, server, isSecure(server, serviceId),
                serverIntrospector(serviceId).getMetadata(server));
    }
    protected Server getServer(String serviceId) {
        return getServer(getLoadBalancer(serviceId));
    }
    protected Server getServer(ILoadBalancer loadBalancer) {
        if (loadBalancer == null) {
            return null;
        }
        return loadBalancer.chooseServer("default");
    }
    protected ILoadBalancer getLoadBalancer(String serviceId) {
        return this.clientFactory.getLoadBalancer(serviceId);
    }
}
```

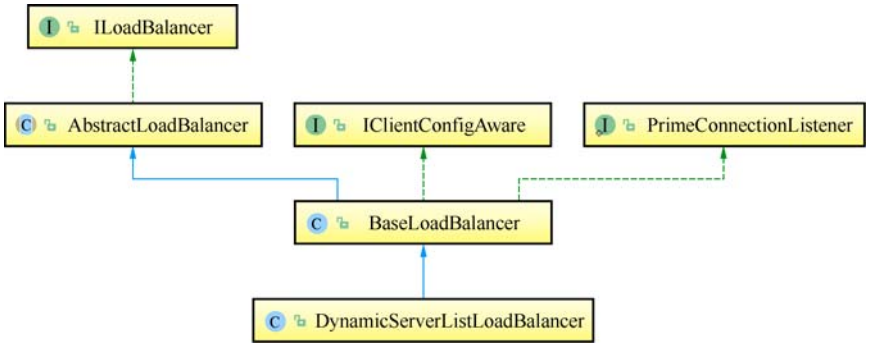
在 RibbonLoadBalancerClient 的源码中，choose()方法用于选择具体服务实例。该方法通过 getServer()方法去获取实例，经过源码跟踪，最终交给 ILoadBalancer 类去选择服务实例。

ILoadBalancer 在 ribbon-loadbalancer 的 jar 包下，ILoadBalancer 是一个接口，该接口定义了一系列实现负载均衡的方法，源码如下：

```
public interface ILoadBalancer {
   public void addServers(List<Server> newServers);
   public Server chooseServer(Object key);
   public void markServerDown(Server server);
   public List<Server> getReachableServers();
   public List<Server> getAllServers();
}
```

其中，addServers()方法用于添加一个 Server 集合，chooseServer()方法用于根据 key 去获取 Server，markServerDown()方法用于标记某个服务下线，getReachableServers()获取可用的 Server 集合，getAllServers()获取所有的 Server 集合。

ILoadBalancer 的子类为 BaseLoadBalancer，BaseLoadBalancer 的实现类为 DynamicServerListLoadBalancer，三者之间的关系如图 6-4 所示。



▲图 6-4　DynamicServerListLoadBalancer 与其接口类的关系

查看 DynamicServerListLoadBalancer 类的源码，DynamicServerListLoadBalancer 需要配置 IClientConfig、IRule、IPing、ServerList、ServerListFilter 和 ILoadBalancer。查看 BaseLoadBalancer 类的源码，在默认的情况下，实现了如下配置。

- IClientConfig ribbonClientConfig：DefaultClientConfigImpl。
- IRule ribbonRule：RoundRobinRule。
- IPing ribbonPing：DummyPing。
- ServerList ribbonServerList：ConfigurationBasedServerList。
- ServerListFilter ribbonServerListFilter：ZonePreferenceServerListFilter。
- ILoadBalancer ribbonLoadBalancer：ZoneAwareLoadBalancer。

IClientConfig 用于配置负载均衡的客户端，IClientConfig 的默认实现类为 DefaultClientConfigImpl。

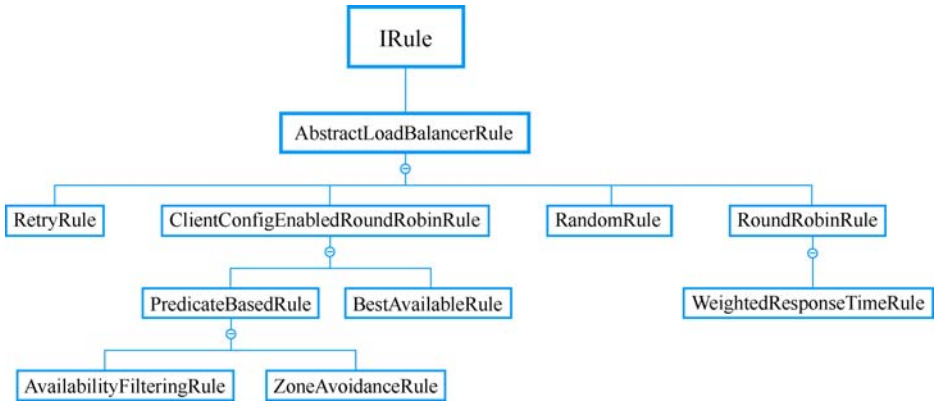
IRule 用于配置负载均衡的策略，IRule 有 3 个方法，其中 choose()是根据 key 来获取 server

实例的，setLoadBalancer()和 getLoadBalancer()是用来设置和获取 ILoadBalancer 的，它的源码如下：

```
public interface IRule{
   public Server choose(Object key);
   public void setLoadBalancer(ILoadBalancer lb);
   public ILoadBalancer getLoadBalancer();
}
```

IRule 有很多默认的实现类，这些实现类根据不同的算法和逻辑来处理负载均衡的策略。IRule 的默认实现类有以下 7 种。在大多数情况下，这些默认的实现类是可以满足需求的，如果有特殊的需求，可以自己实现。IRule 和其实现类之间的关系如图 6-5 所示。

- BestAvailableRule：选择最小请求数。
- ClientConfigEnabledRoundRobinRule：轮询。
- RandomRule：随机选择一个 server。
- RoundRobinRule：轮询选择 server。
- RetryRule：根据轮询的方式重试。
- WeightedResponseTimeRule：根据响应时间去分配一个 weight ，weight 越低，被选择的可能性就越低。
- ZoneAvoidanceRule：根据 server 的 zone 区域和可用性来轮询选择。



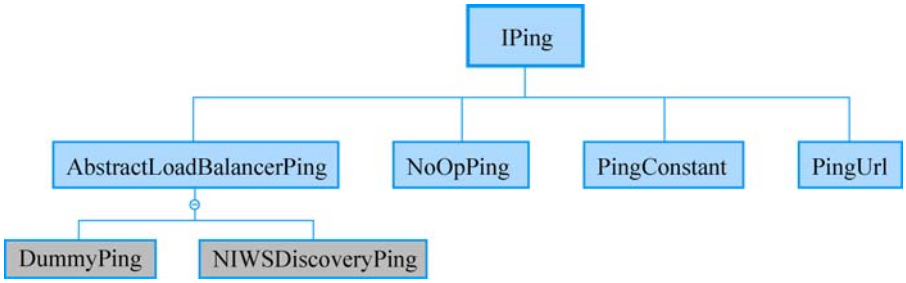
▲图 6-5 IRule 及其实现类

IPing 用于向 server 发送“ping”，来判断该 server 是否有响应，从而判断该 server 是否可用。它有一个 isAlive()方法，源码如下：

```
public interface IPing {
   public boolean isAlive(Server server);
}
```

IPing 的实现类有 PingUrl、PingConstant、NoOpPing、DummyPing 和 NIWSDiscoveryPing。

它们之间的关系如图 6-6 所示。



▲图 6-6　IPing 及其实现类

- PingUrl：真实地去 ping 某个 Url，判断其是否可用。
- PingConstant：固定返回某服务是否可用，默认返回 true，即可用。
- NoOpPing：不去 ping，直接返回 true，即可用。
- DummyPing：直接返回 true，并实现了 initWithNiwsConfig 方法。
- NIWSDiscoveryPing：根据 DiscoveryEnabledServer 的 InstanceInfo 的 InstanceStatus 去判断，如果为 InstanceStatus.UP，则可用，否则不可用。

ServerList 是定义获取所有 server 的注册列表信息的接口，它的代码如下：

```
public interface ServerList<T extends Server> {
   public List<T> getInitialListOfServers();
   public List<T> getUpdatedListOfServers();
}
```

ServerListFilter 接口定义了可根据配置去过滤或者特性动态地获取符合条件的 server 列表的方法，代码如下：

```
public interface ServerListFilter<T extends Server> {
   public List<T> getFilteredListOfServers(List<T> servers);
}
```

阅读 DynamicServerListLoadBalancer 的源码，DynamicServerListLoadBalancer 的构造函数中有一个 initWithNiwsConfig()方法。在该方法中经过一系列的初始化配置，最终执行了 restOfInit()方法。DynamicServerListLoadBalancer 的部分源码如下：

```
public DynamicServerListLoadBalancer(IClientConfig clientConfig) {
      initWithNiwsConfig(clientConfig);
 }
@Override
  public void initWithNiwsConfig(IClientConfig clientConfig) {
       try {
      …//省略代码
       this.serverListUpdater = (ServerListUpdater) ClientFactory
```

```
        .instantiateInstanceWithClientConfig(serverListUpdaterClassName,clientConfig);
          restOfInit(clientConfig);
      } catch (Exception e) {
         …//省略代码
      }
  }
```

在 restOfInit()方法中，有一个 updateListOfServers() 的方法，该方法是用来获取所有的 ServerList 的。

```
void restOfInit(IClientConfig clientConfig) {
      …//省略代码
      updateListOfServers();
     …//省略代码
   }
```

进一步跟踪 updateListOfServers()方法的源码，最终由 serverListImpl.getUpdatedListOfServers() 获取所有的服务列表，代码如下：

```
 public void updateListOfServers() {
      List<T> servers = new ArrayList<T>();
      if (serverListImpl != null) {
         servers = serverListImpl.getUpdatedListOfServers();
      }
      updateAllServerList(servers);
}
```

而 serverListImpl 是 ServerList 接口的具体实现类。跟踪源码，ServerList 的实现类为 DiscoveryEnabledNIWSServerList，这个类在 ribbon-eureka.jar 的 com.netflix.niws.loadbalancer 包下。其中，DiscoveryEnabledNIWSServerList 有 getInitialListOfServers()和 getUpdatedListOfServers()方法，具体代码如下：

```
    @Override
    public List<DiscoveryEnabledServer> getInitialListOfServers(){
        return obtainServersViaDiscovery();
    }
    @Override
    public List<DiscoveryEnabledServer> getUpdatedListOfServers(){
        return obtainServersViaDiscovery();
    }
```

继续跟踪源码，obtainServersViaDiscovery() 方法是根据 eurekaClientProvider.get()方法来获取 EurekaClient 的，再根据 EurekaClient 来获取服务注册列表信息，代码如下：

```
private List<DiscoveryEnabledServer> obtainServersViaDiscovery() {
      …//省略代码
```

```
        EurekaClient eurekaClient = eurekaClientProvider.get();
        if (vipAddresses!=null){
           for (String vipAddress : vipAddresses.split(",")) {

              for (InstanceInfo ii : listOfInstanceInfo) {
                  if (ii.getStatus().equals(InstanceStatus.UP)) {

                      DiscoveryEnabledServer  des  =  new  DiscoveryEnabledServer(ii,
isSecure, shouldUseIpAddr);
                      des.setZone(DiscoveryClient.getZone(ii));
                      serverList.add(des);
                  …//省略代码
              }
           }
       }
       return serverList;
    }
```

其中，eurekaClientProvider 的实现类是 LegacyEurekaClientProvider，LegacyEurekaClientProvider 是一个获取 eurekaClient 实例的类，其代码如下：

```
class LegacyEurekaClientProvider implements Provider<EurekaClient> {
   private volatile EurekaClient eurekaClient;
   @Override
   public synchronized EurekaClient get() {
      if (eurekaClient == null) {
         eurekaClient = DiscoveryManager.getInstance().getDiscoveryClient();
      }
      return eurekaClient;
   }
}
```

EurekaClient 的实现类为 DiscoveryClient，在上一章已经分析了。DiscoveryClient 具有服务注册、获取服务注册列表等功能。

由此可见，负载均衡器是从 Eureka Client 获取服务列表信息的，并根据 IRule 的策略去路由，根据 IPing 去判断服务的可用性。

那么现在还有一个问题，负载均衡器每隔多长时间从 Eureka Client 获取注册信息呢？

在 BaseLoadBalancer 类的源码中，在 BaseLoadBalancer 的构造方法开启了一个 PingTask 任务，代码如下：

```
 public BaseLoadBalancer(String name, IRule rule, LoadBalancerStats stats,
        IPing ping, IPingStrategy pingStrategy) {
     ...//代码省略
     setupPingTask();
     ...//代码省略
```

```
    }
```

在 setupPingTask()的具体代码逻辑里，开启了 ShutdownEnabledTimer 的 PingTask 任务，在默认情况下，变量 pingIntervalSeconds 的值为 10，即每 10 秒向 EurekaClient 发送一次心跳“ping”。

```
    void setupPingTask() {
        if (canSkipPing()) {
            return;
        }
        if (lbTimer != null) {
            lbTimer.cancel();
        }
        lbTimer = new ShutdownEnabledTimer("NFLoadBalancer-PingTimer-" + name,
               true);
        lbTimer.schedule(new PingTask(), 0, pingIntervalSeconds * 1000);
        forceQuickPing();
    }
```

查看 PingTask 的源码，PingTask 创建了一个 Pinger 对象，并执行了 runPinger()方法。

```
class PingTask extends TimerTask {
       public void run() {
          try {
            new Pinger(pingStrategy).runPinger();
          } catch (Exception e) {
            logger.error("LoadBalancer [{}]: Error pinging", name, e);
          }
       }
  }
```

查看 Pinger 的 runPinger()方法，最终根据 pingerStrategy.pingServers(ping, allServers)来获取服务的可用性，如果该返回结果与之前相同，则不向 EurekaClient 获取注册列表；如果不同，则通知 ServerStatusChangeListener 服务注册列表信息发生了改变，进行更新或者重新拉取，代码如下：

```
public void runPinger() throws Exception {
          if (!pingInProgress.compareAndSet(false, true)) {
             return;
          }
          Server[] allServers = null;
          boolean[] results = null;
          Lock allLock = null;
          Lock upLock = null;
          try {
              allLock = allServerLock.readLock();
```

```
                allLock.lock();
                allServers = allServerList.toArray(new Server[allServerList.size()]);
                allLock.unlock();
                int numCandidates = allServers.length;
                results = pingerStrategy.pingServers(ping, allServers);
                final List<Server> newUpList = new ArrayList<Server>();
                final List<Server> changedServers = new ArrayList<Server>();
                for (int i = 0; i < numCandidates; i++) {
                   boolean isAlive = results[i];
                   Server svr = allServers[i];
                   boolean oldIsAlive = svr.isAlive();

                   svr.setAlive(isAlive);

                   if (oldIsAlive != isAlive) {
                      changedServers.add(svr);
                   }
                   if (isAlive) {
                      newUpList.add(svr);
                   }
                }
                upLock = upServerLock.writeLock();
                upLock.lock();
                upServerList = newUpList;
                upLock.unlock();
                notifyServerStatusChangeListener(changedServers);
             } finally {
                pingInProgress.set(false);
             }
          }
```

由此可见，LoadBalancerClient 是在初始化时向 Eureka 获取服务注册列表信息，并且每 10 秒向 EurekaClient 发送“ping”，来判断服务的可用性。如果服务的可用性发生了改变或者服务数量和之前的不一致，则更新或者重新拉取。LoadBalancerClient 有了这些服务注册列表信息，就可以根据具体的 IRule 的策略来进行负载均衡。

最后，回到问题的本身，为什么在 RestTemplate 类的 Bean 上加一个@LoadBalance 注解就可以使用 Ribbon 的负载均衡呢？

全局搜索（IDEA 的快捷键为“Ctrl”+“Shift”+“F”）查看有哪些类用到了 @LoadBalanced 注解。通过搜索，可以发现 LoadBalancerAutoConfiguration 类（LoadBalancer 自动配置类）使用到了该注解，LoadBalancerAutoConfiguration 类的代码如下：

```
@Configuration
@ConditionalOnClass(RestTemplate.class)
@ConditionalOnBean(LoadBalancerClient.class)
```

```
@EnableConfigurationProperties(LoadBalancerRetryProperties.class)
public class LoadBalancerAutoConfiguration {

   @LoadBalanced
   @Autowired(required = false)
   private List<RestTemplate> restTemplates = Collections.emptyList();
   }
   @Bean
   public SmartInitializingSingleton loadBalancedRestTemplateInitializer(
           final List<RestTemplateCustomizer> customizers) {
      return new SmartInitializingSingleton() {
           @Override
           public void afterSingletonsInstantiated() {
               for (RestTemplate restTemplate : LoadBalancerAutoConfiguration. this.
restTemplates) {
                   for (RestTemplateCustomizer customizer : customizers) {
                       customizer.customize(restTemplate);
                  }
              }
           }
       };
   }
   @Configuration
   @ConditionalOnMissingClass("org.springframework.retry.support.RetryTemplate")
   static class LoadBalancerInterceptorConfig {
       @Bean
       public LoadBalancerInterceptor ribbonInterceptor(
               LoadBalancerClient loadBalancerClient,
               LoadBalancerRequestFactory requestFactory) {
           return new LoadBalancerInterceptor(loadBalancerClient, requestFactory);
       }

       @Bean
       @ConditionalOnMissingBean
       public RestTemplateCustomizer restTemplateCustomizer(
               final LoadBalancerInterceptor loadBalancerInterceptor) {
           return new RestTemplateCustomizer() {
               @Override
               public void customize(RestTemplate restTemplate) {
                   List<ClientHttpRequestInterceptor> list = new ArrayList<>(
                          restTemplate.getInterceptors());
                   list.add(loadBalancerInterceptor);
                   restTemplate.setInterceptors(list);
               }
           };
       }
```

```
    }
}
```

在 LoadBalancerAutoConfiguration 类中，首先维护了一个被@LoadBalanced 修饰的 RestTemplate 对象的 List。在初始化的过程中，通过调用 customizer.customize(restTemplate)方法来给 RestTemplate 增加拦截器 LoadBalancerInterceptor。LoadBalancerInterceptor 用于实时拦截，在 LoadBalancerInterceptor 中实现了负载均衡的方法。LoadBalancerInterceptor 类的拦截方法的代码如下：

```
@Override
   public ClientHttpResponse intercept(final HttpRequest request, final byte[] body,
          final ClientHttpRequestExecution execution) throws IOException
{
       final URI originalUri = request.getURI();
       String serviceName = originalUri.getHost();
       return this.loadBalancer.execute(serviceName, requestFactory.createRequest(request,
body, execution));
}
```

综上所述，Ribbon 的负载均衡主要是通过 LoadBalancerClient 来实现的，而 LoadBalancerClient 具体交给了 ILoadBalancer 来处理，ILoadBalancer 通过配置 IRule、IPing 等，向 EurekaClient 获取注册列表的信息，默认每 10 秒向 EurekaClient 发送一次“ping”，进而检查是否需要更新服务的注册列表信息。最后，在得到服务注册列表信息后，ILoadBalancer 根据 IRule 的策略进行负载均衡。

而 RestTemplate 加上@LoadBalance 注解后，在远程调度时能够负载均衡，主要是维护了一个被@LoadBalance 注解的 RestTemplate 列表，并给该列表中的 RestTemplate 对象添加了拦截器。在拦截器的方法中，将远程调度方法交给了 Ribbon 的负载均衡器 LoadBalancerClient 去处理，从而达到了负载均衡的目的。

第 7 章　声明式调用 Feign

在上一章中，讲解了如何使用 RestTemplate 来消费服务，如何结合 Ribbon 在消费服务时做负载均衡。本章将全面讲解 Feign，包括如何使用 Feign 来远程调度其他服务、FeignClient 的各项详细配置，并从源码的角度深入讲解 Feign。

Feign 受 Retrofit、JAXRS-2.0 和 WebSocket 的影响，采用了声明式 API 接口的风格，将 Java Http 客户端绑定到它的内部。Feign 的首要目标是将 Java Http 客户端调用过程变得简单。Feign 的源码地址：https://github.com/OpenFeign/feign。

7.1 写一个 Feign 客户端

本章的案例基于上一章的案例，在 6.3 节的工程基础之上进行改造。本节的案例讲解了如何使用 Feign 进行远程调用。

新建一个 Spring Boot 的 Module 工程，取名为 eureka-feign-client。首先，在工程的 pom 文件中加入相关的依赖，包括继承了主 Maven 工程的 pom 文件、Feign 的起步依赖 spring-cloud-starter-feig、Eureka Client 的起步依赖 spring-cloud-starter-eureka、Web 功能的起步依赖 spring-boot-starter-web，以及 Spring Boot 测试的起步依赖 spring-boot-starter-test，代码如下：

```
<parent>
     <groupId>com.forezp</groupId>
     <artifactId>chapter5-2</artifactId>
     <version>1.0-SNAPSHOT</version>
</parent>
<dependencies>
     <dependency>
          <groupId>org.springframework.cloud</groupId>
          <artifactId>spring-cloud-starter-feign</artifactId>
     </dependency>
     <dependency>
          <groupId>org.springframework.cloud</groupId>
          <artifactId>spring-cloud-starter-eureka</artifactId>
```

```
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
</dependencies>
```

引入这些依赖之后，在工程的配置文件 application.yml 做相关的配置，包括配置程序名为 eureka-feign-client，端口号为 8765，服务注册地址为 http://localhost:8761/eureka/，代码如下：

```
spring:
  application:
    name: eureka-feign-client
server:
  port: 8765
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

在程序的启动类 EurekaFeignClientApplication 加上注解@EnableEurekaClient 开启 Eureka Client 的功能，通过注解@EnableFeignClients 开启 Feign Client 的功能。代码如下：

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

通过以上 3 个步骤，该程序就具备了 Feign 的功能，现在来实现一个简单的 Feign Client。新建一个 EurekaClientFeign 的接口，在接口上加@FeignClient 注解来声明一个 Feign Client，其中 value 为远程调用其他服务的服务名，FeignConfig.class 为 Feign Client 的配置类。在 EurekaClientFeign 接口内部有一个 sayHiFromClientEureka()方法，该方法通过 Feign 来调用 eureka-client 服务的“/hi”的 API 接口，代码如下：

```
@FeignClient(value = "eureka-client",configuration = FeignConfig.class)
```

```
public interface EurekaClientFeign {
    @GetMapping(value = "/hi")
    String sayHiFromClientEureka(@RequestParam(value = "name") String name);
}
```

在FeignConfig类加上@Configuration注解，表明该类是一个配置类，并注入一个BeanName为 feignRetryer 的 Retryer 的 Bean。注入该 Bean 之后，Feign 在远程调用失败后会进行重试。代码如下：

```
@Configuration
public class FeignConfig {
   @Bean
   public Retryer feignRetryer() {
      return new Retryer.Default(100, SECONDS.toMillis(1), 5);
   }
}
```

在 Service 层的 HiService 类注入 EurekaClientFeign 的 Bean，通过 EurekaClientFeign 去调用 sayHiFromClientEureka()方法，其代码如下：

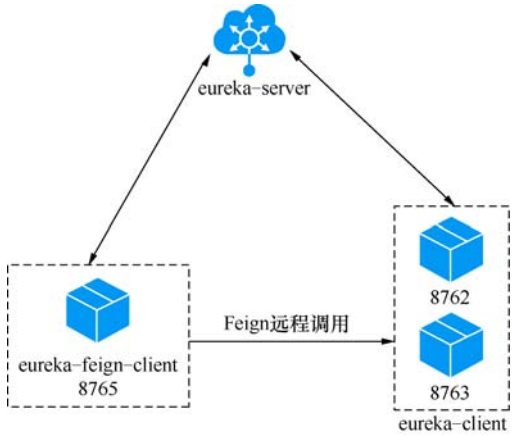
```
@Service
public class HiService {
   @Autowired
   EurekaClientFeign eurekaClientFeign;
   public String sayHi(String name){
      return  eurekaClientFeign.sayHiFromClientEureka(name);
   }
}
```

在 HiController 上加上@RestController 注解，开启 RestController 的功能，写一个 API 接口“/hi”，在该接口调用了 HiService 的 sayHi() 方法。HiService 通过 EurekaClientFeign 远程调用 eureka-client 服务的 API 接口“/hi”。代码如下：

```
@RestController
public class HiController {
   @Autowired
   HiService hiService;
   @GetMapping("/hi")
   public String sayHi(@RequestParam( defaultValue = "forezp",required = false)String
 name){
      return hiService.sayHi(name);
   }
}
```

启动 eureka-server 工程，端口号为 8761；启动两个 eureka-client 工程的实例，端口号分别为 8762 和 8763；启动 eureka-feign-client 工程，端口号为 8765，此时工程的架构如图 7-1

所示。



▲图 7-1　工程的架构图

在浏览器上多次访问 http://localhost:8765/hi，浏览器会轮流显示以下内容：

```
hi forezp,i am from port:8763
hi forezp,i am from port:8762
```

由此可见，Feign Client 远程调用了 eureka-client 服务（存在端口为 8762 和 8763 的两个实例）的“/hi”API 接口，Feign Client 有负载均衡的能力。

查看起步依赖 spring-cloud-starter-feign 的 pom 文件，可以看到该起步依赖默认引入了 Ribbon 和 Hystrix 的依赖，即负载均衡和熔断器的依赖。关于 Hystrix 将在下一章讲解。spring-cloud-starter-feign 的 pom 文件的代码如下：

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-starter</artifactId>
  </dependency>
  <dependency>
         <groupId>org.springframework.cloud</groupId>
           <artifactId>spring-cloud-netflix-core</artifactId>
       </dependency>
       <dependency>
           <groupId>org.springframework</groupId>
           <artifactId>spring-web</artifactId>
       </dependency>
       <dependency>
           <groupId>org.springframework.cloud</groupId>
           <artifactId>spring-cloud-commons</artifactId>
       </dependency>
       <dependency>
```

```
            <groupId>io.github.openfeign</groupId>
            <artifactId>feign-core</artifactId>
        </dependency>
        <dependency>
            <groupId>io.github.openfeign</groupId>
            <artifactId>feign-slf4j</artifactId>
        </dependency>
        <dependency>
            <groupId>io.github.openfeign</groupId>
            <artifactId>feign-hystrix</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-ribbon</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-archaius</artifactId>
        </dependency>
```

7.2 FeignClient 详解

为了深入理解 Feign，下面将从源码的角度来讲解 Feign。首先来查看 FeignClient 注解@FeignClient 的源码，其代码如下：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface FeignClient {
  @AliasFor("name")
  String value() default "";
  @AliasFor("value")
  String name() default "";
  @AliasFor("value")
  String name() default "";
  String url() default "";
  boolean decode404() default false;
  Class<?>[] configuration() default {};
  Class<?> fallback() default void.class;
  Class<?> fallbackFactory() default void.class;
  String path() default "";
  boolean primary() default true;
}
```

FeignClient 注解被@Target(ElementType.TYPE)修饰，表示 FeignClient 注解的作用目标在

接口上。@Retention(RetentionPolicy.RUNTIME)注解表明该注解会在 Class 字节码文件中存在，在运行时可以通过反射获取到。@Documented 表示该注解将被包含在 Javadoc 中。

@FeignClient 注解用于创建声明式 API 接口，该接口是 RESTful 风格的。Feign 被设计成插拔式的，可以注入其他组件和 Feign 一起使用。最典型的是如果 Ribbon 可用，Feign 会和 Ribbon 相结合进行负载均衡。

在代码中，value()和 name()一样，是被调用的服务的 ServiceId。url()直接填写硬编码的 Url 地址。decode404()即 404 是被解码，还是抛异常。configuration()指明 FeignClient 的配置类，默认的配置类为 FeignClientsConfiguration 类，在缺省的情况下，这个类注入了默认的 Decoder、Encoder 和 Contract 等配置的 Bean。fallback()为配置熔断器的处理类。

7.3 FeignClient 的配置

Feign Client 默认的配置类为 FeignClientsConfiguration，这个类在 spring-cloud-netflix-core 的 jar 包下。打开这个类，可以发现这个类注入了很多 Feign 相关的配置 Bean，包括 FeignRetryer、FeignLoggerFactory 和 FormattingConversionService 等。另外，Decoder、Encoder 和 Contract 这 3 个类在没有 Bean 被注入的情况下，会自动注入默认配置的 Bean，即 ResponseEntityDecoder、SpringEncoder 和 SpringMvcContract。默认注入的配置如下。

- Decoder feignDecoder：ResponseEntityDecoder。
- Encoder feignEncoder：SpringEncoder。
- Logger feignLogger：Slf4jLogger。
- Contract feignContract：SpringMvcContract。
- Feign.Builder feignBuilder：HystrixFeign.Builder。

FeignClientsConfiguration 的配置类部分代码如下，@ConditionalOnMissingBean 注解表示如果没有注入该类的 Bean 就会默认注入一个 Bean。

```
@Configuration
public class FeignClientsConfiguration {
...//省略代码
    @Bean
    @ConditionalOnMissingBean
    public Decoder feignDecoder() {
     return new ResponseEntityDecoder(new SpringDecoder(this.messageConverters));
    }

    @Bean
    @ConditionalOnMissingBean
    public Encoder feignEncoder() {
        return new SpringEncoder(this.messageConverters);
    }
```

```
    @Bean
    @ConditionalOnMissingBean
    public Contract feignContract(ConversionService feignConversionService) {
        return new SpringMvcContract(this.parameterProcessors, feignConversionServic
);
    }

...//省略代码
}
```

重写 FeignClientsConfiguration 类中的 Bean，覆盖掉默认的配置 Bean，从而达到自定义配置的目的。例如 Feign 默认的配置在请求失败后，重试次数为 0，即不重试（Retryer.NEVER_RETRY）。现在希望在请求失败后能够重试，这时需要写一个配置 FeignConfig 类，在该类中注入 Retryer 的 Bean，覆盖掉默认的 Retryer 的 Bean，并将 FeignConfig 指定为 FeignClient 的配置类。FeignConfig 类的代码如下：

```
@Configuration
public class FeignConfig {
   @Bean
   public Retryer feignRetryer() {
      return new Retryer.Default(100, SECONDS.toMillis(1), 5);
   }
}
```

在上面的代码中，通过覆盖了默认的 Retryer 的 Bean，更改了该 FeignClient 的请求失败重试的策略，重试间隔为 100 毫秒，最大重试时间为 1 秒，重试次数为 5 次。

7.4 从源码的角度讲解 Feign 的工作原理

Feign 是一个伪 Java Http 客户端，Feign 不做任何的请求处理。Feign 通过处理注解生成 Request 模板，从而简化了 Http API 的开发。开发人员可以使用注解的方式定制 Request API 模板。在发送 Http Request 请求之前，Feign 通过处理注解的方式替换掉 Request 模板中的参数，生成真正的 Request，并交给 Java Http 客户端去处理。利用这种方式，开发者只需要关注 Feign 注解模板的开发，而不用关注 Http 请求本身，简化了 Http 请求的过程，使得 Http 请求变得简单和容易理解。

Feign 通过包扫描注入 FeignClient 的 Bean，该源码在 FeignClientsRegistrar 类中。首先在程序启动时，会检查是否有@EnableFeignClients 注解，如果有该注解，则开启包扫描，扫描被@FeignClient 注解的接口。代码如下：

```
private void registerDefaultConfiguration(AnnotationMetadata metadata,
                BeanDefinitionRegistry registry) {
          Map<String, Object> defaultAttrs = metadata
```

```
                .getAnnotationAttributes(EnableFeignClients.class.getName(), true);

        if (defaultAttrs != null && defaultAttrs.containsKey("defaultConfiguration"))
{
            String name;
            if (metadata.hasEnclosingClass()) {
                name = "default." + metadata.getEnclosingClassName();
            }
            else {
                name = "default." + metadata.getClassName();
            }
            registerClientConfiguration(registry, name,
                    defaultAttrs.get("defaultConfiguration"));
        }
    }
```

当程序的启动类上有@EnableFeignClients 注解。在程序启动后，程序会通过包扫描将有@FeignClient 注解修饰的接口连同接口名和注解的信息一起取出，赋给 BeanDefinitionBuilder，然后根据 BeanDefinitionBuilder 得到 BeanDefinition，最后将 BeanDefinition 注入 IoC 容器中，源码如下：

```
public void registerFeignClients(AnnotationMetadata metadata,
            BeanDefinitionRegistry registry) {
        ...//省略代码
        scanner.setResourceLoader(this.resourceLoader);

                    Map<String, Object> attributes = annotationMetadata
                            .getAnnotationAttributes(
                                    FeignClient.class.getCanonicalName());

                    String name = getClientName(attributes);
                    registerClientConfiguration(registry, name,
                             attributes.get("configuration"));

                    registerFeignClient(registry, annotationMetadata, attributes);
                }
            }
        }
    }
private void registerFeignClient(BeanDefinitionRegistry registry,
            AnnotationMetadata annotationMetadata, Map<String, Object> attributes) {
        String className = annotationMetadata.getClassName();
        BeanDefinitionBuilder definition = BeanDefinitionBuilder
                .genericBeanDefinition(FeignClientFactoryBean.class);
            ...//省略代码
```

```
                    BeanDefinitionHolder holder = new BeanDefinitionHolder(beanDefinition,
className,
                    new String[] { alias });
          BeanDefinitionReaderUtils.registerBeanDefinition(holder, registry);
     }
```

注入 BeanDefinition 之后，通过 JDK 的代理，当调用 Feign Client 接口里面的方法时，该方法会被拦截，源码在 ReflectiveFeign 类，代码如下：

```
public <T> T newInstance(Target<T> target) {
     ...//省略代码
        for (Method method : target.type().getMethods()) {
      if (method.getDeclaringClass() == Object.class) {
        continue;
      } else if(Util.isDefault(method)) {
        DefaultMethodHandler handler = new DefaultMethodHandler(method);
        defaultMethodHandlers.add(handler);
        methodToHandler.put(method, handler);
      } else {
        methodToHandler.put(method, nameToHandler.get(Feign.configKey(target.type(),
method)));
      }
    }
    InvocationHandler handler = factory.create(target, methodToHandler);
    T proxy = (T) Proxy.newProxyInstance(target.type().getClassLoader(), new Class<?>
[]{target.type()}, handler);
    for(DefaultMethodHandler defaultMethodHandler : defaultMethodHandlers) {
      defaultMethodHandler.bindTo(proxy);
    }
    return proxy;
 }
```

在 SynchronousMethodHandler 类进行拦截处理，会根据参数生成 RequestTemplate 对象，该对象是 Http 请求的模板，代码如下：

```
@Override
  public Object invoke(Object[] argv) throws Throwable {
    RequestTemplate template = buildTemplateFromArgs.create(argv);
    Retryer retryer = this.retryer.clone();
    while (true) {
      try {
        return executeAndDecode(template);
      } catch (RetryableException e) {
        retryer.continueOrPropagate(e);
        if (logLevel != Logger.Level.NONE) {
          logger.logRetry(metadata.configKey(), logLevel);
        }
```

```
        continue;
      }
    }
  }
```

在上述代码中，有一个 executeAndDecode()方法，该方法通过 RequestTemplate 生成 Request 请求对象，然后用 Http Client 获取 Response，即通过 Http Client 进行 Http 请求来获取响应，代码如下：

```
Object executeAndDecode(RequestTemplate template) throws Throwable {
  Request request = targetRequest(template);
  ...//省略代码
  response = client.execute(request, options);
  ...//省略代码
}
```

7.5 在 Feign 中使用 HttpClient 和 OkHttp

在 Feign 中，Client 是一个非常重要的组件，Feign 最终发送 Request 请求以及接收 Response 响应都是由 Client 组件完成的。Client 在 Feign 源码中是一个接口，在默认的情况下，Client 的实现类是 Client.Default，Client.Default 是由 HttpURLConnnection 来实现网络请求的。另外，Client 还支持 HttpClient 和 OkhHttp 来进行网络请求。

首先查看 FeignRibbonClient 的自动配置类 FeignRibbonClientAutoConfiguration ，该类在工程启动时注入一些 Bean，其中注入了一个 BeanName 为 feignClient 的 Client 类型的 Bean，代码如下：

```
@ConditionalOnClass({ ILoadBalancer.class, Feign.class })
@Configuration
@AutoConfigureBefore(FeignAutoConfiguration.class)
public class FeignRibbonClientAutoConfiguration {
//代码省略
     @Bean
     @ConditionalOnMissingBean
     public Client feignClient(CachingSpringLoadBalancerFactory cachingFactory,
               SpringClientFactory clientFactory) {
          return new LoadBalancerFeignClient(new Client.Default(null, null),
                    cachingFactory, clientFactory);
     }
//代码省略
}
```

在缺省配置 BeanName 为 FeignClient 的 Bean 的情况下，会自动注入 Client.Default 这个对象，跟踪 Client.Default 源码，Client.Default 使用的网络请求框架为 HttpURLConnection，代码

如下：

```
@Override
   public Response execute(Request request, Options options) throws IOException {
     HttpURLConnection connection = convertAndSend(request, options);
     return convertResponse(connection).toBuilder().request(request).build();
   }
```

那么，如何在 Feign 中使用 HttpClient 的网络请求框架呢？下面继续查看 FeignRibbonClientAutoConfiguration 的源码：

```
@ConditionalOnClass({ ILoadBalancer.class, Feign.class })
@Configuration
@AutoConfigureBefore(FeignAutoConfiguration.class)
public class FeignRibbonClientAutoConfiguration {
...//省略代码

    @Configuration
    @ConditionalOnClass(ApacheHttpClient.class)
    @ConditionalOnProperty(value = "feign.httpclient.enabled", matchIfMissing = true)
    protected static class HttpClientFeignLoadBalancedConfiguration {
         @Autowired(required = false)
         private HttpClient httpClient;
         @Bean
         @ConditionalOnMissingBean(Client.class)
         public Client feignClient(CachingSpringLoadBalancerFactory cachingFactory,
                   SpringClientFactory clientFactory) {
              ApacheHttpClient delegate;
              if (this.httpClient != null) {
                   delegate = new ApacheHttpClient(this.httpClient);
              }
              else {
                   delegate = new ApacheHttpClient();
              }
              return new LoadBalancerFeignClient(delegate, cachingFactory, clientFactory);
         }
    }
...//省略代码
}
```

从代码@ConditionalOnClass(ApacheHttpClient.class)注解可知道，只需要在 pom 文件加上 HttpClient 的 Classpath 即可。另外需要在配置文件 application.yml 中配置 feign.httpclient.enabled 为 true，从@ConditionalOnProperty 注解可知，这个配置可以不写，因为在默认的情况下就为 true。

在工程的 pom 文件加上 feign-httpclient 的依赖，Feign 就会采用 HttpClient 作为网络请求

框架，而不是默认的 HttpURLConnection。代码如下：

```
<dependency>
    <groupId>com.netflix.feign</groupId>
    <artifactId>feign-httpclient</artifactId>
    <version>RELEASE</version>
</dependency>
```

同理，如果想要 Feign 中使用 Okhttp 作为网络请求框架，则只需要在 pom 文件上加上 feign-okhttp 的依赖，代码如下：

```
<dependency>
    <groupId>com.netflix.feign</groupId>
    <artifactId>feign-okhttp</artifactId>
    <version>RELEASE</version>
</dependency>
```

7.6 Feign 是如何实现负载均衡的

FeignRibbonClientAutoConfiguration 类配置了 Client 的类型（包括 HttpURLConnection、OkHttp 和 HttpClient），最终向容器注入的是 Client 的实现类 LoadBalancerFeignClient，即负载均衡客户端。查看 LoadBalancerFeignClient 类中的 execute 方法，即执行请求的方法，代码如下：

```
@Override
   public Response execute(Request request, Request.Options options) throws IOException {
       try {
            URI asUri = URI.create(request.url());
            String clientName = asUri.getHost();
            URI uriWithoutHost = cleanUrl(request.url(), clientName);
            FeignLoadBalancer.RibbonRequestribbonRequest = new FeignLoadBalancer.
RibbonRequest(
                    this.delegate, request, uriWithoutHost);

               IClientConfig requestConfig = getClientConfig(options, clientName);
               return lbClient(clientName).executeWithLoadBalancer(ribbonRequest,
                        requestConfig).toResponse();
          }
          catch (ClientException e) {
               IOException io = findIOException(e);
               if (io != null) {
                     throw io;
               }
```

```
                throw new RuntimeException(e);
            }
        }
```

其中有一个 executeWithLoadBalancer()方法，即通过负载均衡的方式来执行网络请求，代码如下：

```
public T executeWithLoadBalancer(final S request, final IClientConfig requestConfig)
throws ClientException {
       ...//代码省略
        try {
             return command.submit(
                 new ServerOperation<T>() {
                     ...//代码省略
                 })
                .toBlocking()
                 .single();
         } catch (Exception e) {
              ...//代码省略
         }
    }
```

在上述代码中，有一个 submit()方法，进入 submit() 方法的内部可以看出它是 LoadBalancerCommand 类的方法，代码如下：

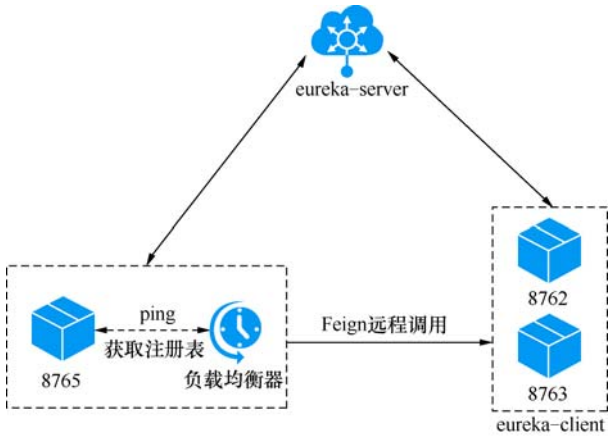
```
   Observable<T> o =
               (server == null ? selectServer() : Observable.just(server))
               .concatMap(new Func1<Server, Observable<T>>() {
                    @Override
                    // Called for each server being selected
                    public Observable<T> call(Server server) {
                        context.setServer(server);
   }}
```

在上述代码中，有一个 selectServe()方法，该方法是选择服务进行负载均衡的方法，代码如下：

```
   private Observable<Server> selectServer() {
          return Observable.create(new OnSubscribe<Server>() {
               @Override
               public void call(Subscriber<? super Server> next) {
                   try {
                        Server server = loadBalancerContext.getServerFromLoadBalancer(
loadBalancerURI, loadBalancerKey);
                        next.onNext(server);
                        next.onCompleted();
                   } catch (Exception e) {
```

```
                    next.onError(e);
                }
            }
        });
    }
```

由上述代码可知，最终负载均衡交给 loadBalancerContext 来处理，即上一章讲述的 Ribbon，这里不再重复。此时案例的架构图如图 7-2 所示。



▲图 7-2　工程的架构图

7.7 总结

总的来说，Feign 的源码实现过程如下。

（1）首先通过@EnableFeignClients 注解开启 FeignClient 的功能。只有这个注解存在，才会在程序启动时开启对@FeignClient 注解的包扫描。

（2）根据 Feign 的规则实现接口，并在接口上面加上@FeignClient 注解。

（3）程序启动后，会进行包扫描，扫描所有的@ FeignClient 的注解的类，并将这些信息注入 IoC 容器中。

（4）当接口的方法被调用时，通过 JDK 的代理来生成具体的 RequestTemplate 模板对象。

（5）根据 RequestTemplate 再生成 Http 请求的 Request 对象。

（6）Request 对象交给 Client 去处理，其中 Client 的网络请求框架可以是 HttpURLConnection、HttpClient 和 OkHttp。

（7）最后 Client 被封装到 LoadBalanceClient 类，这个类结合类 Ribbon 做到了负载均衡。

第 8 章　熔断器 Hystrix

前两章讲述了如何使用 RestTemplate 和 Feign 去消费服务，并详细地讲述了 Ribbon 做负载均衡的原理和 Feign 的工作原理。本章将讲述如何在用 RestTemplate 和 Feign 消费服务时使用熔断器 Hystrix，将从以下 7 个方面进行讲解。

- 什么是 Hystrix。
- Hystrix 解决了什么问题。
- Hystrix 的工作原理。
- 如何在 RestTemplate 和 Ribbon 作为服务消费者时使用 Hystrix。
- 如何在 Feign 作为服务消费者时使用 Hystrix。
- 如何使用 Hystrix Dashboard 监控熔断器的状况。
- 如何使用 Turbine 聚合多个 Hystrix Dashboard。

8.1 什么是 Hystrix

在分布式系统中，服务与服务之间的依赖错综复杂，一种不可避免的情况就是某些服务会出现故障，导致依赖于它们的其他服务出现远程调度的线程阻塞。Hystrix 是 Netflix 公司开源的一个项目，它提供了熔断器功能，能够阻止分布式系统中出现联动故障。Hystrix 是通过隔离服务的访问点阻止联动故障的，并提供了故障的解决方案，从而提高了整个分布式系统的弹性。

8.2 Hystrix 解决了什么问题

在复杂的分布式系统中，可能有几十个服务相互依赖，这些服务由于某些原因，例如机房的不可靠性、网络服务商的不可靠性等，导致某个服务不可用。如果系统不隔离该不可用的服务，可能会导致整个系统不可用。

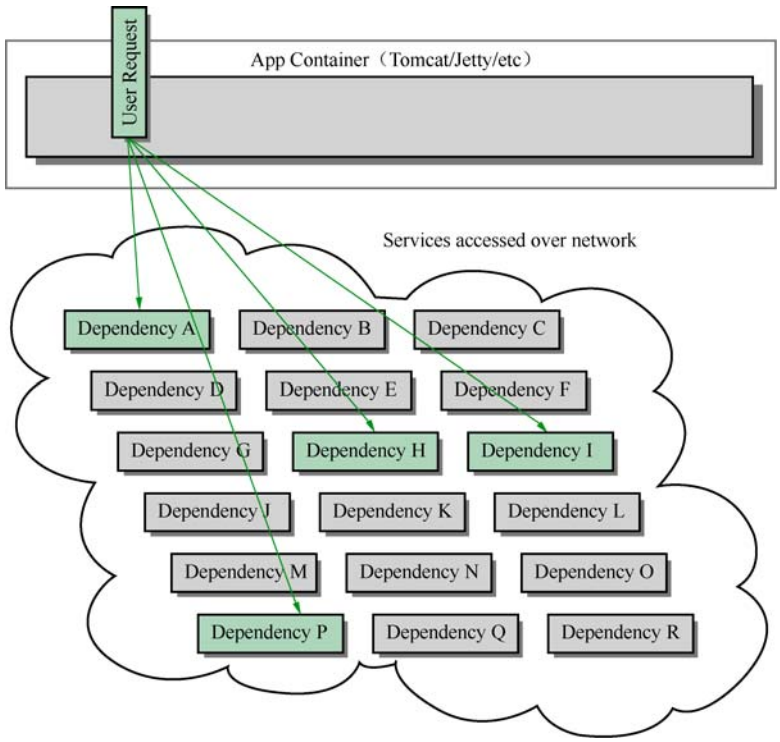
例如，对于依赖 30 个服务的应用程序，每个服务的正常运行时间为 99.99%，对于单个服务来说，99.99%的可用是非常完美的。

有 $99.99^{30} = 99.7\%$的可正常运行时间和 0.3%的不可用时间，那么 10 亿次请求中有 3000000

次失败，实际的情况可能比这更糟糕。

如果不设计整个系统的韧性，即使所有依赖关系表现良好，单个服务只有 0.01%的不可用，由于整个系统的服务相互依赖，最终对整个系统的影响是非常大的。

在微服务系统中，一个用户请求可能需要调用几个服务才能完成。如图 8-1 所示，在所有的服务都处于可用状态时，一个用户请求需要调用 A、H、I 和 P 服务。



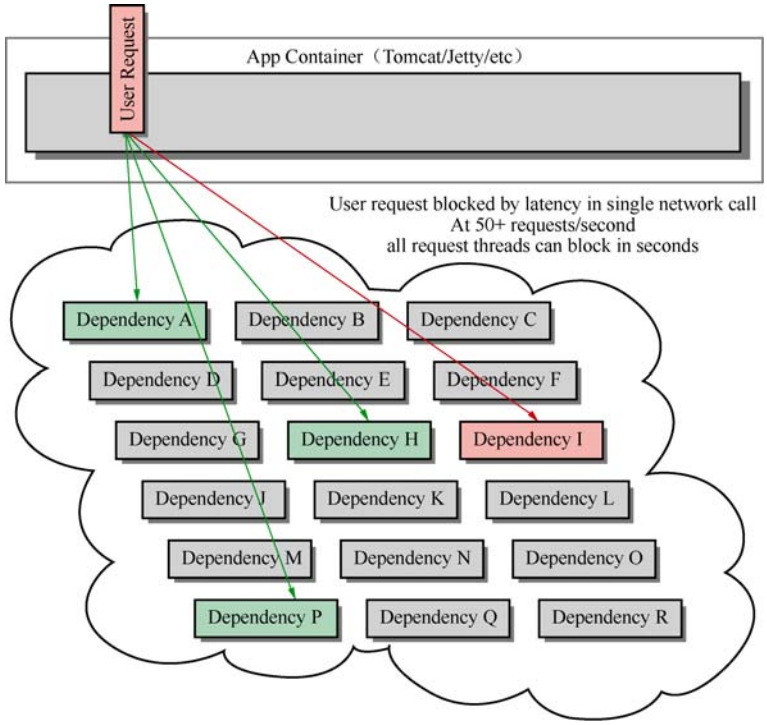
▲图 8-1　正常情况下一个请求的状态（图片来源于网络）

当某一个服务，例如服务 I，出现网络延迟或者故障时，即使服务 A、H 和 P 可用，由于服务 I 的不可用，整个用户请求会处于阻塞状态，并等待服务 I 的响应，如图 8-2 所示。

在高并发的情况下，单个服务的延迟会导致整个请求都处于延迟状态，可能在几秒钟就使整个服务处于线程负载饱和的状态。

某个服务的单个点的请求故障会导致用户的请求处于阻塞状态，最终的结果就是整个服务的线程资源消耗殆尽。由于服务的依赖性，会导致依赖于该故障服务的其他服务也处于线程阻塞状态，最终导致这些服务的线程资源消耗殆尽，直到不可用，从而导致整个问服务系统都不可用，即雪崩效应。

为了防止雪崩效应，因而产生了熔断器模型。Hystrix 是在业界表现非常好的一个熔断器模型实现的开源组件，它是 Spring Cloud 组件不可缺少的一部分。



▲图 8-2　当某个服务出现故障（图片来源于网络）

8.3 Hystrix 的设计原则

总的来说，Hystrix 的设计原则如下。

- 防止单个服务的故障耗尽整个服务的 Servlet 容器（例如 Tomcat）的线程资源。
- 快速失败机制，如果某个服务出现了故障，则调用该服务的请求快速失败，而不是线程等待。
- 提供回退（fallback）方案，在请求发生故障时，提供设定好的回退方案。
- 使用熔断机制，防止故障扩散到其他服务。
- 提供熔断器的监控组件 Hystrix Dashboard，可以实时监控熔断器的状态。

8.4 Hystrix 的工作机制

第 2 章的图 2-5 展示了 Hystrix 的工作机制。首先，当服务的某个 API 接口的失败次数在一定时间内小于设定的阈值时，熔断器处于关闭状态，该 API 接口正常提供服务。当该 API 接口处理请求的失败次数大于设定的阈值时，Hystrix 判定该 API 接口出现了故障，打开熔断器，这时请求该 API 接口会执行快速失败的逻辑（即 fallback 回退的逻辑），不执行业务逻辑，请求的线程不会处于阻塞状态。处于打开状态的熔断器，一段时间后会处于半打开

状态，并将一定数量的请求执行正常逻辑。剩余的请求会执行快速失败，若执行正常逻辑的请求失败了，则熔断器继续打开；若成功了，则将熔断器关闭。这样熔断器就具有了自我修复的能力。

8.5 在 RestTemplate 和 Ribbon 上使用熔断器

本节以案例的形式讲解如何在 RestTemplate 和 Ribbon 作为服务消费者时使用 Hystrix 熔断器。本节的案例在上一章的案例基础之上进行改造。在上一章的 eureka-ribbon-client 工程中，我们使用 RestTempalte 调用了 eureka-client 的“/hi”API 接口，并用 Ribbon 做了负载均衡，本节在此基础上加 Hystrix 熔断器的功能。

首先在工程的 pom 文件中引用 Hystrix 的起步依赖 spring-cloud-starter-hystrix，代码如下：

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

然后在 Spring Boot 的启动类 EurekaRibbonClientApplication 加上@EnableHystrix 注解开启 Hystrix 的熔断器功能，代码如下：

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

修改 RibbonService 的代码，在 hi()方法上加@HystrixCommand 注解。有了@HystrixCommand 注解，hi()方法就启用 Hystrix 熔断器的功能，其中，fallbackMethod 为处理回退（fallback）逻辑的方法。在本例中，直接返回了一个字符串。在熔断器打开的状态下，会执行 fallback 逻辑。fallback 的逻辑最好是返回一些静态的字符串，不需要处理复杂的逻辑，也不需要远程调度其他服务，这样方便执行快速失败，释放线程资源。如果一定要在 fallback 逻辑中远程调度其他服务，最好在远程调度其他服务时，也加上熔断器。案例代码如下：

```
@Service
public class RibbonService {
      @Autowired
      RestTemplate restTemplate;
      @HystrixCommand(fallbackMethod = "hiError")
```

```
        public String hi(String name) {
            return restTemplate.getForObject("http://eureka-client/hi?name="+name,String.class);
        }
        public String hiError(String name) {
            return "hi,"+name+",sorry,error!";
        }
}
```

依次启动工程 eureka-server、eureka-client 和 eureka-ribbon-client。等所有的工程都启动完毕，在浏览器上访问 http://localhost:8765/hi，浏览器会显示：

```
hi forezp,i am from port:8762
```

关闭 eureka-client，即它处于不可用的状态，此时 eureka-ribbon-client 无法调用 eureka-client 的“/hi”接口，访问 http://localhost:8765/hi，浏览器会显示：

```
hi,forezp,sorry,error!
```

由此可见，当 eureka-client 不可用时，调用 eureka-ribbon-client 的“/hi”接口会进入 RibbonService 类的“/hi”方法中。由于 eureka-client 没有响应，判定 eureka-client 不可用，开启了熔断器，最后进入了 fallbackMethod 的逻辑。当熔断器打开了，之后的请求会直接执行 fallbackMethod 的逻辑。这样做的好处就是通过快速失败，请求能够得到及时处理，线程不再阻塞。

8.6 在 Feign 上使用熔断器

由于 Feign 的起步依赖中已经引入了 Hystrix 的依赖，所以在 Feign 中使用 Hystrix 不需要引入任何的依赖。只需要在 eureka-feign-client 工程的配置文件 application.yml 中配置开启 Hystrix 的功能，配置文件 application.yml 中加以下配置：

```
feign:
   hystrix:
     enabled: true
```

然后修改 eureka-feign-client 工程中的 EurekaClientFeign 代码，在@FeignClient 注解的 fallback 配置加上快速失败的处理类。该处理类是作为 Feign 熔断器的逻辑处理类，必须实现被@FeignClient 修饰的接口。例如案例中的 HiHystrix 类实现了接口 EurekaClientFeign，最后需要以 Spring Bean 的形式注入 IoC 容器中。代码如下：

```
@FeignClient(value = "eureka-client",
             configuration = FeignConfig.class,fallback = HiHystrix.class)
public interface EurekaClientFeign {
```

```
    @GetMapping(value = "/hi")
    String sayHiFromClientEureka(@RequestParam(value = "name") String name);
}
```

HiHystrix 作为熔断器的逻辑处理类，需要实现 EurekaClientFeign 接口，并需要在接口方法 sayHiFromClientEureka()里写处理熔断的具体逻辑，同时还需要在 HiHystrix 类上加@Component 注解，注入 IoC 容器中。代码如下：

```
@Component
public class HiHystrix implements EurekaClientFeign {
    @Override
    public String sayHiFromClientEureka(String name) {
        return "hi,"+name+",sorry,error!";
    }
}
```

依次启动工程 eureka-server、eureka-client 和 eureka-feign-client。在浏览器上访问 http://localhost:8765/hi，浏览器会显示：

```
hi forezp,i am from port:8762
```

关闭 eureka-client，即它处于不可用的状态，此时 eureka-feign-client 无法调用 eureka-client 的“/hi”接口，在浏览器上访问 http://localhost:8765/hi，浏览器会显示：

```
hi,forezp,sorry,error!
```

由此可见，当 eureka-client 不可用时，eureka-feign-clien 进入了 fallback 的逻辑处理类（即 HiHystrix），由这个类来执行熔断器打开时的处理逻辑。

8.7 使用 Hystrix Dashboard 监控熔断器的状态

在微服务架构中，为了保证服务实例的可用性，防止服务实例出现故障导致线程阻塞，而出现了熔断器模型。熔断器的状况反映了一个程序的可用性和健壮性，它是一个重要指标。Hystrix Dashboard 是监控 Hystrix 的熔断器状况的一个组件，提供了数据监控和友好的图形化展示界面。本节在上一节的基础上，以案例的形式讲述如何使用 Hystrix Dashboard 监控熔断器的状态。

8.7.1 在 RestTemplate 中使用 Hystrix Dashboard

改造上一节的工程，首先在 eureka-ribbon-client 工程的 pom 文件上加上 Actuator 的起步依赖、Hystrix Dashboard 的起步依赖和 Hystrix 的起步依赖，这 3 个依赖是必需的。代码如下：

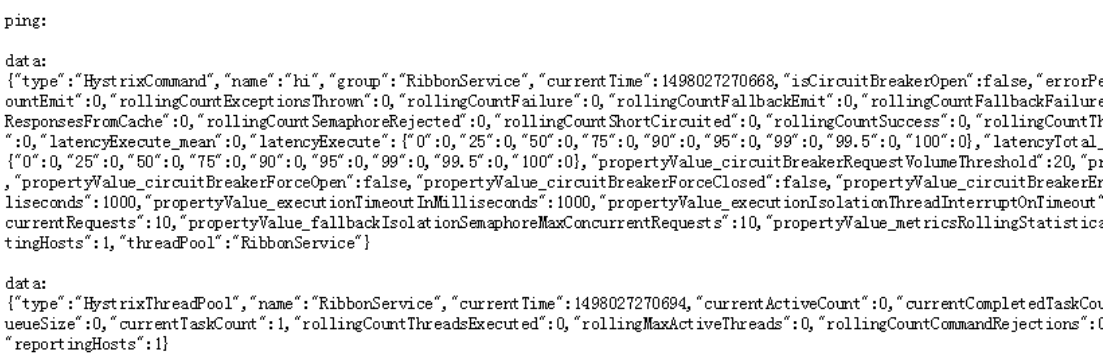
```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
 <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud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

在程序的启动类 EurekaRibbonClientApplication 加上@EnableHystrixDashboard 开启 Hystrix Dashboard 的功能，完整的代码如下：

```
@SpringBootApplication
@EnableEurekaClient
@EnableHystrix
@EnableHystrixDashboard
public class EurekaRibbonClientApplication {
     public static void main(String[] args) {
          SpringApplication.run(EurekaRibbonClientApplication.class, args);
     }
}
```

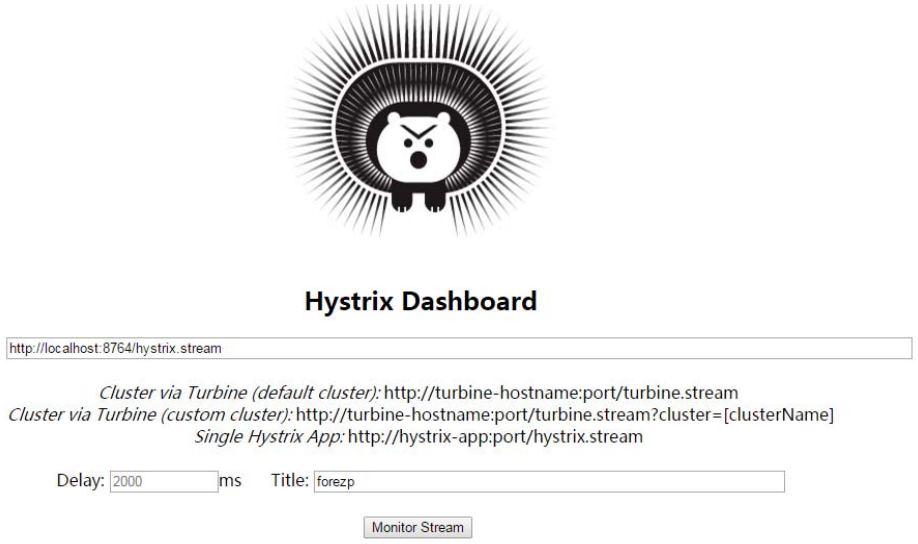
依次启动工程 eureka-server、eureka-client 和 eureka-ribbon-client。在浏览器上访问 http://localhost: 8765/hi。

然后在浏览器上访问 http://localhost:8764/hystrix.stream，浏览器上会显示熔断器的数据指标，如图 8-3 所示。



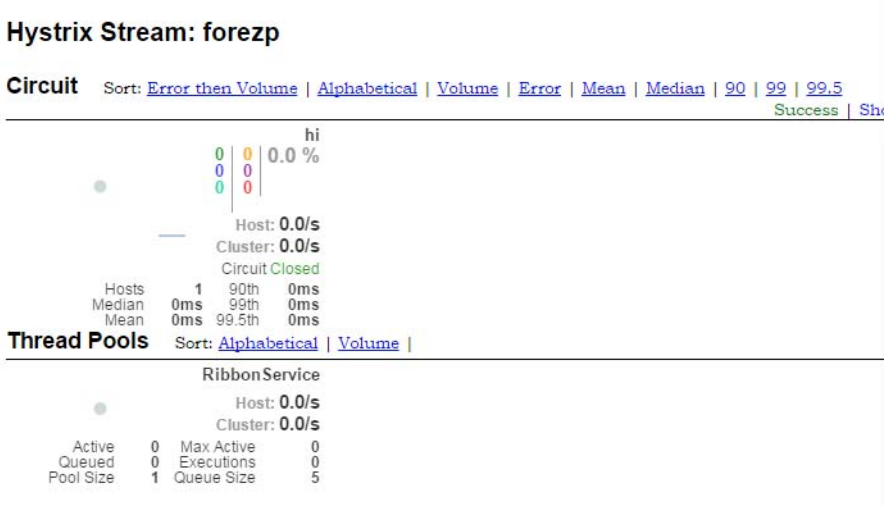
▲图 8-3 Hystrix 的数据指标图

在浏览器上访问 http://localhost:8764/hystrix，浏览器显示的界面如图 8-4 所示。



▲图 8-4　Hystrix Dashboard 的主页图

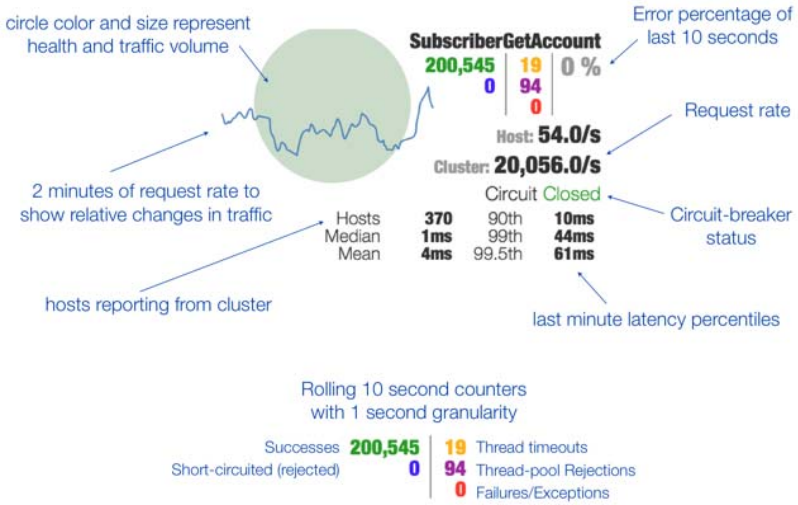
在界面上依次填写 http://localhost:8764/hystrix.stream、2000、forezp（这个可以随意填写），单击“monitor”，进入页面，如图 8-5 所示。



▲图 8-5　eureka-ribbon-client 的 Hystrix Dashboard 展示图

在该页面显示了熔断器的各种数据指标，这些数据指标所表示的含义如图 8-6 所示，该图来自于 Hystrix 的官方文档，更多信息可以查阅官方文档，文档地址：https://github.com/Netflix/

Hystrix/wiki/Dashboard。



▲图 8-6 Hystrix Dashboard 的各种数据指标的含义

8.7.2 在 Feign 中使用 Hystrix Dashboard

同 eureka-ribbon-client 类似，eureka-feign-client 工程的 pom 文件需要加上 Actuator、Hystrix 和 Hystrix Dashboard 的起步依赖。可能有读者会疑惑：Feign 不是自带 Hystrix 吗？为什么还需要加入 spring-cloud-starter-hystrix？这是因为 Feign 自带的 Hystrix 的依赖不是起步依赖。Feign 的起步依赖包含的依赖如下：

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

需要在程序的启动类 EurekaFeignClientApplication 加上注解@ EnableHystrixDashboard 开启 HystrixDashboard 的功能。完整的代码如下：

```
@SpringBootApplication
@EnableEurekaClient
```

```
@EnableFeignClients
@EnableHystrixDashboard
@EnableHystrix
public class EurekaFeignClientApplication {
     public static void main(String[] args) {
          SpringApplication.run(EurekaFeignClientApplication.class, args);
     }
}
```

只需要上述两步就可以在 Feign 中开启 Hystrix Dashboard 的功能。在浏览器上展示 Hystrix Dashboard 的操作步骤同上一节，本节不再演示。

8.8 使用 Turbine 聚合监控

在使用 Hystrix Dashboard 组件监控服务的熔断器状况时，每个服务都有一个 Hystrix Dashboard 主页，当服务数量很多时，监控非常不方便。为了同时监控多个服务的熔断器的状况，Netflix 开源了 Hystrix 的另一个组件 Turbine。Turbine 用于聚合多个 Hystrix Dashboard，将多个 Hystrix Dashboard 组件的数据放在一个页面上展示，进行集中监控。

在上一节的例子上继续进行改造，在主 Maven 工程下新建一个 Module 工程，做为 Turbine 聚合监控的工程，取名为 eureka-monitor-client。首先，在工程的 pom 文件引入工程所需的依赖，包括 turbine、actuator 和 test 的起步依赖，完整的代码如下：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-turbine</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

然后在工程的配置文件 application 加上相关的配置，具体配置代码如下：

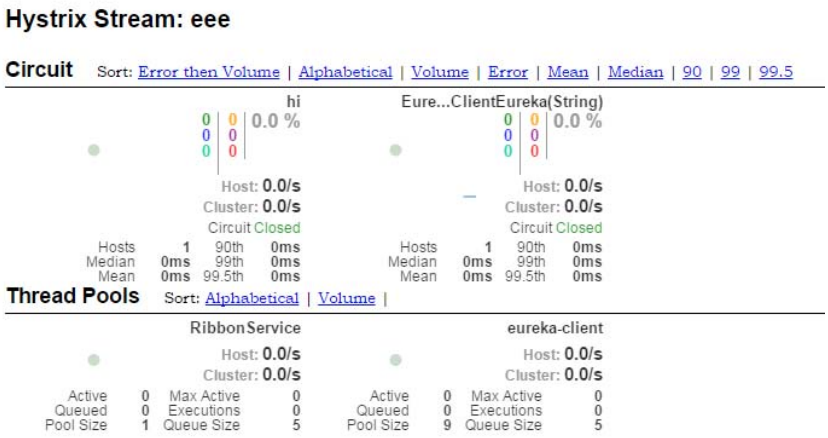
```
spring:
  application.name: service-turbine
server:
  port: 8769
```

```
turbine:
  aggregator:
    clusterConfig: default
  appConfig: eureka-ribbon-client,eureka-feign-client
  clusterNameExpression: new String("default")
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

上述配置代码指定了工程的端口号为 8769，服务名为 service-turbine。turbine.aggregator.clusterConfig 配置了需要监控的服务名，如本例中的 eureka-ribbon-client 和 eureka-feign-client。clusterNameExpression 默认为服务名的集群，此时用默认的即可。turbine.aggregator.clusterConfig 可以不写，因为默认就是 default。最后指定了服务注册中心的地址为 http://localhost:8761/eureka/。

启动工程 eureka-server、eureka-client、eureka-ribbon-client 和 eureka-monitor-client。在浏览器上访问 http://localhost:8764/hi?name=forezp 和 http://localhost:8765/hi?name=forezp。

在浏览器上打开网址 http://localhost:8765/hystrix，这个界面为 Hystrix Dashboard 界面。在界面上依次输入监控流的 Url 地址 http://localhost:8769/turbine.stream、监控时间间隔 2000 毫秒和 title，单击“monitor”，可以看到如图 8-7 所示的界面。



▲图 8-7　Hystrix 的 Turbine 聚合监控

从图 8-7 中可以看到，这个页面聚合了 eureka-ribbon-client 和 eureka-feign-client 的 Hystrix Dashboard 数据。

第 9 章　路由网关 Spring Cloud Zuul

前文已经讲解了 Netflix 的一系列组件，包括服务发现和注册组件 Eureka、负载均衡组件 Ribbon、声明式调用组件 Feign 和熔断器组件 Hystrix。本章讲解 Netflix 构建微服务的另一个组件——智能路由网关组件 Zuul。Zuul 作为微服务系统的网关组件，用于构建边界服务（Edge Service），致力于动态路由、过滤、监控、弹性伸缩和安全。

本章将从以下 3 个方面来讲述 Zuul。

- 为什么需要 Zuul。
- Zuul 的工作原理。
- Zuul 的案例实战。

9.1 为什么需要 Zuul

Zuul 作为路由网关组件，在微服务架构中有着非常重要的作用，主要体现在以下 6 个方面。

- Zuul、Ribbon 以及 Eureka 相结合，可以实现智能路由和负载均衡的功能，Zuul 能够将请求流量按某种策略分发到集群状态的多个服务实例。
- 网关将所有服务的 API 接口统一聚合，并统一对外暴露。外界系统调用 API 接口时，都是由网关对外暴露的 API 接口，外界系统不需要知道微服务系统中各服务相互调用的复杂性。微服务系统也保护了其内部微服务单元的 API 接口，防止其被外界直接调用，导致服务的敏感信息对外暴露。
- 网关服务可以做用户身份认证和权限认证，防止非法请求操作 API 接口，对服务器起到保护作用。
- 网关可以实现监控功能，实时日志输出，对请求进行记录。
- 网关可以用来实现流量监控，在高流量的情况下，对服务进行降级。
- API 接口从内部服务分离出来，方便做测试。

9.2 Zuul 的工作原理

Zuul 是通过 Servlet 来实现的，Zuul 通过自定义的 ZuulServlet（类似于 Spring MVC 的

DispatcServlet）来对请求进行控制。Zuul 的核心是一系列过滤器，可以在 Http 请求的发起和响应返回期间执行一系列的过滤器。Zuul 包括以下 4 种过滤器。

- PRE 过滤器：它是在请求路由到具体的服务之前执行的，这种类型的过滤器可以做安全验证，例如身份验证、参数验证等。
- ROUTING 过滤器：它用于将请求路由到具体的微服务实例。在默认情况下，它使用 Http Client 进行网络请求。
- POST 过滤器：它是在请求已被路由到微服务后执行的。一般情况下，用作收集统计信息、指标，以及将响应传输到客户端。
- ERROR 过滤器：它是在其他过滤器发生错误时执行的。

Zuul 采取了动态读取、编译和运行这些过滤器。过滤器之间不能直接相互通信，而是通过 RequestContext 对象来共享数据，每个请求都会创建一个 RequestContext 对象。Zuul 过滤器具有以下关键特性。

- Type（类型）：Zuul 过滤器的类型，这个类型决定了过滤器在请求的哪个阶段起作用，例如 Pre、Post 阶段等。
- Execution Order（执行顺序）：规定了过滤器的执行顺序，Order 的值越小，越先执行。
- Criteria（标准）：Filter 执行所需的条件。
- Action（行动）：如果符合执行条件，则执行 Action（即逻辑代码）。

Zuul 请求的生命周期如图 9-1 所示，该图来自 Zuul 的官方文档。

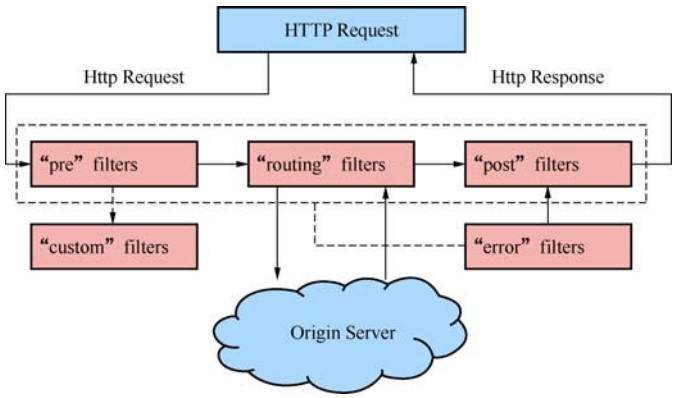
当一个客户端 Request 请求进入 Zuul 网关服务时，网关先进入“pre filter”，进行一系列的验证、操作或者判断。然后交给“routing filter”进行路由转发，转发到具体的服务实例进行逻辑处理、返回数据。当具体的服务处理完后，最后由“post filter”进行处理，该类型的处理器处理完之后，将 Response 信息返回给客户端。

ZuulServlet 是 Zuul 的核心 Servlet。ZuulServlet 的作用是初始化 ZuulFilter，并编排这些 ZuulFilter 的执行顺序。该类中有一个 service()方法，执行了过滤器执行的逻辑。

```
@Override
  public void service（） throws ServletException, IOException {
       try {
           try {
                preRoute();
           } catch (ZuulException e) {
               error(e);
               postRoute();
               return;
           }
           try {
               route();
           } catch (ZuulException e) {
               error(e);
```

```
                postRoute();
                return;
            }
            try {
                postRoute();
            } catch (ZuulException e) {
                error(e);
                return;
            }
        } catch (Throwable e) {
            error(new ZuulException(e, 500, "UNHANDLED_EXCEPTION_" + e.getClass().getName()));
        } finally {
            RequestContext.getCurrentContext().unset();
        }
    }
```

从上面的代码可知，首先执行 preRoute()方法，这个方法执行的是 PRE 类型的过滤器的逻辑。如果执行这个方法时出错了，那么会执行 error(e)和 postRoute()。然后执行 route()方法，该方法是执行 ROUTING 类型过滤器的逻辑。最后执行 postRoute()，该方法执行了 POST 类型过滤器的逻辑。



▲图 9-1　Zuul 请求的生命周期

9.3 案例实战

9.3.1 搭建 Zuul 服务

本章的案例是在上一章案例的基础上进行讲解的。新建一个 Spring Boot 工程，取名为 eureka-zuul-client，在 pom 文件中引入相关依赖，包括继承了主 Maven 工程的 pom 文件，引入 Eureka Client 的起步依赖 spring-cloud-starter-eureka 、Zuul 的起步依赖 spring-cloud-starter-zuul、Web 功能的起步依赖 spring-boot-starter-web，以及 Spring Boot 测试的起步依赖 spring-boot-starter-

test。代码如下：

```
    <parent>
        <groupId>com.forezp</groupId>
        <artifactId>chapter5-2</artifactId>
        <version>1.0-SNAPSHOT</version>
     </parent>
     <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka</artifactId>
        </dependency>
        <dependency>
             <groupId>org.springframework.cloud</groupId>
             <artifactId>spring-cloud-starter-zuul</artifactId>
        </dependency>
        <dependency>
             <groupId>org.springframework.boot</groupId>
             <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
             <groupId>org.springframework.boot</groupId>
             <artifactId>spring-boot-starter-test</artifactId>
             <scope>test</scope>
        </dependency>
     </dependencies>
```

在程序的启动类 EurekaZuulClientApplication 加上@EnableEurekaClient 注解，开启 EurekaClient 的功能；加上@SpringBootApplication 注解，表明自己是一个 Spring Boot 工程；加上@EnableZuulProxy 注解，开启 Zuul 的功能。代码如下：

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

在工程的配置文件 application.yml 中做相关的配置，包括配置服务注册中心的地址为 http://localhost:8761/eureka，程序的端口号为 5000，程序名为 service-zuul。

最后来重点讲解一下 Zuul 路由的配置写法，在本案例中，zuul.routes.hiapi.path 为“/hiapi/**”，zuul.routes.hiapi.serviceId 为“eureka-client”，这两个配置就可以将以“/hiapi”开

头的 Url 路由到 eureka-client 服务。其中，zuul.routes.hiapi 中的“hiapi”是自己定义的，需要指定它的 path 和 serviceId，两者配合使用，就可以将指定类型的请求 Url 路由到指定的 ServiceId。同理，满足以“/ribbonapi”开头的请求 Url 都会被分发到 eureka-ribbon- client，满足以“/feignapi/”开头的请求 Url 都会被分发到 eureka-feign-client 服务。如果某服务存在多个实例，Zuul 结合 Ribbon 会做负载均衡，将请求均分的部分路由到不同的服务实例。

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
server:
  port: 5000
spring:
  application:
    name: service-zuul
zuul:
  routes:
    hiapi:
      path: /hiapi/**
      serviceId: eureka-client
    ribbonapi:
      path: /ribbonapi/**
      serviceId: eureka-ribbon-client
    feignapi:
      path: /feignapi/**
      serviceId: eureka-feign-client
```

依次启动工程 eureka-server、eureka-client、eureka-ribbon-client、eureka-feign-client 和 eureka-zuul-client，其中 eureka-client 启动两个实例，端口为 8762 和 8763。在浏览器上多次访问 http://localhost:5000/hiapi/hi?name=forezp，浏览器会交替显示以下内容：

```
hi forezp,i am from port:8762
hi forezp,i am from port:8763
```

可见 Zuul 在路由转发做了负载均衡。同理，多次访问 http://localhost:5000/feignapi/ hi?name=forezp 和 http://localhost:5000/ribbonapi/hi? name=forezp，也可以看到相似的内容。

如果不需要用 Ribbon 做负载均衡，可以指定服务实例的 Url，用 zuul.routes.hiapi.url 配置指定，这时就不需要配置 zuul.routes.hiapi.serviceId 了。一旦指定了 Url，Zuul 就不能做负载均衡了，而是直接访问指定的 Url，在实际的开发中这种做法是不可取的。修改配置的代码如下：

```
zuul:
  routes:
    hiapi:
```

```
      path: /hiapi/**
      url: http://localhost:8762
```

重新启动 eureka-zuul-service 服务，请求 http://localhost:5000/hiapi/hi?name=forezp，浏览器只会显示以下内容：

```
hi forezp,i am from port:8762
```

如果你想指定 Url，并且想做负载均衡，那么就需要自己维护负载均衡的服务注册列表。首先，将 ribbon.eureka.enabled 改为 false，即 Ribbon 负载均衡客户端不向 Eureka Client 获取服务注册列表信息。然后需要自己维护一份注册列表，该注册列表对应的服务名为 hiapi-v1（这个名字可自定义），通过配置 hiapi-v1.ribbon.listOfServers 来配置多个负载均衡的 Url。代码如下：

```
zuul:
  routes:
    hiapi:
      path: /hiapi/**
      serviceId: hiapi-v1
ribbon:
  eureka:
    enabled: false
hiapi-v1:
  ribbon:
    listOfServers: http://localhost:8762,http://localhost:8763
```

重新启动 eureka-zuul-service 服务，在浏览器上访问 http://localhost:5000/hiapi/hi?name=forezp，浏览器会显示如下内容：

```
hi forezp,i am from port:8762
hi forezp,i am from port:8763
```

9.3.2 在 Zuul 上配置 API 接口的版本号

如果想给每一个服务的 API 接口加前缀，例如 http://localhost:5000 /v1/hiapi/hi?name=forezp/，即在所有的 API 接口上加一个 v1 作为版本号。这时需要用到 zuul.prefix 的配置，配置示例代码如下：

```
zuul:
  routes:
    hiapi:
      path: /hiapi/**
      serviceId: eureka-client
    ribbonapi:
      path: /ribbonapi/**
      serviceId: eureka-ribbon-client
```

```
    feignapi:
      path: /feignapi/**
      serviceId: eureka-feign-client
zuul.prefix: /v1
```

重新启动 eureka-zuul-service 服务，在浏览器上访问 http://localhost:5000/v1/hiapi /hi?name=forezp，浏览器会显示：

```
hi forezp,i am from port:8762
hi forezp,i am from port:8763
```

9.3.3　在 Zuul 上配置熔断器

Zuul 作为 Netflix 组件，可以与 Ribbon、Eureka 和 Hystrix 等组件相结合，实现负载均衡、熔断器的功能。在默认情况下，Zuul 和 Ribbon 相结合，实现了负载均衡的功能。下面来讲解如何在 Zuul 上实现熔断功能。

在 Zuul 中实现熔断功能需要实现 ZuulFallbackProvider 的接口。实现该接口有两个方法，一个是 getRoute()方法，用于指定熔断功能应用于哪些路由的服务；另一个方法 fallbackResponse()为进入熔断功能时执行的逻辑。ZuulFallbackProvider 的源码如下：

```
public interface ZuulFallbackProvider {
public String getRoute();
     public ClientHttpResponse fallbackResponse();
}
```

实现一个针对 eureka-client 服务的熔断器，当 eureka-client 的服务出现故障时，进入熔断逻辑，向浏览器输入一句错误提示，代码如下：

```
@Component
class MyFallbackProvider implements ZuulFallbackProvider {
     @Override
     public String getRoute() {
         return "eureka-client";
     }
     @Override
     public ClientHttpResponse fallbackResponse() {
         return new ClientHttpResponse() {
             @Override
             public HttpStatus getStatusCode() throws IOException {
                 return HttpStatus.OK;
             }
             @Override
             public int getRawStatusCode() throws IOException {
                 return 200;
             }
```

```
            @Override
            public String getStatusText() throws IOException {
                return "OK";
            }
            @Override
            public void close() {
            }
            @Override
            public InputStream getBody() throws IOException {
                return new ByteArrayInputStream("oooops!error, i'm the fallback.".ge
tBytes());
            }
            @Override
            public HttpHeaders getHeaders() {
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.APPLICATION_JSON);
                return headers;
            }
        };
    }
}
```

重新启动 eureka-zuul-client 工程，并且关闭 eureka-client 的所有实例，在浏览器上访问 http://localhost:5000 /hiapi/hi?name=forezp，浏览器显示：

```
oooops!error, i'm the fallback.
```

如果需要所有的路由服务都加熔断功能，只需要在 getRoute()方法上返回“*”的匹配符，代码如下：

```
    @Override
     public String getRoute() {
         return "*";
     }
```

9.3.4 在 Zuul 中使用过滤器

在前面的章节讲述了过滤器的作用和种类，下面来讲解如何实现一个自定义的过滤器。实现过滤器很简单，只需要继承 ZuulFilter，并实现 ZuulFilter 中的抽象方法，包括 filterType()和 filterOrder()，以及 IZuulFilter 的 shouldFilter()和 Object run()的两个方法。其中，filterType()即过滤器的类型，在前文已经讲解过了，它有 4 种类型，分别是“pre”“post”“routing”和“error”。filterOrder()是过滤顺序，它为一个 Int 类型的值，值越小，越早执行该过滤器。shouldFilter()表示该过滤器是否过滤逻辑，如果为 true，则执行 run()方法；如果为 false，则不执行 run()方法。run()方法写具体的过滤的逻辑。在本例中，检查请求的参数中是否传了

token 这个参数，如果没有传，则请求不被路由到具体的服务实例，直接返回响应，状态码为 401。代码如下：

```
@Component
public class MyFilter extends ZuulFilter {
    private static Logger log = LoggerFactory.getLogger(MyFilter.class);
    @Override
    public String filterType() {
        return PRE_TYPE;
    }
    @Override
    public int filterOrder() {
        return 0;
    }
    @Override
    public boolean shouldFilter() {
        return true;
    }
    @Override
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();
        Object accessToken = request.getParameter("token");
        if(accessToken == null) {
             log.warn("token is empty");
             ctx.setSendZuulResponse(false);
             ctx.setResponseStatusCode(401);
             try {
                 ctx.getResponse().getWriter().write("token is empty");
            }catch (Exception e){}
             return null;
        }
        log.info("ok");
        return null;
    }
}
```

重新启动服务，打开浏览器，访问 http://localhost:5000/hiapi/hi?name=forezp，浏览器显示：

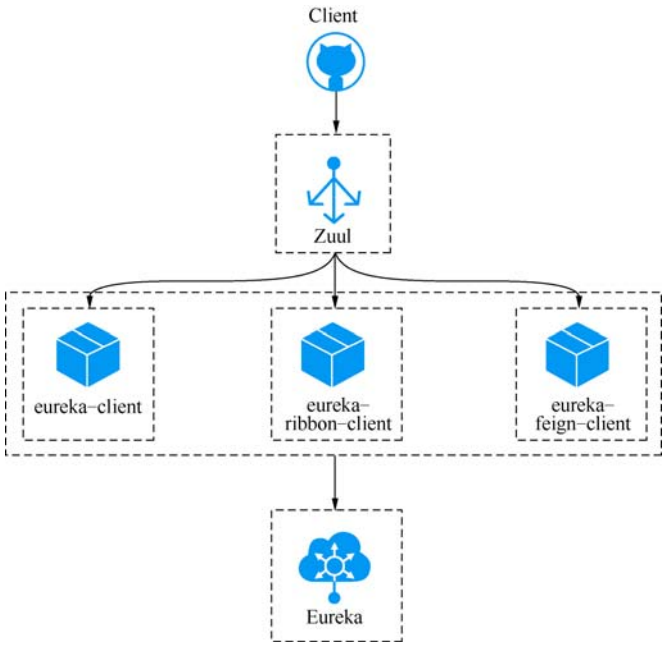
```
token is empty
```

再次在浏览器上输入 http://localhost:5000/hiapi/hi?name=forezp&token=xsddd，即加上了

token 这个请求参数，浏览器显示：

```
hi forezp,i am from port:8762
```

可见，MyFilter 这个 Bean 注入 IoC 容器之后，对请求进行了过滤，并在请求路由转发之前进行了逻辑判断。在实际开发中，可以用此过滤器进行安全验证。本例的架构图如图 9-2 所示。



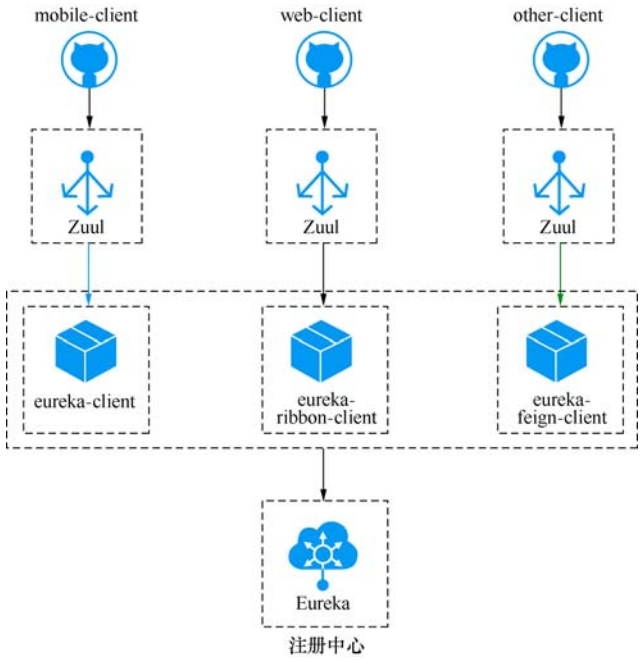
▲图 9-2　本例的架构图

9.3.5　Zuul 的常见使用方式

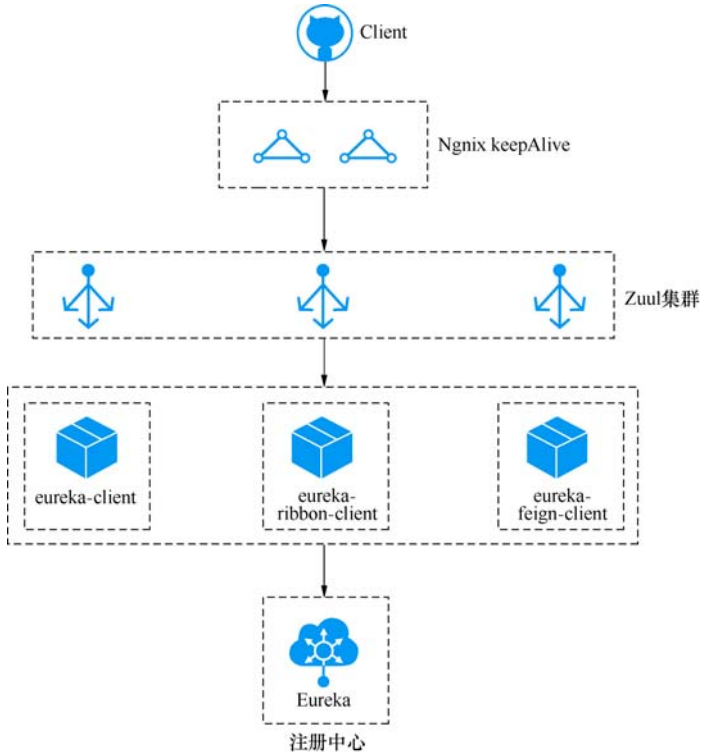
Zuul 是采用了类似于 Spring MVC 的 DispatchServlet 来实现的，采用的是异步阻塞模型，所以性能比 Ngnix 差。由于 Zuul 和其他 Netflix 组件可以相互配合、无缝集成，Zuul 很容易就能实现负载均衡、智能路由和熔断器等功能。在大多数情况下，Zuul 都是以集群的形式存在的。由于 Zuul 的横向扩展能力非常好，所以当负载过高时，可以通过添加实例来解决性能瓶颈。

一种常见的使用方式是对不同的渠道使用不同的 Zuul 来进行路由，例如移动端共用一个 Zuul 网关实例，Web 端用另一个 Zuul 网关实例，其他的客户端用另外一个 Zuul 实例进行路由。这种不同的渠道用不同 Zuul 实例的架构如图 9-3 所示。

另外一种常见的集群是通过 Ngnix 和 Zuul 相互结合来做负载均衡。暴露在最外面的是 Ngnix 主从双热备进行 Keepalive，Ngnix 经过某种路由策略，将请求路由转发到 Zuul 集群上，Zuul 最终将请求分发到具体的服务上。架构图如图 9-4 所示。



▲图 9-3　Zuul 通过不同的渠道来集群



▲图 9-4　Ngnix 和 Zuul 结合进行负载均衡

第 10 章　配置中心 Spring Cloud Config

前面的章节详细讲解了 Spring Cloud Netflix 组件，包括服务注册和发现组件 Eureka、负载均衡组件 Ribbon、声明式调用 Feign、熔断器组件 Hystrix 和路由网关组件 Zuul。本章讲述 Spring Cloud 的另一组件——分布式配置中心 Spring Cloud Config。

本章以案例的形式来全面讲解 Spring Cloud Config 的知识，分为以下 4 个方面。

- Config Server 从本地读取配置文件。
- Config Server 从远程 Git 仓库读取配置文件。
- 搭建高可用 Config Server 集群。
- 使用 Spring Cloud Bus 刷新配置。

10.1 Config Server 从本地读取配置文件

Config Server 可以从本地仓库读取配置文件，也可以从远处 Git 仓库读取。本地仓库是指将所有的配置文件统一写在 Config Server 工程目录下。Config Sever 暴露 Http API 接口，Config Client 通过调用 Config Sever 的 Http API 接口来读取配置文件。

为了讲解得更清楚，本章将不在之前章节的工程的基础上改造，而是重新建工程。和之前的工程一样，采用多 Module 形式。需要新建一个主 Maven 工程，创建过程同 5.2 节，这里不再重复。主要指定了 Spring Boot 的版本为 1.5.3，Spring Cloud 版本为 Dalston.RELEASE。

10.1.1 构建 Config Server

在主 Maven 工程下，创建一个 Module 工程，工程名取为 config-server，其 pom 文件继承主 Maven 工程的 pom 文件，并在 pom 文件中引入 Config Server 的起步依赖 spring-cloud-config-server。pom 文件代码如下：

```
<parent>
    <groupId>com.forezp</groupId>
    <artifactId>chapter10</artifactId>
    <version>1.0-SNAPSHOT</version>
</parent>
<dependencies>
```

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

在程序的启动类 ConfigServerApplication 加上@EnableConfigServer 注解，开启 Config Server 的功能，代码如下：

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

在工程的配置文件 application.yml 中做相关的配置，包括指定程序名为 config-server，端口号为 8769。通过 spring. profiles.active=native 来配置 Config Server 从本地读取配置，读取配置的路径为 classpath 下的 shared 目录。application.yml 配置文件的代码如下：

```
spring:
  cloud:
    config:
      server:
        native:
          search-locations: classpath:/shared
  profiles:
     active: native
  application:
    name: config-server
server:
  port: 8769
```

在工程的 Resources 目录下建一个 shared 文件夹，用于存放本地配置文件。在 shared 目录下，新建一个 config-client-dev.yml 文件，用作 eureka-client 工程的 dev（开发环境）的配置文件。在 config-client-dev.yml 配置文件中，指定程序的端口号为 8762，并定义一个变量 foo，该变量的值为 foo version 1。代码如下：

```
server:
  port: 8762
foo: foo version 1
```

10.1.2　构建 Config Client

新建一个工程，取名为 config-client，该工程作为 Config Client 从 Config Server 读取配置

文件，该工程的 pom 文件继承了主 Maven 工程的 pom 文件，并在其 pom 文件引入 Config 的起步依赖 spring-cloud-starter-config 和 Web 功能的起步依赖 spring-boot-starter-web。代码如下：

```
    <parent>
        <groupId>com.forezp</groupId>
        <artifactId>chapter10</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-config</artifactId>
        </dependency>
    </dependencies>
```

在其配置文件 bootstrap.yml 中做程序的配置，注意这里用的是 bootstrap.yml，而不是 application.yml，bootstrap 相对于 application 具有优先的执行顺序。在 bootstrap.yml 配置文件中指定了程序名为 config-client，向 Url 地址为 http://localhost:8769 的 Config Server 读取配置文件。如果没有读取成功，则执行快速失败（fail-fast），读取的是 dev 文件。bootstrap.yml 配置文件中的变量{spring. application.name}和变量{spring.profiles.active}，两者以“-”相连，构成了向 Config Server 读取的配置文件名，所以本案例在配置中心读取的配置文件名为 config-client-dev.yml 文件。配置文件 bootstrap.yml 的代码如下：

```
spring:
  application:
    name: config-client
  cloud:
    config:
      uri: http://localhost:8769
      fail-fast: true
  profiles:
    active: dev
```

eureka-server 工程启动成功后，启动 eureka-client 工程，你会在控制台的日志中发现 eureka-client 向 Url 地址为 http://localhost:8769 的 Config Server 读取了配置文件。最终程序启动的端口为 8762，这个端口是在 eureka-server 的 Resouces/shared 目录中的 eureka-client-dev.yml 的配置文件中配置的，可见 eureka-client 成功地向 eureka-server 读取了配置文件。

为了进一步验证，在 eureka-clien 工程写一个 API 接口，读取配置文件的 foo 变量，并通过 API 接口返回，代码如下：

```
@SpringBootApplication
@RestController
public class ConfigClientApplication {
     public static void main(String[] args) {
          SpringApplication.run(ConfigClientApplication.class, args);
     }
     @Value("${foo}")
     String foo;
     @RequestMapping(value = "/foo")
     public String hi(){
          return foo;
     }
}
```

打开浏览器，访问 http://localhost:8762/foo，浏览器显示：

```
foo version 1
```

可见 eureka-client 工程成功地向 eureka-server 工程读取了配置文件中 foo 变量的值。

10.2 Config Server 从远程 Git 仓库读取配置文件

Spring Cloud Config 支持从远程 Git 仓库读取配置文件，即 Config Server 可以不从本地的仓库读取，而是从远程 Git 仓库读取。这样做的好处就是将配置统一管理，并且可以通过 Spring Cloud Bus 在不人工启动程序的情况下对 Config Client 的配置进行刷新。本例采用 GitHub 作为远程 Git 仓库。

首先，修改 Config Server 的配置文件 application.yml，代码如下：

```
server:
  port: 8769
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/forezp/SpringcloudConfig
          searchPaths: respo
          username: miles02@163.com
          password:
        label: master
  application:
    name: config-server
```

其中，uri 为远程 Git 仓库的地址，serachPaths 为搜索远程仓库的文件夹地址，username

和 password 为 Git 仓库的登录名和密码。如果是私人 Git 仓库，登录名和密码是必须的；如果是公开的 Git 仓库，可以不需要。label 为 git 仓库的分支名，本例从 master 读取。

将上一节的 eureka-client-dev.yml 上传到远程仓库中，上传的路径为 https://github.com/forezp/SpringcloudConfig。读者可以自己申请 GitHub 账号，并在 GitHub 上创建代码仓库，将 eureka-client-dev.yml 上传到自己的仓库。

重新启动 config-server，config-server 启动成功后，启动 config-client，可以发现 config-client 的端口为 8762。像上一节一样，访问 http://localhost:8762/foo，浏览器显示：

```
foo version 1
```

可见，config-server 从远程 Git 仓库读取了配置文件，config-client 从 config-server 读取了配置文件。

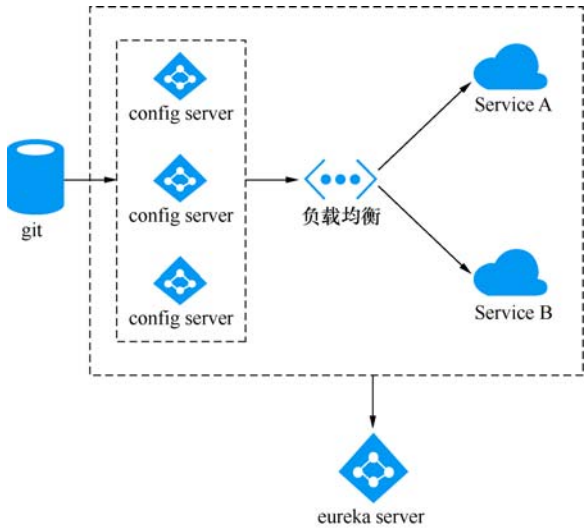
10.3 构建高可用的 Config Server

在上一节讲解了 Config Client 如何从配置中心 Config Server 读取配置文件，配置中心如何从远程 Git 仓库读取配置文件。当服务实例很多时,所有的服务实例需要同时从配置中心 Config Server 读取配置文件，这时可以考虑将配置中心 Config Server 做成一个微服务，并且将其集群化，从而达到高可用。配置中心 Config Server 高可用的架构图如图 10-1 所示。Config Server 和 Config Client 向 Eureka Server 注册，且将 Config Server 多实例集群部署。

10.3.1 构建 Eureka Server

新建一个 eureka-server 工程，eureka-server 工程的 pom 文件继承了主 Maven 的 pom 文件，eureka-server 工程的 pom 文件引入 Eureka Server 的起步依赖 spring-cloud-starter-eureka-server 和 Web 功能的起步依赖 spring-boot- starter-web，配置代码如下：

```
<parent>
    <groupId>com.forezp</groupId>
    <artifactId>chapter10</artifactId>
    <version>1.0-SNAPSHOT</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka-server</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```



▲图 10-1　高可用的 Config Server

在工程的配置文件 application.yml 中做程序的相关配置，指定程序的端口号为 8761，并不自注册（即将配置 register-with-eureka 和 fetch-registry 设置为 false），代码如下：

```
server:
  port: 8761
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
    serviceUrl:
      defaultZone: http://localhost:${server.port}/eureka/
```

在程序的启动类 EurekaServerApplication 上加@EnableEurekaServer 注解，开启 Eureka Server 的功能，代码如下：

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

10.3.2　改造 Config Server

Config Server 作为 Eureka Client，需要在工程中的 pom 文件引入 spring-cloud-starter-eureka 起步依赖，代码如下：

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

在工程的启动类 ConfigServerApplication 加上@EnableEurekaClient 注解，开启 EurekaClient 的功能，代码如下：

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

在工程的配置文件 application.yml 文件中制定服务注册的地址，代码如下：

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

10.3.3 改造 Config Client

和 Config Server 一样作为 Eureka Client，在 pom 文件加上 spring-cloud-starter-eureka 起步依赖，在工程的启动类加上@EnableEurekaClient 注解，开启 EurekaClient 的功能。

在工程的配置文件 application.yml 加上相关配置，指定服务注册中心的地址为 http://localhost:8761/eureka/，向 Service Id 为 config-server 的配置服务读取配置文件，代码如下：

```
spring:
  application:
    name: config-client
  cloud:
    config:
      fail-fast: true
      discovery:
        enabled: true
        serviceId: config-server

  profiles:
    active: dev

server:
  port: 8762
```

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

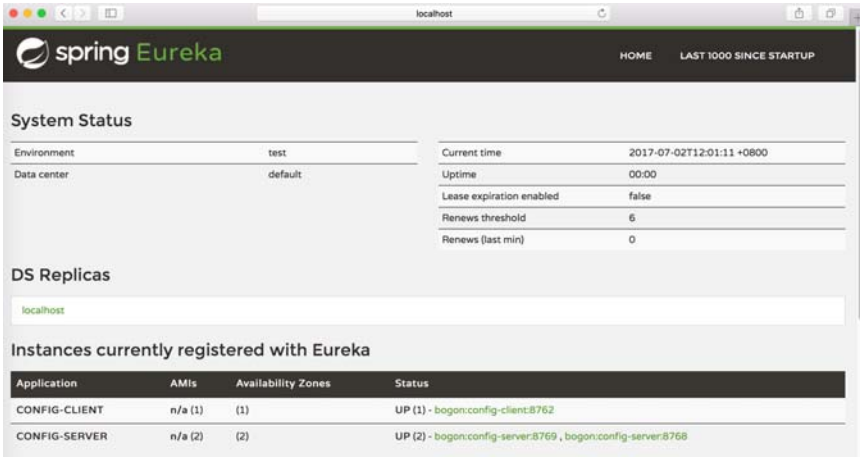
依次启动 eureka-server、config-server 和 config-client 工程，注意这里需要 config-server 启动成功并且向 eureka-server 注册完成后，才能启动 config-client，否则 config-client 找不到 config-server。

通过控制台可以发现，config-client 向地址为 http://localhost:8769 的 config-server 读取了配置文件。访问 http://localhost:8762/foo，浏览器显示：

```
foo version 1
```

可见，config-server 从远程 Git 仓库读取了配置文件，config-client 从 config-server 读取了配置文件。

那么如何搭建高可用的 Config Server 呢？只需要将 Config Server 多实例部署，用 IDEA 开启多个 Config Server 实例，端口分别为 8769 和 8768。在浏览器上访问 Eureka Server 的主页 http://localhost:8761/，界面如图 10-2 所示。



▲图 10-2　Eureka 界面

多次启动 config-client 工程，从控制台可以发现它会轮流地从 http://localhost:8768 和 http://localhost:8769 的 Config Server 读取配置文件，并且做了负载均衡。

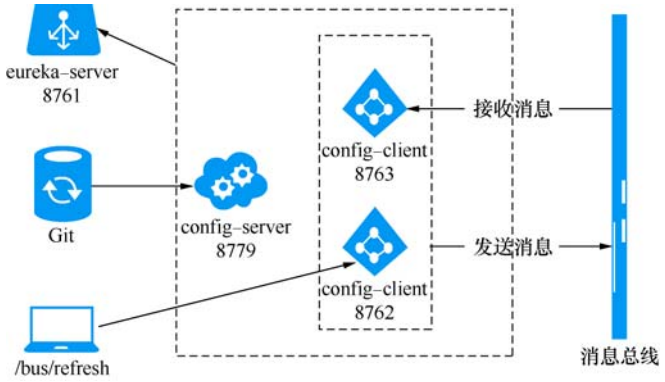
10.4 使用 Spring Cloud Bus 刷新配置

Spring Cloud Bus 是用轻量的消息代理将分布式的节点连接起来，可以用于广播配置文件的更改或者服务的监控管理。一个关键的思想就是，消息总线可以为微服务做监控，也可以实

现应用程序之间相互通信。Spring Cloud Bus 可选的消息代理组建包括 RabbitMQ、AMQP 和 Kafka 等。本节讲述的是用 RabbitMQ 作为 Spring Cloud 的消息组件去刷新更改微服务的配置文件。

为什么需要用 Spring Cloud Bus 去刷新配置呢？

如果有几十个微服务，而每一个服务又是多实例，当更改配置时，需要重新启动多个微服务实例，会非常麻烦。Spring Cloud Bus 的一个功能就是让这个过程变得简单，当远程 Git 仓库的配置更改后，只需要向某一个微服务实例发送一个 Post 请求，通过消息组件通知其他微服务实例重新拉取配置文件。如图 10-3 所示，当远程 Git 仓库的配置更改后，通过发送“/bus/refresh”Post 请求给某一个微服务实例，通过消息组件，通知其他微服务实例，更新配置文件。



▲图 10-3　消息总线更新微服务的配置

本节是在上一节的例子上进行改造的，只需要改造 config-client 工程。首先，需要在 pom 文件中引入用 RabbitMQ 实现的 Spring Cloud Bus 的起步依赖 spring-cloud-starter-bus-amqp。如果读者需要自己实践，则需要安装 RabbitMQ 服务器。pom 文件添加的依赖如下：

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

在工程的配置文件 application.yml 添加 RabbitMQ 的相关配置，host 为 RabbitMQ 服务器的 IP 地址，port 为 RabbitMQ 服务器的端口，username 和 password 为 RabbitMQ 服务器的用户名和密码。通过消息总线更改配置，需要经过安全验证，为了方便讲述，先把安全验证屏蔽掉，也就是将 management. security. enabled 改为 false。代码清单如下：

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
```

```
    username: guest
    password: guest
management:
  security:
    enabled: false
```

最后，需要在更新的配置类上加@ RefreshScope 注解，只有加上了该注解，才会在不重启服务的情况下更新配置，如本例中更新配置文件 foo 变量的值。代码清单如下：

```
@RestController
@RefreshScope
public class ConfigClientApplication {
    @Value("${foo}")
    String foo;
    @GetMapping (value = "/foo")
     public String hi(){
        return foo;
     }
}
```

依次启动工程，其中 config-client 开启两个实例，端口分别为 8762 和 8763。启动完成后，在浏览器上访问 http://localhost:8762/foo 或者 http://localhost:8763/foo ，浏览器显示：

```
foo version 1
```

更改远程 Git 仓库，将 foo 的值改为“foo version 2”。通过 Postman 或者其他工具发送一个 Post 请求 http://localhost:8762/bus/refresh，请求发送成功，再访问 http://localhost:8762/foo 或者 http://localhost:8763/foo，浏览器都会显示：

```
foo version 2
```

可见，通过向 8762 端口的微服务实例发送 Post 请求 http://localhost:8762/bus/refresh，请求刷新配置，由于使用了 Spring Cloud Bus，其他服务实例（如案例中的 8763 端口的服务实例）会接收到刷新配置的消息，也会刷新配置。另外，“a/bus/refresh”API 接口可以指定服务，即使用“destination”参数，例如“/bus/refresh?destination=eureka-client:**”，即刷新服务名为 eureka-client 的所有服务实例。

第 11 章　服务链路追踪 Spring Cloud Sleuth

Spring Cloud Sleuth 是 Spring Cloud 的一个组件，它的主要功能是在分布式系统中提供服务链路追踪的解决方案。

11.1 为什么需要 Spring Cloud Sleuth

微服务架构是一个分布式架构，微服务系统按业务划分服务单元，一个微服务系统往往有很多个服务单元。由于服务单元数量众多，业务的复杂性较高，如果出现了错误和异常，很难去定位。主要体现在一个请求可能需要调用很多个服务，而内部服务的调用复杂性决定了问题难以定位。所以在微服务架构中，必须实现分布式链路追踪，去跟进一个请求到底有哪些服务参与，参与的顺序又是怎样的，从而达到每个请求的步骤清晰可见，出了问题能够快速定位的目的。

以第 2 章的图 2-8 为例来说明，在微服务系统中，一个来自用户的请求先到达前端 A（如前端界面），然后通过远程调用，到达系统的中间件 B、C（如负载均衡、网关等），最后到达后端服务 D、E，后端经过一系列的业务逻辑计算，最后将数据返回给用户。对于这样一个请求，经历了这么多个服务，怎么样将它的请求过程用数据记录下来呢？这就需要用到服务链路追踪。

Google 开源了 Dapper 链路追踪组件，并在 2010 年发表了论文《Dapper, a Large-Scale Distributed Systems Tracing Infrastructure》，这篇论文是业内实现链路追踪的标杆和理论基础，具有很高的参考价值。

目前，常见的链路追踪组件有 Google 的 Dapper、Twitter 的 Zipkin，以及阿里的 Eagleeye（鹰眼）等，它们都是非常优秀的链路追踪开源组件。

本章主要讲述如何在 Spring Cloud Sleuth 中集成 Zipkin。在 Spring Cloud Sleuth 中集成 Zipkin 非常简单，只需要引入相应的依赖并做相关的配置即可。

11.2 基本术语

Spring Cloud Sleuth 采用了 Google 的开源项目 Dapper 的专业术语。

（1）Span：基本工作单元，发送一个远程调度任务就会产生一个 Span，Span 是用一个 64 位 ID 唯一标识的，Trace 是用另一个 64 位 ID 唯一标识的。Span 还包含了其他的信息，例如摘要、时间戳事件、Span 的 ID 以及进程 ID。

（2）Trace：由一系列 Span 组成的，呈树状结构。请求一个微服务系统的 API 接口，这个 API 接口需要调用多个微服务单元，调用每个微服务单元都会产生一个新的 Span，所有由这个请求产生的 Span 组成了这个 Trace。

（3）Annotation：用于记录一个事件，一些核心注解用于定义一个请求的开始和结束，这些注解如下。

- cs-Client Sent：客户端发送一个请求，这个注解描述了 Span 的开始。
- sr-Server Received：服务端获得请求并准备开始处理它，如果将其 sr 减去 cs 时间戳，便可得到网络传输的时间。
- ss-Server Sent：服务端发送响应，该注解表明请求处理的完成（当请求返回客户端），用 ss 的时间戳减去 sr 时间戳，便可以得到服务器请求的时间。
- cr-Client Received：客户端接收响应，此时 Span 结束，如果 cr 的时间戳减去 cs 时间戳，便可以得到整个请求所消耗的时间。

11.3 案例讲解

基本知识讲解完毕，下面来进行案例实战。本章的案例一共有 4 个工程，和之前的工程一样，采用 Maven 工程的多 Module 形式。新建一个主 Maven 工程，创建过程同 5.2 节，在此不再重复。在主 Maven 工程的 pom 文件里指定 Spring Boot 的版本为 1.5.3，Spring Cloud 版本为 Dalston.RELEASE。eureka-server 工程作为服务注册中心，它的创建过程同 5.2 节。zipkin-server 作为链路追踪服务中心，负责存储链路数据。gateway-service 作为服务网关工程，负责请求的转发，同时它也作为链路追踪客户端，负责产生链路数据，并上传给 zipkin-service。user-service 是一个服务提供者，对外暴露 API 接口，同时它也作为链路追踪客户端，负责产生链路数据。

11.3.1 构建 Zipkin Server

新建一个 Module 工程，取名为 zipkin-server，工程的 pom 文件继承了主 Maven 工程的 pom 文件，并引入 Eureka 的起步依赖 spring-cloud-starter-eureka、Zipkin Server 的依赖 zipkin-server，以及 Zipkin Server 的 UI 界面依赖 zipkin-autoconfigure-ui。后两个依赖提供了 Zipkin 的功能和 Zipkin 界面展示的功能，代码如下：

```
<parent>
    <groupId>com.forezp</groupId>
    <artifactId>sleuth</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</parent>
<dependencies>
```

```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka</artifactId>
        </dependency>
        <dependency>
            <groupId>io.zipkin.java</groupId>
            <artifactId>zipkin-server</artifactId>
        </dependency>
        <dependency>
            <groupId>io.zipkin.java</groupId>
            <artifactId>zipkin-autoconfigure-ui</artifactId>
        </dependency>
    </dependencies>
```

在程序的启动类 ZipkinServiceApplication 加上@EnableZipkinServer 注解，开启 ZipkinServer 的功能，加上@EnableEurekaClient 注解，启动 Eureka Client，代码如下：

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

在程序的配置文件 application.yml 文件做程序的相关配置，指定程序名为 zipkin-server，端口号为 9411，服务注册地址为 http://localhost:8761/eureka/，代码如下：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
server:
  port: 9411
spring:
  application:
    name: zipkin-server
```

到此为止，Zipkin Server 就集成完毕了。

11.3.2 构建 User Service

在主 Maven 工程下建一个 Module 工程，取名为 user-service，作为服务提供者，对外暴露 API 接口。user-service 工程的 pom 文件继承了主 Maven 工程的 pom 文件，并引入了 Eureka 的起步依赖 spring-cloud-starter-eureka、Web 起步依赖 spring-boot-starter-web，以及 Zipkin 的起

步依赖 spring-cloud-starter-zipkin，代码如下：

```
<dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-zipkin</artifactId>
            <version>RELEASE</version>
        </dependency>
</dependencies>
```

在程序的配置文件 applicatiom.yml 中，指定程序名为 user-service，端口号为 8762，服务注册地址 http://localhost:8761/eureka/，Zipkin Server 地址为 http://localhost:9411。spring.sleuth.sampler.percentage 为 1.0，即以 100%的概率将链路的数据上传给 Zipkin Server，在默认情况下，该值为 0.1。配置文件代码如下：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
server:
  port: 8762
spring:
  application:
    name: user-service
  zipkin:
    base-url: http://localhost:9411
  sleuth:
    sampler:
      percentage: 1.0
```

在 UserController 类建一个“/user/hi”的 API 接口，对外提供服务，代码如下：

```
@RestController
@RequestMapping("/user")
public class UserController {
     @GetMapping("/hi")
     public String hi(){
```

```
            return "I'm forezp";
        }
    }
```

最后作为 Eureka Client，需要在程序的启动类 UserServiceApplication 加上@EnableEurekaClient 注解，开启 Eureka Client 的功能。

11.3.3 构建 Gateway Service

新建一个名为 gateway-service 的工程，这个工程作为服务网关，将请求转发到 user-service。作为 Zipkin 客户端，需要将链路数据上传给 Zipkin Server，同时它也作为 Eureka Client。在工程的 pom 文件中除了需要继承主 Maven 工程的 pom 文件，还需引入如下的依赖，代码清单如下：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zuul</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zipkin</artifactId>
        <version>RELEASE</version>
    </dependency>
</dependencies>
```

在工程的配置文件 application.yml 中，配置程序名为 gateway-service，端口号为 5000，服务注册地址为 http://localhost:8761/eureka/，Zipkin Server 地址为 http://localhost:9411，以“/user-api/**”开头的 Uri 请求转发到服务名为 user-service 的服务，配置代码如下：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
server:
  port: 5000
spring:
```

```
    application:
      name: gateway-service
    sleuth:
      sampler:
        percentage: 1.0
    zipkin:
      base-url: http://localhost:9411
  zuul:
    routes:
      api-a:
        path: /user-api/**
        serviceId: user-service
```

在程序的启动类 GatewayServiceApplication 上加@EnableEurekaClient 注解，开启 Eureka Client 的功能，加上@EnableZuulProxy 注解，开启 Zuul 代理功能，代码如下：

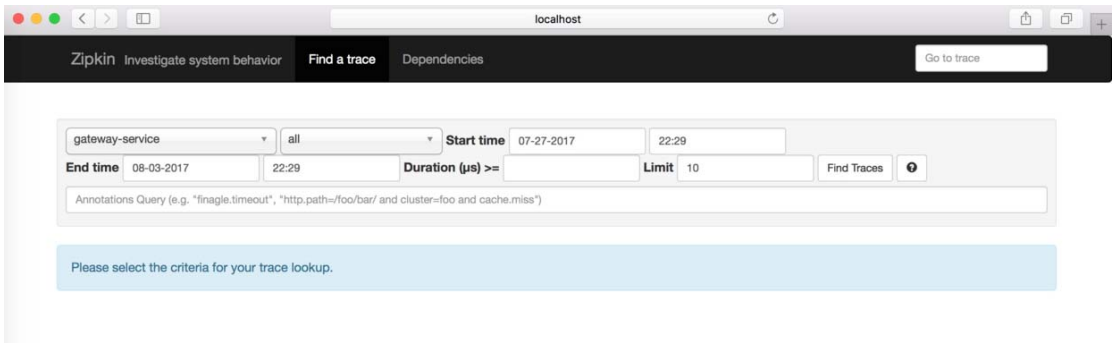
```
@SpringBootApplication
@EnableZuulProxy
@EnableEurekaClient
public class GatewayServiceApplication {
   public static void main(String[] args) {
      SpringApplication.run(GatewayServiceApplication.class, args);
   }
}
```

11.3.4　项目演示

完整的项目搭建完毕，依次启动 eureka-server、zipkin-server、user-service 和 gateway-service。在浏览器上访问 http://localhost:5000/user-api/user/hi，浏览器显示如下：

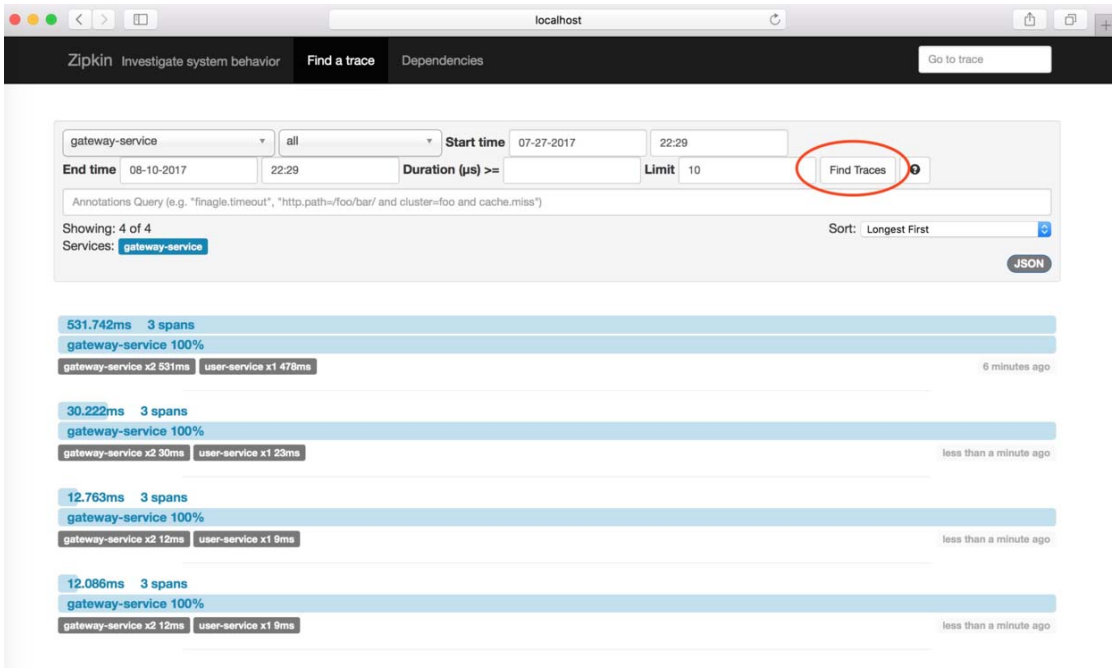
```
I'm forezp
```

访问 http://localhost:9411，即访问 Zipkin 的展示界面，界面显示如图 11-1 所示。



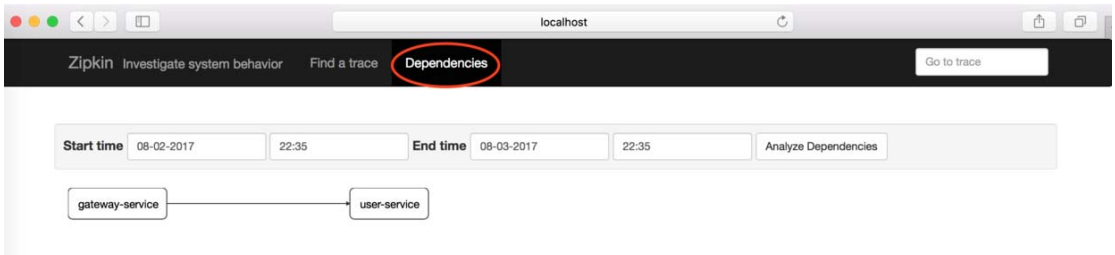
▲图 11-1　Zipkin Server 的主页

这个界面用于展示 Zipkin Server 收集的链路数据，可以根据服务名、开始时间、结束时间、请求消耗的时间等条件来查找。单击“Find Tracks”按钮，界面如图 11-2 所示，从图中可知请求的调用情况，例如请求的调用时间、消耗时间，以及请求调用的链路情况。



▲图 11-2　服务的链路数据展示

单击“Dependences”按钮，可以查看服务的依赖关系。在本案例中，gateway-service 将请求转发到了 user-service，这两个服务的依赖关系如图 11-3 所示。



▲图 11-3　服务的依赖关系

11.4 在链路数据中添加自定义数据

现在需要实现这样一个功能：在链路数据中加上请求的操作人。本案例在 gateway-service 服务中实现。在 gateway-service 工程里新建一个 ZuulFilter 过滤器，它的类型为 post 类型，order

为 900，开启拦截。在过滤器的拦截逻辑方法里，通过 Tracer 的 addTag 方法加上自定义的数据，在本案例中加上了链路的操作人。另外也可以在这个过滤器中获取当前链路的 traceId 信息，traceId 作为链路数据的唯一标识，可以存储在 log 日志中，方便后续查找，本案例只是将 traceId 信息简单地打印在控制台上。代码如下：

```
@Component
public class LoggerFilter extends ZuulFilter {
     @Autowired
     Tracer tracer;
     @Override
     public String filterType() {
          return FilterConstants.POST_TYPE;
     }
     @Override
     public int filterOrder() {
          return 900;
     }
     @Override
     public boolean shouldFilter() {
          return true;
     }
     @Override
     public Object run() {
          tracer.addTag("operator","forezp");
          System.out.print(tracer.getCurrentSpan().traceIdString());
          return null;
     }
}
```

11.5 使用 RabbitMQ 传输链路数据

在上述案例中，最终 gateway-service 收集的数据是通过 Http 上传给 zipkin-server 的。在 Spring Cloud Sleuth 中支持消息组件来传输链路数据，本节使用 RabbitMQ 来传输链路数据。

首先改造 zipkin-server 工程，在其 pom 文件中将 zipkin-server 的依赖去掉，加上 spring-cloud-sleuth-zipkin-stream 和 spring-cloud-starter-stream-rabbit 的依赖，代码如下：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin-stream</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
```

```
</dependency>
```

在工程的配置文件 application.yml 加上 RabbitMQ 的配置，包括 host、端口、用户名、密码，代码如下：

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

在程序的启动类 ZipkinServerApplication 中加上@EnableZipkinStreamServer 注解，开启 ZipkinStreamServer，代码如下：

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

现在来改造 Zipkin Client（包括 gateway-service 工程和 user-service 工程），在它们的 pom 文件中将 spring-cloud-starter-zipkin 依赖改为 spring-cloud-sleuth-zipkin-stream 和 spring-cloud-starter-stream-rabbit，代码如下：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin-stream</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

同时在配置文件 applicayion.yml 加上 RabbitMQ 的配置，同 zipkin-server 工程。这样，就将链路的上传数据从 Http 改为用消息代组件 RabbitMQ。

11.6 在 MySQL 数据库中存储链路数据

在上面的例子中，Zipkin Server 将数据存储在内存中，一旦程序重启，之前的链路数据全部丢失，那么怎么将链路数据存储起来呢？Zipkin 支持将链路数据存储在 MySQL、Elasticsearch 和 Cassandra 数据库中。本节讲解使用 MySQL 存储，下一节讲解用 Elasticsearch 存储。

Zipkin Client 有两种方式将链路数据传输到 Zipkin Server 中，一种是使用 Http，另一种是使用 RabbitMQ。Zipkin Server 通过这两种方式来收集链路数据，并存储在 MySQL 中。这两种方式在具体的编码实现上有着较大的差别，所以分开来讲解。

11.6.1　使用 Http 传输链路数据，并存储在 MySQL 数据库中

本节的案例是在第 11.3 节案例的基础上进行改造的，只需要改造 zipkin-server 工程。在 zipkin-server 工程的 pom 文件加上 Zipkin Server 的依赖 zipkin-server、Zipkin 的 MySQL 存储依赖 zipkin-storage-mysql（这两个依赖的版本都为 1.19.0）、Zipkin Server 的 UI 界面依赖 zipkin-autoconfigure-ui、MySQL 的连接器依赖 mysql-connector-java 和 JDBC 的起步依赖 spring-boot-starter-jdbc。代码如下：

```
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-server</artifactId>
     <version>1.19.0</version>
</dependency>

<dependency>
     <groupId>io.zipkin.java</groupId>
     <artifactId>zipkin-storage-mysql</artifactId>
     <version>1.19.0</version>
</dependency>

<dependency>
     <groupId>io.zipkin.java</groupId>
     <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>

<dependency>
     <groupId>mysql</groupId>
     <artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

在 zipkin-server 工程的配置文件 application.yml 中加上数据源的配置，包括数据库的 Url、用户名、密码和连接驱动，并且需要配置 zipkin.storage.type 为 mysql，代码如下：

```
spring:
  datasource:
    url:jdbc:mysql://localhost:3306/spring-cloud-zipkin?useUnicode=true&characterEnco
ding=utf8&useSSL=false
```

```
    username: root
    password: 123456
    driver-class-name: com.mysql.jdbc.Driver
zipkin:
  storage:
type: mysql
```

另外需要在 MySQL 数据库中初始化数据库脚本，数据库脚本地址为 https://github.com/openzipkin/ zipkin/blob/master/zipkinstorage/mysql/src/main/resources/mysql.sql。本案例中的源码资源文件夹中也包含该数据库脚本。

最后需要在程序的启动类 ZipkinServerApplication 中注入 MySQLStorage 的 Bean，代码如下：

```
@Bean
public MySQLStorage mySQLStorage(DataSource datasource) {
return MySQLStorage.builder().datasource(datasource).executor(Runnable::run).build();
}
```

只需要上述步骤，即可将使用 Http 传输的链路数据存储在 MySQL 数据库中。

11.6.2　使用 RabbitMQ 传输链路数据，并存储在 MySQL 数据库中

本节的案例是在第 11.5 节案例的基础上进行改造的，只需要改造 zipkin-server 的工程。在 zipkin-server 工程的 pom 文件中加上 MySQL 的连接器依赖 mysql-connector-java，JDBC 的起步依赖 spring-boot-starter-jdbc，代码如下：

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

在 zipkin-server 工程的配置文件 application.yml 中加上数据源的配置，包括数据库的 Url、用户名、密码、连接驱动，另外需要配置 zipkin.storage.type 为 mysql，代码如下：

```
spring:
  datasource:
    url:jdbc:mysql://localhost:3306/spring-cloud-zipkin?useUnicode=true&characterEnco
ding=utf8&useSSL=false
    username: root
    password: 123456
```

```
    driver-class-name: com.mysql.jdbc.Driver
zipkin:
  storage:
    type: mysql
```

另外需要在 MySQL 数据库中初始化数据库脚本，具体同上一节。

11.7 在 ElasticSearch 中存储链路数据

在并发高的情况下，使用 MySQL 存储链路数据显然不合理，这时可以选择使用 ElasticSearch 存储。读者需要自行安装 ElasticSearch 和 Kibana（下一节中使用），下载地址为 https://www.elastic.co/products/elasticsearch。安装完成后启动，其中 ElasticSearch 的默认端口号为 9200，Kibana 的默认端口号为 5601。

本节的案例在 11.3.6 节案例的基础上进行改造。首先在 pom 文件中加上 zipkin 的依赖和 zipkin-autoconfigure-storage-elasticsearch-http 的依赖，代码如下：

```
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin</artifactId>
    <version>1.28.0</version>
</dependency>
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-storage-elasticsearch-http</artifactId>
    <version>1.28.0</version>
</dependency>
```

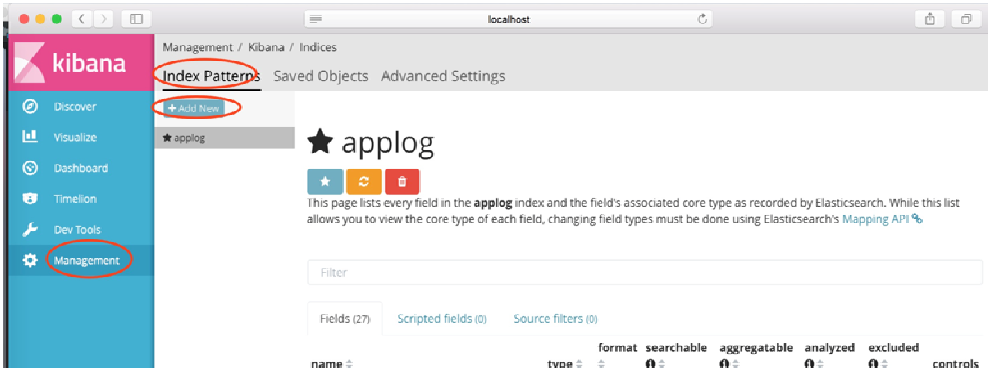
在程序的配置文件 application.yml 中加上 Zipkin 的配置，配置了 zipkin 的存储类型（type）为 elasticsearch，使用的存储组件（StorageComponent）为 elasticsearch，然后需要配置 elasticsearch，包括 hosts（可以配置多个，用“,”隔开）和 index 等。具体配置代码如下：

```
zipkin:
  storage:
    type: elasticsearch
    StorageComponent: elasticsearch
    elasticsearch:
      cluster: elasticsearch
      max-requests: 30
      index: zipkin
      index-shards: 3
      index-replicas: 1
      hosts: localhost:9200
```

只需要这些配置，Zipkin Server 的链路数据就存储在 ElasticSearch 中了。

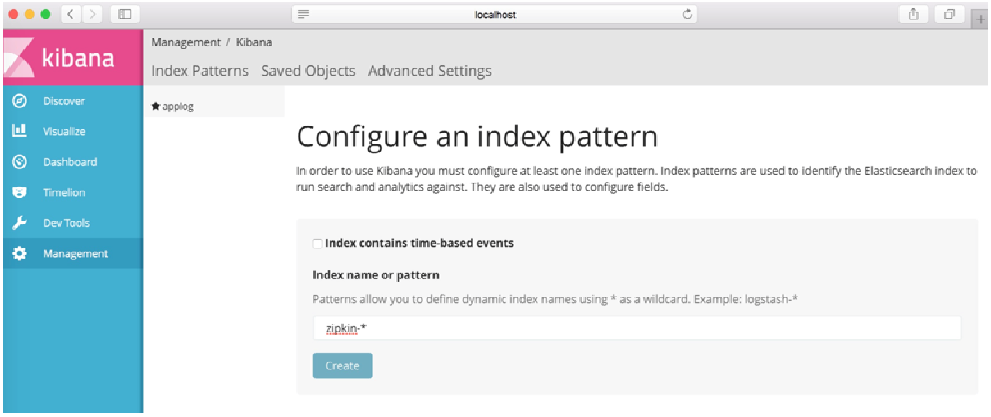
11.8 用 Kibana 展示链路数据

上一节讲述了如何将链路数据存储在 ElasticSearch 中，ElasticSearch 可以和 Kibana 结合，将链路数据展示在 Kibana 上。安装完成 Kibana 后启动，Kibana 默认会向本地端口为 9200 的 ElasticSearch 读取数据。Kibana 默认的端口为 5601，访问 Kibana 的主页 http://localhost:5601，其界面如图 11-4 所示。



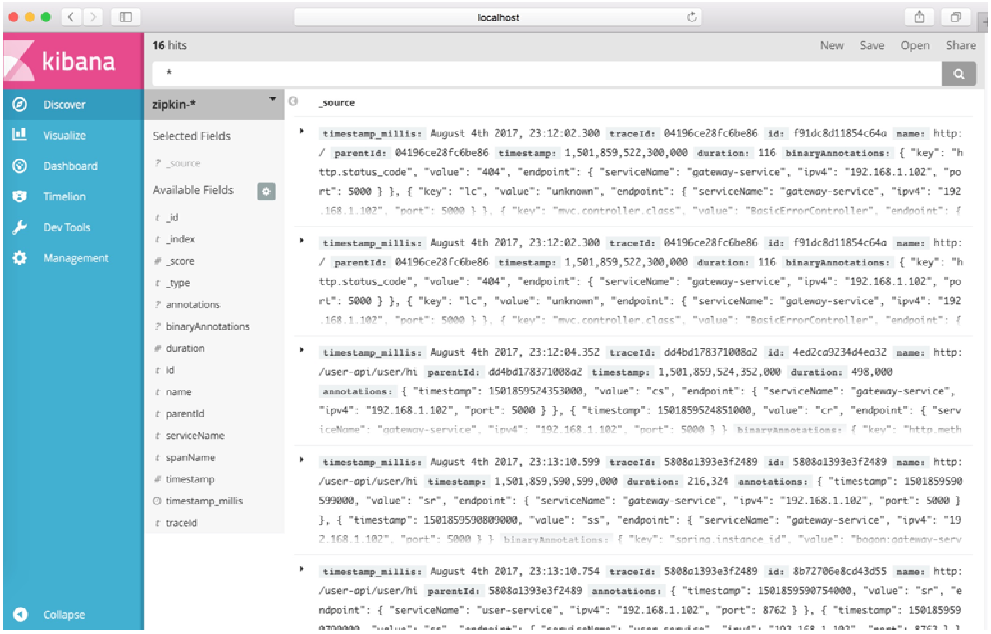
▲图 11-4　Kibana 的主页界面

在图 11-4 的界面中，单击“Management”按钮，然后单击“Add New”，添加一个 index。我们将在上节 ElasticSearch 中写入链路数据的 index 配置为“zipkin”，那么在界面填写为“zipkin-*”，单击“Create”按钮，界面如图 11-5 所示。



▲图 11-5　创建“zipkin”的 Index 界面

创建完成 index 后，单击“Discover”，就可以在界面上展示链路数据了，展示界面如图 11-6 所示。



▲图 11-6　Kibana 展示链路数据界面

第 12 章　微服务监控 Spring Boot Admin

Spring Boot Admin 用于管理和监控一个或者多个 Spring Boot 程序。Spring Boot Admin 分为 Server 端和 Client 端，Client 端可以通过 Http 向 Server 端注册，也可以结合 Spring Cloud 的服务注册组件 Eureka 进行注册。Spring Boot Admin 提供了用 AngularJs 编写的 UI 界面，用于管理和监控。其中监控内容包括 Spring Boot 的监控组件 Actuator 的各个 Http 节点，也支持更高级的功能，包括 Turbine、Jmx、Loglevel 等。

本章以案例的形式来讲解 Spring Boot Admin，主要包括以下的内容。

- 使用 Spring Boot Admin 监控 Spring Cloud 微服务。
- Spring Boot Admin 集成 Turbine，聚合监控微服务系统中熔断器的状况。
- Spring Boot Admin 集成 Security 安全登录界面。

12.1 使用 Spring Boot Admin 监控 Spring Cloud 微服务

本案例需要使用 3 个工程，分别为服务注册中心 Eureka Server、服务客户端 Eureka Client 和 Spring Boot Admin Server。本案例是一个 Maven 多 Module 的工程，需要创建一个主 Maven 工程，主 Maven 工程指定了 Spring Boot 版本为 1.5.3，Spring Cloud 版本为 Dalston.RELEASE。主 Maven 工程的创建过程和服务注册中心 Eureka Server 的创建过程同 5.2 节，这里不再重复，其中 Eureka Server 的端口号为 8761。

12.1.1　构建 Admin Server

在主 Maven 工程下创建一个 Module 工程，取名为 admin-server。程作为 Spring Boot Admin Server 工程，用于对微服务系统进行监控和管理。首先，在 admin-server 工程的 pom 文件引入相关的依赖，包括继承了主 Maven 的 pom 文件、Spring Boot Admin Server 功能的两个依赖 spring-boot-admin-server 和 spring-boot-admin-server-ui、ureka Client 的起步依赖 spring-cloud-starter-eureka、Actuator 的起步依赖 spring-boot-starter-actuator。最后，在管理界面中需要与 JMX-Beans 进行交互，在 pom 文件中引入 Jolokia 的依赖。pom 件的依赖代码如下；

```
<parent>
      <groupId>com.forezp</groupId>
```

```
        <artifactId>chapter12</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <dependencies>
        <dependency>
            <groupId>de.codecentric</groupId>
            <artifactId>spring-boot-admin-server-ui</artifactId>
            <version>1.5.1</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <dependency>
            <groupId>org.jolokia</groupId>
            <artifactId>jolokia-core</artifactId>
        </dependency>
    </dependencies>
```

在 admin-server 工程的配置文件 application.yml 中置服务注册的地址为 http://localhost:8761/eureka/，服务的端口号为 5000。由于 Spring Boot 在 1.5 版本之后，Actuator 的所有 API 接口默认开启了安全验证。为了讲解方便，在本案例中关闭安全验证，即将 management. Security.enabled 改为 false。日志的输出路径为“logs/boot-admin-sample.log”。 Spring Boot Admin 默认开启 env、metrics、dump、jolokia 和 info 等节点。配置代码如下，更多配置可以查阅官方文档 http://codecentric.github.io/spring-boot-admin/1.5.1/。

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
management:
  security:
    enabled: false
server:
  port: 5000
logging:
  file: "logs/boot-admin-sample.log"

spring:
  application:
    name: service-admin
```

```
  boot:
    admin:
      routes:
        endpoints: env,metrics,dump,jolokia,info,configprops,trace,logfile,refresh,fl
yway,liquibase,heapdump,loggers,auditevents,hystrix.stream
```

Spring Boot Admin 支持对日志的管理，也支持 Logback。在 1.5x 版本的 Spring Boot 中，默认的日志为 Logback，所以在工程的 pom 文件中不需要引入 Logback 的依赖，但需要配置 Logback 的 JMXConfigurator。在 Resources 目录下建一个 logback-spring.xml 文件，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/base.xml"/>
    <jmxConfigurator/>
</configuration>
```

在程序的启动类 AdminServerApplication 加上@EnableAdminServer 注解，开启 Admin Server 的功能，加上@EnableEurekaClient 注解，开启 Eureka Client 功能，代码清单如下：

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

这样 Spring Boot Admin Server 工程就创建完成了。

12.1.2 构建 Admin Client

同 admin-server 工程一样，在主 Maven 工程下新建一个 Module 工程，取名为 eureka-client-one。eureka-client-one 工程的 pom 文件继承了主 Maven 的 pom 文件，并在 eureka-client-one 工程的 pom 文件中引入了 Web 功能的起步依赖 spring-boot-starter-web、Eureka 的起步依赖 spring-cloud-starter-eureka、Actuator 的起步依赖 spring-boot-starter-actuator，以及 Jolokia 的依赖 jolokia-core。代码如下：

```
    <parent>
          <groupId>com.forezp</groupId>
          <artifactId>chapter12</artifactId>
          <version>1.0-SNAPSHOT</version>
     </parent>
     <dependencies>
```

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <dependency>
            <groupId>org.jolokia</groupId>
            <artifactId>jolokia-core</artifactId>
        </dependency>
    </dependencies>
```

在工程的配置文件 application.yml 中指定服务注册的地址为 http://localhost:8761/eureka/，程序名为 eureka-client-one，端口号为 8762，日志的输出路径为“logs/eureka-client-one.log”，并关闭 Actuator 模块的安全验证（即配置 management. security. enabled=false）。代码如下：

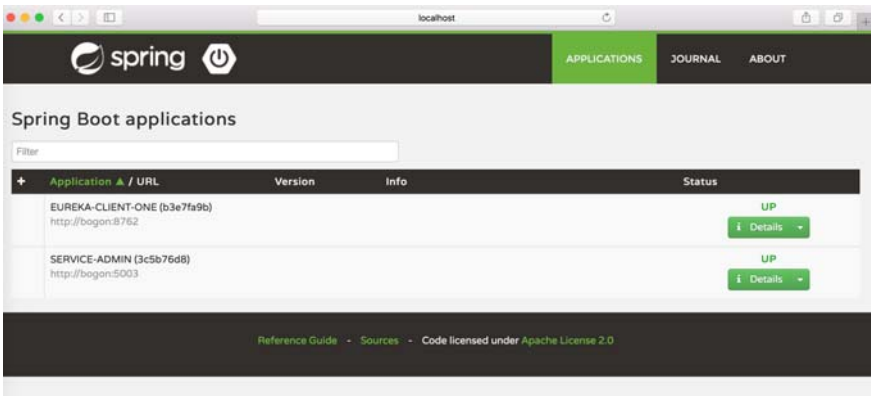
```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
server:
  port: 8762
spring:
  application:
    name: eureka-client-one
logging:
  file: "logs/eureka-client-one.log"
management:
  security:
    enabled: false
```

在程序的启动类 EurekaClientOneApplication 加上@EnableEurekaClient 注解，开启 EurekaClient 的功能。

```
@SpringBootApplication
@EnableEurekaClient
public class EurekaClientOneApplication {

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

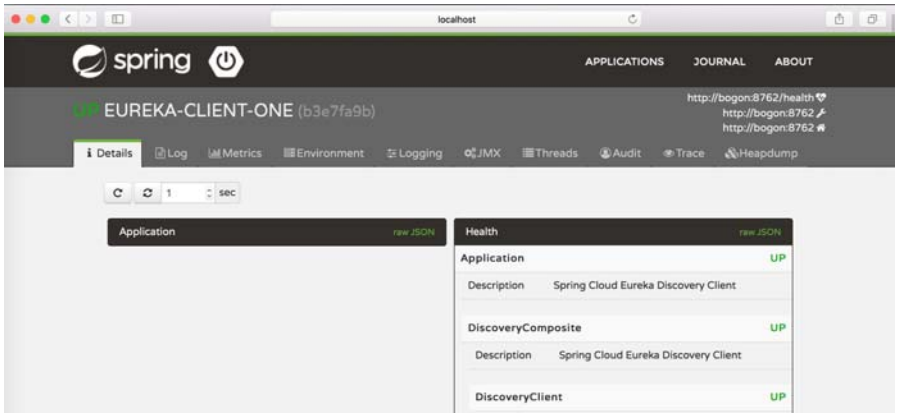
```
        }
    }
```

依次启动 eureka-server、eureka-client-one 和 admin-server，在浏览器上访问 admin-server 的主页 http://localhost:5000/，浏览器显示的 admin-server 的界面如图 12-1 所示。

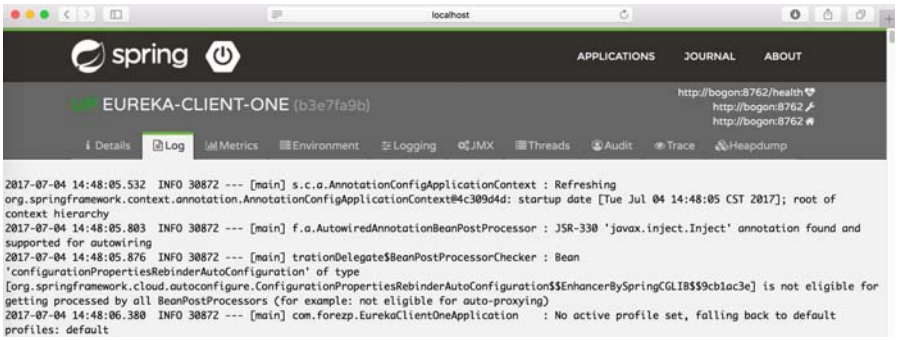


▲图 12-1　Spring Boot Admin Server 的主页

在图 12-1 的右上角，“APPLICATIONS”选项展示了向 Eureka Server 注册的所有客户端实例，如本例的 eureka-client-one 和 service-admin。“JOURNAL”选项为服务注册、下线、剔除的时间线。“ABOUT”选项是关于 Spring Boot Admin 的一些介绍。其中，在“APPLICATIONS”选项的界面展示的客户端实例右侧有一个“Details”按钮，单击该按钮，可以进入客户端实例的详细界面。在详细界面可以查看客户端实例的信息、日志信息、指标信息、环境信息、日志级别管理和 JMX 等。客户端实例的详细界面如图 12-2 所示，客户端实例的日志界面如图 12-3 所示。



▲图 12-2　客户端实例的详细界面



▲图 12-3　客户端实例的日志界面

12.2 在 Spring Boot Admin 中集成 Turbine

Hystrix Dashboard 是一个监控熔断器状况的组件，而 Turbine 是一个可以聚合多个 Hystrix Dashboard 的组件。在 Spring Boot Admin 中，可以很方便地集成 Turbine 组件。集成 Turbine 组件非常简单，只需要引入相关的依赖并做简单的配置即可。

本节案例在上一节案例的基础上进行改造，首先需要两个 Eureka Client 工程，在这两个 Eureka Client 的工程中实现 Hystrix 熔断器和 Hystrix Dashboard 组件。然后，需要一个 Turbine 工程，在这个工程中聚合两个 Eureka Client 的 Hystrix DashBoard。最后，在 Spring Boot Admin Server 中集成 Turbine 组件，这样就可以将 Turbine 界面显示的信息整合在 Spring Boot Admin Server 中，方便统一查看和管理。

12.2.1　改造 Eureka Client

eureka-client-one 工程需要集成 Hystrix 和 Hystrix DashBoard 组件。在 eureka-client-one 工程的 pom 文件引入 Hystrix 的起步依赖 spring-cloud-starter-hystrix 和 Hystrix Dashboard 的起步依赖 spring-cloud-starter-hystrix-dashboard，代码如下：

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

在程序启动类 EurekaClientOneApplication 加上@EnableHystrix 注解，开启 Hystrix 的功能，加上@EnableHystrixDashboard 注解，开启熔断器监控 Hystrix Dashboard 的功能，代码清单如下：

```
@SpringBootApplication
@EnableEurekaClient
@EnableHystrix
@EnableHystrixDashboard
public class EurekaClientOneApplication {

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

写一个 API 接口“/hi”，在方法上加上@HystrixCommand 注解，该注解用于创建一个熔断器，并指明 fallbackMethod（回退方法）为“hiError”方法。在 hiError()方法中，直接返回一个字符串，代码如下：

```
@RestController
public class HiController {
   @Value("${server.port}")
   String port;
   @GetMapping("/hi")
   @HystrixCommand(fallbackMethod = "hiError")
   public String home(@RequestParam String name) {
      return "hi "+name+",i  am lucy and from port:" +port;
   }

   public String hiError(String name) {
      return "hi,"+name+",sorry,error!";
   }
}
```

这样 eukre-client-one 就被改造成了一个具有熔断器功能的 Eureka Client，并且在 API 接口“/hi”使用到了熔断器。

12.2.2 另行构建 Eureka Client

在主 Maven 目录下创建另外一个 Eureka Client 的 Module 工程，该工程取名为 eureka-client-two。它的依赖同 eureka-client-one，并且同样也对外也暴露一个 API 接口“/hi”，在这个 API 接口中也使用到了熔断器。eureka-client-two 工程与 eureka-client-one 工程的代码类似，与 eureka-client-one 工程不同的是，需要在 eureka-client-two 工程的配置文件 application.yml 中指明程序的名称为 eureka-client-two，程序的端口号为 8763，日志输出路径为“logs/eureka-client-two.log”。eureka-client-two 工程的配置文件 application.yml 代码清单如下：

```
eureka:
  client:
    serviceUrl:
```

```
      defaultZone: http://localhost:8761/eureka/
server:
  port: 8763
spring:
  application:
    name: eureka-client-two

logging:
  file: "logs/eureka-client-two.log"

management:
  security:
    enabled: false
```

12.2.3　构建 Turbine 工程

在主 maven 工程下创建一个 Module 工程，取名为 turbine-service。turbine-service 工程提供了 Turbine 组件的功能，聚合了 eureka-client-one 和 eureka-client-two 工程的 Hystrix Dashboard 组件。turbine-service 工程的 pom 文件继承主 Maven 工程的 pom 文件，引入了 Turbine 功能依赖，包括 spring-cloud-starter-turbine 和 spring-cloud-netflix-turbine，并引入了 Actuator 的起步依赖 spring-boot-starter-actuator 和 Jolokia 的依赖 jolokia-core。turbine-service 工程的 pom 文件代码清单如下：

```
<parent>
    <groupId>com.forezp</groupId>
    <artifactId>chapter12</artifactId>
   <version>1.0-SNAPSHOT</version>
</parent>
<dependencies>
     <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-turbine</artifactId>
     </dependency>
     <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-netflix-turbine</artifactId>
     </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
   <dependency>
      <groupId>org.jolokia</groupId>
      <artifactId>jolokia-core</artifactId>
   </dependency>
```

```
</dependencies>
```

在 turbine-service 工程的配置文件 application.yml 中做程序的相关的配置，包括配置程序的名称为 service-turbine，端口号为 8769，服务注册地址为 http://localhost:8761/eureka/。该工程聚合监控了 eureka-client-one 和 eureka-client-two 工程的 Hystrix Dashboard，集群配置采用默认。配置文件 application.yml 的代码清单如下：

```
spring:
  application.name: service-turbine
server:
  port: 8769
turbine:
  aggregator:
   clusterConfig: default
 appConfig: eureka-client-one,eureka-client-two
 clusterNameExpression: new String("default")

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

management:
  security:
    enabled: false
```

在程序的入口类 TurbineServiceApplication 加上@EnableTurbine 注解，开启 Turbine 的功能，代码如下：

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

12.2.4 在 Admin Server 中集成 Turbine

最后需要在 Spring Boot Admin Server 中集成 Turbine。在 admin-server 工程的 pom 文件引入相关依赖，需要引入以下 4 个依赖，代码清单如下：

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-server-ui-turbine</artifactId>
    <version>1.5.1</version>
```

```
</dependency>
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-server-ui-hystrix</artifactId>
    <version>1.5.1</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-turbine</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

在 admin-server 工程的配置文件 application.yml 中配置 Turbine，通过 spring.boot.admin.turbine.clusters 来配置集群的情况，本案例的集群情况使用默认的即可。通过 spring.boot.admin.turbine.location 来配置 Turbine 的服务名，本案例 Turbine 的服务名为 service-turbine，代码清单如下：

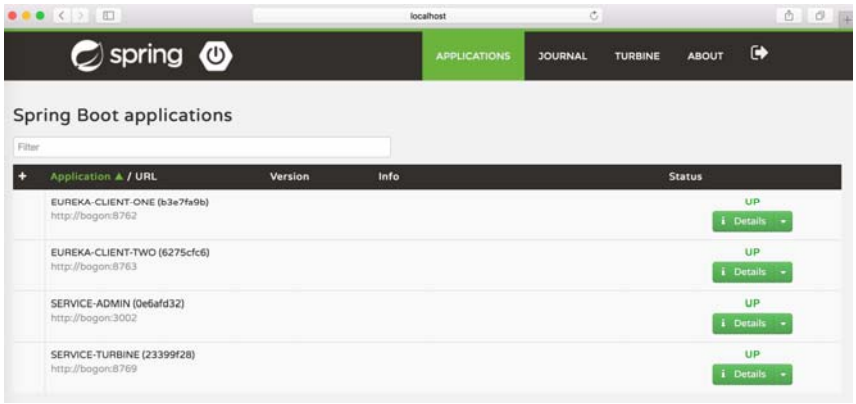
```
spring:
  application:
    name: service-admin
  boot:
    admin:
      routes:
        endpoints: env,metrics,dump,jolokia,info,configprops,trace,logfile,refresh,fl
yway,liquibase,heapdump,loggers,auditevents,hystrix.stream,activiti
      turbine:
        clusters: default
        location: service-turbine
```

在程序的启动类 AdminServerApplication 加上@EnableTurbine 注解开启 Turbine 功能，加上@EnableHystrixDashboard 注解开启 Hystrix Dashboard 的功能，加上@EnableHystrix 注解开启 Hystrix 功能。代码清单如下：

```
@SpringBootApplication
@EnableTurbine
@EnableHystrixDashboard
@EnableHystrix
@EnableAdminServer
public class AdminServerApplication {
```

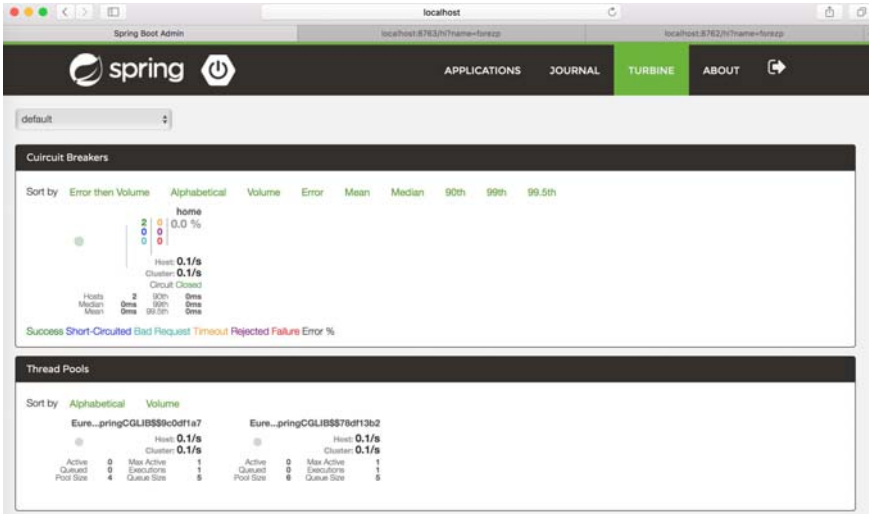
```
    public static void main(String[] args) {
        SpringApplication.run(AdminServerApplication.class, args);
    }
}
```

依次启动工程 eureka-server、eureka-client-one、eureka-client-two、turbine-service 和 admin-server 这 5 个工程，在浏览器上访问 admin-server 的主页 http://localhost:5000/，浏览器显示如图 12-4 所示。这个界面与 12.1 节中案例的界面相比多了一个“TURBINE”的选项按钮。



▲图 12-4　admin-server 集成 Turbine 后的界面

依次请求 http://localhost:8762/hi?name=forezp 和 http://localhost:8763/hi?name=forezp，这两个 API 接口带有熔断器的功能。请求完毕大约 1 分钟后，admin-server 的主页的 Turbine 页面显示如图 12-5 所示。注意：如果浏览器的“TURBINE”选项按钮没有显示出来，可以尝试清空浏览器缓存再访问。



▲图 12-5　admin-server 的 Turbine 界面

12.3 在 Spring Boot Admin 中添加安全登录界面

Spring Boot Admin 提供了非常良好的组件化界面，例如在上一节就很方便地集成了 Turbine 的界面。在生产环境中，不希望通过网址直接访问 Spring Boot Admin Server 的主页界面，因为这个界面包含了太多的服务信息，必须对这个界面的访问进行安全验证。Spring Boot Admin 提供了登录界面的组件，并且和 Spring Boot Security 相结合，需要用户登录才能访问 Spring Boot Admin Server 的界面。

下面来进行具体的案例讲解，本节的案例在上一节的案例基础之上进行改造。在 admin-server 工程的 pom 文件加上相关的依赖，包括登录界面组件的依赖 spring-boot-admin-server-ui-login，以及 Spring Boot Security 的起步依赖 spring-boot-starter-security，代码清单如下：

```
<dependency>
        <groupId>de.codecentric</groupId>
        <artifactId>spring-boot-admin-server-ui-login</artifactId>
        <version>1.5.0</version>
</dependency>
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

在工程的配置文件 application.yml 中做以下的配置，创建一个 security 的 user 用户，它的用户名为 admin，密码为 123456。通过 eureka.instance.metadate-map 配置带上该 security 的 user 用户的信息，因为在工程的 pom 文件加上 Spring Boot Security 的起步依赖以后，该项目的所有资源（包括静态资源 html、css，以及 API 接口等）都是受保护的，需要加上该管理员信息进行安全验证才能访问，代码如下：

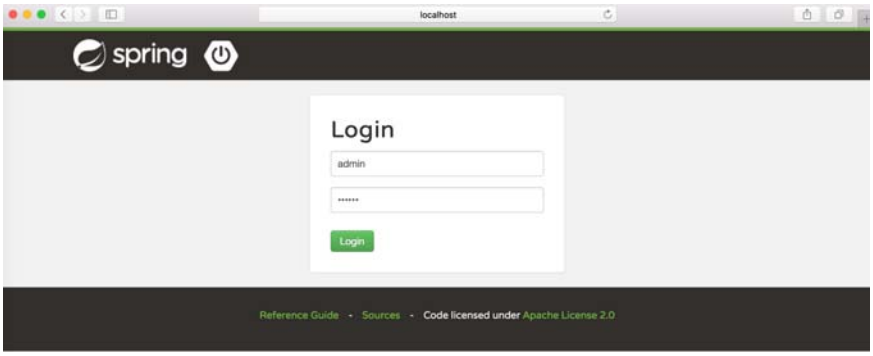
```
security:
  user:
tFoundException
    password: 123456

eureka:
  instance:
    metadata-map:
      user.name: admin
      user.password: 123456
```

然后，在程序中配置 Spring Boot Security，写 SecurityConfig 的配置类，该配置类继承 WebSecurityConfigurerAdapter，并复写了 configure(HttpSecurity http)方法。在该方法下做了各种配置，例如配置了登录界面为“/login.html”，登出界面为“/logout”。这些页面以及静态资

源 css、img 都在引入的 Jar 中，这些资源的访问不需要认证。给这些静态资源加上 permitAll()方法，除上述以外的资源访问需要权限认证，即加上 authenticated()方法。另外这些静态资源界面不支持 CSFR（跨站请求伪造），所以禁用掉 CSFR，最后需要开启 Http 的基本认证，即 httpBasic()方法。代码清单如下：

```
@Configuration
public static class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
       http.formLogin().loginPage("/login.html").
       loginProcessingUrl("/login").permitAll();
       http.logout().logoutUrl("/logout");
       http.csrf().disable();
       http.authorizeRequests()
       .antMatchers("/login.html", "/**/**.css", "/img/**", "/third-party/**")
     .permitAll();
    http.authorizeRequests().antMatchers("/**").authenticated();
        http.httpBasic();
    }
}
```

重新启动 admin-server 工程，在浏览器中访问 http://localhost:5000/，浏览器显示的界面如图 12-6 所示。在界面上输入用户名为 amidn，密码为 123456，单击“Login”按钮，登录成功就会跳转到 Spring Boot Admin 的主页。



▲图 12-6　登录界面

第 13 章　Spring Boot Security 详解

13.1 Spring Security 简介

13.1.1　什么是 Spring Security

Spring Security 是 Spring Resource 社区的一个安全组件，Spring Security 为 JavaEE 企业级开发提供了全面的安全防护。安全防护是一个不断变化的目标，Spring Security 通过版本不断迭代来实现这一目标。Spring Security 采用“安全层”的概念，使每一层都尽可能安全，连续的安全层可以达到全面的防护。Spring Security 可以在 Controller 层、Service 层、DAO 层等以加注解的方式来保护应用程序的安全。Spring Security 提供了细粒度的权限控制，可以精细到每一个 API 接口、每一个业务的方法，或者每一个操作数据库的 DAO 层的方法。Spring Security 提供的是应用程序层的安全解决方案，一个系统的安全还需要考虑传输层和系统层的安全，例如采用 Htpps 协议、服务器部署防火墙等。

13.1.2　为什么选择 Spring Security

使用 Spring Security 有很多原因，其中一个重要原因是它对环境的无依赖性、低代码耦合性。将工程重现部署到一个新的服务器上，不需要为 Spring Security 做什么工作。Spring Security 提供了数十个安全模块，模块与模块间的耦合性低，模块之间可以自由组合来实现特定需求的安全功能，具有较高的可定制性。总而言之，Spring Security 具有很好的可复用性和可定制性。

在安全方面，有两个主要的领域，一是“认证”，即你是谁；二是“授权”，即你拥有什么权限，Spring Security 的主要目标就是在这两个领域。“认证”是认证主体的过程，通常是指可以在应用程序中执行操作的用户、设备或其他系统。“授权”是指决定是否允许已认证的主体执行某一项操作。

安全框架多种多样，那为什么选择 Spring Security 作为微服务开发的安全框架呢？JavaEE 有另一个优秀的安全框架 Apache Shiro，Apache Shiro 在企业级的项目开发中十分受欢迎，一般使用在单体服务中。但在微服务架构中，目前版本的 Apache Shiro 是无能为力的。Spring Security 来自 Spring Resource 社区，采用了注解的方式控制权限，熟悉 Spring 的开发

者很容易上手 Spring Security。另外一个原因就是 Spring Security 易于应用于 Spring Boot 工程，也易于集成到采用 Spring Cloud 构建的微服务系统中。

13.1.3 Spring Security 提供的安全模块

在安全验证方面， Spring Security 提供了很多的安全验证模块。大部分的验证模块来自第三方的权威机构或者一些相关的标准制定组织，Spring Security 自身也提供了一些验证模型。Spring Security 目前支持对以下技术的整合。（注：这部分内容来自于官方文档）

- HTTP BASIC 头认证（一个基于 IETF RFC 的标准）。
- HTTP Digest 头认证（一个基于 IETF RFC 的标准）。
- HTTP X.509 客户端证书交换认证（一个基于 IETF RFC 的标准）。
- LDAP（一种通用的跨平台身份验证，特别是在大型软件架构中）。
- 基于表单的验证。
- OpenID 验证。
- 基于预先建立的请求头的验证。
- Jasig Central Authentication Service，也被称作 CAS，是一个流行的开源单点登录系统。
- 远程方法调用（RMI）和 HttpInvoker（Spring 远程协议）的认证。
- 自动“记住我”的身份验证。
- 匿名验证（允许每一次未经身份验证的调用）。
- Run-as 身份验证（每一次调用都需要提供身份标识）。
- Java 认证和授权服务。
- Java EE 容器认证。
- Kerberos。
- Java 开源的单点登录*。
- OpenNMS 网络管理平台*。
- AppFuse *。
- AndroMDA *。
- Mule ESB *。
- Direct Web Request （DWR）*。
- Grails *。
- Tapestry *。
- JTrac *。
- Jasypt *。
- Roller *。
- Elastic Path *。
- Atlassian Crowd*。
- 自己创建的认证系统。

以上都是 Spring Security 支持的安全验证模块，其中带*的是来自第三方的安全验证模块，Spring Security 对这些模块做了整合和封装。

13.2 Spring Boot Security 与 Spring Security 的关系

在 Spring Security 框架中，主要包含了两个依赖 Jar，分别是 spring-security-web 依赖和 spring-security-config 依赖，代码如下：

```
<dependencies>
<dependency>
     <groupId>org.springframework.security</groupId>
     <artifactId>spring-security-web</artifactId>
     <version>4.2.2.RELEASE</version>
</dependency>
<dependency>
     <groupId>org.springframework.security</groupId>
     <artifactId>spring-security-config</artifactId>
     <version>4.2.2.RELEASE</version>
</dependency>
</dependencies>
```

Spring Boot 对 Spring Security 框架做了封装，仅仅是封装，并没有改动 Spring Security 这两个包的内容，并加上了 Spring Boot 的起步依赖的特性。spring-boot-starter-security 依赖如下：

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

进入 spring-boot-starter-security 的 pom 文件，可以发现 pom 文件包含了 Spring Security 的两个 Jar 包，并移除了这两个 Jar 包的 apo 功能，引入了 aop 的依赖，另外包含了 spring-boot-starter 的依赖。由此可见，spring-boot-starter-security 是对 Spring Security 的一个封装。

13.3 Spring Boot Security 案例详解

13.3.1　构建 Spring Boot Security 工程

使用 IDEA 的 Sping Initializr 方式建一个 Spring Boot 工程。创建完成后，在工程的 pom 文件中入相关依赖，包括版本为 1.5.3 的 Spring Boot 的起步依赖、Security 的起步依赖 spring-boot-starter-security、Web 模版引擎 Thymeleaf 的起步依赖 spring-boot-starter-thymeleaf、Web 功能的起步依赖 spring-boot-starter-web、thymeleaf 和 security 的依赖 thymeleaf-extras-springsecurity4。完整的 pom 依赖如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns=http://maven.apache.org/POM/4.0.0 xmlns:xsi="http://www.w3.org/2001/XM
LSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xs
d/maven-4.0.0.xsd">
     <modelVersion>4.0.0</modelVersion>
     <groupId>com.forezp</groupId>
     <artifactId>springboot-security</artifactId>
     <version>0.0.1-SNAPSHOT</version>
     <packaging>jar</packaging>
     <name>springboot-security</name>
     <description>Demo project for Spring Boot</description>
     <parent>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-parent</artifactId>
          <version>1.5.3.RELEASE</version>
          <relativePath/> <!-- lookup parent from repository -->
     </parent>

     <properties>
          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
          <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
          <java.version>1.8</java.version>
     </properties>
     <dependencies>
          <dependency>
               <groupId>org.springframework.boot</groupId>
               <artifactId>spring-boot-starter-security</artifactId>
          </dependency>
          <dependency>
               <groupId>org.springframework.boot</groupId>
               <artifactId>spring-boot-starter-thymeleaf</artifactId>
          </dependency>
          <dependency>
               <groupId>org.springframework.boot</groupId>
               <artifactId>spring-boot-starter-web</artifactId>
          </dependency>
          <dependency>
               <groupId>org.thymeleaf.extras</groupId>
               <artifactId>thymeleaf-extras-springsecurity4</artifactId>
          </dependency>
          <dependency>
               <groupId>org.springframework.boot</groupId>
               <artifactId>spring-boot-starter-test</artifactId>
               <scope>test</scope>
          </dependency>
```

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

13.3.2　配置 Spring Security

1. 配置 WebSecurityConfigurerAdapter

创建完 Spring Boot 工程并引入工程所需的依赖后，需要配置 Spring Security。新建一个 SecurityConfig 类，作为配置类，它继承了 WebSecurityConfigurerAdapter 类。在 SecurityConfig 类上加@EnableWebSecurity 注解，开启 WebSecurity 的功能，并需要注入 AuthenticationManagerBuilder 类的 Bean。代码如下：

```
@EnableWebSecurity
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
   @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws      Exception {
    auth.inMemoryAuthentication()
     .withUser("forezp").password("123456").roles("USER");
}
```

上述代码做了 Spring Security 的基本配置，并通过 AuthenticationManagerBuilder 在内存中创建了一个认证用户的信息，该认证用户名为 forezp，密码为 123456，有 USER 的角色。上述的代码内容虽少，但做了很多安全防护的工作，包括如下内容。

（1）应用的每一个请求都需要认证。

（2）自动生成了一个登录表单。

（3）可以用 username 和 password 来进行认证。

（4）用户可以注销。

（5）阻止了 CSRF 攻击。

（6）Session Fixation 保护。

（7）安全 Header 集成了以下内容。

- HTTP Strict Transport Security for secure requests
- X-Content-Type-Options integration
- Cache Control

- X-XSS-Protection integration
- XFrame-Options integration to help prevent Clickjacking

（8）集成了以下的 Servlet API 的方法。

- HttpServletRequest#getRemoteUser()
- HttpServletRequest.html#getUserPrincipal()
- HttpServletRequest.html#isUserInRole(java.lang.String)
- HttpServletRequest.html#login(java.lang.String, java.lang.String)
- HttpServletRequest.html#logout()

2. 配置 HttpSecurity

WebSecurityConfigurerAdapter 配置了如何验证用户信息。那么 Spring Security 如何知道是否所有的用户都需要身份验证呢？又如何知道要支持基于表单的身份验证呢？工程的哪些资源需要验证，哪些资源不需要验证？这时就需要配置 HttpSecurity。

新建一个 SecurityConfig 类继承 WebSecurityConfigurerAdapter 类作为 HttpSecurity 的配置类，通过复写 configure（HttpSecurity http）方法来配置 HttpSecurity。本案例的配置代码如下：

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
     @Override
     protected void configure(HttpSecurity http) throws Exception {
          http
               .authorizeRequests()
               .antMatchers("/css/**", "/index").permitAll()
               .antMatchers("/user/**").hasRole("USER")
               .antMatchers("/blogs/**").hasRole("USER")
               .and()
                .formLogin().loginPage("/login").failureUrl("/login-error")
               .and()
               .exceptionHandling().accessDeniedPage("/401");
                http.logout().logoutSuccessUrl("/");
     }
...
}
```

在上述代码中，配置了如下内容。

- 以“/css/**”开头的资源和“/index”资源不需要验证，外界请求可以直接访问这些资源。
- 以“/user/**”和“/blogs/**”开头的资源需要验证，并且需要用户的角色是“Role”。
- 表单登录的地址是“/login”，登录失败的地址是“/login-error”。
- 异常处理会重定向到“/401”界面。

❑ 注销登录成功，重定向到首页。

在上述的配置代码中配置了相关的界面，例如首页、登录页、用户首页等。配置这些界面在 Controller 层的代码如下：

```
@Controller
public class MainController {
     @RequestMapping("/")
     public String root() {
          return "redirect:/index";
     }
     @RequestMapping("/index")
     public String index() {
          return "index";
     }
     @RequestMapping("/user/index")
     public String userIndex() {
          return "user/index";
     }
     @RequestMapping("/login")
     public String login() {
          return "login";
     }
     @RequestMapping("/login-error")
     public String loginError(Model model) {
          model.addAttribute("loginError", true);
          return "login";
     }
     @GetMapping("/401")
     public String accesssDenied() {
          return "401";
     }
}
```

13.3.3　编写相关界面

在上一节中配置了相关的界面，因为界面只是为了演示 Spring Boot Security 的案例，并不是本章的重点，所以界面做得非常简单。

在工程的配置文件 application.yml 中配置 thymeleaf 引擎，模式为 HTML5，编码为 UTF-8，开启热部署。配置代码如下：

```
spring:
  thymeleaf:
    mode: HTML5
    encoding: UTF-8
    cache: false
```

登录界面（login/html）的代码如下：

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
   <head>
       <title>Login page</title>
       <meta charset="utf-8" />
       <link rel="stylesheet" href="/css/main.css" th:href="@{/css/main.css}" />
    </head>
   <body>
       <h1>Login page</h1>
       <p>User 角色用户: forezp / 123456</p>
       <p>Admin 角色用户: admin / 123456</p>
       <p th:if="${loginError}" class="error">用户名或密码错误</p>
       <form th:action="@{/login}" method="post">
           <label for="username">用户名</label>:
           <input type="text" id="username" name="username" autofocus="autofocus" />
<br />
           <label for="password">密码</label>:
           <input type="password" id="password" name="password" /> <br />
           <input type="submit" value="登录" />
       </form>
       <p><a href="/index" th:href="@{/index}">返回首页</a></p>
   </body>
</html>
```

首页（index.html）的代码如下：

```
<!DOCTYPE html><html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymel
eaf.org" xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity4">
   <head>
      <title>Hello Spring Security</title>
      <meta charset="utf-8" />
      <link rel="stylesheet" href="/css/main.css" th:href="@{/css/main.css}" />
   </head>
   <body>
      <h1>Hello Spring Security</h1>
      <p>这个界面没有受保护，你可以进已被保护的界面.</p>
      <div th:fragment="logout"  sec:authorize="isAuthenticated()">
         登录用户: <span sec:authentication="name"></span> |
         用户角色: <span sec:authentication="principal.authorities"></span>
         <div>
            <form action="#" th:action="@{/logout}" method="post">
               <input type="submit" value="登出" />
            </form>
         </div>
     </div>
```

```
      <ul>
         <li>点击<a href="/user/index" th:href="@{/user/index}">去/user/index 已被保护的界
面</a></li>
      </ul>
   </body>
</html>
```

权限不够显示的界面（401.html）代码如下：

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity4">

<body>
     <div >
     <div >
          <h2> 权限不够</h2>
     </div>
          <div sec:authorize="isAuthenticated()">
               <p>已有用户登录</p>
               <p>用户: <span sec:authentication="name"></span></p>
                <p>角色: <span sec:authentication="principal.authorities"></span></p>
          </div>
          <div sec:authorize="isAnonymous()">
               <p>未有用户登录</p>
          </div>
          <p>
               拒绝访问!
          </p>
     </div>
</body>
</html>
```

用户首页（/user/index.html）界面，该资源被 Spring Security 保护，只有拥有“USER”角色的用户才能够访问，其代码如下：

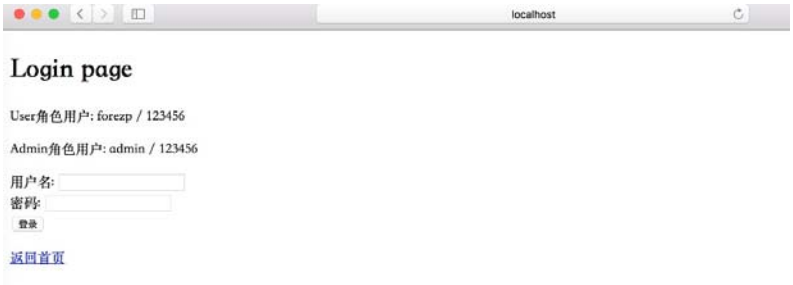
```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
   <head>
      <title>Hello Spring Security</title>
      <meta charset="utf-8" />
      <link rel="stylesheet" href="/css/main.css" th:href="@{/css/main.css}" />
   </head>
   <body>
      <div th:substituteby="index::logout"></div>
      <h1>这个界面是被保护的界面</h1>
      <p><a href="/index" th:href="@{/index}">返回首页</a></p>
```

```
        <p><a  href="/blogs" th:href="@{/blogs}">管理博客</a></p>
    </body>
</html>
```

启动工程，在浏览器上访问 localhost:8080，会被重定向到 localhost:8080/index 界面，如图 13-1 所示。

▲图 13-1　localhost:8080/index 界面

单击上面界面的“去/user/index 保护的界面”文字，由于“/user/index”界面需要“USER”权限，但还没有登录，会被重定向到登录界面“/login.html”，登录界面如图 13-2 所示。



▲图 13-2　登录界面

这时，用具有“USER”角色的用户登录，即用户名为 forezp，密码为 123456。登录成功，界面会被重定向到 http://localhost:8080/user/index 界面，注意该界面是具有“USER”角色的用户才具有访问权限。界面显示如图 13-3 所示。

▲图 13-3　界面 http://localhost:8080/user/index

为了演示“/user/index”界面只有“USER”角色才能访问，新建一个 admin 用户，该用户只有“ADMIN”的角色，没有“USER”角色，所以没有权限访问“/user/index”界面。修改 SecurityConfig 配置类，在这个类新增一个用户 admin，代码如下：

```
@EnableWebSecurity
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
 .....
   @Autowired
   public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
          auth.userDetailsService(userDetailsService());
     }
  @Bean
  public UserDetailsService userDetailsService() {
   InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();
   // 在内存中存放用户信息
     manager.createUser(User.withUsername("forezp").password("123456").roles("USER").
build());
     manager.createUser(User.withUsername("admin").password("123456").roles("ADMIN").
build());
     return manager;
 }
}
```

InMemoryUserDetailsManager 类是将用户信息存放在程序的内存中的。程序启动后，InMemoryUserDetailsManager 会在内存中创建用户的信息。在上述的案例中创建两个用户，forezp 用户具有“USER”角色，admin 用户具有“ADMIN”角色。用 admin 用户去登录，并访问 http://localhost:8080/user/index，这时会被重定向到权限不足的界面，显示的界面如图 13-4 所示。

▲图 13-4　“ADMIN”角色没有权限访问/user/index

这时给 admin 用户加上“USER”角色，修改 SecurityConfig 配置类的代码，具体代码如下：

```
manager.createUser(User.withUsername("admin").password("123456").roles("ADMIN","USER"
).build());
```

再次用 admin 用户访问 http://localhost:8080/user/index 界面，界面可以正常显示。可见 Spring Security 对“/user/index”资源进行了保护，并且只允许具有“USER”角色权限的用户访问。

13.3.4 Spring Security 方法级别上的保护

Spring Security 从 2.0 版本开始，提供了方法级别的安全支持，并提供了 JSR-250 的支持。写一个配置类 SecurityConfig 继承 WebSecurityConfigurerAdapter，并加上相关注解，就可以开启方法级别的保护。代码如下：

```
@EnableWebSecurity
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
}
```

在上面的配置代码中，@ EnableGlobalMethodSecurity 注解开启了方法级别的保护，括号后面的参数可选，可选的参数如下。

- prePostEnabled：Spring Security 的 Pre 和 Post 注解是否可用，即@PreAuthorize 和 @PostAuthorize 是否可用。
- secureEnabled：Spring Security 的 @Secured 注解是否可用。
- jsr250Enabled：Spring Security 对 JSR-250 的注解是否可用。

一般来说，只会用到 prePostEnabled。因为@PreAuthorize 注解和@PostAuthorize 注解更适合方法级别的安全控制，并且支持 Spring EL 表达式，适合 Spring 开发者。其中，@PreAuthorize 注解会在进入方法前进行权限验证，@PostAuthorize 注解在方法执行后再进行权限验证，后一个注解的应用场景很少。

如何在方法上写权限注解呢？例如有权限点字符串“ROLE_ADMIN”，在方法上可以写为@PreAuthorize("hasRole('ADMIN')")，也可以写为@PreAuthorize("hasAuthority('ROLE_ADMIN')")，这二者是等价的。加多个权限点，可以写为@PreAuthorize("hasAnyRole('ADMIN','USER')")，也可以写为@PreAuthorize("hasAnyAuthority('ROLE_ADMIN','ROLE_USER')")。

为了演示方法级别的安全保护，需要写一个 API 接口，在该接口加上权限注解。在本案例中，有一个 Blog（博客）文章列表的 API 接口，只有管理员权限的用户才能删除 Blog，现在来实现该 API 接口。首先，需要创建 Blog 实体类，代码如下：

```
public class Blog {
   private Long id;
   private String name;
   private String content;
   public Blog(Long id, String name, String content) {
      this.id = id;
      this.name = name;
      this.content = content;
   }
```

```
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getContent() {
        return content;
    }
    public void setContent(String content) {
        this.content = content;
    }
}
```

创建 IBlogService 接口类，为了演示方便，没有 DAO 层操作数据库，而是在内存中维护一个 List<Blog>来模拟数据库操作，包括获取所有的 Blog、根据 Id 删除 Blog 的两个方法。接口类代码如下：

```
public interface IBlogService {
    List<Blog> getBlogs();
    void deleteBlog(long id);
}
```

IBlogService 的实现类 BlogService，在构造函数方法上加入了两个 Blog 对象，并实现了 IBlogService 的两个方法。具体代码如下：

```
@Service
public class BlogService implements IBlogService {
    private List<Blog> list=new ArrayList<>();
    public BlogService(){
        list.add(new Blog(1L, " spring in action", "good!"));
        list.add(new Blog(2L,"spring boot in action", "nice!"));
    }

    @Override
    public List<Blog> getBlogs() {
        return list;
    }

    @Override
    public void deleteBlog(long id) {
```

```
        Iterator iter = list.iterator();
        while(iter.hasNext()) {
            Blog blog= (Blog) iter.next();
            if (blog.getId()==id){
               iter.remove();
            }
        }
    }
}
```

在 Controller 层上写两个 API 接口，一个获取所有 Blog 的列表（“/blogs”），另一个根据 Id 删除 Blog（“/blogs/{id}/deletion”）。后一个 API 接口需要“ADMIN”的角色权限，通过注解 @PreAuthorize("has Authority ('ROLE_ADMIN')") 来实现。在调用删除 Blog 接口之前，会判断该用户是否具有“ADMIN”的角色权限。如果有权限，则可以删除；如果没有权限，则显示权限不足的界面。代码如下：

```
@RestController
@RequestMapping("/blogs")
public class BlogController {

   @Autowired
   BlogService blogService;
   @GetMapping
   public ModelAndView list(Model model) {
      List<Blog> list =blogService.getBlogs();
      model.addAttribute("blogsList", list);
      return new ModelAndView("blogs/list", "blogModel", model);
   }
   @PreAuthorize("hasAuthority('ROLE_ADMIN')")
   @GetMapping(value = "/{id}/deletion")
   public ModelAndView delete(@PathVariable("id") Long id, Model model) {
      blogService.deleteBlog(id);
      model.addAttribute("blogsList", blogService.getBlogs());
      return new ModelAndView("blogs/list", "blogModel", model);
   }
}
```

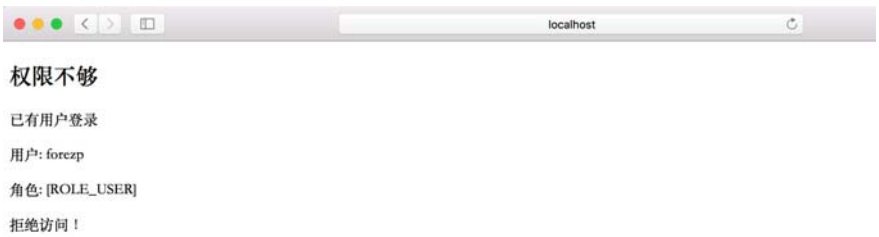
启动用户程序，使用用户名为 admin，密码为 123456 的用户登录，该用户具有“ADMIN”的角色权限。登录成功后，进入“/blogs/list”界面，该界面如图 13-5 所示。

▲图 13-5 /blogs/list 网页

单击“删除”按钮，该删除按钮调用了“/blogs/{id}/deletion”的 API 接口。单击“删除”，删除编号为 2 的博客，删除成功后的界面如图 13-6 所示。

▲图 13-6　/blogs/list 界面删除博客编号为 2 后的界面

为了验证方法级别上的安全验证的有效性，需要用一个没有“ADMIN”角色权限的用户进行删除操作。用户名为 forezp，密码为 123456 的用户只有“USER”的角色权限，没有“ADMIN”的角色权限。用该用户登录，做删除 Blog 的操作，会显示用户权限不足的界面，界面如图 13-7 所示。



▲图 13-7　删除博客权限不足

可见，在方法级别上的安全验证是通过相关的注解和配置来实现的。本例子中的注解写在 Controller 层，如果写在 Service 层也同样生效。对 Spring Security 而言，它只控制方法，不论方法在哪个层级上。

13.3.5　从数据库中读取用户的认证信息

在上述例子中，采用了从内存中配置用户信息，包括用户名、密码、用户的角色权限信息。当用户数量非常多时，这种方式显然是不可行的。本节讲述如何从数据库中读取用户和用户的角色权限信息。本案例中采用的数据库为 MySQL，ORM 框架为 JPA。

在工程的 pom 文件中加上 MySQL 数据库连接的依赖 mysql-connector-java 和 JPA 的起步依赖 spring-boot-starter-data-jpa，代码如下：

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

```
</dependency>
```

在工程的配置文件 application.yml 中配置数据库连接驱动、数据源、数据库用户名和密码，以及 JPA 的相关配置，配置代码如下：

```
spring:
  thymeleaf:
    mode: HTML5
    encoding: UTF-8
    cache: false

  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/spring-security?useUnicode=true&characterEncodin
g=utf8&characterSetResults=utf8
    username: root
    password: 123456

  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
```

创建 User 实体，该类使用了 JPA 的注解@Entity，表示该 Java 对象会被映射到数据库。id 采用的生成策略为自增加，包含了 username 和 password 两个字段，其中 authorities 为权限点的集合。具体代码如下：

```
@Entity
public class User implements UserDetails, Serializable {
     @Id
     @GeneratedValue(strategy = GenerationType.IDENTITY)
     private Long id;
     @Column(nullable = false,  unique = true)
     private String username;
     @Column
     private String password;
     @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
     @JoinTable(name = "user_role", joinColumns = @JoinColumn(name = "user_id", refer
encedColumnName = "id"),
          inverseJoinColumns = @JoinColumn(name = "role_id", referencedColumnName = "id"))
     private List<Role> authorities;
     public User() {
     }
     public Long getId() {
          return id;
     }
```

```
    public void setId(Long id) {
        this.id = id;
    }
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return authorities;
    }
    public void setAuthorities(List<Role> authorities) {
        this.authorities = authorities;
    }
    @Override
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    @Override
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    @Override
    public boolean isAccountNonExpired() {
        return true;
    }
    @Override
    public boolean isAccountNonLocked() {
        return true;
    }
    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }
    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

上面的 User 类实现了 UserDetails 接口，该接口是实现 Spring Security 认证信息的核心接口。其中，getUsername 方法为 UserDetails 接口的方法，这个方法不一定返回 username，也可以是其他的用户信息，例如手机号码、邮箱地址等。getAuthorities( )方法返回的是该用户设置的权限信息，在本例中，从数据库中读取该用户的所有角色信息，权限信息也可以是用户的其

他信息，不一定是角色信息。另外需要读取密码，最后几个方法一般情况下都返回 true，也可以根据自己的需求进行业务判断。UserDetails 接口的代码如下：

```
public interface UserDetails extends Serializable {
     Collection<? extends GrantedAuthority> getAuthorities();
     String getPassword();
     String getUsername();
     boolean isAccountNonExpired();
     boolean isAccountNonLocked();
     boolean isCredentialsNonExpired();
     boolean isEnabled();
}
```

Role 类实现了 GrantedAuthority 接口，并重写了其 getAuthority 方法。权限点可以为任何字符串，不一定是角色名的字符串，关键是 getAuthority 方法如何实现。本例的权限点是从数据库读取 Role 表的 name 字段。Role 类代码如下：

```
@Entity
public class Role implements GrantedAuthority {
     @Id
     @GeneratedValue(strategy = GenerationType.IDENTITY)
     private Long id;
     @Column(nullable = false)
     private String name;
     public Long getId() {
          return id;
     }
     public void setId(Long id) {
          this.id = id;
     }
     @Override
     public String getAuthority() {
          return name;
     }
     public void setName(String name) {
          this.name = name;
     }
     @Override
     public String toString() {
          return name;
     }
}
```

UserDao 继承了 JpaRepository，JpaRepository 默认实现了大多数单表查询的操作。UserDao 中自定义一个根据 username 获取 User 的方法，由于 JPA 已经自动实现了根据 username 字段去查找用户的方法，因此不需要额外的工作。代码如下：

```
public interface UserDao extends JpaRepository<User, Long>{
     User findByUsername(String username);
}
```

Service 层需要实现 UserDetailsService 接口，该接口是根据用户名获取该用户的所有信息，包括用户信息和权限点，代码如下：

```
@Service
public class UserService implements UserDetailsService {
    @Autowired
    private UserDao userRepository;
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundExc
eption {
        return userRepository.findByUsername(username);
    }
}
```

最后需要修改 Spring Sercurity 的配置，让 Spring Sercurity 从数据库中获取用户的认证信息，而不是之前从内存中读取，代码如下：

```
@EnableWebSecurity
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
     //代码省略
     @Autowired
     UserDetailsService userDetailsService;
     @Autowired
     public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception
{
          auth.userDetailsService(userDetailsService);
     }
}
```

在启动应用程序之前，需要在 MySQL 中建一个数据库，数据库名为 spring-security，建库语句如下：

```
CREATE DATABASE spring-security DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci
```

启动程序，JPA 会连接数据库，在数据库自动建表，也可以自己在数据库中建表，MySQL 数据库建表脚本如下：

```
DROP TABLE IF EXISTS 'role';
CREATE TABLE 'role' (
  'id' bigint(20) NOT NULL AUTO_INCREMENT,
  'name' varchar(255) COLLATE utf8_bin NOT NULL,
```

```
  PRIMARY KEY ('id')
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
DROP TABLE IF EXISTS 'user';
CREATE TABLE 'user' (
  'id' bigint(20) NOT NULL AUTO_INCREMENT,
  'password' varchar(255) COLLATE utf8_bin DEFAULT NULL,
  'username' varchar(255) COLLATE utf8_bin NOT NULL,
  PRIMARY KEY ('id'),
  UNIQUE KEY 'UK_sb8bbouer5wak8vyiiy4pf2bx' ('username')
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8 COLLATE=utf8_bin;

DROP TABLE IF EXISTS 'user_role';
CREATE TABLE 'user_role' (
  'user_id' bigint(20) NOT NULL,
  'role_id' bigint(20) NOT NULL,
  KEY 'FKa68196081fvovjhkek5m97n3y' ('role_id'),
  KEY 'FK859n2jvi8ivhui0rl0esws6o' ('user_id'),
  CONSTRAINT 'FK859n2jvi8ivhui0rl0esws6o' FOREIGN KEY ('user_id') REFERENCES 'user' (
'id'),
  CONSTRAINT 'FKa68196081fvovjhkek5m97n3y' FOREIGN KEY ('role_id') REFERENCES 'role'
('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
```

数据库的表建好之后，需要在数据库中初始化用户的信息数据，本案例的初始化用户信息的数据库脚本如下：

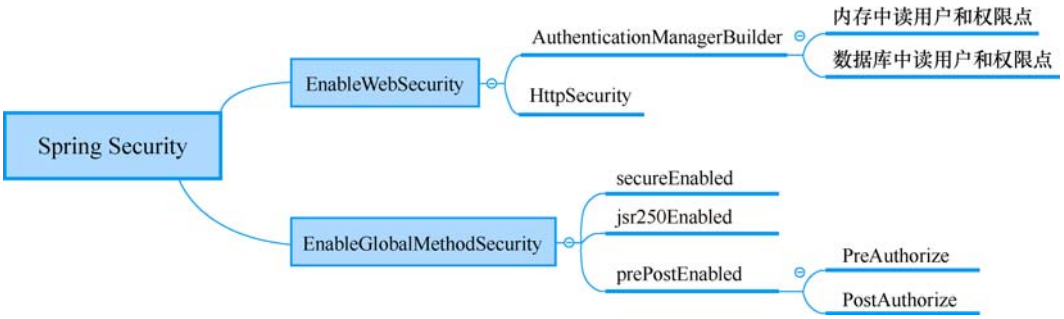
```
INSERT INTO user (id, username, password) VALUES (1, 'forezp', '123456');
INSERT INTO user (id, username, password)  VALUES (2, 'admin', '123456');
INSERT INTO role (id, name) VALUES (1, 'ROLE_USER');
INSERT INTO role (id, name) VALUES (2, 'ROLE_ADMIN');
INSERT INTO user_role (user_id, role_id) VALUES (1, 1);
INSERT INTO user_role (user_id, role_id) VALUES (2, 1);
INSERT INTO user_role (user_id, role_id) VALUES (2, 2);
```

启动程序之后，在浏览器上访问 http://localhost:8080，你会发现跟之前存在内存中的用户信息的认证效果是一样的。这说明 Spring Security 从数据库中获取了用户信息，并用作认证。

13.4 总结

使用 Spring Security 还是比较简单的，没有想象中那么复杂。首先引入 Spring Security 相关的依赖，然后写一个配置类，该配置类继承了 WebSecurityConfigurerAdapter，并在该配置类上加@EnableWebSecurity 注解开启 Web Security。再需要配置 AuthenticationManagerBuilder，AuthenticationManagerBuilder 配置了读取用户的认证信息的方式，可以从内存中读取，也可以从数据库中读取，或者用其他的方式。其次，需要配置 HttpSecurity，HttpSecurity 配置了请求

的认证规则，即哪些 URI 请求需要认证、哪些不需要，以及需要拥有什么权限才能访问。最后，如果需要开启方法级别的安全配置，需要通过在配置类上加@EnableGlobalMethodSecurity 注解开启，方法级别上的安全控制支持 secureEnabled、jsr250Enabled 和 prePostEnabled 这 3 种方式，用的最多的是 prePostEnabled。其中，prePostEnabled 包括 PreAuthorize 和 PostAuthorize 两种形式，一般只用到 PreAuthorize 这种方式。Spring Security 的思维导图如图 13-8 所示。



▲图 13-8　Spring Security 的思维导图

Spring Security 还有一些其他的特性，本章没有讲述，读者可以参看 Spring Security 的官方文档。

第 14 章　使用 Spring Cloud OAuth2 保护微服务系统

上一章全面讲解了 Spring Boot Security，本章将从以下 4 个方面讲述如何在 Spring Cloud 构建的微服务系统中使用 Spring Cloud OAuth2 来保护微服务系统。

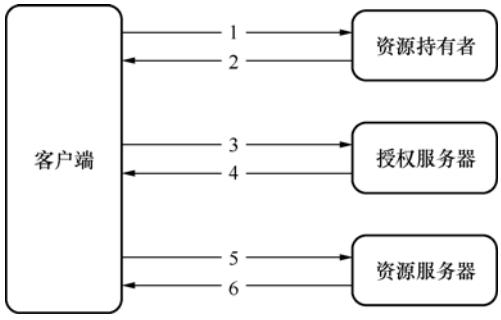
- 什么是 OAuth2。
- 如何使用 Spring OAuth2。
- 案例分析。
- 总结。

14.1 什么是 OAuth2

OAuth2 是一个标准的授权协议。OAuth2 取代了在 2006 年创建的 OAuth1 的工作，OAuth2 对 OAuth1 没有做兼容，即完全废弃了 OAuth1。OAuth2 允许不同的客户端通过认证和授权的形式来访问被其保护起来的资源。在认证和授权的过程中，主要包含以下 3 种角色。

- 服务提供方 Authorization Server。
- 资源持有者 Resource Server。
- 客户端 Client。

OAuth2 的认证流程如图 14-1 所示，具体如下。



▲图 14-1　OAuth2 的认证流程

（1）用户（资源持有者）打开客户端，客户端询问用户授权。

（2）用户同意授权。

（3）客户端向授权服务器申请授权。

（4）授权服务器对客户端进行认证，也包括用户信息的认证，认证成功后授权给予令牌。

（5）客户端获取令牌后，携带令牌向资源服务器请求资源。

（6）资源服务器确认令牌正确无误，向客户端释放资源。

14.2 如何使用 Spring OAuth2

OAuth2 协议在 Spring Resources 中的实现为 Spring OAuth2。Spring OAuth2 分为两部分，分别是 OAuth2 Provider 和 OAuth2 Client，下面来对二者逐一讲解。

14.2.1 OAuth2 Provider

OAuth2 Provider 负责公开被 OAuth2 保护起来的资源。OAuth2 Provider 需要配置代表用户的 OAuth2 客户端信息，被用户允许的客户端就可以访问被 OAuth2 保护的资源。OAuth2 Provider 通过管理和验证 OAuth2 令牌来控制客户端是否有权限访问被其保护的资源。另外，Oauth 2 Provider 还必须为用户提供认证 API 接口。根据认证 API 接口，用户提供账号和密码等信息，来确认客户端是否可以被 OAuth2 Provider 授权。这样做的好处就是第三方客户端不需要获取用户的账号和密码，通过授权的方式就可以访问被 OAuth2 保护起来的资源。

OAuth2 Provider 的角色被分为 Authorization Service（授权服务） 和 Resource Service（资源服务），通常它们不在同一个服务中，可能一个 Authorization Service 对应多个 Resource Service。Spring OAuth2 需配合 Spring Security 一起使用，所有的请求由 Spring MVC 控制器处理，并经过一系列的 Spring Security 过滤器。

在 Spring Security 过滤器链中有以下两个节点，这两个节点是向 Authorization Service 获取验证和授权的。

- 授权节点：默认为/oauth/authorize。
- 获取 Token 节点：默认为/oauth/token。

1. Authorization Server 配置

在配置 Authorization Server 时，需要考虑客户端（Client）从用户获取访问令牌的授权类型（例如授权代码、用户凭据、刷新令牌）。Authorization Server 需要配置客户端的详细信息和令牌服务的实现。

在任何实现了 AuthorizationServerConfigurer 接口的类上加@EnableAuthorizationServer 注解，开启 Authorization Server 的功能，以 Bean 的形式注入 Spring IoC 容器中，并需要实现以下 3 个配置。

- ClientDetailsServiceConfigurer：配置客户端信息。
- AuthorizationServerEndpointsConfigurer：配置授权 Token 的节点和 Token 服务。
- AuthorizationServerSecurityConfigurer：配置 Token 节点的安全策略。

下面来具体讲解这 3 个配置。

（1）ClientDetailsServiceConfigurer

客户端的配置信息既可以放在内存中，也可以放在数据库中，需要配置以下信息。

- clientId：客户端 Id，需要在 Authorization Server 中是唯一的。

- secret：客户端的密码。
- scope：客户端的域。
- authorizedGrantTypes：认证类型。
- authorities：权限信息。

客户端信息可以存储在数据库中，这样就可以通过更改数据库来实时更新客户端信息的数据。Spring OAuth2 已经设计好了数据库的表，且不可变。创建数据库的脚本如下：

```
DROP TABLE IF EXISTS 'clientdetails';
CREATE TABLE 'clientdetails' (
  'appId' varchar(128) NOT NULL,
  'resourceIds' varchar(256) DEFAULT NULL,
  'appSecret' varchar(256) DEFAULT NULL,
  'scope' varchar(256) DEFAULT NULL,
  'grantTypes' varchar(256) DEFAULT NULL,
  'redirectUrl' varchar(256) DEFAULT NULL,
  'authorities' varchar(256) DEFAULT NULL,
  'access_token_validity' int(11) DEFAULT NULL,
  'refresh_token_validity' int(11) DEFAULT NULL,
  'additionalInformation' varchar(4096) DEFAULT NULL,
  'autoApproveScopes' varchar(256) DEFAULT NULL,
  PRIMARY KEY ('appId')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS 'oauth_access_token';
CREATE TABLE 'oauth_access_token' (
  'token_id' varchar(256) DEFAULT NULL,
  'token' blob,
  'authentication_id' varchar(128) NOT NULL,
  'user_name' varchar(256) DEFAULT NULL,
  'client_id' varchar(256) DEFAULT NULL,
  'authentication' blob,
  'refresh_token' varchar(256) DEFAULT NULL,
  PRIMARY KEY ('authentication_id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS 'oauth_approvals';
CREATE TABLE 'oauth_approvals' (
  'userId' varchar(256) DEFAULT NULL,
  'clientId' varchar(256) DEFAULT NULL,
  'scope' varchar(256) DEFAULT NULL,
  'status' varchar(10) DEFAULT NULL,
  'expiresAt' datetime DEFAULT NULL,
  'lastModifiedAt' datetime DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
DROP TABLE IF EXISTS 'oauth_client_details';
CREATE TABLE 'oauth_client_details' (
  'client_id' varchar(256) NOT NULL,
  'resource_ids' varchar(256) DEFAULT NULL,
  'client_secret' varchar(256) DEFAULT NULL,
  'scope' varchar(256) DEFAULT NULL,
  'authorized_grant_types' varchar(256) DEFAULT NULL,
  'web_server_redirect_uri' varchar(256) DEFAULT NULL,
  'authorities' varchar(256) DEFAULT NULL,
  'access_token_validity' int(11) DEFAULT NULL,
  'refresh_token_validity' int(11) DEFAULT NULL,
  'additional_information' varchar(4096) DEFAULT NULL,
  'autoapprove' varchar(256) DEFAULT NULL,
  PRIMARY KEY ('client_id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS 'oauth_client_token';
CREATE TABLE 'oauth_client_token' (
  'token_id' varchar(256) DEFAULT NULL,
  'token' blob,
  'authentication_id' varchar(128) NOT NULL,
  'user_name' varchar(256) DEFAULT NULL,
  'client_id' varchar(256) DEFAULT NULL,
  PRIMARY KEY ('authentication_id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS 'oauth_code';
CREATE TABLE 'oauth_code' (
  'code' varchar(256) DEFAULT NULL,
  'authentication' blob
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS 'oauth_refresh_token';
CREATE TABLE 'oauth_refresh_token' (
  'token_id' varchar(256) DEFAULT NULL,
  'token' blob,
  'authentication' blob
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

（2）AuthorizationServerEndpointsConfigurer

在默认情况下，AuthorizationServerEndpointsConfigurer 配置开启了所有的验证类型，除了密码类型的验证，密码验证只有配置了 authenticationManager 的配置才会开启。AuthorizationServerEndpointsConfigurer 配置由以下 5 项组成。

- authenticationManager ：只有配置了该选项，密码认证才会开启。在大多数情况下都是密码验证，所以一般都会配置这个选项。

- userDetailsService：配置获取用户认证信息的接口，和上一章实现的 userDetailsService 类似。
- authorizationCodeServices：配置验证码服务。
- implicitGrantService：配置管理 implict 验证的状态。
- tokenGranter：配置 Token Granter。

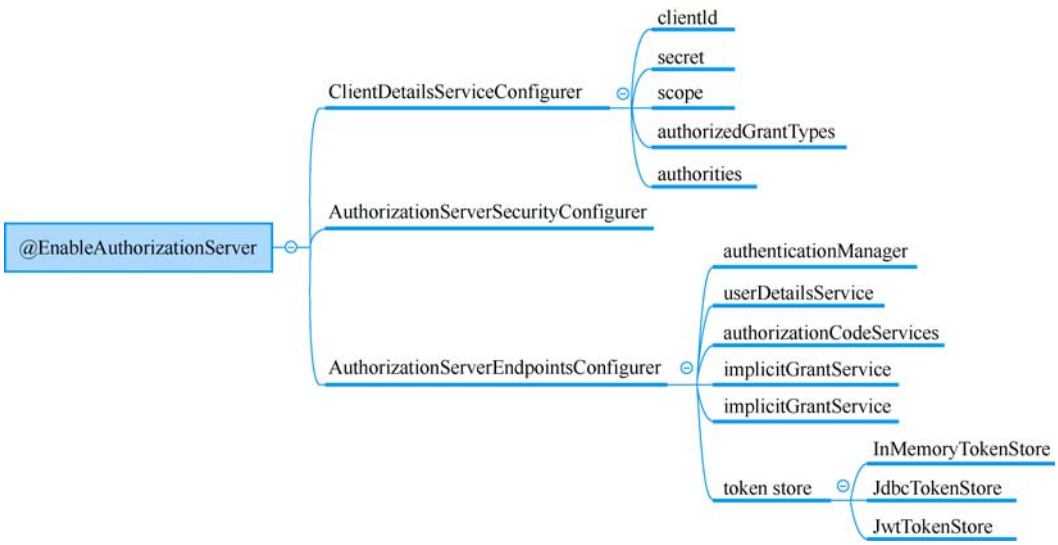
另外，需要设置 Token 的管理策略，目前支持以下 3 种。

- InMemoryTokenStore：Token 存储在内存中。
- JdbcTokenStore：Token 存储在数据库中。需要引入 spring-jdbc 的依赖包，并配置数据源，以及初始化 Spring OAuth2 的数据库脚本，即上一节的数据库脚本。
- JwtTokenStore：采用 JWT 形式，这种形式没有做任何的存储，因为 JWT 本身包含了用户验证的所有信息，不需要存储。采用这种形式，需要引入 spring-jwt 的依赖。

（3）AuthorizationServerSecurityConfigurer

如果资源服务和授权服务是在同一个服务中，用默认的配置即可，不需要做其他任何的配置。但是如果资源服务和授权服务不在同一个服务中，则需要做一些额外配置。如果采用 RemoteTokenServices（远程 Token 校验），资源服务器的每次请求所携带的 Token 都需要从授权服务做校验。这时需要配置“/oauth/check_token”校验节点的校验策略。

Authorization Server 的配置比较复杂，细节较多，Authorization Server 的配置思维导图如图 14-2 所示。通过在实现了 AuthorizationServerConfigurer 接口的类上加@EnableAuthorizationServer 注解，开启 AuthorizationServer 的功能，并注入 IoC 容器中。然后需要配置 ClientDetailsServiceConfigurer、AuthorizationServerSecurityConfigurer 和 AuthorizationServerEndpointsConfigurer，它们有很多可选的配置，需要读者慢慢理解。



▲图 14-2 Authorzition Server 的配置思维导图

2. Resource Server 的配置

下面讲解 Resource Server 的配置，Resource Server（可以是授权服务器，也可以是其他的资源服务）提供了受 OAuth2 保护的资源，这些资源为 API 接口、Html 页面、Js 文件等。Spring OAuth2 提供了实现此保护功能的 Spring Security 认证过滤器。在加了@Configuration 注解的配置类上加@EnableResourceServer 注解，开启 Resource Server 的功能，并使用 ResourceServerConfigurer 进行配置（如有必要），需要配置以下的内容：

- tokenServices：定义 Token Service。例如用 ResourceServerTokenservices 类，配置 Token 是如何编码和解码的。如果 Resource Server 和 Authorization Server 在同一个工程上，则不需要配置 tokenServices，如果不在同一个程序就需要配置。也可以用 RemoteTokenServices 类，即 Resource Server 采用远程授权服务器进行 Token 解码，这时也不需要配置此选项，本章案例采用此方式。
- resourceId：配置资源 Id。

14.2.2 OAuth2 Client

OAuth2 Client（客户端）用于访问被 OAuth2 保护起来的资源。客户端需要提供用于存储用户的授权码和访问令牌的机制，需要配置如下两个选项。

- Protected Resource Configuration（受保护资源配置）。
- Client Configuration（客户端配置）。

1. Protected Resource Configuration

使用 OAuth2ProtectedResourceDetails 类型的 Bean 来定义受保护的资源，受保护的资源具有以下属性。

- Id：资源的 Id，它在 Spring OAuth2 协议中没有用到，用于客户端寻找资源，不需要做配置，默认即可。
- clientId：OAuth2 Client 的 Id，和之前 OAuth2 Provider 中配置的一一对应。
- clientSecret：客户端密码，和之前 OAuth2 Provider 中配置的一一对应。
- accessTokenUri：获取 Token 的 API 节点。
- scope：客户端的域。
- clientAuthenticationScheme：有两种客户端验证类型，分别为 Http Basic 和 Form，默认为 Http Basic。
- userAuthorizationUri：如果用户需要授权访问资源，则用户将被重定向到的认证 Url。

2. Cilent Configuration

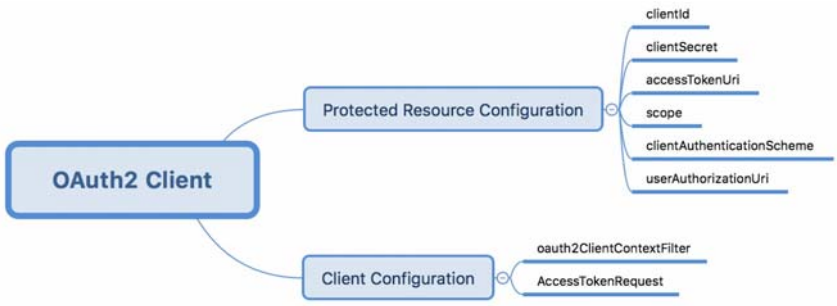
对于 OAuth2 Client 的配置，可以使用@ EnableOAuth2Client 注解来简化配置，另外还需要配置以下两项。

- 创建一个过滤器 Bean（Bean 的 Id 为 oauth2ClientContextFilter），用来存储当前请求

和上下文的请求。在请求期间，如果用户需要进行身份验证，则用户会被重定向到 OAuth2 的认证 Url。

- 在 Request 域内创建 AccessTokenRequest 类型的 Bean。

OAuth2 Client 配置相对简单一些，它的配置选项如图 14-3 所示。

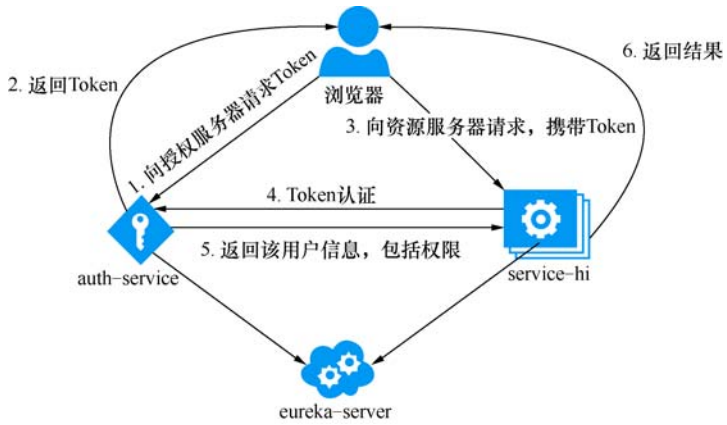


▲图 14-3 OAuth2 Client 的配置选项

14.3 案例分析

首先来看案例的架构设计，在这个案例中有 3 个工程，分别是服务注册中心工程 eureka-server、授权中心 Uaa 工程 auth-service 和资源工程 service-hi，如图 14-4 所示。

首先，浏览器向 auth-service 服务器提供客户端信息、用户名和密码，请求获取 Token。auth-service 确认这些信息无误后，根据该用户的信息生成 Token 并返回给浏览器。浏览器在以后的每次请求都需要携带 Token 给资源服务 service-hi，资源服务器获取到请求携带的 Token 后，通过远程调度将 Token 给授权服务 auth-service 确认。auth-service 确认 Token 正确无误后，将该 Token 对应的用户的权限信息返回给资源服务 service-hi。如果该 Token 对应的用户具有访问该 API 接口的权限，就正常返回请求结果，否则返回权限不足的错误提示。



▲图 14-4 案例架构设计图

14.3.1　编写 Eureka Server

整个项目采用的是 Maven 多 Module 的形式，并制定了项目的 Spring Boot 版本为 1.5.3，Spring Cloud 版本为 Dalston.RELEASE。

在主 Maven 工程下，创建一个 eureka-server 的 module 工程，创建完成后，在 eureka-server 工程的 pom 文件中引入 eureka server 的起步依赖，代码如下：

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

在 eureka-server 工程的配置文件 application.yml 中配置 Eureka Server，包括配置了程序的端口号为 8761，host 为 localhost，并配置了不自注册。具体的配置代码如下：

```
server:
  port: 8761
eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone:http://${eureka.instance.hostname}:${server.port}/eureka/
```

在程序的启动类 EurekaServerApplication 上加@EnableEurekaServer 注解，开启 Eureka Server 的功能，代码如下：

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

14.3.2　编写 Uaa 授权服务

1. 依赖管理 pom 文件

在主 Maven 工程下创建一个 Module 工程，取名为 auth-service，作为 Uaa 服务（授权服务），在 auth-service 工程的 pom 文件里引入工程所需的依赖，代码如下：

```
<dependencies>
     <dependency>
          <groupId>org.springframework.cloud</groupId>
          <artifactId>spring-cloud-starter-oauth2</artifactId>
     </dependency>
     <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-data-jpa</artifactId>
     </dependency>
     <dependency>
          <groupId>mysql</groupId>
          <artifactId>mysql-connector-java</artifactId>
     </dependency>
     <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-web</artifactId>
     </dependency>
     <dependency>
          <groupId>org.springframework.cloud</groupId>
          <artifactId>spring-cloud-starter-eureka</artifactId>
     </dependency>
</dependencies>
```

其中，spring-cloud-starter-oauth2 是对 spring-cloud-starter-security、spring-security-oauth2 和 spring-security-jwt 这 3 个起步依赖的整合。在工程中使用了 MySQL 数据库，引入了 MySQL 的连接驱动依赖 mysql-connector-java 和 JPA 的起步依赖 spring-boot-starter-data-jpa。在工程中使用了 Web 功能，引入了 Web 的起步依赖 spring-boot-starter-web。这个工程作为 Eureka Client，引入了 Eureka 的起步依赖 spring-cloud-starter-eureka。

2. 配置文件 application.yml

在工程的配置文件 application.yml 做如下配置：

```
spring:
  application:
    name: service-auth
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/spring-cloud-auth?useUnicode=true&characterEncod
ing=utf8&characterSetResults=utf8
    username: root
password: 123456
  jpa:
    hibernate:
```

```
      ddl-auto: update
    show-sql: true
server:
  context-path: /uaa
  port: 5000
security:
  oauth2:
    resource:
      filter-order: 3
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

在上面的配置中，配置了程序名为 service-auth，程序的端口号为 5000，context-path 为“/uaa”；配置了 MySQL 数据库的相关配置，包括数据源、用户和密码，其中数据库名为 spring-cloud-auth，需要初始化 12.3.1 节的数据库脚本；使用 JPA 作为 ORM 框架，并对 JPA 做了相关的配置；配置了服务注册中心的地址为 http://localhost:8761/eureka/；配置 security.oauth2.resource.filter-order 为 3，在 Spring Boot 1.5.x 版本，这是固定写法，在 Spring Boot 1.5.x 版本之前，默认即可。

3. 配置 Spring Security

由于 auth-service 需要对外暴露检查 Token 的 API 接口，所以 auth-service 也是一个资源服务，需要在工程中引入 Spring Security，并做相关的配置，对 auth-service 资源进行保护。配置代码如下：

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
   @Autowired
   private UserServiceDetail userServiceDetail;
   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http
             .authorizeRequests().anyRequest().authenticated()
             .and()
             .csrf().disable();
   }
   @Override
   protected void configure(AuthenticationManagerBuilder auth) throws Exception {
```

```
        auth.userDetailsService(userServiceDetail).passwordEncoder(new BCryptPasswordEn
coder());
    }
    @Override
    @Bean
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }
}
```

WebSecurityConfig 类通过@EnableWebSecurity 注解开启 Web 保护功能，通过@EnableGlobalMethodSecurity 注解开启在方法上的保护功能。WebSecurityConfig 类继承了 WebSecurityConfigurerAdapter 类，并复写了以下 3 个方法来做相关的配置.

- configure(HttpSecurity http)：HttpSecurity 中配置了所有的请求都需要安全验证
- configure(AuthenticationManagerBuilder auth)：AuthenticationManagerBuilder 中配置了验证的用户信息源和密码加密的策略，并且向 IoC 容器注入了 AuthenticationManager 对象。这需要在 OAuth2 中配置，因为在 OAuth2 中配置了 AuthenticationManager，密码验证才会开启。在本例中，采用的是密码验证。
- authenticationManagerBean()：配置了验证管理的 Bean。

UserServiceDetail 这个类和 13.3.4 节中的 UserService 是一样的，实现了 UserDetailsService 接口，并使用了 BCryptPasswordEncoder 对密码进行加密，代码如下：

```
@Service
public class UserServiceDetail implements UserDetailsService {
    @Autowired
    private UserDao userRepository;
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundExce
ption {
        return userRepository.findByUsername(username);
    }
}
```

UserDao 类继承了 JpaRepository，在 UserDao 类中写一个根据用户名获取用户的方法，代码如下：

```
public interface UserDao extends JpaRepository<User, Long>{
     User findByUsername(String username);
}
```

与 13.3.4 节一样，User 类需要实现 UserDetails 接口，Role 类需要实现 GrantedAuthority 接口，这里就不再重复列出这两个类的代码了。

4. 配置 Authorization Server

首先列出配置代码，然后根据代码对每一个配置做详细说明，代码如下：

```
@SpringBootApplication
@EnableResourceServer
@EnableEurekaClient
public class ServiceAuthApplication {
     @Autowired
     @Qualifier("dataSource")
     private DataSource dataSource;
     public static void main(String[] args) {
          SpringApplication.run(ServiceAuthApplication.class, args);
     }

     @Configuration
     @EnableAuthorizationServer
     protected  class OAuth2AuthorizationConfig extends AuthorizationServerConfigurer
Adapter {
       //private TokenStore tokenStore = new InMemoryTokenStore();
       JdbcTokenStore tokenStore=new JdbcTokenStore(dataSource);

       @Autowired
       @Qualifier("authenticationManagerBean")
       private AuthenticationManager authenticationManager;

       @Autowired
       private UserServiceDetail userServiceDetail;
       @Override
       public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
         clients.inMemory()
            .withClient("browser")
            .authorizedGrantTypes("refresh_token", "password")
            .scopes("ui")
            .and()
            .withClient("service-hi")
            .secret("123456")
            .authorizedGrantTypes("client_credentials", "refresh_token","password")
            .scopes("server");
          }
     @Override
     public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws E
xception {
          endpoints
               .tokenStore(tokenStore)
               .authenticationManager(authenticationManager)
```

```
                .userDetailsService(userServiceDetail);
        }
    @Override
    public void configure(AuthorizationServerSecurityConfigurer oauthServer) throws
Exception {
        oauthServer
                .tokenKeyAccess("permitAll()")
                .checkTokenAccess("isAuthenticated()");
        }
    }
}
```

在程序启动类 ServiceAuthApplication 加上@EnableEurekaClient 注解，开启 Eureka Client 客户端的功能，加上@EnableResourceServer 注解，开启 Resource Server。程序需要对外暴露获取 Token 的 API 接口和验证 Token 的 API 接口，所以该程序也是一个资源服务。

OAuth2AuthorizationConfig 类继承 AuthorizationServerConfigurerAdapter，并在这个类上加上注解@EnableAuthorizationServer，开启授权服务的功能。作为授权服务需要配置 3 个选项，分别为 ClientDetailsServiceConfigurer、AuthorizationServerEndpointsConfigurer 和 AuthorizationServer SecurityConfigurer。

其中，ClientDetailsServiceConfigurer 配置了客户端的一些基本信息，clients.inMemory()方法配置了将客户端的信息存储在内存中，.withClient("browser")方法创建了一个 clientId 为 browser 的客户端，authorizedGrantTypes("refresh_token", "password")方法配置了验证类型为 refresh_token 和 password，.scopes("ui")方法配置了客户端域为“ui”。接着创建了另一个 client，它的 Id 为“service-hi”。

AuthorizationServerEndpointsConfigurer 需要配置 tokenStore、authenticationManager 和 userServiceDetail。其中，tokenStore（Token 的存储方式）采用的方式是将 Token 存储在内存中，即使用 InMemoryTokenStore。如果资源服务和授权服务是同一个服务，用 InMemoryTokenStore 是最好的选择。如果资源服务和授权服务不是同一个服务，则不用 InMemoryTokenStore 进行存储 Token。因为当授权服务出现故障，需要重启服务，之前存在内存中 Token 全部丢失，导致资源服务的 Token 全部失效。另外一种方式是用 JdbcTokenStore，即使用数据库去存储，使用 JdbcTokenStore 存储需要引入连接数据库依赖，如本例中的 MySQL 连接器、JPA，并且需要初始化 14.2.1 节的数据库脚本。authenticationManager 需要配置 AuthenticationManager 这个 Bean，这个 Bean 来源于 WebSecurityConfigurerAdapter 中的配置，只有配置了这个 Bean 才会开启密码类型的验证。最后配置了 userDetailsService，用来读取验证用户的信息。

AuthorizationServerSecurityConfigurer 配置了获取 Token 的策略，在本案例中对获取 Token 请求不进行拦截，只需要验证获取 Token 的验证信息，这些信息准确无误，就返回 Token。另外配置了检查 Token 的策略。

5. 暴露 Remote Token Services 接口

本案例采用 RemoteTokenServices 这种方式对 Token 进行验证。如果其他资源服务需要验证 Token，则需要远程调用授权服务暴露的验证 Token 的 API 接口。本案例中验证 Token 的 API 接口的代码如下：

```
@RestController
@RequestMapping("/users")
public class UserController {

     @RequestMapping(value = "/current", method = RequestMethod.GET)
     public Principal getUser(Principal principal) {
          return principal;
     }
}
```

6. 获取 Token

启动 auth-service 服务，在终端上用 Curl 命令模拟请求获取 Token，Curl 命令如下：

```
curl service-hi:123456@localhost:5000/uaa/oauth/token -d grant_type=password -d usern
ame=fzp -d password=123456
```

返回结果如下：

```
{"access_token":"50c2476f-34fd-4c44-a608-6aa7021c1cb9","token_type":"bearer","refresh
_token":"37b8baaf-6365-4efc-8eef-397a1039f56d","expires_in":43199,"scope":"server"}
```

也可以用 postman 或者 ajax 请求获取 Token。获取 Token 的 API 接口使用了基本认证（Http Basic Authentication）。基本认证是一种用来允许 Web 浏览器或其他客户端程序在请求时提供用户名和口令形式的身份凭证来验证客户端的。用户名和口令形式的身份凭证是在用户名后追加一个冒号，然后串接上口令，将拼接后的字符串用 Base64 算法编码得到的。如本案例中客户端的用户名为 service-hi，口令为 123456，将它们组合为 service-hi:123456，进行 Base64 加密，得到“c2VydmljZS1oaToxMjM0NTY=”。

使用 ajax 方式进行请求获取 Token 的代码如下：

```
$.ajax({
        url: 'localhost:5000/uaa/oauth/token',
        datatype: 'json',
        type: 'post',
        headers: {'Authorization': 'Basic c2VydmljZS1oaToxMjM0NTY='},
        async: false,
        data: {
            username: fzp,
            password: 123456,
```

```
            grant_type: 'password'
        },
        success: function (data) {
        },
        error: function () {
      }
    });
```

那么如何使用得到的 Token 呢？在用户访问受保护的资源时，在请求的 Header 中加上参数名为“Authorization”，参数值为“Bearer {Token}”的参数。

14.3.3 编写 service-hi 资源服务

1. 项目依赖

在主 Maven 工程下，创建一个 Module 工程，取名为 service-hi，这个工程作为资源服务。在 service-hi 工程的 pom 文件引入项目所需的依赖，代码如下：

```
<dependencies>
     <dependency>
          <groupId>org.springframework.cloud</groupId>
          <artifactId>spring-cloud-starter-eureka</artifactId>
     </dependency>
     <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-web</artifactId>
     </dependency>
     <dependency>
          <groupId>org.springframework.cloud</groupId>
          <artifactId>spring-cloud-starter-feign</artifactId>
     </dependency>
     <dependency>
          <groupId>org.springframework.cloud</groupId>
          <artifactId>spring-cloud-starter-oauth2</artifactId>
     </dependency>
     <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-data-jpa</artifactId>
     </dependency>
     <dependency>
          <groupId>mysql</groupId>
          <artifactId>mysql-connector-java</artifactId>
     </dependency>
</dependencies>
```

在工程中用到了 MySQL 数据库，采用了 JPA 的 ORM 框架来操作数据库，所以需要在工

程的 pom 文件引入 JPA 的起步依赖 spring-boot-starter-data-jpa 和 MySQL 数据库连接器依赖 mysql-connector-java。作为 Eureka Client，需要在工程的 pom 文件引入 Eureka 的起步依赖 spring-cloud-starter-eureka。作为 Web 服务器，需要在工程的 pom 文件引入 Web 的起步依赖 spring-boot-starter-web。另外使用 Feign 作为远程调度框架，需要在工程的 pom 文件引入 Feign 的起步依赖 spring-cloud-starter-feign。最后作为资源服务器，需要在工程的 pom 文件引入 OAuth2 的起步依赖 spring-cloud-starter- oauth2。

2. 配置文件 application.yml

在工程的配置文件 application.yml 做程序的相关配置，具体配置如下：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
server:
  port: 8762
spring:
  application:
    name: service-hi
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/spring-cloud-auth?useUnicode=true&characterEncod
ing=utf8&characterSetResults=utf8
    username: root
    password: 123456

  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true

security:
  oauth2:
    resource:
      user-info-uri: http://localhost:5000/uaa/users/current
    client:
      clientId: service-hi
      clientSecret: 123456
      accessTokenUri: http://localhost:5000/uaa/oauth/token
      grant-type: client_credentials,password
      scope: server
```

在上面的配置文件中，配置了程序名为 service-hi，端口为 8762，服务注册中心的地址为 http://localhost:8761/eureka/，配置了数据库的连接驱动、数据库连接地址、数据库用户名和密码，

以及 JPA 的相关配置。然后配置了 security.oauth2.resource，指定了 user-info-uri 的地址，用于获取当前 Token 的用户信息，配置了 security.oauth2.client 的相关信息，以及 clientId、clientSecret 等信息，这些配置需要和在 Uaa 服务中配置的一一对应。

3. 配置 Resource Server

service-hi 工程作为 Resource Server（资源服务），需要配置 Resource Server 的相关配置，配置代码如下：

```
@Configuration
@EnableResourceServer
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ResourceServerConfigurer extends ResourceServerConfigurerAdapter {

   @Override
   public void configure(HttpSecurity http) throws Exception {

      http.authorizeRequests()
             .antMatchers("/user/registry").permitAll()
             .anyRequest().authenticated();
   }
}
```

在 ResourceServerConfigurer 类上加@EnableResourceServer 注解，开启 Resource Server 的功能，加@EnableGlobalMethodSecurity 注解，开启方法级别的保护。ResourceServerConfigurer 类继承 ResourceServerConfigurerAdapter 类，并重写 configure(HttpSecurity http)方法，通过 ant 表达式，配置哪些请求需要验证，哪些请求不需要验证。如本案例中“/user/register”的接口不需要验证，其他所有的请求都需要验证。

4. 配置 OAuth2 Client

OAuth2 Client 用来访问被 OAuth2 保护的资源。service-hi 作为 OAuth2 Client，它的配置代码如下：

```
@EnableOAuth2Client
@EnableConfigurationProperties
@Configuration
public class OAuth2ClientConfig {

   @Bean
   @ConfigurationProperties(prefix = "security.oauth2.client")
   public ClientCredentialsResourceDetails clientCredentialsResourceDetails() {
      return new ClientCredentialsResourceDetails();
   }
```

```
    @Bean
    public RequestInterceptor oauth2FeignRequestInterceptor(){
        return new OAuth2FeignRequestInterceptor(new DefaultOAuth2ClientContext(), clie
ntCredentialsResourceDetails());
    }

    @Bean
    public OAuth2RestTemplate clientCredentialsRestTemplate() {
        return new OAuth2RestTemplate(clientCredentialsResourceDetails());
    }
}
```

在 14.2 节讲述了如何配置 OAuth2 Client，简单来说，需要配置 3 个选项：一是配置受保护的资源的信息，即 ClientCredentialsResourceDetails；二是配置一个过滤器，存储当前请求和上下文；三是在 Request 域内创建 AccessTokenRequest 类型的 Bean。

现在通过上述代码来具体说明，在 OAuth2ClientConfig 类上加@EnableOAuth2Client 注解，开启 OAuth2 Client 的功能；并配置了一个 ClientCredentialsResourceDetails 类型的 Bean，该 Bean 是通过读取配置文件中前缀为 security.oauth2.client 的配置来获取 Bean 的配置属性的；注入一个 OAuth2FeignRequestInterceptor 类型过滤器的 Bean；最后注入了一个用于向 Uaa 服务请求的 OAuth2RestTemplate 类型的 Bean。

到目前为止，授权服务、资源服务和 OAuth2 客户端都已经搭建完毕，现在写一个注册 API 接口来做测试。

5. 编写用户注册接口

首先编写一个 User 类，在本案例总共采用了 JPA 作为 ORM 框架，需要在 User 类加上 JPA 的注解，同 Uaa 服务的 User 类一样，代码如下：

```
@Entity
public class User  {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false,  unique = true)
    private String username;

    @Column
    private String password;
  …省略部分代码
}
```

数据操作类 UserDao 继承了 JpaRepository，UserDao 具备了基本的操作数据库单表的基本方法，代码如下：

```
public interface UserDao extends JpaRepository<User, Long> {
}
```

Service 层的 UserServiceImpl 类包含一个创建用户逻辑的方法，其中用到 BCryptPasswordEncoder 类来加密密码，代码如下：

```
@Service
public class UserServiceImpl implements UserService {
    private static final BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();

    @Autowired
    private UserDao userDao;
    @Override
    public User create(User user) {
        String hash = encoder.encode(user.getPassword());
        user.setPassword(hash);
        User u=userDao.save(user);
        return u;
    }
}
```

编写 UserController 类，在类中有一个注册的 API 接口，代码如下：

```
@RestController
@RequestMapping("/user")
public class UserController {
    @Autowired
    private UserServiceImpl userService;
    @RequestMapping(value = "/registry",method = RequestMethod.POST)
    public User createUser( @RequestParam("username") String username
    , @RequestParam("password") String password) {
        return userService.create(username,password);
    }
}
```

编写一个测试类 HiController，其中有 3 个接口：第一个 API 接口“/hi”，不需要任何权限，只需要验证 Header 中的 Token 正确与否，Token 正确即可访问；第二个 API 接口“/hello”，需要“ROLE_ADMIN”权限；第三个接口“/getPrinciple”，用户获取当前 Token 用户信息。代码如下：

```
@RestController
public class HiController {
```

```
   Logger logger= LoggerFactory.getLogger(HiController.class);
   @Value("${server.port}")
String port;

   @RequestMapping("/hi")
   public String home() {
      return "hi :"+",i am from port:" +port;
   }

   @PreAuthorize("hasAuthority('ROLE_ADMIN')")
   @RequestMapping("/hello")
   public String hello (){
      return "hello you!";
   }
   @GetMapping("/getPrinciple")
   public OAuth2Authentication getPrinciple(OAuth2Authentication oAuth2Authentication
, Principal principal,
                        Authentication authentication){

    logger.info(oAuth2Authentication.getUserAuthentication().getAuthorities().toString());
          logger.info(oAuth2Authentication.toString());
          logger.info("principal.toString()"+principal.toString());
          logger.info("principal.getName()"+principal.getName());   logger.info("auth
entication:"+authentication.getAuthorities().toString());
          return oAuth2Authentication;
    }
}
```

下面来演示整个流程。

（1）通过 Curl 命令模拟请求，调用注册的 API 接口，注册一个用户，Curl 命令如下：

```
curl -d "username=miya&password=123456" "localhost:8762/user/registry"
```

注册成功，返回结果如下：

```
{"id":4,"username":"miya","password":"$2a$10$BYslYRseJmfdBl.SKtOD2e13XcZ69e.j4CUbpS.H
xKufaTKGtpEQG","authorities":null,"enabled":true,"accountNonExpired":true,"accountNon
Locked":true,"credentialsNonExpired":true}
```

（2）通过 Curl 命令模拟请求，调用获取 Token 的 API 接口，Curl 命令如下：

```
curl service-hi:123456@localhost:5000/uaa/oauth/token -d grant_type=password -d usern
ame=miya -d password=123456
```

获取 Token 成功，返回结果如下：

```
{"access_token":"baea9ed0-1970-4fd4-b616-fde2a75a41ee","token_type":"bearer","refresh
_token":"3b7a4a89-b5b8-43c4-bb3d-5a38171caf5b","expires_in":43182,"scope":"server"}
```

（3）通过 Curl 命令模拟请求，访问不需要权限点的接口“/hi”，Curl 命令如下：

```
curl -l -H "Authorization:Bearer baea9ed0-1970-4fd4-b616-fde2a75a41ee" -X GET "localh
ost:8762/hi"
```

返回结果如下：

```
hi :i am from port:8762
```

（4）通过 Curl 命令模拟请求，访问需要有“ROLE_ADMIN”权限点的 API 接口“/hello”，Curl 命令如下：

```
curl -l -H "Authorization:Bearer baea9ed0-1970-4fd4-b616-fde2a75a41ee" -X GET "localh
ost:8762/hello"
```

由于该用户没有“ROLE_ADMIN”权限点，所以没有权限访问该 API 接口，返回结果如下：

```
{"error":"access_denied","error_description":"不允许访问"}
```

（5）在数据库中给予该用户“ROLE_ADMIN”权限，在数据库中执行以下脚本：

```
INSERT INTO 'role' VALUES ('1', 'ROLE_USER'), ('2', 'ROLE_ADMIN');
INSERT INTO 'user_role' VALUES('4', '2');
```

（6）给予该用户“ROLE_ADMIN 权限”后，重新访问 API 接口“/hello”，Curl 命令如下：

```
curl -l -H "Authorization:Bearer baea9ed0-1970-4fd4-b616-fde2a75a41ee" -X GET "localh
ost:8762/hello"
```

获取的返回结果如下：

```
hello you!
```

从上面的请求返回的结果可知，给该用户加上“ROLE_ADMIN”权限后，该请求能够获取正常的返回结果。由此可见，被 Spring Cloud OAuth2 保护的资源服务，是需要验证请求的用户信息和该用户所具有的权限的，验证通过，则正确返回结果，否则返回“不允许访问”的结果。

14.4 总结

本章案例的架构有改进之处，例如在资源服务器加一个登录接口，该接口不受 Spring

Security 保护。登录成功后，service-hi 远程调用 auth-service 获取 Token 返回给浏览器，浏览器以后所有的请求都需要携带该 Token。

这个架构存在的缺陷就是每次请求都需要资源服务内部远程调度 auth-service 服务来验证 Token 的正确性，以及该 Token 对应的用户所具有的权限，额外多了一次内部请求。如果在高并发的情况下，auth-service 需要集群部署，并且需要做缓存处理。本案例中没有做以上这些优化工作。下一章将讲述如何使用 Spring Security OAuth2 以 JWT 的形式来保护 Spring Cloud 构建的微服务系统。

第 15 章　使用 Spring Security OAuth2 和 JWT 保护微服务系统

上一章讲述了如何通过 Spring Security OAuth2 来保护 Spring Cloud 架构的微服务系统。上一章的系统有一个缺陷，即每次请求都需要经过 Uaa 服务去验证当前 Token 的合法性，并且需要查询该 Token 对应的用户的权限。在高并发场景下，会存在性能瓶颈，改善的方法是将 Uaa 服务集群部署并加上缓存。本章针对上一章的系统的缺陷，采用 Spring Security OAuth2 和 JWT 的方式，避免每次请求都需要远程调度 Uaa 服务。采用 Spring Security OAuth2 和 JWT 的方式，Uaa 服务只验证一次，返回 JWT。返回的 JWT 包含了用户的所有信息，包括权限信息。

本章主要从以下 3 个方面来讲解。

- JWT 详解。
- Spring Security OAuth2 和 JWT 保护微服务系统案例详解。
- 总结。

15.1 JWT 简介

本节将主要从以下 4 个方面讲解 JWT。

- 什么是 JWT。
- JWT 的应用场景。
- JWT 的结构。
- 如何使用 JWT。

15.1.1　什么是 JWT

JSON Web Token（JWT）是一种开放的标准（RFC 7519），JWT 定义了一种紧凑且自包含的标准，该标准旨在将各个主体的信息包装为 JSON 对象。主体信息是通过数字签名进行加密和验证的。常使用 HMAC 算法或 RSA（公钥/私钥的非对称性加密）算法对 JWT 进行签名，安全性很高。下面进一步解释它的特点。

- 紧凑型（compact）：由于是加密后的字符串，JWT 数据体积非常小，可通过 POST 请求参数或 HTTP 请求头发送。另外，数据体积小意味着传输速度很快。

- 自包含（self-contained）：JWT 包含了主体的所有信息，所以避免了每个请求都需要向 Uaa 服务验证身份，降低了服务器的负载。

15.1.2　JWT 的结构

JWT 由 3 个部分组成，分别以“.”分隔，组成部分如下。

- Header（头）。
- Payload（有效载荷）。
- Signature（签名）。

因此，JWT 的通常格式如下：

```
xxxxx.yyyyy.zzzzz
```

下面依次来讲解这 3 个组成部分。

（1）Header

Header 通常由两部分组成：令牌的类型（即 JWT）和使用的算法类型，如 HMAC、SHA256 和 RSA。例如：

```
{
   “alg”：“HS256”,
   “typ”：“JWT”
}
```

将 Header 用 Base64 编码作为 JWT 的第一部分。

（2）Payload

这是 JWT 的第二部分，包含了用户的一些信息和 Claim（声明、权利）。有 3 种类型的 Claim：保留、公开和私人。一个典型的 Payload 如下：

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

将 Payload 进行 Base64 编码作为 JWT 的第二部分。

（3）Signature

要创建签名部分，需要将 Base64 编码后的 Header、Payload 和秘钥进行签名，一个典型的格式如下：

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

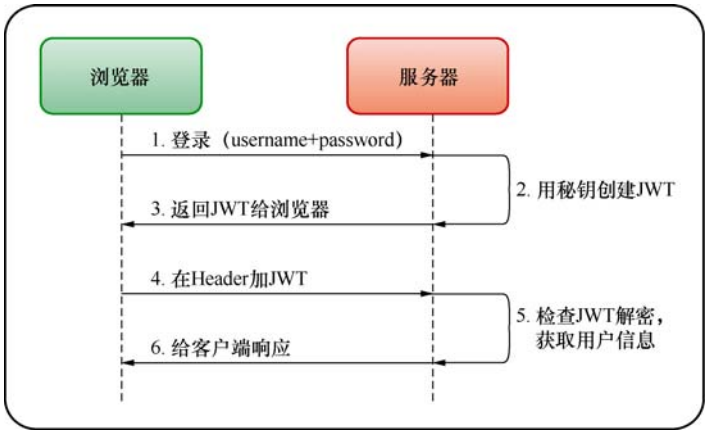
15.1.3 JWT 的应用场景

什么时候应该使用 JWT 呢？JWT 的使用场景如下。

- 认证：这是使用 JWT 最常见的场景。一旦用户登录成功获取 JWT 后，后续的每个请求将携带该 JWT。该 JWT 包含了用户信息、权限点等信息，根据该 JWT 包含的信息，资源服务可以控制该 JWT 可以访问的资源范围。因为 JWT 的开销很小，并且能够在不同的域中使用，单点登录是一个广泛使用 JWT 的场景。
- 信息交换：JWT 是在各方之间安全传输信息的一种方式，JWT 使用签名加密，安全性很高。另外，当使用 Header 和 Payload 计算签名时，还可以验证内容是否被篡改。

15.1.4 如何使用 JWT

下面来看最常见的应用场景，即认证，如图 15-1 所示。客户端通过提供用户名、密码向服务器请求获取 JWT，服务器判断用户名和密码正确无误之后，将用户信息和权限点经过加密以 JWT 的形式返回给客户端。在以后的每次请求中，获取到该 JWT 的客户端都需要携带该 JWT，这样做的好处就是以后的请求都不需要通过 Uaa 服务来判断该请求的用户以及该用户的权限。在微服务系统中，可以利用 JWT 实现单点登录。



▲图 15-1 JWT 认证流程

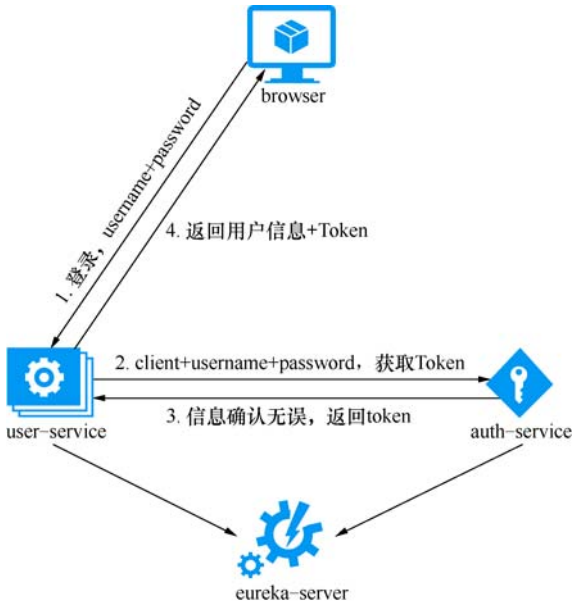
15.2 案例分析

15.2.1 案例架构设计

在本案例中有 3 个工程，分别为 eureka-server、auth-service 和 user-service。其中 auth-service 和 user-service 向 eureka-server 注册服务。auth-service 负责授权，授权需要用户提供客户端的 clientId 和 password，以及授权用户的 username 和 password。这些信息准备无误之后，auth-service 返回 JWT，该 JWT 包含了用户的基本信息和权限点信息，并通过 RSA 加密。user-service 作

为资源服务，它的资源已经被保护起来了，需要相应的权限才能访问。user-service 服务得到用户请求的 JWT 后，先通过公钥解密 JWT，得到该 JWT 对应的用户的信息和用户的权限信息，再判断该用户是否有权限访问该资源。

其中，在 user-service 服务的登录 API 接口（登录 API 接口不受保护）中，当用户名和密码验证正确后，通过远程调用向 auth-service 获取 JWT，并返回 JWT 给用户。用户获取到 JWT 之后，以后的每次请求都需要在请求头中传递该 JWT，从而资源服务能够根据 JWT 来进行权限验证。架构图如图 15-2 所示。



▲图 15-2　案例架构图

15.2.2　编写主 Maven 工程

使用 IDEA 创建一个 Maven 工程作为主 Maven 工程，采用的 Spring Boot 版本为 1.5.3，Spring Cloud 版本为 Dalston，JDK 版本为 1.8，工程的 pom 文件的代码如下：

```
<?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.forezp</groupId>
   <artifactId>cloud-oauth2-jwt</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>pom</packaging>
```

```
    <parent>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-parent</artifactId>
       <version>1.5.2.RELEASE</version>
       <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <properties>
       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
       <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
       <java.version>1.8</java.version>
       <spring-cloud.version>Dalston.RELEASE</spring-cloud.version>
    </properties>

    <dependencies>
       <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-test</artifactId>
          <scope>test</scope>
       </dependency>
    </dependencies>

    <dependencyManagement>
       <dependencies>
          <dependency>
             <groupId>org.springframework.cloud</groupId>
             <artifactId>spring-cloud-dependencies</artifactId>
             <version>${spring-cloud.version}</version>
             <type>pom</type>
             <scope>import</scope>
          </dependency>
       </dependencies>
    </dependencyManagement>
</project>
```

15.2.3 编写 Eureka Server

在主 Maven 工程下，创建一个 eureka-server 的 Module 工程，作为服务注册中心的工程。在工程的 pom 文件引入相应的依赖，包括继承了主 Maven 工程的 pom 文件，并引入 Eureka Server 的起步依赖，代码如下：

```
<parent>
     <groupId>com.forezp</groupId>
     <artifactId>cloud-oauth2-jwt</artifactId>
     <version>1.0-SNAPSHOT</version>
</parent>
<dependencies>
```

```
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka-server</artifactId>
    </dependency>
</dependencies>
```

在工程的配置文件 application.yml 中，配置程序的端口号为 8761，并配置不自注册，配置代码如下：

```
server:
  port: 8761
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
    serviceUrl:
      defaultZone: http://localhost:${server.port}/eureka/
```

在程序的启动类 EurekaServerApplication 上加@SpringBootApplication 注解，开启 Eureka Server 功能。代码如下：

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

只需要以上几步，Eureka Server 工程就搭建完毕了。

15.2.4　编写 Uaa 授权服务

1. 引入依赖

在主 Maven 工程下新建一个 Module 工程，取名为 uaa-service。工程的 pom 文件继承了主 Maven 工程的 pom 文件，并引入工程所需的依赖，包括连接数据库的依赖 mysql-connector-java 和 JPA 的起步依赖 spring-boot-starter-data-jpa、Web 的起步依赖 spring-boot-starter-web、Eureka 客户端的起步依赖 spring-cloud-starter-eureka，以及 Spring Cloud OAuth2 的起步依赖 spring-cloud-starter-oauth2。

其中，Spring Cloud OAuth2 的起步依赖包含了 Spring Security OAuth2 和 Spring Security JWT 的等依赖，代码如下：

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

```
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-oauth2</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
    </dependencies>
```

2. 配置文件

在程序的配置文件 aplication.yml 中配置程序的名称为 uaa-service，端口号为 9999，以及连接数据库驱动、JPA 的配置和服务的注册地址，代码如下：

```
spring:
  application:
    name: uaa-service
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/spring-cloud-auth?useUnicode=true&characterEncod
ing=utf8&characterSetResults=utf8
    username: root
    password: 123456
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
server:
  port: 9999
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

3. 配置 Spring Security

uaa-service 服务对外提供获取 JWT 的 API 接口，uaa-service 服务是一个授权服务器，同时也是资源服务器，需要配置该服务的 Spring Security，配置代码如下：

```
@Configuration
@EnableWebSecurity
class WebSecurityConfig extends WebSecurityConfigurerAdapter {

   @Override
   @Bean
   public AuthenticationManager authenticationManagerBean() throws Exception {
      return super.authenticationManagerBean();
   }
   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http
          .csrf().disable()
             .exceptionHandling()
             .authenticationEntryPoint((request, response, authException)     -> resp
onse.sendError(HttpServletResponse.SC_UNAUTHORIZED))
          .and()
             .authorizeRequests()
             .antMatchers("/**").authenticated()
          .and()
             .httpBasic();
   }

   @Autowired
   UserServiceDetail userServiceDetail;
   @Override
   protected void configure(AuthenticationManagerBuilder auth) throws Exception {
      auth.userDetailsService(userServiceDetail)
             .passwordEncoder(new BCryptPasswordEncoder());
   }
}
```

在上面的配置类中，通过@EnableWebSecurity 注解开启 Web 资源的保护功能。在 configure（HttpSecurity http）方法中配置所有的请求都需要验证，如果请求验证不通过，则重定位到 401 的界面。在 configure(AuthenticationManagerBuilder auth)方法中配置验证的用户信息源、密码加密的策略。向 IoC 容器注入 AuthenticationManager 对象的 Bean，该 Bean 在 OAuth2 的配置中使用，因为只有在 OAuth2 中配置了 AuthenticationManager，密码类型的验证才会开启。在本案例中，采用的是密码类型的验证。

采用 BCryptPasswordEncoder 对密码进行加密，在创建用户时，密码加密也必须使用这个类。

使用了 UserServiceDetail 这个类，这个类与 11.3.4 节中的 UserService 类是一样的，实现了 UserDetailsService 接口，代码如下：

```
@Service
public class UserServiceDetail implements UserDetailsService {
```

```
    @Autowired
    private UserDao userRepository;
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundExce
ption {
        return userRepository.findByUsername(username);
    }
}
```

UserDao 继承 JpaRepository，有一个根据用户名获取用户的方法，代码如下：

```
public interface UserDao extends JpaRepository<User, Long>{
     User findByUsername(String username);
}
```

与之前章节一样，User 对象需要实现 UserDetails 接口，Role 对象需要实现 GrantedAuthority 接口，在这里就不重复列出这两个类的代码了，请读者查看 11.3.4 节。

4. 配置 Authorization Server

在 OAuth2Config 这个类中配置 AuthorizationServer，其代码如下：

```
@Configuration
@EnableAuthorizationServer
public class OAuth2Config extends AuthorizationServerConfigurerAdapter {
    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
        clients.inMemory()
                .withClient("user-service")
                .secret("123456")
                .scopes("service")
                .authorizedGrantTypes("refresh_token", "password")
                .accessTokenValiditySeconds(3600);
    }

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exc
eption {
        endpoints.tokenStore(tokenStore()).tokenEnhancer(jwtTokenEnhancer()).authentica
tionManager(authenticationManager);
    }

    @Autowired
    @Qualifier("authenticationManagerBean")
    private AuthenticationManager authenticationManager;

    @Bean
```

```
    public TokenStore tokenStore() {
        return new JwtTokenStore(jwtTokenEnhancer());
    }

    @Bean
    protected JwtAccessTokenConverter jwtTokenEnhancer() {
        KeyStoreKeyFactory keyStoreKeyFactory = new KeyStoreKeyFactory(new ClassPathRes
ource("fzp-jwt.jks"), "fzp123".toCharArray());
        JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
        converter.setKeyPair(keyStoreKeyFactory.getKeyPair("fzp-jwt"));
        return converter;
    }
}
```

在上面的配置代码中，OAuth2Config 类继承了 AuthorizationServerConfigurerAdapter 类，并在 OAuth2Config 类加上@EnableAuthorizationServer 注解，开启 Authorization Server 的功能。作为 Authorization Server 需要配置两个选项，即 ClientDetailsServiceConfigurer 和 AuthorizationServerEndpointsConfigurer。

其中，ClientDetailsServiceConfigurer 配置了客户端的一些基本信息，clients.inMemory()方法是将客户端的信息存储在内存中，.withClient("user-service")方法创建了一个 ClientId 为“user-service”的客户端，.authorizedGrantTypes("refresh_token", "password")方法配置类验证类型为 refresh_token 和 password，.scopes("service ")方法配置了客户端域为“service”，.accessTokenValiditySeconds(3600)方法配置了 Token 的过期时间为 3600 秒。

AuthorizationServerEndpointsConfigurer 配置了 tokenStore 和 authenticationManager。其中 tokenStore 使用 JwtTokenStore，JwtTokenStore 并没有做任何存储，tokenStore 需要一个 JwtAccessTokenConverter 对象，该对象用于 Token 转换。本案例中使用了非对称性加密 RSA 对 JWT 进行加密。

authenticationManager 需要配置 AuthenticationManager 这个 Bean，这个 Bean 来源于 WebSecurityConfigurerAdapter 的配置，只有配置了这个 Bean 才会开启密码类型的验证。

5. 生成 jks 文件

在 AuthorizationServerEndpointsConfigurer 的配置中，配置 JwtTokenStore 时需要使用 jks 文件作为 Token 加密的秘钥。那么 jks 文件是怎样生成的呢？在本案例中，jks 文件是使用 Java keytool 生成的，在生成 jks 文件之前需要保证 Jdk 已经安装。打开计算机终端，输入以下命令：

```
keytool -genkeypair -alias fzp-jwt -validity 3650 -keyalg RSA -dname "CN=jwt,OU=jtw,O
=jtw,L=zurich,S=zurich,C=CH" -keypass fzp123 -keystore fzp-jwt.jks -storepass fzp123
```

在上面的命令中，-alias 选项为别名，-keypass 和-storepass 为密码选项，-validity 为配置 jks 文件的过期时间（单位：天）。

获取的 jks 文件作为私钥，只允许 Uaa 服务持有，并用作加密 JWT。那么 user-serive 这样

的资源服务，是如何解密 JWT 的呢？这时就需要使用 jks 文件的公钥。获取 jks 文件的公钥命令如下：

```
keytool -list -rfc --keystore fzp-jwt.jks | openssl x509 -inform pem -pubkey
```

在计算机终端输入上面的命令，提示需要密码，本例的密码为“fzp123”，输入即可，显示的公钥信息如下：

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAgRvxhIKCnrjIoT3mxfqd
hx+Dq8bgVVlSdfjVopD5a05FDuqpKrV3IQNAkntzmz5UUzd4VJf8MBIwk+F0aJYq
J89nEiBrrSrJxcOuZ1yFnvKh4VGXJwU4uGnf8kCvxmpZ5eegzUa+EI1qINm6ariV
jjQk8es0VglbvT6IoIyPtEv8flad5iHtkSUgVq43Gbo0oexdi/nxV+gXX81wiYJ4
c5/mom0ehV3fl9/evLMECe7E6T3t5WTcuD1TzteA0jr5PFVdXHDnyy6BsXT72mnB
90YxY2LEGPrU5cJcruiJ6UpXQFVtkPh5yZu0HWbFQ4dEMDxAsKMiXBC0LFdqd86f
8wIDAQAB
-----END PUBLIC KEY-----
```

新建一个 public.cert 文件，将上面的公钥信息复制到 public.cert 文件中并保存。并将 public.cert 文件放在资源服务的工程的 Resource 目录下。到目前为止，Uaa 授权服务已经搭建完毕。

需要注意的是，Maven 在项目编译时，可能会将 jks 文件编译，导致 jks 文件乱码，最后不可用。需要在工程的 pom 文件中添加以下内容：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-resources-plugin</artifactId>
        <configuration>
            <nonFilteredFileExtensions>
                <nonFilteredFileExtension>cert</nonFiltere dFileExtension>
                <nonFilteredFileExtension>jks</nonFilteredFileExtension>
            </nonFilteredFileExtensions>
         </configuration>
</plugin>
```

15.2.5 编写 user-service 资源服务

1. 依赖管理 pom 文件

user-service 工程的 pom 文件继承了主 Maven 工程的 pom 文件。在 user-service 工程的 pom 文件中引入 Web 功能的起步依赖 spring-boot-starter-web、OAuth2 的起步依赖 spring-cloud-starter-oauth2、数据库连接依赖 mysql-connector-java、JPA 的起步依赖 spring-boot-starter-data-jpa、Eureka 的起步依赖 spring-cloud-starter-eureka 和声明式调用 Feign 和 Hystrix 的起步依赖。user-service 工程的 pom 文件代码如下：

```
<parent>
         <groupId>com.forezp</groupId>
         <artifactId>cloud-oauth2-jwt</artifactId>
         <version>1.0-SNAPSHOT</version>
</parent>
<dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-oauth2</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>

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

        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-hystrix</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-feign</artifactId>
        </dependency>
</dependencies>
```

2. 配置文件 application.yml

在工程的配置文件 application.yml 中，配置程序名为 user-service，端口号为 9090，服务的注册地址为 http://localhost:8761/eureka/，以及连接的数据库的地址、用户名、密码和 JPA 的相关配置。另外，需要配置 feign.hystrix.enable 为 true，即开启 Feign 的 Hystrix 功能。完整的配置代码如下：

```
server.port: 9090

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
spring:
  application:
    name: user-service
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/spring-cloud-auth?useUnicode=true&characterEncod
ing=utf8&characterSetResults=utf8
    username: root
    password: 123456
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
feign:
  hystrix:
   enabled: true
```

3. 配置 Resource Server

在配置 Resource Server 之前，需要注入 JwtTokenStore 类型的 Bean。建一个 JwtConfig 类，加上 @Configuration 注解，开启配置文件的功能。JwtTokenStore 类型的 Bean 需要配置一个 JwtAccessTokenConverter 类型的 Bean，该 Bean 用作 JWT 转换器。JwtAccessTokenConverter 需要设置 VerifierKey，VerifierKey 为公钥，存放在 Resource 目录下的 public.cert 文件中。JwtConfig 类的代码如下：

```
@Configuration
public class JwtConfig {
   @Autowired
   JwtAccessTokenConverter jwtAccessTokenConverter;

   @Bean
   @Qualifier("tokenStore")
   public TokenStore tokenStore() {
      return new JwtTokenStore(jwtAccessTokenConverter);
   }

   @Bean
   protected JwtAccessTokenConverter jwtTokenEnhancer() {
      JwtAccessTokenConverter converter =  new JwtAccessTokenConverter();
      Resource resource = new ClassPathResource("public.cert");
```

```
        String publicKey ;
        try {
           publicKey = new String(FileCopyUtils.copyToByteArray(resource.getInputStream()));
        } catch (IOException e) {
           throw new RuntimeException(e);
        }
        converter.setVerifierKey(publicKey);
        return converter;
    }
}
```

然后配置 Resource Server，新建一个 ResourceServerConfig 的类，该类继承了 ResourceServerConfigurerAdapter 类，在 ResourceServerConfig 类上加@EnableResourceServer 注解，开启 Resource Server 功能。作为 Resource Server，需要配置 HttpSecurity 和 ResourceServerSecurityConfigurer 这两个选项。HttpSecurity 配置了哪些请求需要验证，哪些请求不需要验证。在本案例中，“ /user/login”（登录）和“/user/register”（注册）两个 API 接口不需要验证，其他请求都需要验证。ResourceServerSecurityConfigurer 需要配置 tokenStore，tokenStore 为之前注入 IoC 容器中的 tokenStore。代码如下：

```
@Configuration
@EnableResourceServer
public class ResourceServerConfig extends ResourceServerConfigurerAdapter{
   @Autowired
   TokenStore tokenStore;
   @Override
   public void configure(HttpSecurity http) throws Exception {
      http
             .csrf().disable()
             .authorizeRequests()
             .antMatchers("/user/login","/user/register").permitAll()
             .antMatchers("/**").authenticated();
   }
   @Override
   public void configure(ResourceServerSecurityConfigurer resources) throws Exception {
      resources.tokenStore(tokenStore);
   }
}
```

4. 配置 Spring Security

新建一个配置类 GlobalMethodSecurityConfig，在此类中通过@EnableGlobalMethodSecurity (prePostEnabled = true)注解开启方法级别的安全验证。代码如下：

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
```

```
public class GlobalMethodSecurityConfig {
}
```

5. 编写用户注册接口

这里用到了 User 和 Role 两个实体类，这两个类的代码和 11.3.4 节是一样的，在此不再重复。Dao 层的 UserDao 类继承了 JpaRepository 类，并有一个根据用户名获取用户的方法，代码如下：

```
public interface UserDao extends JpaRepository<User, Long> {
   User findByUsername(String username);
}
```

Service 层的 UserService 写一个插入用户的方法，代码如下：

```
@Service
public class UserServiceDetail {
   @Autowired
   private UserDao userRepository;
   public User insertUser(String username,String  password){
     User user=new User();
     user.setUsername(username);
     user.setPassword(BPwdEncoderUtil.BCryptPassword(password));
     return userRepository.save(user);
   }
}
```

在 UserServiceDetail 类中使用到了工具类 BPwdEncoderUtil，其中 BCryptPasswordEncoder 是 Spring Security 的加密类，BPwdEncoderUtil 类的代码如下：

```
public class BPwdEncoderUtil {
   private static final BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
   public static String  BCryptPassword(String password){
      return encoder.encode(password);
   }
   public static boolean matches(CharSequence rawPassword, String encodedPassword){
      return encoder.matches(rawPassword,encodedPassword);
   }
}
```

在 Web 层，在 UserController 中写一个注册的 API 接口“/user/register”，代码如下：

```
@RestController
@RequestMapping("/user")
public class UserController {
   @Autowired
   UserServiceDetail userServiceDetail;
```

```
   @PostMapping("/register")
   public User postUser(@RequestParam("username") String username ,@RequestParam("pas
sword") String password){
       //参数判断，省略
      return userServiceDetail.insertUser(username,password);
   }
```

启动所有的工程，使用 Curl 注册一个账号，代码如下：

```
curl -d "username=miyaa&password=123456" "localhost:9090/user/register"
```

返回结果为：

```
{"id":6,"username":"miyya","password":"$2a$10$d.ETlomhatNDxO4Olhx9C.qa6dviEEVeAZ9RsUH
bqYWp4jnPCdVYK","authorities":null,"enabled":true,"accountNonExpired":true,"accountNo
nLocked":true,"credentialsNonExpired":true}
```

6. 编写用户登录接口

在 Service 层中，在 UserServiceDetail 中添加一个 login（登录）方法，代码如下：

```
@Service
public class UserServiceDetail {
   @Autowired
   private UserDao userRepository;
   @Autowired
AuthServiceClient client;
   public UserLoginDTO login(String username,String password){
      User user=userRepository.findByUsername(username);
      if (null == user) {
        throw new UserLoginException("error username");
      }
      if(!BPwdEncoderUtil.matches(password,user.getPassword())){
         throw new UserLoginException("error password");
      }
          JWT jwt=client.getToken("Basic dXNlci1zZXJ2aWNlOjEyMzQ1Ng== ","password",us
ername,password);
         if(jwt==null){
         throw new UserLoginException("error internal");
      }
      UserLoginDTO userLoginDTO=new UserLoginDTO();
      userLoginDTO.setJwt(jwt);
      userLoginDTO.setUser(user);
      return userLoginDTO;
   }
}
```

其中，AuthServiceClient 为 Feign 的客户端，所以需要程序的启动类 UserServiceApplication

通过@EnableFeignClients 开启 Feign 客户端的功能。代码如下：

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

AuthServiceClient 通过向 uaa-service 服务远程调用“/oauth/token”API 接口，获取 JWT。在“/oauth/token”API 接口中需要在请求头传入 Authorization 信息，并需要传请求参数认证类型 grant_type、用户名 username 和密码 password，代码如下：

```
@FeignClient(value = "uaa-service",fallback =AuthServiceHystrix.class )
public interface AuthServiceClient {

   @PostMapping(value = "/oauth/token")
   JWT getToken(@RequestHeader(value = "Authorization") String authorization, @Reques
tParam("grant_type") String type,
            @RequestParam("username") String username, @RequestParam("password") Str
ing password);

}
```

其中，AuthServiceHystrix 为 AuthServiceClient 的熔断器，代码如下：

```
@Component
public class AuthServiceHystrix implements AuthServiceClient {
   @Override
   public JWT getToken(String authorization, String type, String username, String pas
sword) {
      return null;
   }
}
```

JWT 为一个 JavaBean，它包含了 access_token、token_type 和 refresh_token 等信息，代码如下：

```
public class JWT {
   private String access_token;
   private String token_type;
   private String refresh_token;
   private int expires_in;
   private String scope;
   private String jti;
   ...省略getter setter
```

```
}
```

UserLoginDTO 包含了一个 User 和一个 JWT 对象，用于返回数据的实体：

```
public class UserLoginDTO {
    private JWT jwt;
    private User user;
   ...省略 getter setter
}
```

登录异常类 UserLoginException，继承自 RuntimeException，定义了一个构造方法，代码如下：

```
public class UserLoginException extends RuntimeException{
   public UserLoginException(String message) {
      super(message);
   }
}
```

异常统一处理类为 ExceptionHandle 类，在该类中加上@ControllerAdvice 注解表明该类是一个异常统一处理类。通过@ExceptionHandler 注解配置了统一处理 UserLoginException 类的异常方法，统一返回了异常的 message 信息，代码如下：

```
@ControllerAdvice
@ResponseBody
public class ExceptionHandle {
   @ExceptionHandler(UserLoginException.class)
   public ResponseEntity<String> handleException(Exception e) {
      return new ResponseEntity(e.getMessage(), HttpStatus.OK);
   }
}
```

在 Web 层的 UserController 类写一个登录的 API 接口“/user/login”，代码如下：

```
@RestController
@RequestMapping("/user")
public class UserController {
   @Autowired
   UserServiceDetail userServiceDetail;
   @PostMapping("/login")
   public UserLoginDTO login(@RequestParam("username") String username ,   @RequestPa
ram("password") String password){
      //参数判断，省略
      return userServiceDetail.login(username,password);
   }
}
```

在“/user/login”API 接口中，需要的请求参数为用户名和密码。首先会根据用户名查询数据库，获取用户，如果用户存在，判断密码是否正确。如果密码正确，通过 Feign 客户端远程调用 uaa-service，获取 JWT，获取成功，将用户和 JWT 封装成 UserLoginDTO 对象返回。现在使用 Curl 调用登录 API 接口，执行命令如下：

```
curl user-service:123456@localhost:9999/oauth/token -d grant_type=password -d username=miyaa
-d password=123456
```

命令执行成功后，返回了 User 信息和 JWT 的信息，由于 JWT 信息太长，在这里就不展示了。

7. 测试

编写一个“/foo”的 API 接口，该 API 接口需要“Role_ADMIN”权限才能访问，代码如下：

```
@RestController
@RequestMapping("/foo")
public class WebController {
   @RequestMapping(method = RequestMethod.GET)
   @PreAuthorize("hasAuthority('ROLE_ADMIN')")
   public String getFoo() {
      return "i'm foo, " + UUID.randomUUID().toString();
 }
}
```

以 username 为“miyaa”，密码为“123456”登录，登录成功后返回了 JWT 对象，JWT 中有一个 access_token 的字符串。将该 Token 放在请求头重中，进行请求，代码如下：

```
curl -l -H "Authorization:Bearer {access_token} " -X GET "localhost:9090/foo"
```

返回结果如下：

```
{"error":"access_denied","error_description":"不允许访问"}
```

从上面的返回信息可知，该用户没有权限访问该 API 接口。这是正常的，因为新注册的“miyaa”这个用户并没有“Role_ADMIN”权限。为了方便演示，现在给“miyaa”这个用户赋予“Role_ADMIN”的权限，直接在数据库中插入以下数据，数据库脚本如下：

```
INSERT INTO 'role' VALUES ('1', 'ROLE_USER'), ('2', 'ROLE_ADMIN');
INSERT INTO 'user_role' VALUES('6', '2');
```

插入数据后，重新登录并获取 access_token，重新请求“/foo”API 接口，返回结果如下：

```
i'm foo, 84db63b5-bdd1-4326-855c-5d19ce0d5b89
```

可见，当给“miyaa”这个用户赋予“Role_ADMIN”权限之后，该用户具有访问“/foo”API 接口的权限。

15.3 总结

在本案例中，用户通过登录接口来获取授权服务的 Token。用户获取 Token 成功后，在以后每次访问资源服务的请求中都需要携带该 Token。资源服务通过公钥解密 Token，解密成功后可以获取用户信息和权限信息，从而判断该 Token 所对应的用户是谁，具有什么权限。

这个架构的优点在于，一次获取 Token，多次使用，不再每次询问 Uaa 服务该 Token 所对应的用户信息和用户的权限信息。这个架构也有缺点，例如一旦用户的权限发生了改变，该 Token 中存储的权限信息并没有改变，需要重新登录获取新的 Token。就算重新获取了 Token，如果原来的 Token 没有过期，仍然是可以使用的，所以需要根据具体的业务场景来设置 Token 的过期时间。一种改进方式是将登录成功后获取的 Token 缓存在网关上，如果用户的权限更改，将网关上缓存的 Token 删除。当请求经过网关，判断请求的 Token 在缓存中是否存在，如果缓存中不存在该 Token，则提示用户重新登录。

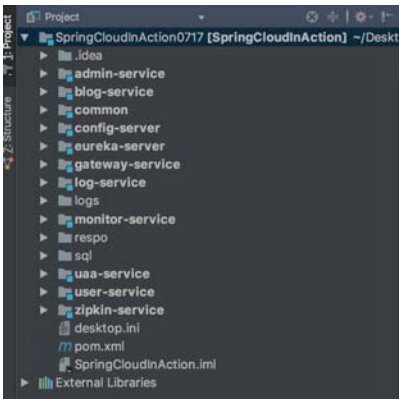
第 16 章　使用 Spring Cloud 构建微服务综合案例

本章利用一个使用 Spring Cloud 构建微服务的综合案例对前面所有章节的内容进行整合和总结，这个案例也是我在实际工作中对 Spring Cloud 构建微服务内容的一个提炼。希望通过本章内容，为读者提供一整套使用 Spring Cloud 构建微服务的解决方案。

16.1 案例介绍

16.1.1　工程结构

本章采用 Maven 多 Module 工程的结构，一共有 11 个 Module 工程。其中有 10 个 Module 工程为微服务工程，这 10 个微服务工程构成了一个完整的微服务系统。微服务系统包含 8 个微服务系统的基础服务，提供了一整套微服务治理的功能，它们分别是配置中心 config-server、注册中心 eureka-server、授权中心 Uaa 服务 uaa-service、Turbine 聚合监控服务 monitoring-service、链路追踪服务 zipkin-service、聚合监控服务 admin-service、路由网关服务 gateway-service、日志服务 log-service。另外还包含了两个资源服务 user-service 和 blog-service，对外暴露 API 接口。除此之外，还有一个 common 的 Module 工程，为资源服务提供基本的工具类。最后，在工程的目录下还有 3 个文件夹，sql 文件夹存放项目的 sql 文件，logs 文件夹存放项目的工程日志，respo 文件夹目录存放项目的配置文件。完整的项目结构如图 16-1 所示。



▲图 16-1　完整的项目结构

16.1.2　使用的技术栈

本案例使用到的技术栈如表 16-1 所示。其中，Spring Cloud Netflix 包括 Eureka、Hystrix、Hystrix Dashboard、Turbine、Ribbon 和 Zuul 组件，微服务系统提供了基本的服务治理功能。Spring Cloud OAuth2 包括 Spring OAuth2 和 Spring Boot Security，为微服务系统提供了一套安全解决方案。Spring Cloud Sleuth 为微服务下系统提供了链路追踪的能力。Turbine 聚合了 Hystrix Dashboard，为微服务系统服务的熔断器提供了监控。Spring Boot Admin 是一个非常优秀的监控组件，为微服务提供一系列的监控能力。Feign 声明式调用组件，为微服务提供远程调度功能。本项目采用的数据库为 MySQL 数据库，采用的 ORM 框架为 Spring Data JPA。本项目的 API 接口文档采用了 Swagger2 框架生成在线 API 接口文档，API 接口采用 RESTful 风格。

表 16-1　　案例中使用到的技术栈

使用的技术栈	技术栈说明
Eureka	服务注册和发现
Spring Cloud Config	分布式服务配置中心
Spring Cloud OAuth2	包括 Spring OAuth2 和 Spring Boot Security，为微服务提供一整套的安全解决方案
Feign	声明式服务调用，用于消费服务
Ribbon	负载均衡
Hystrix	熔断器
Hystrix Dashboard	熔断器仪表盘，用于监控熔断器的状况
Turbine	聚合多个 Hystrix Dashboard
Spring Cloud Sleuth	集成 Zipkin，用于服务链路追踪
Spring Boot Admin	聚合监控微服务的状况
Zuul	服务网关，用于服务智能路由、负载均衡
Spring Data JPA	数据库采用 MySQL，实体对象持久化采用 JPA
Swagger	API 接口文档组件
RESTful API	本案例的接口采用 RESTful 风格
RabbitMQ	消息服务器，用于发送日志消息

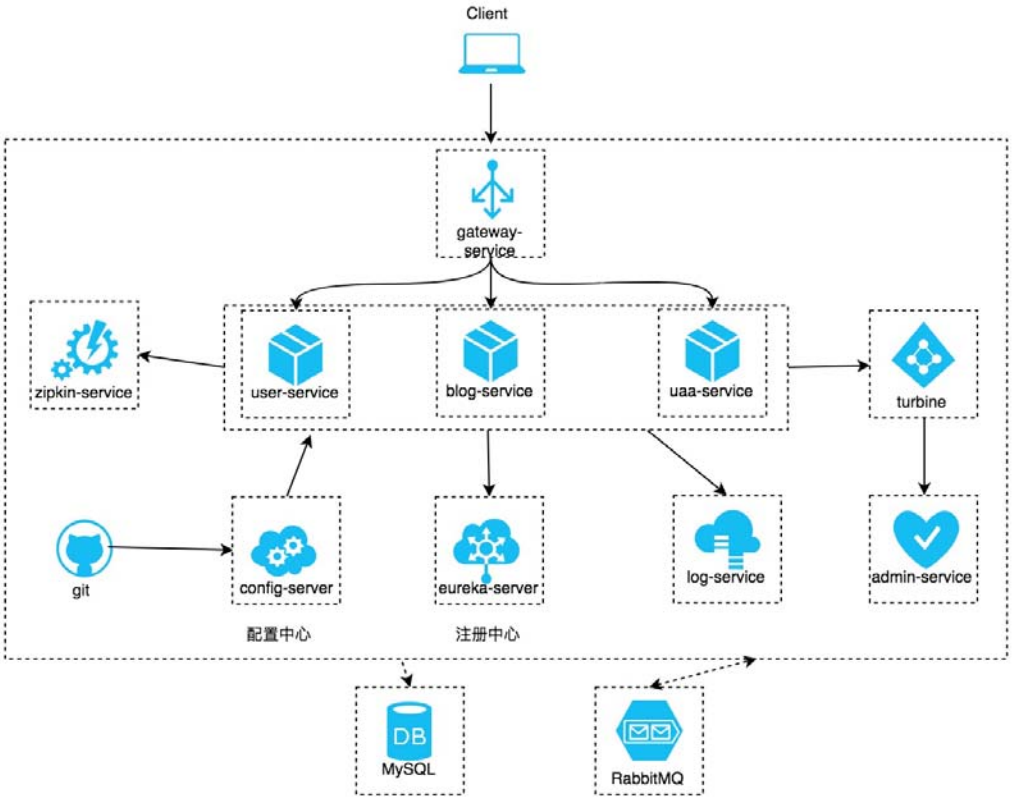
16.1.3　工程架构

本案例一共有 10 个微服务，每个服务各司其职，相互协作，构成了一个完整的微服务系统，工程的架构图如图 16-2 所示。

在这个系统中，所有的服务都向服务注册中心 eureka-server 进行服务注册。eureka-server 作为服务注册中心主要有两个好处，一是所有的服务都向服务注册中心注册，能够方便查看每

个服务的状况、服务是否可用，以及每个服务有哪些服务实例；二是服务注册中心维护了一份服务注册列表，每个服务实例都能够获取服务注册列表，获取的注册列表可用于 Ribbon 的负载均衡，也可以用于 Zuul 的智能路由。

config-server 作为配置中心，所有服务的配置文件由 config-server 统一管理，config-server 可以从远程 Git 仓库读取，也可以从本地仓库读取。如果将配置文件放在远程仓库，配合 Spring Cloud Bus，可以在不人工重启服务的情况下，进行全局服务的配置刷新。



▲图 16-2 工程架构图

gateway-service 为网关服务，使用的是 Zuul 组件，Zuul 组件可以实现智能路由、负载均衡的功能。gateway-service 作为一个边界服务，对外统一暴露 API 接口，其他的服务 API 接口只提供给内部服务调用，不提供给外界直接调用，这就很方便实现统一鉴权、安全验证的功能。有些读者可能会有疑惑，为什么在自己的电脑上运行微服务系统时可以直接访问内部微服务的 API 接口呢？这需要设置网络环境，例如配置防火墙，只允许外部请求访问微服务系统的 gateway-service 服务端口，其他的端口不对外开放。另外，可以通过 Docker 部署微服务，由于 Docker 采用沙盒原理，可以实现只允许内部服务调用，同时只暴露 gateway-service 服务的端口给外部调用。

zipkin-service 是 Spring Cloud Sluth 的组件，它可以查看每个请求在微服务系统中的链路关系。

turbine-service 聚合了 user-serice 和 blog-service 的 Hystrix Dashboard，可以查看这两个服务熔断器的监控状况。

admin-service 是一个 Spring Boot Admin Server 工程，提供了非常强大的服务监控功能，可以查看每个向 eureka-server 注册的服务的健康状态、日志、注册时间线等，并能够集成 Turbine。

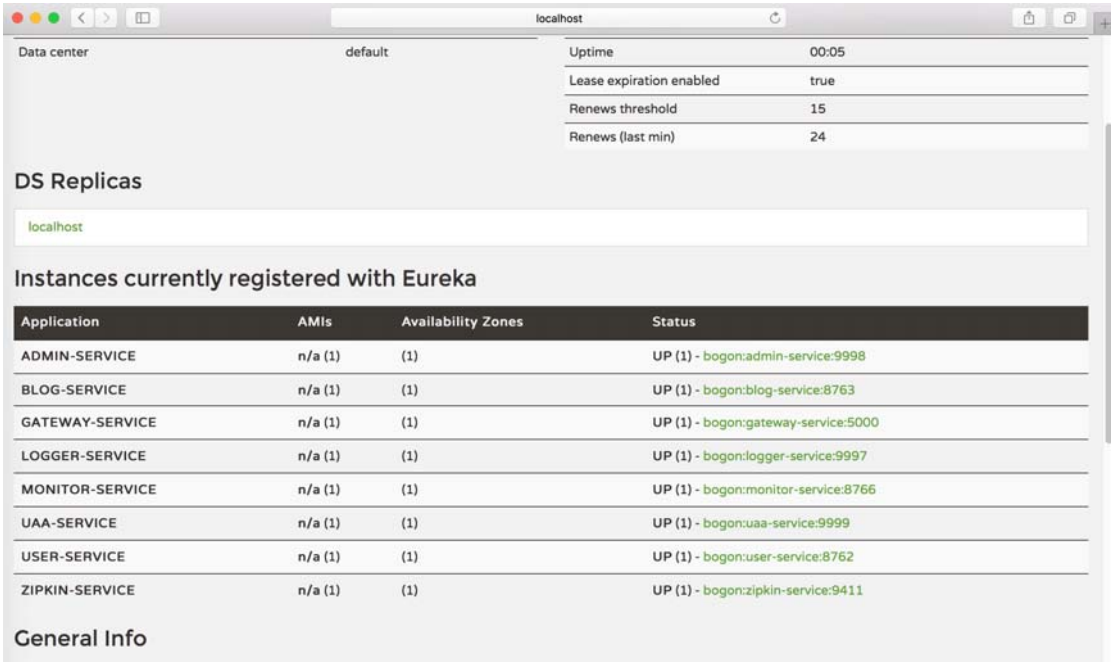
uaa-service 集成了 Spring Cloud OAuth2，由这个服务统一授权，并返回 Token。其他的应用服务，例如 user-service 和 blog-service 作为资源服务，它们的 API 接口资源是受保护的，需要验证 Token，并鉴权后才能访问。

user-service 和 blog-service 作为资源服务，对外暴露 API 接口资源。

log-service 为日志服务，user-service 和 blog-service 服务通过 RabbitMQ 向 log-service 发送业务操作日志的消息，日志服务统一持久化操作日志。该日志服务只持久化资源操作的日志，如 API 接口的请求。如果有大量的日志需处理，可以使用 ELK 组件进行处理。

16.1.4　功能展示

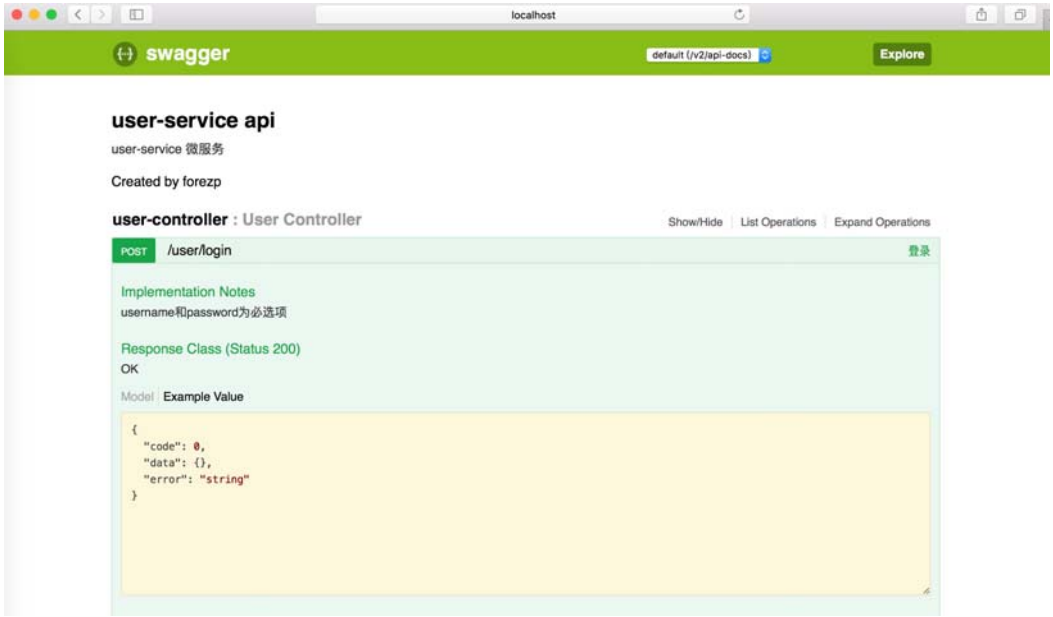
依次启动 eureka-server、config-server、zipkin-service 以及其他的微服务。等整个微服务系统完全启动之后，在浏览器上访问 http://localhost:8761，即 Eureka 的主页，可以查看服务注册的情况。除了 config-server 没有向 eureka-server 注册外，其他服务都已经注册成功，并且是可用状态（Status Up），如图 16-3 所示。



▲图 16-3　Eureka 主页

整个系统的 API 接口采用 RESTful 风格。REST 全称是 Representational State Transfer（表述状态转移），由 Roy Fielding 在其博士论文中首次提出。REST 本身没有创造新的技术、组件或服务，REST 的理念就是在现有的技术之上，更好地使用现有的 Web 规范。REST 能够很好地展现资源，每个资源都由 URI/ID 唯一标识，根据唯一标识，客户端可以更好地使用资源。

API 接口文档采用 Swagger2 框架生成在线文档。user-serivice 工程和 blog-service 工程中集成了 Swagger2，集成 Swagger2 只需要引入依赖，并做相关的配置，然后在具体的 Controller 上写注解，就可以实现 Swagger2 的在线文档功能。访问 user-serivice 的 Swagger2 主页 http://localhost:8762/swagger-ui.html，如图 16-4 所示，该界面显示了 user-service 的 UserController 的“/user/login”接口。

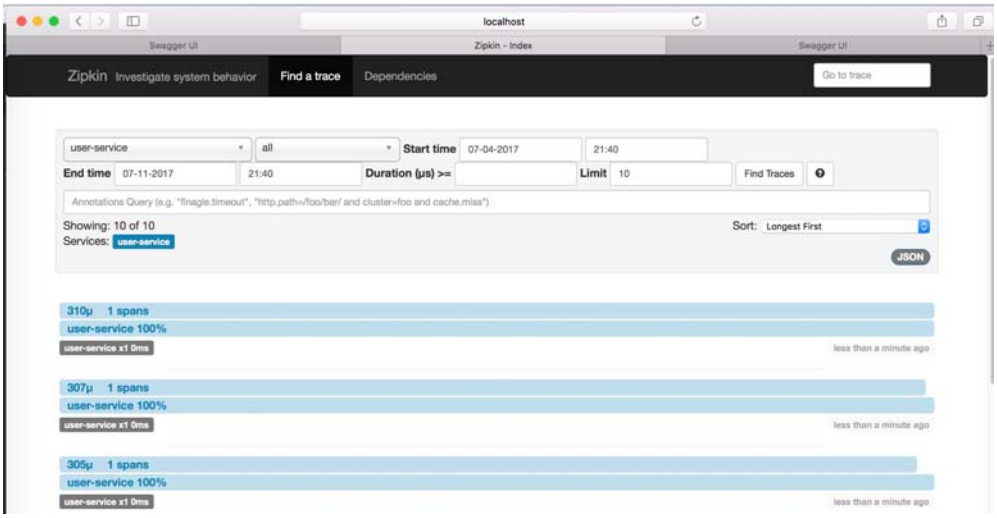


▲图 16-4 user-service 的 Swagger2 主页

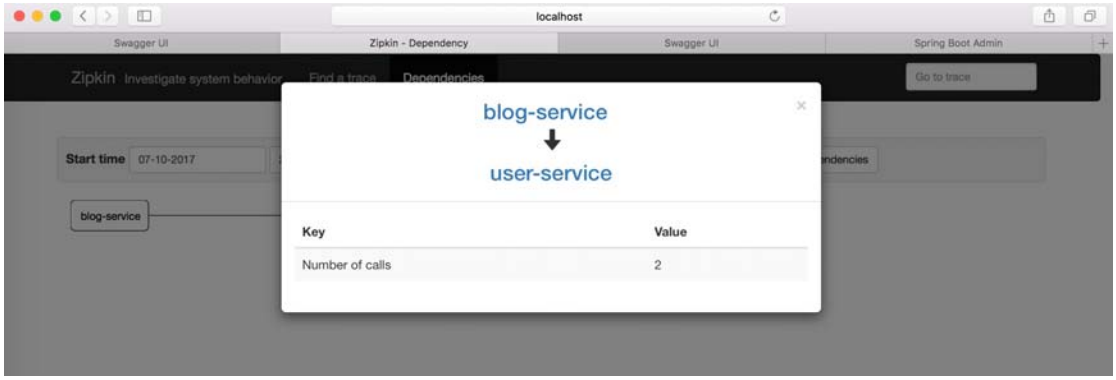
访问 zipkin-service 的主页面 http://localhost:9411/，界面显示如图 16-5 所示。zipkin-service 工程集成了 Spring Cloud Sleuth 组件，该页面显示了服务的链路调用情况。可以根据不同的条件去搜索服务的链路调用情况，例如服务名、请求时长、调用时间等。

图 16-6 展示了服务的依赖情况。Spring Cloud Sleuth 通过服务的相互调用情况来判断服务的依赖情况。在服务数量众多的情况下，服务的依赖比较复杂，通过这个界面可以很清楚地查看服务的依赖情况。例如在本系统中，blog-service 服务依赖了 user-service 服务。

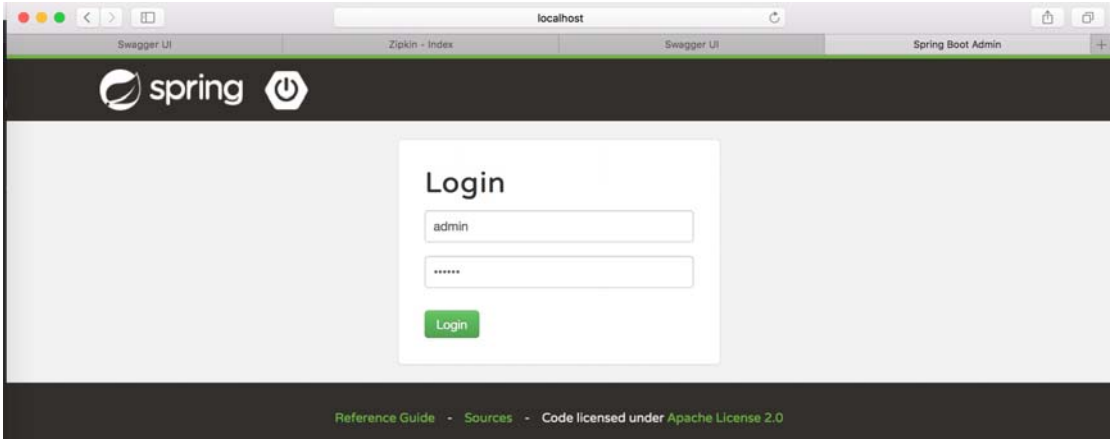
在浏览器上访问 http://localhost:9998，展示了 admin-service 的登录界面，如图 16-7 所示。admin-service 作为一个综合监控的服务，需要对访问者进行身份认证才能访问它的主页，本案例的登录用户名为 admin，密码为 123456。



▲图 16-5　zipkin-service 主页

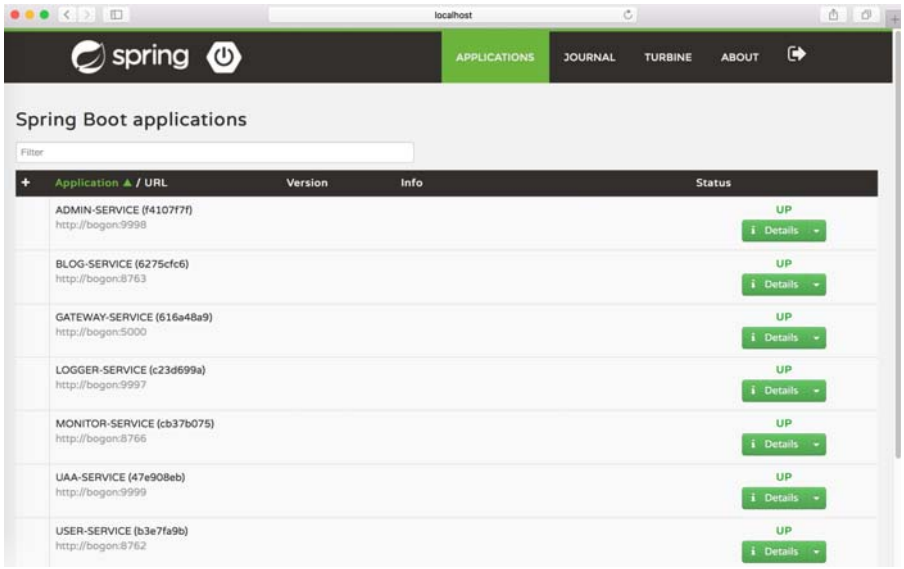


▲图 16-6　Zipkin-service 的服务依赖界面

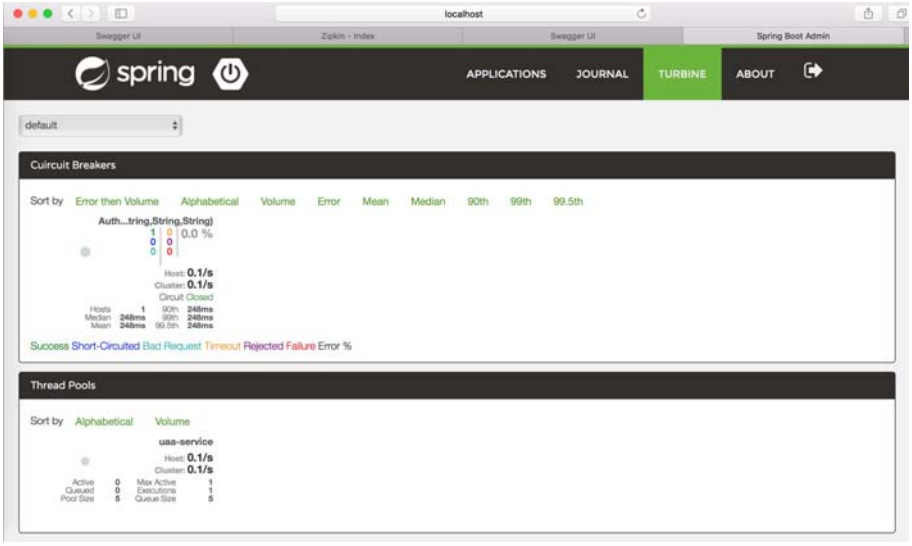


▲图 16-7　admin-service 的登录界面

admin-service 服务通过向 eureka-server 注册，获取服务注册中心 eureka-server 的所有服务注册列表。获取到服务注册列表后，会请求服务列中服务的 Actuator 模块所暴露的 API 接口，从而对每一个服务实例的健康状态进行实时监控。如图 16-8 所示，展示了每个向 eureka-server 注册的服务的健康状态。单击服务实例右边的“Details”按钮，可以获取更多的服务实例的健康信息，包括服务实例的注册的时间线、JMX、日志等。admin-service 很容易集成 Turbine 的功能，在本案例中，Turbine 聚合监控了 user-service 和 blog-service 的 Hystrix Dashboard，如图 16-9 所示。



▲图 16-8　服务的注册状态



▲图 16-9　admin-service 集成了 Turbine

16.2 案例详解

本案例是 Spring Cloud 构建微服务的综合样板案例，读者可以通过学习本案例来全面了解 Spring Cloud。本案例是作者在实际工作的一个总结，对实际开发有较高的参考价值，可以直接拿来应用，进行项目的开发。本案例的代码量较大，所以不会对所有的代码进行讲解，一是篇幅有限，二是很多代码与之前章节中的代码重复。读者可以下载源码，一边看源码，一边看本章内容来理解。有关源码下载，请到本书前言中关注我的公众号获取。

16.2.1 准备工作

本案例一共有 10 个微服务，运行本案例对计算机的硬件要求需要运行内存达 8GB 的 i5 处理器。本案例采用 IDEA 作为开发工具，Maven 的版本为 3.x，JDK 版本为 1.8。另外使用到了 MySQL 数据库、RabbitMQ 消息组件，读者需要提前安装好。

本章的案例跟之前的章节一样，采用 Maven 多 Module 的形式。在主 Maven 工程的 pom 文件引入 Spring Boot 的版本是 1.5.3、Spring Cloud 版本为 Dalston.RELEASE，并指定整个项目的编码 UTF-8，以及一些公共的依赖。其余 Module 工程的 pom 文件继承主 Maven 工程的 pom 文件，不用再引入主 Maven 工程的 pom 文件已经存在的依赖和配置。这样做的好处就是方便统一管理整个项目的依赖和配置，例如配置了整个项目的 Spring Boot 和 Spring Cloud 的版本。

16.2.2 构建主 Maven 工程

新建一个 Maven 工程，作为主 Maven 工程，其 pom 文件的 packaging 元素为 pom，在默认情况下，该元素为 jar。主 Maven 工程编译后不会生成任何的 Jar 包。除此之外，在 pom 文件中指定了项目的 Spring Boot 版本为 1.5.3，Spring Cloud 版本为 Dalston.RELEASE，项目的编码为 UTF-8。另外在 pom 文件中加上了整个项目的公共依赖、Spring Boot 测试的起步依赖 spring-boot-starter-test 和 Jolokia 的依赖 jolokia-core。其中，Jolokia 是一个利用 JSON 通过 Http 实现 JMX 远程管理的开源组件。在主 Maven 工程下有 11 个 Module 工程，完整的 pom 文件的代码清单如下所示：

```
<?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.forezp</groupId>
   <artifactId>SpringCloudInAction</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>pom</packaging>
```

```
    <modules>
      //...代码省略
    </modules>
    <parent>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-parent</artifactId>
       <version>1.5.3.RELEASE</version>
       <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <properties>
       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
       <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
       <java.version>1.8</java.version>
       <spring-cloud.version>Dalston.RELEASE</spring-cloud.version>
    </properties>
    <dependencies>
       <dependency>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-test</artifactId>
           <scope>test</scope>
       </dependency>
       <dependency>
          <groupId>org.jolokia</groupId>
          <artifactId>jolokia-core</artifactId>
       </dependency>
    </dependencies>
    <dependencyManagement>
       <dependencies>
           <dependency>
              <groupId>org.springframework.cloud</groupId>
              <artifactId>spring-cloud-dependencies</artifactId>
              <version>${spring-cloud.version}</version>
              <type>pom</type>
              <scope>import</scope>
           </dependency>
       </dependencies>
    </dependencyManagement>
</project>
```

16.2.3 构建 eureka-server 工程

eureka-server 作为服务注册中心，在 5.2 节已经详细讲解过，主要分为以下 3 个步骤。

（1）其 pom 文件继承了主 Maven 的 pom 文件，并且引入 Eureka Server 的起步依赖 spring-cloud-starter-eureka-server 、Web 的起步依赖 spring-boot-starter-web。

（2）在其配置文件 application.ym 指定 eureka-server 的端口号为 8761，并不自注册，即配置 eureka.client.register-with-eureka 和 eureka.client.fetch-registry 为 false。配置文件 application.ym

代码如下：

```
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
    serviceUrl:
      defaultZone: http://localhost:${server.port}/eureka/
```

（3）在程序的启动类 EurekaServerApplication 加上@EnableEurekaServer 注解，开启 EurekaServer 的功能。

经过上述 3 步，eureka-server 就搭建好了。

Eureka Client 向 eureka-server 注册，也同样需要 3 步。一是在工程的依赖管理文件 pom 引入 spring-cloud-starter-eureka 的起步依赖，二是在配置文件 application.yml 配置服务注册中心的 Uri，三是在程序的入口类加上@EnableEurekaClient，开启 Eureka Client 的功能。

16.2.4　构建 config-server 工程

本案例的所有服务的配置文件都放在 config-server 工程中统一管理。config-server 可以从本地仓库读取配置文件，也可以从远程 Git 仓库读取，本案例从本地仓库读取。

实现过程也很简单，需要完成以下的 3 个步骤。

（1）工程的 pom 文件继承了主 Maven 的 pom 文件，然后引入起步依赖 spring-cloud-config-server。

（2）在工程的配置文件 application.ym 中配置程序的端口号为 8769，程序名为 config-server；配置 spring.cloud.profiles.active 为 native，指定 config-server 从本地仓库读取配置文件。spring.cloud.config.server.native.search-locations 为 classpath:/shared，即指定 config-server 从本地 Resources/shared 目录下读取配置文件，这时就可以将其他服务的配置文件放在 Resources/shared 的目录下。

（3）在程序的启动 Aplication 类上加上注解@EnableConfigServer，开启 ConfigServer 的功能。

需要注意的是，其他服务的配置文件的命名格式是{applicationName}-{activeProfile}.yml。例如，user-service 服务的 applicationName 为 user-service，它的 activeProfile 为 pro，那么在配置服务中的配置文件命名为 user-service-pro.yml 或者 user-service-pro.properties。另外，所有的服务可以共享一个公共的配置文件，在 Resources/shared 中创建一个 application.yml 配置文件，作为所有服务共享的配置文件。

作为 Config Client 从 config-server 读取配置文件，需要在工程的 pom 文件中引入 config 的起步依赖 spring-cloud-starter-config，并在 application.yml 做相关配置。现在以 user-service 服务为例来讲解，它的配置文件 application.yml 代码如下：

```
spring:
  application:
```

```
    name: user-service
  cloud:
    config:
      uri: http://localhost:8769
      fail-fast: true
  profiles:
    active: pro
```

该配置文件指定了程序名为 user-service，activeProfile 为 pro，向 Uri 为 http://localhost:8769 的 Config Server 读取服务的配置文件，配置文件名为 user-service-pro.yml。并且设置了 fail-fast 属性为 true，即如果读取配置文件不成功，实行快速失败的策略。

16.2.5 构建 zipkin-service 工程

在微服务系统中，服务的调用链路关系会很复杂。这时需要在项目中集成 Spring Cloud Sleuth，方便查看服务的链路调用关系。Spring Cloud Sleuth 很容易地集成 Zipkin，Zipkin 分为 Zipkin Server 和 Zipkin Client。首选讲解作为 Zipkin Server 的 zipkin-service 工程，它的实现过程需要如下两个步骤。

在 zipkin-service 工程的 pom 文件中加上 zipkin-server 和 zipkin-autoconfigure-ui 的依赖，代码如下：

```
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-server</artifactId>
</dependency>
<dependency>
   <groupId>io.zipkin.java</groupId>
   <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>
```

在程序的入口类 ZipkinServiceApplication 加上注解@EnableZipkinServer，开启 Zipkin Server 的功能。

在本案例中，user-service 和 blog-service 作为 Zipkin Client，那么如何实现一个 Zipkin Client 呢？只需要两步，一是在工程的 pom 文件中引入 zipkin 的起步依赖 pring-cloud-starter-zipkin，代码如下：

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

二是需要在配置文件 application.yml 中指定 Zipkin Server 的 Url，本案例的 Zipkin Server 的 Url 为 http://localhost:9411。

```
spring:
  zipkin:
    base-url: http://localhost:9411
```

16.2.6　构建 monitoring-service 工程

monitoring-service 工程集成了 Turbine 组件，用于聚合多个 Hystrix Dashboard 。Hystrix Dashboard 是监控 Hystrix 熔断器状况的组件。在本案例中，user-service 和 blog-service 集成了 Hystrix Dashboard，monitoring-service 的功能就是将这两个工程 Hystrix Dashboard 聚合在一起。

如何在 monitoring-service 工程中集成 Turbine 呢?

首先，需要在 monitoring-service 工程的 pom 文件中引入 Turbine 的起步依赖 spring-cloud-starter-turbine，代码如下：

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

然后需要在程序的入口类 MonitorServiceApplication 加上注解@EnableTurbine，开启 Turbine 的功能。

最后在工程的配置文件 application.yml（本案例的配置文件存放在 config-server 的 Resources/shared 目录下，本工程在 config-server 中对应的配置文件为 monitoring-service-pro，其他服务也类似，就不再说明）上做相关的配置。配置 turbine.aggregator.clusterConfig 为默认，即聚合监控的集群配置为 default。turbine.appConfig 选项配置了聚合 Hystrix Dashboard 的服务名，本案例中聚合了 user-service 和 account-service 服务。在 clusterConfig 为默认的情况下，turbine.clusterNameExpression 也填写默认的即可。配置文件 application.yml 的代码如下：

```
turbine:
  aggregator:
    clusterConfig: default
  appConfig: user-service , blog-service
  clusterNameExpression: new String("default")
```

下面来讲解如何在微服务中使用集成 Hystrix 和 Hystrix Dashboard。在本案例中，user-service 和 blog-service 集成了这两个组件，现以 user-service 为例来讲解。

Hystrix 提供了熔断器功能，主要用于服务与服务之间的调用，它有快速失败、服务降级的功能。某个服务出现故障时，Hystrix 能够让调用这个服务的其他服务快速失败，防止了线程阻塞导至线程资源耗尽情况的发生，也防止了微服务系统中的“雪崩”效应的产生。所以，Hystrix 在微服务系统中具有非常重要的作用。

Hystrix Dashboard 为监控 Hystrix 熔断器状况的组件，能够实时查看熔断器的状况，例如熔断器是否开启、关闭等。

本案例中使用的服务的调用框架为 Feign，Feign 默认集成了 Hystrix。在 Dalston 版本中，

Feign 的 Hystrix 是默认不开启的，需要在工程的配置文件 application.properties 加上 feign.hystrix.enable=true 的配置。

下面以具体代码来详细讲解。首先在工程的 pom 文件中引入依赖，包括 Feign、Hystrix-Dashboard 和 Hystrix 的起步依赖，代码如下：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

然后在程序的启动类 UserServiceApplication 加上@EnableFeignClients、@EnableHystrix-Dashboard 和@EnableHystrix，分别开启了 FeignClients、HystrixDashboard 和 Hystrix 的功能 。

最后需要在工程的配置文件 application.yml 中加上 Feign，开启 Hystrix 的配置，配置代码如下：

```
feign:
  hystrix:
    enabled: true
```

在 user-service 服务中需要调用 uaa-service 的服务来获取 JWT。在 AuthServiceClient 接口上通过@FeignClient 注解来声明一个 FeignClient，注解的 value 为服务名 （如本案例的 uaa-service），fallback 为熔断器的熔断处理类（如本案例的 AuthServiceHystrix 类）。代码如下：

```
@FeignClient(value = "uaa-service",fallback =AuthServiceHystrix.class )
public interface AuthServiceClient {

    @PostMapping(value = "/oauth/token")
    JWT getToken(@RequestHeader(value = "Authorization") String authorization, @Reque
stParam("grant_type") String type,
                 @RequestParam("username") String username, @RequestParam("password")
 String password);
}
```

AuthServiceHystrix 类需要实现 AuthServiceClient 接口，并且将该类以 Bean 的形式注入 Spring IoC 容器中，代码如下：

```
@Component
```

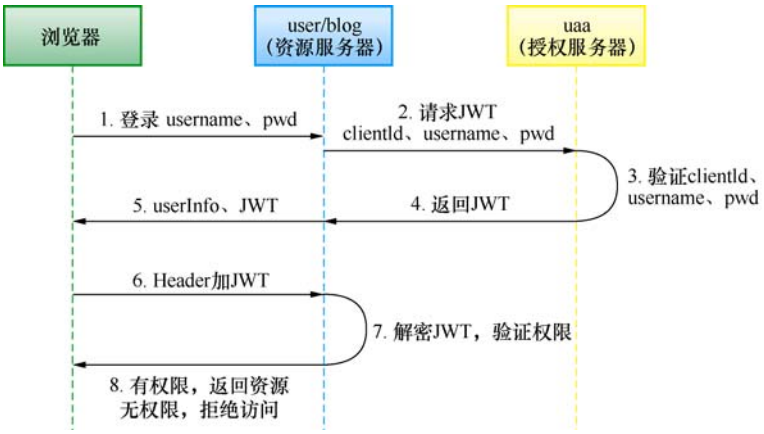
```
public class AuthServiceHystrix implements AuthServiceClient {
     @Override
     public JWT getToken(String authorization, String type, String username,
     String password) {
          System.out.println("--------opps getToken hystrix---------");
          return null;
     }
}
```

这样，一个具有 Feign 功能、熔断器功能和熔断器仪表盘（Hystrix Dashboard）功能的客户端就搭建好了。

16.2.7　构建 uaa-service 工程

在本案例中，采用 Spring Cloud OAuth2 来进行权限和安全的认证。关于如何使用 Spring Cloud OAuth2 进行鉴权和认证，在第 14 章和第 15 章中已经详细介绍过，这里不再重复。

案例中有 3 个角色，一是客户端，例如浏览器；二是资源服务器，例如 user-service 和 blog-service，它们的某些资源是被保护起来的，需要认证通过才能访问；三是授权服务，例如 uaa-service，它负责给用户颁发 JWT。案列中 uaa-service 模块的架构图和流程如图 16-10 所示。



▲图 16-10　uaa-service 模块的架构图

（1）首先，浏览器请求 user-service 的登录接口（登录接口不设置权限认证），将用户名和密码传给 user-service。user-service 根据用户名查询数据库获取用户信息，并进行密码的判断。

（2）用户密码判断准确无误后，通过 Feign 远程调用 uaa-service 获取 Token，需要传 clientId（和 uaa 服务设置一致的 clientId）、用户名、密码。

（3）uaa-service 接受到请求之后，会判断请求传的参数 clientId、用户名和密码的正确性。

（4）判断上一步请求参数准备无误后，uaa-service 根据配置的策略返回 JWT 给 user-service。JWT 是通过 RSA 加密的，它包含了用户名、权限信息和过期时间等。

（5）user-service 获取到 JWT 之后，连同用户信息一起返回给浏览器。

（6）当浏览器需要访问有权限认证的资源服务，需要在请求的 Header 加参数名“Authorization”和参数值为“Beare {Token}”。

（7）资源服务器通过用 SpringMVC 的拦截器对请求进行拦截，将 Header 中的 Token 取出，用公钥解密 Token，解密之后能够得到该 Token 包含的用户信息和权限信息，从而进行权限认证。

（8）当该 Token 对应的用户有权限访问该资源，资源服务器会返回该资源；若没有权限，则返回没有权限访问该资源。

在实际开发中，读者可能会遇到这样的需求，例如某个请求如何才能根据 Token 来获取当前的用户。这在第 15 章中没有讲述，特在此补充。在项目中，我封装了一个当前请求获取用户、权限、Token 的一个 UserUtils 类，部分代码如下：

```
public class UserUtils {
    /**
     * 获取当前请求的用户
     * @return
     */
    public static String getCurrentPrinciple() {
        return (String) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
    }

     /**
     * 获取当前请求的 token
     * @return
     */
    public static String getCurrentToken() {
        return HttpUtils.getHeaders(HttpUtils.getHttpServletRequest()).get(AUTHORIZATION);
    }

     /**
     * 获取当前请求的权限信息
     * @return
     */
    public static List<SimpleGrantedAuthority> getCurrentAuthorities() {
        return (List<SimpleGrantedAuthority>) SecurityContextHolder.getContext().getA
uthentication().getAuthorities();
    }

}
```

16.2.8 构建 gateway-service 工程

gateway-service 作为路由网关工程，集成了 Spring Cloud Zuul 组件。Spring Cloud Zuul 有路由转发、过滤、鉴权的功能。本案例只使用到了路由转发的功能。

在工程中实现 Zuul 的路由转发功能，需要以下 3 个步骤。

（1）在工程的 pom 文件引入 Zuul 的起步依赖 spring-cloud-starter-zuul，代码如下：

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

（2）在程序的启动类 GatewayServiceApplication 加上@EnableZuulProxy，开启 Zuul 的代理功能。

（3）在程序的配置文件 application.yml 中做相关的配置，配置代码如下：

```
zuul:
  host:
    connect-timeout-millis: 20000
    socket-timeout-millis: 20000
  routes:
    user-service:
      path: /userapi/**
      serviceId: user-service
      sensitiveHeaders:
    blog-service:
      path: /blogapi/**
      serviceId: blog-service
      sensitiveHeaders:
server:
  port: 5000
```

在上述的配置文件 application.yml 中配置了程序的端口号为 5000；配置 zuul.host.connect-timeout-millis 为 20000 毫秒，即 Zuul 连接的超时时间为 20 秒；配置 zuul.host.socket-timeout-millis 为 20000 毫秒，即 socket 的连接超时时间为 20 秒。另外配置了以“serapi/**”开头的请求都转发到 user-service，以“/blogapi/**”开头的请求都转发到 blog-service，最后配置 sensitiveHeaders 为空。

读者可能会有疑问，sensitiveHeaders 这个配置是做什么的呢？sensitiveHeaders 直接翻译为敏感的头部信息。那为什么要将 sensitiveHeaders 设置为空呢？带着这些疑问，我查阅了源码，源码在 ZuulProperties 类，我截取了 sensitiveHeaders 的如下部分源码：

```
    /**
     *List of sensitive headers that are not passed to downstream requests.
     */
     private Set<String> sensitiveHeaders = new LinkedHashSet<>(
               Arrays.asList("Cookie", "Set-Cookie", "Authorization"));
```

sensitiveHeaders 为 LinkedHashSet 集合，里面默认存储了“Cookie”“Set-Cookie”和“Authorization”。由上面的注释可知，敏感头部是不被通过的，即 Zuul 将请求路由转发到其他服务时，会将敏感头部信息去掉。在本案例中，在 Header 中加入的 Token 是以“Authorization”作

为 key，以“Bearer {token}”为 value 的方式加入的。如果 sensitiveHeaders 采用默认配置，Zuul 会将 Header 的“Authorization”值去掉，然后将请求转发给其他服务。到达其他服务的请求由于 Zuul 将 Header 中的 Token 去掉了，所以无法鉴权该请求，该请求会被驳回，提示无权限访问该资源。

16.2.9 构建 admin-service 工程

admin-service 工程集成了 Spring Boot Admin Server，该工程需要向 Eureka Server 注册，获取 Eureka Server 的注册列表信息。获取注册列表信息后，Spring Boot Admin Server 会请求注册列表信息中服务 Actuator 的 API 接口，从而获取这些服务的监控信息。所以，所有向 Eureka Server 注册的其他服务需要加上 Actuator 起步依赖 spring-boot-starter-actuator，Spring Boot 版本为 1.5x。Actuator 开启了安全验证，在本案例中将安全验证关闭，也就是将 management.security.enabled 设置为 false。

另外，在项目中使用了 JMX-bean 的管理功能，需要引入 Jolokia 的依赖。如果需要进行日志管理，要在工程的 Resources 目录下加上 logback-spring.xml 的配置，配置代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <include resource="org/springframework/boot/logging/logback/base.xml"/>
  <jmxConfigurator/>
</configuration>
```

Spring Boot Admin Server 支持集成 Turbine，需要在工程的 pom 文件中引入 Turbine 的起步依赖。在程序的入口类加上@EnableTurbine 注解开启 Turbine。在工程的配置文件 application.yml 中做相关的配置，包括配置 Turbine 的集群方式为默认，配置了 Turbine 的服务名为 monitor-service，代码如下：

```
spring:
  boot:
    admin:
      turbine:
        clusters: default
        location: monitor-service
```

Spring Boot Admin Server 支持安全登录，需要引入登录界面的模块依赖 spring-boot-admin-server-ui-login 和 Spring Security 的起步依赖 spring-boot-starter-security。然后需要在工程做 WebSecurity 的配置，这些配置在第 12 章已经详细讲解过，就不再重复。

关于 Spring Boot Admin，在第 12 章已经做了完整的案例讲解，读者可以参考第 12 章。另外也可以参考官方文档，文档地址为 https://codecentric.github.io/spring-boot-admin/1.5.1。

16.2.10 构建 user-service 工程

在本案例中，user-service 作为资源服务，直接对外提供 API 接口资源，user-service 具有

以下角色或者能力。

- 作为 Eureka Client，向 eureka-server 服务注册中心注册。
- 作为 Config Client，从 config-server 读取配置文件。
- 作为 Zipkin Client，上传链路追踪数据给 Zipkin Server。
- 作为 Spring Boot Admin Client，Spring Boot Admin Server 会定期检查 user-service 的健康状态。
- 作为资源服务器，user-service 的大部分资源需要鉴权，才能访问。
- user-service 用 Feign 作为声明式调用框架，并启动了 Hystrix 熔断器，集成了 Hystrix Dashboard 组件。
- 集成了在线 API 文档框架 Swagger2，采用 RESTful 风格的 API 设计。
- 用 MySQL 作为数据库，并用 JPA 作为 ORM 框架。

大部分的基础服务，包括 eureka-server、config-server、admin-service、zipkin-service、monitor-service、uaa-service、gateway-service，最终都是要为资源服务（例如 user-serive）提供服务治理的能力。这些能力包括服务注册、分布式配置、监控、链路追踪、负载均衡、熔断、路由转发、权限认证等，构成了一个完善的服务治理系统。而资源服务对外直接提供资源，例如 API 资源、静态资源等。

在 user-service 中，一共有 3 个 API 接口，包括“/user/registry”（注册）、“/user/login”（登录）、“/user/{username}”（根据用户名获取用户信息），整体的 API 接口设计采用 RESTful 风格。下面以登录接口为例来做详细说明。

新建一个 UserController 的类，在该类上加@RestController 注解，开启 RestController 功能，加上@RequestMapping("/user")注解，配置了 UserController 类整体的 Url 映射为“/user”。在 UserController 类里面有一个登录的接口“/user/login”。在接口的方法上加@ApiOperation，该注解为 Swagger2 生成 API 文档的注解，其中 value 为 API 接口的名称，notes 为 API 接口的说明。在接口的方法上加@PostMapping("/login")注解，表明该接口为 Post 类型的请求，Url 映射为“/login”。@RequestParam 注解为 API 接口所需的参数的注解，在“/user/login”需要传 username 和 password 两个参数。登录接口的具体代码如下：

```
@RestController
@RequestMapping("/user")
public class UserController {
     @ApiOperation(value = "登录", notes = "username 和 password 为必选项")
     @PostMapping("/login")
     public RespDTO login(@RequestParam String username , @RequestParam String passwo
rd){
          //参数判读省略
       return   userService.login(username,password);
     }
}
```

其中，UserService 为具体登录的逻辑，首先用 UserDao 根据 username 获取用户。如果用户不存在，抛出异常，则提示用户不存在；如果用户存在，校验密码的正确性。如果密码正确，通过 AuthServiceClient 远程调用 uaa-service，获取 JWT，获取 JWT 成功后，将 JWT 封装在 LoginDTO 中。

```
@Service
public class UserService {
     @Autowired
     UserDao userDao;
     @Autowired
     AuthServiceClient authServiceClient;
     public RespDTO login(String username , String password){
         User user= userDao.findByUsername(username);
         if(null==user){
              throw new CommonException(ErrorCode.USER_NOT_FOUND);
         }
         if(!BPwdEncoderUtils.matches(password,user.getPassword())){
              throw new CommonException(ErrorCode.USER_PASSWORD_ERROR);
         }
          JWT jwt = authServiceClient.getToken("Basic dWFhLXNlcnZpY2U6MTIzNDU2", "pas
sword", username, password);
         // 获得用户菜单
         if(null==jwt){
              throw new CommonException(ErrorCode.GET_TOKEN_FAIL);
         }
         LoginDTO loginDTO=new LoginDTO();
         loginDTO.setUser(user);
         loginDTO.setToken(jwt.getAccess_token());
         return RespDTO.onSuc(loginDTO);
     }
}
```

RespDTO 为 API 接口返回数据的统一封装类。CommonException 为自定义的 Runtime 类异常，CommonException 会被异常处理器 CommonExceptionHandler 统一处理，异常统一处理的代码如下：

```
@ControllerAdvice
@ResponseBody
public class CommonExceptionHandler {
     @ExceptionHandler(CommonException.class)
     public ResponseEntity<RespDTO> handleException(Exception e) {
          RespDTO resp = new RespDTO();
          CommonException taiChiException = (CommonException) e;
          resp.code = taiChiException.getCode();
          resp.error = e.getMessage();
          return new ResponseEntity(resp, HttpStatus.OK);
```

```
    }
}
```

16.2.11　构建 blog-service 工程

blog-service 的功能和 user-serive 的功能类似，作为资源服务，对外暴露 API 接口。在本案例中，一共有 3 个 API 接口，分别为“/blog”（发布微博）、“/blog/{username}”（获取某个用户的所有微博）、“/blog/{id}/detail”（获取某条微博的详细信息，包括发布者的详细信息）。现在以获取某条微博的详细信息的 API 接口为例来进行案例详解。其中，这个 API 接口的 Controller 层的代码如下：

```
@RestController
@RequestMapping("/blog")
public class BlogController {
  @Autowired
  BlogService blogService;

  @ApiOperation(value = "获取博文的详细信息", notes = "获取博文的详细信息")
  @PreAuthorize("hasAuthority('ROLE_USER')")
  @GetMapping("/{id}/detail")
  public RespDTO getBlogDetail(@PathVariable Long id){
        return RespDTO.onSuc(blogService.findBlogDetail(id));
    }
}
```

在 Service 层中，首先根据 id 去数据库查询 Blog，如果该 id 对应的 Blog 存在，就通过远程调度 user-service，获取 Blog 发布者的详细信息。代码如下：

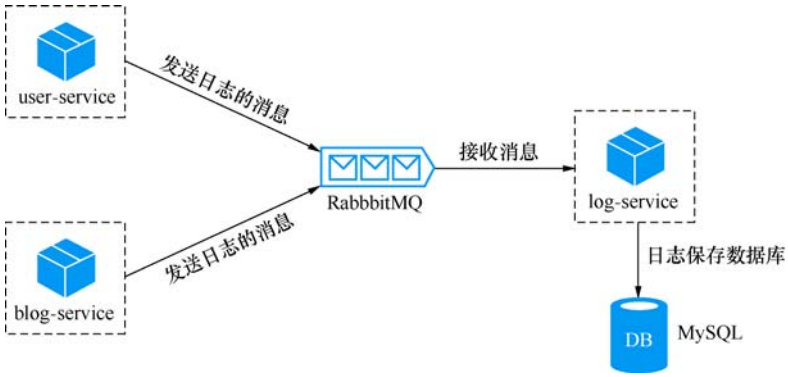
```
public BlogDetailDTO findBlogDetail(Long id) {
        Blog blog = blogDao.findOne(id);
        if (null == blog) {
            throw new CommonException(ErrorCode.BLOG_IS_NOT_EXIST);
        }
        RespDTO<User> respDTO = userServiceClient.
        getUser(UserUtils.getCurrentToken(), blog.getUsername());
        if (respDTO==null) {
            throw new CommonException(ErrorCode.RPC_ERROR);
        }
        BlogDetailDTO blogDetailDTO = new BlogDetailDTO();
        blogDetailDTO.setBlog(blog);
        blogDetailDTO.setUser(respDTO.data);
        return blogDetailDTO;
}
```

16.2.12　构建 log-service 工程

log-service 为日志收集的服务，该服务只收集一些比较重要的日志，并进行持久化，持久

化的数据库为 MySQL。日志服务架构如图 16-11 所示。

在日志服务的架构中，user-service 和 blog-service 发送日志消息给 RabbitMQ 服务器。log-service 通过监听 RabbitMQ 服务器获取日志信息，并通过 JPA 保存日志信息到 MySQL 数据库中。user-service 和 blog-service 通过在 Controller 上的方法加自定义注解@SysLogger，然后通过 AOP 拦截该注解，进行日志信息的提取。提出的信息包括 Controller 方法的方法名、参数、操作人、IP 等。最后，将提取的日志信息发送到 RabbitMQ 服务器。下面来进行详细的代码讲解。



▲图 16-11 日志架构图

在 log-service 的 pom 文件引入 RabbitMQ 的起步依赖 spring-boot-starter-amqp，代码如下：

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

在 log-service 工程的配置文件 application.yml 中做 RabbitMQ 的配置，包括配置了主机 Host、端口、RabbitMQ 的用户名和密码，开启 PublisherConfirm 机制，VritualHost 为默认的根目录，代码如下：

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
    publisher-confirms: true
    virtual-host: /
```

在 log-service 工程写一个 Receiver 类，该类用于接收 RabbitMQ 服务器的消息，并将消息交给 SysLogService 进行数据库的持久化。代码如下：

```
@Component
public class Receiver {
    private CountDownLatch latch = new CountDownLatch(1);
    @Autowired
    SysLogService sysLogService;
    public void receiveMessage(String message) {
        System.out.println("Received <" + message + ">");
        SysLog  sysLog=  JSON.parseObject(message,SysLog.class);
        sysLogService.saveLogger(sysLog);
        latch.countDown();
    }
}
```

在上述代码中，CountDownLatch 起到了类似于信号量的作用。只有其他的线程完成了一系列的操作，通过 latch.countDown()释放信号，其他被阻塞的线程获取到信号才能被唤醒。在接收消息时，是通过 FastJson 进行数据序列化操作的。

然后将 Receiver 类注册到 MessageListenerAdapter 中，注册的方法是将 Receiver 类和 Receiver 类中接收消息的方法名传到 MessageListenerAdapter 构造器中，代码如下：

```
@Bean
 MessageListenerAdapter listenerAdapter(Receiver receiver) {
    return new MessageListenerAdapter(receiver, "receiveMessage");
 }
```

在 Serivce 层，通过 SysLogDAO 将消息保存在 MySQL 数据库中，代码如下：

```
@Service
public class SysLogService {
     @Autowired
     SysLogDAO sysLogDAO;
     public void saveLogger(SysLog sysLog){
          sysLogDAO.save(sysLog);
     }
}
```

log-service 接收消息、消息序列化、保存消息到数据库中代码都已经实现。下面来看看 user-service 和 blog-service 是如何发送消息的。

首先，和 log-service 一样，在工程的 pom 文件中引入 RabitMQ 的起步依赖，然后在工程的配置 application.yml 中做 RabitMQ 相关的配置，最后通过 AmqpTemplate 来发送日志代码如下：

```
@Service
public class LoggerService {
     @Autowired
```

```
    private AmqpTemplate rabbitTemplate;
    public void log(SysLog sysLog){
        rabbitTemplate.convertAndSend(RabbitConfig.queueName, JSON.toJSONString(sysLog));
    }
}
```

那么如何调用发送日志到 LoggerService 呢？在本案例中用 AOP 的方式进行实现。通过自定义注解@SysLogger，然后写一个 AOP 切面，拦截被@SysLogger 注解修饰的方法，提取方法中的信息，再通过 LoggerService 发送，这样做的好处就是业务代码和发送日志的代码进行了松耦合。@SysLogger 注解的代码如下：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface SysLogger {
    String value() default "";
}
```

写一个 SysLoggerAspect 类，类上方加上@Aspect 注解，开启切面。通过@Pointcut()注解来声明一个切点，切点为 SysLogger 注解。通过@Before()注解，表明在进入切点之前进行拦截。在切点的方法里是具体的处理逻辑，包括获取方法名、方法的参数、当前请求的 IP 地址、用户名、当前请求的时间信息。最后将这些信息封装在一个 SysLog 的实体类中，交给 LoggerService 进行处理。具体的代码如下：

```
@Aspect
@Component
public class SysLoggerAspect {
    @Autowired
    private LoggerService loggerService;
    @Pointcut("@annotation(com.forezp.annotation.SysLogger)")
    public void loggerPointCut() {
    }
    @Before("loggerPointCut()")
    public void saveSysLog(JoinPoint joinPoint) {
        MethodSignature signature = (MethodSignature) joinPoint.getSignature();
        Method method = signature.getMethod();
        SysLog sysLog = new SysLog();
        SysLogger sysLogger = method.getAnnotation(SysLogger.class);
        if(sysLogger != null){
            //注解上的描述
            sysLog.setOperation(sysLogger.value());
        }
        //请求的方法名
        String className = joinPoint.getTarget().getClass().getName();
        String methodName = signature.getName();
```

```
            sysLog.setMethod(className + "." + methodName + "()");
            //请求的参数
            Object[] args = joinPoint.getArgs();
            String params="";
            for(Object o:args){
                 params+=JSON.toJSONString(o);
            }
            if(!StringUtils.isEmpty(params)) {
                 sysLog.setParams(params);
            }
            //设置 IP 地址
            sysLog.setIp(HttpUtils.getIpAddress());
            //用户名
            String username = UserUtils.getCurrentPrinciple();
            if(!StringUtils.isEmpty(username)) {
                 sysLog.setUsername(username);
            }
            sysLog.setCreateDate(new Date());
            //保存系统日志
            loggerService.log(sysLog);
        }
}
```

那么如何使用 @SysLogger 注解呢？只需要在方法上加上@SysLogger 注解即可。例如在登录的方法上加上注解@SysLogger("login")，当有用户请求登录 API 接口时，在执行登录方法之前，会进入 SysLoggerAspect 类的切点进行一系列的逻辑处理。最后由 LoggerService 向 RabbitMQ 服务器发送消息，由 log-service 接收消息，进行持久化。登录 API 接口的代码如下：

```
@ApiOperation(value = "登录", notes = "username 和 password 为必选项")
   @PostMapping("/login")
   @SysLogger("login")
   public RespDTO login(@RequestParam String username , @RequestParam String password){
       //参数判断省略
      return   userService.login(username,password);
  }
```

16.3 启动源码工程

安装环境，包括 JDK 1.8、Maven 3.x、RabbitMQ、MySQL、开发工具 IDEA。在 MySQL 数据库中，初始化工程 sql 文件夹下有 sys-user.sql、sys-log.sql 和 sys-blog.sql 这 3 个数据库脚本。

修改 user-service、uaa-service、blog-service 和 log-service 的配置文件，包括 MySQL 数据库和 RabbitMQ 的连接配置信息。

依次启动 eureka-server、config-server、zipkin-service，再启动其他的服务。

16.4 项目演示

首先新注册一个用户，使用 Curl 进行请求，代码如下：

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/
json' -d '{
   "password": "123456",
   "username": "miya"
 }' 'http://localhost:5000/userapi/user/registry'
```

注册成功，返回该用户信息如下：

```
{"id":14,"username":"miya","password":"$2a$10$EphEfhPLiZAb7suZNcsyeOk9uUj/Dh8vSve0ihI
m/2xe8xJs2r4Vy"}
```

使用 Curl 调用登录接口，在该接口中，如果用户名和密码正确，会远程调用 uaa-service 服务获取 JWT。

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/
json' 'http://localhost:5000/userapi/user/login?username=miya&password=123456'
```

登录成功，返回信息如下，其中由于 Token 字段的值太长，就没有展示出来。

```
{
  "code": 0,
  "error": "",
  "data": {
    "user": {
      "id": 14,
      "username": "miya",
      "password": "$2a$10$EphEfhPLiZAb7suZNcsyeOk9uUj/Dh8vSve0ihIm/2xe8xJs2r4Vy"
    },
    "token": "{Token}"
  }
}
```

使用 Curl 调用根据用户名获取用户的 API 接口，代码如下：

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/
json' --header 'Authorization: Bearer {Token}' 'http://localhost:5000/userapi/user/mi
ya'
```

因为该 API 接口需要“ROLE_USER”的权限，而“miya”这个用户是没有这个权限的，所以返回的结果如下：

```
{
  "error": "access_denied",
  "error_description": "不允许访问"
}
```

这时手动数据库插入数据，给“miya”用户“ROLE_USER”的权限，插入数据的脚本如下：

```
INSERT INTO 'user_role' VALUES ('14', '1');
```

重新登录“miya”用户，获取 Token，然后再次请求，就可以得到用户信息了。其他接口类似，就不再演示了。

打开 sys-log 数据库，可以发现刚才调用的登录接口所产生的日志已经存储起来了。

16.5 总结

在本案例中，包括了用 Spring Cloud 构建微服务的一系列基本组件和框架，为读者提供了一个完整的案例。本案例是作者对自己工作和学习的一个总结，读者可以参考甚至直接应用于实际的项目开发中。特别是 Spring Cloud OAuth2 这个模块，作者花费了大量的时间和精力去学习和整理。作者的写作宗旨就是将复杂的问题简单化，力求使整个项目的结构和层次清晰，从而帮助更多的人。希望这个案例能够真正让读者有所收获！

方志朋在 Spring Cloud 中国社区具有广泛的知名度，他为社区中的 Spring Cloud 企业使用者和爱好者提供热心的帮助，其扎实的技术功底和孜孜不倦的钻研精神给大家留下了深刻的印象。希望本书的出版能为广大读者提升微服务的架构能力带来裨益。

——Spring Cloud 中国社区合伙人　任浩军

Spring Cloud 可以说是 2017 年度主流的微服务开发框架之一。从单体架构到分布式架构，再到微服务架构，每个技术团队都在寻求一条系统变革的道路。作者结合架构演进的历史，从全局到细节详细阐述了 Spring Cloud 架构的特点和优势，以及 Spring Cloud 各个组件的功能原理和使用方式。这是一本可以让你从零开始学习 Spring Cloud 的书，也是一本可以让你深入了解架构的书。如果你想学习 Spring Cloud，那么，请从这本书开始！

——宜信高级架构师　梁　鑫

Spring Cloud 为微服务架构提供了一种解决方案，它因提供众多功能强大的模块而成为当前热门的微服务框架。本书以参考用例和源码解读为切入点，由浅入深地对 Spring Cloud 进行了细致的讲解，让读者能够了解 Spring Cloud 的内部原理，并在实际开发中运用 Spring Cloud。新手完全可以通过此书入门，有工作经验的读者研读此书后也会有不一样的收获，强烈推荐本书！

——Spring Cloud 中国社区成员　唐旺辰

异步社区
人民邮电出版社
www.epubit.com.cn

异步社区　www.epubit.com.cn
新浪微博　@人邮异步社区
投稿/反馈邮箱　contact@epubit.com.cn

封面设计：广领设计
分类建议：计算机／系统架构／Java
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-47522-0