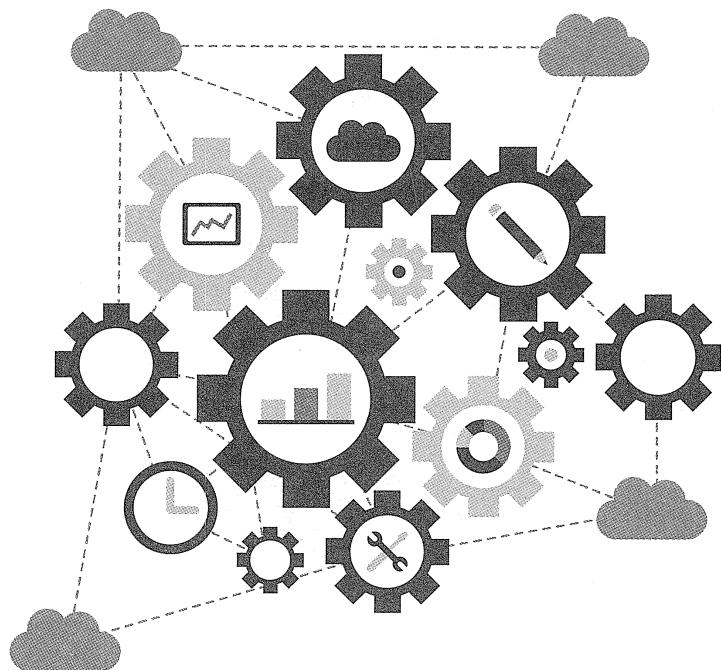


Spring Cloud与Docker

微服务架构实战

周立 著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

作为一部帮助大家实现微服务架构落地的作品，本书基于 Spring Cloud Camden SR4 Docker 1.13.0，覆盖了微服务理论、微服务开发框架（Spring Cloud）以及运行平台（Docker）三大主题。全书可分为三部分，第 1 章对微服务架构进行了系统的介绍；第 2~11 章使用 Spring Cloud 开发框架编写了一个“电影售票系统”；第 12~14 章则讲解了如何将微服务应用运行在 Docker 之上。全书 Demo 驱动学习，以连贯的场景、具体的代码示例来引导读者学习相关知识，最终使用特定的技术栈实现微服务架构的落地。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

Spring Cloud 与 Docker 微服务架构实战 / 周立著. —北京：电子工业出版社，2017.5
ISBN 978-7-121-31271-7

I. ①S… II. ①周… 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 070057 号

策划编辑：张春雨

责任编辑：徐津平

印 刷：北京天宇星印刷厂

装 订：北京天宇星印刷厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×980 1/16 印张：17 字数：342.18 千字

版 次：2017 年 5 月第 1 版

印 次：2017 年 5 月第 1 次印刷

定 价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

序 1

2016 年国庆假期之后，我所在的公司因为业务需要，想搭建一个 API 网关来综合治理已有业务调用服务（我司之前采用的是当当的 Dubbo 扩展框架 Dubbox）。前期，我和同事们在技术选型环节，讨论了诸多目前比较红火的技术框架和工具。最后达成一致，采用微服务来重构和调整原先这些 Dubbox 服务，并决定使用 Spring Cloud（以下简称 sc）来实现 API 网关，争取在 2017 年能顺利平滑地从 Dubbox 过度到 sc。而具体的 API 网关 demo 研发工作就落实到了我这里。

在开始研发工作之前，我参阅了包括官网在内很多 sc 研发资料，也去全球最大的同性技术交友网站 GitHub 上找了很多代码来仔细研读。但感觉老外的这些 Guide（指南）总是讲得不是很通透。也许是有些概念他们觉得太基础了，就直接略过不表。因此我也感到很迷茫，老是问自己，到底应该如何去实现这个 API 网关，完成公司指派给我的研发任务呢？

幸好，某一天我看到《Spring Cloud 与 Docker 实战微服务》这本开源书。根据书中例子，我几乎没有费什么大工夫就搭建了一个 API 网关的 demo。甚至其中某些讲解点，看了之后能让我一下子恍然大悟，回头再看那些老外的 Guide，我终于明白了其中的“奥秘”。我真的非常感激这位开源书作者，他深入浅出地将 sc 所涉及的各种知识点和工具的使用做了完整和详细的叙述。从此，我也记住了此书作者的网名 itmuch。

几天后，我将 demo 做了细化和扩展，并在 oschina 的码云网站上开源分享了出去（具体网址见 <http://git.oschina.net/darkranger/spring-cloud-books>）。而无巧不巧，itmuch 居然在我项目下的评论区留言了。经过加 QQ、加微信等一系列同性技术交友过程（你们懂的）取得了联系，也终于知道了这位 itmuch 的真名，那就是此书作者周立同学。在闲聊过程中，他透露了自己正在以那本《Spring Cloud 与 Docker 实战微服务》开源书为基础，继续扩展和具体深入 sc 这套微服务开发体系所包含的所有技术点，准备出版成册，让更多的朋友和企业能学习和借鉴 sc 这套东西开发符合自己业务场景的微服务框架。并邀请我为新书做校对和修正工作。正巧，我也越来越喜欢钻研 sc，也希望对自己碰到的一些问题向他指教，所以就答应了下来。这其中的过程真的一言难叙，总算最后我也不辱使命地完成了这本书的校对工作。

我可以很负责地说，本书是周立同学本人在工作和学习 sc 后总结出的精华，书中每段代码、每个字都是他自己写的，绝无任何抄袭之举。完全可以说是一个努力勤奋、能独立思考、认真做事的同学的良心之作。这样的“业界良心”在如今这个充满浮躁的社会中已不多见。希望有更多的读者能珍惜此书，感谢周立同学给我们的帮助。除此之外，我也希望读者能在阅读完此书后，可以自己写点代码，亲身去实践一把，感受 sc 的精妙之处。

最后，在仓促完成本文之前，值此新春佳节之际，我也祝大家新年快乐，家庭安康，财源滚滚，爱情事业双丰收。

2017 年 1 月 25 日

农历丙申年腊月二十八

吴峻申 青客机器人有限公司架构师

序 2

2013 年，我在 EMC 听了一个关于 Docker 与测试的分享，才第一次近距离认识 Docker。在 2014 年底时，在项目上开始接触 Docker。2015 年上半年，我读了两本书：*The Phoenix Project* 和 *Migrating to Cloud-Native Application Architectures*。这两本书让我对 DevOps、微服务和云原生架构有了初步的认识。

2015 年 9 月，我以首席架构师的身份加入麻袋理财，当时第一件事情就是借助 DaoCloud 在公司内部推行基于 Docker 的基础落地的方案。花了三个月，一个简易的方案就已经可以正常运作。但是在这个过程中，却发现和应用的契合度不是太高，需要对应用的架构做改造。

2016 年年初，当时正好有一个项目要做 2.0，之前是一个典型的单体应用（使用 Spring MVC），这次准备做微服务改造，以满足业务对技术快速迭代、横向扩展的要求。我当时对 Spring Boot 和 Spring Cloud 已经有所耳闻，但是还停留于 Demo 的地步。正好借着这个机会，准备推广 Spring Boot。之后有个全新的项目，我们完全按照微服务架构，使用 Spring Boot 和 Cloud 进行开发，并采用 CI/CD 自动化流程和容器化部署。

2016 年 10 月份时，一次偶然的计划，Spring Cloud 中国社区的许进找到了我，让我把团队在实践过程中的经验总结在社区做了分享，从而认识了本书的作者周立。当时周立正好在写一本书，他希望我能够帮他进行 review，我就欣然答应了。

看到了书的标题《Spring Cloud 与 Docker 微服务实战》，这不就是我一直在做的工作吗？于是我连夜把这本书读了一遍，感觉相见恨晚，如果一年前有这本书，那我就可以少走很多弯路了。

本书用一个例子贯穿始终，讲解了 Spring Cloud 的经典组件、微服务架构，以及与 Docker 的集成。书中提供了详细的代码，可以让读者在了解基础概念的同时，可以马上脚踏实地地撸起袖子写代码。

王天青 DaoCloud 首席架构师

2017 年 3 月

序 3

最近几年，微服务的概念非常火爆，由于它确实能解决传统单体应用所带来的种种问题（比如代码可维护性低、部署不灵活、不够稳定、不易扩展，等等），所以大家对“如何成功实施微服务架构”越来越感兴趣。在 Java 技术栈中，Spring Cloud 独树一帜，提供了一整套微服务解决方案，它基于 Spring Boot 而构建，延续了 Spring 体系一贯的“简单可依赖”，但是由于微服务本身涉及的技术或概念比较广，所以在正式“入坑”之前，最好能有一本实战性强的书籍作为参考。但是很遗憾，Spring Cloud 太新了，国内几乎没有一本完整讲解其用法的新书。

今年年初，我偶然得知周立兄在编写 Spring Cloud 相关的书籍，感到非常惊喜，在和他交流的过程中，我能感觉到他对技术的把控力以及对知识分享的热情！阅读这本书的过程是非常愉悦的，不仅仅是因为它结构之清晰，文风之流畅，更重要的是实战性极强，相信大家能在本书的指导下顺利地基于 Spring Cloud & Docker 打造出自己的微服务应用。

杜云飞 上海小虫数据技术合伙人，风控大数据负责人

序 4

在 Spring 尚未出现的“蛮荒”时代，Java 程序员们还在迷茫地创造着各种“语法糖”来试图提高生产效率。然而无论怎么努力，Java 语言仍被许多人冠以“裹脚布”的名号——毕竟你一不小心就会把它写得又臭又长。

随着 Spring 体系的出现与逐步完善，似乎有一种经历着 Java 工业革命的感觉。的确，任何事物都各有利弊，但我仍然想说，Spring 团队给 Java 程序员们带来了春天（就像它的名字一样），它神奇地把“裹脚布”变成了“丝绸”，因为它最大的特质可以用两个字来形容——优雅。相信你如果使用过 Spring Framework、Spring MVC、Spring Data、Spring Boot 或 Spring Cloud 等一系列框架，并研读过它们的源代码，就一定能够体会到“优雅”二字的含义。

尽管 Spring 家族拥有如此多而美好的大块“语法糖”，但它们过去在国内的传播似乎都不怎么顺利。我经常说，国内对新技术的广泛应用一般比国外要晚三到五年，无论后端、前端还是架构理念。这是许多因素导致的，比如信息闭塞、语言不通，甚至固步自封。我相信随着国内互联网人才越来越多，新技术应用的延时一定会越来越短。或许很多人为了旧系统的稳定而不愿升级，这可以理解，但我希望人们可以拥抱新的事物，而不是排斥。现如今微服务架构理念兴起，人们急需一个快捷、稳定、一站式的分布式微服务解决方案，Spring Cloud 正是为此而诞生。可国内熟知 Spring Cloud 的人目前仍寥寥无几，大部分人从未听说过，想要学习的人不知从何开始，对官方的英文文档也一知半解。人们需要一本能把他们领进 Spring Cloud 这扇门的“红宝书”，这便是本书的目的，也是本书作者周立的初衷——希望能够为减少国内新技术的延时而出一份力。

我与周立在 2016 年相识，在短暂的交流后我们都产生了相见恨晚的感觉。遇见志同道合的人不容易，我们的技术理念很相似。他有着对技术的热忱、灵活的头脑，以及开源分享技术的无私精神，正是这股精神促使他做了许多分享技术的事情，并且编写了这本书（相信我，写书并不赚钱）。我十分欣赏周立身上的这些特质，因此当他跟我提到想出书并找我帮忙时，我毫不犹豫地答应了他。我相信他未来一定能够成为某一技术领域的专家，这是他的目标，他也具备这样的潜质。

本书的切入点非常好，它并不纠结于冗长的源码解读或原理解释，而是更多地注重实战，这在如今互联网爆炸式发展的时代相当重要。现在人们更倾向于使用敏捷开发尽快做出产品来进行试错，并在后续版本中快速迭代。因此本书的实战经验在软件工程层面上会给予阅读者很大提升，它可以让你更快地搭建分布式微服务架构，然后把精力留在编写业务逻辑上，提高你的生产力，并最终做出更好的产品——这也是 Spring 团队一直希望达到的效果。

现在，让我们随本书进入 Spring Cloud 的世界，一起感受它的优雅吧！

张英磊 云账房 CTO

2017 年 3 月 29 日

前言

随着业务的发展，笔者当时所在公司的项目越来越臃肿。随着代码的堆砌，项目变得越来越复杂、开发效率越来越低、越来越难以维护，小伙伴们苦不堪言，毫无幸福感可言。

我们迫切需要能够解放生产力、放飞小伙伴的“良药”，于是，微服务进入视野。然而，微服务究竟是什么，众说纷纭，没有人能说清楚什么是微服务。不仅如此，大家对微服务的态度也是泾渭分明，吹捧者、贬低者比比皆是，在笔者的QQ群、微信群中硝烟四起。笔者参加了不少交流会，感觉许多分享常常停留在理论阶段。一场会下来，觉得似乎懂了，却苦于没有对应的技术栈去实现这些理论。

Docker、Jenkins 等工具笔者均有涉猎，然而使用什么技术栈去实践微服务架构，在很长一段时间内都是笔者心中的疑问。

2015 年中，笔者偶然在 GitHub 上看到一个名为 Spring Cloud 的框架，它基于 Spring Boot，配置简单、设计优雅，并且大多组件经过了生产环境的考验。笔者花 1 个月左右的时间详细研究了 Spring Cloud 的相关组件后，体会更深。然而，技术选型必须要进行客观、多维度、全方位的分析，而不应由我个人的主观意见作为决定因素。文档丰富程度、社区活跃度、技术栈生态、开发效率、运行效率、成功案例等，都是我们选型的重要因素。经过调研，其他几点都很 OK，只缺成功案例——在当时，国内几乎没什么成功案例，甚至连中文的博客、相关资料都没有。

这让笔者陷入两难，在这一过程中，公司一边继续使用阿里巴巴开源的 Dubbo（Dubbo 虽然在国内非常流行，但毕竟有段时间没有维护了，开源生态也不是很好），一边在笔者的组织下进行一些 Spring Cloud 相关的技术分享。一方面是希望借此开拓小伙伴们的眼界，另一方面也希望能将两者相互印证，看能否在现有平台上借鉴 Spring Cloud 的设计或使用其部分组件。

2016 年 8 月，笔者有幸代表公司参加了全球微服务架构高峰论坛。会上，Josh Long 对 Spring Cloud 的讲解在现场引起了不小的轰动，也让笔者眼前一亮。会后笔者咨询 Josh，Spring Cloud 能否用于生产、是否大规模使用、国内是否已有成功案例，对方一一给出了肯定的回答。这一回答消除了笔者最后的一点疑虑，开始考虑从 Dubbo 逐步迁移至

Spring Cloud 的规划与方案。会后，笔者心想，不妨将 Spring Cloud 相关知识总结成一个“系列博客”，一来是加深自己的理解，二来也算是丰富 Spring Cloud 的中文资料。于是，笔者创建了自己的博客 (<http://www.itmuch.com>)，并开始了系列博客的编写。写了两篇后，笔者将博客链接分享到微信群中，没成想，恰好被 Josh Long 看到，并引用到 Spring 官方博客中去了。这让笔者感到无比荣幸的同时，也让自己贡献开源社区的欲望空前强烈，于是乎，一口气又写了两篇。

再后来，笔者成立了微服务/Spring Cloud/Docker 相关的 QQ 群 (157525002)，在 QQ 群小伙伴的鼓励下，笔者决定写一本 Spring Cloud 开源书 (<https://github.com/eacdy/spring-cloud-book>)，没想到竟然获得开源中国的推荐。再然后，笔者在许进的邀请下，联合创办了 Spring Cloud 中国社区。最后，在群管理员冯靖的引荐下，认识了网红级的大牛张开涛，开涛帮忙引荐了电子工业出版社编辑侠少。从此，笔者正式撰写实体书。

本以为，有了开源书的撰写经验，实体书应该是较为轻松的一件事。然而，样稿发出后，却被侠少鄙视……主要是语文是体育老师教的，病句满天飞，况且，理论不是我的专长。期间一度想要放弃，多亏了侠少的鼓励，总算坚持写下去……

仓促完稿之际，感慨万千，激动与感激交织，于是，本段不可免俗，进入老生常谈的“鸣谢”环节——感谢我的家人，写书是个费时费力的活儿，在近半年的时间，我的父母和妻子给予了我极大的支持；感谢电子工业出版社小伙伴们辛苦工作，没有刘佳禾、孙奇俏、侠少等可爱的朋友们，我的书不可能面世；衷心感谢丁露、冯靖、张英磊、王天青、吴峻申（N 本书的作者）在百忙之中帮忙校对；衷心感谢 QQ 群、微信群的朋友们，你们给了笔者最大的帮助和支持！（注：排名不分先后。）

特别鸣谢：感谢吴峻申给笔者提出很多中肯实用的建议和意见；感谢张英磊帮忙重绘、美化书中绝大部分架构图。

谨以此书献给想要学习微服务、Spring Cloud、Docker 又不知从何开始的读者朋友们。希望本书能切切实实地帮助你使用特定技术栈实现微服务架构的落地，也希望本书不会令你失望。本书很多理论性的内容并未展开，例如 Cloud Native、12-factor App、DDD 等，但笔者都在文中以 TIPS、拓展阅读或 WARNING 的形式进行了标记，这部分内容希望读者能够自行拓展阅读。

排版约定

在阅读本书时，你会遇到不同类型的提示信息，这里展示一些例子及其含义。



推荐阅读按此格式展示。



Tips 按此格式展示。



Warning 按此格式展示。



测试

测试按此格式展示。

读者服务

轻松注册成为博文视点社区用户 (www.broadview.com.cn)，您即可享受以下服务。

- **下载资源：**本书所提供的示例代码及资源文件均可在【下载资源】处下载。
- **提交勘误：**您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **与作者交流：**在页面下方【读者评论】处留下您的疑问或观点，与作者和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/31271>

二维码：



配套代码下载地址

- 1-11 章配套代码：

<https://github.com/itmuch/spring-cloud-docker-microservice-book-code>

- 12-14 章配套代码：

<https://github.com/itmuch/spring-cloud-docker-microservice-book-code-docker>

目录

1	微服务架构概述	1
1.1	单体应用架构存在的问题	1
1.2	如何解决单体应用架构存在的问题	3
1.3	什么是微服务	3
1.4	微服务架构的优点与挑战	4
1.4.1	微服务架构的优点	5
1.4.2	微服务架构面临的挑战	5
1.5	微服务设计原则	6
1.6	如何实现微服务架构	7
1.6.1	技术选型	7
1.6.2	架构图及常用组件	8
2	微服务开发框架——Spring Cloud	9
2.1	Spring Cloud 简介	9
2.2	Spring Cloud 特点	10
2.3	Spring Cloud 版本	10
2.3.1	版本简介	10
2.3.2	子项目一览	11
2.3.3	Spring Cloud/Spring Boot 版本兼容性	12
3	开始使用 Spring Cloud 实战微服务	13
3.1	Spring Cloud 实战前提	13
3.1.1	技术储备	13
3.1.2	工具及软件版本	14
3.2	服务提供者与服务消费者	15

3.3 编写服务提供者	15
3.3.1 手动编写项目	15
3.3.2 使用 Spring Initializr 快速创建 Spring Boot 项目	20
3.4 编写服务消费者	22
3.5 为项目整合 Spring Boot Actuator	23
3.6 硬编码有哪些问题	26
4 微服务注册与发现	27
4.1 服务发现简介	27
4.2 Eureka 简介	29
4.3 Eureka 原理	29
4.4 编写 Eureka Server	31
4.5 将微服务注册到 Eureka Server 上	33
4.6 Eureka Server 的高可用	34
4.7 为 Eureka Server 添加用户认证	37
4.8 Eureka 的元数据	39
4.8.1 改造用户微服务	39
4.8.2 改造电影微服务	39
4.9 Eureka Server 的 REST 端点	41
4.10 Eureka 的自我保护模式	49
4.11 多网卡环境下的 IP 选择	50
4.12 Eureka 的健康检查	51
5 使用 Ribbon 实现客户端侧负载均衡	53
5.1 Ribbon 简介	53
5.2 为服务消费者整合 Ribbon	54
5.3 使用 Java 代码自定义 Ribbon 配置	57
5.4 使用属性自定义 Ribbon 配置	60
5.5 脱离 Eureka 使用 Ribbon	61
6 使用 Feign 实现声明式 REST 调用	63
6.1 Feign 简介	64

6.2	为服务消费者整合 Feign	64
6.3	自定义 Feign 配置.....	66
6.4	手动创建 Feign.....	69
6.4.1	修改用户微服务.....	70
6.4.2	修改电影微服务.....	73
6.5	Feign 对继承的支持	75
6.6	Feign 对压缩的支持	76
6.7	Feign 的日志.....	77
6.8	使用 Feign 构造多参数请求.....	79
6.8.1	GET 请求多参数的 URL	79
6.8.2	POST 请求包含多个参数	81
7	使用 Hystrix 实现微服务的容错处理.....	82
7.1	实现容错的手段	82
7.1.1	雪崩效应.....	83
7.1.2	如何容错.....	83
7.2	使用 Hystrix 实现容错.....	85
7.2.1	Hystrix 简介.....	85
7.2.2	通用方式整合 Hystrix.....	86
7.2.3	Hystrix 断路器的状态监控与深入理解.....	89
7.2.4	Hystrix 线程隔离策略与传播上下文.....	90
7.2.5	Feign 使用 Hystrix	93
7.3	Hystrix 的监控	98
7.4	使用 Hystrix Dashboard 可视化监控数据	100
7.5	使用 Turbine 聚合监控数据.....	102
7.5.1	Turbine 简介	102
7.5.2	使用 Turbine 监控多个微服务	103
7.5.3	使用消息中间件收集数据.....	105
8	使用 Zuul 构建微服务网关	110
8.1	为什么要使用微服务网关	110
8.2	Zuul 简介	112

8.3 编写 Zuul 微服务网关.....	112
8.4 Zuul 的路由端点.....	115
8.5 路由配置详解.....	116
8.6 Zuul 的安全与 Header	119
8.6.1 敏感 Header 的设置	119
8.6.2 忽略 Header.....	120
8.7 使用 Zuul 上传文件	121
8.8 Zuul 的过滤器	124
8.8.1 过滤器类型与请求生命周期	124
8.8.2 编写 Zuul 过滤器.....	125
8.8.3 禁用 Zuul 过滤器.....	127
8.9 Zuul 的容错与回退.....	127
8.10 Zuul 的高可用	130
8.10.1 Zuul 客户端也注册到了 Eureka Server 上	130
8.10.2 Zuul 客户端未注册到 Eureka Server 上.....	131
8.11 使用 Sidecar 整合非 JVM 微服务.....	132
8.11.1 编写 Node.js 微服务	133
8.11.2 编写 Sidecar.....	134
8.11.3 Sidecar 的端点.....	136
8.11.4 Sidecar 与 Node.js 微服务分离部署	136
8.11.5 Sidecar 原理分析	137
8.12 使用 Zuul 聚合微服务	139
9 使用 Spring Cloud Config 统一管理微服务配置.....	144
9.1 为什么要统一管理微服务配置.....	144
9.2 Spring Cloud Config 简介	145
9.3 编写 Config Server.....	146
9.4 编写 Config Client.....	149
9.5 Config Server 的 Git 仓库配置详解	151
9.6 Config Server 的健康状况指示器.....	154
9.7 配置内容的加解密	155
9.7.1 安装 JCE	155

9.7.2	Config Server 的加解密端点	155
9.7.3	对称加密	155
9.7.4	存储加密的内容	156
9.7.5	非对称加密	157
9.8	使用/refresh 端点手动刷新配置	158
9.9	使用 Spring Cloud Bus 自动刷新配置	159
9.9.1	Spring Cloud Bus 简介	159
9.9.2	实现自动刷新	160
9.9.3	局部刷新	161
9.9.4	架构改进	162
9.9.5	跟踪总线事件	163
9.10	Spring Cloud Config 与 Eureka 配合使用	163
9.11	Spring Cloud Config 的用户认证	164
9.12	Config Server 的高可用	166
9.12.1	Git 仓库的高可用	166
9.12.2	RabbitMQ 的高可用	167
9.12.3	Config Server 自身的高可用	167
10	使用 Spring Cloud Sleuth 实现微服务跟踪	169
10.1	为什么要实现微服务跟踪	169
10.2	Spring Cloud Sleuth 简介	170
10.3	整合 Spring Cloud Sleuth	171
10.4	Spring Cloud Sleuth 与 ELK 配合使用	174
10.5	Spring Cloud Sleuth 与 Zipkin 配合使用	178
10.5.1	Zipkin 简介	178
10.5.2	编写 Zipkin Server	178
10.5.3	微服务整合 Zipkin	179
10.5.4	使用消息中间件收集数据	183
10.5.5	存储跟踪数据	185

11 Spring Cloud 常见问题与总结	188
11.1 Eureka 常见问题	188
11.1.1 Eureka 注册服务慢	188
11.1.2 已停止的微服务节点注销慢或不注销	189
11.1.3 如何自定义微服务的 Instance ID	190
11.1.4 Eureka 的 UNKNOWN 问题总结与解决	192
11.2 Hystrix/Feign 整合 Hystrix 后首次请求失败	193
11.2.1 原因分析	193
11.2.2 解决方案	193
11.3 Turbine 聚合的数据不完整	193
11.4 Spring Cloud 各组件配置属性	195
11.4.1 Spring Cloud 的配置	195
11.4.2 原生配置	196
11.5 Spring Cloud 定位问题思路总结	196
12 Docker 入门	199
12.1 Docker 简介	199
12.2 Docker 的架构	199
12.3 安装 Docker	201
12.3.1 系统要求	201
12.3.2 移除非官方软件包	201
12.3.3 设置 Yum 源	201
12.3.4 安装 Dokcer	202
12.3.5 卸载 Docker	203
12.4 配置镜像加速器	204
12.5 Docker 常用命令	204
12.5.1 Docker 镜像常用命令	205
12.5.2 Docker 容器常用命令	206
13 将微服务运行在 Docker 上	211
13.1 使用 Dockerfile 构建 Docker 镜像	211
13.1.1 Dockerfile 常用指令	212

13.1.2 使用 Dockerfile 构建镜像	216
13.2 使用 Docker Registry 管理 Docker 镜像	218
13.2.1 使用 Docker Hub 管理镜像	218
13.2.2 使用私有仓库管理镜像	220
13.3 使用 Maven 插件构建 Docker 镜像	222
13.3.1 快速入门	222
13.3.2 插件读取 Dockerfile 进行构建	224
13.3.3 将插件绑定在某个 phase 执行	225
13.3.4 推送镜像	226
13.4 常见问题与总结	228
14 使用 Docker Compose 编排微服务	229
14.1 Docker Compose 简介	229
14.2 安装 Docker Compose	230
14.2.1 安装 Compose	230
14.2.2 安装 Compose 命令补全工具	230
14.3 Docker Compose 快速入门	231
14.3.1 基本步骤	231
14.3.2 入门示例	231
14.3.3 工程、服务、容器	232
14.4 docker-compose.yml 常用命令	232
14.5 docker-compose 常用命令	236
14.6 Docker Compose 网络设置	238
14.6.1 基本概念	238
14.6.2 更新容器	239
14.6.3 links	239
14.6.4 指定自定义网络	239
14.6.5 配置默认网络	240
14.6.6 使用已存在的网络	241
14.7 综合实战：使用 Docker Compose 编排 Spring Cloud 微服务	241
14.7.1 编排 Spring Cloud 微服务	241
14.7.2 编排高可用的 Eureka Server	245

14.7.3 编排高可用 Spring Cloud 微服务集群及动态伸缩	246
14.8 常见问题与总结	249
后记	250

微服务架构概述



微服务架构是当前软件开发领域的技术热点。它在各种博客、社交媒体和会议演讲上的出镜率非常之高，笔者相信大家也都听说过微服务这个名词。然而微服务似乎又是非常虚幻的——我们找不到微服务的完整定义，以至于很多人认为是在炒作概念。

那么什么是微服务呢？它解决了哪些问题？它又具有哪些特点？诸多问题，本章都将为你一一解答。同时，微服务理论性的内容，互联网上已有很多，本书不会过多提及。笔者会尽量把篇幅花在微服务的具体实战内容上。

1.1 单体应用架构存在的问题

一个归档包（例如 war 格式）包含所有功能的应用程序，通常称为单体应用。而架构单体应用的方法论，就是单体应用架构。

以一个电影售票系统为例，架构如图 1-1 所示。

如图 1-1 所示，该应用尽管已经进行了模块化，但由于 UI 和若干业务模块最终都被打包在一个 war 包中，该 war 包包含了整个系统所有的业务功能，这样的应用系统称为单体应用。

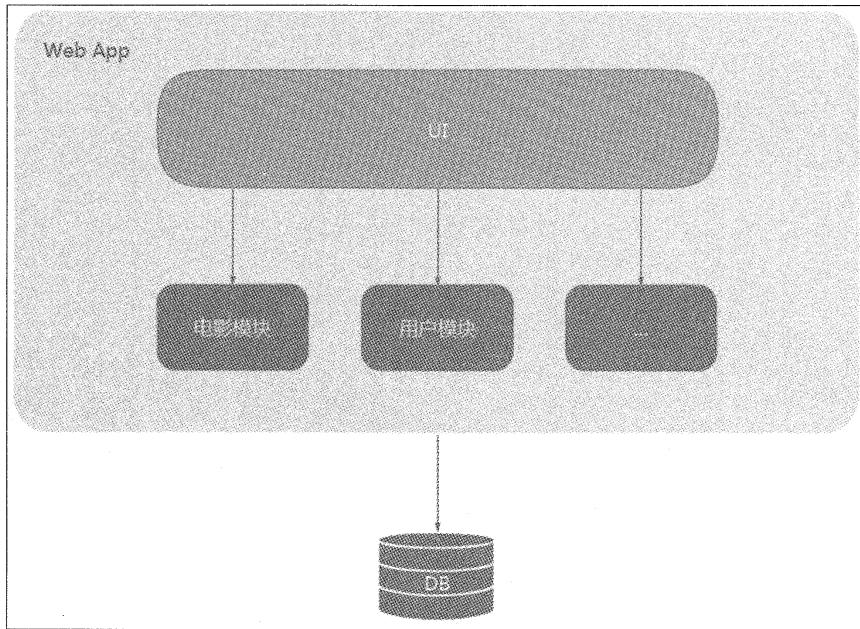


图 1-1 电影售票系统单体架构示意图

相信很多项目都是从单体应用开始的。单体应用比较容易部署、测试，在项目的初期，单体应用可以很好地运行。然而，随着需求的不断增加，越来越多的人加入开发团队，代码库也在飞速地膨胀。慢慢地，单体应用变得越来越臃肿，可维护性、灵活性逐渐降低，维护成本越来越高。下面列举了单体应用存在的一些问题：

- 复杂性高：以笔者经手的一个百万行级别的单体应用为例，整个项目包含的模块非常多、模块的边界模糊、依赖关系不清晰、代码质量参差不齐、混乱地堆砌在一起……整个项目非常复杂。每次修改代码都心惊胆战，甚至添加一个简单的功能，或者修改一个 Bug 都会带来隐含的缺陷。
- 技术债务：随着时间推移、需求变更和人员更迭，会逐渐形成应用程序的技术债务，并且越积越多。“不坏不修（Not broken, don't fix）”，这在软件开发中非常常见，在单体应用中这种思想更甚。已使用的系统设计或代码难以被修改，因为应用程序中的其他模块可能会以意料之外的方式使用它。
- 部署频率低：随着代码的增多，构建和部署的时间也会增加。而在单体应用中，每次功能的变更或缺陷的修复都会导致需要重新部署整个应用。全量部署的方式耗时长、影响范围大、风险高，这使得单体应用项目上线部署的频率较低。而部署频率低又导致两次发布之间会有大量的功能变更和缺陷修复，出错概率比较高。
- 可靠性差：某个应用 Bug，例如死循环、OOM 等，可能会导致整个应用的崩溃。

- 扩展能力受限：单体应用只能作为一个整体进行扩展，无法根据业务模块的需要进行伸缩。例如，应用中有的模块是计算密集型的，它需要强劲的 CPU；有的模块则是 IO 密集型的，需要更大的内存。由于这些模块部署在一起，不得不在硬件的选择上做出妥协。
- 阻碍技术创新：单体应用往往使用统一的技术平台或方案解决所有的问题，团队中的每个成员都必须使用相同的开发语言和框架，要想引入新框架或新技术平台会非常困难。例如，一个使用 Struts 2 构建的、有 100 万行代码的单体应用，如果想要换用 Spring MVC，毫无疑问切换的成本是非常高的。

综上，随着业务需求的发展，功能的不断增加，单体架构很难满足互联网时代业务快速变化的需要。那么，如何解决单体应用架构存在的问题呢？

1.2 如何解决单体应用架构存在的问题

综上所述，单体应用架构存在很多的问题。有没有一种架构模式可以有助于解决这些问题呢？

微服务就是这样的一种架构模式。下面将着重介绍什么是微服务，以及使用微服务架构有哪些优点与挑战。

1.3 什么是微服务

就目前来看，微服务本身并没有一个严格的定义，每个人对微服务的理解都不同。Martin Fowler 在他的博客中是这样描述微服务的。

In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

用中文表述就是，微服务架构风格是一种将一个单一应用程序开发为一组小型服务的方法，每个服务运行在自己的进程中，服务间通信采用轻量级通信机制（通常用 HTTP 资源 API）。这些服务围绕业务能力构建并且可通过全自动部署机制独立部署。这些服务共用一个最小型的集中式的管理，服务可用不同的语言开发，使用不同的数据存储技术。

从中可以看到，微服务架构应具备以下特性：

- 每个微服务可独立运行在自己的进程中。
- 一系列独立运行的微服务共同构建起整个系统。
- 每个服务为独立的业务开发，一个微服务只关注某个特定的功能，例如订单管理、用户管理等。
- 微服务之间通过一些轻量的通信机制进行通信，例如通过 RESTful API 进行调用。
- 可以使用不同的语言与数据存储技术。
- 全自动的部署机制。

还以电影售票系统为例，使用微服务来架构该应用，架构图如图 1-2 所示。

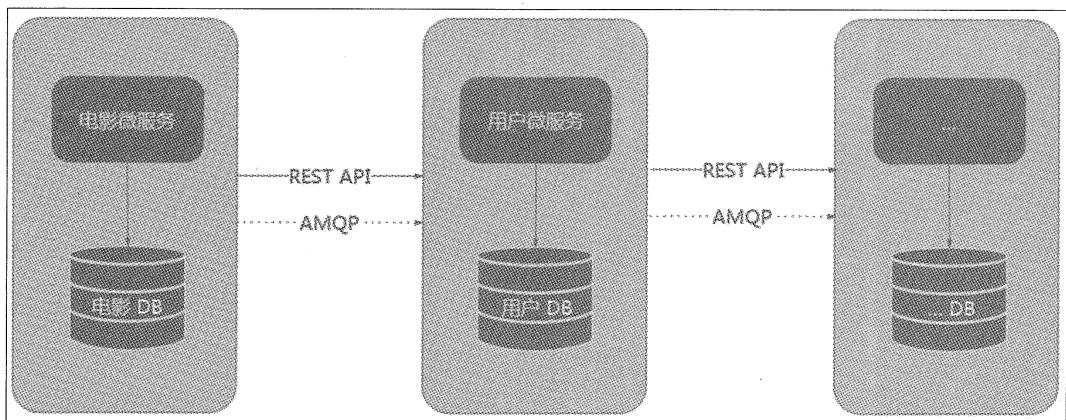


图 1-2 电影售票系统微服务架构示意图

将整个应用分解为多个微服务，各个微服务独立运行在自己的进程中，并分别有自己的数据库，微服务之间使用 REST 或者其他协议通信。



Martin Fowler《微服务》博客原文：<http://www.martinfowler.com/articles/microservices.html>，译文：<http://blog.cuicc.com/blog/2015/07/22/microservices/>。

1.4 微服务架构的优点与挑战

相对单体应用架构来说，微服务架构有着显著的优点。但是，微服务并非是完美的，使用微服务也为我们的工作带来了一定的挑战。先来分析一下使用微服务有哪些优点。

1.4.1 微服务架构的优点

微服务架构有如下优点。

- 易于开发和维护：一个微服务只会关注一个特定的业务功能，所以它业务清晰、代码量较少。开发和维护单个微服务相对简单。而整个应用是由若干个微服务构建而成的，所以整个应用也会被维持在一个可控状态。
- 单个微服务启动较快：单个微服务代码量较少，所以启动会比较快。
- 局部修改容易部署：单体应用只要有修改，就得重新部署整个应用，微服务解决了这样的问题。一般来说，对某个微服务进行修改，只需要重新部署这个服务即可。
- 技术栈不受限：在微服务架构中，可以结合项目业务及团队的特点，合理地选择技术栈。例如某些服务可使用关系型数据库 MySQL；某些微服务有图形计算的需求，可以使用 Neo4j；甚至可根据需要，部分微服务使用 Java 开发，部分微服务使用 Node.js 开发。
- 按需伸缩：可根据需求，实现细粒度的扩展。例如，系统中的某个微服务遇到了瓶颈，可以结合这个微服务的业务特点，增加内存、升级 CPU 或者是增加节点。

综上所述，单体应用架构的缺点，恰恰是微服务的优点，而这些优点使得微服务看起来简直非常完美。然而完美的东西并不存在，就像银弹不存在一样。下面来讨论使用微服务会带来哪些挑战。

1.4.2 微服务架构面临的挑战

微服务虽然有很多吸引人的地方，但它并不是免费的午餐，使用它是有代价的。本节将讨论使用微服务架构面临的挑战。

- 运维要求较高：更多的服务意味着更多的运维投入。在单体架构中，只需要保证一个应用的正常运行。而在微服务中，需要保证几十甚至几百个服务的正常运行与协作，这给运维带来了很大的挑战。
- 分布式固有的复杂性：使用微服务构建的是分布式系统。对于一个分布式系统，系统容错、网络延迟、分布式事务等都会带来巨大的挑战。
- 接口调整成本高：微服务之间通过接口进行通信。如果修改某一个微服务的 API，可能所有使用了该接口的微服务都需要做调整。
- 重复劳动：很多服务可能都会使用到相同的功能，而这个功能并没有达到分解为一个微服务的程度，这个时候，可能各个服务都会开发这一功能，从而导致代码重复。

尽管可以使用共享库来解决这个问题（例如可以将这个功能封装成公共组件，需要该功能的微服务引用该组件），但共享库在多语言环境下就不一定行得通了。

1.5 微服务设计原则

和数据库设计中的 N 范式一样，微服务也有一定的设计原则，这些原则指导我们更加合理地架构微服务。

- **单一职责原则**

单一职责原则指的是一个单元（类、方法或者服务等）只应关注整个系统功能中单独、有界限的一部分。单一职责原则可以帮助我们更优雅地开发、更敏捷地交付。

单一职责原则是 SOLID 原则之一。有兴趣的读者可前往 [https://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)) 进行扩展阅读。

- **服务自治原则**

服务自治是指每个微服务应具备独立的业务能力、依赖与运行环境。在微服务架构中，服务是独立的业务单元，应该与其他服务高度解耦。每个微服务从开发、测试、构建、部署，都应当可以独立运行，而不应该依赖其他的服务。

- **轻量级通信机制**

微服务之间应该通过轻量级的通信机制进行交互。笔者认为，轻量级的通信机制应具备两点：首先是它的体量较轻；其次是它应该是跨语言、跨平台的。例如我们所熟悉的 REST 协议，就是一个典型的“轻量级通信机制”；而例如 Java 的 RMI 则协议就不大符合轻量级通信机制的要求，因为它绑定了 Java 语言。

微服务架构中，常用的协议有 REST、AMQP、STOMP、MQTT 等。

- **微服务粒度**

微服务的粒度是难点，也常常是争论的焦点。应当使用合理的粒度划分微服务，而不是一味地把服务做小。代码量的多少不能作为微服务划分的依据，因为不同的微服务本身的业务复杂性不同，代码量也不同。

在微服务的设计阶段，就应确定其边界。微服务之间应相对独立并保持松耦合。笔者认为，领域驱动设计（Domain Driven Design，简称 DDD）中的“界限上下文（Bounded Context）”可作为划分微服务边界、确定微服务粒度的重要依据。限于篇幅，DDD 的内容无法展开讲解，对 DDD 感兴趣的读者朋友们可阅读《Domain Driven Design Quickly》快速入门，也可阅读 DDD 开山鼻祖 Eric Evans 的《领域驱动设计：软件核心复杂性应对之道》深入理解。同时，在划分微服务的过程中，还应综合考量团队

的现状。康威定律（Conway's Law）相信大家已经很熟悉了，笔者不再赘述。

微服务架构的演进是一个循序渐进的过程。在演进过程中，常常会根据业务的变化，对微服务进行重构，甚至是重新划分，从而让架构更加合理。最终，当微服务的开发、部署、测试以及运维的效率很高，并且成本很低时，一个好的微服务架构就形成了。



- 《Domain Driven Design Quickly》阅读地址：<https://www.infoq.com/minibooks/domain-driven-design-quickly/>。
- 《Domain Driven Design Quickly》中文版《领域驱动设计精简版》：<http://www.infoq.com/cn/minibooks/domain-driven-design-quickly-new>。

1.6 如何实现微服务架构

至此，不仅知道了微服务架构的定义及其优缺点，还总结了一些指导性的原则，为合理架构微服务提供了理论支持。

下面来探讨一下，如何实现微服务架构。

1.6.1 技术选型

相对单体应用的交付，微服务应用的交付要复杂很多，不仅需要开发框架的支持，还需要一些自动化的部署工具，以及 IaaS、PaaS 或 CaaS 的支持。

下面从开发和运行平台两个维度考虑挑选技术选型。

- 开发框架的选择

可使用 Spring Cloud 作为微服务开发框架。

首先，Spring Cloud 具备开箱即用的生产特性，可大大提升开发效率；再者，Spring Cloud 的文档丰富、社区活跃，遇到问题比较容易获得支持；更为可贵的是，Spring Cloud 为微服务架构提供了完整的解决方案。

当然，也可使用其他的开发框架或者解决方案来实现微服务，例如 Dubbo、Dropwizard、Armada 等。

- 运行平台

微服务并不绑定运行平台，将微服务部署在 PC Server，或者阿里云、AWS 等云计算平台都是可以的。出于轻量、灵活、应用支撑等方面的考虑，本书将演示如何在 Docker 上部署微服务。



- Dubbo 的 GitHub: <https://github.com/alibaba/dubbo>。
- Dropwizard 的 GitHub: <https://github.com/dropwizard/dropwizard>。
- Armada 的 GitHub: <https://github.com/armadaplatform/armada>。

1.6.2 架构图及常用组件

在进入实战之前，先来通览一下微服务架构，如图 1-3 所示。

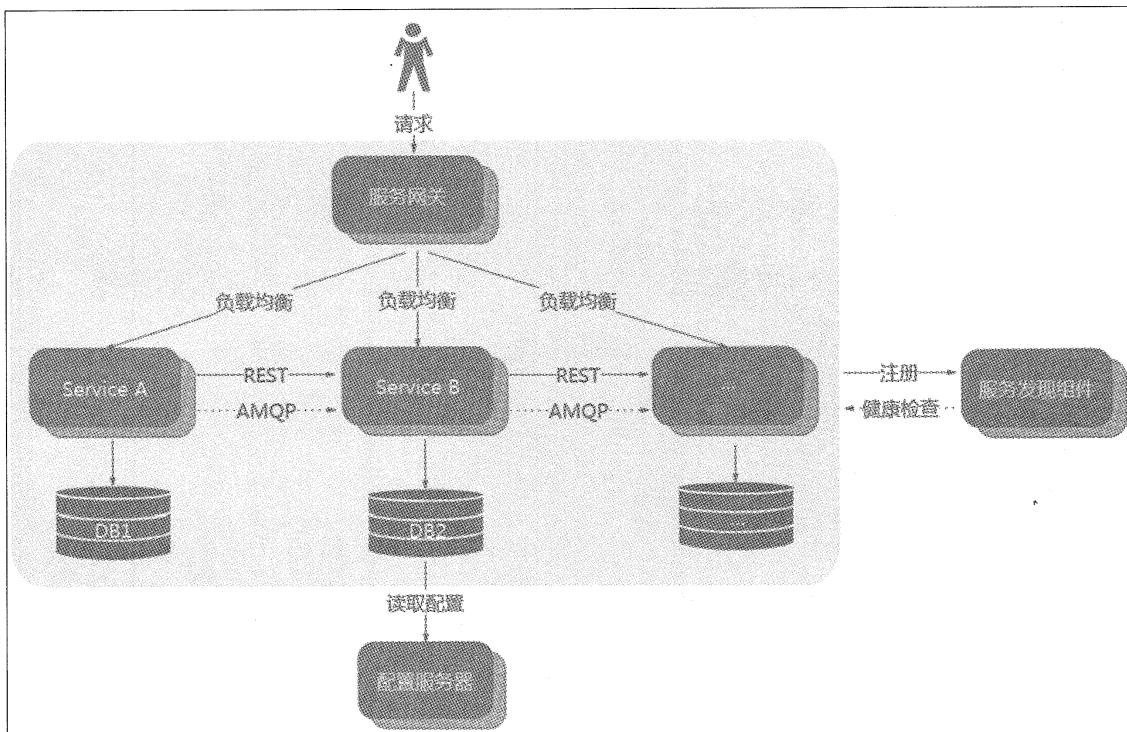


图 1-3 微服务架构图

图 1-3 不严谨地表示了一个微服务应用的架构。图中将所有服务都注册到服务发现组件上，服务之间使用轻量级的通信机制通信。由图可以看到，除了 service A、service B 等，还有服务发现组件、服务网关、配置服务器等组件。这些组件分别是什么？它们的作用又是什么？

读者可以带着疑问继续阅读下去，本书将在实战的过程中进行详细的讲解。



之所以说图 1-3 不严谨，是因为配置服务器可以注册到服务发现组件上；而服务发现组件也可以从配置服务器读取配置信息。

微服务开发框架——Spring Cloud



2.1 Spring Cloud 简介

尽管 Spring Cloud 带有“Cloud”的字样，但它并不是云计算解决方案，而是在 Spring Boot 基础上构建的，用于快速构建分布式系统的通用模式的工具集。

使用 Spring Cloud 开发的应用程序非常适合在 Docker 或者 PaaS（例如 Cloud Foundry）上部署，所以又叫作云原生应用（Cloud Native Application）。云原生（Cloud Native）可简单理解为面向云环境的软件架构。说到云原生，就不得不提一下《十二要素应用宣言（12-factor Apps）》，这是云原生架构的方法论与最佳实践。限于篇幅，本书不作赘述，有兴趣的读者可参考本节的拓展阅读。



- 《Cloud Native Application》电子书：<https://pivotal.io/platform-as-a-service/migrating-to-cloud-native-application-architectures-ebook>。
- 《十二要素应用宣言（12-factor Apps）》中文版：https://12factor.net/zh_cn/。

2.2 Spring Cloud 特点

Spring Cloud 有以下特点：

- 约定优于配置。
- 适用于各种环境。开发、部署在 PC Server 或各种云环境（例如阿里云、AWS 等）均可。
- 隐藏了组件的复杂性，并提供声明式、无 xml 的配置方式。
- 开箱即用，快速启动。
- 轻量级的组件。Spring Cloud 整合的组件大多比较轻量。例如 Eureka、Zuul，等等，都是各自领域轻量级的实现。
- 组件丰富，功能齐全。Spring Cloud 为微服务架构提供了非常完整的支持。例如，配置管理、服务发现、断路器、微服务网关等。
- 选型中立、丰富。例如，Spring Cloud 支持使用 Eureka、Zookeeper 或 Consul 实现服务发现。
- 灵活。Spring Cloud 的组成部分是解耦的，开发人员可按需灵活挑选技术选型。

2.3 Spring Cloud 版本

大多数 Spring 项目都是以“主版本号. 次版本号. 增量版本号. 里程碑版本号”的形式命名版本号的，例如 Spring Framework 稳定版本 4.3.5.RELEASE、里程碑版本 5.0.0.M4 等。其中，主版本号表示项目的重大重构；次版本号表示新特性的添加和变化；增量版本号一般表示 Bug 修复；里程碑版本号表示某版本号的里程碑。

然而，Spring Cloud 并未使用这种方式管理版本。下面来详细探讨一下 Spring Cloud 的版本。

2.3.1 版本简介

先看一下 Spring Cloud 的版本，如图 2-1 所示。

由图 2-1 可知，Spring Cloud 是以英文单词 SRX (X 为数字) 的形式命名版本号的。那么英文单词和 SR 分别表示什么呢？

Spring Cloud 是一个综合项目，它包含很多的子项目。由于子项目也维护着自己的版本号，Spring Cloud 采用了这种版本命名方式，从而避免与子项目的版本混淆。其中，英

文单词叫作“release train”，Angel、Brixton、Camden 等都是伦敦地铁站的名称，它们按照字母顺序发行，可将其理解为主版本的演进。SR 表示“Service Release”，一般表示 Bug 修复；在 SR 版本发布之前，会先发布一个 Release 版本，例如 Camden RELEASE。

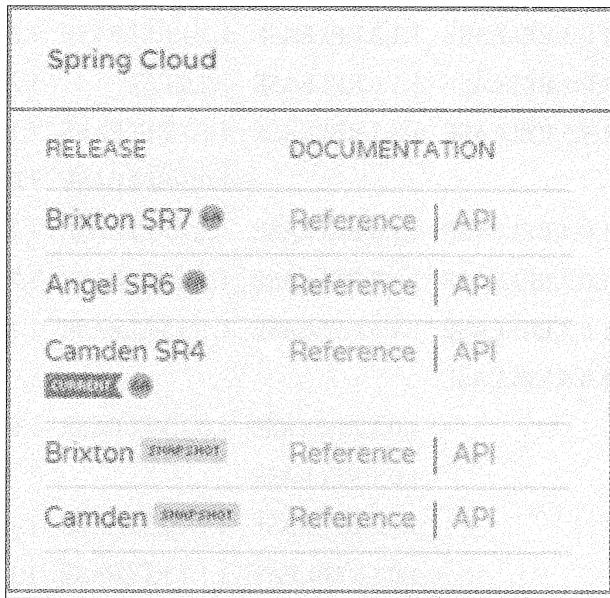


图 2-1 Spring Cloud 版本

经过以上讲解，相信大家就能很好地理解 Spring Cloud 的版本了。例如，Camden SR3 表示 Camden 版本的第 3 次 Bug 修复版本。



- Spring Cloud 版本发布记录可详见：<https://github.com/spring-cloud/spring-cloud-release/releases>，从中可清晰地看到 Spring Cloud 版本发布的时间及先后顺序。
- Spring Cloud 版本演进计划：<https://github.com/spring-cloud/spring-cloud-release/milestones>，从中可了解 Spring Cloud 未来的版本演进计划。
- 事实上，Spring 有不少项目使用类似的命名方式。例如 Spring Data (<http://projects.spring.io/spring-data/>)、Spring Cloud Stream (<http://cloud.spring.io/spring-cloud-stream/>) 等。

2.3.2 子项目一览

理解 Spring Cloud 的版本后，来看一下各版本 Spring Cloud 包含的子项目及版本，如表 2-1 所示。

表 2-1 Spring Cloud 各版本组件

Component	Angel.SR6	Brixton.SR7	Camden.SR4	Camden.BUILD-SNAPSHOT
spring-cloud-aws	1.0.4.RELEASE	1.1.3.RELEASE	1.1.3.RELEASE	1.1.4.BUILD-SNAPSHOT
spring-cloud-bus	1.0.3.RELEASE	1.1.2.RELEASE	1.2.1.RELEASE	1.2.2.BUILD-SNAPSHOT
spring-cloud-cli	1.0.6.RELEASE	1.1.6.RELEASE	1.2.0.RC1	1.2.0.BUILD-SNAPSHOT
spring-cloud-commons	1.0.5.RELEASE	1.1.3.RELEASE	1.1.7.RELEASE	1.1.8.BUILD-SNAPSHOT
spring-cloud-contract			1.0.3.RELEASE	1.0.4.BUILD-SNAPSHOT
spring-cloud-config	1.0.4.RELEASE	1.1.3.RELEASE	1.2.1.RELEASE	1.2.2.BUILD-SNAPSHOT
spring-cloud-netflix	1.0.7.RELEASE	1.1.7.RELEASE	1.2.4.RELEASE	1.2.5.BUILD-SNAPSHOT
spring-cloud-security	1.0.3.RELEASE	1.1.3.RELEASE	1.1.3.RELEASE	1.1.3.BUILD-SNAPSHOT
spring-cloud-starters	1.0.6.RELEASE			
spring-cloud-cloudfoundry		1.0.1.RELEASE	1.0.1.RELEASE	1.0.2.BUILD-SNAPSHOT
spring-cloud-cluster		1.0.1.RELEASE		
spring-cloud-consul		1.0.2.RELEASE	1.1.2.RELEASE	1.1.3.BUILD-SNAPSHOT
spring-cloud-sleuth		1.0.11.RELEASE	1.1.1.RELEASE	1.1.2.BUILD-SNAPSHOT
spring-cloud-stream		1.0.2.RELEASE	Brooklyn.RC1	Brooklyn.BUILD-SNAPSHOT
spring-cloud-zookeeper		1.0.3.RELEASE	1.0.3.RELEASE	1.0.4.BUILD-SNAPSHOT
spring-boot	1.2.8.RELEASE	1.3.8.RELEASE	1.4.2.RELEASE	1.4.2.BUILD-SNAPSHOT
spring-cloud-task		1.0.3.RELEASE	1.0.3.RELEASE	1.0.4.BUILD-SNAPSHOT

从中不难发现，Angel 版本包含的子项目相对较少；Brixton、Camden 则提供了更多的组件。相信随着项目的迭代，Spring Cloud 会提供更多的组件与更丰富的特性，从而让开发更加简单、快速。

2.3.3 Spring Cloud/Spring Boot 版本兼容性

- Angel 版本基于 Spring Boot 1.2.x 构建，在一些场景下，与 Spring Boot 1.3.x 及以上版本不兼容。
- Brixton 版本基于 Spring Boot 1.3.x 构建，也可使用 1.4.x 进行测试，与 Spring Boot 1.2.x 不兼容。
- Camden 版本基于 Spring Boot 1.4.x 构建，也可使用 1.5.x 进行测试。

读者可前往 <http://projects.spring.io/spring-cloud/> 查看版本兼容性。



开始使用 Spring Cloud 实战微服务

本章正式开始使用 Spring Cloud 进行微服务实践。

3.1 Spring Cloud 实战前提

Spring Cloud 不一定适合所有人。先来探讨一下，玩转 Spring Cloud 需要具备什么样的技术能力，以及在实战中会使用到哪些工具。

3.1.1 技术储备

Spring Cloud 并不是面向零基础开发人员的，它有一定的学习曲线。

- 语言基础：Spring Cloud 是一个基于 Java 语言的工具套件，所以学习它需要一定的 Java 基础。当然，Spring Cloud 同样也支持使用 Scala、Groovy 等语言进行开发。本书的示例代码都是使用 Java 编写的。
- Spring Boot：Spring Cloud 是基于 Spring Boot 构建的，因此它延续了 Spring Boot 的契约模式以及开发方式。如果大家对 Spring Boot 不熟悉，建议花一点时间入门。

当然，本书会尽量照顾到不熟悉 Spring Boot 的读者，由浅入深地进行讲解。

- 项目管理与构建工具：目前业界比较主流的项目管理与构建工具有 Maven 和 Gradle 等，本书采用的是目前相对主流的 Maven。大家也可使用 Gradle 管理与构建项目。并且，Maven 与 Gradle 项目可以互相转换，例如，使用以下命令即可将 Maven 项目转换为 Gradle 项目。

```
gradle init --type pom
```

3.1.2 工具及软件版本

目前 Spring Cloud 正在飞速地发展，是业界有名的“版本帝”。2015 年 7 月，Spring Cloud 才刚刚发布 Angel RELEASE；而 2016 年 5 月，Spring Cloud 就已经发布到 Brixton RELEASE；到 2016 年 9 月，Spring Cloud 又发布了新一代产品 Camden RELEASE。随着版本的演进，Spring Cloud 带来了更丰富的组件、更强大的功能，并解决了很多遗留的 Bug。

新版本未必代表着完美，但老版本往往意味着过时或即将过时。基于这个原则，笔者使用目前最新的 Release 版本进行讲解。涉及到的新特性，笔者会尽量标记并做出讲解。

下面列出了笔者所使用的各个软件及其版本。

- JDK：Spring Cloud 官方建议使用 JDK 1.8。当然，Spring Cloud 也支持通过一定的配置，使用 JDK 1.7 进行开发。笔者使用的是 JDK 1.8。
- Spring Boot：本书使用 Spring Boot 1.4.3.RELEASE。
- Spring Cloud：本书使用 Spring Cloud Camden SR4。
- IDE 的选择：选择一款强大的 IDE 往往能够事半功倍。笔者使用的 IDE 是 Spring 官方提供的 Spring Tool Suite 3.8.3，这是一个基于 Eclipse 的 IDE。当然，使用 IntelliJ IDEA 等 IDE 进行开发也是可以的。
- Maven：笔者使用 Maven 3.3.9 构建项目。和 Spring Boot、Spring Cloud 一样，Maven 3.3.x 默认也是运行在 JDK 1.8 之上的。如果想使用 1.8 以下版本的 JDK，需要做一些额外的配置。



目前，Spring Cloud 版本演进速度很快，不同版本之间有一定差异，建议大家在学习时，尽量选用与本书一致的软件版本。要知道，学习是有成本的，这个成本是时间和精力，降低学习成本的重要方法之一就是少踩坑。因此，建议使用与本书相同的版本进行学习，掌握相关知识并具备解决问题的能力后，再按照项目需求挑选适合生产的版本。

3.2 服务提供者与服务消费者

使用微服务构建的是分布式系统，微服务之间通过网络进行通信。我们使用服务提供者与服务消费者来描述微服务之间的调用关系。表 3-1 解释了服务提供者与服务消费者。

表 3-1 服务提供者与服务消费者

名词	定义
服务提供者	服务的被调用方（即：为其他服务提供服务的服务）
服务消费者	服务的调用方（即：依赖其他服务的服务）

我们继续以电影售票系统为例。如图 3-1，用户向电影微服务发起了一个购票的请求。在进行购票的业务操作前，电影微服务需要调用用户微服务的接口，查询当前用户的余额是多少，是不是符合购票标准等。在这种场景下，用户微服务就是一个服务提供者，电影微服务则是一个服务消费者。

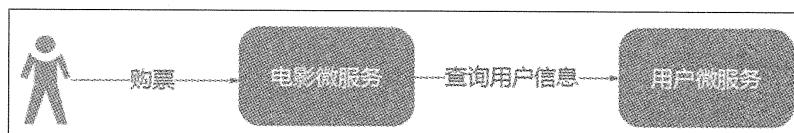


图 3-1 服务提供者与服务消费者

围绕该场景，先来编写一个用户微服务，然后编写一个电影微服务。

3.3 编写服务提供者

本节将编写一个服务提供者（用户微服务），该服务可通过主键查询用户信息。为便于测试，使用 Spring Data JPA 作为持久层框架，使用 H2 作为数据库。

3.3.1 手动编写项目

- 创建一个 Maven 项目，它的 ArtifactId 是 microservice-simple-provider-user，pom.xml 的内容如下：

```

<?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.itmuch.cloud</groupId>
<artifactId>microservice-simple-provider-user</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<!-- 引入spring boot的依赖 -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.3.RELEASE</version>
</parent>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <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-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
    </dependency>
</dependencies>

<!-- 引入spring cloud的依赖 -->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
```

```
<version>Camden.SR4</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>

<!-- 添加spring-boot的maven插件 -->
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
</project>
```

其中，spring-boot-starter-web 提供了 Spring MVC 的支持；spring-boot-starter-data-jpa 提供了 Spring Data JPA 的支持。

2. 准备好建表语句，在项目的 classpath 下建立 schema.sql，并添加如下内容：

```
drop table user if exists;
create table user (id bigint generated by default as identity, username varchar(40), name varchar(20), age int(3), balance decimal(10,2), primary key (id));
```

3. 准备几条数据，在项目的 classpath 下建立文件 data.sql，并添加如下内容：

```
insert into user (id, username, name, age, balance) values (1, 'account1', '张三', 20, 100.00);
insert into user (id, username, name, age, balance) values (2, 'account2', '李四', 28, 180.00);
insert into user (id, username, name, age, balance) values (3, 'account3', '王五', 32, 280.00);
```

4. 创建用户实体类：

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

```

@Column
private String username;
@Column
private String name;
@Column
private Integer age;
@Column
private BigDecimal balance;
...
// getters and setters
}

```

5. 创建 DAO:

```

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
}

```

6. 创建 Controller:

```

@RestController
public class UserController {
    @Autowired
    private UserRepository userRepository;

    @GetMapping("/{id}")
    public User findById(@PathVariable Long id) {
        User findOne = this.userRepository.findOne(id);
        return findOne;
    }
}

```

Controller 中用到的 @GetMapping，是 Spring 4.3 提供的新注解。它是一个组合注解，等价于 @RequestMapping(method = RequestMethod.GET)，用于简化开发。同理还有 @PostMapping、@PutMapping、@DeleteMapping、@PatchMapping 等。

7. 编写启动类，在类上使用 @SpringBootApplication 声明这是一个 Spring Boot 项目。

```

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

```

@SpringBootApplication是一个组合注解，它整合了@Configuration、@EnableAutoConfiguration 和@ComponentScan注解，并开启了 Spring Boot 程序的组件扫描和自动配置功能。在开发 Spring Boot 程序的过程中，常常会组合使用 @Configuration、@EnableAutoConfiguration 和@ComponentScan等注解，所以 Spring Boot 提供了 @SpringBootApplication，来简化开发。

8. 编写配置文件，命名为 application.yml。

```
server:  
  port: 8000  
spring:  
  jpa:  
    generate-ddl: false  
    show-sql: true  
    hibernate:  
      ddl-auto: none  
  datasource:          # 指定数据源  
    platform: h2          # 指定数据源类型  
    schema: classpath:schema.sql  # 指定h2数据库的建表脚本  
    data: classpath:data.sql    # 指定h2数据库的数据脚本  
logging:           # 配置日志级别，让hibernate打印执行的SQL  
  level:  
    root: INFO  
    org.hibernate: INFO  
    org.hibernate.type.descriptor.sql.BasicBinder: TRACE  
    org.hibernate.type.descriptor.sql.BasicExtractor: TRACE
```

在传统的 Web 开发中，常使用 properties 格式文件作为配置文件。Spring Boot 以及 Spring Cloud 支持使用 properties 或者 yml 格式的文件作为配置文件。

yml 文件格式是 YAML (Yet Another Markup Language) 编写的文件格式，YAML 和 properties 格式的文件可互相转换，例如本节中的 application.yml，就等价于如下的 properties 文件。

```
server.port=8000  
spring.jpa.generate-ddl=false  
spring.jpa.show-sql=true  
spring.jpa.hibernate.ddl-auto=none  
spring.datasource.platform=h2  
spring.datasource.schema=classpath:schema.sql  
spring.datasource.data=classpath:data.sql
```

```
logging.level.root=INFO
logging.level.org.hibernate=INFO
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
logging.level.org.hibernate.type.descriptor.sql.BasicExtractor=TRACE
```

从中不难看出，YAML 比 properties 结构清晰；可读性、可维护性也更强，并且语法非常简洁。因此，本书使用 YAML 格式作为配置文件。另外，yml 有严格的缩进，请大家注意。



测试

访问：<http://localhost:8000/1>，获得结果：

```
{
    "id": 1,
    "username": "account1",
    "name": "张三",
    "age": 20,
    "balance": 100
}
```

说明已可通过 ID 查询用户信息。

3.3.2 使用 Spring Initializr 快速创建 Spring Boot 项目

之前是手动创建项目的。事实上，也可使用 Spring Initializr 快速创建项目。虽然 Spring Initializr 不能生成应用程序的业务代码，但它可生成基本的项目结构。这样就可以把更多精力放在具体的业务代码上，而无须过分关注项目搭建的过程。

Spring Initializr 有以下几种用法：

- 通过网页使用。
- 通过 Spring Tool Suite 使用。
- 通过 IntelliJ IDEA 使用。
- 使用 Spring Boot CLI 使用。

笔者以第一种方式为例进行讲解，其他方式大致类似，请读者自行发掘。

1. 访问：<http://start.spring.io/>，会看到类似图 3-2 的界面。
2. 按照需求，选择项目类型（Maven 或 Gradle）、Spring Boot 的版本，并填写项目元数据以及所需依赖。如点击“Switch to the full version”按钮，还可指定额外的信息，例如 Java 版本、打包方式等。

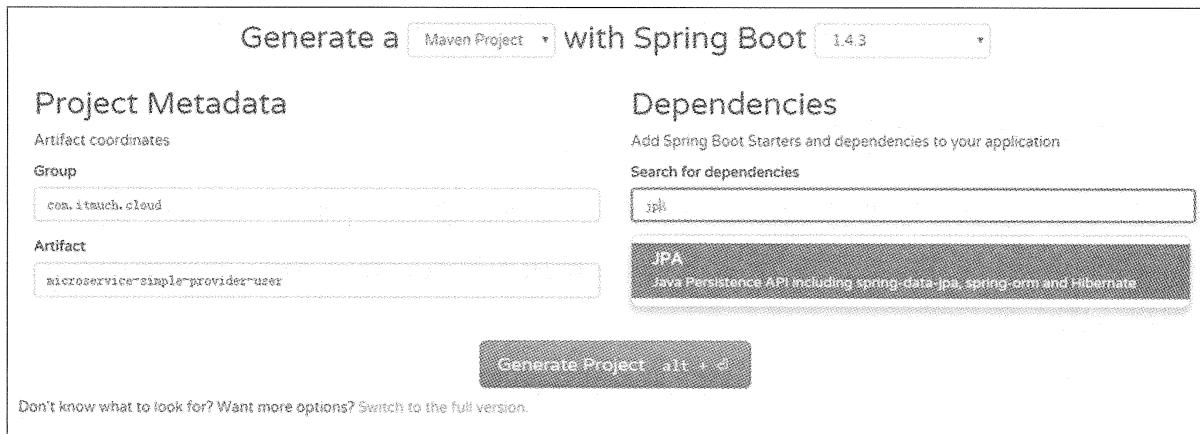
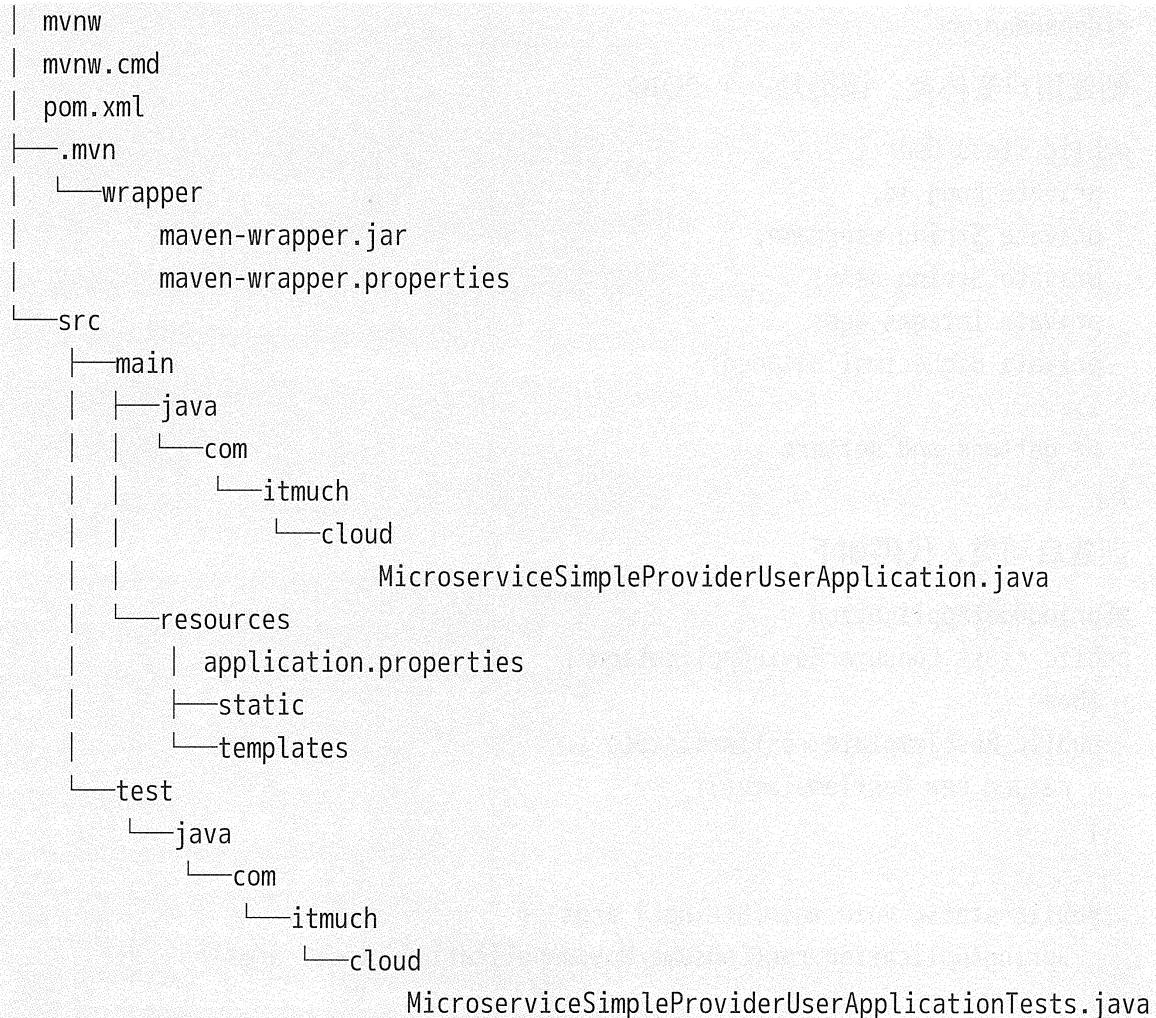


图 3-2 Spring Initializr 首页

3. 点击 Generate Project 按钮，就能获得一个名为 microservice-simple-provider-user.zip 的压缩包文件。解压后，项目结构如下：



将项目导入到 IDE 中，就可以进入 Spring Boot/Spring Cloud 开发之旅了。

3.4 编写服务消费者

前文编写了一个服务提供者（用户微服务），本节来编写一个服务消费者（电影微服务）。该服务非常简单，它使用 RestTemplate 调用用户微服务的 API，从而查询指定 id 的用户信息。

1. 创建一个 Maven 项目，ArtifactId 是 microservice-simple-consumer-movie。
2. 和用户微服务一样，电影微服务也需引入 Spring Boot 及 Spring Cloud 的依赖管理，在此基础上，添加以下依赖。

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

3. 创建用户实体类，该类是一个 POJO。

```
public class User {
    private Long id;
    private String username;
    private String name;
    private Integer age;
    private BigDecimal balance;
    ...
    // getters and setters
}
```

4. 创建启动类，代码如下。

```
@SpringBootApplication
public class ConsumerMovieApplication {
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

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

@Bean 是一个方法注解，作用是实例化一个 Bean 并使用该方法的名称命名。在本例中，添加@Bean 注解的restTemplate() 方法，等价于RestTemplate restTemplate = new RestTemplate();。

5. 创建 Controller，在其中使用 RestTemplate 请求用户微服务的 API。

```
@RestController
public class MovieController {
    @Autowired
    private RestTemplate restTemplate;

    @GetMapping("/user/{id}")
    public User findById(@PathVariable Long id) {
        return this.restTemplate.getForObject("http://localhost:8000/" + id, User.
            class);
    }
}
```

6. 编写配置文件 application.yml:

```
server:
  port: 8010
```

这样，一个电影微服务就完成了，so easy！

测试

访问： <http://127.0.0.1:8010/user/1>，结果如下：

```
{
    "id": 1,
    "username": "account1",
    "name": "张三",
    "age": 20,
    "balance": 100
}
```

说明电影微服务可以正常使用 RestTemplate 调用用户微服务的 API。

3.5 为项目整合 Spring Boot Actuator

Spring Boot Actuator 提供了很多监控端点。可使用<http://{ip}:{port}/{endpoint}>的形式访问这些端点，从而了解应用程序的运行状况。

Actuator 提供的端点，如表 3-2 所示。

表 3-2 Spring Boot Actuator 常用端点及描述

端点	描述	HTTP 方法
autoconfig	显示自动配置的信息	GET
beans	显示应用程序上下文所有的 Spring bean	GET
configprops	显示所有 @ConfigurationProperties 的配置属性列表	GET
dump	显示线程活动的快照	GET
env	显示应用的环境变量	GET
health	显示应用程序的健康指标，这些值由 HealthIndicator 的实现类提供	GET
info	显示应用的信息，可使用 info.* 属性自定义 info 端点公开的数据	GET
mappings	显示所有的 URL 路径	GET
metrics	显示应用的度量标准信息	GET
shutdown	关闭应用（默认情况下不启用，如需启用，需设置 endpoints.shutdown.enabled=true）	POST
trace	显示跟踪信息（默认情况下为最近 100 个 HTTP 请求）	GET

由于在后面有大量的章节需要用到 Actuator，不妨先来为项目整合 Actuator，以项目 microservice-simple-provider-user 为例。

为项目添加以下依赖。

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

这样，就整合好 Actuator 了。是不是十分简单呢？同理，也可为项目 microservice-simple-consumer-movie 整合 Actuator。

测试

- 访问 `http://localhost:8000/health`，可获得类似如下的结果。

```
{
    "status": "UP",
    "diskSpace": {
        "status": "UP",
        "fs": [
            {
                "path": "/",
                "free": 95381776256,
                "total": 1000000000000
            }
        ]
    }
}
```

```
        "total": 214750457856,
        "free": 183337144320,
        "threshold": 10485760
    },
    "db": {
        "status": "UP",
        "database": "H2",
        "hello": 1
    }
}
```

其中， UP 表示运行正常，除 UP 外，还有 DOWN、OUT_OF_SERVICE、UNKNOWN 等状态。

2. 访问 <http://localhost:8000/info>，可看到以下内容。

```
{}
```

由结果可知，info 端点并没有返回任何数据给我们。可使用 info.* 属性来自定义 info 端点公开的数据，例如：

```
info:
  app:
    name: @project.artifactId@
    encoding: @project.build.sourceEncoding@
  java:
    source: @java.version@
    target: @java.version@
```

这样，重启后，再次访问 <http://localhost:8000/info>，就会看到类似如下的内容。

```
{
  "app": {
    "name": "microservice-simple-provider-user",
    "encoding": "UTF-8",
    "java": {
      "source": "1.8.0_92",
      "target": "1.8.0_92"
    }
  }
}
```

由结果可知，info 端点返回了项目名称、编码、Java 版本等信息。

Actuator 端点众多，其他端点读者可参考 Spring Boot 文档自行测试，笔者不作赘述。

3.6 硬编码有哪些问题

至此，已经实现了一个用户微服务和电影微服务，并在电影微服务中使用 RestTemplate 调用用户微服务中的 RESTful API。一切都是那么的自然、简单、perfect！

那么真的完美吗？来分析一下电影微服务的代码，在 MovieController.java 中：

```
@GetMapping("/user/{id}")
public User findById(@PathVariable Long id) {
    return this.restTemplate.getForObject("http://localhost:8000/" + id, User.class);
}
```

由代码可知，我们是把提供者的网络地址（IP 和端口等）硬编码在代码中的，当然，也可将其提取到配置文件中去。例如：

```
user:
  userServiceUrl: http://localhost:8000/
```

代码改为：

```
@Value("user.userServiceUrl")
private String userServiceUrl;

@GetMapping("/user/{id}")
public User findById(@PathVariable Long id) {
    return this.restTemplate.getForObject(this.userServiceUrl + id, User.class);
}
```

在传统的应用程序中，一般都是这么做的。然而，这种方式有很多问题。

- 适用场景有局限：如果服务提供者的网络地址（IP 和端口）发生了变化，将会影响服务消费者。例如，用户微服务的网络地址发生变化，就需要修改电影微服务的配置，并重新发布。这显然是不可取的。
- 无法动态伸缩：在生产环境中，每个微服务一般都会部署多个实例，从而实现容灾和负载均衡。在微服务架构的系统中，还需要系统具备自动伸缩的能力，例如动态增减节点等。硬编码无法适应这种需求。

那么要如何解决这些问题呢？请大家继续阅读下去。

微服务注册与发现



4.1 服务发现简介

通过前文的讲解，我们知道硬编码提供者地址的方式有不少问题。要想解决这些问题，服务消费者需要一个强大的服务发现机制，服务消费者使用这种机制获取服务提供者的网络信息。不仅如此，即使服务提供者的信息发生变化，服务消费者也无须修改配置文件。

服务发现组件提供这种能力。在微服务架构中，服务发现组件是一个非常关键的组件。

使用服务发现组件后的架构图，如图 4-1 所示。

服务提供者、服务消费者、服务发现组件这三者之间的关系大致如下：

- 各个微服务在启动时，将自己的网络地址等信息注册到服务发现组件中，服务发现组件会存储这些信息。
- 服务消费者可从服务发现组件查询服务提供者的网络地址，并使用该地址调用服务提供者的接口。
- 各个微服务与服务发现组件使用一定机制（例如心跳）通信。服务发现组件如长时间无法与某微服务实例通信，就会注销该实例。

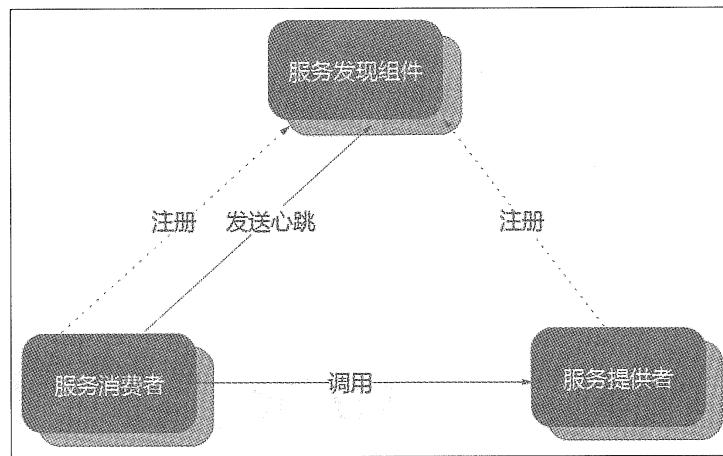


图 4-1 服务发现架构图

- 微服务网络地址发生变更（例如实例增减或者 IP 端口发生变化等）时，会重新注册到服务发现组件。使用这种方式，服务消费者就无须人工修改提供者的网络地址了。

综上，服务发现组件应具备以下功能。

- 服务注册表：是服务发现组件的核心，它用来记录各个微服务的信息，例如微服务的名称、IP、端口等。服务注册表提供查询 API 和管理 API，查询 API 用于查询可用的微服务实例，管理 API 用于服务的注册和注销。
- 服务注册与服务发现：服务注册是指微服务在启动时，将自己的信息注册到服务发现组件上的过程。服务发现是指查询可用微服务列表及其网络地址的机制。
- 服务检查：服务发现组件使用一定机制定时检测已注册的服务，如发现某实例长时间无法访问，就会从服务注册表中移除该实例。

综上，使用服务发现的好处是显而易见的。Spring Cloud 提供了多种服务发现组件的支持，例如 Eureka、Consul 和 Zookeeper 等。本书将以 Eureka 为例，为大家详细讲解服务注册与发现。



- 目前市面上的书籍中所提到的服务注册、服务发现或注册中心等名词，多数场景下都可理解为服务发现组件。
- 服务发现的方式可分为服务器端发现和客户端发现，由于原理相通，本书不再赘述。

4.2 Eureka 简介

Eureka 是 Netflix 开源的服务发现组件，本身是一个基于 REST 的服务。它包含 Server 和 Client 两部分。Spring Cloud 将它集成在子项目 Spring Cloud Netflix 中，从而实现微服务的注册与发现。



- Eureka 的 GitHub： <https://github.com/Netflix/Eureka>。
- Netflix 是一家在线影片租赁提供商。

4.3 Eureka 原理

在分析 Eureka 的原理之前，先来了解一下 Region 和 Availability Zone，如图 4-2 所示。

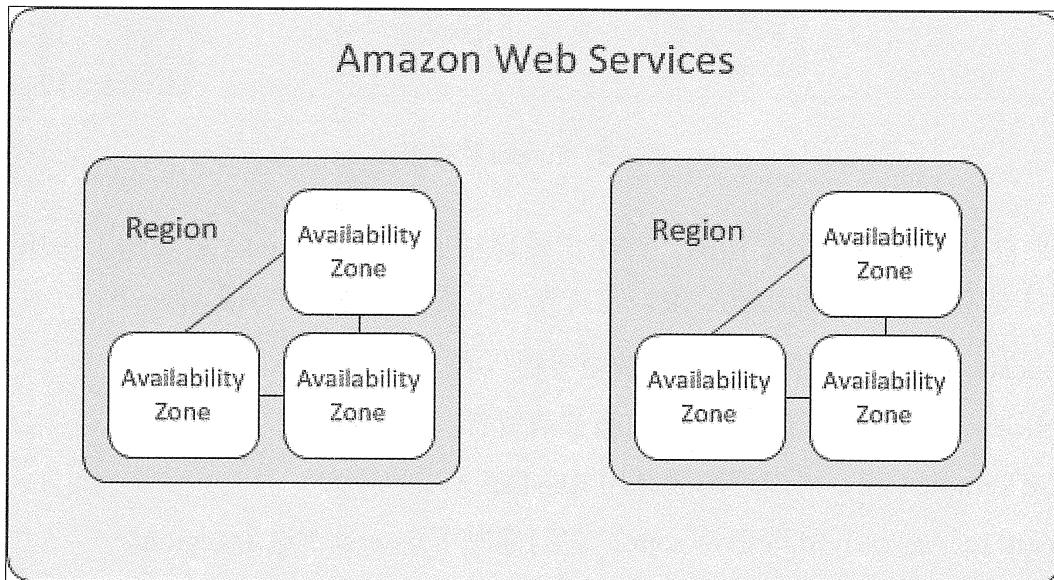


图 4-2 Region 与 Availability Zone

Region 和 Availability Zone 均是 AWS 的概念。其中，Region 表示 AWS 中的地理位置，每个 Region 都有多个 Availability Zone，各个 Region 之间完全隔离。AWS 通过这种方式实现了最大的容错和稳定性。

Spring Cloud 默认使用的 Region 是 us-east-1，在非 AWS 环境下，可以将 Availability Zone 理解成机房，将 Region 理解为跨机房的 Eureka 集群。

对 Region 和 Availability Zone 感兴趣的读者可前往 <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html> 进行扩展阅读。

理解 Region 和 Availability Zone 后，来分析一下 Eureka 的原理，Eureka 架构如图 4-3 所示。

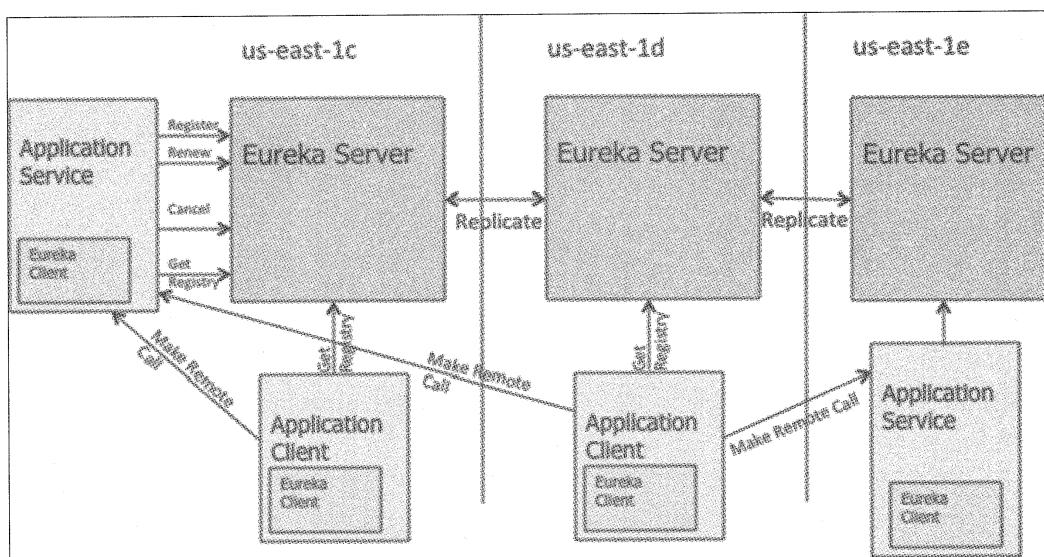


图 4-3 Eureka 架构图

图 4-3 来自 Eureka 官方的架构图，该图比较详细地描述了 Eureka 集群的工作原理。图中的组件非常多，概念也比较抽象，笔者先来用通俗易懂的文字翻译一下：

- Application Service 相当于本书中的服务提供者。
- Application Client 相当于本书中的服务消费者。
- Make Remote Call，可以理解成调用 RESTful API 的行为。
- us-east-1c、us-east-1d 等都是 zone，它们都属于 us-east-1 这个 region。

由图 4-3 可知，Eureka 包含两个组件：Eureka Server 和 Eureka Client，它们的作用如下：

- Eureka Server 提供服务发现的能力，各个微服务启动时，会向 Eureka Server 注册自己的信息（例如 IP、端口、微服务名称等），Eureka Server 会存储这些信息。
- Eureka Client 是一个 Java 客户端，用于简化与 Eureka Server 的交互。
- 微服务启动后，会周期性（默认 30 秒）地向 Eureka Server 发送心跳以续约自己的“租期”。
- 如果 Eureka Server 在一定时间内没有接收到某个微服务实例的心跳，Eureka Server 将会注销该实例（默认 90 秒）。

- 默认情况下，Eureka Server 同时也是 Eureka Client。多个 Eureka Server 实例，互相之间通过复制的方式，来实现服务注册表中数据的同步。
- Eureka Client 会缓存服务注册表中的信息。这种方式有一定的优势——首先，微服务无须每次请求都查询 Eureka Server，从而降低了 Eureka Server 的压力；其次，即使 Eureka Server 所有节点都宕掉，服务消费者依然可以使用缓存中的信息找到服务提供者并完成调用。

综上，Eureka 通过心跳检查、客户端缓存等机制，提高了系统的灵活性、可伸缩性和可用性。

4.4 编写 Eureka Server

本节来编写一个 Eureka Server。

1. 创建一个 ArtifactId 是 microservice-discovery-eureka 的 Maven 工程，并为项目添加以下依赖。

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

2. 编写启动类，在启动类上添加 @EnableEurekaServer 注解，声明这是一个 Eureka Server。

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

3. 在配置文件 application.yml 中添加以下内容。

```
server:
  port: 8761
eureka:
  client:
    registerWithEureka: false
```

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

简要讲解一下 application.yml 中的配置属性：

- eureka.client.registerWithEureka：表示是否将自己注册到 Eureka Server，默认为 true。由于当前应用就是 Eureka Server，故而设为 false。
- eureka.client.fetchRegistry：表示是否从 Eureka Server 获取注册信息，默认为 true。因为这是一个单点的 Eureka Server，不需要同步其他的 Eureka Server 节点的数据，故而设为 false。
- eureka.client.serviceUrl.defaultZone：设置与 Eureka Server 交互的地址，查询服务和注册服务都需要依赖这个地址。默认是http://localhost:8761/eureka；多个地址可使用，分隔。

这样一个 Eureka Server 就编写完成了。

测试

启动 Eureka Server，访问`http://localhost:8761/`，可看到如图 4-4 所示的界面。

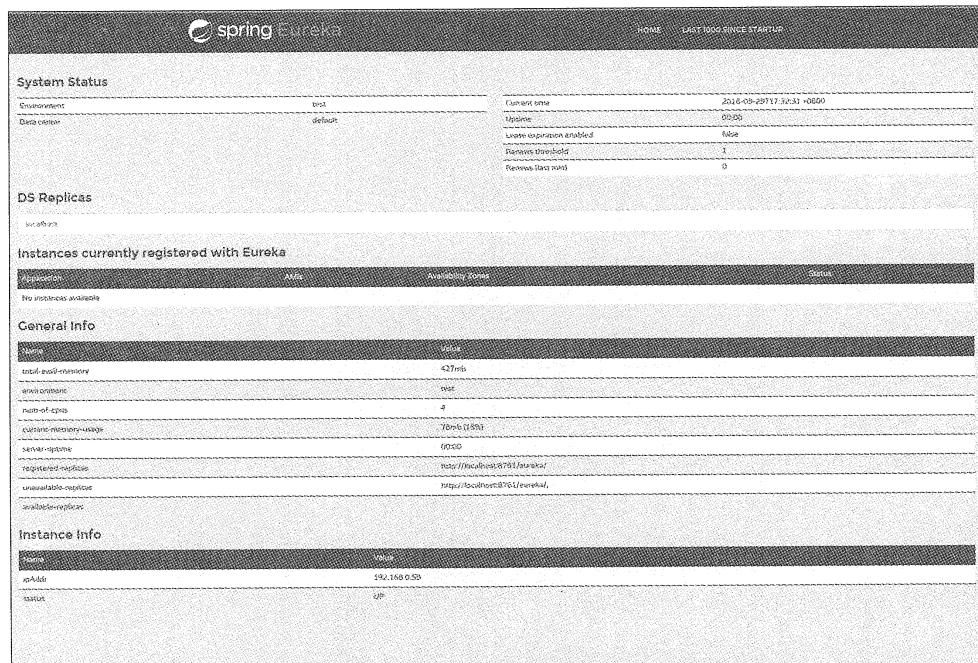


图 4-4 Eureka 首页

由图可知，Eureka Server 的首页展示了很多信息，例如当前实例的系统状态、注册到 Eureka Server 上的服务实例、常用信息、实例信息等。显然，当前还没有任何微服务实例被注册到 Eureka Server 上。

4.5 将微服务注册到 Eureka Server 上

本节将之前编写的用户微服务注册到 Eureka Server 上。

1. 复制项目 microservice-simple-provider-user，将 ArtifactId 修改为 microservice-provider-user。
2. 在 pom.xml 中添加以下依赖。

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

3. 在配置文件 application.yml 中添加以下配置。

```
spring:
  application:
    name: microservice-provider-user
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
```

其中，spring.application.name 用于指定注册到 Eureka Server 上的应用名称；eureka.instance.prefer-ip-address = true 表示将自己的 IP 注册到 Eureka Server。如不配置该属性或将其设置为 false，则表示注册微服务所在操作系统的 hostname 到 Eureka Server。

4. 编写启动类，在启动类上添加@EnableDiscoveryClient 注解，声明这是一个 Eureka Client。

```
@EnableDiscoveryClient
@SpringBootApplication
public class ProviderUserApplication {
    public static void main(String[] args) {
```

```

    SpringApplication.run(ProviderUserApplication.class, args);
}
}

```

也可以使用`@EnableEurekaClient`注解替代`@EnableDiscoveryClient`。在 Spring Cloud 中，服务发现组件有多种选择，例如 Zookeeper、Consul 等。`@EnableDiscoveryClient`为各种服务组件提供了支持，该注解是 spring-cloud-commons 项目的注解，是一个高度的抽象；而`@EnableEurekaClient`表明是 Eureka 的 Client，该注解是 spring-cloud-netflix 项目中的注解，只能与 Eureka 一起工作。当 Eureka 在项目的 classpath 中时，两个注解没有区别。

这样就可以将用户微服务注册到 Eureka Server 上了。同理，将电影微服务也注册到 Eureka Server 上，配置电影微服务的 `spring.application.name` 为 `microservice-consumer-movie`，详见本书配套代码中的 `microservice-consumer-movie` 项目。

测试

- 启动 `microservice-discovery-eureka`。
- 启动 `microservice-provider-user`。
- 启动 `microservice-consumer-movie`。
- 访问 `http://localhost:8761/`，可看到如图 4-5 的界面。

DS Replicas			
localhost			
Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
MICROSERVICE-CONSUMER-MOVIE	n/a (1)	{1}	UP (1) - QH-20160301NAV/microservice-consumer-movie:8010
MICROSERVICE-PROVIDER-USER	n/a (1)	{1}	UP (1) - QH-20160303NAV/microservice-provider-user:8000

图 4-5 Eureka Server 上的微服务列表

由图可知，此时用户微服务、电影微服务已经被注册到 Eureka Server 上了。

4.6 Eureka Server 的高可用

有分布式应用开发经验的读者应该能够看出，前文编写的单节点 Eureka Server 并不适合线上生产环境。Eureka Client 会定时连接 Eureka Server，获取服务注册表中的信息并缓存在本地。微服务在消费远程 API 时总是使用本地缓存中的数据。因此一般来说，即使

Eureka Server 发生宕机，也不会影响到服务之间的调用。但如果 Eureka Server 宕机时，某些微服务也出现了不可用的情况，Eureka Client 中的缓存若不被更新，就可能会影响到微服务的调用，甚至影响到整个应用系统的高可用性。因此，在生产环境中，通常会部署一个高可用的 Eureka Server 集群。

Eureka Server 可以通过运行多个实例并相互注册的方式实现高可用部署，Eureka Server 实例会彼此增量地同步信息，从而确保所有节点数据一致。事实上，节点之间相互注册是 Eureka Server 的默认行为，还记得前文编写单节点 Eureka Server 时，额外配置了 eureka.client.registerWithEureka=false、eureka.client.fetchRegistry=false 吗？

本节在前文的基础上，构建一个双节点 Eureka Server 集群。

1. 复制项目 microservice-discovery-eureka，将 ArtifactId 修改为 microservice-discovery-eureka-ha。
2. 配置系统的 hosts，Windows 系统的 hosts 文件路径是 C:\Windows\System32\drivers\etc\hosts；Linux 及 Mac OS 等系统的文件路径是 /etc/hosts。
127.0.0.1 peer1 peer2
3. 将 application.yml 修改如下：让两个节点的 Eureka Server 相互注册。

```
spring:  
  application:  
    name: microservice-discovery-eureka-ha  
---  
spring:  
  # 指定profile=peer1  
  profiles: peer1  
server:  
  port: 8761  
eureka:  
  instance:  
    # 指定当profile=peer1时，主机名是peer1  
    hostname: peer1  
  client:  
    serviceUrl:  
      # 将自己注册到peer2这个Eureka上面去  
      defaultZone: http://peer2:8762/eureka/  
---  
spring:  
  profiles: peer2
```

```

server:
  port: 8762
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone: http://peer1:8761/eureka/

```

如上，使用连字符（---）将该 application.yml 文件分为三段。第二段和第三段分别为 spring.properties 指定了一个值，该值表示它所在的那段内容应用在哪个 Profile 里。第一段由于并未指定 spring.profiles，因此这段内容会对所有 Profile 生效。

经过以上分析，不难理解，我们定义了 peer1 和 peer2 这两个 Profile。当应用以 peer1 这个 Profile 启动时，配置该 Eureka Server 的主机名为 peer1，并将其注册到 <http://peer2:8762/eureka/>；反之，当应用以 profile=peer2 时，Eureka Server 会注册到 peer1 节点的 Eureka Server。

测试

1. 打包项目，并使用以下命令启动两个 Eureka Server 节点。

```

java -jar microservice-discovery-eureka-ha-0.0.1-SNAPSHOT.jar --spring.
profiles.active=peer1
java -jar microservice-discovery-eureka-ha-0.0.1-SNAPSHOT.jar --spring.
profiles.active=peer2

```

通过 spring.profiles.active 指定使用哪个 profile 启动。

2. 访问 <http://peer1:8761>，会发现“registered-replicas”中已有 peer2 节点；同理，访问 <http://peer2:8762>，也能发现其中的“registered-replicas”有 peer1 节点，如图 4-6 所示。

将应用注册到 Eureka Server 集群上

以 microservice-provider-user 项目为例，只须修改 eureka.client.serviceUrl.defaultZone，配置多个 Eureka Server 地址，就可以将其注册到 Eureka Server 集群了。示例：

```

eureka:
  client:
    serviceUrl:
      defaultZone: http://peer1:8761/eureka/,http://peer2:8762/eureka/

```

The screenshot shows the Eureka Server interface. At the top, it displays 'DS Replicas' and 'peer2'. Below this, under 'Instances currently registered with Eureka', there is a table with columns: Application, AMIs, Availability Zones, and Status. One entry is listed: MICROSERVICE-DISCOVERY-EUREKA-HA, n/a (2) (2), Status: UP (2) - itmuch:microservice-discovery-eureka-ha:8761, itmuch:microservice-discovery-eureka-ha:8762. Under 'General Info', there is another table with columns: Name and Value. The entries include: total-avail-memory (409mb), environment (test), num-of-cpus (4), current-memory-usage (255mb (62%)), server-upptime (00:00), registered-replicas (http://peer2:8762/eureka/), unavailable-replicas (empty), and available-replicas (http://peer2:8762/eureka/).

图 4-6 高可用的 Eureka 首页

这样就可以将服务注册到 Eureka Server 集群上了。

当然，微服务即使只配置 Eureka Server 集群中的某个节点，也能正常注册到 Eureka Server 集群，因为多个 Eureka Server 之间的数据会相互同步。例如：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://peer1:8761/eureka/
```

正常情况下，这种方式与配置多个 Server 节点的效果是一样的。不过为适应某些极端场景，笔者建议在客户端配置多个 Eureka Server 节点。

4.7 为 Eureka Server 添加用户认证

在前面的示例中，Eureka Server 是允许匿名访问的，本节来构建一个需要登录才能访问的 Eureka Server。

1. 复制项目microservice-discovery-eureka，将 ArtifactId 修改为microservice-discovery-eureka-authenticating。
2. 在 pom.xml 中添加spring-boot-starter-security的依赖，该依赖为 Eureka Server 提供用户认证的能力。

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

3. 在 application.yml 中添加以下内容：

```
security:
  basic:
    enabled: true          # 开启基于HTTP basic的认证
    user:
      name: user           # 配置登录的账号是user
      password: password123 # 配置登录的密码是password123
```

这样就为 Eureka Server 添加了基于 HTTP basic 的认证。如果不设置这段内容，账号默认是 user，密码是一个随机值，该值会在启动时打印出来。



测试

1. 启动 microservice-discovery-eureka-authenticating。
2. 访问 <http://localhost:8761/> 需要身份验证的对话框，如图 4-7 所示。

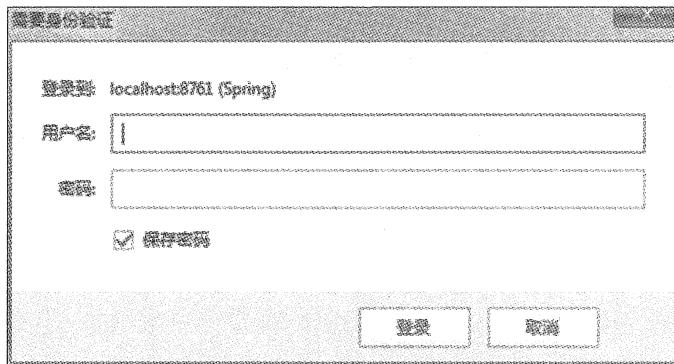


图 4-7 Eureka Server 登录界面

3. 输入账号 user、密码 password123，就可登录并访问 Eureka Server。

将微服务注册到需认证的 Eureka Server

如何才能将微服务注册到需认证的 Eureka Server 上呢？答案非常简单，只须将 eureka.client.serviceUrl.defaultZone 配置为 `http://user:password@EUREKA_HOST:EUREKA_PORT/eureka/` 这种形式，就可以将 HTTP basic 认证添加到 Eureka Client 了。因此，只须稍作修改，就可以将应用注册到本例的 Eureka Server 了：

```
eureka:  
  client:  
    serviceUrl:  
      defaultZone: http://user:password123@localhost:8761/eureka/
```

对于更复杂的需求，可创建一个类型为DiscoveryClientOptionalArgs的@Bean，并向其中注入ClientFilter。

4.8 Eureka 的元数据

Eureka 的元数据有两种，分别是标准元数据和自定义元数据。

标准元数据指的是主机名、IP 地址、端口号、状态页和健康检查等信息，这些信息都会被发布在服务注册表中，用于服务之间的调用。自定义元数据可以使用eureka.instance.metadata-map 配置，这些元数据可以在远程客户端中访问，但一般不会改变客户端的行为，除非客户端知道该元数据的含义。

下面笔者通过一个简单的示例，帮助大家更好地理解 Eureka 的元数据。

4.8.1 改造用户微服务

1. 复制项目microservice-provider-user，将 ArtifactId 修改为microservice-provider-user-my-metadata。
2. 修改 application.yml，使用eureka.instance.metadata-map 属性为该微服务添加应自定义的元数据：

```
eureka:  
  client:  
    serviceUrl:  
      defaultZone: http://localhost:8761/eureka/  
  instance:  
    prefer-ip-address: true  
    metadata-map:  
      # 自定义的元数据，key/value都可以随便写。  
      my-metadata: 我自定义的元数据
```

4.8.2 改造电影微服务

1. 复制项目microservice-consumer-movie，将 ArtifactId 修改为microservice-consumer-movie-understanding-metadata。

2. 修改 Controller。

```
@RestController
public class MovieController {
    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private DiscoveryClient discoveryClient;

    @GetMapping("/user/{id}")
    public User findById(@PathVariable Long id) {
        return this.restTemplate.getForObject("http://localhost:8000/" + id, User.class);
    }

    /**
     * 查询microservice-provider-user服务的信息并返回
     * @return microservice-provider-user服务的信息
     */
    @GetMapping("/user-instance")
    public List<ServiceInstance> showInfo() {
        return this.discoveryClient.getInstances("microservice-provider-user");
    }
}
```

使用 DiscoveryClient.getInstances(serviceId)，可查询指定微服务在 Eureka 上的实例列表。



测试

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user-my-metadata。
3. 启动 microservice-consumer-movie-understanding-metadata。
4. 访问 <http://localhost:8761/eureka/apps> 可查看 Eureka 的 metadata。
5. 访问 <http://localhost:8010/user-instance>，可以返回类似如下的内容。

```
[  
 {  
 "host": "192.168.1.106",
```

```
"port": 8000,
"metadata": {
    "my-metadata": "我自定义的元数据"
},
"secure": false,
"uri": "http://192.168.1.106:8000",
"instanceInfo": {
    "instanceId": "itmuch:microservice-provider-user:8000",
    "app": "MICROSERVICE-PROVIDER-USER",
    "appGroupName": null,
    "ipAddr": "192.168.1.106",
    "sid": "na",
    "homePageUrl": "http://192.168.1.106:8000/",
    "statusPageUrl": "http://192.168.1.106:8000/info",
    "healthCheckUrl": "http://192.168.1.106:8000/health",
    "metadata": {
        "my-metadata": "我自定义的元数据"
    },
    ...
},
"serviceId": "MICROSERVICE-PROVIDER-USER"
}
]
```

可以看到，使用 DiscoveryClient 的 API 获得了用户微服务的各种信息，其中包括了标准元数据和自定义元数据。例如 IP、端口等信息都是标准元数据，用于服务之间的调用；同时，自定义的元数据 my-metadata，也可通过客户端查询到，但是并不会改变客户端的行为。

4.9 Eureka Server 的 REST 端点

Eureka Server 提供了一些 REST 端点。非 JVM 的微服务可使用这些 REST 端点操作 Eureka，从而实现注册与发现。事实上，前文所讲的 Eureka Client 就是一个使用 Java 编写的、操作这些 REST 端点的类库。也可分析这些 REST 端点，编写其他语言的 Eureka Client。

表 4-1 是 Eureka 提供的 REST 端点，可以使用 XML 或者 JSON 与这些端点通信，默认是 XML。

表 4-1 Eureka 的 REST 端点一览表

Operation	HTTP action	Description
Register new application instance	POST /eureka/apps/appID	Input:JSON/XMLpayload HTTP Code: 204 on success
De-register application instance	DELETE /eureka/apps/appID/instanceID	HTTP Code: 200 on success
Send application instance heartbeat	PUT /eureka/apps/appID/instanceID	HTTP Code: 200 on success 404 if instanceID doesn't exist
Query for all instances	GET /eureka/apps	HTTP Code: 200 on success Output:JSON/XML
Query for all appID instances	GET /eureka/apps/appID	HTTP Code: 200 on success Output:JSON/XML
Query for a specific appID/instanceID	GET /eureka/apps/appID/instanceID	HTTP Code: 200 on success Output:JSON/XML
Query for a specific instanceID	GET /eureka/instances/instanceID	HTTP Code: 200 on success Output:JSON/XML
Take instance out of service	PUT /eureka/apps/appID/instanceID/status?value=OUT_OF_SERVICE	HTTP Code: 200 on success 500 on failure
Put instance back into service (remove override)	DELETE /eureka/apps/appID/instanceID/status?value=UP (The value=UP is optional, it is used as a suggestion for the fallback status due to removal of the override)	HTTP Code: 200 on success 500 on failure
Update metadata	PUT /eureka/apps/appID/instanceID/metadata?key=value	HTTP Code: 200 on success 500 on failure
Query for all instances under a particular vip address	GET /eureka/vips/vipAddress	HTTP Code: 200 on success Output:JSON/XML 404 if the vipAddress does not exist.
Query for all instances under a particular secure vip address	GET /eureka/svips/svipAddress	HTTP Code: 200 on success Output:JSON/XML 404 if the svipAddress does not exist.

表 4-1 中的 appID 是应用程序的名称，instanceID 是与实例相关联的唯一 ID。在 AWS 环境中，instanceID 表示微服务实例的实例 ID，在非 AWS 环境则表示实例的主机名。

示例

注册微服务到 Eureka Server 上

使用 REST 端点向 Eureka Server 注册微服务时，需要 POST 一个符合以下 XSD 的 XML (或 JSON) 请求体：

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xsd:element name="instance">
    <xsd:complexType>
      <xsd:all>
        <!-- hostName in ec2 should be the public dns name, within ec2 public dns
            name will always resolve to its private IP -->
        <xsd:element name="hostName" type="xsd:string" />
        <xsd:element name="app" type="xsd:string" />
        <xsd:element name="ipAddr" type="xsd:string" />
        <xsd:element name="vipAddress" type="xsd:string" />
        <xsd:element name="secureVipAddress" type="xsd:string" />
        <xsd:element name="status" type="statusType" />
        <xsd:element name="port" type="xsd:positiveInteger" minOccurs="0" />
        <xsd:element name="securePort" type="xsd:positiveInteger" />
        <xsd:element name="homePageUrl" type="xsd:string" />
        <xsd:element name="statusPageUrl" type="xsd:string" />
        <xsd:element name="healthCheckUrl" type="xsd:string" />
        <xsd:element ref="dataCenterInfo" minOccurs="1" maxOccurs="1" />
        <!-- optional -->
        <xsd:element ref="leaseInfo" minOccurs="0" />
        <!-- optional app specific metadata -->
        <xsd:element name="metadata" type="appMetadataType" minOccurs="0" />
      </xsd:all>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="dataCenterInfo">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="name" type="dcNameType" />
        <!-- metadata is only required if name is Amazon -->
```

```
<xsd:element name="metadata" type="amazonMetdataType" minOccurs="0" />
</xsd:all>
</xsd:complexType>
</xsd:element>

<xsd:element name="leaseInfo">
<xsd:complexType>
<xsd:all>
    <!-- (optional) if you want to change the length of lease - default if 90
        secs -->
    <xsd:element name="evictionDurationInSecs" minOccurs="0" type="
        xsd:positiveInteger" />
</xsd:all>
</xsd:complexType>
</xsd:element>

<xsd:simpleType name="dcNameType">
    <!-- Restricting the values to a set of value using 'enumeration' -->
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="MyOwn" />
        <xsd:enumeration value="Amazon" />
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="statusType">
    <!-- Restricting the values to a set of value using 'enumeration' -->
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="UP" />
        <xsd:enumeration value="DOWN" />
        <xsd:enumeration value="STARTING" />
        <xsd:enumeration value="OUT_OF_SERVICE" />
        <xsd:enumeration value="UNKNOWN" />
    </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="amazonMetdataType">
    <!-- From <a class="jive-link-external-small" href="http://docs.
        amazonwebservices.com/AWSEC2/latest/DeveloperGuide/index.html?AESDG-chapter
        -instancedata.html"
    -->
```

```
target="_blank">http://docs.amazonwebservices.com/AWSEC2/latest/
DeveloperGuide/index.html?AESDG-chapter-instancedata.html</a> -->
<xsd:all>
  <xsd:element name="ami-launch-index" type="xsd:string" />
  <xsd:element name="local-hostname" type="xsd:string" />
  <xsd:element name="availability-zone" type="xsd:string" />
  <xsd:element name="instance-id" type="xsd:string" />
  <xsd:element name="public-ipv4" type="xsd:string" />
  <xsd:element name="public-hostname" type="xsd:string" />
  <xsd:element name="ami-manifest-path" type="xsd:string" />
  <xsd:element name="local-ipv4" type="xsd:string" />
  <xsd:element name="hostname" type="xsd:string" />
  <xsd:element name="ami-id" type="xsd:string" />
  <xsd:element name="instance-type" type="xsd:string" />
</xsd:all>
</xsd:complexType>

<xsd:complexType name="appMetadataType">
  <xsd:sequence>
    <!-- this is optional application specific name, value metadata -->
    <xsd:any minOccurs="0" maxOccurs="unbounded" processContents="skip" />
  </xsd:sequence>
</xsd:complexType>

</xsd:schema>
```

下面来使用 REST 端点注册微服务。

1. 启动 microservice-discovery-eureka。
2. 编写一个符合上面 XSD 的 XML，命名为 rest-api-test.xml

```
<instance>
  <instanceId>itmuch:rest-api-test:9000</instanceId>
  <hostName>itmuch</hostName>
  <app>REST-API-TEST</app>
  <ipAddr>127.0.0.1</ipAddr>
  <vipAddress>rest-api-test</vipAddress>
  <secureVipAddress>rest-api-test</secureVipAddress>
  <status>UP</status>
  <port enabled="true">9000</port>
```

```

<securePort enabled="false">443</securePort>
<homePageUrl>http://127.0.0.1:9000/</homePageUrl>
<statusPageUrl>http://127.0.0.1:9000/info</statusPageUrl>
<healthCheckUrl>http://127.0.0.1:9000/health</healthCheckUrl>
<dataCenterInfo class="com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo">
    <name>MyOwn</name>
</dataCenterInfo>
</instance>

```

3. 使用 curl 命令测试。

```
cat ./rest-api-test.xml | curl -v -X POST -H "Content-type: application/xml" -d
@- http://localhost:8761/eureka/apps/rest-api-test
```

终端将会输出类似以下的内容：

```

* upload completely sent off: 644 out of 644 bytes
< HTTP/1.1 204
< Content-Type: application/xml
< Date: Mon, 19 Dec 2016 05:12:21 GMT

```

此时，查看 Eureka Server 首页，会发现微服务已经成功注册，如图 4-8 所示：

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
REST-API-TEST	n/a (1)	(1)	UP (1) - itmuch:rest-api-test:9000

图 4-8 Eureka Server 上的微服务列表

查看某微服务的所有实例

在浏览器上输入地址：<http://localhost:8761/eureka/apps/rest-api-test>，将会看到类似于如下的内容。

```

<application>
<name>REST-API-TEST</name>
<instance>
    <instanceId>itmuch:rest-api-test:9000</instanceId>
    <hostName>itmuch</hostName>
    <app>REST-API-TEST</app>
    <ipAddr>127.0.0.1</ipAddr>
    <status>UP</status>
    <overriddenStatus>UNKNOWN</overriddenStatus>

```

```
<port enabled="true">9000</port>
<securePort enabled="false">443</securePort>
<countryId>1</countryId>
<dataCenterInfo class="com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo">
    <name>MyOwn</name>
</dataCenterInfo>
<leaseInfo>
    <renewalIntervalInSecs>30</renewalIntervalInSecs>
    <durationInSecs>90</durationInSecs>
    <registrationTimestamp>1482124342257</registrationTimestamp>
    <lastRenewalTimestamp>1482124342257</lastRenewalTimestamp>
    <evictionTimestamp>0</evictionTimestamp>
    <serviceUpTimestamp>1482124341564</serviceUpTimestamp>
</leaseInfo>
<metadata class="java.util.Collections$EmptyMap"/>
<homePageUrl>http://127.0.0.1:9000/</homePageUrl>
<statusPageUrl>http://127.0.0.1:9000/info</statusPageUrl>
<healthCheckUrl>http://127.0.0.1:9000/health</healthCheckUrl>
<vipAddress>rest-api-test</vipAddress>
<secureVipAddress>rest-api-test</secureVipAddress>
<isCoordinatingDiscoveryServer>false</isCoordinatingDiscoveryServer>
<lastUpdatedTimestamp>1482124342257</lastUpdatedTimestamp>
<lastDirtyTimestamp>1482124341559</lastDirtyTimestamp>
<actionType>ADDED</actionType>
</instance>
</application>
```

从中，可以看到此微服务的所有实例，以及实例的详细信息。

注销微服务实例

将上面注册的微服务实例注销。

```
curl -v -X DELETE http://localhost:8761/eureka/apps/rest-api-test/itmuch:rest-api-test:9000
```

将会看到以下输出：

```
< HTTP/1.1 200
< Content-Type: application/xml
< Content-Length: 0
< Date: Tue, 20 Dec 2016 02:27:17 GMT
```

事实上，由于 rest-api-test 这个微服务并不存在，因此 Eureka Server 过一段时间后也会自动将该微服务注销。



- 本节仅挑选了两个常用端点进行测试，限于篇幅，其他端点请各位读者自行测试。
- 一些语言已有 Eureka Client 的开源实现，例如 Node.js 的 Eureka Client：<https://www.npmjs.com/package/eureka-js-client>。其他语言的 Eureka Client，读者可在 GitHub 或其他平台搜索相关实现。
- 本书使用的测试工具是 cURL，该工具在 Windows、Linux、Mac OS 均平台均有对应版本。
- 读者也可使用其他工具进行测试，例如 PostMan 等。图 4-9 是 PostMan 的测试界面，可使用它可视化地进行测试。

The screenshot shows the PostMan application interface. At the top, there are two tabs: 'localhost:8761/eure' (active) and 'https://localhost:8761/eure'. Below the tabs, the method is set to 'POST' and the URL is 'localhost:8761/eureka/apps/rest-api-test'. Under the 'Body' tab, the content type is set to 'raw' and the data is displayed as XML:

```
1 <instance>
2   <instanceId>itmuch:rest-api-test:9000</instanceId>
3   <hostName>itmuch</hostName>
4   <app>REST-API-TEST</app>
5   <ipAddr>127.0.0.1</ipAddr>
6   <vipAddress>rest-api-test</vipAddress>
7   <secureVipAddress>rest-api-test</secureVipAddress>
8   <status>UP</status>
9   <port enabled="true">9000</port>
10  <securePort enabled="false">443</securePort>
11  <homePageUrl>http://127.0.0.1:9000/</homePageUrl>
12  <statusPageUrl>http://127.0.0.1:9000/info</statusPageUrl>
13  <healthCheckUrl>http://127.0.0.1:9000/health</healthCheckUrl>
14
15  <dataCenterInfo class="com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo">
16    <name>MyOwn</name>
17  </dataCenterInfo>
18 </instance>
```

图 4-9 PostMan 测试界面

4.10 Eureka 的自我保护模式

本节来探讨 Eureka 的自我保护模式。进入自我保护模式最直观的体现，是 Eureka Server 首页输出的警告，如图 4-10 所示。

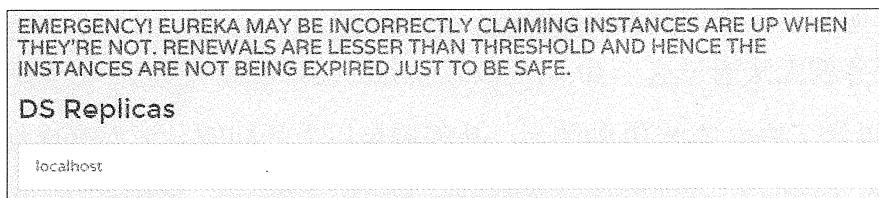


图 4-10 Eureka Server 自我保护模式界面

默认情况下，如果 Eureka Server 在一定时间内没有接收到某个微服务实例的心跳，Eureka Server 将会注销该实例（默认 90 秒）。但是当网络分区故障发生时，微服务与 Eureka Server 之间无法正常通信，以上行为可能变得非常危险了——因为微服务本身其实是健康的，此时本不应该注销这个微服务。

Eureka 通过“自我保护模式”来解决这个问题——当 Eureka Server 节点在短时间内丢失过多客户端时（可能发生了网络分区故障），那么这个节点就会进入自我保护模式。一旦进入该模式，Eureka Server 就会保护服务注册表中的信息，不再删除服务注册表中的数据（也就是不会注销任何微服务）。当网络故障恢复后，该 Eureka Server 节点会自动退出自我保护模式。

综上，自我保护模式是一种应对网络异常的安全保护措施。它的架构哲学是宁可同时保留所有微服务（健康的微服务和不健康的微服务都会保留），也不盲目注销任何健康的微服务。使用自我保护模式，可以让 Eureka 集群更加的健壮、稳定。

在 Spring Cloud 中，可以使用`eureka.server.enable-self-preservation = false`禁用自我保护模式。



- Eureka 官方关于自我保护模式的介绍：<https://github.com/Netflix/eureka/wiki/Understanding-Eureka-Peer-to-Peer-Communication>。
- Eureka 的 FAQ，其中讲到了自我保护模式：<https://github.com/Netflix/eureka/wiki/FAQ>。
- Eureka 与 Zookeeper 做服务发现的对比：<http://dockone.io/article/78>。
- Eureka 不注销任何微服务的讨论：<http://stackoverflow.com/questions/32616329/eureka-never-unregisters-a-service>。

4.11 多网卡环境下的 IP 选择

对于多网卡的服务器，各个微服务注册到 Eureka Server 上的 IP 要如何指定呢？

指定 IP 在某些场景下很有用。例如某台服务器有 eth0、eth1 和 eth2 三块网卡，但是只有 eth1 可以被其他的服务器访问；如果 Eureka Client 将 eth0 或者 eth2 注册到 Eureka Server 上，其他微服务就无法通过这个 IP 调用该微服务的接口。

Spring Cloud 提供了按需选择 IP 的能力，从而避免以上的问题。下面来详细讨论。

1. 忽略指定名称的网卡

例如：

```
spring:
  cloud:
    inetutils:
      ignored-interfaces:
        - docker0
        - veth.*
```

eureka:

```
  instance:
    prefer-ip-address: true
```

这样就可以忽略 docker0 网卡以及所有以 veth 开头的网卡。

2. 使用正则表达式，指定使用的网络地址

示例：

```
spring:
  cloud:
    inetutils:
      preferredNetworks:
        - 192.168
        - 10.0
```

eureka:

```
  instance:
    prefer-ip-address: true
```

3. 只使用站点本地地址

示例：

```
spring:
  cloud:
```

```
inetutils:  
useOnlySiteLocalInterfaces: true  
eureka:  
  instance:  
    prefer-ip-address: true
```

这样就可强制使用站点本地地址。

4. 手动指定 IP 地址

在某些极端场景下，可以手动指定注册到 Eureka Server 的微服务 IP。示例：

```
eureka:  
  instance:  
    prefer-ip-address: true  
    ip-address: 127.0.0.1
```



本节相关代码，详见本书配套代码中的 microservice-provider-user-ip 项目。



- 站点本地地址与链路本地地址：<https://4sysops.com/archives/ipv6-tutorial-part-6-site-local-addresses-and-link-local-addresses/>。
- 源码分析：<http://www.itmuch.com/spring-cloud-code-read/spring-cloud-code-read-eureka-registry-ip/>。

4.12 Eureka 的健康检查

先来看一下 Eureka 首页，如图 4-11 所示。

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
MICROSERVICE-PROVIDER-USER	n/a (1)	{1}	UP (1) - itmuch/microservice-provider-user:8000

图 4-11 Eureka Server 上的微服务列表

由图可见，在 Status 一栏有个 UP，表示应用程序状态正常。应用状态还有其他取值，例如 DOWN、OUT_OF_SERVICE、UNKNOWN 等。只有标记为“UP”的微服务会被请求。

前文讲过，Eureka Server 与 Eureka Client 之间使用心跳机制来确定 Eureka Client 的状态， 默认情况下，服务器端与客户端的心跳保持正常，应用程序就会始终保持“UP”状态。

以上机制并不能完全反映应用程序的状态。举个例子，微服务与 Eureka Server 之间的心跳正常，Eureka Server 认为该微服务“UP”；然而，该微服务的数据源发生了问题（例如因为网络抖动，连不上数据源），根本无法正常工作。

前文说过，Spring Boot Actuator 提供了 /health 端点，该端点可展示应用程序的健康信息。那么如何才能将该端点中的健康状态传播到 Eureka Server 呢？

要实现这一点，只须启用 Eureka 的健康检查。这样，应用程序就会将自己的健康状态传播到 Eureka Server。开启的方法非常简单，只须为微服务配置以下内容，就可以开启健康检查。

```
eureka:  
  client:  
    healthcheck:  
      enabled: true
```

某些场景下，可能希望更细粒度地控制健康检查，此时可实现 com.netflix.appinfo.HealthCheckHandler 接口。



- eureka.client.healthcheck.enabled=true 只能配置在 application.yml 中，如果配置在 bootstrap.yml（后文有详解）中，可能会导致一些不良的副作用，例如应用注册到 Eureka Server 上的状态是 UNKNOWN。
- 当 eureka.client.healthcheck.enabled=true 时，/pause 端点（该端点由 Spring Boot Actuator 提供，用于暂停应用）无法正常工作，详见：<https://github.com/spring-cloud/spring-cloud-netflix/issues/1571>，经笔者测试，发现当 eureka.client.healthcheck.enabled=true 时，请求 /pause 端点无法将应用在 Eureka Server 上的状态标记为 DOWN。由于该 Bug 尚未修复，建议读者留意。



Eureka 健康检查相关博客：<https://jmnarloch.wordpress.com/2015/09/02/spring-cloud-fixing-eureka-application-status/>。

使用 Ribbon 实现客户端侧负载均衡

经过前文的讲解，已经实现了微服务的注册与发现。启动各个微服务时，Eureka Client 会把自己的网络信息注册到 Eureka Server 上。世界似乎更美好了一些。

然而，这样的架构依然有一些问题，比如负载均衡。一般来说，在生产环境中，各个微服务都会部署多个实例。那么服务消费者要如何将请求分摊到多个服务提供者实例上呢？

5.1 Ribbon 简介

Ribbon 是 Netflix 发布的负载均衡器，它有助于控制 HTTP 和 TCP 客户端的行为。为 Ribbon 配置服务提供者地址列表后，Ribbon 就可基于某种负载均衡算法，自动地帮助服务消费者去请求。Ribbon 默认为我们提供了很多的负载均衡算法，例如轮询、随机等。当然，我们也可为 Ribbon 实现自定义的负载均衡算法。

在 Spring Cloud 中，当 Ribbon 与 Eureka 配合使用时，Ribbon 可自动从 Eureka Server 获取服务提供者地址列表，并基于负载均衡算法，请求其中一个服务提供者实例。图 5-1 展示了 Ribbon 与 Eureka 配合使用时的大致架构。



Ribbon 的 GitHub： <https://github.com/Netflix/ribbon>。

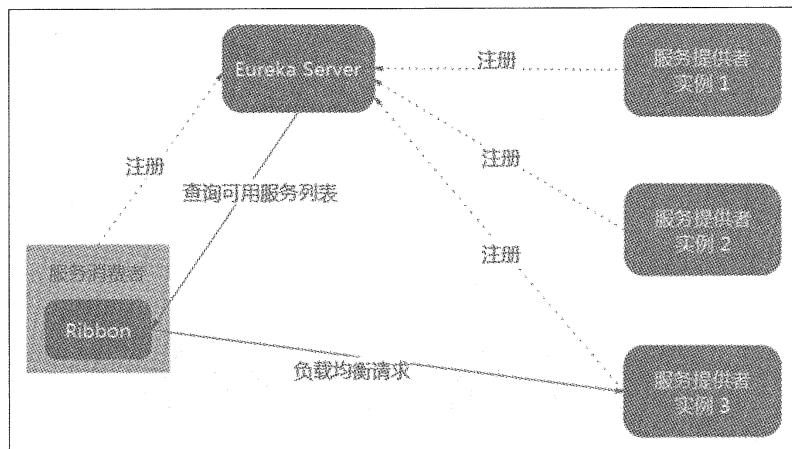


图 5-1 Eureka 与 Ribbon 配合使用架构图

5.2 为服务消费者整合 Ribbon

本节来为前文编写的电影微服务整合 Ribbon。

1. 复制项目 microservice-consumer-movie，将 ArtifactId 修改为 microservice-consumer-movie-ribbon。
2. 为项目引入 Ribbon 的依赖，Ribbon 的依赖是：

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

由于前文中已为电影微服务添加了 spring-cloud-starter-eureka，该依赖已经包含了 spring-cloud-starter-ribbon，所以无须再次引入。

3. 为 RestTemplate 添加 @LoadBalanced 注解。

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

回顾以前的代码，使用如下方法实例化 RestTemplate：

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

两者对比可以发现，只须添加注解`@LoadBalanced`注解，就可为`RestTemplate`整合`Ribbon`，使其具备负载均衡的能力。

4. 将 Controller 代码修改如下：

```
@RestController
public class MovieController {
    private static final Logger LOGGER = LoggerFactory.getLogger(MovieController.class);
    @Autowired
    private RestTemplate restTemplate;
    @Autowired
    private LoadBalancerClient loadBalancerClient;

    @GetMapping("/user/{id}")
    public User findById(@PathVariable Long id) {
        return this.restTemplate.getForObject("http://microservice-provider-user/" + id, User.class);
    }

    @GetMapping("/log-instance")
    public void logUserInstance() {
        ServiceInstance serviceInstance = this.loadBalancerClient.choose(
            "microservice-provider-user");
        // 打印当前选择的是哪个节点
        MovieController.LOGGER.info("{}:{}:{}",
            serviceInstance.getServiceId(),
            serviceInstance.getHost(), serviceInstance.getPort());
    }
}
```

由代码可知，我们将请求的地址改成了`http://microservice-provider-user/`。`microservice-provider-user`是用户微服务的虚拟主机名（virtual host name），当`Ribbon`和`Eureka`配合使用时，会自动将虚拟主机名映射成微服务的网络地址。在新增的`logUserInstance()`方法中可使用`LoadBalancerClient`的 API 更加直观地获取当前选择的用户微服务节点。



测试

1. 启动`microservice-discovery-eureka`。
2. 启动2个或更多`microservice-provider-user`实例。

3. 启动 microservice-consumer-movie-ribbon。
4. 访问 <http://localhost:8761>，结果如图 5-2 所示。

Instances currently registered with Eureka			
Application	Atts	Availability Zones	Status
MICROSERVICE-CONSUMER-MOVIE	n/a {1}	{1}	UP (1) - QH-20160301NAV7.microservice-consumer-movie:8010
MICROSERVICE-PROVIDER-USER	n/a {2}	{2}	UP (2) - QH-20160301NAV7.microservice-provider-user:8001 , QH-20160301NAV7.microservice-provider-user:8000

图 5-2 Eureka Server 上的微服务列表

5. 多次访问 <http://localhost:8010/user/1>，返回如下结果：

```
{
    "id": 1,
    "username": "account1",
    "name": "张三",
    "age": 20,
    "balance": 100
}
```

同时，两个用户微服务实例都会打印查询如下日志：

```
Hibernate: select user0_.id as id1_0_0_, user0_.age as age2_0_0_, user0_.
    balance as balance3_0_0_, user0_.name as name4_0_0_, user0_.username as
    username5_0_0_ from user user0_ where user0_.id=?
...

```

6. 多次访问 <http://localhost:8010/log-user-instance>，控制台会打印如下的日志：

```
2016-11-04 17:17:10.839 INFO 28596 --- [nio-8010-exec-9] c.i.c.s.user.con
troller.MovieController : microservice-provider-user:192.168.0.59:8001
2016-11-04 17:17:11.068 INFO 28596 --- [io-8010-exec-10] c.i.c.s.user.con
troller.MovieController : microservice-provider-user:192.168.0.59:8000
2016-11-04 17:17:11.258 INFO 28596 --- [nio-8010-exec-1] c.i.c.s.user.con
troller.MovieController : microservice-provider-user:192.168.0.59:8001
2016-11-04 17:17:11.496 INFO 28596 --- [nio-8010-exec-2] c.i.c.s.user.con
troller.MovieController : microservice-provider-user:192.168.0.59:8000
2016-11-04 17:17:11.688 INFO 28596 --- [nio-8010-exec-3] c.i.c.s.user.con
troller.MovieController : microservice-provider-user:192.168.0.59:8001
2016-11-04 17:17:12.015 INFO 28596 --- [nio-8010-exec-4] c.i.c.s.user.con
troller.MovieController : microservice-provider-user:192.168.0.59:8000
```

可以看到，此时请求会均匀分布到两个用户微服务节点上，说明已经实现了负载均衡。



- 虚拟主机名与虚拟 IP 非常类似，如果大家接触过 HAProxy 或 Heartbeat，理解虚拟主机名就非常容易了。如果无法理解虚拟主机名，可将其简单理解成为提供者的服务名称，因为在默认情况下，虚拟主机名和服务名称是一致的。当然，也可使用配置属性`eureka.instance.virtual-host-name`或者`eureka.instance.secure-virtual-host-name`指定虚拟主机名。
- 不能将`restTemplate.getForObject(...)`与`loadBalancerClient.choose(...)`写在同一个方法中，两者之间会有冲突——因为此时代码中的`restTemplate`实际上是一个 Ribbon 客户端，本身已经包含了“choose”的行为。



虚拟主机名不能包含“_”之类的字符，否则 Ribbon 在调用时会报异常。相关 Issue 详见：<https://github.com/spring-cloud/spring-cloud-netflix/issues/1582>。

5.3 使用 Java 代码自定义 Ribbon 配置

很多场景下，可能根据需要自定义 Ribbon 的配置，例如修改 Ribbon 的负载均衡规则等。Spring Cloud Camden 允许使用 Java 代码或属性自定义 Ribbon 的配置，两种方式是等价的。

本节来讨论如何使用 Java 代码自定义 Ribbon 的配置。

在 Spring Cloud 中，Ribbon 的默认配置如下（格式是 BeanType beanName: ClassName）：

- IClientConfig ribbonClientConfig: DefaultClientConfigImpl
- IRule ribbonRule: ZoneAvoidanceRule
- IPing ribbonPing: NoOpPing
- ServerList ribbonServerList: ConfigurationBasedServerList
- ServerListFilter ribbonServerListFilter: ZonePreferenceServerListFilter
- ILoadBalancer ribbonLoadBalancer: ZoneAwareLoadBalancer

简单说明一下，例如以下代码：

```
@Bean
@ConditionalOnMissingBean
public IRule ribbonRule(IClientConfig config) {
    ZoneAvoidanceRule rule = new ZoneAvoidanceRule();
    rule.initWithNiwisConfig(config);
    return rule;
}
// 来自org.springframework.cloud.netflix.ribbon.RibbonClientConfiguration
```

BeanType 是 IRule, beanName 是 ribbonRule, ClassName 是 ZoneAvoidanceRule, 这是一种根据服务提供者所在 Zone 的性能以及服务提供者可用性综合计算, 选择提供者节点的负载均衡规则。

在 Spring Cloud 中, Ribbon 默认的配置类是 RibbonClientConfiguration。也可使用一个 POJO 自定义 Ribbon 的配置 (自定义配置会覆盖默认配置)。这种配置是细粒度的, 不同的 Ribbon 客户端可以使用不同的配置。

下面来为名为 microservice-provider-user 的 Ribbon 客户端自定义配置。

1. 复制项目 microservice-consumer-movie-ribbon , 将 ArtifactId 修改为 microservice-consumer-movie-ribbon-customizing 。
2. 创建 Ribbon 的配置类。

```
/**
 * 该类为Ribbon的配置类
 * 注意：该类不应该在主应用程序上下文的@ComponentScan 中。
 * @author 周立
 */
@Configuration
public class RibbonConfiguration {
    @Bean
    public IRule ribbonRule() {
        // 负载均衡规则，改为随机
        return new RandomRule();
    }
}
```

3. 创建一个空类，并在其上添加 @Configuration 注解和 @RibbonClient 注解。

```
/**  
 * 使用RibbonClient，为特定name的Ribbon Client自定义配置。  
 * 使用@RibbonClient的configuration属性，指定Ribbon的配置类。  
 * 可参考的示例：  
 * http://spring.io/guides/gs/client-side-load-balancing/  
 * @author 周立  
 */  
  
@Configuration  
@RibbonClient(name = "microservice-provider-user", configuration =  
    RibbonConfiguration.class)  
public class TestConfiguration {  
}
```

由代码可知，使用 @RibbonClient 注解的 configuration 属性，即可自定义指定名称 Ribbon 客户端的配置。



测试

1. 启动 microservice-discovery-eureka。
2. 启动 2 个或更多 microservice-provider-user 实例。
3. 启动 microservice-consumer-movie-ribbon-customizing。
4. 多次访问 <http://localhost:8010/log-user-instance>，可获得类似于如下的日志：

```
2016-11-01 10:56:44.717 INFO 5624 --- [nio-8010-exec-2] c.i.c.study.user.  
service.MovieService : microservice-provider-user:192.168.0.59:8000  
2016-11-01 10:56:46.657 INFO 5624 --- [nio-8010-exec-6] c.i.c.study.user.  
service.MovieService : microservice-provider-user:192.168.0.59:8000  
2016-11-01 10:56:46.849 INFO 5624 --- [io-8010-exec-10] c.i.c.study.user.  
service.MovieService : microservice-provider-user:192.168.0.59:8001  
2016-11-01 10:56:46.996 INFO 5624 --- [nio-8010-exec-9] c.i.c.study.user.  
service.MovieService : microservice-provider-user:192.168.0.59:8000  
2016-11-01 10:56:47.148 INFO 5624 --- [nio-8010-exec-7] c.i.c.study.user.  
service.MovieService : microservice-provider-user:192.168.0.59:8001  
2016-11-01 10:56:47.312 INFO 5624 --- [nio-8010-exec-2] c.i.c.study.user.  
service.MovieService : microservice-provider-user:192.168.0.59:8001
```

此时请求会随机分布到两个用户微服务节点上，说明已经实现了 Ribbon 的自定义配置。



必须注意的是，本例中的RibbonConfiguration类不能包含在主应用程序上下文的 @ComponentScan 中，否则该类中的配置信息就被所有的@RibbonClient 共享。

因此，如果只想自定义某一个 Ribbon 客户端的配置，必须防止@Configuration 注解的类所在的包与 @ComponentScan 扫描的包重叠，或应显式指定 @ComponentScan 不扫描 @Configuration 类所在的包。

5.4 使用属性自定义 Ribbon 配置

从 Spring Cloud Netflix 1.2.0 开始（Spring Cloud Camden SR4 使用的版本是 1.2.4），Ribbon 支持使用属性自定义 Ribbon 客户端。这种方式比使用 Java 代码配置的方式更加方便。

支持的属性如下，配置的前缀是 <clientName>.ribbon.。

- NFLoadBalancerClassName：配置 ILoadBalancer 的实现类
- NFLoadBalancerRuleClassName：配置 IRule 的实现类
- NFLoadBalancerPingClassName：配置 IPing 的实现类
- NIWSServerListClassName：配置 ServerList 的实现类
- NIWSServerListFilterClassName：配置 ServerListFilter 的实现类

下面采用属性来修改名为 microservice-provider-user 的 Ribbon 客户端的负载均衡规则。

复制项目 microservice-consumer-movie-ribbon，将 ArtifactId 修改为 microservice-consumer-movie-ribbon-customizing-properties。在项目的 application.yml 中添加以下内容即可：

```
microservice-provider-user:
  ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
```

这样，就可以将负载均衡规则修改为随机。由代码不难看出，使用属性自定义的方式比用 Java 代码配置方便很多。



测试

1. 启动 microservice-discovery-eureka。
2. 启动两个或更多 microservice-provider-user 实例。
3. 启动 microservice-consumer-movie-ribbon-customizing-properties。

4. 多次访问`http://localhost:8010/log-user-instance`，查看日志，会发现此时请求依然会随机分布到两个用户微服务节点上。

5.5 脱离 Eureka 使用 Ribbon

在前文的示例中，是将 Ribbon 与 Eureka 配合使用的。但现实中可能不具备这样的条件，例如一些遗留的微服务，它们可能并没有注册到 Eureka Server 上，甚至根本不是使用 Spring Cloud 开发的，此时要想使用 Ribbon 实现负载均衡，要怎么办呢？

Ribbon 支持脱离 Eureka 使用，此时，架构如图 5-3 所示。

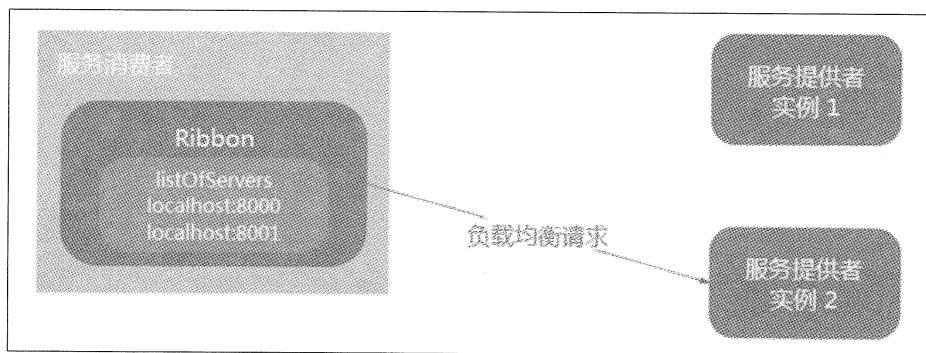


图 5-3 脱离 Eureka 使用 Ribbon 架构图

下面笔者通过一个简单的示例为大家讲解如何脱离 Eureka 使用 Ribbon。

1. 复制项目`microservice-consumer-movie-ribbon`，将`ArtifactId`修改为`microservice-consumer-movie-without-eureka`。
2. 为了让测试更具说服力，干脆为项目去掉 Eureka 的依赖`spring-cloud-starter-eureka`，只使用 Ribbon 的依赖`spring-cloud-starter-ribbon`。在项目的`pom.xml`中，找到：

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

修改为：

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

3. 去掉启动类上的@EnableDiscoveryClient注解。
4. 将 application.yml 改成如下：

```
server:
  port: 8010
spring:
  application:
    name: microservice-consumer-movie
microservice-provider-user:
  ribbon:
    listOfServers: localhost:8000,localhost:8001
```

其中，属性microservice-provider-user.ribbon.listOfServers用于为名为microservice-provider-user的 Ribbon 客户端设置请求的地址列表。



测试

1. 启动两个或更多 microservice-simple-provider-user 实例。

```
java -jar microservice-simple-provider-user-0.0.1-SNAPSHOT.jar
java -jar microservice-simple-provider-user-0.0.1-SNAPSHOT.jar --server.port=8001
```

2. 启动 microservice-consumer-movie-without-eureka。

3. 多次访问<http://localhost:8010/log-user-instance>，控制台打印如下日志。

```
2016-11-09 17:56:32.223 INFO 21844 --- [nio-8010-exec-2] c.i.c.s.user.controller.MovieController : microservice-provider-user:localhost:8001
2016-11-09 17:56:32.657 INFO 21844 --- [nio-8010-exec-3] c.i.c.s.user.controller.MovieController : microservice-provider-user:localhost:8000
2016-11-09 17:56:33.121 INFO 21844 --- [nio-8010-exec-4] c.i.c.s.user.controller.MovieController : microservice-provider-user:localhost:8001
2016-11-09 17:56:33.555 INFO 21844 --- [nio-8010-exec-5] c.i.c.s.user.controller.MovieController : microservice-provider-user:localhost:8000
2016-11-09 17:56:34.015 INFO 21844 --- [nio-8010-exec-6] c.i.c.s.user.controller.MovieController : microservice-provider-user:localhost:8001
2016-11-09 17:56:38.842 INFO 21844 --- [nio-8010-exec-7] c.i.c.s.user.controller.MovieController : microservice-provider-user:localhost:8000
```

由结果可知，尽管电影微服务和用户微服务此时并没有注册到 Eureka 上，Ribbon 仍可正常工作，请求依旧会分摊到两个用户微服务节点上。

使用 Feign 实现声明式 REST 调用

前文的示例中是使用 RestTemplate 实现 REST API 调用的，代码大致如下：

```
public User findById(Long id) {  
    return this.ribbonRestTemplate.getForObject("http://microservice-provider-user/"  
        + id, User.class);  
}
```

由代码可知，我们是使用拼接字符串的方式构造 URL 的，该 URL 只有一个参数。然而在现实中，URL 中往往有多个参数。如果这时还使用这种方式构造 URL，那么就会变得很低效，并且难以维护。举个例子，想要请求这样的 URL：

<http://localhost:8010/search?name=张三&username=account1&age=20>

如使用拼接字符串的方式构建请求 URL，那么代码可编写如下：

```
public User[] findById(String name, String username, Integer age) {  
    Map<String, Object> paramMap = Maps.newHashMap();  
    paramMap.put("name", name);  
    paramMap.put("username", username);  
    paramMap.put("age", age);  
    return this.restTemplate.getForObject("http://microservice-provider-user/search?  
        name={name}&username={username}&age={age}", User[].class,
```

```
    paramMap);  
}
```

在这里，URL 仅包含 3 个参数。如果 URL 更加复杂，例如有 10 个以上的参数，那么代码会变得难以维护。

如何解决这种问题呢？读者可带着这个疑问阅读本章。

6.1 Feign 简介

Feign 是 Netflix 开发的声明式、模板化的 HTTP 客户端，其灵感来自 Retrofit、JAXRS-2.0 以及 WebSocket。Feign 可帮助我们更加便捷、优雅地调用 HTTP API。

在 Spring Cloud 中，使用 Feign 非常简单——创建一个接口，并在接口上添加一些注解，代码就完成了。Feign 支持多种注解，例如 Feign 自带的注解或者 JAX-RS 注解等。

Spring Cloud 对 Feign 进行了增强，使 Feign 支持了 Spring MVC 注解，并整合了 Ribbon 和 Eureka，从而让 Feign 的使用更加方便。



Feign 的 GitHub： <https://github.com/OpenFeign/feign>。

6.2 为服务消费者整合 Feign

前文电影微服务是使用 RestTemplate（负载均衡通过整合 Ribbon 实现）调用 RESTful API 的。本节让电影微服务使用 Feign，实现声明式的 RESTful API 调用。

1. 复制项目 microservice-consumer-movie，将 ArtifactId 修改为 microservice-consumer-movie-feign。
2. 添加 Feign 的依赖。

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

3. 创建一个 Feign 接口，并添加 @FeignClient 注解。

```
@FeignClient(name = "microservice-provider-user")  
public interface UserFeignClient {
```

```
@RequestMapping(value = "/{id}", method = RequestMethod.GET)
public User findById(@PathVariable("id") Long id);
}
```

@FeignClient 注解中的 microservice-provider-user 是一个任意的客户端名称，用于创建 Ribbon 负载均衡器。在本例中，由于使用了 Eureka，所以 Ribbon 会把 microservice-provider-user 解析成 Eureka Server 服务注册表中的服务。当然，如果不想使用 Eureka，可使用 service.ribbon.listOfServers 属性配置服务器列表（详见 5.5 节）。

还可使用 url 属性指定请求的 URL（URL 可以是完整的 URL 或者主机名），例如 @FeignClient(name = "microservice-provider-user", url = "http://localhost:8000/")。

4. 修改 Controller 代码，让其调用 Feign 接口。

```
@RestController
public class MovieController {
    @Autowired
    private UserFeignClient userFeignClient;

    @GetMapping("/user/{id}")
    public User findById(@PathVariable Long id) {
        return this.userFeignClient.findById(id);
    }
}
```

5. 修改启动类，为其添加 @EnableFeignClients 注解，如下：

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

这样，电影微服务就可以用 Feign 去调用用户微服务的 API 了。

测试

1. 启动 microservice-discovery-eureka。
2. 启动 2 个或更多 microservice-provider-user 实例。

3. 启动 microservice-consumer-movie-feign。
4. 多次访问 <http://localhost:8010/user/1>，返回如下结果。

```
{
    "id": 1,
    "username": "account1",
    "name": "张三",
    "age": 20,
    "balance": 100
}
```

两个用户微服务实例都会打印类似如下的日志。

```
Hibernate: select user0_.id as id1_0_0_, user0_.age as age2_0_0_, user0_.
    balance as balance3_0_0_, user0_.name as name4_0_0_, user0_.username as
    username5_0_0_ from user user0_ where user0_.id=?
...
...
```

以上结果说明，我们不但实现了声明式的 REST API 调用，同时还实现了客户端侧的负载均衡。

6.3 自定义 Feign 配置

本节来讨论如何自定义 Feign 的配置。

在 Spring Cloud 中，Feign 的默认配置类是 `FeignClientsConfiguration`，该类定义了 Feign 默认使用的编码器、解码器、所使用的契约等。

Spring Cloud 允许通过注解 `@FeignClient` 的 `configuration` 属性自定义 Feign 的配置，自定义配置的优先级比 `FeignClientsConfiguration` 要高。

在 Spring Cloud 文档中可看到以下段落，描述了 Spring Cloud 提供的默认配置。另外，有的配置尽管没有提供默认值，但是 Spring 也会扫描其中列出的类型（也就是说，这部分配置也能自定义）。

Spring Cloud Netflix provides the following beans by default for feign (BeanType beanName: ClassName):

- Decoder `feignDecoder: ResponseEntityDecoder` (which wraps a `SpringDecoder`)
- Encoder `feignEncoder: SpringEncoder`

- Logger feignLogger: Slf4jLogger
- Contract feignContract: SpringMvcContract
- Feign.Builder feignBuilder: HystrixFeign.Builder
- Client feignClient: if Ribbon is enabled it is a LoadBalancerFeignClient, otherwise the default feign client is used.

The OkHttpClient and ApacheHttpClient feign clients can be used by setting feign.okhttp.enabled or feign.httpClient.enabled to true, respectively, and having them on the classpath.

Spring Cloud Netflix *does not* provide the following beans by default for feign, but still looks up beans of these types from the application context to create the feign client:

- Logger.Level
- Retryer
- ErrorDecoder
- Request.Options
- Collection<RequestInterceptor>

由此可知，在 Spring Cloud 中，Feign 默认使用的契约是 SpringMvcContract，因此它可以使用 Spring MVC 的注解。下面来自定义 Feign 的配置，让它使用 Feign 自带的注解进行工作。

1. 复制项目 microservice-consumer-movie-feign，将 ArtifactId 修改为 microservice-consumer-movie-feign-customizing。
2. 创建 Feign 的配置类。

```
/**  
 * 该类为Feign的配置类  
 * 注意：该不应该在主应用程序上下文的@ComponentScan中。  
 * @author 周立  
 */  
@Configuration  
public class FeignConfiguration {  
    /**  
     * 将契约改为feign原生的默认契约。这样就可以使用feign自带的注解了。  
     * @return 默认的feign契约  
    }
```

```

    */
@Bean
public Contract feignContract() {
    return new feign.Contract.Default();
}
}

```

3. Feign 接口修改为如下，使用@FeignClient 的 configuration 属性指定配置类，同时，将 findById 上的 Spring MVC 注解修改为 Feign 自带的注解。

```

/**
 * 使用@FeignClient的configuration属性，指定feign的配置类。
 * @author 周立
 */
@FeignClient(name = "microservice-provider-user", configuration =
    FeignConfiguration.class)
public interface UserFeignClient {
    /**
     * 使用feign自带的注解@RequestLine
     * @see https://github.com/OpenFeign/feign
     * @param id 用户id
     * @return 用户信息
     */
    @RequestLine("GET /{id}")
    public User findById(@Param("id") Long id);
}

```

类似地，还可自定义 Feign 的编码器、解码器、日志打印，甚至为 Feign 添加拦截器。例如，一些接口需要进行基于 Http Basic 的认证后才能调用，配置类可以这样写：

```

@Configuration
public class FooConfiguration {
    @Bean
    public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
        return new BasicAuthRequestInterceptor("user", "password");
    }
}

```



测试

- 启动 microservice-discovery-eureka。

2. 启动 microservice-provider-user。
3. 启动 microservice-consumer-movie-feign-customizing。
4. 访问 `http://localhost:8010/user/1`，可获得如下结果。

```
{  
    "id": 1,  
    "username": "account1",  
    "name": "张三",  
    "age": 20,  
    "balance": 100  
}
```

说明已实现 Feign 配置的自定义。



和 Ribbon 配置自定义一样，本例中的 `FeignConfiguration` 类也不能包含在主应用程序上下文的 `@ComponentScan` 中，否则该类中的配置信息就会被所有的 `@FeignClient` 共享。

因此，如果只想自定义某个 Feign 客户端的配置，必须防止 `@Configuration` 注解的类所在的包与 `@ComponentScan` 扫描的包重叠，或应显式指定 `@ComponentScan` 不扫描 `@Configuration` 类所在的包。

6.4 手动创建 Feign

在某些场景下，前文自定义 Feign 的方式满足不了需求，此时可使用 Feign Builder API (<https://github.com/OpenFeign/feign/#basics>) 手动创建 Feign。

本节围绕以下场景，为大家讲解如何手动创建 Feign。

- 用户微服务的接口需要登录后才能调用，并且对于相同的 API，不同角色的用户有不同的行为。
- 让电影微服务中的同一个 Feign 接口，使用不同的账号登录，并调用用户微服务的接口。

6.4.1 修改用户微服务

首先来改写用户微服务。

1. 复制项目 microservice-provider-user , 将 ArtifactId 修改为 microservice-provider-user-with-auth 。
2. 为项目添加以下依赖。

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

3. 创建 Spring Security 的配置类。

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // 所有的请求，都需要经过HTTP basic认证
        http.authorizeRequests().anyRequest().authenticated().and().httpBasic();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        // 明文编码器。这是一个不做任何操作的密码编码器，是Spring提供给我们做
        // 明文测试的
        // A password encoder that does nothing. Useful for testing where working
        // with plain text
        return NoOpPasswordEncoder.getInstance();
    }

    @Autowired
    private CustomUserDetailsService userDetailsService;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(this.userDetailsService).passwordEncoder(this.
```

```
        passwordEncoder());
    }

@Component
class CustomUserDetailsService implements UserDetailsService {
    /**
     * 模拟两个账户：
     * ① 账号是user，密码是password1，角色是user-role
     * ② 账号是admin，密码是password2，角色是admin-role
     */
    @Override
    public UserDetails loadUserByUsername(String username) throws
        UsernameNotFoundException {
        if ("user".equals(username)) {
            return new SecurityUser("user", "password1", "user-role");
        } else if ("admin".equals(username)) {
            return new SecurityUser("admin", "password2", "admin-role");
        } else {
            return null;
        }
    }
}

class SecurityUser implements UserDetails {
    private static final long serialVersionUID = 1L;

    public SecurityUser(String username, String password, String role) {
        super();
        this.username = username;
        this.password = password;
        this.role = role;
    }

    public SecurityUser() {
    }

    private Long id;
    private String username;
    private String password;
```

```

private String role;

@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    Collection<GrantedAuthority> authorities = new ArrayList<GrantedAuthority>();
    SimpleGrantedAuthority authority = new SimpleGrantedAuthority(this.role);
    authorities.add(authority);
    return authorities;
}

...
// 省略实现UserDetails类的方法、getters、setters
}
}
}

```

代码中模拟了两个账号： user 和 admin，它们的密码分别是 password1 和 password2，角色分别是 user-role 和 admin-role。

4. 修改 Controller，在其中打印当前登录的用户信息。

```

@RestController
public class UserController {
    @Autowired
    private UserRepository userRepository;
    private static final Logger LOGGER = LoggerFactory.getLogger(UserController.
        class);

    @GetMapping("/{id}")
    public User findById(@PathVariable Long id) {
        Object principal = SecurityContextHolder.getContext().getAuthentication().
            getPrincipal();
        if (principal instanceof UserDetails) {
            UserDetails user = (UserDetails) principal;
            Collection<? extends GrantedAuthority> collection = user.getAuthorities();
            for (GrantedAuthority c : collection) {
                // 打印当前登录用户的信息
                UserController.LOGGER.info("当前用户是{}，角色是{}", user.getUsername(),
                    c.getAuthority());
            }
        } else {
            // do other things
        }
    }
}

```

```
        }
        User findOne = this.userRepository.findOne(id);
        return findOne;
    }
}
```

用户微服务测试

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user-with-auth。
3. 访问`http://localhost:8000/1`，将会弹出登录对话框，如图 6-1 所示。



图 6-1 用户微服务登录对话框

4. 使用 user/password1 登录，可看到类似如下的日志。

```
[nio-8000-exec-5] c.i.c.study.controller.UserController : 当前用户是user，角  
色是user-role
```

5. 使用 admin/password2 登录，可看到类似如下的日志。

```
[nio-8000-exec-9] c.i.c.study.controller.UserController : 当前用户是admin，  
角色是admin-role
```

6.4.2 修改电影微服务

修改好用户微服务后，接下来修改电影微服务。

1. 复制项目 microservice-consumer-movie-feign，将 ArtifactId 修改为 microservice-consumer-movie-feign-manual。
2. 去掉 Feign 接口 UserFeignClient 上的 @FeignClient 注解

3. 去掉启动类上的@EnableFeignClients 注解。
4. 修改 Controller 如下：

```
@Import(FeignClientsConfiguration.class)
@RestController
public class MovieController {
    private UserFeignClient userUserFeignClient;

    private UserFeignClient adminUserFeignClient;

    @Autowired
    public MovieController(Decoder decoder, Encoder encoder, Client client,
        Contract contract) {
        // 这边的decoder、encoder、client、contract，可以Debug看看是什么实例
        this.userUserFeignClient = Feign.builder().client(client).encoder(encoder).
            decoder(decoder).contract(contract)
            .requestInterceptor(new BasicAuthRequestInterceptor("user", "password1"))
            .target(UserFeignClient.class, "http://microservice-provider-user/")
            );
        this.adminUserFeignClient = Feign.builder().client(client).encoder(encoder).
            decoder(decoder).contract(contract)
            .requestInterceptor(new BasicAuthRequestInterceptor("admin", "password2"))
            )
            .target(UserFeignClient.class, "http://microservice-provider-user/");
    }

    @GetMapping("/user-user/{id}")
    public User findByIdUser(@PathVariable Long id) {
        return this.userUserFeignClient.findById(id);
    }

    @GetMapping("/user-admin/{id}")
    public User findByIdAdmin(@PathVariable Long id) {
        return this.adminUserFeignClient.findById(id);
    }
}
```

其中，@Import 导入的FeignClientsConfiguration 是 Spring Cloud 为 Feign 默认提供的配置类。

userUserFeignClient 登录账号 user, adminUserFeignClient 登录账号 admin, 它们使用的是同一个接口 UserFeignClient。



测试

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user-with-auth。
3. 启动 microservice-consumer-movie-feign-manual。
4. 访问`http://localhost:8010/user-user/1`, 可以正常获得查询结果, 同时可以看到用户微服务打印类似以下的日志:
[nio-8000-exec-5] c.i.c.study.controller.UserController : 当前用户是user, 角色是user-role
5. 访问`http://localhost:8010/user-admin/1`, 可以正常获得查询结果, 同时可以看到用户微服务打印类似以下的日志:
[nio-8000-exec-5] c.i.c.study.controller.UserController : 当前用户是admin, 角色是admin-role

由测试不难发现, 手动创建 Feign 的方式更加灵活。

6.5 Feign 对继承的支持

Feign 支持继承。使用继承, 可将一些公共操作分组到一些父接口中, 从而简化 Feign 的开发, 以下是个简单的例子。

基础接口: UserService.java

```
public interface UserService {  
    @RequestMapping(method = RequestMethod.GET, value = "/users/{id}")  
    User getUser(@PathVariable("id") long id);  
}
```

服务提供者 Controller: UserResource.java

```
@RestController  
public class UserResource implements UserService {  
    //...  
}
```

服务消费者：UserClient.java

```
@FeignClient("users")
public interface UserClient extends UserService {
}
```



尽管 Feign 的继承可帮助我们进一步简化 Feign 的开发，但 Spring Cloud 官方指出——通常情况下，不建议在服务器端与客户端之间共享接口，因为这种方式造成了服务器端与客户端代码的紧耦合。并且，Feign 本身并不使用 Spring MVC 的工作机制（方法参数映射不被继承）。



笔者认为，应客观看待“紧耦合”与“方便性”，并在权衡利弊后作出取舍——放弃开发的方便性或接受代码的紧耦合。以下几篇帖子对“紧耦合”与“方便性”进行了深入的讨论，有兴趣的读者可前往查看：

- <https://github.com/spring-cloud/spring-cloud-netflix/issues/951>
- <https://github.com/spring-cloud/spring-cloud-netflix/issues/659>
- <https://github.com/spring-cloud/spring-cloud-netflix/issues/646>
- <https://jmnarlock.wordpress.com/2015/08/19/spring-cloud-designing-feign-client/>

6.6 Feign 对压缩的支持

一些场景下，可能需要对请求或响应进行压缩，此时可使用以下属性启用 Feign 的压缩功能。

```
feign.compression.request.enabled=true
feign.compression.response.enabled=true
```

对于请求的压缩，Feign 还提供了更为详细的设置，例如：

```
feign.compression.request.enabled=true
feign.compression.request.mime-types=text/xml,application/xml,application/json
feign.compression.request.min-request-size=2048
```

其中，`feign.compression.request.mime-types`用于支持的媒体类型列表，默认是 `text/xml`、`application/xml` 以及 `application/json`。

feign.compression.request.min-request-size 用于设置请求的最小阈值，默认是 2048。



该特性在 Spring Cloud Camden SR4 中并不生效。笔者已向官方反馈该问题，相信未来的版本中很快会被解决。详见：<https://github.com/spring-cloud/spring-cloud-netflix/issues/1580>。

6.7 Feign 的日志

很多场景下，需要了解 Feign 处理请求的具体细节，那么如何满足这种需求呢？

Feign 对日志的处理非常灵活，可为每个 Feign 客户端指定日志记录策略，每个 Feign 客户端都会创建一个 logger。默认情况下，logger 的名称是 Feign 接口的完整类名。需要注意的是，Feign 的日志打印只会对 DEBUG 级别做出响应。

我们可为每个 Feign 客户端配置各自的 Logger.Level 对象，告诉 Feign 记录哪些日志。Logger.Level 的值有以下选择。

- NONE：不记录任何日志（默认值）。
- BASIC：仅记录请求方法、URL、响应状态代码以及执行时间。
- HEADERS：记录 BASIC 级别的基础上，记录请求和响应的 header。
- FULL：记录请求和响应的 header，body 和元数据。

下面来为前面编写的 UserFeignClient 添加日志打印，将它的日志级别设置为 FULL。

1. 复制项目 microservice-consumer-movie-feign，将 ArtifactId 修改为 microservice-consumer-movie-feign-logging。
2. 编写 Feign 配置类：

```
@Configuration
public class FeignLogConfiguration {
    @Bean
    Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }
}
```

3. 修改 Feign 的接口，指定配置类：

```
@FeignClient(name = "microservice-provider-user", configuration =
    FeignLogConfiguration.class)
public interface UserFeignClient {
    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public User findById(@PathVariable("id") Long id);
}
```

4. 在 application.yml 中添加以下内容，指定 Feign 接口的日志级别为 DEBUG：

```
logging:
  level:
    com.itmuch.cloud.study.user.feign.UserFeignClient: DEBUG # 将Feign接口的日
    志级别设置成DEBUG，因为Feign的Logger.Level只对DEBUG作出响应。
```



测试

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user。
3. 启动 microservice-consumer-movie-feign-logging。
4. 访问 <http://localhost:8010/user/1>，可以看到类似于如下的日志。

```
2016-11-13 19:16:01.150 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient : [UserFeignClient#findById] ---> GET http://micro
service-provider-user/1 HTTP/1.1
2016-11-13 19:16:01.150 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient : [UserFeignClient#findById] ---> END HTTP (0-byte
body)
2016-11-13 19:16:01.164 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient : [UserFeignClient#findById] <--- HTTP/1.1 200 (13
ms)
2016-11-13 19:16:01.165 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient : [UserFeignClient#findById] content-type: applica
tion/json;charset=UTF-8
2016-11-13 19:16:01.165 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient : [UserFeignClient#findById] date: Sun, 13 Nov 2016
11:16:01 GMT
2016-11-13 19:16:01.165 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient : [UserFeignClient#findById] transfer-encoding:
```

```
chunked
2016-11-13 19:16:01.165 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient : [UserFeignClient#findById] x-application-context
: microservice-provider-user:8000
2016-11-13 19:16:01.165 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient : [UserFeignClient#findById]
2016-11-13 19:16:01.167 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient : [UserFeignClient#findById] {"id":1,"username":
"account1","name":"张三","age":20,"balance":100.00}
2016-11-13 19:16:01.167 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient : [UserFeignClient#findById] <--- END HTTP (72-byte
body)
```

可以看到，Feign 已将请求的各种细节非常详细地记录了下来。

5. 将日志 Feign 的日志级别改为 BASIC，重启项目后再次访问`http://localhost:8010/user/1`，日志如下：

```
2016-11-13 19:18:42.562 DEBUG 20176 --- [provider-user-4] c.i.c.study.user.
feign.UserFeignClient : [UserFeignClient#findById] ---> GET http://micro
service-provider-user/1 HTTP/1.1
2016-11-13 19:18:42.582 DEBUG 20176 --- [provider-user-4] c.i.c.study.user.
feign.UserFeignClient : [UserFeignClient#findById] <--- HTTP/1.1 200 (20
ms)
```

此时，只打印了请求方法、请求的 URL 和相应状态码和响应的时间。

6.8 使用 Feign 构造多参数请求

本节探讨如何使用 Feign 构造多参数的请求，以 GET 以及 POST 方法的请求为例，其他方法（例如 DELETE、PUT 等）的请求原理相通，读者可自行研究。

6.8.1 GET 请求多参数的 URL

假设请求的 URL 包含多个参数，例如`http://microservice-provider-user/get?id=1&username=张三`，要如何构造呢？

我们知道，Spring Cloud 为 Feign 添加了 Spring MVC 的注解支持，那么不妨按照 Spring MVC 的写法尝试一下：

```
@FeignClient("microservice-provider-user")
public interface UserFeignClient {
    @RequestMapping(value = "/get", method = RequestMethod.GET)
    public User get0(User user);
}
```

然而，这种写法并不正确，控制台会输出类似如下的异常。

```
feign.FeignException: status 405 reading UserFeignClient#get0(User); content:
{"timestamp":1482676142940,"status":405,"error":"Method Not Allowed","exception":"
    org.springframework.web.HttpRequestMethodNotSupportedException","message":"
    Request method 'POST' not supported","path":"/get"}
```

由异常可知，尽管指定了 GET 方法，Feign 依然会使用 POST 方法发送请求。

正确写法如下：

- 方法一

```
@FeignClient(name = "microservice-provider-user")
public interface UserFeignClient {
    @RequestMapping(value = "/get", method = RequestMethod.GET)
    public User get1(@RequestParam("id") Long id, @RequestParam("username") String
        username);
}
```

这是最为直观的方式，URL 有几个参数，Feign 接口中的方法就有几个参数。使用@RequestParam注解指定请求的参数是什么。

- 方法二

多参数的 URL 也可使用 Map 来构建。当目标 URL 参数非常多时，可使用这种方式简化 Feign 接口的编写。

```
@FeignClient(name = "microservice-provider-user")
public interface UserFeignClient {
    @RequestMapping(value = "/get", method = RequestMethod.GET)
    public User get2(@RequestParam Map<String, Object> map);
}
```

在调用时，可使用类似以下的代码。

```
public User get(String username, String password) {
    HashMap<String, Object> map = Maps.newHashMap();
    map.put("id", "1");
```

```
map.put("username", "张三");
return this.userFeignClient.get2(map);
}
```

6.8.2 POST 请求包含多个参数

下面来讨论如何使用 Feign 构造包含多个参数的 POST 请求。假设服务提供者的 Controller 是这样编写的：

```
@RestController
public class UserController {
    @PostMapping("/post")
    public User post(@RequestBody User user) {
        ...
    }
}
```

要如何使用 Feign 去请求呢？答案非常简单，示例：

```
@FeignClient(name = "microservice-provider-user")
public interface UserFeignClient {
    @RequestMapping(value = "/post", method = RequestMethod.POST)
    public User post(@RequestBody User user);
}
```



- 本节相关代码，详见本书配套代码中的 microservice-provider-user-multiple-params 项目和 microservice-consumer-movie-feign-multiple-params 项目。
- 除本节讲解的方式外，还可编写自己的编码器来构造多参数的请求，但这种方式编码成本较高，代码可重用性较低。故此，本书不再赘述。

7 使用 Hystrix 实现微服务的容错处理

至此，已用 Eureka 实现了微服务的注册与发现，Ribbon 实现了客户端侧的负载均衡，Feign 实现了声明式的 API 调用。

本章讨论如何使用 Hystrix 实现微服务的容错。

7.1 实现容错的手段

如果服务提供者响应非常缓慢，那么消费者对提供者的请求就会被强制等待，直到提供者响应或超时。在高负载场景下，如果不作任何处理，此类问题可能会导致服务消费者的资源耗竭甚至整个系统的崩溃。例如，曾经发生过一个案例——某电子商务网站在一个黑色星期五发生过载。过多的并发请求，导致用户支付的请求延迟很久都没有响应，在等待很长时间后最终失败。支付失败又导致用户重新刷新页面并再次尝试支付，进一步增加了服务器的负载，最终整个系统都崩溃了。

当依赖的服务不可用时，服务自身会不会被拖垮？这是我们要考虑的问题。

7.1.1 雪崩效应

微服务架构的应用系统通常包含多个服务层。微服务之间通过网络进行通信，从而支撑起整个应用系统，因此，微服务之间难免存在依赖关系。我们知道，任何微服务都并非100%可用，网络往往也很脆弱，因此难免有些请求会失败。

我们常把“基础服务故障”导致“级联故障”的现象称为雪崩效应。雪崩效应描述的是提供者不可用导致消费者不可用，并将不可用逐渐放大的过程。

如图7-1，A作为服务提供者（基础服务），B为A的服务消费者，C和D是B的服务消费者。当A不可用引起了B的不可用，并将不可用像滚雪球一样放大到C和D时，雪崩效应就形成了。

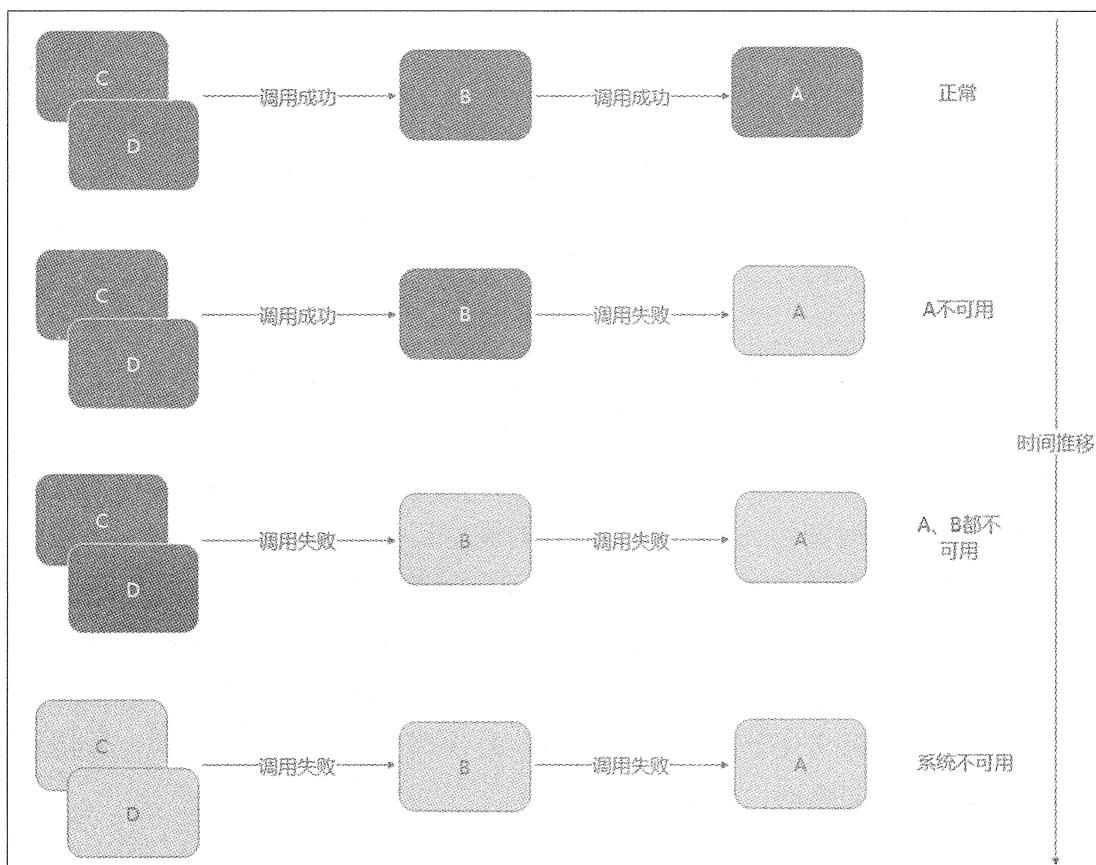


图 7-1 雪崩效应形成过程

7.1.2 如何容错

要想防止雪崩效应，必须有一个强大的容错机制。该容错机制需实现以下两点。

- 为网络请求设置超时

必须为网络请求设置超时。正常情况下，一个远程调用一般在几十毫秒内就能得到响应了。如果依赖的服务不可用或者网络有问题，那么响应时间就会变得很长（几十秒）。

通常情况下，一次远程调用对应着一个线程/进程。如果响应太慢，这个线程/进程就得不到释放。而线程/进程又对应着系统资源，如果得不到释放的线程/进程越积越多，资源就会逐渐被耗尽，最终导致服务的不可用。

因此，必须为每个网络请求设置超时，让资源尽快释放。

- 使用断路器模式

试想一下，如果家里没有断路器，当电流过载时（例如功率过大、短路等），电路不断开，电路就会升温，甚至可能烧断电路、引发火灾。使用断路器，电路一旦过载就会跳闸，从而可以保护电路的安全。在电路超载的问题被解决后，只须关闭断路器，电路就可以恢复正常。

同理，如果对某个微服务的请求有大量超时（常常说明该微服务不可用），再去让新的请求访问该服务已经没有任何意义，只会无谓消耗资源。例如，设置了超时时间为 1 秒，如果短时间内有大量的请求无法在 1 秒内得到响应，就没有必要再去请求依赖的服务了。

断路器可理解为对容易导致错误的操作的代理。这种代理能够统计一段时间内调用失败的次数，并决定是正常请求依赖的服务还是直接返回。

断路器可以实现快速失败，如果它在一段时间内检测到许多类似的错误（例如超时），就会在之后的一段时间内，强迫对该服务的调用快速失败，即不再请求所依赖的服务。这样，应用程序就无须再浪费 CPU 时间去等待长时间的超时。

断路器也可自动诊断依赖的服务是否已经恢复正常。如果发现依赖的服务已经恢复正常，那么就会恢复请求该服务。使用这种方式，就可以实现微服务的“自我修复”——当依赖的服务不正常时打开断路器时快速失败，从而防止雪崩效应；当发现依赖的服务恢复正常时，又会恢复请求。

断路器状态转换的逻辑如图 7-2 所示，简单讲解一下。

- 正常情况下，断路器关闭，可正常请求依赖的服务。
- 当一段时间内，请求失败率达到一定阈值（例如错误率达到 50%，或 100 次/分钟等），断路器就会打开。此时，不会再去请求依赖的服务。

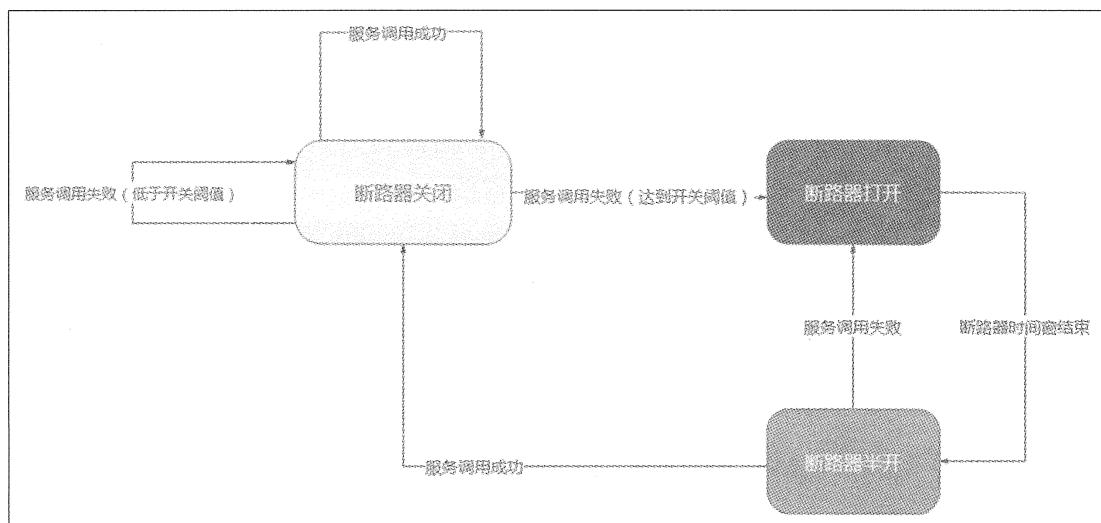


图 7-2 断路器状态转换图

- 断路器打开一段时间后，会自动进入“半开”状态。此时，断路器可允许一个请求访问依赖的服务。如果该请求能够调用成功，则关闭断路器；否则继续保持打开状态。

综上，我们可通过以上两点机制保护应用，从而防止雪崩效应并提升应用的可用性。

7.2 使用 Hystrix 实现容错

Hystrix 是一个实现了超时机制和断路器模式的工具类库。先来了解什么是 Hystrix。

7.2.1 Hystrix 简介

Hystrix 是由 Netflix 开源的一个延迟和容错库，用于隔离访问远程系统、服务或者第三方库，防止级联失败，从而提升系统的可用性与容错性。Hystrix 主要通过以下几点实现延迟和容错。

- 包裹请求：使用 HystrixCommand（或 HystrixObservableCommand）包裹对依赖的调用逻辑，每个命令在独立线程中执行。这使用到了设计模式中的“命令模式”。
- 跳闸机制：当某服务的错误率超过一定阈值时，Hystrix 可以自动或者手动跳闸，停止请求该服务一段时间。
- 资源隔离：Hystrix 为每个依赖都维护了一个小型的线程池（或者信号量）。如果该线程池已满，发往该依赖的请求就被立即拒绝，而不是排队等候，从而加速失败判定。

- 监控：Hystrix 可以近乎实时地监控运行指标和配置的变化，例如成功、失败、超时、以及被拒绝的请求等。
- 回退机制：当请求失败、超时、被拒绝，或当断路器打开时，执行回退逻辑。回退逻辑可由开发人员自行提供，例如返回一个缺省值。
- 自我修复：断路器打开一段时间后，会自动进入“半开”状态。断路器打开、关闭、半开的逻辑转换，前面已经详细探讨过了，本节不再赘述。



- Hystrix 的 GitHub：<https://github.com/Netflix/Hystrix>。
- 命令模式：https://en.wikipedia.org/wiki/Command_pattern。

7.2.2 通用方式整合 Hystrix

在 Spring Cloud 中，整合 Hystrix 非常方便。以项目 microservice-consumer-movie-ribbon 为例，我们来为它整合 Hystrix。

1. 复制项目 microservice-consumer-movie-ribbon，将 ArtifactId 修改为 microservice-consumer-movie-ribbon-hystrix。
2. 为项目添加以下依赖。

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

3. 在启动类上添加注解 @EnableCircuitBreaker 或 @EnableHystrix，从而为项目启用断路器支持。
4. 修改 MovieController，让其中的 findById 方法具备容错能力。

```
@RestController
public class MovieController {
    private static final Logger LOGGER = LoggerFactory.getLogger(MovieController.
        class);
    @Autowired
    private RestTemplate restTemplate;
    @Autowired
    private LoadBalancerClient loadBalancerClient;

    @HystrixCommand(fallbackMethod = "findByIdFallback")
```

```
@GetMapping("/user/{id}")
public User findById(@PathVariable Long id) {
    return this restTemplate.getForObject("http://microservice-provider-user/" +
        id, User.class);
}

public User findByIdFallback(Long id) {
    User user = new User();
    user.setId(-1L);
    user.setName("默认用户");
    return user;
}
...
}
```

由代码可知，为 `findById` 方法编写了一个回退方法 `findByIdFallback`，该方法与 `findById` 方法具有相同的参数与返回值类型，该方法返回了一个默认的 `User`。

在 `findById` 方法上，使用注解 `@HystrixCommand` 的 `fallbackMethod` 属性，指定回退方法是 `findByIdFallback`。注解 `@HystrixCommand` 由名为 `javanica` (<https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica>) 的 `Hystrix contrib` 库提供。`javanica` 是一个 `Hystrix` 的子项目，用于简化 `Hystrix` 的使用。Spring Cloud 自动将 Spring bean 与该注解封装在一个连接到 `Hystrix` 断路器的代理中。

`@HystrixCommand` 的配置非常灵活，可使用注解 `@HystrixProperty` 的 `commandProperties` 属性来配置 `@HystrixCommand`。例如：

```
@HystrixCommand(fallbackMethod = "findByIdFallback", commandProperties = {
    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds",
        value = "5000"),
    @HystrixProperty(name = "metrics.rollingStats.timeInMilliseconds", value =
        "10000")
}, threadPoolProperties = {
    @HystrixProperty(name = "coreSize", value = "1"),
    @HystrixProperty(name = "maxQueueSize", value = "10")
})
@GetMapping("/user/{id}")
public User findById(@PathVariable Long id) {
    // ...
}
```

限于篇幅，笔者就不一一讲解 Hystrix 的配置属性了。

读者可前往 <https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica#configuration> 详细了解注解 @HystrixCommand 如何使用。

Hystrix 配置属性可详见 Hystrix Wiki: <https://github.com/Netflix/Hystrix/wiki/Configuration>。

测试

1. 启动项目 microservice-discovery-eureka。
2. 启动项目 microservice-provider-user。
3. 启动项目 microservice-consumer-movie-ribbon-hystrix。
4. 访问 <http://localhost:8010/user/1>，可获得如下的结果。

```
{  
    "id": 1,  
    "username": "account1",  
    "name": "张三",  
    "age": 20,  
    "balance": 100  
}
```

5. 停止 microservice-provider-user。
6. 再次访问 <http://localhost:8010/user/1>，获得如下结果。

```
{  
    "id": -1,  
    "username": null,  
    "name": "默认用户",  
    "age": null,  
    "balance": null  
}
```

说明当用户微服务不可用时，进入了回退方法。

我们知道，当请求失败、被拒绝、超时或者断路器打开时，都会进入回退方法。但进入回退方法并不意味着断路器已经被打开。那么，如何才能明确了解断路器当前的状态呢？请听下回分解。

7.2.3 Hystrix 断路器的状态监控与深入理解

还记得之前为项目引入了 Spring Boot Actuator 吗？

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

断路器的状态也会暴露在 Actuator 提供的/health 端点中，这样就可以直观地了解断路器的状态。下面来做一点实验，深入理解断路器的状态转换。

测试

1. 启动项目 microservice-discovery-eureka。
2. 启动项目 microservice-provider-user。
3. 启动项目 microservice-consumer-movie-ribbon-hystrix。
4. 访问 `http://localhost:8010/user/1`，可正常获得结果。
5. 访问 `http://localhost:8010/health`，可获得类似如下结果。

```
{
    "status": "UP",
    "hystrix": {
        "status": "UP"
    }
    ...
}
```

可以看到此时，Hystrix 的状态是 UP，也就是一切正常，此时断路器是关闭的。

6. 停止 microservice-provider-user，访问 `http://localhost:8010/user/1`，可获得如下结果。

```
{
    "id": -1,
    "username": null,
    "name": "默认用户",
    "age": null,
    "balance": null
}
```

7. 访问 `http://localhost:8010/health`，可获得如下结果。

```
{
  "status": "UP",
  "hystrix": {
    "status": "UP"
  }
  ...
}
```

我们发现，尽管执行了回退逻辑，返回了默认用户，但此时 Hystrix 的状态依然是 UP，这是因为我们的失败率还没有达到阈值（默认是 5 秒内 20 次失败）。这是很多初学者会遇到的误区，这边再次强调——执行回退逻辑并不代表断路器已经打开。请求失败、超时、被拒绝以及断路器打开时等都会执行回退逻辑。

- 持续快速地访问 <http://localhost:8010/user/1>，直到请求快速返回。

```
{
  "status": "UP",
  "hystrix": {
    "status": "CIRCUIT_OPEN",
    "openCircuitBreakers": [
      "MovieController::findById"
    ]
  }
  ...
}
```

可以看到，Hystrix 的状态是 CIRCUIT_OPEN，说明断路器已经打开，不会再对请求用户微服务了。我们也可使用类似的方法测试断路器从打开转半开以及从半开自动恢复等过程。

7.2.4 Hystrix 线程隔离策略与传播上下文

本节相对复杂，为了讲解这个问题，先来阅读一下 Hystrix 官方 Wiki (<https://github.com/Netflix/Hystrix/wiki/Configuration#execution.isolation.strategy>)。

`execution.isolation.strategy`

This property indicates which isolation strategy `HystrixCommand.run()` executes with, one of the following two choices:

- THREAD — it executes on a separate thread and concurrent requests are limited by the number of threads in the thread-pool

- SEMAPHORE —it executes on the calling thread and concurrent requests are limited by the semaphore count

Thread or Semaphore

The default, and the recommended setting, is to run commands using thread isolation (THREAD).

Commands executed in threads have an extra layer of protection against latencies beyond what network timeouts can offer.

Generally the only time you should use semaphore isolation (SEMAPHORE) is when the call is so high volume (hundreds per second, per instance) that the overhead of separate threads is too high; this typically only applies to non-network calls.

简单翻译一下，Hystrix 的隔离策略有两种：分别是线程隔离和信号量隔离。

- THREAD (线程隔离): 使用该方式，`HystrixCommand` 将会在单独的线程上执行，并发请求受线程池中的线程数量的限制。
- SEMAPHORE (信号量隔离): 使用该方式，`HystrixCommand` 将会在调用线程上执行，开销相对较小，并发请求受到信号量个数的限制。

Hystrix 中默认并且推荐使用线程隔离 (THREAD)，因为这种方式有一个除网络超时以外的额外保护层。

一般来说，只有当调用负载非常高时（例如每个实例每秒调用数百次）才需要使用信号量隔离，因为这种场景下使用 THREAD 开销会比较高。信号量隔离一般仅适用于非网络调用的隔离。

可使用`execution.isolation.strategy` 属性指定隔离策略。

了解 Hystrix 的隔离策略后，再来看一下 Spring Cloud 官方的文档：

If you want some thread local context to propagate into a `@HystrixCommand` the default declaration will not work because it executes the command in a thread pool (in case of timeouts). You can switch Hystrix to use the same thread as the caller using some configuration, or directly in the annotation, by asking it to use a different "Isolation Strategy". For example:

```
@HystrixCommand(fallbackMethod = "stubMyService",
    commandProperties = {
```

```

        @HystrixProperty(name="execution.isolation.strategy", value=
    SEMAPHORE")
    }
)
...

```

The same thing applies if you are using `@SessionScope` or `@RequestScope`. You will know when you need to do this because of a runtime exception that says it can't find the scoped context.

You also have the option to set the `hystrix.shareSecurityContext` property to true. Doing so will auto configure an Hystrix concurrency strategy plugin hook who will transfer the `SecurityContext` from your main thread to the one used by the Hystrix command. Hystrix does not allow multiple `hystrix concurrency strategy` to be registered so an extension mechanism is available by declaring your own `HystrixConcurrencyStrategy` as a Spring bean. Spring Cloud will lookup for your implementation within the Spring context and wrap it inside its own plugin.

简单翻译一下：

如果你想传播线程本地的上下文到`@HystrixCommand`，默认声明将不会工作，因为它会在线程池中执行命令（在超时的情况下）。你可以使用一些配置，让 Hystrix 使用相同的线程，或者直接在注解中让 Hystrix 使用不同的隔离策略。例如：

```

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

```

这也适用于使用`@SessionScope`或者`@RequestSession`的情况。你会知道什么时候需要这样做，因为会发生一个运行时异常，说它找不到作用域上下文（scoped context）。

你还可将`hystrix.shareSecurityContext`属性设置为 true，这样将会自动配置一个 Hystrix 并发策略插件的 hook，这个 hook 会将 `SecurityContext` 从主线程传输到 Hystrix 的命令。因为 Hystrix 不允许注册多个 Hystrix 并发策略，所以可以声明 `HystrixConcurrencyStrategy` 为一个 Spring bean 来实现扩展。Spring Cloud 会在 Spring 的上下文中查找你的实现，并将其包装在自己的插件中。

总结

把 Spring Cloud 和 Hystrix 的文档对照阅读，就能很好地理解相关概念。在此，笔者总结如下：

- Hystrix 的隔离策略有 THREAD 和 SEMAPHORE 两种，默认是 THREAD。
- 正常情况下，保持默认即可。
- 如果发生找不到上下文的运行时异常，可考虑将隔离策略设置为 SEMAPHORE。



- 在 Github 上的相关 issue，可帮助大家理解：<https://github.com/spring-cloud/spring-cloud-netflix/issues/1336>。
- 服务熔断、降级、限流、异步 RPC -- Hystrix：<http://blog.csdn.net/chunlongyu/article/details/53259014>。
- 分布式系统延迟和容错框架 Hystrix：<http://blog.csdn.net/fight4gold/article/details/51252217>。

7.2.5 Feign 使用 Hystrix

前文是使用注解 @HystrixCommand 的 fallbackMethod 属性实现回退的。然而，Feign 是以接口形式工作的，它没有方法体，前文讲解的方式显然不适用于 Feign。

那么 Feign 要如何整合 Hystrix 呢？不仅如此，如何实现 Feign 的回退呢？

事实上，Spring Cloud 默认已为 Feign 整合了 Hystrix，只要 Hystrix 在项目的 classpath 中，Feign 默认就会用断路器包裹所有方法。下面来详细探讨如何实现 Feign 的回退。

7.2.5.1 为 Feign 添加回退

本节为前文编写的 UserFeignClient 添加回退。

1. 复制项目 microservice-consumer-movie-feign，将 ArtifactId 修改为 microservice-consumer-movie-feign-hystrix-fallback。
2. 将之前编写的 Feign 接口修改成如下内容：

```
/**  
 * Feign的fallback测试  
 * 使用@FeignClient的fallback属性指定回退类  
 * @author 周立  
 */
```

```
@FeignClient(name = "microservice-provider-user", fallback = FeignClientFallback
    .class)
public interface UserFeignClient {
    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public User findById(@PathVariable("id") Long id);

}

/**
 * 回退类FeignClientFallback需实现Feign Client接口
 * FeignClientFallback也可以是public class, 没有区别
 * @author 周立
 */
@Component
class FeignClientFallback implements UserFeignClient {
    @Override
    public User findById(Long id) {
        User user = new User();
        user.setId(-1L);
        user.setUsername("默认用户");
        return user;
    }
}
```

由代码可知，只须使用@FeignClient注解的fallback属性，就可为指定名称的Feign客户端添加回退。

测试

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user。
3. 启动 microservice-consumer-movie-feign-hystrix-fallback。
4. 访问`http://localhost:8010/user/1`，可正常获得结果。
5. 停止 microservice-provider-user。
6. 再次访问`http://localhost:8010/user/1`，可获得如下结果。说明当用户微服务不可用时，进入了回退的逻辑。

```
{  
    "id": -1,  
    "username": "默认用户",  
    "name": null,  
    "age": null,  
    "balance": null  
}
```

7.2.5.2 通过 Fallback Factory 检查回退原因

很多场景下，需要了解回退的原因，此时可使用注解@FeignClient的 fallbackFactory 属性。

下面来编写一个示例，为 Feign 打印回退日志。

1. 复制项目microservice-consumer-movie-feign，将 ArtifactId 修改为 microservice-consumer-movie-feign-hystrix-fallback-factory。
2. 将 UserFeignClient 改为如下内容。

```
@FeignClient(name = "microservice-provider-user", fallbackFactory =  
    FeignClientFallbackFactory.class)  
public interface UserFeignClient {  
    @RequestMapping(value = "/{id}", method = RequestMethod.GET)  
    public User findById(@PathVariable("id") Long id);  
}  
  
/**  
 * UserFeignClient的fallbackFactory类，该类需实现FallbackFactory接口，并覆写  
 * create方法  
 * The fallback factory must produce instances of fallback classes that  
 * implement the interface annotated by {@link FeignClient}.  
 * @author 周立  
 */  
@Component  
class FeignClientFallbackFactory implements FallbackFactory<UserFeignClient> {  
    private static final Logger LOGGER = LoggerFactory.getLogger(  
        FeignClientFallbackFactory.class);  
  
    @Override  
    public UserFeignClient create(Throwable cause) {  
        return new UserFeignClient() {
```

```

@Override
public User findById(Long id) {
    // 日志最好放在各个fallback方法中，而不要直接放在create方法中。
    // 否则在引用启动时，就会打印该日志。
    // 详见https://github.com/spring-cloud/spring-cloud-netflix/issues/1471
    FeignClientFallbackFactory.LOGGER.info("fallback; reason was:", cause);
    User user = new User();
    user.setId(-1L);
    user.setUsername("默认用户");
    return user;
}
};

}

}

```

这样，当 Feign 发生回退时，就会打印日志。



测试

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user。
3. 启动 microservice-consumer-movie-feign-hystrix-fallback-factory。
4. 访问`http://localhost:8010/user/1`能正常获得结果。
5. 停止 microservice-provider-user。
6. 再次访问`http://localhost:8010/user/1`，可获得如下结果。

```
{
    "id": -1,
    "username": "默认用户",
    "name": null,
    "age": null,
    "balance": null
}
```

并且，控制台会输出类似如下的日志。

```
INFO 23296 --- [provider-user-1] c.i.c.s.u.f.FeignClientFallbackFactory : fallback; reason was: com.netflix.client.ClientException: Load balancer does not have available server for client: microservice-provider-user
```

说明进入了回退类中的回退方法。



1. fallbackFactory 属性还有很多其他的用途，例如让不同的异常返回不同的回退结果，从而使 Feign 的回退更加灵活。例如：

```
@Component
class FeignClientFallbackFactory implements FallbackFactory<
    UserFeignClient> {
    @Override
    public UserFeignClient create(Throwable cause) {
        return new UserFeignClient() {
            @Override
            public User findById(Long id) {
                User user = new User();
                if (cause instanceof IllegalArgumentException) {
                    user.setId(-1L);
                } else {
                    user.setId(-2L);
                }
                return user;
            }
        };
    }
}
```

2. 在特定的时间窗口中，create(Throwable cause) 中的 cause 可能是 null。这是 Feign 的 Bug，该 Bug 在 Feign 9.4.0 中已被解决。由于 Spring Cloud Camden SR4 使用的 Feign 版本是 9.3.1，因此，在使用该特性时，如需访问 cause 变量中的属性，需添加一些判断代码，从而规避空指针异常。对该 Bug 感兴趣的朋友可详见该 Issue：<https://github.com/OpenFeign/feign/issues/464>。

7.2.5.3 为 Feign 禁用 Hystrix

前文说过，在 Spring Cloud 中，只要 Hystrix 在项目的 classpath 中，Feign 就会使用断路器包裹 Feign 客户端的所有方法。这样虽然方便，但很多场景下并不需要该功能。如何为 Feign 禁用 Hystrix 呢？

为指定 Feign 客户端禁用 Hystrix

借助 Feign 的自定义配置，可轻松为指定名称的 Feign 客户端禁用 Hystrix。例如：

```

@Configuration
public class FeignDisableHystrixConfiguration {
    @Bean
    @Scope("prototype")
    public Feign.Builder feignBuilder() {
        return Feign.builder();
    }
}

```

想要禁用 Hystrix 的@FeignClient引用该配置类即可，例如：

```

@FeignClient(name = "user", configuration = FeignDisableHystrixConfiguration.class)
public interface UserFeignClient {
    //...
}

```

全局禁用 Hystrix

也可为 Feign 全局禁用 Hystrix。只须在 application.yml 中配置feign.hystrix.enabled = false即可。

7.3 Hystrix 的监控

除实现容错外，Hystrix 还提供了近乎实时的监控。HystrixCommand 和 HystrixObservableCommand 在执行时，会生成执行结果和运行指标，比如每秒执行的请求数、成功数等，这些监控数据对分析应用系统的状态很有用。

使用 Hystrix 的模块hystrix-metrics-event-stream，就可将这些监控的指标信息以text/event-stream的格式暴露给外部系统。spring-cloud-starter-hystrix已包含该模块，在此基础上，只须为项目添加spring-boot-starter-actuator，就可使用/hystrix.stream 端点获得 Hystrix 的监控信息了。

如上所述，前文的项目microservice-consumer-movie-ribbon-hystrix已具备监控 Hystrix 的能力。下面来做一点测试。

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user。
3. 启动 microservice-consumer-movie-ribbon-hystrix。
4. 访问<http://localhost:8010/hystrix.stream>，可看到浏览器一直处于请求的状态，页面空白。这是因为此时项目中注解了@HystrixCommand的方法还没有被执行，因此也没有

任何的监控数据。

5. 访问`http://localhost:8010/user/1`后，再次访问`http://localhost:8010/hystrix.stream`，可看到页面会重复出现类似于以下的内容。

```
data: {"type": "HystrixCommand", "name": "findById", "group": "MovieController", "currentTime": 1479303396533, "isCircuitBreakerOpen": false, "errorPercentage": 0, "errorCode": 0, "requestCount": 0, "rollingCountBadRequests": 0, "rollingCountCollapsedRequests": 0, "rollingCountEmit": 0, "rollingCountExceptionsThrown": 0, "rollingCountFailure": 0, "rollingCountFallbackEmit": 0, "rollingCountFallbackFailure": 0, "rollingCountFallbackMissing": 0, "rollingCountFallbackRejection": 0, "rollingCountFallbackSuccess": 0, "rollingCountResponsesFromCache": 0, "rollingCountSemaphoreRejected": 0, "rollingCountShortCircuited": 0, "rollingCountSuccess": 0, "rollingCountThreadPoolRejected": 0, "rollingCountTimeout": 0, "currentConcurrentExecutionCount": 1, "rollingMaxConcurrentExecutionCount": 0, "latencyExecute_mean": 0, "latencyExecute": ...}
```

这是因为系统会不断地刷新以获得实时的监控数据。Hystrix 的监控指标非常全面，例如 HystrixCommand 的名称、group 名称、断路器状态、错误率、错误数等。

Feign 项目的 Hystrix 监控

启动前文的`microservice-consumer-movie-feign-hystrix-fallback`项目，并使用类似的方式测试，然后访问`http://localhost:8010/hystrix.stream`，发现返回的是 404。这是为什么呢？

查看项目的依赖树会发现，项目甚至连`hystrix-metrics-event-stream`的依赖都没有。那么如何解决该问题呢？

解决方案如下：

1. 复制项目`microservice-consumer-movie-feign-hystrix-fallback`，将`ArtifactId`修改为`microservice-consumer-movie-feign-hystrix-fallback-stream`。
2. 为项目添加`spring-cloud-starter-hystrix`的依赖。

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

3. 在启动类上添加`@EnableCircuitBreaker`，这样就可使用`/hystrix.stream`端点监控 Hystrix 了。

7.4 使用 Hystrix Dashboard 可视化监控数据

前面讨论了 Hystrix 的监控，但访问/hystrix.stream 端点获得的数据是以文字形式展示的。很难通过这些数据，一眼看出系统当前的运行状态。

可使用 Hystrix Dashboard，从让监控数据图形化、可视化。

下面来编写一个 Hystrix Dashboard。

1. 创建一个 Maven 项目，ArtifactId 是 microservice-hystrix-dashboard，并为项目添加以下依赖。

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

2. 编写启动类，在启动类上添加 @EnableHystrixDashboard。

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

3. 在配置文件 application.yml 中添加如下内容。

```
server:
  port: 8030
```

这样，一个简单的 Hystrix Dashboard 就完成了。由配置可知，我们并没有把 Hystrix Dashboard 注册到 Eureka Server 上。

测试

1. 访问 <http://localhost:8030/hystrix>，可看到 Hystrix Dashboard 的主页，如图 7-3 所示。

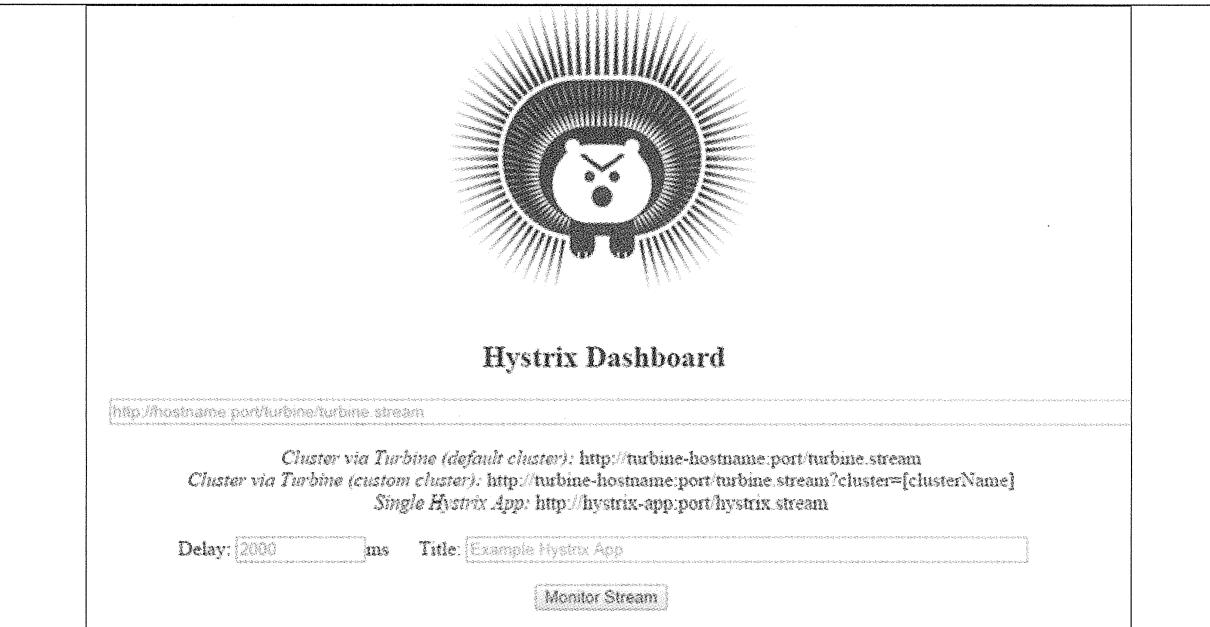


图 7-3 Hystrix Dashboard 主页

2. 在上一节测试的基础上，在 URL 一栏输入 `http://localhost:8010/hystrix.stream`，随意设置一个 Title，并点击 Monitor Stream 按钮后，即可看到类似图 7-4 的界面。

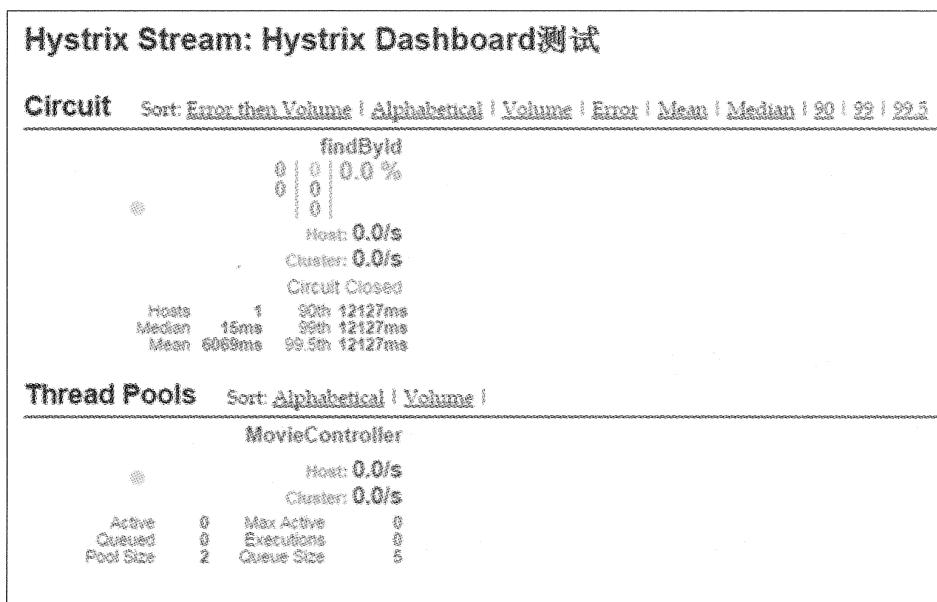


图 7-4 Hystrix Dashboard 监控页面

图 7-4 详细展示了 findById 这个 HystrixCommand 的各种指标，但这些指标的含义是什么呢？图 7-5 来自 Hystrix Dashboard 的 Wiki 页面，详细说明了每个指标的含义。

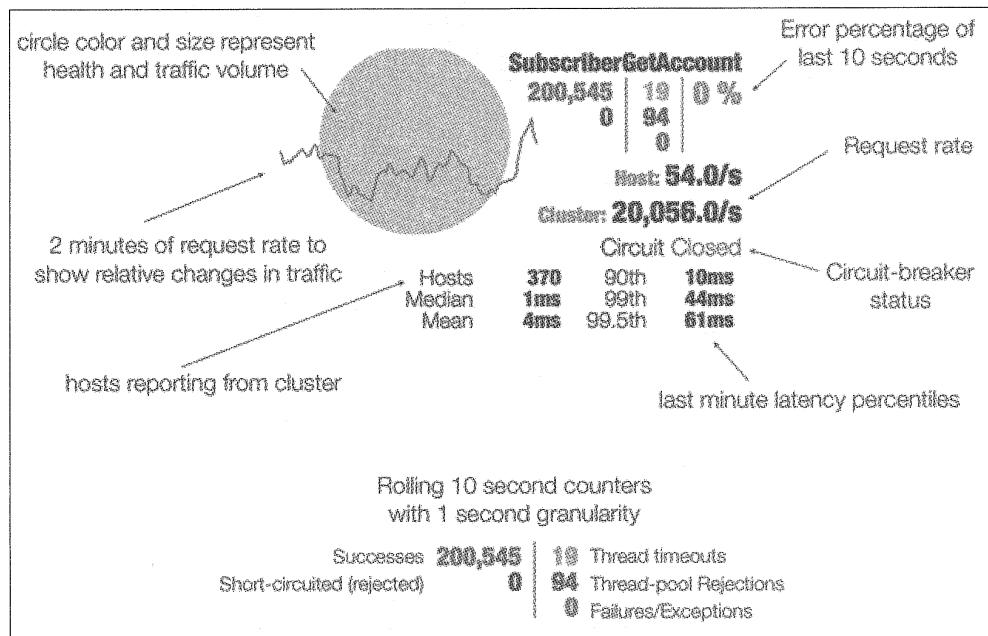


图 7-5 Hystrix Dashboard 指标解释



简单起见，笔者并没有把 Hystrix Dashboard 注册到 Eureka Server 上。在生产环境中，也可将 Hystrix Dashboard 注册到 Eureka Server 上，更方便地管理 Hystrix Dashboard。

7.5 使用 Turbine 聚合监控数据

前文中使用/hystrix.stream 端点监控单个微服务实例。然而，使用微服务架构的应用系统一般会包含若干个微服务，每个微服务通常都会部署多个实例。如果每次只能查看单个实例的监控数据，就必须在 Hystrix Dashboard 上切换想要监控的地址，这显然很不方便。如何解决该问题呢？

7.5.1 Turbine 简介

Turbine 是一个聚合 Hystrix 监控数据的工具，它可将所有相关/hystrix.stream 端点的数据聚合到一个组合的/turbine.stream 中，从而让集群的监控更加方便。

引入 Turbine 后，架构如图 7-6 所示。

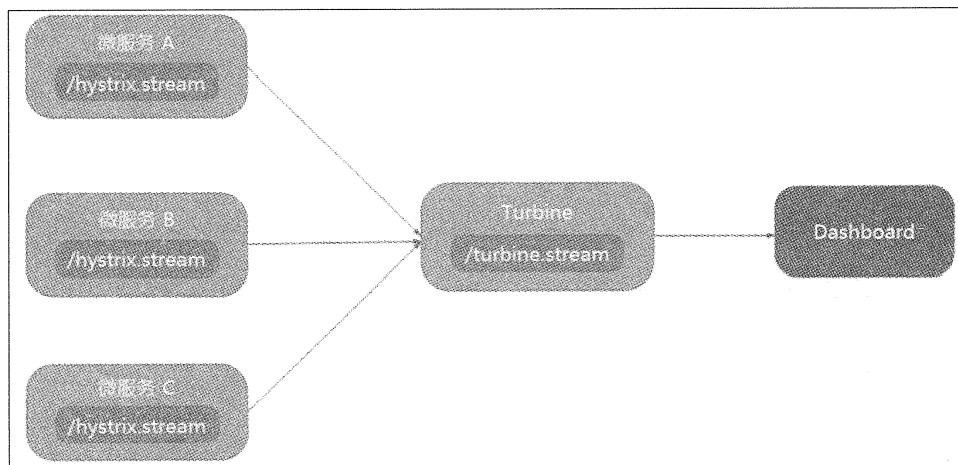


图 7-6 微服务引入 Turbine 架构图



Turbine 的 GitHub: <https://github.com/Netflix/Turbine>。

7.5.2 使用 Turbine 监控多个微服务

本节来编写一个 Turbine 项目。

- 为了让 Turbine 监控 2 个以上的微服务，先将项目 microservice-consumer-movie-feign-hystrix-fallback-stream 的 application.yml 改成如下内容：

```
server:  
  port: 8020  
spring:  
  application:  
    name: microservice-consumer-movie-feign-hystrix-fallback-stream  
eureka:  
  client:  
    serviceUrl:  
      defaultZone: http://localhost:8761/eureka/  
instance:  
  prefer-ip-address: true
```

- 创建一个 Maven 项目，ArtifactId 是 microservice-hystrix-turbine，并为项目添加以下依赖。

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

3. 在启动类上添加@EnableTurbine注解。

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

4. 编写配置文件 application.yml。

```
server:
  port: 8031
spring:
  application:
    name: microservice-hystrix-turbine
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
turbine:
  appConfig: microservice-consumer-movie,microservice-consumer-movie-feign-
    hystrix-fallback-stream
  clusterNameExpression: "'default'"
```

使用以上配置，Turbine会在Eureka Server中找到microservice-consumer-movie和microservice-consumer-movie-feign-hystrix-fallback-stream这两个微服务，并聚合两个微服务的监控数据。

测试

1. 启动项目 microservice-discovery-eureka。
2. 启动项目 microservice-provider-user。

3. 启动项目 microservice-consumer-movie-ribbon-hystrix。
4. 启动项目 microservice-consumer-movie-feign-hystrix-fallback-stream。
5. 启动项目 microservice-hystrix-turbine。
6. 启动项目 microservice-hystrix-dashboard。
7. 访问 <http://localhost:8010/user/1>, 让 microservice-consumer-movie-ribbon-hystrix 微服务产生监控数据。
8. 访问 <http://localhost:8020/user/1>, 让 microservice-consumer-movie-feign-hystrix-fallback-stream 微服务产生监控数据。
9. 打开 Hystrix Dashboard 首页 <http://localhost:8030/hystrix.stream>, 在 URL 一栏填入 <http://localhost:8031/turbine.stream>, 随意指定一个 Title 并点击 Monitor Stream 按钮后, 结果类似于图 7-7。

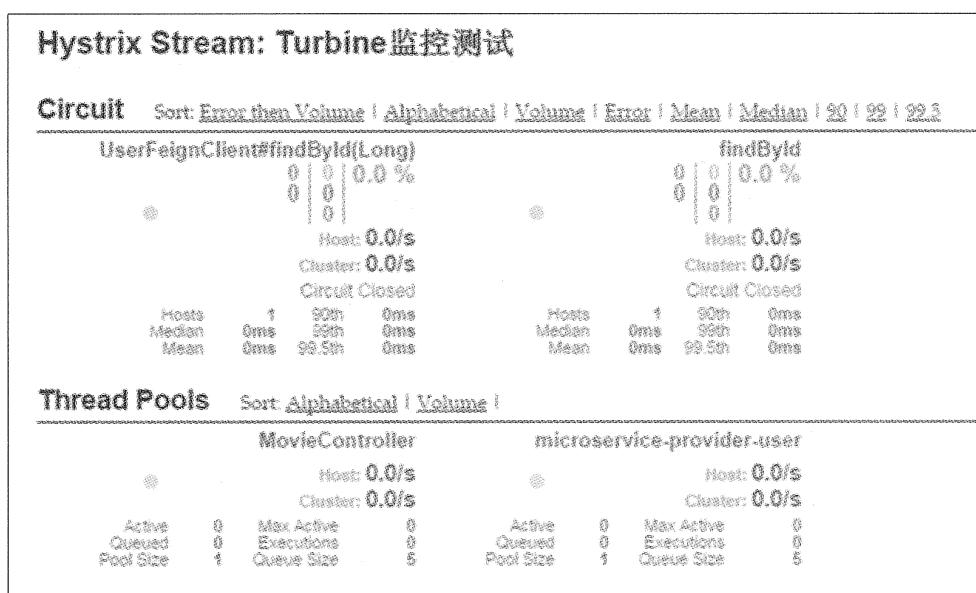


图 7-7 Turbine 监控多个微服务

7.5.3 使用消息中间件收集数据

一些场景下, 前文的方式无法正常工作(例如微服务与 Turbine 网络不通), 此时, 可借助消息中间件实现数据收集。各个微服务将 Hystrix Command 的监控数据发送至消息中间件, Turbine 消费消息中间件中的数据。

下面笔者以 RabbitMQ 为例进行演示。

7.5.3.1 安装 RabbitMQ

先来安装 RabbitMQ，以 Windows 操作系统为例。

一、安装 RabbitMQ

1. 安装 Erlang/OTP 19.2

RabbitMQ 依赖 Erlang，先来安装 Erlang。

通过官方下载页面：<http://www.erlang.org/downloads>，获取 exe 安装包，双击打开，按照提示即可完成安装。

2. 安装 RabbitMQ Server 3.6.6

通过官方下载页面<http://www.rabbitmq.com/install-windows.html>，获取 exe 安装包，双击打开，按照提示即可完成安装。

3. 安装完成

安装完成后，在“计算机-管理-服务和应用程序-服务”中，就能看到名为 RabbitMQ 的服务了，如图 7-8 所示。



图 7-8 RabbitMQ 注册为服务

二、安装 RabbitMQ 管理插件

为了更加方便地管理 RabbitMQ，接着安装 RabbitMQ 的管理插件。

1. 将目录切换到 RabbitMQ 中的 sbin 目录，例如：

```
cd D:\Program Files\RabbitMQ Server\rabbitmq_server-3.6.6\sbin>
```

当然，也可点击开始菜单中的“RabbitMQ Command Prompt (sbin dir)”菜单，直接切换到 sbin 目录。

- 执行以下命令，安装管理插件。

```
rabbitmq-plugins enable rabbitmq_management
```

- 访问`http://localhost:15672/`，输入默认账号 guest，密码 guest，即可看到如图 7-9 的界面。

File descriptors (?)	Socket descriptors (?)	Erlang processes	Memory	Disk space	Rates mode	Info
0 8192 available	0 7280 available	259 1046376 available	54MB 6.3GB high watermark	16GB 8MB low watermark	basic	Disc 1 Stats

图 7-9 RabbitMQ 首页

这样就可使用图形化的界面管理 RabbitMQ 了。

7.5.3.2 改造微服务

要想使用消息中间件收集监控数据，需要改造前文编写的微服务，以 `microservice-consumer-movie-ribbon-hystrix` 为例。

- 复制项目 `microservice-consumer-movie-ribbon-hystrix`，将 `ArtifactId` 修改为 `microservice-consumer-movie-ribbon-hystrix-turbine-mq`。
- 添加以下依赖。

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

- 在配置文件 application.yml 中添加如下内容，连接 RabbitMQ。

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

这样，微服务就改造完成了。

7.5.3.3 改造 Turbine

改造完微服务后，接下来改造 Turbine。

- 复制项目 microservice-hystrix-turbine，将 ArtifactId 修改为 microservice-hystrix-turbine-mq。
- 在 pom.xml 中，添加如下内容。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-turbine-stream</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

同时，删除：

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

3. 修改启动类，将注解`@EnableTurbine`修改为`@EnableTurbineStream`。
4. 修改配置文件`application.yml`，添加如下内容。

```
spring:  
  rabbitmq:  
    host: localhost  
    port: 5672  
    username: guest  
    password: guest
```

同时，删除：

```
turbine:  
  appConfig: microservice-consumer-movie,microservice-consumer-movie-feign-hystrix-fallback-stream  
  clusterNameExpression: "'default'"
```

这样，Turbine 就改造完成了。



测试

1. 启动项目`microservice-discovery-eureka`。
2. 启动项目`microservice-provider-user`。
3. 启动项目`microservice-consumer-movie-ribbon-hystrix-turbine-mq`。
4. 启动项目`microservice-hystrix-turbine-mq`。
5. 访问`http://localhost:8010/user/1`，可正常获得结果。
6. 访问`http://localhost:8031/`，会发现 Turbine 能够持续不断地显示监控数据。



在 Spring Cloud Camden SR4 中，依赖`spring-cloud-starter-turbine` 不能与`spring-cloud-starter-turbine-stream` 共存，否则启动时会报异常。不仅如此，这两个依赖使用的 Turbine 版本也不相同，`spring-cloud-starter-turbine` 使用的 Turbine 版本是 1.0.0，而`spring-cloud-starter-turbine-stream` 使用的 Turbine 版本是 2.0.0-DP.2。笔者已在 GitHub 上提出相关 Issue，详见<https://github.com/spring-cloud/spring-cloud-netflix/issues/1629>。

使用 Zuul 构建微服务网关

8.1 为什么要使用微服务网关

经过前文的讲解，微服务架构已经初具雏形，但还有一些问题——不同的微服务一般会有不同的网络地址，而外部客户端（例如手机 APP）可能需要调用多个服务的接口才能完成一个业务需求。例如一个电影购票的手机 APP，可能会调用多个微服务的接口，才能完成一次购票的业务流程，如图 8-1 所示。

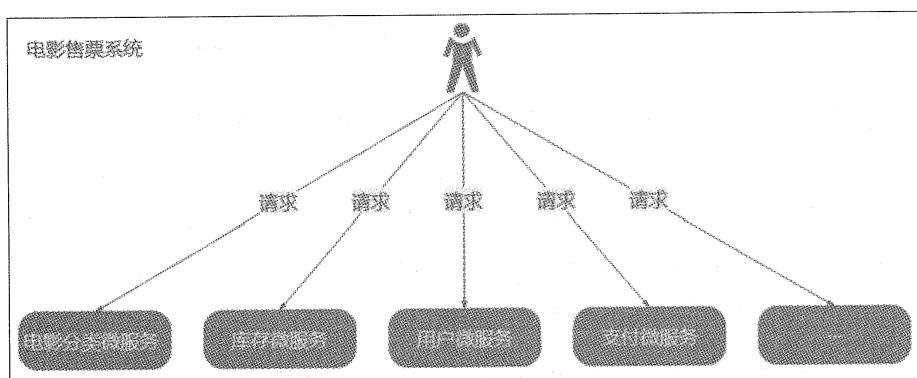


图 8-1 用户请求多个微服务

如果让客户端直接与各个微服务通信，会有以下的问题：

- 客户端会多次请求不同的微服务，增加了客户端的复杂性。
- 存在跨域请求，在一定场景下处理相对复杂。
- 认证复杂，每个服务都需要独立认证。
- 难以重构，随着项目的迭代，可能需要重新划分微服务。例如，可能将多个服务合并在一个或者将一个服务拆分成多个。如果客户端直接与微服务通信，那么重构将会很难实施。
- 某些微服务可能使用了防火墙/浏览器不友好的协议，直接访问会有一定的困难。

以上问题可借助微服务网关解决。微服务网关是介于客户端和服务器端之间的中间层，所有的外部请求都会先经过微服务网关。使用微服务网关后，架构可演变成图 8-2。

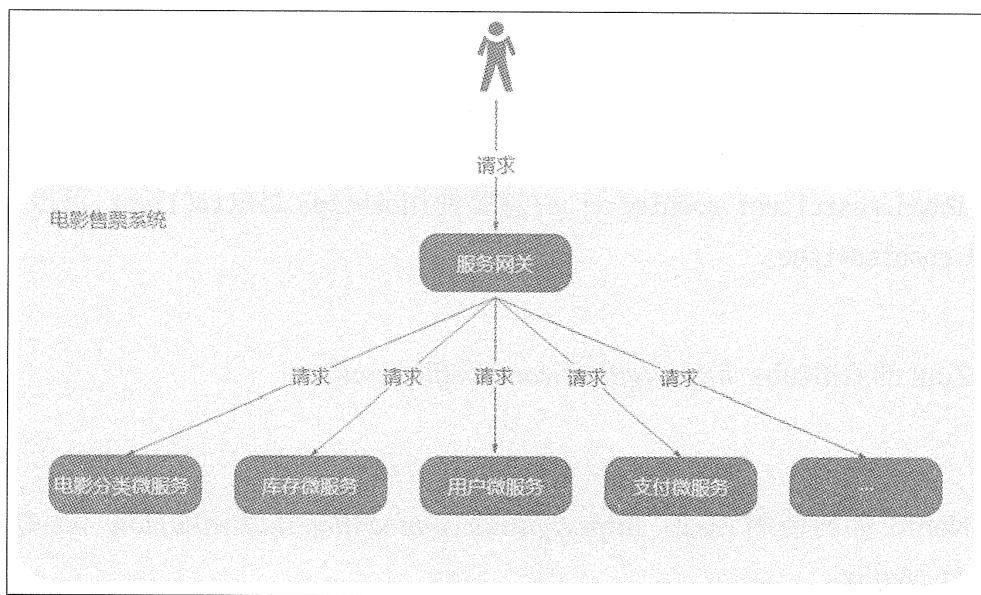


图 8-2 使用微服务网关

如图，微服务网关封装了应用程序的内部结构，客户端只须跟网关交互，而无须直接调用特定微服务的接口。这样，开发就可以得到简化。不仅如此，使用微服务网关还有以下优点：

- 易于监控。可在微服务网关收集监控数据并将其推送到外部系统进行分析。
- 易于认证。可在微服务网关上进行认证，然后再将请求转发到后端的微服务，而无须在每个微服务中进行认证。
- 减少了客户端与各个微服务之间的交互次数。

8.2 Zuul 简介

Zuul 是 Netflix 开源的微服务网关，它可以和 Eureka、Ribbon、Hystrix 等组件配合使用。Zuul 的核心是一系列的过滤器，这些过滤器可以完成以下功能。

- 身份认证与安全：识别每个资源的验证要求，并拒绝那些与要求不符的请求。
- 审查与监控：在边缘位置追踪有意义的数据和统计结果，从而带来精确的生产视图。
- 动态路由：动态地将请求路由到不同的后端集群。
- 压力测试：逐渐增加指向集群的流量，以了解性能。
- 负载分配：为每一种负载类型分配对应容量，并弃用超出限定值的请求。
- 静态响应处理：在边缘位置直接建立部分响应，从而避免其转发到内部集群。
- 多区域弹性：跨越 AWS Region 进行请求路由，旨在实现 ELB (Elastic Load Balancing) 使用的多样化，以及让系统的边缘更贴近系统的使用者。

Spring Cloud 对 Zuul 进行了整合与增强。目前，Zuul 使用的默认 HTTP 客户端是 Apache HTTP Client，也可以使用 RestClient 或者 okhttp3.OkHttpClient。如果想要使用 RestClient，可以设置 `ribbon.restclient.enabled=true`；想要使用 okhttp3.OkHttpClient，可以设置 `ribbon.okhttp.enabled=true`。



Zuul 的 GitHub：<https://github.com/Netflix/zuul>。



Netflix 如何使用 Zuul：<https://github.com/Netflix/zuul/wiki/How-We-Use-Zuul-At-Netflix>。

8.3 编写 Zuul 微服务网关

本节将编写一个简单的微服务网关。在本例中会将 Zuul 注册到 Eureka Server 上。

1. 创建一个 Maven 工程，ArtifactId 是 `microservice-gateway-zuul`，并为项目添加以下依赖。

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

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

2. 在启动类上添加注解`@EnableZuulProxy`，声明一个 Zuul 代理。该代理使用 Ribbon 来定位注册在 Eureka Server 中的微服务；同时，该代理还整合了 Hystrix，从而实现了容错，所有经过 Zuul 的请求都会在 Hystrix 命令中执行。

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

3. 编写配置文件 application.yml。

```
server:
  port: 8040
spring:
  application:
    name: microservice-gateway-zuul
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

这样，一个简单的微服务网关就编写完成了。从配置可知，此时仅是添加了 Zuul 的依赖，并将 Zuul 注册到 Eureka Server 上。

测试：路由规则

1. 启动项目 microservice-discovery-eureka。
2. 启动项目 microservice-provider-user。
3. 启动项目 microservice-consumer-movie-ribbon。
4. 启动项目 microservice-gateway-zuul。
5. 访问`http://localhost:8040/microservice-consumer-movie/user/1`，请求会被转发到`http:`

//localhost:8010/user/1 (电影微服务)。

6. 访问http://localhost:8040/microservice-provider-user/1 , 请求会被转发到http://localhost:8000/1 (用户微服务)。

说明默认情况下，Zuul 会代理所有注册到 Eureka Server 的微服务，并且 Zuul 的路由规则如下：

`http://ZUUL_HOST:ZUUL_PORT/微服务在Eureka上的serviceId/**` 会被转发到 serviceId 对应的微服务。

测试：负载均衡

1. 启动项目 microservice-discovery-eureka。
2. 启动多个 microservice-provider-user 实例。
3. 启动项目 microservice-gateway-zuul。此时，Eureka Server 首页如图 8-3 所示。

Instances currently registered with Eureka			
Application	AtMs	Availability Zones	Status
MICROSERVICE-GATEWAY-ZUUL	n/a {1}	{1}	UP {1} - itmuch:microservice-gateway-zuul:8040
MICROSERVICE-PROVIDER-USER	n/a {2}	{2}	UP {2} - itmuch:microservice-provider-user:8000 , itmuch:microservice-provider-user:8001

图 8-3 Eureka Server 上的微服务列表

4. 多次访问http://localhost:8040/microservice-provider-user/1 , 会发现两个用户微服务节点都会打印类似以下的日志。

```
Hibernate: select user0_.id as id1_0_0_, user0_.age as age2_0_0_, user0_.
balance as balance3_0_0_, user0_.name as name4_0_0_, user0_.username as
username5_0_0_ from user user0_ where user0_.id=?
```

说明 Zuul 可以使用 Ribbon 达到负载均衡的效果。

测试：Hystrix 容错与监控

1. 启动项目 microservice-discovery-eureka。
2. 启动项目 microservice-provider-user。
3. 启动项目 microservice-consumer-movie-ribbon。
4. 启动项目 microservice-gateway-zuul。

5. 启动项目 microservice-hystrix-dashboard。
6. 访问 <http://localhost:8040/microservice-consumer-movie/user/1>，能获得预期结果。
7. Hystrix Dashboard 中输入 <http://localhost:8040/hystrix.stream>，随意指定一个 Title 并点击 Monitor Stream 按钮后，结果如图 8-4 所示。

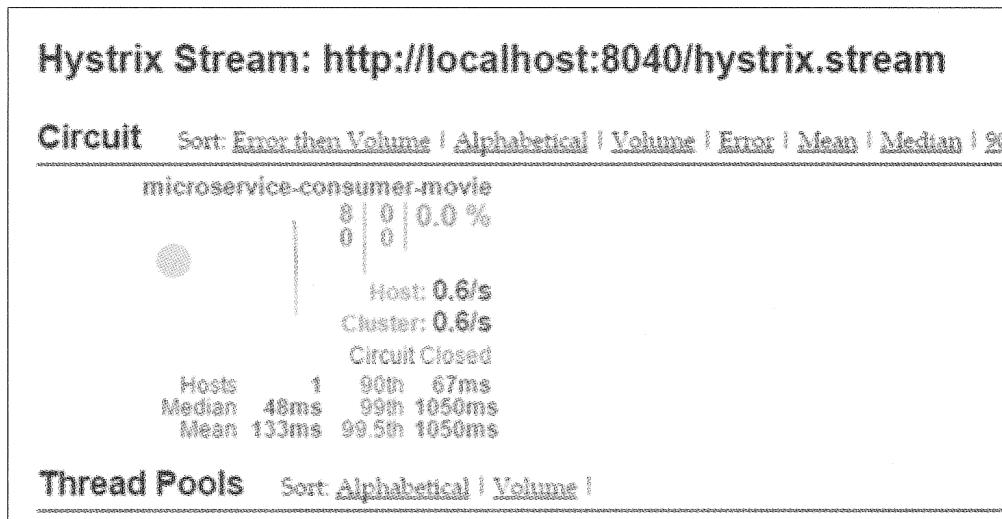


图 8-4 Hystrix Dashboard 监控页面

说明 Zuul 已经整合了 Hystrix。

8.4 Zuul 的路由端点

当@EnableZuulProxy 与 Spring Boot Actuator 配合使用时，Zuul 会暴露一个路由管理端点 /routes。借助这个端点，可以方便、直观地查看以及管理 Zuul 的路由。

/routes 端点的使用非常简单，使用 GET 方法访问该端点，即可返回 Zuul 当前映射的路由列表；使用 POST 方法访问该端点就会强制刷新 Zuul 当前映射的路由列表（尽管路由会自动刷新，Spring Cloud 依然提供了强制立即刷新的方式）。

由于 spring-cloud-starter-zuul 已经包含了 spring-boot-starter-actuator，因此之前编写的 microservice-gateway-zuul 已具备路由管理的能力。

下面来做一些测试。

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user。

3. 启动 microservice-consumer-movie。
4. 启动 microservice-gateway-zuul。
5. 使用浏览器访问 `http://localhost:8040/routes`，可获得如下的结果。

```
{
    "/microservice-provider-user/**": "microservice-provider-user",
    "/microservice-consumer-movie/**": "microservice-consumer-movie"
}
```

从中可以直观地看出从路径到微服务的映射。

也可使用类似方式测试 Zuul 路由的自动刷新与强制刷新。



Zuul 路由相关代码: `org.springframework.cloud.netflix.zuul.RoutesEndpoint`。

8.5 路由配置详解

前文已经编写了一个简单的 Zuul 网关，并让该网关代理了所有注册到 Eureka Server 的微服务。但在现实中可能只想让 Zuul 代理部分微服务，又或者需要对 URL 进行更加精确的控制。

Zuul 的路由配置非常灵活、简单，本节通过几个示例，详细讲解 Zuul 的路由配置。

1. 自定义指定微服务的访问路径。

配置 `zuul.routes.` 指定微服务的 `serviceId` = 指定路径即可。例如：

```
zuul:
  routes:
    microservice-provider-user: /user/**
```

这样设置，`microservice-provider-user` 微服务就会被映射到 `/user/**` 路径。

2. 忽略指定微服务。

忽略服务非常简单，可以使用 `zuul.ignored-services` 配置需要忽略的服务，多个用逗号分隔。例如：

```
zuul:
  ignored-services: microservice-provider-user,microservice-consumer-movie
```

这样就可让 Zuul 忽略 `microservice-provider-user` 和 `microservice-consumer-movie` 微服务，只代理其他微服务。