Broadview®
www.broadview.com.cn

Spring社区倾力打造的微服务架构开源解决方案
助力快速构建微服务架构体系

# Spring Cloud
# 微服务实战

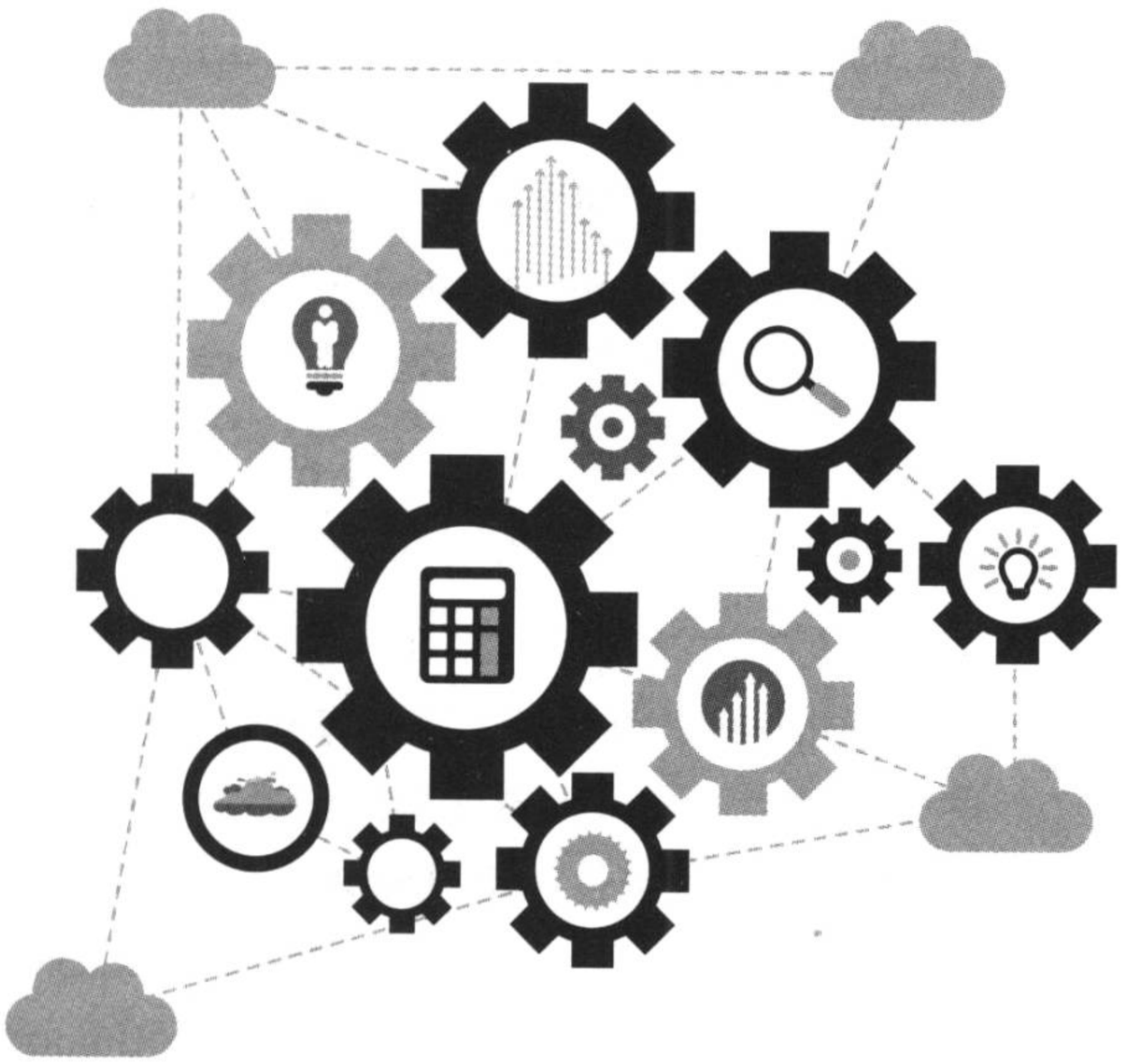
翟永超 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

# Spring Cloud

## 微服务实战

翟永超 著



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

## 内 容 简 介

本书从时下流行的微服务架构概念出发，详细介绍了 Spring Cloud 针对微服务架构中几大核心要素的解决方案和基础组件。对于各个组件的介绍，本书主要以示例与源码结合的方式来帮助读者更好地理解这些组件的使用方法以及运行原理。同时，在介绍的过程中，还包含了作者在实践中所遇到的一些问题和解决思路，可供读者在实践中作为参考。

本书适合所有 Java 开发人员，尤其适合正在做微服务架构技术选型或正在实施微服务架构的团队查阅和参考。

**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 微服务实战 / 翟永超著. —北京：电子工业出版社，2017.5
ISBN 978-7-121-31301-1

Ⅰ. ①S… Ⅱ. ①翟… 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 071305 号

策划编辑：张春雨
责任编辑：刘　舫
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：27.5　　字数：586 千字
版　　次：2017 年 5 月第 1 版
印　　次：2017 年 5 月第 1 次印刷
定　　价：89.00 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。

# 推荐序一

我和 Java 很有缘，2003 年在富士通南大实习的时候，就开始用 Struts/JSP/Hibernate/MySQL 做第一个 Java 项目——SPIF（http://www.fujitsu.com/cn/products/software/applications/spif/）。

工作之后，主要做 J2EE 的开发，并开始慢慢接触和使用 Spring。加入 EMC 之后，很有幸和 Spring 成为一个大家庭（EMC 收购了 VMware，VMware 收购了 Spring）。2012 年，我和 Spring Data/XD 的负责人 Mark Pollack 在中国开过一次小会，和他探讨了一些关于 Data Pipeline 的想法。后来我也看了很多 Josh Long（龙应春）的演讲视频，并在 2016 年终于有幸能够与他在一个技术大会上同场演讲，并在会议之后做了很多深入沟通。

2015 年上半年，我读了两本书：*The Phoenix Project* 和 *Migrating to Cloud-Native Application Architectures*，让我对 DevOps、微服务和云原生架构有了初步的认识，也让我对 Netflix 的那套 OSS 套件有了一个初步印象。

我是在 2015 年 9 月加入麻袋理财之后开始接触 Spring Boot 的，试用之后感觉它很神奇，再也没有被 Spring 之前那些烦琐配置所束缚。当时正好和一个架构师讨论要做一个项目的升级改造，决定采用 Spring Boot 和微服务架构。开始的时候，服务治理还是用了 Dubbo。之后因为对 Spring Cloud 有了比较深刻的认识，在之后一个全新项目上，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。

因为使用了 Spring Cloud，让我对 Spring Cloud 的相关信息特别关注。一个偶然的机会，我认识了 Spring Cloud 中国社区的负责人许进、翟永超（本书作者）和周立，探讨了很多使用 Spring Cloud 的经验，感觉与他们和 Spring Cloud 相见恨晚。

翟永超本人写了很多关于 Spring Cloud 使用的博客，不同于一般作者，他写的内容更

加贴近实际，是自己工作经验的深刻总结，可以拿来直接用于生产。

有一次我们聊到关于配置中心（Spring Cloud Config）如何在生产中使用，他解答了我很多问题，并告诉我他写了一本书，书中就会包含这些内容。这让我对这本书充满期待。后面也有幸见到了本人，一个瘦瘦高高的书生，一看就是一个很有内涵的技术人。畅聊之后，翟永超就把书发给了我，让我先睹为快。

我把翟永超的书仔细拜读了一遍，最大的收获就是让我对 Spring Cloud 的认识又上升了一个层次。我之前对 Spring Cloud 的理解更多的是知其然，但是却不知道其所以然，对 Spring Cloud 里面的逻辑知之甚少。而读了翟永超的《Spring Cloud 微服务实战》一书后，让我对 Spring Cloud 各个组件的认识提升了一个层次，同时也让我对 Spring Cloud 各个组件的实现原理有了初步的认识，因此我建议所有打算将 Spring Cloud 用于生产的朋友一定要好好读一下这本书。

王天青

DaoCloud 首席架构师

2017 年 3 月

# 推荐序二

2016 年 10 月开始，我在冰鉴科技负责微服务架构迁移相关的调研和筹建工作。我比较了 Dubbo、Dubbox、Motan、Spring Cloud 等框架后，最终锁定在 Spring Cloud 上。这是一个非常年轻的框架，关于它的中文文档少之又少，更不用说有深度的技术干货了。

当我的团队在利用搜索引擎进行相关检索时，永超的技术博客十分显眼地排在了前列，我非常感激他贡献的这个系列的文章，这在我们团队做微服务架构迁移的工作中，起到了关键作用。后来有一天，我俩在一个架构群中相识，一番讨论后发现是博主本人并且他有写书计划时，我告诉了我的团队，我们不谋而合地决定要在该书出版时迅速收入囊中，做到人手一本。而今天对于我来说更是非常荣幸，能够给永超的新书写推荐序。

Spring Cloud 是一个微服务架构实施的综合性解决框架，而在如何构建微服务的选择上，由于我们团队是从 SSM（Spring + Spring MVC + MyBatis）框架开始演进的，基于让演进中改动最小的初衷，我们决定使用 Spring Boot 做微服务构建。我们从对 Spring Boot 的调研开始就一直关注着永超的技术博客，在第一次接触 Spring Boot 的时候就被它“习惯优于配置”的设计概念深深吸引，这无疑简化了做业务逻辑开发同事的工作量，也使得他们可以不用关注配置细节。本书中也有关于 Spring Boot 基础知识的详细讲解以及一个案例工程带你快速构建属于你的第一个微服务。

如开头所述，为了将系统微服务化，我们也一直在对 Spring Cloud 进行相关调研。这本书也是国内市场上为数不多的、全面讲解 Spring Cloud 微服务的中文图书。本书详细讲解了 Spring Cloud 生态的各类组件，涵盖了服务治理组件 Eureka、客户端负载均衡组件 Ribbon、服务容错保护组件 Hystrix、声明式服务调用组件 Feign、API 网关治理组件 Zuul、分布式配置中心组件 Config、消息总线组件 Bus、消息驱动组件 Stream、分布式服务跟踪组件 Sleuth。这包含了我们在实施微服务中需要深入了解的各个轮子，是一本需要仔细

研读、反复阅读的精品之作。

最后，预祝永超在 Spring Cloud 的学习和工作中再创佳绩，也希望读者朋友能够在阅读完本书后快速地搭建好实施微服务过程中的基础脚手架，并在未来工作中能够将团队的一些实践通过 Spring Cloud 中国社区进行交流，为开源贡献自己的一份力量。

朱清

冰鉴科技信息技术部总监

Spring Cloud 中国社区联合创始人

2017.03.27

# 推荐序三

收到本书作者翟永超的邀请为这本书写推荐序，其实我是很谨慎的。抱着对读者负责、对技术严谨的态度，不能在完全不懂 Spring Cloud 的基础上妄加评论。就像 2009 年的云计算和现在的大数据，“Big data is like teenage sex: everyone talks about it, nobody really knows how to do it, everyone thinks everyone else is doing it, so everyone claims they are doing it.”。所以我概读了书中的内容，的确是一本好书，特别是在基于技术实践的阐述中又不失对“微服务化”理论层面的讲解以及发展演进过程的说明。

结合在云计算行业中为大量企业级客户做的服务案例，“集中化”的系统架构确实在企业级客户业务中受到越来越多的挑战，随着业务变化对 IT 需求的不断增加，处于逐渐失控的状态。CIO 们受到越来越大的挑战，希望做到数据驱动业务，那第一个阶段就要做去中心化的改造。如书中所阐述，“微服务化”其实并不是简单的技术革新，而是对团队组织，系统架构，系统研发，自动化测试、发布、运维都提出了一系列的变革要求。所以我觉得，不管是架构师、运维经理、研发主管还是 CIO 都可以从本书中有所收获。

同样，阿里云的企业级中间件 EDAS（基于阿里系的 Dubbo 开源项目）配合强大的飞天云平台与 Docker 服务的支持，在大中型企业客户业务中得到更多的验证，如森马服饰、来伊份、正佳广场、中石化的易派客电商平台等。与这些商业化的中间件产品相比，Spring Cloud 得到了更多热衷开源项目的人的支持，相信在有足够团队技术能力的保障下，也会取得越来越多的成功案例。书如其人，值得认真拜读，我会推荐给更多的人，为翟永超点赞。

李俊涛

上海驻云科技执行总监

# 推荐语

本书从时下流行的微服务架构概念出发，结合 Spring Cloud 的解决方案，深入浅出地剖析了其在构建微服务架构中所需的各个基础设施和技术要点，包括服务治理、容错保护、API 网关、配置管理、消息总线等。作者不仅对如何使用各个组件做了详细介绍，还从原理上做了很多分析，可以帮助读者更好地理解 Spring Cloud 的运行原理，这有助于我们在实战中有效地排错和做进一步扩展。本书是微服务架构方面非常不错的实战书籍，强烈推荐正在做微服务实践或打算实施微服务的团队作为参考资料。

南志文

百联全渠道研发总监

Spring Cloud 的诞生对于没有足够资金投入或者技术储备实力的技术团队是一种福音。利用 Spring Cloud 的一站式解决方案，可以很轻松地搭建起微服务架构的软件系统，大大减少了开发成本，从容应对业务的快速发展。本书是国内第一本 Spring Cloud 的实战书籍，给我带来了无限惊喜。作者由浅入深地讲解了基于 Spring Cloud 构建微服务所需要的各个核心组件，并配有大量实战代码，理论和实践兼备，读后收获颇丰。强烈推荐给每一位对 Spring Cloud 感兴趣或是打算使用 Spring Cloud 的技术人员阅读。

覃罗春

德比软件产品开发负责人

当下最火的词无外乎就是“微服务”了，但是很多创业公司想要实现微服务架构体系还需要做很多方面的工作才可以逐步实现，所需花费的成本还是较大的。而近年来 Spring

Boot/Cloud 生态架构体系的出现为行业提供了一站式解决方案，解决了不少公司的架构选型和维护方面的难题。本书是国内第一本以Spring Cloud为技术蓝本的微服务类实战书籍，不仅结合实际案例介绍了 Spring Cloud 的使用，还从源码的角度深入分析了原理实现，强烈推荐每一位开发者和架构师收藏和学习。

程超

合众支付资深技术专家

随着微服务架构的兴起，企业 IT 架构开始变革，国内出现首批微服务实战布道者。本书作者翟永超，作为 Spring Cloud 中国社区联合发起人和国内首批 Spring Cloud 实践与布道者，发表的博文超过数百万次访问量。他的《Spring Cloud 微服务实战》一书，包含大量生产实战经验，把 Spring Cloud 常用组件通过案例剖析，可帮助企业和开发者快速实施微服务架构。

许进（xujin.org）

Spring Cloud 中国社区创始人

中间件高级研发工程师

近几年，微服务概念逐渐深入人心，国内各家互联网公司都引入了相应的实践。而被应用最多的就是 Spring Cloud 这套被戏称为“全家桶”的微服务框架。它几乎实现了微服务的所有功能，而且又完美符合微服务的基础理论，可帮助大家提高工作效率。但是，国内关于 Spring Cloud 的中文资料相对比较匮乏，很多学习者遍寻入门而不得。在此大背景下，有一些有识之士无私贡献了自己的绵薄之力，本书作者翟永超就是其中一位。这本书对于广大需要在公司中实践微服务的人们来说绝对是一本可以快速上手实现微服务的工作手册。我希望这本书犹如一颗种子，能在国内互联网环境的土壤中生根发芽，最后变成一棵参天大树。

吴峻申

上海青客机器人有限公司架构师

# 前言

“微服务”架构在这几年被广泛传播，变得非常火热，以至于关于微服务架构相关的开源框架和工具都变得越来越活跃，比如：Netflix OSS、Dubbo、Apache Thrift 等。Spring Cloud 也因为 Spring 社区在企业应用领域的广泛知名度和强大影响力，受到了广大架构师与开发者的高度关注。

从接触 Spring Cloud 开始，我除了被其庞大的项目结构震撼之外，还被其所要完成的远大目标所吸引。该项目不同于其他 Spring 的优秀项目，它不再是一个基础框架类，而是一个更高层次的、架构视角的综合性大型项目，其目标旨在构建一套标准化的微服务解决方案，让架构师、开发者在使用微服务理念构建应用系统的时候，面对各个环节的问题都可以找到相应的组件来处理。引用网友戏称的一个比喻：Spring Cloud 可以说是 Spring 社区为微服务架构提供的一个“全家桶”套餐。由于“套餐”中的组件通过一个社区进行包装与整合，使得“套餐”中各个组件之间的配合变得更加和谐，这可以有效减少我们在组件的选型和整合上花费的精力，所以它可以帮助我们快速构建起基础的微服务架构系统。

虽然，Spring Cloud 提供了很多我们期待的内容，但是因其涵盖的内容非常广泛，并且知识跨度较大，因此对于很多初学者来说就像被专业名词轰炸了一样，入门的难度也就大大提高了。同时，中文文档与资料的匮乏，以及官方文档的内容对于使用描述并不够细致等问题，也直接提升了使用者的学习门槛。这些看似都不是什么大问题，但是却在一定程度上阻碍了 Spring Cloud 在国内的推广与发展，毕竟任何一项优秀技术都需要有大批的实践者才能得到不断优化、完善和发扬光大。作为一名 Spring 社区的忠实粉丝和长期实践者，自然希望可以有更多的开发者可以参与到 Spring Cloud 的使用和贡献中来，笔者也就萌生了想要编写一些入门文章的念头，一方面对自身知识的掌握做一些整理，另一方面也希望这些内容可以成为后来者的学习资料。于是就开始坚持着写了一些基础的入门文章和示例，没有想到会受到不少 Spring 爱好者的持续关注，在创建了相关的 QQ 交流群之后，短短一个月的时间，交流群的人数就突破了 1000 人。由于在交流过程中发现很多问题重复出现，

而这些问题并没有得到很好的整理，也没有办法被搜索引擎收录，于是就创建了 Spring Cloud 中文社区论坛，以帮助收集交流过程中提出和解决的各种问题，方便将来学习者可以搜索到这些前人踩过的坑。

之后，有幸在电子工业出版社计算机出版分社的张春雨先生的邀请下，开始编写这本关于 Spring Cloud 的入门书籍。在这本书的编写期间，由于工作、家庭等因素，使得与大家交流的时间变得越来越少，但好在有诸多网友和热心爱好者帮忙一起维护着交流群与论坛，为大家提供了很多宝贵的学习资源，我也从中得到了不少启发和收获。同时，感谢后来建议并牵头整合目前国内 Spring Cloud 学习资源的许进，他在此期间承担了很多沟通和网站维护工作，为 Spring Cloud 中国社区付出了不少精力，后续我也会重新加入进来，继续编写在线免费入门教程，以帮助更多的爱好者快速入门 Spring Cloud。我们也欢迎更多的爱好者参与进来，一起为社区贡献力量，以帮助 Spring Cloud 在国内被更好地应用与成长。

---

轻松注册成为博文视点社区用户（www.broadview.com.cn），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/31301

二维码：

# 目录

第 1 章　基础知识 ..... 1
什么是微服务架构 ..... 1
与单体系统的区别 ..... 1
如何实施微服务 ..... 2
为什么选择 Spring Cloud ..... 6
Spring Cloud 简介 ..... 7
版本说明 ..... 8
第 2 章　微服务构建：Spring Boot ..... 11
框架简介 ..... 12
快速入门 ..... 13
项目构建与解析 ..... 13
实现 RESTful API ..... 17
配置详解 ..... 20
配置文件 ..... 20
自定义参数 ..... 22
参数引用 ..... 22
使用随机数 ..... 23
命令行参数 ..... 23
多环境配置 ..... 24
加载顺序 ..... 25

监控与管理 ..... 26
初识 actuator ..... 27
原生端点 ..... 28
小结 ..... 38

第 3 章　服务治理：Spring Cloud Eureka ..... 39

服务治理 ..... 39
Netflix Eureka ..... 40
搭建服务注册中心 ..... 41
注册服务提供者 ..... 43
高可用注册中心 ..... 46
服务发现与消费 ..... 48
Eureka 详解 ..... 51
基础架构 ..... 52
服务治理机制 ..... 52
源码分析 ..... 56
配置详解 ..... 65
服务注册类配置 ..... 65
服务实例类配置 ..... 67
跨平台支持 ..... 71

第 4 章　客户端负载均衡：Spring Cloud Ribbon ..... 73

客户端负载均衡 ..... 73
RestTemplate 详解 ..... 75
GET 请求 ..... 75
POST 请求 ..... 77
PUT 请求 ..... 79
DELETE 请求 ..... 79
源码分析 ..... 80
负载均衡器 ..... 91
负载均衡策略 ..... 109
配置详解 ..... 123

自动化配置 ..... 124
Camden 版本对 RibbonClient 配置的优化 ..... 125
参数配置 ..... 127
与 Eureka 结合 ..... 127
重试机制 ..... 128

第 5 章　服务容错保护：Spring Cloud Hystrix ..... 130

快速入门 ..... 131
原理分析 ..... 135
工作流程 ..... 135
断路器原理 ..... 144
依赖隔离 ..... 148
使用详解 ..... 151
创建请求命令 ..... 151
定义服务降级 ..... 154
异常处理 ..... 157
命令名称、分组以及线程池划分 ..... 158
请求缓存 ..... 159
请求合并 ..... 166
属性详解 ..... 172
Command 属性 ..... 174
collapser 属性 ..... 184
threadPool 属性 ..... 185
Hystrix 仪表盘 ..... 187
Turbine 集群监控 ..... 192
构建监控聚合服务 ..... 192
与消息代理结合 ..... 196

第 6 章　声明式服务调用：Spring Cloud Feign ..... 199

快速入门 ..... 200
参数绑定 ..... 202
继承特性 ..... 205

Ribbon 配置 ..... 209
全局配置 ..... 209
指定服务配置 ..... 209
重试机制 ..... 210
Hystrix 配置 ..... 211
全局配置 ..... 211
禁用 Hystrix ..... 211
指定命令配置 ..... 212
服务降级配置 ..... 212
其他配置 ..... 214
请求压缩 ..... 214
日志配置 ..... 215

第 7 章　API 网关服务：Spring Cloud Zuul ..... 217

快速入门 ..... 219
构建网关 ..... 220
请求路由 ..... 221
请求过滤 ..... 223
路由详解 ..... 226
传统路由配置 ..... 226
服务路由配置 ..... 228
服务路由的默认规则 ..... 229
自定义路由映射规则 ..... 229
路径匹配 ..... 230
路由前缀 ..... 233
本地跳转 ..... 234
Cookie 与头信息 ..... 235
Hystrix 和 Ribbon 支持 ..... 236
过滤器详解 ..... 238
过滤器 ..... 238
请求生命周期 ..... 239
核心过滤器 ..... 240
异常处理 ..... 244

禁用过滤器 ..... 256
动态加载 ..... 257
动态路由 ..... 257
动态过滤器 ..... 261

第 8 章　分布式配置中心：Spring Cloud Config ..... 267

快速入门 ..... 267
构建配置中心 ..... 268
配置规则详解 ..... 269
客户端配置映射 ..... 272
服务端详解 ..... 274
基础架构 ..... 274
Git 配置仓库 ..... 276
SVN 配置仓库 ..... 279
本地仓库 ..... 279
本地文件系统 ..... 279
健康监测 ..... 280
属性覆盖 ..... 281
安全保护 ..... 281
加密解密 ..... 282
高可用配置 ..... 286
客户端详解 ..... 286
URI 指定配置中心 ..... 287
服务化配置中心 ..... 287
失败快速响应与重试 ..... 290
获取远程配置 ..... 292
动态刷新配置 ..... 293

第 9 章　消息总线：Spring Cloud Bus ..... 295

消息代理 ..... 295
RabbitMQ 实现消息总线 ..... 296
基本概念 ..... 297

安装与使用 ..... 298
快速入门 ..... 302
整合 Spring Cloud Bus ..... 306
原理分析 ..... 307
指定刷新范围 ..... 308
架构优化 ..... 309
RabbitMQ 配置 ..... 310
Kafka 实现消息总线 ..... 312
Kafka 简介 ..... 312
快速入门 ..... 313
整合 Spring Cloud Bus ..... 315
Kafka 配置 ..... 318
深入理解 ..... 318
源码分析 ..... 320
其他消息代理的支持 ..... 342

第 10 章　消息驱动的微服务：Spring Cloud Stream ..... 344

快速入门 ..... 344
核心概念 ..... 349
绑定器 ..... 350
发布-订阅模式 ..... 351
消费组 ..... 353
消息分区 ..... 354
使用详解 ..... 355
开启绑定功能 ..... 355
绑定消息通道 ..... 356
消息生产与消费 ..... 360
响应式编程 ..... 366
消费组与消息分区 ..... 368
消息类型 ..... 370
绑定器详解 ..... 373
绑定器 SPI ..... 373
自动化配置 ..... 374

多绑定器配置 ..... 374
RabbitMQ 与 Kafka 绑定器 ..... 376
配置详解 ..... 376
基础配置 ..... 377
绑定通道配置 ..... 377
绑定器配置 ..... 379

第 11 章　分布式服务跟踪：Spring Cloud Sleuth ..... 386

快速入门 ..... 386
准备工作 ..... 386
实现跟踪 ..... 389
跟踪原理 ..... 390
抽样收集 ..... 392
与 Logstash 整合 ..... 394
与 Zipkin 整合 ..... 397
HTTP 收集 ..... 398
消息中间件收集 ..... 402
收集原理 ..... 404
数据存储 ..... 414
API 接口 ..... 417

附录 A　Starter POMs ..... 419

后记 ..... 421

# 第 1 章

# 基础知识

在进行 Spring Cloud 的具体内容介绍之前，我们先通过本章学习一些关于微服务架构以及 Spring Cloud 的基础知识。对 Spring Cloud 能够解决的具体问题有一个大致的了解，以帮助我们更好地理解后续章节对各个组件的介绍。

## 什么是微服务架构

“微服务”一词源于 Martin Fowler 的名为 *Microservices* 的博文，可以在他的官方博客上找到：`http://martinfowler.com/articles/microservices.html`。

简单地说，微服务是系统架构上的一种设计风格，它的主旨是将一个原本独立的系统拆分成多个小型服务，这些小型服务都在各自独立的进程中运行，服务之间通过基于 HTTP 的 RESTful API 进行通信协作。被拆分成的每一个小型服务都围绕着系统中的某一项或一些耦合度较高的业务功能进行构建，并且每个服务都维护着自身的数据存储、业务开发、自动化测试案例以及独立部署机制。由于有了轻量级的通信协作基础，所以这些微服务可以使用不同的语言来编写。

### 与单体系统的区别

在以往传统的企业系统架构中，我们针对一个复杂的业务需求通常使用对象或业务类型来构建一个单体项目。在项目中我们通常将需求分为三个主要部分：数据库、服务端处理、前端展现。在业务发展初期，由于所有的业务逻辑在一个应用中，开发、测试、部署都还比较容易且方便。但是，随着企业的发展，系统为了应对不同的业务需求会不断为该单体项目增加不同的业务模块；同时随着移动端设备的进步，前端展现模块已经不仅仅局限于 Web 的形式，这对于系统后端向前端的支持需要更多的接口模块。单体应用由于面对的业务需求更为宽泛，不断扩大的需求会使得单体应用变得越来越臃肿。单体应用的问题

就逐渐凸显出来，由于单体系统部署在一个进程内，往往我们修改了一个很小的功能，为了部署上线会影响其他功能的运行。并且，单体应用中的这些功能模块的使用场景、并发量、消耗的资源类型都各有不同，对于资源的利用又互相影响，这样使得我们对各个业务模块的系统容量很难给出较为准确的评估。所以，单体系统在初期虽然可以非常方便地进行开发和使用，但是随着系统的发展，维护成本会变得越来越大，且难以控制。

为了解决单体系统变得庞大臃肿之后产生的难以维护的问题，微服务架构诞生了并被大家所关注。我们将系统中的不同功能模块拆分成多个不同的服务，这些服务都能够独立部署和扩展。由于每个服务都运行在自己的进程内，在部署上有稳固的边界，这样每个服务的更新都不会影响其他服务的运行。同时，由于是独立部署的，我们可以更准确地为每个服务评估性能容量，通过配合服务间的协作流程也可以更容易地发现系统的瓶颈位置，以及给出较为准确的系统级性能容量评估。

## 如何实施微服务

在实施微服务之前，我们必须要知道，微服务虽然有非常多吸引人的优点，但是也因为服务的拆分引发了诸多原本在单体应用中没有的问题。

- **运维的新挑战：**在微服务架构中，运维人员需要维护的进程数量会大大增加。有条不紊地将这些进程编排和组织起来不是一件容易的事，传统的运维人员往往很难适应这样的改变。我们需要运维人员有更多的技能来应对这样的挑战，运维过程需要更多的自动化，这就要求运维人员具备一定的开发能力来编排运维过程并让它们能自动运行起来。
- **接口的一致性：**虽然我们拆分了服务，但是业务逻辑上的依赖并不会消除，只是从单体应用中的代码依赖变为了服务间的通信依赖。而当我们对原有接口进行了一些修改，那么交互方也需要协调这样的改变来进行发布，以保证接口的正确调用。我们需要更完善的接口和版本管理，或是严格地遵循开闭原则。
- **分布式的复杂性：**由于拆分后的各个微服务都是独立部署并运行在各自的进程内，它们只能通过通信来进行协作，所以分布式环境的问题都将是微服务架构系统设计时需要考虑的重要因素，比如网络延迟、分布式事务、异步消息等。

尽管微服务架构有很多缺点和问题，但是其实现的敏捷开发和自动化部署等优点依然被广大优秀架构师和开发者所青睐，所以解决这些问题就是这几年诸多架构大师努力的目标。

在架构师对于一个大型系统架构的设计与实施的过程中，面对环境、资源、团队等各种因素的影响，几乎不会出现完全相同的架构设计。对于微服务架构而言更是如此，由于

并没有一个标准或正式的定义，每位架构师都根据自身理解与实际情况来进行设计，并在发展的过程中不断演化与完善。经过多年的发展，Martin Fowler 在 *Microservices* 一文中，提炼出了微服务架构的九大特性，用于指导大家设计架构。

**服务组件化**

组件，是一个可以独立更换和升级的单元。就像 PC 中的 CPU、内存、显卡、硬盘一样，独立且可以更换升级而不影响其他单元。

在微服务架构中，需要我们对服务进行组件化分解。服务，是一种进程外的组件，它通过 HTTP 等通信协议进行协作，而不是像传统组件那样以嵌入的方式协同工作。每一个服务都独立开发、部署，可以有效避免一个服务的修改引起整个系统的重新部署。

打一个不恰当的比喻，如果我们的 PC 组件以服务的方式构建，那么只维护主板和一些必要外设之后，计算能力通过一组外部服务实现，我们只需要告诉 PC 从哪个地址来获得计算能力，通过服务定义的计算接口来实现我们使用过程中的计算需求，从而实现 CPU 组件的服务化。这样原本复杂的 PC 服务得到了轻量化的实现，我们甚至只需要更换服务地址就能升级 PC 的计算能力。

**按业务组织团队**

当决定如何划分微服务时，通常也意味着我们要开始对团队进行重新规划与组织。按以往的方式，我们往往会从技术的层面将团队划分为多个，比如 DBA 团队、运维团队、后端团队、前端团队、设计师团队等。若我们继续按这种方式组织团队来实施微服务架构开发，当有一个服务出现问题需要更改时，可能是一个非常简单的变动，比如对人物描述增加一个字段，这需要从数据存储开始考虑一直到设计和前端，虽然大家的修改都非常小，但这会引起跨团队的时间耗费和预算审批。

在实施微服务架构时，需要采用不同的团队分割方法。由于每一个微服务都是针对特定业务的宽栈或是全栈实现，既要负责数据的持久化存储，又要负责用户的接口定义等各种跨专业领域的职能。因此，面对大型项目的时候，对于微服务团队的拆分更加建议按业务线的方式进行拆分，一方面可以有效减少服务内部修改所产生的内耗；另一方面，团队边界可以变得更为清晰。

**做“产品”的态度**

在实施微服务架构的团队中，每个小团队都应该以做产品的方式，对其产品的整个生命周期负责。而不是以项目的模式，以完成开发与交付并将成果交接给维护者为最终目标。

开发团队通过了解服务在具体生产环境中的情况，可以增加他们对具体业务的理解，比如，很多时候，一些业务中发生的特殊或异常情况，很可能产品经理都并不知晓，但细心的开发者很容易通过生产环境发现这些特殊的潜在问题或需求。

所以，我们需要用做“产品”的态度来对待每一个微服务，持续关注服务的运作情况，并不断分析以帮助用户来改善业务功能。

### 智能端点与哑管道

在单体应用中，组件间直接通过函数调用的方式进行交互协作。而在微服务架构中，由于服务不在一个进程中，组件间的通信模式发生了改变，若仅仅将原本在进程内的方法调用改成 RPC 方式的调用，会导致微服务之间产生烦琐的通信，使得系统表现更为糟糕，所以，我们需要更粗粒度的通信协议。

在微服务架构中，通常会使用以下两种服务调用方式：

- 第一种，使用 HTTP 的 RESTful API 或轻量级的消息发送协议，实现信息传递与服务调用的触发。
- 第二种，通过在轻量级消息总线上传递消息，类似 RabbitMQ 等一些提供可靠异步交换的中间件。

> 在极度强调性能的情况下，有些团队会使用二进制的消息发送协议，例如 protobuf。即使是这样，这些系统仍然会呈现出“智能端点和哑管道”的特点，这是为了在易读性与高效性之间取得平衡。当然大多数 Web 应用或企业系统并不需要在这两者间做出选择，能够获得易读性已经是一个极大的胜利了。
>
> ——Martin Fowler

### 去中心化治理

当我们采用集中化的架构治理方案时，通常在技术平台上都会制定统一的标准，但是每一种技术平台都有其短板，这会导致在碰到短板时，不得不花费大力气去解决，并且可能因为其底层原因解决得不是很好，最终成为系统的瓶颈。

在实施微服务架构时，通过采用轻量级的契约定义接口，使得我们对于服务本身的具体技术平台不再那么敏感，这样整个微服务架构系统中的各个组件就能针对其不同的业务特点选择不同的技术平台，终于不会出现杀鸡用牛刀或是杀牛用指甲钳的尴尬处境了。

> 不是每一个问题都是钉子，不是每一个解决方案都是锤子。

### 去中心化管理数据

我们在实施微服务架构时，都希望让每一个服务来管理其自有的数据库，这就是数据管理的去中心化。

在去中心化过程中，我们除了将原数据库中的存储内容拆分到新的同平台的其他数据库实例中之外(如把原本存储在MySQL中的表拆分后，存储到多个不同的MySQL实例中)，也可以将一些具有特殊结构或业务特性的数据存储到一些其他技术的数据库实例中（如把日志信息存储到 MongoDB 中或把用户登录信息存储到 Redis 中）。

虽然数据管理的去中心化可以让数据管理更加细致化，通过采用更合适的技术可让数据存储和性能达到最优。但是，由于数据存储于不同的数据库实例中后，数据一致性也成为微服务架构中亟待解决的问题之一。分布式事务本身的实现难度就非常大，所以在微服务架构中，我们更强调在各服务之间进行“无事务”的调用，而对于数据一致性，只要求数据在最后的处理状态是一致的即可；若在过程中发现错误，通过补偿机制来进行处理，使得错误数据能够达到最终的一致性。

### 基础设施自动化

近年来云计算服务与容器化技术的不断成熟，运维基础设施的工作变得越来越容易。但是，当我们实施微服务架构时，数据库、应用程序的个头虽然都变小了，但是因为拆分的原因，数量成倍增长。这使得运维人员需要关注的内容也成倍增长，并且操作性任务也会成倍增长，这些问题若没有得到妥善解决，必将成为运维人员的噩梦。

所以，在微服务架构中，务必从一开始就构建起“持续交付”平台来支撑整个实施过程，该平台需要两大内容，缺一不可。

- **自动化测试：**每次部署前的强心剂，尽可能地获得对正在运行的软件的信心。
- **自动化部署：**解放烦琐枯燥的重复操作以及对多环境的配置管理。

### 容错设计

在单体应用中，一般不存在单个组件故障而其他部件还在运行的情况，通常是一挂全挂。而在微服务架构中，由于服务都运行在独立的进程中，所以存在部分服务出现故障，而其他服务正常运行的情况。比如，当正常运作的服务 B 调用到故障服务 A 时，因故障服务 A 没有返回，线程挂起开始等待，直到超时才能释放，而此时若触发服务 B 调用服务 A 的请求来自服务 C，而服务 C 频繁调用服务 B 时，由于其依赖服务 A，大量线程被挂起等待，最后导致服务 A 也不能正常服务，这时就会出现故障的蔓延。

所以，在微服务架构中，快速检测出故障源并尽可能地自动恢复服务是必须被设计和

考虑的。通常，我们都希望在每个服务中实现监控和日志记录的组件，比如服务状态、断路器状态、吞吐量、网络延迟等关键数据的仪表盘等。

### 演进式设计

通过上面的几点特征，我们已经能够体会到，要实施一个完美的微服务架构，需要考虑的设计与成本并不小，对于没有足够经验的团队来说，甚至要比单体应用付出更多的代价。

所以，在很多情况下，架构师都会以演进的方式进行系统的构建。在初期，以单体系统的方式来设计和实施，一方面系统体量初期并不会很大，构建和维护成本都不高。另一方面，初期的核心业务在后期通常也不会发生巨大的改变。随着系统的发展或者业务的需要，架构师会将一些经常变动或是有一定时间效应的内容进行微服务处理，并逐渐将原来在单体系统中多变的模块逐步拆分出来，而稳定不太变化的模块就形成一个核心微服务存在于整个架构之中。

## 为什么选择 Spring Cloud

近几年很多人对于微服务架构的热情非常高，但是回头看“微服务”被提及也有很多年了。无数的架构师和开发者在实际项目中实践该设计理念并为此付出了诸多努力，同时也分享了他们在微服务架构中针对不同应用场景出现的各种问题的各种解决方案和开源框架，其中也不乏国内互联网企业的杰出贡献。

- **服务治理：**阿里巴巴开源的 Dubbo 和当当网在其基础上扩展的 DubboX、Netflix 的 Eureka、Apache 的 Consul 等。
- **分布式配置管理：**百度的 Disconf、Netflix 的 Archaius、360 的 QConf、Spring Cloud 的 Config、淘宝的 Diamond 等。
- **批量任务：**当当网的 Elastic-Job、LinkedIn 的 Azkaban、Spring Cloud 的 Task 等。
- **服务跟踪：**京东的 Hydra、Spring Cloud 的 Sleuth、Twitter 的 Zipkin 等。
- ……

上面列举了一些在实施微服务架构初期，就需要被我们考虑进去的问题，以及针对这些问题的开源解决方案。可以看到国内、国外的技术公司都在贡献着他们的智慧。我们搜索微服务架构的实施方案时会发现，几乎大部分的分享主要以理论或是一个粗轮廓框架为主，整合了来自不同公司或组织的诸多开源框架，并加入针对自身业务的一些优化，所以找不到一个完全相同的架构方案。

前面我们介绍了一些关于微服务的理念以及特性，分析了实施微服务的优点和缺点，

而这些缺点通常就是这些框架出现的源头，大家都是为了解决或弥补业务拆分后所引出的诸多问题来设计出这些解决方案。而当我们作为一个新手，准备实施微服务架构时，为了避免踩前辈们踩过的坑，我们不得不在这些核心问题上做出选择，而选择又是如此之多，这必然会导致在做技术选型的初期，需要花费巨大的调研、分析与实验精力。

Spring Cloud 的出现，可以说是对微服务架构的巨大支持和强有力的技术后盾。它不像我们之前所列举的框架那样，只是解决微服务中的某一个问题，而是一个解决微服务架构实施的综合性解决框架，它整合了诸多被广泛实践和证明过的框架作为实施的基础部件，又在该体系基础上创建了一些非常优秀的边缘组件。

打个不太恰当的比喻：我们自己对各个问题选择框架来实施微服务架构就像在 DIY 电脑一样，我们对各环节的选择自由度很高，但是最终结果很有可能因为一条内存质量不行就点不亮了，总是让人不怎么放心。当然，如果你是一名高手，这些自然都不是问题，然而千军易得、良将难求。而使用 Spring Cloud 来实施就像直接购买品牌机一样，在 Spring 社区的整合之下，做了大量的兼容性测试，保证了其拥有更好的稳定性，如果要在 Spring Cloud 架构下使用非原装组件时，就需要对其基础有足够的了解。

Spring Cloud 也许对很多已经实施微服务并自成体系的团队不具备足够的吸引力，但是对于还未实施微服务或是未成体系的团队，这必将是一个非常有吸引力的框架选择。不论其项目的发展目标，还是 Spring 的强大背景，亦或其极高的社区活跃度，都是未来企业架构师必须了解和接触的重要框架，有一天成为微服务架构的标准解决方案也并非不可能。

## Spring Cloud 简介

Spring Cloud 是一个基于 Spring Boot 实现的微服务架构开发工具。它为微服务架构中涉及的配置管理、服务治理、断路器、智能路由、微代理、控制总线、全局锁、决策竞选、分布式会话和集群状态管理等操作提供了一种简单的开发方式。

Spring Cloud 包含了多个子项目（针对分布式系统中涉及的多个不同开源产品，还可能会新增），如下所述。

- Spring Cloud Config：配置管理工具，支持使用 Git 存储配置内容，可以使用它实现应用配置的外部化存储，并支持客户端配置信息刷新、加密/解密配置内容等。
- Spring Cloud Netflix：核心组件，对多个 Netflix OSS 开源套件进行整合。
  - Eureka：服务治理组件，包含服务注册中心、服务注册与发现机制的实现。
  - Hystrix：容错管理组件，实现断路器模式，帮助服务依赖中出现的延迟和为故障提供强大的容错能力。
  - Ribbon：客户端负载均衡的服务调用组件。

  - Feign：基于 Ribbon 和 Hystrix 的声明式服务调用组件。
  - Zuul：网关组件，提供智能路由、访问过滤等功能。
  - Archaius：外部化配置组件。
- Spring Cloud Bus：事件、消息总线，用于传播集群中的状态变化或事件，以触发后续的处理，比如用来动态刷新配置等。
- Spring Cloud Cluster：针对 ZooKeeper、Redis、Hazelcast、Consul 的选举算法和通用状态模式的实现。
- Spring Cloud Cloudfoundry：与 Pivotal Cloudfoundry 的整合支持。
- Spring Cloud Consul：服务发现与配置管理工具。
- Spring Cloud Stream：通过 Redis、Rabbit 或者 Kafka 实现的消费微服务，可以通过简单的声明式模型来发送和接收消息。
- Spring Cloud AWS：用于简化整合 Amazon Web Service 的组件。
- Spring Cloud Security：安全工具包，提供在 Zuul 代理中对 OAuth2 客户端请求的中继器。
- Spring Cloud Sleuth：Spring Cloud 应用的分布式跟踪实现，可以完美整合 Zipkin。
- Spring Cloud ZooKeeper：基于 ZooKeeper 的服务发现与配置管理组件。
- Spring Cloud Starters：Spring Cloud 的基础组件，它是基于 Spring Boot 风格项目的基础依赖模块。
- Spring Cloud CLI：用于在 Groovy 中快速创建 Spring Cloud 应用的 Spring Boot CLI 插件。
- ……

本书将对其中一些较为常用的组件进行介绍、分析，并演示其使用方法。

## 版本说明

当我们通过搜索引擎查找一些 Spring Cloud 的文章或示例时，往往可以在依赖中看到很多不同的版本名字，比如 Angel.SR6、Brixton.SR5 等，为什么 Spring Cloud 没有像其他 Spring 的项目使用类似 1.x.x 的版本命名规则呢？这些版本之间又有什么区别呢？在学习之初，非常有必要弄清楚这些版本的意义和内容，这样才能在我们使用 Spring Cloud 时，指导我们选择更为合适的版本进行架构与开发。

### 版本名与版本号

由于 Spring Cloud 不像 Spring 社区其他一些项目那样相对独立，它是一个拥有诸多子项目的大型综合项目，可以说是对微服务架构解决方案的综合套件组合，其包含的各个子

项目也都独立进行着内容更新与迭代，各自都维护着自己的发布版本号。因此每一个 Spring Cloud 的版本都会包含多个不同版本的子项目，为了管理每个版本的子项目清单，避免 Spring Cloud 的版本号与其子项目的版本号相混淆，没有采用版本号的方式，而是通过命名的方式。

这些版本的名字采用了伦敦地铁站的名字，根据字母表的顺序来对应版本时间顺序，比如最早的 Release 版本为 Angel，第二个 Release 版本为 Brixton……

经过上面的解释，不难猜出，之前所提到的 Angel.SR6、Brixton.SR5 中的 SR6、SR5 就是版本号了。

当一个版本的 Spring Cloud 项目的发布内容积累到临界点或者一个严重 bug 解决可用后，就会发布一个“service releases”版本，简称 SRX 版本，其中 X 是一个递增的数字，所以 Brixton.SR5 就是 Brixton 的第 5 个 Release 版本。

**版本区别**

下面列出的是开始编写此书时各版本的版本构成表，我们可以快速查阅当前各版本所包含的子项目，以及各子项目的版本号，以此来决定需要选择哪个版本。

| Component | Angel.SR6 | Brixton.SR5 | Camden.M1 | Camden.BUILD-SNAPSHOT |
|---|---|---|---|---|
| spring-cloud-aws | 1.0.4.RELEASE | 1.1.1.RELEASE | 1.1.1.RELEASE | 1.1.2.BUILD-SNAPSHOT |
| spring-cloud-bus | 1.0.3.RELEASE | 1.1.1.RELEASE | 1.2.0.M1 | 1.2.0.BUILD-SNAPSHOT |
| spring-cloud-cli | 1.0.6.RELEASE | 1.1.5.RELEASE | 1.2.0.M1 | 1.2.0.BUILD-SNAPSHOT |
| spring-cloud-commons | 1.0.5.RELEASE | 1.1.1.RELEASE | 1.1.1.RELEASE | 1.1.2.BUILD-SNAPSHOT |
| spring-cloud-contract | | | 1.0.0.M2 | 1.0.0.BUILD-SNAPSHOT |
| spring-cloud-config | 1.0.4.RELEASE | 1.1.3.RELEASE | 1.2.0.M1 | 1.2.0.BUILD-SNAPSHOT |
| spring-cloud-netflix | 1.0.7.RELEASE | 1.1.5.RELEASE | 1.2.0.M1 | 1.2.0.BUILD-SNAPSHOT |
| spring-cloud-security | 1.0.3.RELEASE | 1.1.2.RELEASE | 1.1.2.RELEASE | 1.1.3.BUILD-SNAPSHOT |
| spring-cloud-starters | 1.0.6.RELEASE | | | |
| spring-cloud-cloudfoundry | | 1.0.0.RELEASE | 1.0.0.RELEASE | 1.0.1.BUILD-SNAPSHOT |
| spring-cloud-cluster | | 1.0.1.RELEASE | | |
| spring-cloud-consul | | 1.0.2.RELEASE | 1.1.0.M1 | 1.1.0.BUILD-SNAPSHOT |
| spring-cloud-sleuth | | 1.0.6.RELEASE | 1.0.6.RELEASE | 1.0.7.BUILD-SNAPSHOT |
| spring-cloud-stream | | 1.0.2.RELEASE | Brooklyn.M1 | Brooklyn.BUILD-SNAPSHOT |
| spring-cloud-zookeeper | | 1.0.2.RELEASE | 1.0.2.RELEASE | 1.0.3.BUILD-SNAPSHOT |
| spring-boot | 1.2.8.RELEASE | 1.3.7.RELEASE | 1.4.0.RELEASE | 1.4.0.RELEASE |
| spring-cloud-task | | 1.0.2.RELEASE | 1.0.2.RELEASE | 1.0.3.BUILD-SNAPSHOT |

不难看出，最初的 Angel 版本相对来说拥有的子项目较少，Brixton、Camden 则拥有更

全的子项目，所以能提供更多的组件支持。Brixton 的发布子项目更为稳定，Camden 则更为前瞻。

由于 Brixton 版本包含了大部分的 Spring Cloud 子项目，并且均为 Release 版本，所以本文所有示例以及讲解内容的编写均采用 Brixton.SR5 版本，基于 Spring Boot 1.3.7 版本。

---

**注意：**在本书完成时，Brixton 版本已经升级到 SR7，本书中的示例均可直接使用 Brixton.SR7 来完成，但是在使用 Brixton 版本的时候需要注意 Spring Boot 的版本，必须使用 1.3.x 版本，而不能使用 1.4.x 版本，否则会出现各种问题。若一定要使用 Spring 1.4.x 版本的话，必须将 Spring Cloud 版本升级到 Camden 版本，目前 Camden 已经发布 Release 版本，所以可以放心使用，最新版本为 Camden.SR3。另外，Camden 版本虽然可以兼容 Brixton 版本中的各种实现方法，但是在升级后读者会发现有一些方法已经被标注为过期，对于这些过期的使用方法，笔者后续会在博客中逐个编写博文来说明这些更新内容。所以，有兴趣的读者可以关注我的博客（http://blog.didispace.com），来持续获取 Spring Cloud 的一些最新动态。

---

# 第 2 章

# 微服务构建：Spring Boot

在展开 Spring Cloud 的微服务架构部署之前，我们先通过本章的内容来了解一下用于构建微服务的基础框架——Spring Boot。对于 Spring Boot 已经有深入了解的读者可以直接跳过本章，进入后续章节学习 Spring Cloud 各个组件的使用。

如果对于 Spring Boot 还不了解的话，建议先读完本章内容之后再继续学习后续关于 Spring Cloud 的内容。由于 Spring Cloud 的构建基于 Spring Boot 实现，在后续的示例中我们将大量使用 Spring Boot 来构建微服务架构中的基础设施以及一些试验中使用的微服务。为了能够辅助后续内容的介绍，确保读者有一定的 Spring Boot 基础，在这里先对 Spring Boot 做一个简单的介绍，以保证读者能够有一定的基础去理解后续介绍的内容并顺利完成后续的一些示例试验。

在这里介绍 Spring Boot 的目的除了它是 Spring Cloud 的基础之外，也由于其自身的各项优点，如自动化配置、快速开发、轻松部署等，非常适合用作微服务架构中各项具体微服务的开发框架。所以我们强烈推荐使用 Spring Boot 来构建微服务，它不仅可以帮助我们快速地构建微服务，还可以轻松简单地整合 Spring Cloud 实现系统服务化，而如果使用了传统的 Spring 构建方式的话，在整合过程中我们还需要做更多的依赖管理工作才能让它们完好地运行起来。

在本章中我们将介绍下面这些与后续介绍有密切联系的内容：

- 如何构建 Spring Boot 项目
- 如何实现 RESTful API 接口
- 如何实现多环境的 Spring Boot 应用配置
- 深入理解 Spring Boot 配置的启动机制
- Spring Boot 应用的监控与管理

更多关于 Spring Boot 的使用细节，读者可以参阅 Spring Boot 的官方文档或是其他资料来进一步学习。

## 框架简介

对于很多 Spring 框架的初学者来说，经常会因为其繁杂的配置文件而却步。而对于很多老手来说，每次新构建项目总是会重复复制粘贴一些差不多的配置文件这样枯燥乏味的事。作为一名优秀的程序员或架构师，我们总会想尽办法来避免这样的重复劳动，比如，通过 Maven 等构建工具来创建针对不同场景的脚手架工程，在需要新建项目时通过这些脚手架来初始化我们自定义的标准工程，并根据需要做一些简单修改以达到简化原有配置过程的效果。这样的做法虽然减少了工作量，但是这些配置依然大量散布在我们的工程中，大部分情况下我们并不会去修改这些内容，但为什么还要反复出现在我们的工程中呢？实在有些碍眼！

Spring Boot 的出现可以有效改善这类问题，Spring Boot 的宗旨并非要重写 Spring 或是替代 Spring，而是希望通过设计大量的自动化配置等方式来简化 Spring 原有样板化的配置，使得开发者可以快速构建应用。

除了解决配置问题之外，Spring Boot 还通过一系列 Starter POMs 的定义，让我们整合各项功能的时候，不需要在 Maven 的 `pom.xml` 中维护那些错综复杂的依赖关系，而是通过类似模块化的 Starter 模块定义来引用，使得依赖管理工作变得更为简单。

在如今容器化大行其道的时代，Spring Boot 除了可以很好融入 Docker 之外，其自身就支持嵌入式的 Tomcat、Jetty 等容器。所以，通过 Spring Boot 构建的应用不再需要安装 Tomcat，将应用打包成 war，再部署到 Tomcat 这样复杂的构建与部署动作，只需将 Spring Boot 应用打成 jar 包，并通过 `java -jar` 命令直接运行就能启动一个标准化的 Web 应用，这使得 Spring Boot 应用变得非常轻便。

Spring Boot 对于构建、部署等做了这么多的优化，自然不能少了对开发环节的优化。整个 Spring Boot 的生态系统都使用到了 Groovy，很自然的，我们完全可以通过使用 Gradle 和 Groovy 来开发 Spring Boot 应用，比如下面短短的不足 100 个字符的代码，通过编译打包，使用 `java -jar` 命令就能启动一个返回“hello”的 RESTful API。

```
@RestController
class App {

    @RequestMapping("/")
    String home() {
```

```
        "hello"
    }

}
```

说了这么多 Spring Boot 带来的颠覆性框架特性，下面我们就通过后续内容来体验一下使用 Spring Boot 构建微服务的过程，以对 Spring Boot 有一个直观的感受。

# 快速入门

在本节中，我们将逐步指引读者创建一个 Spring Boot 的基础项目，并且实现一个简单的 RESTful API，通过这个例子对 Spring Boot 有一个初步的了解，并体验其结构简单、开发迅速的特性。

## 项目构建与解析

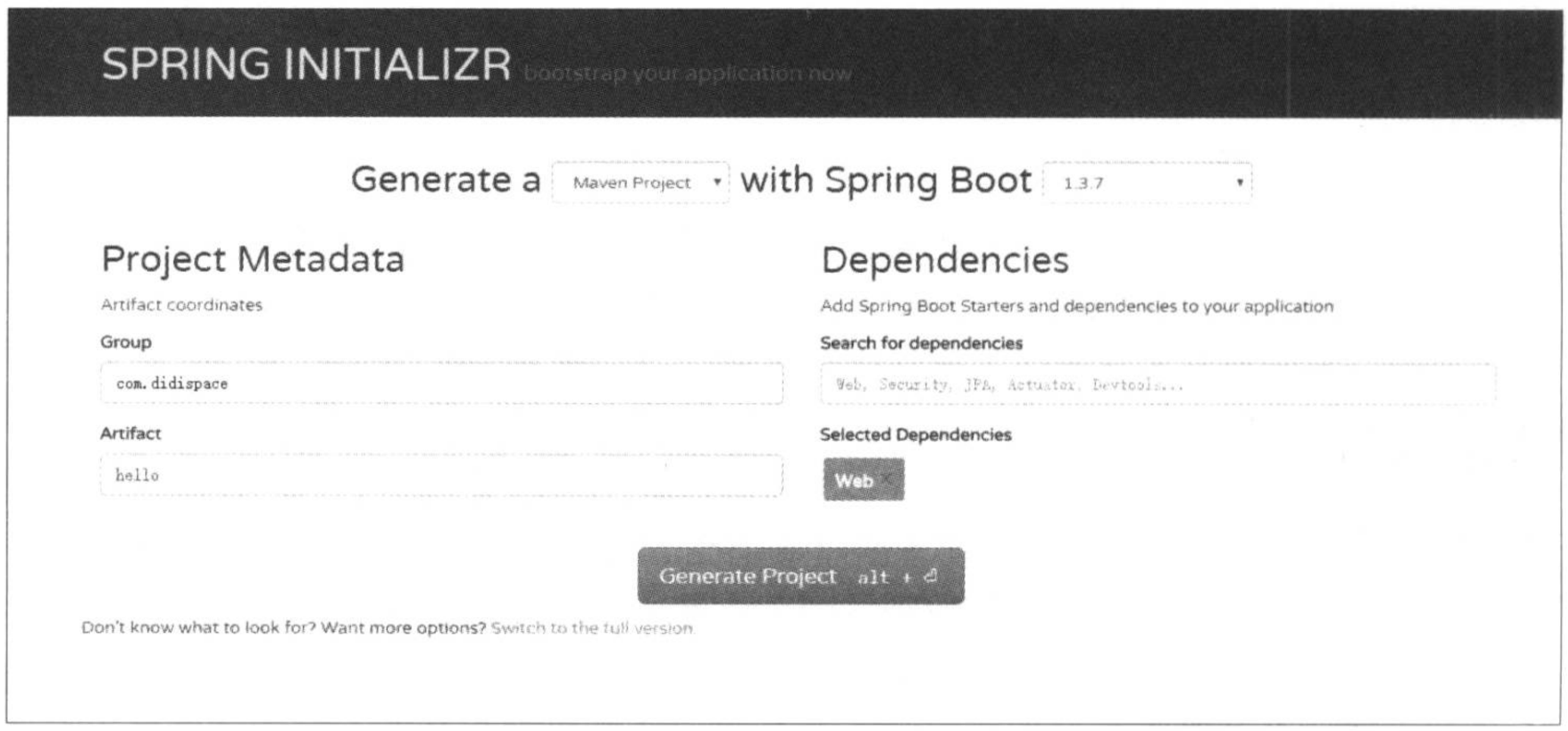
### 系统及工具版本要求

- Java 7 及以上版本
- Spring Framework 4.2.7 及以上版本
- Maven 3.2 及以上版本 / Gradle 1.12 及以上版本

本书内容均采用 Java 1.8.0_73、Spring Boot 1.3.7 调试通过。

### 构建 Maven 项目

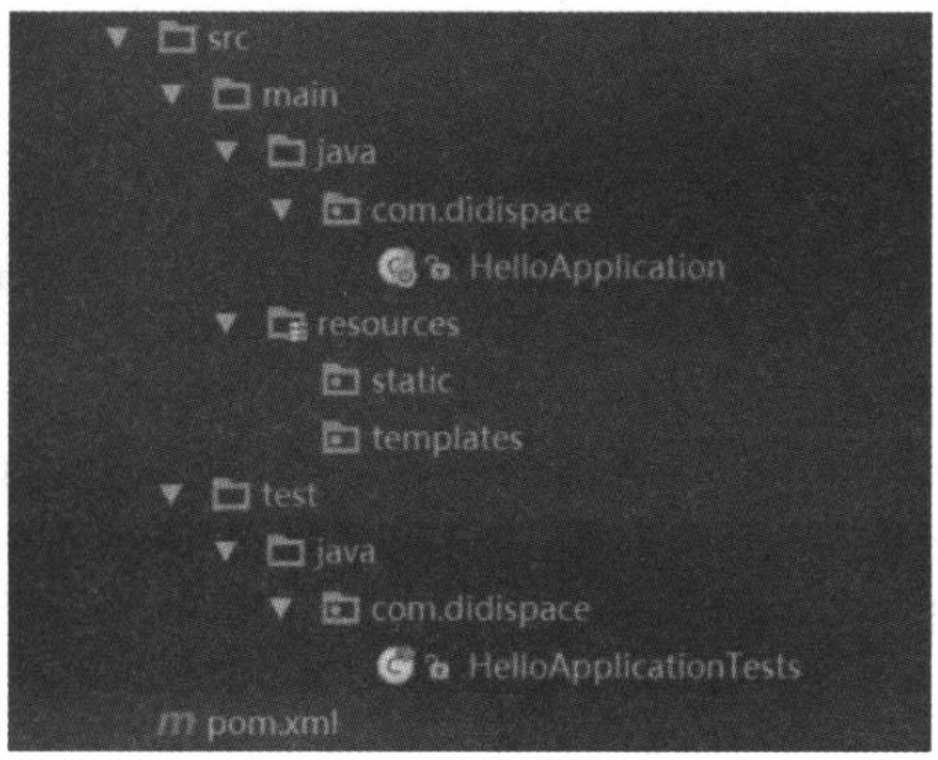
1. 通过官方的 Spring Initializr 工具来产生基础项目。
2. 访问 `http://start.spring.io/`，如下图所示，该页面提供了以 Maven 或 Gradle 构建 Spring Boot 项目的功能。
3. 选择构建工具 `Maven Project`、Spring Boot 版本选择 `1.3.7`，填写 Group 和 Artifact 信息，在 `Search for dependencies` 中可以搜索需要的其他依赖包，这里我们要实现 RESTful API，所以可以添加 `Web` 依赖。
4. 单击 `Generate Project` 按钮下载项目压缩包。
5. 解压项目包，并用 IDE 以 `Maven` 项目导入，以 `IntelliJ IDEA 14` 为例。
6. 从菜单中选择 `File-->New-->Project from Existing Sources...`。
7. 选择解压后的项目文件夹，单击 `OK` 按钮。




8. 单击 Import project from external model 并选择 Maven，一直单击 Next 按钮。
9. 若你的环境中有多个版本的 JDK，选择 Java SDK 的时候请选择 Java 7 以上的版本。

### 工程结构解析

在完成了上面的步骤之后，我们就创建了一个最基础的 Spring Boot 工程。




如上图所示，Spring Boot 的基础结构有三大块（具体路径根据用户生成项目时填写的 Group 和 Artifact 有所差异）。

- src/main/java：主程序入口 HelloApplication，可以通过直接运行该类来启动 Spring Boot 应用。

- src/main/resources：配置目录，该目录用来存放应用的一些配置信息，比如应用名、服务端口、数据库链接等。由于我们引入了 Web 模块，因此产生了 static 目录与 templates 目录，前者用于存放静态资源，如图片、CSS、JavaScript 等；后者用于存放 Web 页面的模板文件，这里我们主要演示提供 RESTful API，所以这两个目录并不会用到。
- src/test/：单元测试目录，生成的 HelloApplicationTests 通过 JUnit 4 实现，可以直接用运行 Spring Boot 应用的测试。后文中，我们会演示如何在该类中测试 RESTful API。

### Maven 配置分析

打开当前工程下的 pom.xml 文件，看看生成的项目都引入了哪些依赖来构建 Spring Boot 工程，内容大致如下所示。

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

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.didispace</groupId>
    <artifactId>spring-boot-hello</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>spring-boot-hello</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.7.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

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

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

```
        </dependency>

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

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

在基础信息部分，groupId 和 artifactId 对应生成项目时页面上输入的内容。另外，我们还可以注意到，打包形式为 jar：<packaging>jar</packaging>，正如我们之前所介绍的，Spring Boot 默认将该 Web 应用打包为 jar 的形式，而非 war 的形式，因为默认的 Web 模块依赖会包含嵌入式的 Tomcat，这样使得我们的应用 jar 自身就具备了提供 Web 服务的能力，后续我们会演示如何启动它。

父项目 parent 配置指定为 spring-boot-starter-parent 的 1.3.7 版本，该父项目中定义了 Spring Boot 版本的基础依赖以及一些默认配置内容，比如，配置文件 application.properties 的位置等。

在项目依赖 dependencies 配置中，包含了下面两项。

- spring-boot-starter-web：全栈 Web 开发模块，包含嵌入式 Tomcat、Spring MVC。
- spring-boot-starter-test：通用测试模块，包含 JUnit、Hamcrest、Mockito。

这里所引用的 web 和 test 模块，在 Spring Boot 生态中被称为 Starter POMs。Starter POMs 是一系列轻便的依赖包，是一套一站式的 Spring 相关技术的解决方案。开发者在使用和整合模块时，不必再去搜寻样例代码中的依赖配置来复制使用，只需要引入对应的模块包即可。比如，开发 Web 应用的时候，就引入 spring-boot-starter-web，希望应用具备访问数据库能力的时候，那就再引入 spring-boot-starter-jdbc 或是更好用的 spring-boot-starter-data-jpa。在使用 Spring Boot 构建应用的时候，各项功能模块的整合不再像传统 Spring 应用的开发方式那样，需要在 pom.xml 中做大量的依赖配置，

而是通过使用 Starter POMs 定义的依赖包，使得功能模块整合变得非常轻巧，易于理解与使用。

Spring Boot 的 Starter POMs 采用 `spring-boot-starter-*`的命名方式，*代表一个特别的应用功能模块，比如这里的 web、test。Spring Boot 工程本身的结构非常简单，大量的学习要点还是将来在对这些 Starter POMs 的使用之上。在本书中，由于主要讲述 Spring Cloud 的微服务组件内容，因此对各个 Spring Boot 的模块内容不做详尽讲解。对于初学者来说，我们也不必一次性地将所有模块的详细用法都掌握牢固，只需了解每个模块能做什么即可。对于本书所使用的 Spring Boot 1.3.7 版本的所有 Starter POMs 的功能索引可以参见附录 A，可以根据自身的需要来查询能够支持的模块，再去搜索这个模块的使用方法来实现我们的需求。

最后，项目构建的 build 部分，引入了 Spring Boot 的 Maven 插件，该插件非常实用，可以帮助我们方便地启停应用，这样在开发时就不用每次去找主类或是打包成 jar 来运行微服务，只需要通过 `mvn spring-boot:run` 命令就可以快速启动 Spring Boot 应用。

## 实现 RESTful API

在 Spring Boot 中创建一个 RESTful API 的实现代码同 Spring MVC 应用一样，只是不需要像 Spring MVC 那样先做很多配置，而是像下面这样直接开始编写 Controller 内容：

- 新建 `package`，命名为 `com.didispace.web`，可根据实际的构建情况修改成自己的路径。
- 新建 `HelloController` 类，内容如下所示。

```
@RestController
public class HelloController {

    @RequestMapping("/hello")
    public String index() {
        return "Hello World";
    }

}
```

- 启动该应用，通过浏览器访问 `http://localhost:8080/hello`，我们可以看到返回了预期结果：`Hello World`。

### 启动 Spring Boot 应用

启动 Spring Boot 应用的方式有很多种：

- 作为一个 Java 应用程序，可以直接通过运行拥有 main 函数的类来启动。
- 在 Maven 配置中，之前提到了 spring-boot 插件，可以使用它来启动，比如执行 `mvn spring-boot:run` 命令，或是直接单击 IDE 中对 Maven 插件的工具，例如 IntelliJ 中的支持：

- 在服务器上部署运行时，通常先使用 `mvn install` 将应用打包成 jar 包，再通过 `java -jar xxx.jar` 来启动应用。

### 编写单元测试

功能实现之后，我们要养成随手写配套单元测试的习惯，这在微服务架构中尤为重要。通常，我们实施微服务架构的时候，已经实现了前后端分离的项目与架构部署。那么在实现后端服务的时候，单元测试是在开发过程中用来验证代码正确性非常好的手段，并且这些单元测试将会很好地支持我们未来可能会进行的重构。

在 Spring Boot 中实现单元测试同样非常方便，下面我们打开 `src/test/` 下的测试入口 `com.didispace.HelloApplicationTests` 类，编写一个简单的单元测试来模拟 HTTP 请求，测试之前实现的 `/hello` 接口，该接口应返回 `Hello World` 字符串。

具体代码实现如下所示。

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = HelloApplication.class)
@WebAppConfiguration
public class HelloApplicationTests {

    private MockMvc mvc;

    @Before
    public void setUp() throws Exception {
        mvc = MockMvcBuilders.standaloneSetup(new HelloController()).build();
    }

    @Test
    public void hello() throws Exception {
        mvc.perform(MockMvcRequestBuilders.get("/hello").accept(MediaType.
APPLICATION_JSON))
```

```
                .andExpect(status().isOk())
                .andExpect(content().string(equalTo("Hello World")));
    }

}
```

代码解析如下。

- @RunWith(SpringJUnit4ClassRunner.class)：引入 Spring 对 JUnit4 的支持。
- @SpringApplicationConfiguration(classes = HelloApplication.class)：指定 Spring Boot 的启动类。
- @WebAppConfiguration：开启 Web 应用的配置，用于模拟 ServletContext。
- MockMvc 对象：用于模拟调用 Controller 的接口发起请求，在@Test 定义的 hello 测试用例中，perform 函数执行一次请求调用，accept 用于执行接收的数据类型，andExpect 用于判断接口返回的期望值。
- @Before：JUnit 中定义在测试用例@Test 内容执行前预加载的内容，这里用来初始化对 HelloController 的模拟。

注意引入下面的静态引用，让 status、content、equalTo 函数可用：

```
import static org.hamcrest.Matchers.equalTo;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
```

### 快速入门总结

在本节中，我们通过 Spring 官方的项目构建工具 Spring Initializr 生成了一个 Spring Boot 的基础项目，并详细介绍了该基础项目的依赖结构以及 Maven 的 spring-boot 插件，接着在该项目中实现一个输出“Hello World”的 RESTful 接口以及针对该接口的单元测试用例，完成了一个虽然简单，但涵盖了项目构建、服务开发、单元测试的全套开发内容。由于我们在后续使用 Spring Cloud 的时候会构建多个 Spring Boot 的项目来实现微服务架构中的基础设施以及微服务应用，所以通过本节的学习，我们已经具备了通过使用 Spring Boot 来构建简单微服务项目的基本能力。

在下一节中，我们将对后续在 Spring Cloud 组件使用过程中会涉及的配置内容做一些介绍和解释。

## 配置详解

在上一节中，我们轻松地实现了一个简单的 RESTful API 应用，体验了 Spring Boot 的诸多优点。我们用非常少的代码就成功实现了一个 Web 应用，这是传统 Spring 应用无法办到的。虽然在实现 Controller 时用到的代码是一样的，但是在配置方面，相信大家也注意到了，在上面的例子中，除了 Maven 的配置之外，没有引入任何其他配置。

这就是之前我们提到的，Spring Boot 针对常用的开发场景提供了一系列自动化配置来减少原本复杂而又几乎很少改动的模板化配置内容。但是，我们还是需要了解如何在 Spring Boot 中修改这些自动化的配置内容，以应对一些特殊的场景需求，比如，我们在同一台主机上需要启动多个基于 Spring Boot 的 Web 应用，若不为每个应用指定特别的端口号，那么默认的 8080 端口必将导致冲突。

后续我们在使用 Spring Cloud 的各个组件的时候，其实有大量的工作都会是针对配置文件的。所以我们有必要深入了解一些关于 Spring Boot 中的配置文件的知识，比如配置方式、如何实现多环境配置、配置信息的加载顺序等。

### 配置文件

在快速入门示例中，我们介绍 Spring Boot 的工程结构时，提到过 `src/main/resources` 目录是 Spring Boot 的配置目录，所以当要为应用创建个性化配置时，应在该目录下进行。

Spring Boot 的默认配置文件位置为 `src/main/resources/application.properties`。关于 Spring Boot 应用的配置内容都可以集中在该文件中，根据我们引入的不同 Starter 模块，可以在这里定义容器端口号、数据库连接信息、日志级别等各种配置信息。比如，我们需要自定义 Web 模块的服务端口号，可以在 `application.properties` 中添加 `server.port=8888` 来指定服务端口为 8888，也可以通过 `spring.application.name=hello` 来指定应用名（该名字在后续 Spring Cloud 中会被注册为服务名）。

Spring Boot 的配置文件除了可以使用传统的 properties 文件之外，还支持现在被广泛推荐使用的 YAML 文件。

> YAML（英语发音为/ˈjæməl/，尾音类似 camel 骆驼）是一个可读性高，用来表达资料序列的格式。YAML 参考了其他多种语言，包括 C 语言、Python、Perl，并从 XML、电子邮件的数据格式（RFC 2822）中获得灵感。Clark Evans 在 2001 年首次发表了这种语言，Ingy döt Net 与 Oren Ben-Kiki 也是这种语言的共同设计者。目前已经有数种编程语言或脚本语言支持（或者说解析）这种语言。YAML

> 是 YAML Ain't a Markup Language（YAML 不是一种标记语言）的缩写。在开发这种语言时，YAML 的意思其实是：Yet Another Markup Language（仍是一种标记语言），但为了强调这种语言以数据作为中心，而不是以标记语言为重点，而用反向缩略语重新命名。YAML 的语法和其他高阶语言类似，并且可以简单地表达清单、散列表、标量等形态。它使用空白符号缩排和大量依赖外观的特色，特别适合用来表达或编辑数据结构、各种设定文档、文件大纲（例如，许多电子邮件标题格式和 YAML 非常接近）。尽管它比较适合表达阶层式（hierarchical model）的数据结构，不过也有精致的语法可以表示关联性（relational model）的资料。由于 YAML 使用空白符号和分行来分隔资料，使得它特别适合用 grep / Python / Perl / Ruby 操作。其让人最容易上手的特色是巧妙避开各种封闭符号，如引号、各种括号等，这些符号在巢状结构时会变得复杂而难以辨认。
>
> ——维基百科

YAML 采用的配置格式不像 properties 的配置那样以单纯的键值对形式来表示，而是以类似大纲的缩进形式来表示。下面是一段 YAML 配置信息：

```
environments:
   dev:
      url: http://dev.bar.com
      name: Developer Setup
   prod:
      url: http://foo.bar.com
      name: My Cool App
```

与其等价的 properties 配置如下所示：

```
environments.dev.url=http://dev.bar.com
environments.dev.name=Developer Setup
environments.prod.url=http://foo.bar.com
environments.prod.name=My Cool App
```

通过 YAML 的配置方式我们可以看到，配置信息利用阶梯化缩进的方式，其结构更为清晰易读，同时配置内容的字符量也得到显著减少。除此之外，YAML 还可以在一个单个文件中通过使用 `spring.profiles` 属性来定义多个不同的环境配置。例如下面的内容，在指定为 test 环境时，`server.port` 将使用 8882 端口；而在 prod 环境中，`server.port` 将使用 8883 端口；如果没有指定环境，`server.port` 将使用 8881 端口。

```
server:
   port: 8881
---
spring:
   profiles: test
```

```
server:
  port: 8882
---
spring:
  profiles: prod
server:
  port: 8883
```

■**注意** YAML 目前还有一些不足，它无法通过@PropertySource 注解来加载配置。但是，YAML 将属性加载到内存中保存的时候是有序的，所以当配置文件中的信息需要具备顺序含义时，YAML 的配置方式比起 properties 配置文件更有优势。

## 自定义参数

除了可以在 Spring Boot 的配置文件中设置各个 Starter 模块中预定义的配置属性，也可以在配置文件中定义一些我们需要的自定义属性。比如在 application.properties 中添加：

```
book.name=SpringCloudInAction
book.author=ZhaiYongchao
```

然后，在应用中可以通过@Value 注解来加载这些自定义的参数，比如：

```
@Component
public class Book {

  @Value("${book.name}")
  private String name;
  @Value("${book.author}")
  private String author;

  // 省略getter和setter
}
```

@Value 注解加载属性值的时候可以支持两种表达式来进行配置，如下所示。

- 一种是上面介绍的 PlaceHolder 方式，格式为${...}，大括号内为 PlaceHolder。
- 另一种是使用 SpEL 表达式（Spring Expression Language），格式为#{...}，大括号内为 SpEL 表达式。

## 参数引用

在 application.properties 中的各个参数之间可以直接通过使用 PlaceHolder 的

方式来进行引用，就像下面的设置：

```
book.name=SpringCloud
book.author=ZhaiYongchao
book.desc=${book.author} is writing《${book.name}》
```

book.desc 参数引用了上文中定义的 book.name 和 book.author 属性，最后该属性的值就是 ZhaiYongchao is writing《SpringCloud》。

## 使用随机数

在一些特殊情况下，我们希望有些参数每次被加载的时候不是一个固定的值，比如密钥、服务端口等。在 Spring Boot 的属性配置文件中，可以通过使用${random}配置来产生随机的 int 值、long 值或者 string 字符串，这样我们就可以容易地通过配置随机生成属性，而不是在程序中通过编码来实现这些逻辑。

```
${random}的配置方式主要有以下几种，读者可作为参考使用。
# 随机字符串
com.didispace.blog.value=${random.value}
# 随机int
com.didispace.blog.number=${random.int}
# 随机long
com.didispace.blog.bignumber=${random.long}
# 10以内的随机数
com.didispace.blog.test1=${random.int(10)}
# 10~20的随机数
com.didispace.blog.test2=${random.int[10,20]}
```

该配置方式可以设置应用端口等场景，以避免在本地调试时出现端口冲突的麻烦。

## 命令行参数

回顾一下在本章的“快速入门”小节中，我们还介绍了如何启动 Spring Boot 应用，其中提到了使用命令 java -jar 来启动的方式。该命令除了启动应用之外，还可以在命令行中指定应用的参数，比如 java -jar xxx.jar --server.port=8888，直接以命令行的方式来设置 server.port 属性，并将启动应用的端口设为 8888。

在用命令行方式启动 Spring Boot 应用时，连续的两个减号 -- 就是对 application.properties 中的属性值进行赋值的标识。所以，java -jar xxx.jar--server.port=8888 命令，等价于在 application.properties 中添加属性 server.port=8888。

通过命令行来修改属性值是 Spring Boot 非常重要的一个特性。通过此特性，理论上已

经使得应用的属性在启动前是可变的，所以其中的端口号也好、数据库连接也好，都是可以在应用启动时发生改变的，而不同于以往的 Spring 应用通过 Maven 的 Profile 在编译器中进行不同环境的构建。Spring Boot 的这种方式，可以让应用程序的打包内容贯穿开发、测试以及线上部署，而 Maven 不同 Profile 的方案为每个环境所构建的包，其内容本质上是不同的。但是，如果每个参数都需要通过命令行来指定，这显然也不是一个好的方案，所以下面我们看看如何在 Spring Boot 中实现多环境的配置。

## 多环境配置

我们在开发应用的时候，通常同一套程序会被应用和安装到几个不同的环境中，比如开发、测试、生产等。其中每个环境的数据库地址、服务器端口等配置都不同，如果在为不同环境打包时都要频繁修改配置文件的话，那必将是个非常烦琐且容易发生错误的事。

对于多环境的配置，各种项目构建工具或是框架的基本思路是一致的，通过配置多份不同环境的配置文件，再通过打包命令指定需要打包的内容之后进行区分打包，Spring Boot 也不例外，或者说实现起来更加简单。

在 Spring Boot 中，多环境配置的文件名需要满足 `application-{profile}.properties` 的格式，其中`{profile}`对应你的环境标识，如下所示。

- `application-dev.properties`：开发环境。
- `application-test.properties`：测试环境。
- `application-prod.properties`：生产环境。

至于具体哪个配置文件会被加载，需要在 `application.properties` 文件中通过 `spring.profiles.active` 属性来设置，其值对应配置文件中的`{profile}`值。如 `spring.profiles.active=test` 就会加载 `application-test.properties` 配置文件内容。

下面，以不同环境配置不同的服务端口为例，进行样例实验。

- 针对各环境新建不同的配置文件 `application-dev.properties`、`application-test.properties`、`application-prod.properties`。
- 在这三个文件中均设置不同的 `server.port` 属性，例如，dev 环境设置为 1111，test 环境设置为 2222，prod 环境设置为 3333。
- `application.properties` 中设置 `spring.profiles.active=dev`，意为默认以 dev 环境设置。
- 测试不同配置的加载。

- 执行 `java -jar xxx.jar`，可以观察到服务端口被设置为 1111，也就是默认的开发环境（dev）。
- 执行 `java -jar xxx.jar --spring.profiles.active=test`，可以观察到服务端口被设置为 2222，也就是测试环境的配置（test）。
- 执行 `java -jar xxx.jar --spring.profiles.active=prod`，可以观察到服务端口被设置为 3333，也就是生产环境的配置（prod）。

按照上面的实验，可以如下总结多环境的配置思路：

- 在 `application.properties` 中配置通用内容，并设置 `spring.profiles.active=dev`，以开发环境为默认配置。
- 在 `application-{profile}.properties` 中配置各个环境不同的内容。
- 通过命令行方式去激活不同环境的配置。

### 加载顺序

在上面的例子中，我们将 Spring Boot 应用需要的配置内容都放在了项目工程中，已经能够通过 `spring.profiles.active` 或是通过 Maven 来实现多环境的支持。但是，当团队逐渐壮大，分工越来越细致之后，往往不需要让开发人员知道测试或是生产环境的细节，而是希望由每个环境各自的负责人（QA 或是运维）来集中维护这些信息。那么如果还是以这样的方式存储配置内容，对于不同环境配置的修改就不得不去获取工程内容来修改这些配置内容，当应用非常多的时候就变得非常不方便。同时，配置内容对开发人员都可见，这本身也是一种安全隐患。对此，出现了很多将配置内容外部化的框架和工具，后续将要介绍的 Spring Cloud Config 就是其中之一，为了后续能更好地理解 Spring Cloud Config 的加载机制，我们需要对 Spring Boot 对数据文件的加载机制有一定的了解。

为了能够更合理地重写各属性的值，Spring Boot 使用了下面这种较为特别的属性加载顺序：

1. 在命令行中传入的参数。
2. `SPRING_APPLICATION_JSON` 中的属性。`SPRING_APPLICATION_JSON` 是以 JSON 格式配置在系统环境变量中的内容。
3. `java:comp/env` 中的 JNDI 属性。
4. Java 的系统属性，可以通过 `System.getProperties()` 获得的内容。
5. 操作系统的环境变量。
6. 通过 `random.*` 配置的随机属性。

7. 位于当前应用 jar 包之外，针对不同{profile}环境的配置文件内容，例如 application-{profile}.properties 或是 YAML 定义的配置文件。
8. 位于当前应用 jar 包之内，针对不同{profile}环境的配置文件内容，例如 application-{profile}.properties 或是 YAML 定义的配置文件。
9. 位于当前应用 jar 包之外的 application.properties 和 YAML 配置内容。
10. 位于当前应用 jar 包之内的 application.properties 和 YAML 配置内容。
11. 在@Configuration 注解修改的类中，通过@PropertySource 注解定义的属性。
12. 应用默认属性，使用 SpringApplication.setDefaultProperties 定义的内容。

优先级按上面的顺序由高到低，数字越小优先级越高。

可以看到，其中第 7 项和第 9 项都是从应用 jar 包之外读取配置文件，所以，实现外部化配置的原理就是从此切入，为其指定外部配置文件的加载位置来取代 jar 包之内的配置内容。通过这样的实现，我们的工程在配置中就变得非常干净，只需在本地放置开发需要的配置即可，而不用关心其他环境的配置，由其对应环境的负责人去维护即可。

## 监控与管理

在微服务架构中，我们将原本庞大的单体系统拆分成多个提供不同服务的应用。虽然各个应用的内部逻辑因分解而得以简化，但是由于部署应用的数量成倍增长，使得系统的维护复杂度大大提升。对于运维人员来说，随着应用的不断增多，系统集群中出现故障的频率也变得越来越高，虽然在高可用机制的保护下，个别故障不会影响系统的对外服务，但是这些频繁出现的故障需要被及时发现和处理才能长期保证系统处于健康可用状态。为了能对这些成倍增长的应用做到高效运维，传统的运维方式显然是不合适的，所以我们需要实现一套自动化的监控运维机制，而这套机制的运行基础就是不间断地收集各个微服务应用的各项指标情况，并根据这些基础指标信息来制定监控和预警规则，更进一步甚至做到一些自动化的运维操作等。

为了让运维系统能够获取各个微服务应用的相关指标以及实现一些常规操作控制，我们需要开发一套专门用于植入各个微服务应用的接口供监控系统采集信息。而这些接口往往有很大一部分指标都是类似的，比如环境变量、垃圾收集信息、内存信息、线程池信息等。既然这些信息那么通用，难道就没有一个标准化的实现框架吗？

当我们决定用 Spring Boot 来作为微服务框架时，除了它强大的快速开发功能之外，还因为它在 Starter POMs 中提供了一个特殊依赖模块 spring-boot-starter-actuator。引

入该模块能够自动为 Spring Boot 构建的应用提供一系列用于监控的端点。同时，Spring Cloud 在实现各个微服务组件的时候，进一步为该模块做了不少扩展，比如，为原生端点增加了更多的指标和度量信息（比如在整合 Eureka 的时候会为/health 端点增加相关的信息），并且根据不同的组件还提供了更多有空的端点（比如，为 API 网关组件 Zuul 提供了/routes 端点来返回路由信息）。

spring-boot-starter-actuator 模块的实现对于实施微服务的中小团队来说，可以有效地省去或大大减少监控系统在采集应用指标时的开发量。当然，它也并不是万能的，有时候也需要对其做一些简单的扩展来帮助我们实现自身系统个性化的监控需求。所以，在本节将详细介绍一些关于 spring-boot-starter-actuator 模块的内容，包括原生提供的端点以及一些常用的扩展和配置方式等。

## 初识 actuator

下面，我们通过对“快速入门”小节中实现的 Spring Boot 应用增加 spring-boot-starter-actuator 模块功能，来对它有一个直观的认识。

在现有的 Spring Boot 应用中引入该模块非常简单，只需要在 pom.xml 的 dependency 节点中，新增 spring-boot-starter-actuator 的依赖即可，具体如下：

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

增加该依赖之后，重新启动应用。此时，我们可以在控制台中看到如下图所示的输出：

```
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/trace || /trace.json],methods=[GET],produces=[application
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/info || /info.json],methods=[GET],produces=[application/j
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/env/{name:.*}],methods=[GET],produces=[application/json]}
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/env || /env.json],methods=[GET],produces=[application/jso
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/dump || /dump.json],methods=[GET],produces=[application/j
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/autoconfig || /autoconfig.json],methods=[GET],produces=[a
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/health || /health.json],produces=[application/json]}" ont
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/mappings || /mappings.json],methods=[GET],produces=[appli
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/beans || /beans.json],methods=[GET],produces=[application
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/configprops || /configprops.json],methods=[GET],produces=
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/metrics/{name:.*}],methods=[GET],produces=[application/js
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/metrics || /metrics.json],methods=[GET],produces=[applica
```

上图显示了一批端点定义，这些端点并非我们自己在程序中创建的，而是由 spring-boot-starter-actuator 模块根据应用依赖和配置自动创建出来的监控和管理端点。通过这些端点，我们可以实时获取应用的各项监控指标，比如访问/health 端点。我们可以获得如下信息：

```
{
    "status": "UP",
    "diskSpace": {
```

```
        "status": "UP",
        "total": 491270434816,
        "free": 383870214144,
        "threshold": 10485760
    }
}
```

在没有引入其他依赖之前，该端点的内容较为简单，后续我们在使用 Spring Cloud 的各个组件之后，它的返回会变得非常丰富，这些内容将帮助我们制定更为个性化的监控策略。

## 原生端点

通过在快速入门示例中添加 `spring-boot-starter-actuator` 模块，我们已经对它有了一个初步的认识。接下来，我们详细介绍一下 `spring-boot-starter-actuator` 模块中已经实现的一些原生端点。根据端点的作用，可以将原生端点分为以下三大类。

- **应用配置类：**获取应用程序中加载的应用配置、环境变量、自动化配置报告等与 Spring Boot 应用密切相关的配置类信息。
- **度量指标类：**获取应用程序运行过程中用于监控的度量指标，比如内存信息、线程池信息、HTTP 请求统计等。
- **操作控制类：**提供了对应用的关闭等操作类功能。

下面我们来详细了解一下这三类端点都分别可以为我们提供怎样的有用信息和强大功能，以及我们如何去扩展和配置它们。

### 应用配置类

由于 Spring Boot 为了改善传统 Spring 应用繁杂的配置内容，采用了包扫描和自动化配置的机制来加载原本集中于 XML 文件中的各项内容。虽然这样的做法让我们的代码变得非常简洁，但是整个应用的实例创建和依赖关系等信息都被离散到了各个配置类的注解上，这使我们分析整个应用中资源和实例的各种关系变得非常困难。而这类端点可以帮助我们轻松获取一系列关于 Spring 应用配置内容的详细报告，比如自动化配置的报告、Bean 创建的报告、环境属性的报告等。

- /autoconfig：该端点用来获取应用的自动化配置报告，其中包括所有自动化配置的候选项。同时还列出了每个候选项是否满足自动化配置的各个先决条件。所以，该端点可以帮助我们方便地找到一些自动化配置为什么没有生效的具体原因。该报告内容将自动化配置内容分为以下两部分。
  - `positiveMatches` 中返回的是条件匹配成功的自动化配置。

- negativeMatches 中返回的是条件匹配不成功的自动化配置。

```
{
    "positiveMatches": { // 条件匹配成功的
        "EndpointWebMvcAutoConfiguration": [
            {
                "condition": "OnClassCondition",
                "message": "@ConditionalOnClass classes found:
javax.servlet.Servlet,org.springframework.web.servlet.DispatcherServlet"
            },
            {
                "condition": "OnWebApplicationCondition",
                "message": "found web application StandardServletEnvironment"
            }
        ],
        ...
    },
    "negativeMatches": {  // 条件匹配不成功的
        "HealthIndicatorAutoConfiguration.DataSourcesHealthIndicatorConfiguration": [
            {
                "condition": "OnClassCondition",
                "message": "required @ConditionalOnClass classes not found:
org.springframework.jdbc.core.JdbcTemplate"
            }
        ],
        ...
    }
}
```

从如上示例中我们可以看到，每个自动化配置候选项中都有一系列的条件，比如上面没有成功匹配的 HealthIndicatorAutoConfiguration.DataSourcesHealth-IndicatorConfiguration 配置，它的先决条件是需要在工程中包含 org.springframework.jdbc.core.JdbcTemplate 类，由于我们没有引入相关的依赖，它就不会执行自动化配置内容。所以，当我们发现有一些期望的配置没有生效时，就可以通过该端点来查看没有生效的具体原因。

- /beans：该端点用来获取应用上下文中创建的所有 Bean。

```
[
    {
        "context": "hello:dev:8881",
        "parent": null,
        "beans": [
            {
                "bean": "org.springframework.boot.autoconfigure.web.
```

```
DispatcherServletAutoConfiguration$DispatcherServletConfiguration",
                    "scope": "singleton",
                    "type": "org.springframework.boot.autoconfigure.web.
DispatcherServletAutoConfiguration$DispatcherServletConfiguration$$EnhancerBySpring
CGLIB$$3440282b",
                    "resource": "null",
                    "dependencies": [
                        "serverProperties",
                        "spring.mvc.CONFIGURATION_PROPERTIES",
                        "multipartConfigElement"
                    ]
                },
                {
                    "bean": "dispatcherServlet",
                    "scope": "singleton",
                    "type": "org.springframework.web.servlet.DispatcherServlet",
                    "resource": "class path resource
[org/springframework/boot/autoconfigure/web/DispatcherServletAutoConfiguration$Disp
atcherServletConfiguration.class]",
                    "dependencies": []
                }
            ]
        }
    ]
```

在如上示例中，我们可以看到在每个 Bean 中都包含了下面这些信息。

- bean：Bean 的名称。
- scope：Bean 的作用域。
- type：Bean 的 Java 类型。
- resource：class 文件的具体路径。
- dependencies：依赖的 Bean 名称。

- /configprops：该端点用来获取应用中配置的属性信息报告。从下面该端点返回示例的片段中，我们看到返回了关于该短信的配置信息，`prefix` 属性代表了属性的配置前缀，`properties` 代表了各个属性的名称和值。所以，我们可以通过该报告来看到各个属性的配置路径，比如我们要关闭该端点，就可以通过使用 `endpoints.configprops.enabled=false` 来完成设置。

```
{
    "configurationPropertiesReportEndpoint": {
        "prefix": "endpoints.configprops",
        "properties": {
            "id": "configprops",
            "sensitive": true,
```

```
            "enabled": true
        }
    },
    ...
}
```

- /env：该端点与/configprops 不同，它用来获取应用所有可用的环境属性报告。包括环境变量、JVM 属性、应用的配置属性、命令行中的参数。从下面该端点返回的示例片段中，可以看到它不仅返回了应用的配置属性，还返回了系统属性、环境变量等丰富的配置信息，其中还包括了应用还没有使用的配置，所以它可以帮助我们方便地看到当前应用可以加载的配置信息，并配合@ConfigurationProperties 注解将它们引入到我们的应用程序中来进行使用。另外，为了配置属性的安全，对于一些类似密码等敏感信息，该端点都会进行隐私保护，但是我们需要让属性名中包含 password、secret、key 这些关键词，这样该端点在返回它们的时候会使用*来替代实际的属性值。

```
{
    "profiles": [
        "dev"
    ],
    "server.ports": {
        "local.server.port": 8881
    },
    "servletContextInitParams": {

    },
    "systemProperties": {
        "idea.version": "2016.1.3",
        "java.runtime.name": "Java(TM) SE Runtime Environment",
        "sun.boot.library.path": "C:\\Program Files\\Java\\jdk1.8.0_91\\jre\\bin",
        "java.vm.version": "25.91-b15",
        "java.vm.vendor": "Oracle Corporation",
        ...
    },
    "systemEnvironment": {
        "configsetroot": "C:\\WINDOWS\\ConfigSetRoot",
        "RABBITMQ_BASE": "E:\\tools\\rabbitmq",
        ...
    },
    "applicationConfig: [classpath:/application-dev.properties]": {
        "server.port": "8881"
    },
    "applicationConfig: [classpath:/application.properties]": {
        "server.port": "8885",
```

```
            "spring.profiles.active": "dev",
            "info.app.name": "spring-boot-hello",
            "info.app.version": "v1.0.0",
            "spring.application.name": "hello"
        }
    }
```

- /mappings：该端点用来返回所有 Spring MVC 的控制器映射关系报告。从下面的示例片段中，我们可以看到该报告的信息与我们在启用 Spring MVC 的 Web 应用时输出的日志信息类似，其中 bean 属性标识了该映射关系的请求处理器，method 属性标识了该映射关系的具体处理类和处理函数。

```
{
    "/webjars/**": {
        "bean": "resourceHandlerMapping"
    },
    "/**": {
        "bean": "resourceHandlerMapping"
    },
    "/**/favicon.ico": {
        "bean": "faviconHandlerMapping"
    },
    "{[/hello]}": {
        "bean": "requestMappingHandlerMapping",
        "method": "public java.lang.String com.didispace.web.HelloController.index()"
    },
    "{[/mappings || /mappings.json],methods=[GET],produces=[application/json]}": {
        "bean": "endpointHandlerMapping",
        "method": "public java.lang.Object
org.springframework.boot.actuate.endpoint.mvc.EndpointMvcAdapter.invoke()"
    },
    ...
}
```

- /info：该端点用来返回一些应用自定义的信息。默认情况下，该端点只会返回一个空的 JSON 内容。我们可以在 application.properties 配置文件中通过 info 前缀来设置一些属性，比如下面这样：

```
info.app.name=spring-boot-hello
info.app.version=v1.0.0
```

再访问/info 端点，我们可以得到下面的返回报告，其中就包含了上面我们在应用中自定义的两个参数。

```
{
    "app": {
```

```
        "name": "spring-boot-hello",
        "version": "v1.0.0"
    }
}
```

### 度量指标类

上面我们所介绍的应用配置类端点所提供的信息报告在应用启动的时候就已经基本确定了其返回内容，可以说是一个静态报告。而度量指标类端点提供的报告内容则是动态变化的，这些端点提供了应用程序在运行过程中的一些快照信息，比如内存使用情况、HTTP 请求统计、外部资源指标等。这些端点对于我们构建微服务架构中的监控系统非常有帮助，由于 Spring Boot 应用自身实现了这些端点，所以我们可以很方便地利用它们来收集我们想要的信息，以制定出各种自动化策略。下面，我们就来分别看看这些强大的端点功能。

- /metrics：该端点用来返回当前应用的各类重要度量指标，比如内存信息、线程信息、垃圾回收信息等。

```
{
  "mem": 541305,
  "mem.free": 317864,
  "processors": 8,
  "instance.uptime": 33376471,
  "uptime": 33385352,
  "systemload.average": -1,
  "heap.committed": 476672,
  "heap.init": 262144,
  "heap.used": 158807,
  "heap": 3701248,
  "nonheap.committed": 65856,
  "nonheap.init": 2496,
  "nonheap.used": 64633,
  "nonheap": 0,
  "threads.peak": 22,
  "threads.daemon": 20,
  "threads.totalStarted": 26,
  "threads": 22,
  "classes": 7669,
  "classes.loaded": 7669,
  "classes.unloaded": 0,
  "gc.ps_scavenge.count": 7,
  "gc.ps_scavenge.time": 118,
  "gc.ps_marksweep.count": 2,
  "gc.ps_marksweep.time": 234,
  "httpsessions.max": -1,
  "httpsessions.active": 0,
```

```
    "gauge.response.beans": 55,
    "gauge.response.env": 10,
    "gauge.response.hello": 5,
    "gauge.response.metrics": 4,
    "gauge.response.configprops": 153,
    "gauge.response.star-star": 5,
    "counter.status.200.beans": 1,
    "counter.status.200.metrics": 3,
    "counter.status.200.configprops": 1,
    "counter.status.404.star-star": 2,
    "counter.status.200.hello": 11,
    "counter.status.200.env": 1
}
```

从上面的示例中，我们看到有如下这些重要的度量值。

- 系统信息：包括处理器数量 processors、运行时间 uptime 和 instance.uptime、系统平均负载 systemload.average。
- mem.*：内存概要信息，包括分配给应用的总内存数量以及当前空闲的内存数量。这些信息来自 java.lang.Runtime。
- heap.*：堆内存使用情况。这些信息来自 java.lang.management.MemoryMXBean 接口中 getHeapMemoryUsage 方法获取的 java.lang.management.MemoryUsage。
- nonheap.*：非堆内存使用情况。这些信息来自 java.lang.management.MemoryMXBean 接口中 getNonHeapMemoryUsage 方法获取的 java.lang.management.MemoryUsage。
- threads.*：线程使用情况，包括线程数、守护线程数（daemon）、线程峰值（peak）等，这些数据均来自 java.lang.management.ThreadMXBean。
- classes.*：应用加载和卸载的类统计。这些数据均来自 java.lang.management.ClassLoadingMXBean。
- gc.*：垃圾收集器的详细信息，包括垃圾回收次数 gc.ps_scavenge.count、垃圾回收消耗时间 gc.ps_scavenge.time、标记-清除算法的次数 gc.ps_marksweep.count、标记-清除算法的消耗时间 gc.ps_marksweep.time。这些数据均来自 java.lang.management.GarbageCollectorMXBean。
- httpsessions.*：Tomcat 容器的会话使用情况。包括最大会话数 httpsessions.max 和活跃会话数 httpsessions.active。该度量指标信息仅在引入嵌入式 Tomcat 作为应用容器的时候才会提供。

- gauge.*：HTTP 请求的性能指标之一，它主要用来反映一个绝对数值。比如上面示例中的 gauge.response.hello: 5，它表示上一次 hello 请求的延迟时间为 5 毫秒。
- counter.*：HTTP 请求的性能指标之一，它主要作为计数器来使用，记录了增加量和减少量。上述示例中的 counter.status.200.hello: 11，它代表了 hello 请求返回 200 状态的次数为 11。

  对于 gauge.*和 counter.*的统计，这里有一个特殊的内容请求 star-star，它代表了对静态资源的访问。这两类度量指标非常有用，我们不仅可以使用它默认的统计指标，还可以在程序中轻松地增加自定义统计值。只需要通过注入 org.springframework.boot.actuate.metrics.CounterService 和 org.springframework.boot.actuate.metrics.GaugeService 来实现自定义的统计指标信息。比如我们可以像下面这样自定义实现对 hello 接口的访问次数统计。

```
@RestController
public class HelloController {

    @Autowired
    private CounterService counterService;

    @RequestMapping("/hello")
    public String greet() {
        counterService.increment("didispace.hello.count");
        return "";
    }
}
```

  /metrics 端点可以提供应用运行状态的完整度量指标报告，这项功能非常实用，但是对于监控系统中的各项监控功能，它们的监控内容、数据收集频率都有所不同，如果每次都通过全量获取报告的方式来收集，略显粗暴。所以，我们还可以通过 /metrics/{name} 接口来更细粒度地获取度量信息，比如可以通过访问 /metrics/mem.free 来获取当前可用内存数量。
- /health：该端点在一开始的示例中我们已经使用过了，它用来获取应用的各类健康指标信息。在 spring-boot-starter-actuator 模块中自带实现了一些常用资源的健康指标检测器。这些检测器都通过 HealthIndicator 接口实现，并且会根据依赖关系的引入实现自动化装配，比如下面列出的这些。

| 检测器 | 功能 |
| --- | --- |
| DiskSpaceHealthIndicator | 低磁盘空间检测 |
| DataSourceHealthIndicator | 检测 DataSource 的连接是否可用 |
| MongoHealthIndicator | 检测 Mongo 数据库是否可用 |
| RabbitHealthIndicator | 检测 Rabbit 服务器是否可用 |
| RedisHealthIndicator | 检测 Redis 服务器是否可用 |
| SolrHealthIndicator | 检测 Solr 服务器是否可用 |

有时候，我们可能还会用到一些 Spring Boot 的 Starter POMs 中还没有封装的产品来进行开发，比如，当使用 RocketMQ 作为消息代理时，由于没有自动化配置的检测器，所以需要自己来实现一个用来采集健康信息的检测器。我们可以在 Spring Boot 的应用中，为 `org.springframework.boot.actuate.health.HealthIndicator` 接口实现一个对 RocketMQ 的检测器类：

```
@Component
public class RocketMQHealthIndicator implements HealthIndicator {
    @Override
    public Health health() {
        int errorCode = check();
        if (errorCode != 0) {
            return Health.down().withDetail("Error Code", errorCode).build();
        }
        return Health.up().build();
    }

    private int check() {
        // 对监控对象的检测操作
    }
}
```

通过重写 `health()` 函数可实现健康检查，在返回的 `Heath` 对象中，共有两项内容，一个是状态信息，除了该示例中的 `UP` 与 `DOWN` 之外，还有 `UNKNOWN` 和 `OUT_OF_SERVICE`，可以根据需要来实现返回；还有一个详细信息，采用 Map 的方式存储，在这里通过 `withDetail` 函数，注入了一个 Error Code 信息，我们也可以填入其他信息，比如，检测对象的 IP 地址、端口等。

重新启动应用，并访问 `/health` 接口，我们在返回的 JSON 字符串中，将会包含如下信息：

```
"rocketMQ": {
  "status": "UP"
}
```

- /dump：该端点用来暴露程序运行中的线程信息。它使用 `java.lang.management.ThreadMXBean` 的 `dumpAllThreads` 方法来返回所有含有同步信息的活动线程详情。
- /trace：该端点用来返回基本的 HTTP 跟踪信息。默认情况下，跟踪信息的存储采用 `org.springframework.boot.actuate.trace.InMemoryTraceRepository` 实现的内存方式，始终保留最近的 100 条请求记录。它记录的内容格式如下所示：

```
[
    {
        "timestamp": 1482570022463,
        "info": {
            "method": "GET",
            "path": "/metrics/mem",
            "headers": {
                "request": {
                    "host": "localhost:8881",
                    "connection": "keep-alive",
                    "cache-control": "no-cache",
                    "user-agent": "Mozilla/5.0 (Windows NT 10.0; WOW64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.143 Safari/537.36",
                    "postman-token": "9817ea4d-ad9d-b2fc-7685-9dff1a1bc193",
                    "accept": "*/*",
                    "accept-encoding": "gzip, deflate, sdch",
                    "accept-language": "zh-CN,zh;q=0.8"
                },
                "response": {
                    "X-Application-Context": "hello:dev:8881",
                    "Content-Type": "application/json;charset=UTF-8",
                    "Transfer-Encoding": "chunked",
                    "Date": "Sat, 24 Dec 2016 09:00:22 GMT",
                    "status": "200"
                }
            }
        }
    },
    ...
]
```

### 操作控制类

仔细的读者可能会发现，我们在“初识 actuator”小节中的示例的控制台中输出的所有监控端点，已经在介绍应用配置类端点和度量指标类端点时都讲解完了。那么还有哪些是操作控制类端点呢？实际上，由于之前介绍的所有端点都是用来反映应用自身的属性或是

运行中的状态，相对于操作控制类端点没有那么敏感，所以它们都是默认启用的。而操作控制类端点拥有更强大的控制能力，如果要使用它们的话，需要通过属性来配置开启操作。

在原生端点中，只提供了一个用来关闭应用的端点：/shutdown（在后续我们引入了Eureka 之后，会引入更多控制端点）。可以通过如下配置开启它：

```
endpoints.shutdown.enabled=true
```

在配置了上述属性之后，只需要访问该应用的/shutdown 端点就能实现关闭该应用的远程操作。由于开放关闭应用的操作本身是一件非常危险的事，所以真正在线上使用的时候，需要对其加入一定的保护机制，比如定制 actuator 的端点路径、整合 Spring Security 进行安全校验等。

# 小结

本章我们通过构建一个最基本的 Spring Boot 工程，让大家对其有了一个最直观的感受。同时，也为后续构建各类 Spring Cloud 组件和微服务应用做了一些基础准备工作。另外，我们对 Spring Boot 中的配置原理以及监控管理做了深入的介绍，因为这些内容将在后续的介绍中有所涉及，并且它们有助于理解 Spring Cloud 组件的运行原理。更多关于使用 Spring Boot 开发微服务应用的内容，我们不在本书中详细介绍，读者可参阅官方文档或其他书籍资料来学习。

# 第 3 章

# 服务治理：Spring Cloud Eureka

Spring Cloud Eureka 是 Spring Cloud Netflix 微服务套件中的一部分，它基于 Netflix Eureka 做了二次封装，主要负责完成微服务架构中的服务治理功能。Spring Cloud 通过为 Eureka 增加了 Spring Boot 风格的自动化配置，我们只需通过简单引入依赖和注解配置就能让 Spring Boot 构建的微服务应用轻松地与 Eureka 服务治理体系进行整合。

在本章中，我们将指引读者学习下面这些核心内容，并构建起用于服务治理的基础设施。

- 构建服务注册中心
- 服务注册与服务发现
- Eureka 的基础架构
- Eureka 的服务治理机制
- Eureka 的配置

## 服务治理

服务治理可以说是微服务架构中最为核心和基础的模块，它主要用来实现各个微服务实例的自动化注册与发现。为什么我们在微服务架构中那么需要服务治理模块呢？微服务系统没有它会有什么不好的地方吗？

在最初开始构建微服务系统的时候可能服务并不多，我们可以通过做一些静态配置来完成服务的调用。比如，有两个服务 A 和 B，其中服务 A 需要调用服务 B 来完成一个业务操作时，为了实现服务 B 的高可用，不论采用服务端负载均衡还是客户端负载均衡，都需要手工维护服务 B 的具体实例清单。但是随着业务的发展，系统功能越来越复杂，相应的微服务应用也不断增加，我们的静态配置就会变得越来越难以维护。并且面对不断发展的

业务，我们的集群规模、服务的位置、服务的命名等都有可能发生变化，如果还是通过手工维护的方式，那么极易发生错误或是命名冲突等问题。同时，对于这类静态内容的维护也必将消耗大量的人力。

为了解决微服务架构中的服务实例维护问题，产生了大量的服务治理框架和产品。这些框架和产品的实现都围绕着服务注册与服务发现机制来完成对微服务应用实例的自动化管理。

- **服务注册：**在服务治理框架中，通常都会构建一个注册中心，每个服务单元向注册中心登记自己提供的服务，将主机与端口号、版本号、通信协议等一些附加信息告知注册中心，注册中心按服务名分类组织服务清单。比如，我们有两个提供服务 A 的进程分别运行于 192.168.0.100:8000 和 192.168.0.101:8000 位置上，另外还有三个提供服务 B 的进程分别运行于 192.168.0.100:9000、192.168.0.101:9000、192.168.0.102:9000 位置上。当这些进程均启动，并向注册中心注册自己的服务之后，注册中心就会维护类似下面的一个服务清单。另外，服务注册中心还需要以心跳的方式去监测清单中的服务是否可用，若不可用需要从服务清单中剔除，达到排除故障服务的效果。

| 服务名 | 位置 |
| --- | --- |
| 服务 A | 192.168.0.100:8000、192.168.0.101:8000 |
| 服务 B | 192.168.0.100:9000、192.168.0.101:9000、192.168.0.102:9000 |

- **服务发现：**由于在服务治理框架下运作，服务间的调用不再通过指定具体的实例地址来实现，而是通过向服务名发起请求调用实现。所以，服务调用方在调用服务提供方接口的时候，并不知道具体的服务实例位置。因此，调用方需要向服务注册中心咨询服务，并获取所有服务的实例清单，以实现对具体服务实例的访问。比如，现有服务 C 希望调用服务 A，服务 C 就需要向注册中心发起咨询服务请求，服务注册中心就会将服务 A 的位置清单返回给服务 C，如按上例服务 A 的情况，C 便获得了服务 A 的两个可用位置 192.168.0.100:8000 和 192.168.0.101:8000。当服务 C 要发起调用的时候，便从该清单中以某种轮询策略取出一个位置来进行服务调用，这就是后续我们将会介绍的客户端负载均衡。这里我们只是列举了一种简单的服务治理逻辑，以方便理解服务治理框架的基本运行思路。实际的框架为了性能等因素，不会采用每次都向服务注册中心获取服务的方式，并且不同的应用场景在缓存和服务剔除等机制上也会有一些不同的实现策略。

## Netflix Eureka

Spring Cloud Eureka，使用 Netflix Eureka 来实现服务注册与发现，它既包含了服务端

组件，也包含了客户端组件，并且服务端与客户端均采用 Java 编写，所以 Eureka 主要适用于通过 Java 实现的分布式系统，或是与 JVM 兼容语言构建的系统。但是，由于 Eureka 服务端的服务治理机制提供了完备的 RESTful API，所以它也支持将非 Java 语言构建的微服务应用纳入 Eureka 的服务治理体系中来。只是在使用其他语言平台的时候，需要自己来实现 Eureka 的客户端程序。不过庆幸的是，在目前几个较为流行的开发平台上，都已经有了一些针对 Eureka 注册中心的客户端实现框架，比如.NET 平台的 Steeltoe、Node.js 的 eureka-js-client 等。

Eureka 服务端，我们也称为服务注册中心。它同其他服务注册中心一样，支持高可用配置。它依托于强一致性提供良好的服务实例可用性，可以应对多种不同的故障场景。如果 Eureka 以集群模式部署，当集群中有分片出现故障时，那么 Eureka 就转入自我保护模式。它允许在分片故障期间继续提供服务的发现和注册，当故障分片恢复运行时，集群中的其他分片会把它们的状态再次同步回来。以在 AWS 上的实践为例，Netflix 推荐每个可用的区域运行一个 Eureka 服务端，通过它来形成集群。不同可用区域的服务注册中心通过异步模式互相复制各自的状态，这意味着在任意给定的时间点每个实例关于所有服务的状态是有细微差别的。

Eureka 客户端，主要处理服务的注册与发现。客户端服务通过注解和参数配置的方式，嵌入在客户端应用程序的代码中，在应用程序运行时，Eureka 客户端向注册中心注册自身提供的服务并周期性地发送心跳来更新它的服务租约。同时，它也能从服务端查询当前注册的服务信息并把它们缓存到本地并周期性地刷新服务状态。

下面我们来构建一些简单示例，学习如何使用 Eureka 构建注册中心以及进行注册与发现服务。

## 搭建服务注册中心

首先，创建一个基础的 Spring Boot 工程，命名为 `eureka-server`，并在 `pom.xml` 中引入必要的依赖内容，代码如下：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/>
</parent>

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

```
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

通过@EnableEurekaServer 注解启动一个服务注册中心提供给其他应用进行对话。这一步非常简单，只需在一个普通的 Spring Boot 应用中添加这个注解就能开启此功能，比如下面的例子：

```
@EnableEurekaServer
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

在默认设置下，该服务注册中心也会将自己作为客户端来尝试注册它自己，所以我们需要禁用它的客户端注册行为，只需在 application.properties 中增加如下配置：

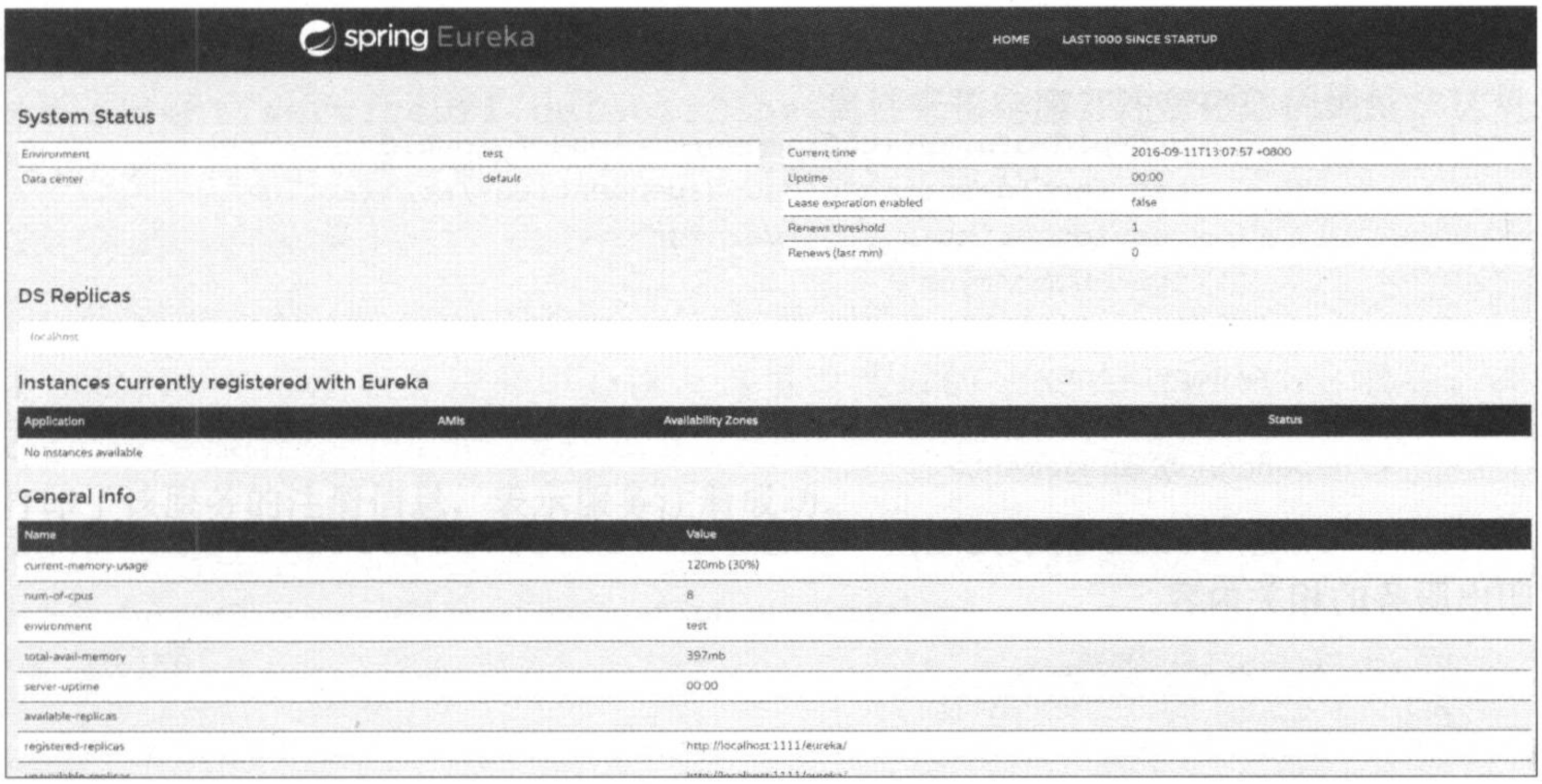
```
server.port=1111

eureka.instance.hostname=localhost
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${serve
r.port}/eureka/
```

由于后续内容也都会在本地运行，为了与后续要进行注册的服务区分，这里将服务注册中心的端口通过 server.port 属性设置为 1111。

- eureka.client.register-with-eureka：由于该应用为注册中心，所以设置为 false，代表不向注册中心注册自己。

- eureka.client.fetch-registry：由于注册中心的职责就是维护服务实例，它并不需要去检索服务，所以也设置为 false。

在完成了上面的配置后，启动应用并访问 http://localhost:1111/。可以看到如下图所示的 Eureka 信息面板，其中 Instances currently registered with Eureka 栏是空的，说明该注册中心还没有注册任何服务。




## 注册服务提供者

在完成了服务注册中心的搭建之后，接下来我们尝试将一个既有的 Spring Boot 应用加入 Eureka 的服务治理体系中去。

可以使用上一章中实现的快速入门工程来进行改造，将其作为一个微服务应用向服务注册中心发布自己。首先，修改 pom.xml，增加 Spring Cloud Eureka 模块的依赖，具体代码如下所示：

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

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

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

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Brixton.SR5</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

接着，改造/hello 请求处理接口，通过注入 DiscoveryClient 对象，在日志中打印出服务的相关内容。

```
@RestController
public class HelloController {

    private final Logger logger = Logger.getLogger(getClass());

    @Autowired
    private DiscoveryClient client;

    @RequestMapping(value = "/hello", method = RequestMethod.GET)
    public String index() {
        ServiceInstance instance = client.getLocalServiceInstance();
        logger.info("/hello, host:" + instance.getHost() + ", service_id:" +
instance.getServiceId());
        return "Hello World";
    }

}
```

然后，在主类中通过加上@EnableDiscoveryClient 注解，激活 Eureka 中的 DiscoveryClient 实现（自动化配置，创建 DiscoveryClient 接口针对 Eureka 客户端的 EurekaDiscoveryClient 实例），才能实现上述 Controller 中对服务信息的输出。

```
@EnableDiscoveryClient
@SpringBootApplication
public class HelloApplication {
```

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

}
```

最后，我们需要在 application.properties 配置文件中，通过 spring.application.name 属性来为服务命名，比如命名为 hello-service。再通过 eureka.client.serviceUrl.defaultZone 属性来指定服务注册中心的地址，这里我们指定为之前构建的服务注册中心地址，完整配置如下所示：

```
spring.application.name=hello-service

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

下面我们分别启动服务注册中心以及这里改造后的 hello-service 服务。在 hello-service 服务控制台中，Tomcat 启动之后，com.netflix.discovery.DiscoveryClient 对象打印了该服务的注册信息，表示服务注册成功。

```
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
c.n.e.EurekaDiscoveryClientConfiguration : Updating port to 8080
com.didispace.HelloApplication           : Started HelloApplication in 7.218
seconds (JVM running for 11.646)
com.netflix.discovery.DiscoveryClient    : DiscoveryClient_HELLO-SERVICE/PC-
201602152056:hello-service-registration status: 204
```

而此时在服务注册中心的控制台中，可以看到类似下面的输出，名为 hello-service 的服务被注册成功了。

```
c.n.e.registry.AbstractInstanceRegistry  : Registered instance
HELLO-SERVICE/PC-201602152056:hello-service with status UP (replication=true)
```

我们也可以通过访问 Eureka 的信息面板，在 Instances currently registered with Eureka 一栏中看到服务的注册信息。

Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
|---|---|---|---|
| HELLO-SERVICE | n/a (1) | (1) | UP (1) - PC-201602152056:hello-service |

通过访问 http://localhost:8080/hello，直接向该服务发起请求，在控制台中可以看到如下输出：

```
com.didispace.web.HelloController        : /hello, host:PC-201602152056,
service_id:hello-service
```

这些输出内容就是之前我们在 HelloController 中注入的 DiscoveryClient 接口对象，从服务注册中心获取的服务相关信息。

## 高可用注册中心

在微服务架构这样的分布式环境中，我们需要充分考虑发生故障的情况，所以在生产环境中必须对各个组件进行高可用部署，对于微服务如此，对于服务注册中心也一样。但是到本节为止，我们一直都在使用单节点的服务注册中心，这在生产环境中显然并不合适，我们需要构建高可用的服务注册中心以增强系统的可用性。

Eureka Server 的设计一开始就考虑了高可用问题，在 Eureka 的服务治理设计中，所有节点即是服务提供方，也是服务消费方，服务注册中心也不例外。是否还记得在单节点的配置中，我们设置过下面这两个参数，让服务注册中心不注册自己：

```
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Eureka Server 的高可用实际上就是将自己作为服务向其他服务注册中心注册自己，这样就可以形成一组互相注册的服务注册中心，以实现服务清单的互相同步，达到高可用的效果。下面我们就来尝试搭建高可用服务注册中心的集群。可以在本章第 1 节中实现的服务注册中心的基础之上进行扩展，构建一个双节点的服务注册中心集群。

- 创建 `application-peer1.properties`，作为 peer1 服务中心的配置，并将 serviceUrl 指向 peer2：

```
spring.application.name=eureka-server
server.port=1111

eureka.instance.hostname=peer1
eureka.client.serviceUrl.defaultZone=http://peer2:1112/eureka/
```

- 创建 `application-peer2.properties`，作为 peer2 服务中心的配置，并将 serviceUrl 指向 peer1：

```
spring.application.name=eureka-server
server.port=1112

eureka.instance.hostname=peer2
eureka.client.serviceUrl.defaultZone=http://peer1:1111/eureka/
```

- 在`/etc/hosts` 文件中添加对 peer1 和 peer2 的转换，让上面配置的 host 形式的 serviceUrl 能在本地正确访问到；Windows 系统路径为 `C:\Windows\System32\drivers\etc\hosts`。

```
127.0.0.1 peer1
127.0.0.1 peer2
```

- 通过 `spring.profiles.active` 属性来分别启动 peer1 和 peer2：

```
java -jar eureka-server-1.0.0.jar --spring.profiles.active=peer1
java -jar eureka-server-1.0.0.jar --spring.profiles.active=peer2
```

此时访问 peer1 的注册中心 http://localhost:1111/，如下图所示，我们可以看到，registered-replicas 中已经有 peer2 节点的 eureka-server 了。同样的，我们访问 peer2 的注册中心 http://localhost:1112/，也能看到 registered-replicas 中已经有 peer1 节点，并且这些节点在可用分片（available-replicase）之中。我们也可以尝试关闭 peer1，刷新 http://localhost:1112/，可以看到 peer1 的节点变为了不可用分片（unavailable-replicas）。

**DS Replicas**

peer2

**Instances currently registered with Eureka**

| Application | AMIs | Availability Zones | Status |
|---|---|---|---|
| EUREKA-SERVER | n/a (2) | (2) | UP (2) - Lenovo-zhaiyc:eureka-server:1111 , Lenovo-zhaiyc:eureka-server:1112 |

**General Info**

| Name | Value |
|---|---|
| total-avail-memory | 748mb |
| environment | test |
| num-of-cpus | 8 |
| current-memory-usage | 263mb (35%) |
| server-uptime | 00:00 |
| registered-replicas | http://peer2:1112/eureka/ |
| unavailable-replicas | |
| available-replicas | http://peer2:1112/eureka/, |

- 在设置了多节点的服务注册中心之后，服务提供方还需要做一些简单的配置才能将服务注册到 Eureka Server 集群中。我们以 hello-service 为例，修改 application.properties 配置文件，如下所示：

```
spring.application.name=hello-service

eureka.client.serviceUrl.defaultZone=http://peer1:1111/eureka/,http://peer2:111
2/eureka/
```

上面的配置主要对 eureka.client.serviceUrl.defaultZone 属性做了改动，将注册中心指向了之前我们搭建的 peer1 与 peer2。

下面，我们启动该服务，通过访问 http://localhost:1111/ 和 http://localhost:1112/，可以观察到 hello-service 服务同时被注册到了 peer1 和 peer2

上。若此时断开 peer1，由于 compute-service 同时也向 peer2 注册，因此在 peer2 上的其他服务依然能访问到 hello-service，从而实现了服务注册中心的高可用。

如我们不想使用主机名来定义注册中心的地址，也可以使用 IP 地址的形式，但是需要在配置文件中增加配置参数 eureka.instance.prefer-ip-address=true，该值默认为 false。

## 服务发现与消费

通过上面的内容介绍与实践，我们已经搭建起微服务架构中的核心组件——服务注册中心（包括单节点模式和高可用模式）。同时，还对上一章中实现的 Spring Boot 入门程序做了改造。通过简单的配置，使该程序注册到 Eureka 注册中心上，成为该服务治理体系下的一个服务，命名为 hello-service。现在我们已经有了服务注册中心和服务提供者，下面就来尝试构建一个服务消费者，它主要完成两个目标，发现服务以及消费服务。其中，服务发现的任务由 Eureka 的客户端完成，而服务消费的任务由 Ribbon 完成。Ribbon 是一个基于 HTTP 和 TCP 的客户端负载均衡器，它可以在通过客户端中配置的 ribbonServerList 服务端列表去轮询访问以达到均衡负载的作用。当 Ribbon 与 Eureka 联合使用时，Ribbon 的服务实例清单 RibbonServerList 会被 DiscoveryEnabledNIWSServerList 重写，扩展成从 Eureka 注册中心中获取服务端列表。同时它也会用 NIWSDiscoveryPing 来取代 IPing，它将职责委托给 Eureka 来确定服务端是否已经启动。在本章中，我们对 Ribbon 不做详细的介绍，读者只需要理解它在 Eureka 服务发现的基础上，实现了一套对服务实例的选择策略，从而实现对服务的消费。下一章我们会对 Ribbon 做详细的介绍和分析。

下面我们通过构建一个简单的示例，看看在 Eureka 的服务治理体系下如何实现服务的发现与消费。

- 首先，我们做一些准备工作。启动之前实现的服务注册中心 eureka-server 以及 hello-service 服务，为了实验 Ribbon 的客户端负载均衡功能，我们通过 java -jar 命令行的方式来启动两个不同端口的 hello-service，具体如下：

```
java -jar hello-service-0.0.1-SNAPSHOT.jar --server.port=8081
java -jar hello-service-0.0.1-SNAPSHOT.jar --server.port=8082
```

- 在成功启动两个 hello-service 服务之后，如下图所示，从 Eureka 信息面板中可以看到名为 HELLO-SERVICE 的服务中出现了两个实例单元，分别是通过命令行启动的 8081 端口和 8082 端口的服务。

Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
|---|---|---|---|
| HELLO-SERVICE | n/a (2) | (2) | UP (2) - PC-201602152056:hello-service:8081 , PC-201602152056:hello-service:8082 |

- 创建一个 Spring Boot 的基础工程来实现服务消费者，取名为 ribbon-consumer，并在 pom.xml 中引入如下的依赖内容。较之前的 hello-service，我们新增了 Ribbon 模块的依赖 spring-cloud-starter-ribbon。

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.3.7.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>

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

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

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

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Brixton.SR5</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

- 创建应用主类 ConsumerApplication，通过@EnableDiscoveryClient 注解让该应用注册为 Eureka 客户端应用，以获得服务发现的能力。同时，在该主类中创建 RestTemplate 的 Spring Bean 实例，并通过@LoadBalanced 注解开启客户端负载均衡。

```
@EnableDiscoveryClient
@SpringBootApplication
public class ConsumerApplication {

  @Bean
  @LoadBalanced
  RestTemplate restTemplate() {
    return new RestTemplate();
  }
  public static void main(String[] args) {
    SpringApplication.run(ConsumerApplication.class, args);
  }

}
```

- 创建 ConsumerController 类并实现/ribbon-consumer 接口。在该接口中，通过在上面创建的 RestTemplate 来实现对 HELLO-SERVICE 服务提供的/hello 接口进行调用。可以看到这里访问的地址是服务名 HELLO-SERVICE，而不是一个具体的地址，在服务治理框架中，这是一个非常重要的特性，也符合在本章一开始对服务治理的解释。

```
@RestController
public class ConsumerController {

    @Autowired
    RestTemplate restTemplate;

    @RequestMapping(value = "/ribbon-consumer", method = RequestMethod.GET)
    public String helloConsumer() {
        return restTemplate.getForEntity("http://HELLO-SERVICE/hello",
String.class).getBody();
    }

}
```

- 在 application.properties 中配置 Eureka 服务注册中心的位置，需要与之前的 HELLO-SERVICE 一样，不然是发现不了该服务的，同时设置该消费者的端口为 9000，不能与之前启动的应用端口冲突。

```
spring.application.name=ribbon-consumer
server.port=9000

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

- 启动 ribbon-consumer 应用后，我们可以在 Eureka 信息面板中看到，当前除了 HELLO-SERVICE 之外，还多了我们实现的 RIBBON-CONSUMER 服务。

Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
|---|---|---|---|
| HELLO-SERVICE | n/a (2) | (2) | UP (2) - PC-201602152056:hello-service:8081 , PC-201602152056:hello-service:8082 |
| RIBBON-CONSUMER | n/a (1) | (1) | UP (1) - PC-201602152056:ribbon-consumer:9000 |

- 通过向 http://localhost:9000/ribbon-consumer 发起 GET 请求，成功返回了“Hello World”。此时，我们可以在 ribbon-consumer 应用的控制台中看到如下信息，Ribbon 输出了当前客户端维护的 HELLO-SERVICE 的服务列表情况。其中包含了各个实例的位置，Ribbon 就是按照此信息进行轮询访问，以实现基于客户端的负载均衡。另外还输出了一些其他非常有用的信息，如对各个实例的请求总数量、第一次连接信息、上一次连接信息、总的请求失败数量等。

```
c.n.l.DynamicServerListLoadBalancer     : DynamicServerListLoadBalancer for client
HELLO-SERVICE initialized:
DynamicServerListLoadBalancer:{NFLoadBalancer:name=HELLO-SERVICE,current list of
Servers=[PC-201602152056:8082, PC-201602152056:8081],Load balancer stats=Zone stats:
{defaultzone=[Zone:defaultzone;  Instance count:2; Active connections count: 0;
Circuit breaker tripped count: 0; Active connections per server: 0.0;]
},Server stats: [[Server:PC-201602152056:8082;  Zone:defaultZone; Total Requests:0;
Successive connection failure:0;  Total blackout seconds:0; Last connection made:Thu Jan
01 08:00:00 CST 1970;  First connection made: Thu Jan 01 08:00:00 CST 1970;  Active
Connections:0; total failure count in last (1000) msecs:0; average resp time:0.0;  90
percentile resp time:0.0;  95 percentile resp time:0.0;  min resp time:0.0;  max resp
time:0.0;  stddev resp time:0.0]
, [Server:PC-201602152056:8081; Zone:defaultZone; Total Requests:0; Successive
connection failure:0;  Total blackout seconds:0; Last connection made:Thu Jan 01 08:00:00
CST 1970;  First connection made: Thu Jan 01 08:00:00 CST 1970;  Active Connections:0;
total failure count in last (1000) msecs:0; average resp time:0.0;  90 percentile resp
time:0.0;  95 percentile resp time:0.0;  min resp time:0.0;  max resp time:0.0;  stddev
resp time:0.0]
]}ServerList:org.springframework.cloud.netflix.ribbon.eureka.DomainExtractingServer
List@cc7240
```

再尝试发送几次请求，并观察启动的两个 HELLO-SERVICE 的控制台，可以看到两个控制台会交替打印下面的日志，这是我们之前在 HelloController 中实现的对服务信息的输出，可以用来判断当前 ribbon-consumer 对 HELLO-SERVICE 的调用是否是负载均衡的。

```
    com.didispace.web.HelloController       : /hello, host:PC-201602152056,
service_id:hello-service
```

## Eureka 详解

在上一节中，我们通过一个简单的服务注册与发现示例，构建了 Eureka 服务治理体系

中的三个核心角色：服务注册中心、服务提供者以及服务消费者。通过上述示例，相信读者对于 Eureka 的服务治理机制已经有了一些初步的认识。至此，我们已经学会了如何构建服务注册中心（包括单节点和高可用部署），也知道了如何使用 Eureka 的注解和配置将 Spring Boot 应用纳入 Eureka 的服务治理体系，成为服务提供者或是服务消费者。同时，对于客户端负载均衡的服务消费也有了一些简单的接触。但是，在实践中，我们的系统结构往往都要比上述示例复杂得多，如果仅仅依靠之前构建的服务治理内容，大多数情况是无法完全直接满足业务系统需求的，我们还需要根据实际情况来做一些配置、调整和扩展。所以，在本节中，我们将详细介绍 Eureka 的基础架构、节点间的通信机制以及一些进阶的配置等。

## 基础架构

在“服务治理”一节中，我们所讲解的示例虽然简单，但是麻雀虽小、五脏俱全。它已经包含了整个 Eureka 服务治理基础架构的三个核心要素。

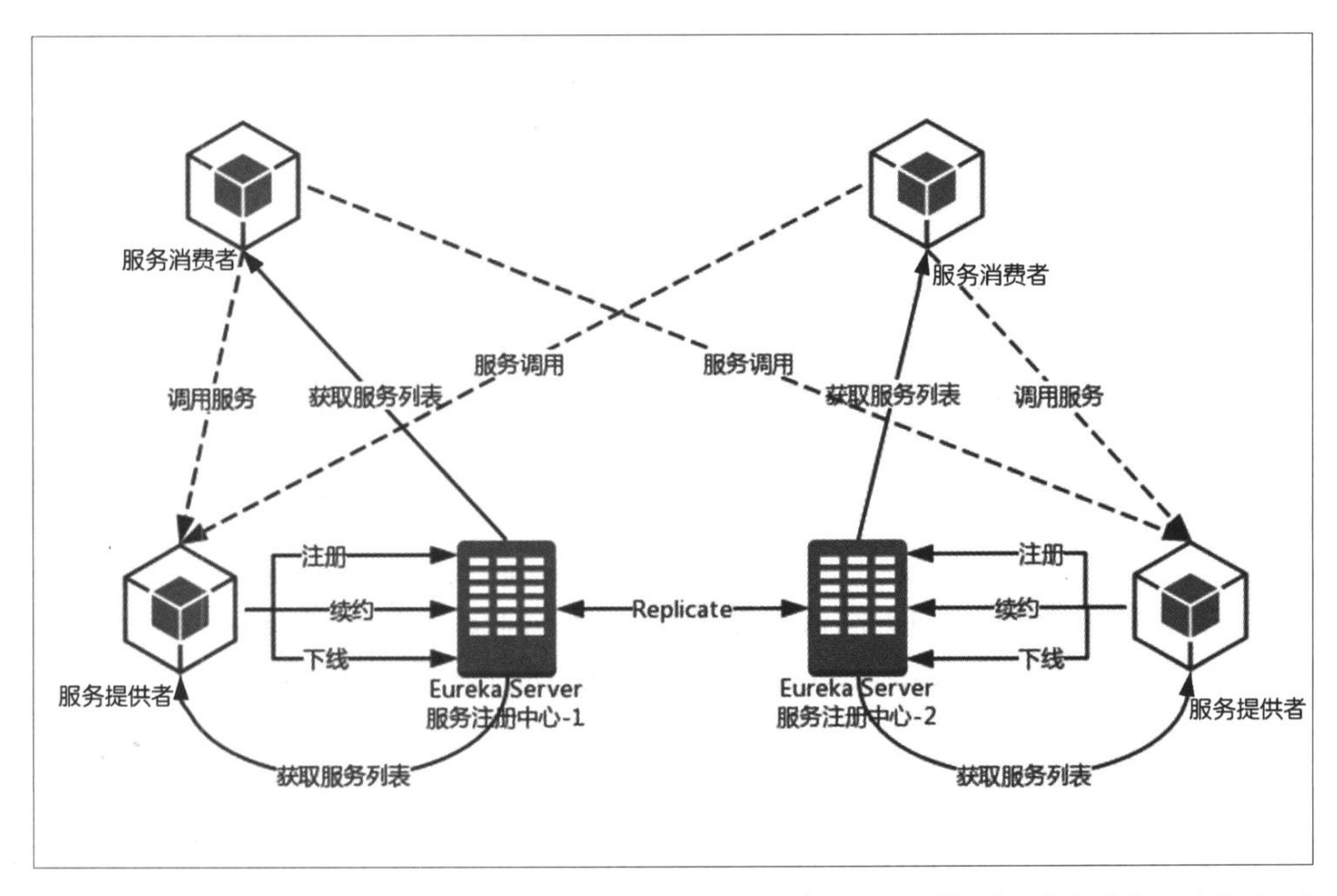
- **服务注册中心：**Eureka 提供的服务端，提供服务注册与发现的功能，也就是在上一节中我们实现的 `eureka-server`。
- **服务提供者：**提供服务的应用，可以是 Spring Boot 应用，也可以是其他技术平台且遵循 Eureka 通信机制的应用。它将自己提供的服务注册到 Eureka，以供其他应用发现，也就是在上一节中我们实现的 `HELLO-SERVICE` 应用。
- **服务消费者：**消费者应用从服务注册中心获取服务列表，从而使消费者可以知道去何处调用其所需要的服务，在上一节中使用了 Ribbon 来实现服务消费，另外后续还会介绍使用 Feign 的消费方式。

很多时候，客户端既是服务提供者也是服务消费者。

## 服务治理机制

在体验了 Spring Cloud Eureka 通过简单的注解配置就能实现强大的服务治理功能之后，我们来进一步了解一下 Eureka 基础架构中各个元素的一些通信行为，以此来理解基于 Eureka 实现的服务治理体系是如何运作起来的。以下图为例，其中有这样几个重要元素：

- “服务注册中心-1”和“服务注册中心-2”，它们互相注册组成了高可用集群。
- “服务提供者”启动了两个实例，一个注册到“服务注册中心-1”上，另外一个注册到“服务注册中心-2”上。
- 还有两个“服务消费者”，它们也都分别只指向了一个注册中心。




根据上面的结构，下面我们来详细了解一下，从服务注册开始到服务调用，及各个元素所涉及的一些重要通信行为。

**服务提供者**

服务注册

“服务提供者”在启动的时候会通过发送 REST 请求的方式将自己注册到 Eureka Server 上，同时带上了自身服务的一些元数据信息。Eureka Server 接收到这个 REST 请求之后，将元数据信息存储在一个双层结构 Map 中，其中第一层的 key 是服务名，第二层的 key 是具体服务的实例名。（我们可以回想一下之前在实现 Ribbon 负载均衡的例子中，Eureka 信息面板中一个服务有多个实例的情况，这些内容就是以这样的双层 Map 形式存储的。）

在服务注册时，需要确认一下 `eureka.client.register-with-eureka=true` 参数是否正确，该值默认为 `true`。若设置为 `false` 将不会启动注册操作。

服务同步

如架构图中所示，这里的两个服务提供者分别注册到了两个不同的服务注册中心上，也就是说，它们的信息分别被两个服务注册中心所维护。此时，由于服务注册中心之间因互相注册为服务，当服务提供者发送注册请求到一个服务注册中心时，它会将该请求转发

给集群中相连的其他注册中心，从而实现注册中心之间的服务同步。通过服务同步，两个服务提供者的服务信息就可以通过这两台服务注册中心中的任意一台获取到。

服务续约

在注册完服务之后，服务提供者会维护一个心跳用来持续告诉 Eureka Server：“我还活着”，以防止 Eureka Server 的“剔除任务”将该服务实例从服务列表中排除出去，我们称该操作为服务续约（Renew）。

关于服务续约有两个重要属性，我们可以关注并根据需要来进行调整：

```
eureka.instance.lease-renewal-interval-in-seconds=30
eureka.instance.lease-expiration-duration-in-seconds=90
```

`eureka.instance.lease-renewal-interval-in-seconds` 参数用于定义服务续约任务的调用间隔时间，默认为 30 秒。`eureka.instance.lease-expiration-duration-in-seconds` 参数用于定义服务失效的时间，默认为 90 秒。

**服务消费者**

获取服务

到这里，在服务注册中心已经注册了一个服务，并且该服务有两个实例。当我们启动服务消费者的时候，它会发送一个 REST 请求给服务注册中心，来获取上面注册的服务清单。为了性能考虑，Eureka Server 会维护一份只读的服务清单来返回给客户端，同时该缓存清单会每隔 30 秒更新一次。

获取服务是服务消费者的基础，所以必须确保 `eureka.client.fetch-registry=true` 参数没有被修改成 `false`，该值默认为 `true`。若希望修改缓存清单的更新时间，可以通过 `eureka.client.registry-fetch-interval-seconds=30` 参数进行修改，该参数默认值为 30，单位为秒。

服务调用

服务消费者在获取服务清单后，通过服务名可以获得具体提供服务的实例名和该实例的元数据信息。因为有这些服务实例的详细信息，所以客户端可以根据自己的需要决定具体调用哪个实例，在 Ribbon 中会默认采用轮询的方式进行调用，从而实现客户端的负载均衡。

对于访问实例的选择，Eureka 中有 Region 和 Zone 的概念，一个 Region 中可以包含多个 Zone，每个服务客户端需要被注册到一个 Zone 中，所以每个客户端对应一个 Region 和一个 Zone。在进行服务调用的时候，优先访问同处一个 Zone 中的服务提供方，若访问不到，就访问其他的 Zone，更多关于 Region 和 Zone 的知识，我们会在后续的源码解读中介绍。

服务下线

在系统运行过程中必然会面临关闭或重启服务的某个实例的情况，在服务关闭期间，我们自然不希望客户端会继续调用关闭了的实例。所以在客户端程序中，当服务实例进行正常的关闭操作时，它会触发一个服务下线的 REST 请求给 Eureka Server，告诉服务注册中心："我要下线了"。服务端在接收到请求之后，将该服务状态置为下线（DOWN），并把该下线事件传播出去。

**服务注册中心**

失效剔除

有些时候，我们的服务实例并不一定会正常下线，可能由于内存溢出、网络故障等原因使得服务不能正常工作，而服务注册中心并未收到"服务下线"的请求。为了从服务列表中将这些无法提供服务的实例剔除，Eureka Server 在启动的时候会创建一个定时任务，默认每隔一段时间（默认为 60 秒）将当前清单中超时（默认为 90 秒）没有续约的服务剔除出去。

自我保护

当我们在本地调试基于 Eureka 的程序时，基本上都会碰到这样一个问题，在服务注册中心的信息面板中出现类似下面的红色警告信息：

```
EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT.
RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST
TO BE SAFE.
```

实际上，该警告就是触发了 Eureka Server 的自我保护机制。之前我们介绍过，服务注册到Eureka Server之后，会维护一个心跳连接，告诉Eureka Server自己还活着。Eureka Server在运行期间，会统计心跳失败的比例在 15 分钟之内是否低于 85%，如果出现低于的情况（在单机调试的时候很容易满足，实际在生产环境上通常是由于网络不稳定导致），Eureka Server 会将当前的实例注册信息保护起来，让这些实例不会过期，尽可能保护这些注册信息。但是，在这段保护期间内实例若出现问题，那么客户端很容易拿到实际已经不存在的服务实例，会出现调用失败的情况，所以客户端必须要有容错机制，比如可以使用请求重试、断路器等机制。

由于本地调试很容易触发注册中心的保护机制，这会使得注册中心维护的服务实例不那么准确。所以，我们在本地进行开发的时候，可以使用 `eureka.server.enable-self-preservation=false` 参数来关闭保护机制，以确保注册中心可以将不可用的实例正确剔除。

## 源码分析

上面，我们对 Eureka 中各个核心元素的通信行为做了详细的介绍，相信大家已经对 Eureka 的运行机制有了一定的了解。为了更深入地理解它的运作和配置，下面我们结合源码来分别看看各个通信行为是如何实现的。

在看具体源码之前，我们先回顾一下之前所实现的内容，从而找到一个合适的切入口去分析。首先，对于服务注册中心、服务提供者、服务消费者这三个主要元素来说，后两者（也就是 Eureka 客户端）在整个运行机制中是大部分通信行为的主动发起者，而注册中心主要是处理请求的接收者。所以，我们可以从 Eureka 的客户端作为入口看看它是如何完成这些主动通信行为的。

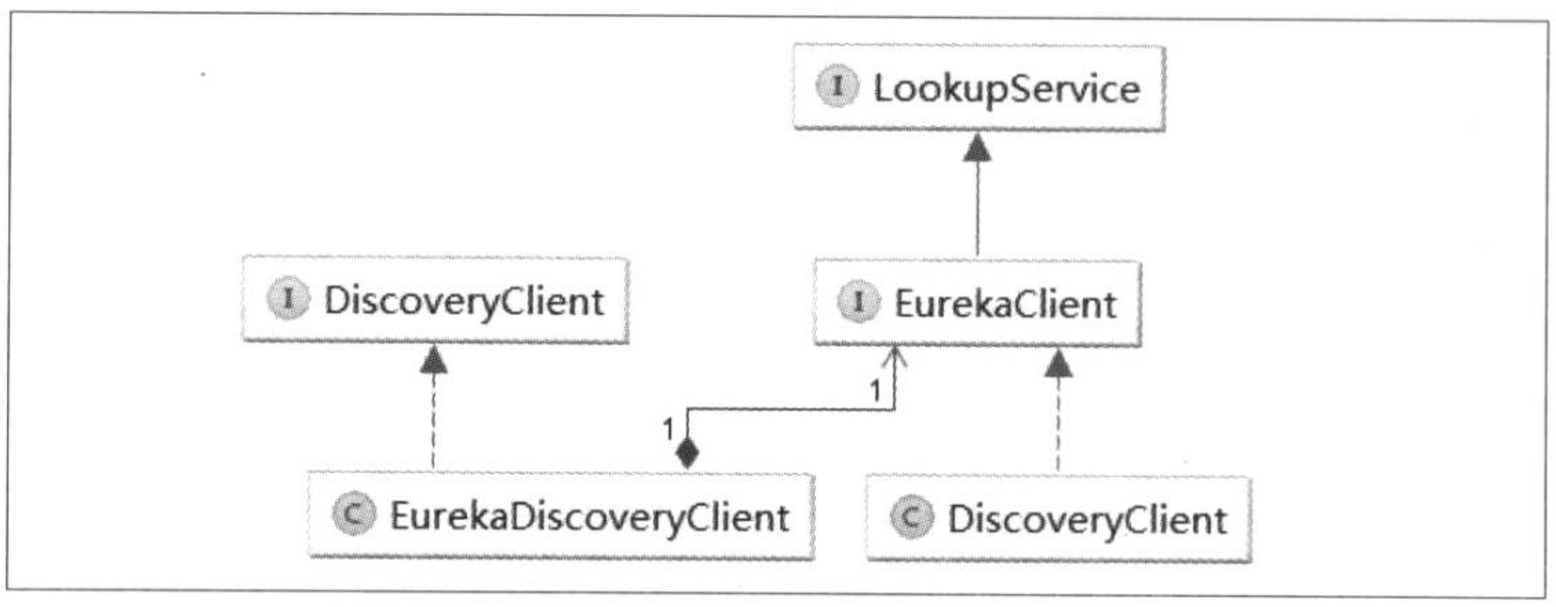
我们在将一个普通的 Spring Boot 应用注册到 Eureka Server 或是从 Eureka Server 中获取服务列表时，主要就做了两件事：

- 在应用主类中配置了`@EnableDiscoveryClient`注解。
- 在 `application.properties` 中用 `eureka.client.serviceUrl.defaultZone` 参数指定了服务注册中心的位置。

顺着上面的线索，我们来看看`@EnableDiscoveryClient`的源码，具体如下：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import(EnableDiscoveryClientImportSelector.class)
public @interface EnableDiscoveryClient {

}
```

从该注解的注释中我们可以知道，它主要用来开启 `DiscoveryClient` 的实例。通过搜索 `DiscoveryClient`，我们可以发现有一个类和一个接口。通过梳理可以得到如下图所示的关系：



其中，左边的 org.springframework.cloud.client.discovery.DiscoveryClient 是 Spring Cloud 的接口，它定义了用来发现服务的常用抽象方法，通过该接口可以有效地屏蔽服务治理的实现细节，所以使用 Spring Cloud 构建的微服务应用可以方便地切换不同服务治理框架，而不改动程序代码，只需要另外添加一些针对服务治理框架的配置即可。org.springframework.cloud.netflix.eureka.EurekaDiscoveryClient 是对该接口的实现，从命名来判断，它实现的是对 Eureka 发现服务的封装。所以 EurekaDiscoveryClient 依赖了 Netflix Eureka 的 com.netflix.discovery.EurekaClient 接口，EurekaClient 继承了 LookupService 接口，它们都是 Netflix 开源包中的内容，主要定义了针对 Eureka 的发现服务的抽象方法，而真正实现发现服务的则是 Netflix 包中的 com.netflix.discovery.DiscoveryClient 类。

接下来，我们就来详细看看 DiscoveryClient 类吧。先解读一下该类头部的注释，注释的大致内容如下所示：

```
这个类用于帮助与Eureka Server互相协作。

Eureka Client负责下面的任务:
- 向Eureka Server注册服务实例
- 向Eureka Server服务租约
- 当服务关闭期间，向Eureka Server取消租约
- 查询Eureka Server中的服务实例列表

Eureka Client还需要配置一个Eureka Server的URL列表。
```

在具体研究 Eureka Client 负责完成的任务之前，我们先看看在哪里对 Eureka Server 的 URL 列表进行配置。根据我们配置的属性名 eureka.client.serviceUrl.defaultZone，通过 serviceUrl 可以找到该属性相关的加载属性，但是在 SR5 版本中它们都被 @Deprecated 标注为不再建议使用，并@link 到了替代类 com.netflix.discovery.endpoint.EndpointUtils，所以我们可以在该类中找到下面这个函数：

```
    public static Map<String, List<String>> getServiceUrlsMapFromConfig(
            EurekaClientConfig clientConfig, String instanceZone, boolean
preferSameZone) {
        Map<String, List<String>> orderedUrls = new LinkedHashMap<>();
        String region = getRegion(clientConfig);
        String[] availZones = clientConfig.getAvailabilityZones(clientConfig.getRegion());
        if (availZones == null || availZones.length == 0) {
            availZones = new String[1];
            availZones[0] = DEFAULT_ZONE;
        }
        ……
```

```
    int myZoneOffset = getZoneOffset(instanceZone, preferSameZone, availZones);

    String zone = availZones[myZoneOffset];
    List<String> serviceUrls = clientConfig.getEurekaServerServiceUrls(zone);
    if (serviceUrls != null) {
        orderedUrls.put(zone, serviceUrls);
    }
    ……
    return orderedUrls;
}
```

### Region、Zone

在上面的函数中，可以发现，客户端依次加载了两个内容，第一个是 Region，第二个是 Zone，从其加载逻辑上我们可以判断它们之间的关系：

- 通过 `getRegion` 函数，我们可以看到它从配置中读取了一个 Region 返回，所以一个微服务应用只可以属于一个 Region，如果不特别配置，默认为 default。若我们要自己设置，可以通过 `eureka.client.region` 属性来定义。

```
public static String getRegion(EurekaClientConfig clientConfig) {
    String region = clientConfig.getRegion();
    if (region == null) {
        region = DEFAULT_REGION;
    }
    region = region.trim().toLowerCase();
    return region;
}
```

- 通过 `getAvailabilityZones` 函数，可以知道当我们没有特别为 Region 配置 Zone 的时候，将默认采用 defaultZone，这也是我们之前配置参数 `eureka.client.serviceUrl.defaultZone` 的由来。若要为应用指定 Zone，可以通过 `eureka.client.availability-zones` 属性来进行设置。从该函数的 `return` 内容，我们可以知道 Zone 能够设置多个，并且通过逗号分隔来配置。由此，我们可以判断 Region 与 Zone 是一对多的关系。

```
public String[] getAvailabilityZones(String region) {
    String value = this.availabilityZones.get(region);
    if (value == null) {
        value = DEFAULT_ZONE;
    }
    return value.split(",");
}
```

serviceUrls

在获取了 Region 和 Zone 的信息之后，才开始真正加载 Eureka Server 的具体地址。它根据传入的参数按一定算法确定加载位于哪一个 Zone 配置的 serviceUrls。

```
int myZoneOffset = getZoneOffset(instanceZone, preferSameZone, availZones);
String zone = availZones[myZoneOffset];
List<String> serviceUrls = clientConfig.getEurekaServerServiceUrls(zone);
```

具体获取 serviceUrls 的实现，我们可以详细查看 getEurekaServerServiceUrls 函数的具体实现类 EurekaClientConfigBean，该类是 EurekaClientConfig 和 EurekaConstants 接口的实现，用来加载配置文件中的内容，这里有非常多有用的信息，我们先说一下此处我们关心的，关于 defaultZone 的信息。通过搜索 defaultZone，我们可以很容易找到下面这个函数，它具体实现了如何解析该参数的过程，通过此内容，我们就可以知道，eureka.client.serviceUrl.defaultZone 属性可以配置多个，并且需要通过逗号分隔。

```
public List<String> getEurekaServerServiceUrls(String myZone) {
    String serviceUrls = this.serviceUrl.get(myZone);
    if (serviceUrls == null || serviceUrls.isEmpty()) {
        serviceUrls = this.serviceUrl.get(DEFAULT_ZONE);
    }
    if (!StringUtils.isEmpty(serviceUrls)) {
        final String[] serviceUrlsSplit =
StringUtils.commaDelimitedListToStringArray(serviceUrls);
        List<String> eurekaServiceUrls = new ArrayList<>(serviceUrlsSplit.length);
        for (String eurekaServiceUrl : serviceUrlsSplit) {
            if (!endsWithSlash(eurekaServiceUrl)) {
                eurekaServiceUrl += "/";
            }
            eurekaServiceUrls.add(eurekaServiceUrl);
        }
        return eurekaServiceUrls;
    }
    return new ArrayList<>();
}
```

当我们在微服务应用中使用 Ribbon 来实现服务调用时，对于 Zone 的设置可以在负载均衡时实现区域亲和特性：Ribbon 的默认策略会优先访问同客户端处于一个 Zone 中的服务端实例，只有当同一个 Zone 中没有可用服务端实例的时候才会访问其他 Zone 中的实例。所以通过 Zone 属性的定义，配合实际部署的物理结构，我们就可以有效地设计出对区域性故障的容错集群。

服务注册

在理解了多个服务注册中心信息的加载后，我们再回头看看 DiscoveryClient 类是如何实现“服务注册”行为的，通过查看它的构造类，可以找到它调用了下面这个函数：

```
private void initScheduledTasks() {
    ...
    if (clientConfig.shouldRegisterWithEureka()) {
        ...
        // InstanceInfo replicator
        instanceInfoReplicator = new InstanceInfoReplicator(
                this,
                instanceInfo,
                clientConfig.getInstanceInfoReplicationIntervalSeconds(),
                2); // burstSize
        ...

instanceInfoReplicator.start(clientConfig.getInitialInstanceInfoReplicationInterval
Seconds());
    } else {
        logger.info("Not registering with Eureka server per configuration");
    }
}
```

从上面的函数中，可以看到一个与服务注册相关的判断语句 if (clientConfig.shouldRegisterWithEureka())。在该分支内，创建了一个 InstanceInfoReplicator 类的实例，它会执行一个定时任务，而这个定时任务的具体工作可以查看该类的 run() 函数，具体如下所示：

```
public void run() {
    try {
        discoveryClient.refreshInstanceInfo();
        Long dirtyTimestamp = instanceInfo.isDirtyWithTime();
        if (dirtyTimestamp != null) {
            discoveryClient.register();
            instanceInfo.unsetIsDirty(dirtyTimestamp);
        }
    } catch (Throwable t) {
        logger.warn("There was a problem with the instance info replicator", t);
    } finally {
        Future next = scheduler.schedule(this, replicationIntervalSeconds,
TimeUnit.SECONDS);
        scheduledPeriodicRef.set(next);
    }
}
```

相信大家都发现了 discoveryClient.register();这一行，真正触发调用注册的地方就在这里。继续查看 register()的实现内容，如下所示：

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

通过属性命名，大家基本也能猜出来，注册操作也是通过 REST 请求的方式进行的。同时，我们能看到发起注册请求的时候，传入了一个 com.netflix.appinfo.InstanceInfo 对象，该对象就是注册时客户端给服务端的服务的元数据。

服务获取与服务续约

顺着上面的思路，我们继续来看 DiscoveryClient 的 initScheduledTasks 函数，不难发现在其中还有两个定时任务，分别是“服务获取”和“服务续约”：

```
private void initScheduledTasks() {
    if (clientConfig.shouldFetchRegistry()) {
        // registry cache refresh timer
        int registryFetchIntervalSeconds = clientConfig.getRegistryFetchInterval-
Seconds();
        int expBackOffBound = clientConfig.getCacheRefreshExecutorExponential-
BackOffBound();
        scheduler.schedule(
                new TimedSupervisorTask(
                        "cacheRefresh",
                        scheduler,
                        cacheRefreshExecutor,
                        registryFetchIntervalSeconds,
                        TimeUnit.SECONDS,
                        expBackOffBound,
                        new CacheRefreshThread()
                ),
                registryFetchIntervalSeconds, TimeUnit.SECONDS);
```

```
        }
        if (clientConfig.shouldRegisterWithEureka()) {
            int renewalIntervalInSecs = instanceInfo.getLeaseInfo().getRenewalIntervalInSecs();
            int expBackOffBound = clientConfig.getHeartbeatExecutorExponentialBackOffBound();
            logger.info("Starting heartbeat executor: " + "renew interval is: " +
renewalIntervalInSecs);

            // Heartbeat timer
            scheduler.schedule(
                    new TimedSupervisorTask(
                            "heartbeat",
                            scheduler,
                            heartbeatExecutor,
                            renewalIntervalInSecs,
                            TimeUnit.SECONDS,
                            expBackOffBound,
                            new HeartbeatThread()
                    ),
                    renewalIntervalInSecs, TimeUnit.SECONDS);
            // InstanceInfo replicator
            ……
        }
    }
```

从源码中我们可以发现，“服务获取”任务相对于“服务续约”和“服务注册”任务更为独立。“服务续约”与“服务注册”在同一个 if 逻辑中，这个不难理解，服务注册到 Eureka Server 后，自然需要一个心跳去续约，防止被剔除，所以它们肯定是成对出现的。从源码中，我们更清楚地看到了之前所提到的，对于服务续约相关的时间控制参数：

```
eureka.instance.lease-renewal-interval-in-seconds=30
eureka.instance.lease-expiration-duration-in-seconds=90
```

而“服务获取”的逻辑在独立的一个 if 判断中，其判断依据就是我们之前所提到的 `eureka.client.fetch-registry=true` 参数，它默认为 `true`，大部分情况下我们不需要关心。为了定期更新客户端的服务清单，以保证客户端能够访问确实健康的服务实例，“服务获取”的请求不会只限于服务启动，而是一个定时执行的任务，从源码中我们可以看到任务运行中的 `registryFetchIntervalSeconds` 参数对应的就是之前所提到的 `eureka.client.registry-fetch-interval-seconds=30` 配置参数，它默认为 30 秒。

继续向下深入，我们能分别发现实现“服务获取”和“服务续约”的具体方法，其中“服务续约”的实现较为简单，直接以 REST 请求的方式进行续约：

```
boolean renew() {
    EurekaHttpResponse<InstanceInfo> httpResponse;
```

```
        try {
            httpResponse = eurekaTransport.registrationClient.sendHeartBeat
(instanceInfo.getAppName(), instanceInfo.getId(), instanceInfo, null);
            logger.debug("{} - Heartbeat status: {}", PREFIX + appPathIdentifier,
httpResponse.getStatusCode());
            if (httpResponse.getStatusCode() == 404) {
                REREGISTER_COUNTER.increment();
                logger.info("{} - Re-registering apps/{}", PREFIX + appPathIdentifier,
instanceInfo.getAppName());
                return register();
            }
            return httpResponse.getStatusCode() == 200;
        } catch (Throwable e) {
            logger.error("{} - was unable to send heartbeat!", PREFIX + appPathIdentifier, e);
            return false;
        }
    }
```

而“服务获取”则复杂一些，会根据是否是第一次获取发起不同的 REST 请求和相应的处理。具体的实现逻辑跟之前类似，有兴趣的读者可以继续查看服务客户端的其他具体内容，以了解更多细节。

服务注册中心处理

通过上面的源码分析，可以看到所有的交互都是通过 REST 请求来发起的。下面我们来看看服务注册中心对这些请求的处理。Eureka Server 对于各类 REST 请求的定义都位于 `com.netflix.eureka.resources` 包下。

以“服务注册”请求为例：

```
    @POST
    @Consumes({"application/json", "application/xml"})
    public Response addInstance(InstanceInfo info,
                    @HeaderParam(PeerEurekaNode.HEADER_REPLICATION) String
isReplication) {
        logger.debug("Registering instance {} (replication={})", info.getId(),
isReplication);
        // validate that the instanceinfo contains all the necessary required fields
        ...
        // handle cases where clients may be registering with bad DataCenterInfo with
missing data
        DataCenterInfo dataCenterInfo = info.getDataCenterInfo();
        if (dataCenterInfo instanceof UniqueIdentifier) {
            String dataCenterInfoId = ((UniqueIdentifier) dataCenterInfo).getId();
            if (isBlank(dataCenterInfoId)) {
                boolean experimental = "true".equalsIgnoreCase(
                        serverConfig.getExperimental("registration.validation.
```

```
dataCenterInfoId"));
            if (experimental) {
                String entity = "DataCenterInfo of type " + dataCenterInfo.getClass()
                                + " must contain a valid id";
                return Response.status(400).entity(entity).build();
            } else if (dataCenterInfo instanceof AmazonInfo) {
                AmazonInfo amazonInfo = (AmazonInfo) dataCenterInfo;
                String effectiveId = amazonInfo.get(AmazonInfo.MetaDataKey.instanceId);
                if (effectiveId == null) {
                    amazonInfo.getMetadata().put(
                            AmazonInfo.MetaDataKey.instanceId.getName(), info.getId());
                }
            } else {
                logger.warn("Registering DataCenterInfo of type {} without an
appropriate id",
                        dataCenterInfo.getClass());
            }
        }
    }

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

在对注册信息进行了一堆校验之后，会调用 org.springframework.cloud.netflix.eureka.server.InstanceRegistry 对象中的 register(InstanceInfo info, int leaseDuration, boolean isReplication)函数来进行服务注册：

```
public void register(InstanceInfo info, int leaseDuration, boolean isReplication) {
    if (log.isDebugEnabled()) {
        log.debug("register " + info.getAppName() + ", vip " + info.getVIPAddress()
                + ", leaseDuration " + leaseDuration + ", isReplication "
                + isReplication);
    }
    this.ctxt.publishEvent(new EurekaInstanceRegisteredEvent(this, info,
            leaseDuration, isReplication));

    super.register(info, leaseDuration, isReplication);
}
```

在注册函数中，先调用 publishEvent 函数，将该新服务注册的事件传播出去，然后调用 com.netflix.eureka.registry.AbstractInstanceRegistry 父类中的注册实现，将 InstanceInfo 中的元数据信息存储在一个 ConcurrentHashMap 对象中。正如我们之前所说的，注册中心存储了两层 Map 结构，第一层的 key 存储服务名：InstanceInfo 中的 appName 属性，第二层的 key 存储实例名：InstanceInfo 中的

instanceId 属性。

服务端的请求和接收非常类似，对于其他的服务端处理，这里不再展开叙述，读者可以根据上面的脉络来自己查看其内容（这里包含很多细节内容）来帮助和加深理解。

## 配置详解

在分析了 Eureka 的部分源码之后，相信大家对 Eureka 的服务治理机制已经有了进一步的理解。在本节中，我们从使用的角度对 Eureka 中一些常用配置内容进行详细的介绍，以帮助我们根据自身环境与业务特点来进行个性化的配置调整。

在 Eureka 的服务治理体系中，主要分为服务端与客户端两个不同的角色，服务端为服务注册中心，而客户端为各个提供接口的微服务应用。当我们构建了高可用的注册中心之后，该集群中所有的微服务应用和后续将要介绍的一些基础类应用（如配置中心、API 网关等）都可以视作该体系下的一个微服务（Eureka 客户端）。服务注册中心也一样，只是高可用环境下的服务注册中心除了作为客户端之外，还为集群中的其他客户端提供了服务注册的特殊功能。所以，Eureka 客户端的配置对象存在于所有 Eureka 服务治理体系下的应用实例中。在实际使用 Spring Cloud Eureka 的过程中，我们所做的配置内容几乎都是对 Eureka 客户端配置进行的操作，所以了解这部分的配置内容，对于用好 Eureka 非常有帮助。

Eureka 客户端的配置主要分为以下两个方面。

- 服务注册相关的配置信息，包括服务注册中心的地址、服务获取的间隔时间、可用区域等。
- 服务实例相关的配置信息，包括服务实例的名称、IP 地址、端口号、健康检查路径等。

而 Eureka 服务端更多地类似于一个现成产品，大多数情况下，我们不需要修改它的配置信息。所以在本书中，我们对此不进行过多的介绍，有兴趣的读者可以查看 org.springframework.cloud.netflix.eureka.server.EurekaServerConfigBean 类的定义来做进一步的学习，这些参数均以 eureka.server 作为前缀。另外值得一提的是，我们在学习本书内容进行本地调试的时候，可以通过设置该类中的 enableSelfPreservation 参数来关闭注册中心的“自我保护”功能，以防止关闭的实例无法被服务注册中心剔除的问题，这一点我们在“服务治理机制”一节中也有所介绍。

### 服务注册类配置

关于服务注册类的配置信息，我们可以通过查看 org.springframework.cloud.netflix.eureka.EurekaClientConfigBean 的源码来获得比官方文档中更为详尽

的内容，这些配置信息都以 eureka.client 为前缀。下面我们针对一些常用的配置信息做进一步的介绍和说明。

**指定注册中心**

在本章第 1 节的示例中，我们演示了如何将一个 Spring Boot 应用纳入 Eureka 的服务治理体系，除了引入 Eureka 的依赖之外，就是在配置文件中指定注册中心，主要通过 eureka.client.serviceUrl 参数实现。该参数的定义如下所示，它的配置值存储在 HashMap 类型中，并且设置有一组默认值，默认值的 key 为 defaultZone、value 为 http://localhost:8761/eureka/。

```
    private Map<String, String> serviceUrl = new HashMap<>();
    {
        this.serviceUrl.put(DEFAULT_ZONE, DEFAULT_URL);
    }

    public static final String DEFAULT_URL = "http://localhost:8761" + DEFAULT_PREFIX
+ "/";
    public static final String DEFAULT_ZONE = "defaultZone";
```

由于之前实现的服务注册中心使用了 1111 端口，所以我们做了如下配置，来将应用注册到对应的 Eureka 服务端中。

```
    eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

当构建了高可用的服务注册中心集群时，我们可以为参数的 value 值配置多个注册中心的地址（通过逗号分隔）。比如下面的例子：

```
    eureka.client.serviceUrl.defaultZone=http://peer1:1111/eureka/,http://peer2:111
2/eureka/
```

另外，为了服务注册中心的安全考虑，很多时候我们都会为服务注册中心加入安全校验。这个时候，在配置 serviceUrl 时，需要在 value 值的 URL 中加入相应的安全校验信息，比如 http://<username>:<password>@localhost:1111/eureka。其中，<username>为安全校验信息的用户名，<password>为该用户的密码。

**其他配置**

下面整理了 org.springframework.cloud.netflix.eureka.EurekaClientConfigBean 中定义的常用配置参数以及对应的说明和默认值，这些参数均以 eureka.client 为前缀。

| 参数名 | 说明 | 默认值 |
| --- | --- | --- |
| enabled | 启用 Eureka 客户端 | true |

续表

| 参数名 | 说明 | 默认值 |
| --- | --- | --- |
| registryFetchIntervalSeconds | 从 Eureka 服务端获取注册信息的间隔时间，单位为秒 | 30 |
| instanceInfoReplicationIntervalSeconds | 更新实例信息的变化到 Eureka 服务端的间隔时间，单位为秒 | 30 |
| initialInstanceInfoReplicationIntervalSeconds | 初始化实例信息到 Eureka 服务端的间隔时间，单位为秒 | 40 |
| eurekaServiceUrlPollIntervalSeconds | 轮询 Eureka 服务端地址更改的间隔时间，单位为秒。当我们与 Spring Cloud Config 配合，动态刷新 Eureka 的 serviceURL 地址时需要关注该参数 | 300 |
| eurekaServerReadTimeoutSeconds | 读取 Eureka Server 信息的超时时间，单位为秒 | 8 |
| eurekaServerConnectTimeoutSeconds | 连接 Eureka Server 的超时时间，单位为秒 | 5 |
| eurekaServerTotalConnections | 从 Eureka 客户端到所有 Eureka 服务端的连接总数 | 200 |
| eurekaServerTotalConnectionsPerHost | 从 Eureka 客户端到每个 Eureka 服务端主机的连接总数 | 50 |
| eurekaConnectionIdleTimeoutSeconds | Eureka 服务端连接的空闲关闭时间，单位为秒 | 30 |
| heartbeatExecutorThreadPoolSize | 心跳连接池的初始化线程数 | 2 |
| heartbeatExecutorExponentialBackOffBound | 心跳超时重试延迟时间的最大乘数值 | 10 |
| cacheRefreshExecutorThreadPoolSize | 缓存刷新线程池的初始化线程数 | 2 |
| cacheRefreshExecutorExponentialBackOffBound | 缓存刷新重试延迟时间的最大乘数值 | 10 |
| useDnsForFetchingServiceUrls | 使用 DNS 来获取 Eureka 服务端的 serviceUrl | false |
| registerWithEureka | 是否要将自身的实例信息注册到Eureka服务端 | true |
| preferSameZoneEureka | 是否偏好使用处于相同 Zone 的 Eureka 服务端 | true |
| filterOnlyUpInstances | 获取实例时是否过滤，仅保留 UP 状态的实例 | true |
| fetchRegistry | 是否从 Eureka 服务端获取注册信息 | true |

### 服务实例类配置

关于服务实例类的配置信息，我们可以通过查看 `org.springframework.cloud.netflix.eureka.EurekaInstanceConfigBean` 的源码来获取详细内容，这些配置信息都以 `eureka.instance` 为前缀，下面我们针对一些常用的配置信息做一些详细的说明。

### 元数据

在org.springframework.cloud.netflix.eureka.EurekaInstanceConfigBean的配置信息中，有一大部分内容都是对服务实例元数据的配置，那么什么是服务实例的元数据呢？它是 Eureka 客户端在向服务注册中心发送注册请求时，用来描述自身服务信息的对象，其中包含了一些标准化的元数据，比如服务名称、实例名称、实例 IP、实例端口等用于服务治理的重要信息；以及一些用于负载均衡策略或是其他特殊用途的自定义元数据信息。

在使用 Spring Cloud Eureka 的时候，所有的配置信息都通过 org.springframework.cloud.netflix.eureka.EurekaInstanceConfigBean 进行加载，但在真正进行服务注册的时候，还是会包装成 com.netflix.appinfo.InstanceInfo 对象发送给 Eureka 服务端。这两个类的定义非常相似，我们可以直接查看 com.netflix.appinfo.InstanceInfo 类中的详细定义来了解原生 Eureka 对元数据的定义。其中，Map<String, String> metadata = new ConcurrentHashMap<String, String>() 是自定义的元数据信息，而其他成员变量则是标准化的元数据信息。Spring Cloud 的 EurekaInstanceConfigBean 对原生元数据对象做了一些配置优化处理，在后续的介绍中，我们也会提到这些内容。

我们可以通过 eureka.instance.<properties>=<value>的格式对标准化元数据直接进行配置，其中<properties>就是 EurekaInstanceConfigBean 对象中的成员变量名。而对于自定义元数据，可以通过 eureka.instance.metadataMap.<key>=<value>的格式来进行配置，比如：

```
eureka.instance.metadataMap.zone=shanghai
```

接下来，我们将针对一些常用的元数据配置做进一步的介绍和说明。

### 实例名配置

实例名，即 InstanceInfo 中的 instanceId 参数，它是区分同一服务中不同实例的唯一标识。在 Netflix Eureka 的原生实现中，实例名采用主机名作为默认值，这样的设置使得在同一主机上无法启动多个相同的服务实例。所以，在 Spring Cloud Eureka 的配置中，针对同一主机中启动多实例的情况，对实例名的默认命名做了更为合理的扩展，它采用了如下默认规则：

```
${spring.cloud.client.hostname}:${spring.application.name}:${spring.application
.instance_id:${server.port}}
```

对于实例名的命名规则，我们可以通过 eureka.instance.instanceId 参数来进行配置。比如，在本地进行客户端负载均衡调试时，需要启动同一服务的多个实例，如果我们直接启动同一个应用必然会产生端口冲突。虽然可以在命令行中指定不同的

server.port 来启动，但是这样还是略显麻烦。实际上，我们可以直接通过设置 server.port=0 或者使用随机数 server.port=${random.int[10000,19999]}来让 Tomcat 启动的时候采用随机端口。但是这个时候我们会发现注册到 Eureka Server 的实例名都是相同的，这会使得只有一个服务实例能够正常提供服务。对于这个问题，我们就可以通过设置实例名规则来轻松解决：

```
eureka.instance.instanceId=${spring.application.name}:${random.int}}
```

通过上面的配置，利用应用名加随机数的方式来区分不同的实例，从而实现在同一主机上，不指定端口就能轻松启动多个实例的效果。

**端点配置**

在 InstanceInfo 中，我们可以看到一些 URL 的配置信息，比如 homePageUrl、statusPageUrl、healthCheckUrl，它们分别代表了应用主页的 URL、状态页的 URL、健康检查的 URL。其中，状态页和健康检查的 URL 在 Spring Cloud Eureka 中默认使用了 spring-boot-actuator 模块提供的/info 端点和/health 端点。虽然我们在之前的示例中并没有对这些端点做具体的设置，但是实际上这些 URL 地址的配置非常重要。为了服务的正常运作，我们必须确保 Eureka 客户端的/health 端点在发送元数据的时候，是一个能够被注册中心访问到的地址，否则服务注册中心不会根据应用的健康检查来更改状态（仅当开启了 healthcheck 功能时，以该端点信息作为健康检查标准）。而/info 端点如果不正确的话，会导致在 Eureka 面板中单击服务实例时，无法访问到服务实例提供的信息接口。

大多数情况下，我们并不需要修改这几个 URL 的配置，但是在一些特殊情况下，比如，为应用设置了 context-path，这时，所有 spring-boot-actuator 模块的监控端点都会增加一个前缀。所以，我们就需要做类似如下的配置，为/info 和/health 端点也加上类似的前缀信息：

```
management.context-path=/hello

eureka.instance.statusPageUrlPath=${management.context-path}/info
eureka.instance.healthCheckUrlPath=${management.context-path}/health
```

另外，有时候为了安全考虑，也有可能会修改/info 和/health 端点的原始路径。这个时候，我们也需要做一些特殊的配置，比如像下面这样：

```
endpoints.info.path=/appInfo
endpoints.health.path=/checkHealth

eureka.instance.statusPageUrlPath=/${endpoints.info.path}
eureka.instance.healthCheckUrlPath=/${endpoints.health.path}
```

在上面所举的两个示例中，我们使用了 eureka.instance.statusPageUrlPath 和 eureka.instance.healthCheckUrlPath 参数，这两个配置值有一个共同特点，它们都使用相对路径来进行配置。由于 Eureka 的服务注册中心默认会以 HTTP 的方式来访问和暴露这些端点，因此当客户端应用以 HTTPS 的方式来暴露服务和监控端点时，相对路径的配置方式就无法满足需求了。所以，Spring Cloud Eureka 还提供了绝对路径的配置参数，具体示例如下所示：

```
eureka.instance.statusPageUrl=https://${eureka.instance.hostname}/info
eureka.instance.healthCheckUrl=https://${eureka.instance.hostname}/health
eureka.instance.homePageUrl=https://${eureka.instance.hostname}/
```

**健康检测**

默认情况下，Eureka 中各个服务实例的健康检测并不是通过 spring-boot-actuator 模块的/health 端点来实现的，而是依靠客户端心跳的方式来保持服务实例的存活。在 Eureka 的服务续约与剔除机制下，客户端的健康状态从注册到注册中心开始都会处于 UP 状态，除非心跳终止一段时间之后，服务注册中心将其剔除。默认的心跳实现方式可以有效检查客户端进程是否正常运作，但却无法保证客户端应用能够正常提供服务。由于大多数微服务应用都会有一些其他的外部资源依赖，比如数据库、缓存、消息代理等，如果我们的应用与这些外部资源无法联通的时候，实际上已经不能提供正常的对外服务了，但是因为客户端心跳依然在运行，所以它还是会被服务消费者调用，而这样的调用实际上并不能获得预期的结果。

在 Spring Cloud Eureka 中，我们可以通过简单的配置，把 Eureka 客户端的健康检测交给 spring-boot-actuator 模块的/health 端点，以实现更加全面的健康状态维护。详细的配置步骤如下所示：

- 在 pom.xml 中引入 spring-boot-starter-actuator 模块的依赖。
- 在 application.properties 中增加参数配置 eureka.client.healthcheck.enabled=true。
- 如果客户端的/health 端点路径做了一些特殊处理，请参考前文介绍端点配置时的方法进行配置，让服务注册中心可以正确访问到健康检测端点。

**其他配置**

除了上面介绍的配置参数外，下面整理了一些 org.springframework.cloud.netflix.eureka.EurekaInstanceConfigBean 中定义的配置参数以及对应的说明和默认值，这些参数均以 eureka.instance 为前缀。

| 参数名 | 说明 | 默认值 |
|---|---|---|
| preferIpAddress | 是否优先使用 IP 地址作为主机名的标识 | false |
| leaseRenewalIntervalInSeconds | Eureka 客户端向服务端发送心跳的时间间隔，单位为秒 | 30 |
| leaseExpirationDurationInSeconds | Eureka 服务端在收到最后一次心跳之后等待的时间上限，单位为秒。超过该时间之后服务端会将该服务实例从服务清单中剔除，从而禁止服务调用请求被发送到该实例上 | 90 |
| nonSecurePort | 非安全的通信端口号 | 80 |
| securePort | 安全的通信端口号 | 443 |
| nonSecurePortEnabled | 是否启用非安全的通信端口号 | true |
| securePortEnabled | 是否启用安全的通信端口号 | |
| appname | 服务名，默认取 `spring.application.name` 的配置值，如果没有则为 `unknown` | |
| hostname | 主机名，不配置的时候将根据操作系统的主机名来获取 | |

在上面的这些配置中，除了前三个配置参数在需要的时候可以做一些调整，其他的参数配置大多数情况下不需要进行配置，使用默认值即可。

## 跨平台支持

在“Eureka 详解”一节中，我们对 Spring Cloud Eureka 的源码做了较为详细的分析，在分析过程中相信大家已经发现，Eureka 的通信机制使用了 HTTP 的 REST 接口实现，这也是 Eureka 同其他服务注册工具的一个关键不同点。由于 HTTP 的平台无关性，虽然 Eureka Server 通过 Java 实现，但是在其下的微服务应用并不限于使用 Java 来进行开发。

跨平台本身就是微服务架构的九大特性之一，只有实现了对技术平台的透明，才能更好地发挥不同语言对不同业务处理能力的优势，从而打造更为强大的大型系统。

目前除了 Eureka 的 Java 客户端之外，还有很多其他语言平台对其的支持，比如 eureka-js-client、python-eureka 等。若我们有志于自己为一门语言来开发客户端程序，也并非特别复杂，只需要根据上面提到的那些用户服务协调的通信请求实现就能实现服务的注册与发现，需要了解更多关于 Eureka 的 REST API 内容可以查看 Eureka 官方 WiKi 中的 *Eureka-REST-operations* 一文 https://github.com/Netflix/eureka/wiki/Eureka-REST-operations。

### 通信协议

默认情况下，Eureka 使用 Jersey 和 XStream 配合 JSON 作为 Server 与 Client 之间的通信协议。你也可以选择实现自己的协议来代替。

- Jersey 是 JAX-RS 的参考实现，它包含三个主要部分。
  - **核心服务器**（Core Server）：通过提供 JSR 311 中标准化的注释和 API 标准化，你可以用直观的方式开发 RESTful Web 服务。
  - **核心客户端**（Core Client）：Jersey 客户端 API 帮助你与 REST 服务轻松通信。
  - **集成**（Integration）：Jersey 还提供可以轻松集成 Spring、Guice、Apache Abdera 的库。
- XStream 是用来将对象序列化成 XML（JSON）或反序列化为对象的一个 Java 类库。XStream 在运行时使用 Java 反射机制对要进行序列化的对象树的结构进行探索，并不需要对对象做出修改。XStream 可以序列化内部字段，包括 private 和 final 字段，并且支持非公开类以及内部类。默认情况下，XStream 不需要配置映射关系，对象和字段将映射为同名 XML 元素。但是当对象和字段名与 XML 中的元素名不同时，XStream 支持指定别名。XStream 支持以方法调用的方式，或是 Java 标注的方式指定别名。XStream 在进行数据类型转换时，使用系统默认的类型转换器。同时，也支持用户自定义的类型转换器。

JAX-RS 是将在 Java EE 6 中引入的一种新技术。JAX-RS 即 Java API for RESTful Web Services，是一个 Java 编程语言的应用程序接口，支持按照表述性状态转移（REST）架构风格创建 Web 服务。JAX-RS 使用了 Java SE 5 引入的 Java 标注来简化 Web 服务的客户端和服务端的开发和部署。包括：

- @Path，标注资源类或者方法的相对路径。
- @GET、@PUT、@POST、@DELETE，标注方法是 HTTP 请求的类型。
- @Produces，标注返回的 MIME 媒体类型。
- @Consumes，标注可接受请求的 MIME 媒体类型。
- @PathParam、@QueryParam、@HeaderParam、@CookieParam、@MatrixParam、@FormParam，标注方法的参数来自 HTTP 请求的不同位置，例如，@PathParam 来自 URL 的路径，@QueryParam 来自 URL 的查询参数，@HeaderParam 来自 HTTP 请求的头信息，@CookieParam 来自 HTTP 请求的 Cookie。

之前在分析 Eureka Server 端源码的时候，查看请求处理时可以看到很多上面这些注解的定义。

第 4 章

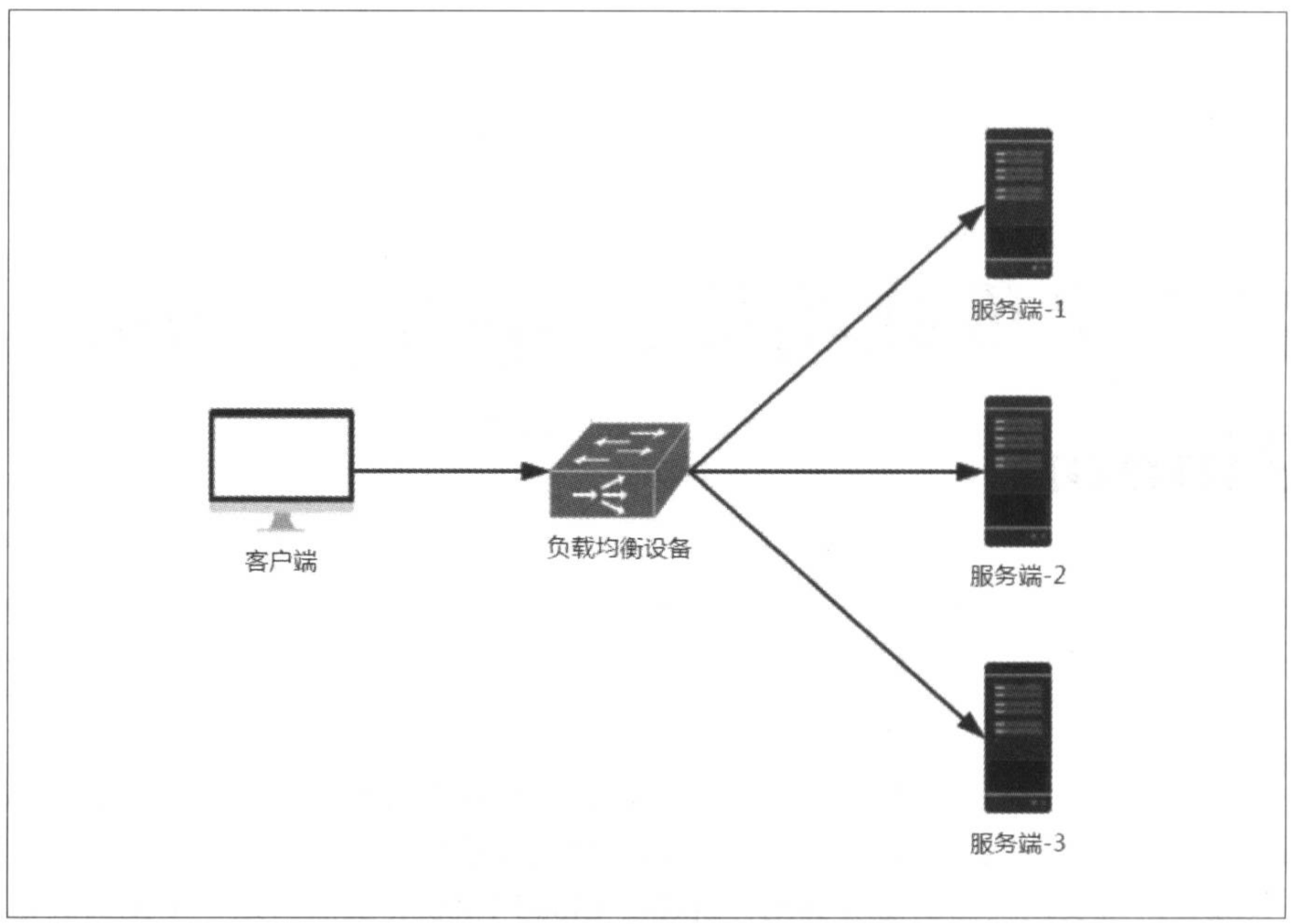
# 客户端负载均衡：Spring Cloud Ribbon

Spring Cloud Ribbon 是一个基于 HTTP 和 TCP 的客户端负载均衡工具，它基于 Netflix Ribbon 实现。通过 Spring Cloud 的封装，可以让我们轻松地将面向服务的 REST 模板请求自动转换成客户端负载均衡的服务调用。Spring Cloud Ribbon 虽然只是一个工具类框架，它不像服务注册中心、配置中心、API 网关那样需要独立部署，但是它几乎存在于每一个 Spring Cloud 构建的微服务和基础设施中。因为微服务间的调用，API 网关的请求转发等内容，实际上都是通过 Ribbon 来实现的，包括后续我们将要介绍的 Feign，它也是基于 Ribbon 实现的工具。所以，对 Spring Cloud Ribbon 的理解和使用，对于我们使用 Spring Cloud 来构建微服务非常重要。

在这一章中，我们将具体介绍如何使用 Ribbon 来实现客户端的负载均衡，并且通过源码分析来了解 Ribbon 实现客户端负载均衡的基本原理。

## 客户端负载均衡

负载均衡在系统架构中是一个非常重要，并且是不得不去实施的内容。因为负载均衡是对系统的高可用、网络压力的缓解和处理能力扩容的重要手段之一。我们通常所说的负载均衡都指的是服务端负载均衡，其中分为硬件负载均衡和软件负载均衡。硬件负载均衡主要通过在服务器节点之间安装专门用于负载均衡的设备，比如 F5 等；而软件负载均衡则是通过在服务器上安装一些具有均衡负载功能或模块的软件来完成请求分发工作，比如 Nginx 等。不论采用硬件负载均衡还是软件负载均衡，只要是服务端负载均衡都能以类似下图的架构方式构建起来：




硬件负载均衡的设备或是软件负载均衡的软件模块都会维护一个下挂可用的服务端清单，通过心跳检测来剔除故障的服务端节点以保证清单中都是可以正常访问的服务端节点。当客户端发送请求到负载均衡设备的时候，该设备按某种算法（比如线性轮询、按权重负载、按流量负载等）从维护的可用服务端清单中取出一台服务端的地址，然后进行转发。

而客户端负载均衡和服务端负载均衡最大的不同点在于上面所提到的服务清单所存储的位置。在客户端负载均衡中，所有客户端节点都维护着自己要访问的服务端清单，而这些服务端的清单来自于服务注册中心，比如上一章我们介绍的 Eureka 服务端。同服务端负载均衡的架构类似，在客户端负载均衡中也需要心跳去维护服务端清单的健康性，只是这个步骤需要与服务注册中心配合完成。在 Spring Cloud 实现的服务治理框架中，默认会创建针对各个服务治理框架的 Ribbon 自动化整合配置，比如 Eureka 中的 `org.springframework.cloud.netflix.ribbon.eureka.RibbonEurekaAutoConfiguration`，Consul 中的 `org.springframework.cloud.consul.discovery.RibbonConsulAutoConfiguration`。在实际使用的时候，我们可以通过查看这两个类的实现，以找到它们的配置详情来帮助我们更好地使用它。

通过 Spring Cloud Ribbon 的封装，我们在微服务架构中使用客户端负载均衡调用非常简单，只需要如下两步：

- 服务提供者只需要启动多个服务实例并注册到一个注册中心或是多个相关联的服务注册中心。
- 服务消费者直接通过调用被@LoadBalanced 注解修饰过的 RestTemplate 来实现面向服务的接口调用。

这样，我们就可以将服务提供者的高可用以及服务消费者的负载均衡调用一起实现了。

## RestTemplate 详解

在上一章中，我们已经通过引入 Ribbon 实现了服务消费者的客户端负载均衡功能，读者可以通过查看第 3 章中的“服务发现与消费”一节来获取实验示例。其中，我们使用了一个非常有用的对象 RestTemplate。该对象会使用 Ribbon 的自动化配置，同时通过配置@LoadBalanced 还能够开启客户端负载均衡。之前我们演示了通过 RestTemplate 实现了最简单的服务访问，下面我们将详细介绍 RestTemplate 针对几种不同请求类型和参数类型的服务调用实现。

### GET 请求

在 RestTemplate 中，对 GET 请求可以通过如下两个方法进行调用实现。

第一种：getForEntity 函数。该方法返回的是 ResponseEntity，该对象是 Spring 对 HTTP 请求响应的封装，其中主要存储了 HTTP 的几个重要元素，比如 HTTP 请求状态码的枚举对象 HttpStatus（也就是我们常说的 404、500 这些错误码）、在它的父类 HttpEntity 中还存储着 HTTP 请求的头信息对象 HttpHeaders 以及泛型类型的请求体对象。比如下面的例子，就是访问 USER-SERVER 服务的/user 请求，同时最后一个参数 didi 会替换 url 中的{1}占位符，而返回的 ResponseEntity 对象中的 body 内容类型会根据第二个参数转换为 String 类型。

```
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<String> responseEntity = restTemplate.getForEntity("http://USER-
SERVICE/user?name={1}", String.class, "didi");
    String body = responseEntity.getBody();
```

若我们希望返回的 body 是一个 User 对象类型，也可以这样实现：

```
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<User> responseEntity = restTemplate.getForEntity("http://USER-
SERVICE/user?name={1}", User.class, "didi");
    User body = responseEntity.getBody();
```

上面的例子是比较常用的方法，getForEntity 函数实际上提供了以下三种不同的重载实现。

- getForEntity(String url, Class responseType, Object... urlVariables)：该方法提供了三个参数，其中 url 为请求的地址，responseType 为请求响应体 body 的包装类型，urlVariables 为 url 中的参数绑定。GET 请求的参数绑定通常使用 url 中拼接的方式，比如 http://USER-SERVICE/user?name=didi，我们可以像这样自己将参数拼接到 url 中，但更好的方法是在 url 中使用占位符并配合 urlVariables 参数实现 GET 请求的参数绑定，比如 url 定义为 http://USER-SERVICE/user?name={1}，然后可以这样来调用：getForEntity("http://USER-SERVICE/user?name={1}", String.class, "didi")，其中第三个参数 didi 会替换 url 中的{1}占位符。这里需要注意的是，由于 urlVariables 参数是一个数组，所以它的顺序会对应 url 中占位符定义的数字顺序。
- getForEntity(String url, Class responseType, Map urlVariables)：该方法提供的参数中，只有 urlVariables 的参数类型与上面的方法不同。这里使用了 Map 类型，所以使用该方法进行参数绑定时需要在占位符中指定 Map 中参数的 key 值，比如 url 定义为 http://USER-SERVICE/user?name={name}，在 Map 类型的 urlVariables 中，我们就需要 put 一个 key 为 name 的参数来绑定 url 中{name}占位符的值，比如：

```
    RestTemplate restTemplate = new RestTemplate();
    Map<String, String> params = new HashMap<>();
    params.put("name", "dada");
    ResponseEntity<String> responseEntity = restTemplate.getForEntity("http://USER-
SERVICE/user?name={name}", String.class, params);
```

- getForEntity(URI url, Class responseType)：该方法使用 URI 对象来替代之前的 url 和 urlVariables 参数来指定访问地址和参数绑定。URI 是 JDK java.net 包下的一个类，它表示一个统一资源标识符（Uniform Resource Identifier）引用。比如下面的例子：

```
    RestTemplate restTemplate = new RestTemplate();
    UriComponents uriComponents = UriComponentsBuilder.fromUriString(
            "http://USER-SERVICE/user?name={name}")
            .build()
            .expand("dodo")
            .encode();
    URI uri = uriComponents.toUri();
    ResponseEntity<String> responseEntity = restTemplate.getForEntity(uri,
String.class).getBody();
```

更多关于如何定义一个 URI 的方法可以参见 JDK 文档，这里不做详细说明。

第二种：getForObject 函数。该方法可以理解为对 getForEntity 的进一步封装，它通过 HttpMessageConverterExtractor 对 HTTP 的请求响应体 body 内容进行对象转换，实现请求直接返回包装好的对象内容。比如：

```
RestTemplate restTemplate = new RestTemplate();
String result = restTemplate.getForObject(uri, String.class);
```

当 body 是一个 User 对象时，可以直接这样实现：

```
RestTemplate restTemplate = new RestTemplate();
User result = restTemplate.getForObject(uri, User.class);
```

当不需要关注请求响应除 body 外的其他内容时，该函数就非常好用，可以少一个从 Response 中获取 body 的步骤。它与 getForEntity 函数类似，也提供了三种不同的重载实现。

- getForObject(String url, Class responseType, Object... urlVariables)：与 getForEntity 的方法类似，url 参数指定访问的地址，responseType 参数定义该方法的返回类型，urlVariables 参数为 url 中占位符对应的参数。
- getForObject(String url, Class responseType, Map urlVariables)：在该函数中，使用 Map 类型的 urlVariables 替代上面数组形式的 urlVariables，因此使用时在 url 中需要将占位符的名称与 Map 类型中的 key 一一对应设置。
- getForObject(URI url, Class responseType)：该方法使用 URI 对象来替代之前的 url 和 urlVariables 参数使用。

## POST 请求

在 RestTemplate 中，对 POST 请求时可以通过如下三个方法进行调用实现。

第一种：postForEntity 函数。该方法同 GET 请求中的 getForEntity 类似，会在调用后返回 ResponseEntity<T>对象，其中 T 为请求响应的 body 类型。比如下面这个例子，使用 postForEntity 提交 POST 请求到 USER-SERVICE 服务的/user 接口，提交的 body 内容为 user 对象，请求响应返回的 body 类型为 String。

```
RestTemplate restTemplate = new RestTemplate();
User user = new User("didi", 30);
ResponseEntity<String> responseEntity =
restTemplate.postForEntity("http://USER-SERVICE/user", user, String.class);
String body = responseEntity.getBody();
```

postForEntity 函数也实现了三种不同的重载方法。

- postForEntity(String url, Object request, Class responseType, Object... uriVariables)

- postForEntity(String url, Object request, Class responseType, Map uriVariables)
- postForEntity(URI url, Object request, Class responseType)

这些函数中的参数用法大部分与 getForEntity 一致，比如，第一个重载函数和第二个重载函数中的 uriVariables 参数都用来对 url 中的参数进行绑定使用；responseType 参数是对请求响应的 body 内容的类型定义。这里需要注意的是新增加的 request 参数，该参数可以是一个普通对象，也可以是一个 HttpEntity 对象。如果是一个普通对象，而非 HttpEntity 对象的时候，RestTemplate 会将请求对象转换为一个 HttpEntity 对象来处理，其中 Object 就是 request 的类型，request 内容会被视作完整的 body 来处理；而如果 request 是一个 HttpEntity 对象，那么就会被当作一个完成的 HTTP 请求对象来处理，这个 request 中不仅包含了 body 的内容，也包含了 header 的内容。

第二种：postForObject 函数。该方法也跟 getForObject 的类型类似，它的作用是简化 postForEntity 的后续处理。通过直接将请求响应的 body 内容包装成对象来返回使用，比如下面的例子：

```
RestTemplate restTemplate = new RestTemplate();
User user = new User("didi", 20);
String postResult = restTemplate.postForObject("http://USER-SERVICE/user", user,
String.class);
```

postForObject 函数也实现了三种不同的重载方法：

- postForObject(String url, Object request, Class responseType, Object... uriVariables)
- postForObject(String url, Object request, Class responseType, Map uriVariables)
- postForObject(URI url, Object request, Class responseType)

这三个函数除了返回的对象类型不同，函数的传入参数均与 postForEntity 一致，因此可参考之前 postForEntity 的说明。

第三种：postForLocation 函数。该方法实现了以 POST 请求提交资源，并返回新资源的 URI，比如下面的例子：

```
User user = new User("didi", 40);
URI responseURI = restTemplate.postForLocation("http://USER-SERVICE/user", user);
```

postForLocation 函数也实现了三种不同的重载方法：

- postForLocation(String url, Object request, Object...urlVariables)

- postForLocation(String url, Object request, Map urlVariables)
- postForLocation(URI url, Object request)

由于 postForLocation 函数会返回新资源的 URI，该 URI 就相当于指定了返回类型，所以此方法实现的 POST 请求不需要像 postForEntity 和 postForObject 那样指定 responseType。其他的参数用法相同。

## PUT 请求

在 RestTemplate 中，对 PUT 请求可以通过 put 方法进行调用实现，比如：

```
RestTemplate restTemplate = new RestTemplate();
Long id = 10001L;
User user = new User("didi", 40);
restTemplate.put("http://USER-SERVICE/user/{1}", user, id);
```

put 函数也实现了三种不同的重载方法：

- put(String url, Object request, Object... urlVariables)
- put(String url, Object request, Map urlVariables)
- put(URI url, Object request)

put 函数为 void 类型，所以没有返回内容，也就没有其他函数定义的 responseType 参数，除此之外的其他传入参数定义与用法与 postForObject 基本一致。

## DELETE 请求

在 RestTemplate 中，对 DELETE 请求可以通过 delete 方法进行调用实现，比如：

```
RestTemplate restTemplate = new RestTemplate();
Long id = 10001L;
restTemplate.delete("http://USER-SERVICE/user/{1}", id);
```

delete 函数也实现了三种不同的重载方法：

- delete(String url, Object... urlVariables)
- delete(String url, Map urlVariables)
- delete(URI url)

由于我们在进行 REST 请求时，通常都将 DELETE 请求的唯一标识拼接在 url 中，所以 DELETE 请求也不需要 request 的 body 信息，就如上面的三个函数实现一样，非常简单。url 指定 DELETE 请求的位置，urlVariables 绑定 url 中的参数即可。

## 源码分析

相信很多熟悉 Spring 的读者看到这里一定会产生这样的疑问：RestTemplate 不是 Spring 自己就提供的吗？跟 Ribbon 的客户端负载均衡又有什么关系呢？在本节中，我们将透过现象看本质，探索一下 Ribbon 是如何通过 RestTemplate 实现客户端负载均衡的。

首先，回顾一下之前的消费者示例：我们是如何实现客户端负载均衡的？仔细观察一下之前的实现代码，可以发现在消费者的例子中，可能就@LoadBalanced 这个注解是之前没有接触过的，并且从命名上来看也与负载均衡相关。我们不妨以此为线索来看看 Spring Cloud Ribbon 的源码实现。

从@LoadBalanced 注解源码的注释中可以知道，该注解用来给 RestTemplate 做标记，以使用负载均衡的客户端（LoadBalancerClient）来配置它。

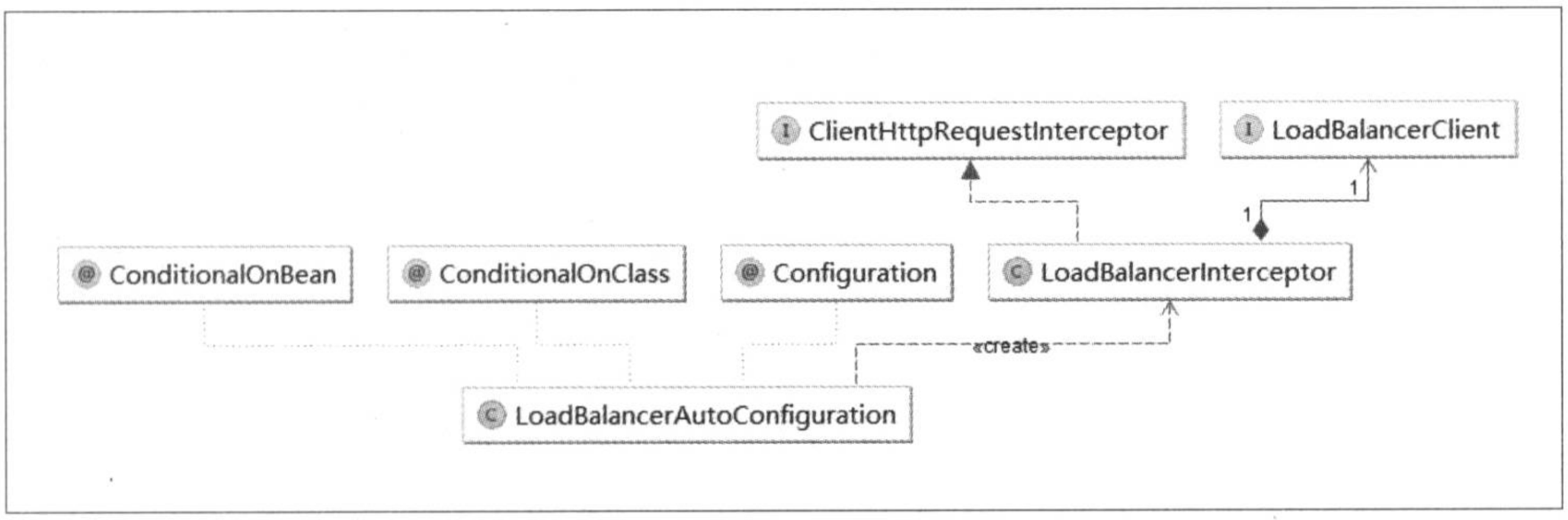
通过搜索 LoadBalancerClient 可以发现，这是 Spring Cloud 中定义的一个接口：

```
public interface LoadBalancerClient {

    ServiceInstance choose(String serviceId);

    <T> T execute(String serviceId, LoadBalancerRequest<T> request) throws
IOException;

    URI reconstructURI(ServiceInstance instance, URI original);

}
```

从该接口中，我们可以通过定义的抽象方法来了解客户端负载均衡器中应具备的几种能力。

- ServiceInstance choose(String serviceId)：根据传入的服务名 serviceId，从负载均衡器中挑选一个对应服务的实例。
- T execute(String serviceId, LoadBalancerRequest request) throws IOException：使用从负载均衡器中挑选出的服务实例来执行请求内容。
- URI reconstructURI(ServiceInstance instance, URI original)：为系统构建一个合适的 host:port 形式的 URI。在分布式系统中，我们使用逻辑上的服务名称作为 host 来构建 URI（替代服务实例的 host:port 形式）进行请求，比如 http://myservice/path/to/service。在该操作的定义中，前者 ServiceInstance 对象是带有 host 和 port 的具体服务实例，而后者 URI 对象则是使用逻辑服务名定义为 host 的 URI，而返回的 URI 内容则是通过 ServiceInstance 的服务实例详情拼接出的具体 host:post 形式的请求地址。

顺着 LoadBalancerClient 接口的所属包 org.springframework.cloud.client.loadbalancer，我们对其内容进行整理，可以得出如下图所示的关系。




从类的命名上可初步判断 LoadBalancerAutoConfiguration 为实现客户端负载均衡器的自动化配置类。通过查看源码，我们可以验证这一点假设：

```
@Configuration
@ConditionalOnClass(RestTemplate.class)
@ConditionalOnBean(LoadBalancerClient.class)
public class LoadBalancerAutoConfiguration {

    @LoadBalanced
    @Autowired(required = false)
    private List<RestTemplate> restTemplates = Collections.emptyList();

    @Bean
    public SmartInitializingSingleton loadBalancedRestTemplateInitializer(
            final List<RestTemplateCustomizer> customizers) {
        return new SmartInitializingSingleton() {
            @Override
            public void afterSingletonsInstantiated() {
                for (RestTemplate restTemplate : LoadBalancerAutoConfiguration.
this.restTemplates) {
                    for (RestTemplateCustomizer customizer : customizers) {
                        customizer.customize(restTemplate);
                    }
                }
            }
        };
    }

    @Bean
    @ConditionalOnMissingBean
```

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

    @Bean
    public LoadBalancerInterceptor ribbonInterceptor(
            LoadBalancerClient loadBalancerClient) {
        return new LoadBalancerInterceptor(loadBalancerClient);
    }

}
```

从 LoadBalancerAutoConfiguration 类头上的注解可以知道，Ribbon 实现的负载均衡自动化配置需要满足下面两个条件。

- @ConditionalOnClass(RestTemplate.class)：RestTemplate 类必须存在于当前工程的环境中。
- @ConditionalOnBean(LoadBalancerClient.class)：在 Spring 的 Bean 工程中必须有 LoadBalancerClient 的实现 Bean。

在该自动化配置类中，主要做了下面三件事：

- 创建了一个 LoadBalancerInterceptor 的 Bean，用于实现对客户端发起请求时进行拦截，以实现客户端负载均衡。
- 创建了一个 RestTemplateCustomizer 的 Bean，用于给 RestTemplate 增加 LoadBalancerInterceptor 拦截器。
- 维护了一个被@LoadBalanced 注解修饰的 RestTemplate 对象列表，并在这里进行初始化，通过调用 RestTemplateCustomizer 的实例来给需要客户端负载均衡的 RestTemplate 增加 LoadBalancerInterceptor 拦截器。

接下来，我们看看 LoadBalancerInterceptor 拦截器是如何将一个普通的 RestTemplate 变成客户端负载均衡的：

```
public class LoadBalancerInterceptor implements ClientHttpRequestInterceptor {

    private LoadBalancerClient loadBalancer;

    public LoadBalancerInterceptor(LoadBalancerClient loadBalancer) {
        this.loadBalancer = loadBalancer;
    }

    @Override
    public ClientHttpResponse intercept(final HttpRequest request, final byte[] body,
            final ClientHttpRequestExecution execution) throws IOException {
        final URI originalUri = request.getURI();
        String serviceName = originalUri.getHost();
        return this.loadBalancer.execute(serviceName,
                new LoadBalancerRequest<ClientHttpResponse>() {
                    @Override
                    public ClientHttpResponse apply(final ServiceInstance instance)
                            throws Exception {
                        HttpRequest serviceRequest = new ServiceRequestWrapper(request,
                                instance);
                        return execution.execute(serviceRequest, body);
                    }
                });
    }

    private class ServiceRequestWrapper extends HttpRequestWrapper {

        private final ServiceInstance instance;

        public ServiceRequestWrapper(HttpRequest request, ServiceInstance instance) {
            super(request);
            this.instance = instance;
        }

        @Override
        public URI getURI() {
            URI uri = LoadBalancerInterceptor.this.loadBalancer.reconstructURI(
                    this.instance, getRequest().getURI());
            return uri;
        }
    }
}
```

通过源码以及之前的自动化配置类，我们可以看到在拦截器中注入了 LoadBalancerClient 的实现。当一个被@LoadBalanced 注解修饰的 RestTemplate 对象向外发起 HTTP 请

求时，会被 LoadBalancerInterceptor 类的 intercept 函数所拦截。由于我们在使用 RestTemplate 时采用了服务名作为 host，所以直接从 HttpRequest 的 URI 对象中通过 getHost()就可以拿到服务名，然后调用 execute 函数去根据服务名来选择实例并发起实际的请求。

分析到这里，LoadBalancerClient 还只是一个抽象的负载均衡器接口，所以我们还需要找到它的具体实现类来进一步进行分析。通过查看 Ribbon 的源码，可以很容易地在 org.springframework.cloud.netflix.ribbon 包下找到对应的实现类 RibbonLoadBalancerClient。

```
    public <T> T execute(String serviceId, LoadBalancerRequest<T> request) throws
IOException {
        ILoadBalancer loadBalancer = getLoadBalancer(serviceId);
        Server server = getServer(loadBalancer);
        if (server == null) {
            throw new IllegalStateException("No instances available for " + serviceId);
        }
        RibbonServer ribbonServer = new RibbonServer(serviceId, server,
isSecure(server,
                serviceId), serverIntrospector(serviceId).getMetadata(server));

        RibbonLoadBalancerContext context = this.clientFactory
                .getLoadBalancerContext(serviceId);
        RibbonStatsRecorder statsRecorder = new RibbonStatsRecorder(context, server);

        try {
            T returnVal = request.apply(ribbonServer);
            statsRecorder.recordStats(returnVal);
            return returnVal;
        }
        catch (IOException ex) {
            statsRecorder.recordStats(ex);
            throw ex;
        }
        catch (Exception ex) {
            statsRecorder.recordStats(ex);
            ReflectionUtils.rethrowRuntimeException(ex);
        }
        return null;
    }
```

可以看到，在 execute 函数的实现中，第一步做的就是通过 getServer 根据传入的服务名 serviceId 去获得具体的服务实例：

```
protected Server getServer(ILoadBalancer loadBalancer) {
    if (loadBalancer == null) {
        return null;
    }
    return loadBalancer.chooseServer("default");
}
```

通过 getServer 函数的实现源码，我们可以看到这里获取具体服务实例的时候并没有使用 LoadBalancerClient 接口中的 choose 函数，而是使用了 Netflix Ribbon 自身的 ILoadBalancer 接口中定义的 chooseServer 函数。

我们先来认识一下这个 ILoadBalancer 接口：

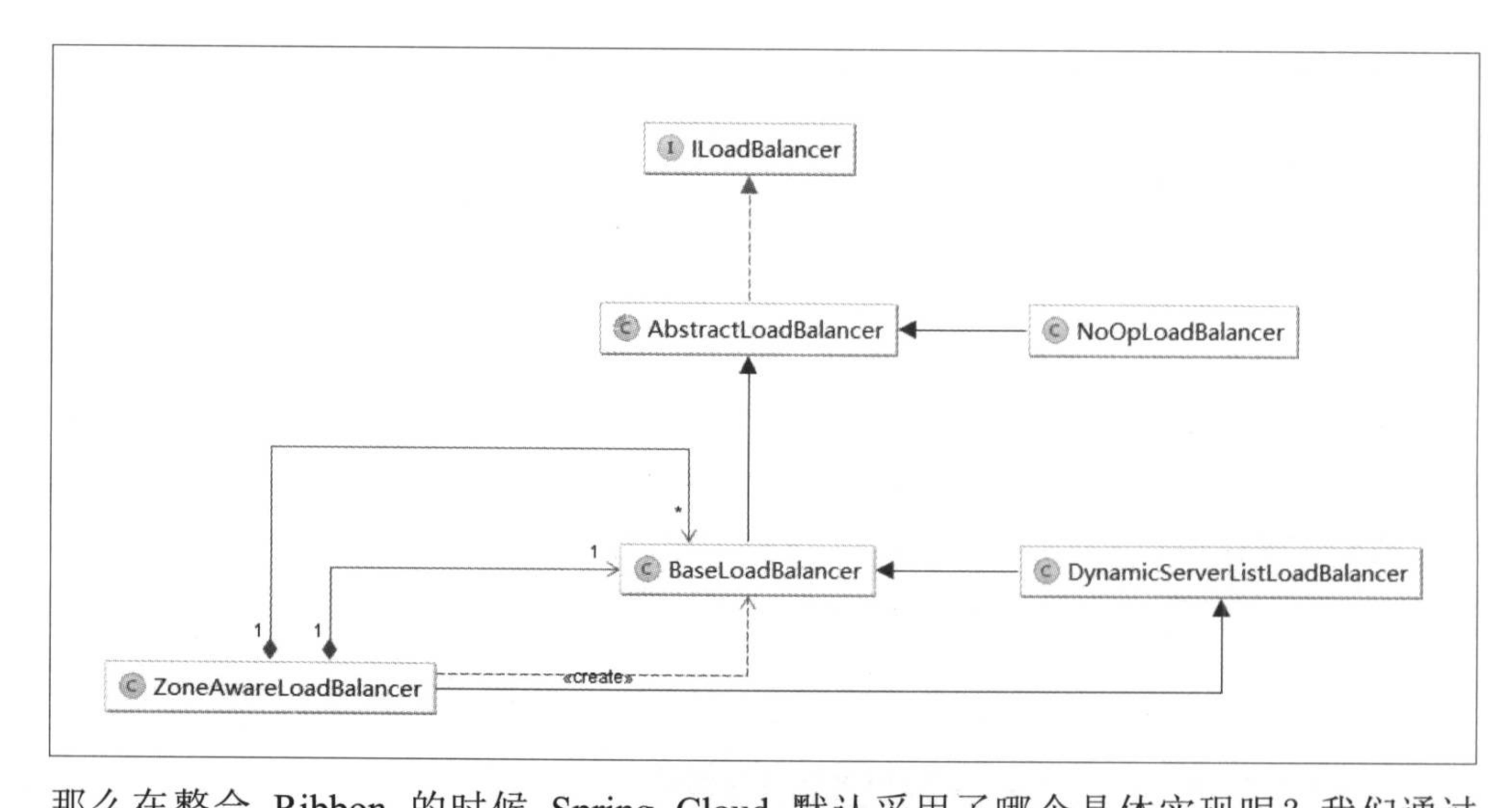
```
public interface ILoadBalancer {

    public void addServers(List<Server> newServers);

    public Server chooseServer(Object key);

    public void markServerDown(Server server);

    public List<Server> getReachableServers();

    public List<Server> getAllServers();
}
```

可以看到，在该接口中定义了一个客户端负载均衡器需要的一系列抽象操作（未列举过期函数）。

- addServers：向负载均衡器中维护的实例列表增加服务实例。
- chooseServer：通过某种策略，从负载均衡器中挑选出一个具体的服务实例。
- markServerDown：用来通知和标识负载均衡器中某个具体实例已经停止服务，不然负载均衡器在下一次获取服务实例清单前都会认为服务实例均是正常服务的。
- getReachableServers：获取当前正常服务的实例列表。
- getAllServers：获取所有已知的服务实例列表，包括正常服务和停止服务的实例。

在该接口定义中涉及的 Server 对象定义是一个传统的服务端节点，在该类中存储了服务端节点的一些元数据信息，包括 host、port 以及一些部署信息等。

而对于该接口的实现，我们整理出如下图所示的结构。可以看到，BaseLoadBalancer 类实现了基础的负载均衡，而 DynamicServerListLoadBalancer 和 ZoneAwareLoadBalancer 在负载均衡的策略上做了一些功能的扩展。




那么在整合 Ribbon 的时候 Spring Cloud 默认采用了哪个具体实现呢？我们通过 RibbonClientConfiguration 配置类，可以知道在整合时默认采用了 ZoneAware-LoadBalancer 来实现负载均衡器。

```
@Bean
@ConditionalOnMissingBean
public ILoadBalancer ribbonLoadBalancer(IClientConfig config,
        ServerList<Server> serverList, ServerListFilter<Server> serverListFilter,
        IRule rule, IPing ping) {
    ZoneAwareLoadBalancer<Server> balancer = LoadBalancerBuilder.newBuilder()
            .withClientConfig(config).withRule(rule).withPing(ping)
            .withServerListFilter(serverListFilter).withDynamicServerList(serverList)
            .buildDynamicServerListLoadBalancer();
    return balancer;
}
```

下面，我们再回到 RibbonLoadBalancerClient 的 execute 函数逻辑，在通过 ZoneAwareLoadBalancer 的 chooseServer 函数获取了负载均衡策略分配到的服务实例对象 Server 之后，将其内容包装成 RibbonServer 对象（该对象除了存储了服务实例的信息之外，还增加了服务名 serviceId、是否需要使用 HTTPS 等其他信息），然后使用该对象再回调 LoadBalancerInterceptor 请求拦截器中 LoadBalancerRequest 的 apply(final ServiceInstance instance) 函数，向一个实际的具体服务实例发起请求，从而实现一开始以服务名为 host 的 URI 请求到 host:post 形式的实际访问地址的转换。

在 apply(final ServiceInstance instance) 函数中传入的 ServiceInstance

接口对象是对服务实例的抽象定义。在该接口中暴露了服务治理系统中每个服务实例需要提供的一些基本信息，比如 serviceId、host、port 等，具体定义如下：

```
public interface ServiceInstance {

    String getServiceId();

    String getHost();

    int getPort();

    boolean isSecure();

    URI getUri();

    Map<String, String> getMetadata();
}
```

而上面提到的具体包装 Server 服务实例的 RibbonServer 对象就是 ServiceInstance 接口的实现，可以看到它除了包含 Server 对象之外，还存储了服务名、是否使用 HTTPS 标识以及一个 Map 类型的元数据集合。

```
protected static class RibbonServer implements ServiceInstance {

    private final String serviceId;
    private final Server server;
    private final boolean secure;
    private Map<String, String> metadata;

    protected RibbonServer(String serviceId, Server server) {
        this(serviceId, server, false, Collections.<String, String> emptyMap());
    }

    protected RibbonServer(String serviceId, Server server, boolean secure,
            Map<String, String> metadata) {
        this.serviceId = serviceId;
        this.server = server;
        this.secure = secure;
        this.metadata = metadata;
    }

    // 省略实现 ServiceInstance 的一些获取 Server 信息的 get 函数
    ...
}
```

那么 apply(final ServiceInstance instance) 函数在接收到了具体

ServiceInstance 实例后，是如何通过 LoadBalancerClient 接口中的 reconstructURI 操作来组织具体请求地址的呢？

```
@Override
public ClientHttpResponse apply(final ServiceInstance instance)
        throws Exception {
    HttpRequest serviceRequest = new ServiceRequestWrapper(request, instance);
    return execution.execute(serviceRequest, body);
}
```

从 apply 的实现中，可以看到它具体执行的时候，还传入了 ServiceRequestWrapper 对象，该对象继承了 HttpRequestWrapper 并重写了 getURI 函数，重写后的 getURI 通过调用 LoadBalancerClient 接口的 reconstructURI 函数来重新构建一个 URI 来进行访问。

```
private class ServiceRequestWrapper extends HttpRequestWrapper {

    private final ServiceInstance instance;

    ...

    @Override
    public URI getURI() {
        URI uri = LoadBalancerInterceptor.this.loadBalancer.reconstructURI(
                this.instance, getRequest().getURI());
        return uri;
    }
}
```

在 LoadBalancerInterceptor 拦截器中，ClientHttpRequestExecution 的实例具体执行 execution.execute(serviceRequest, body)时，会调用 InterceptingClientHttpRequest 下 InterceptingRequestExecution 类的 execute 函数，具体实现如下：

```
    public ClientHttpResponse execute(HttpRequest request, byte[] body) throws
IOException {
        if (this.iterator.hasNext()) {
            ClientHttpRequestInterceptor nextInterceptor = this.iterator.next();
            return nextInterceptor.intercept(request, body, this);
        }
        else {
            ClientHttpRequest delegate = requestFactory.createRequest(request.getURI(),
request.getMethod());
            delegate.getHeaders().putAll(request.getHeaders());
            if (body.length > 0) {
                StreamUtils.copy(body, delegate.getBody());
```

```
        }
        return delegate.execute();
    }
}
```

可以看到，在创建请求的时候 requestFactory.createRequest(request.getURI(), request.getMethod())；，这里的 request.getURI() 会调用之前介绍的 ServiceRequestWrapper 对象中重写的 getURI 函数。此时，它就会使用 RibbonLoadBalancerClient 中实现的 reconstructURI 来组织具体请求的服务实例地址。

```
public URI reconstructURI(ServiceInstance instance, URI original) {
    Assert.notNull(instance, "instance can not be null");
    String serviceId = instance.getServiceId();
    RibbonLoadBalancerContext context = this.clientFactory
            .getLoadBalancerContext(serviceId);
    Server server = new Server(instance.getHost(), instance.getPort());
    boolean secure = isSecure(server, serviceId);
    URI uri = original;
    if (secure) {
        uri = UriComponentsBuilder.fromUri(uri).scheme("https").build().toUri();
    }
    return context.reconstructURIWithServer(server, uri);
}
```

从 reconstructURI 函数中我们可以看到，它通过 ServiceInstance 实例对象的 serviceId，从 SpringClientFactory 类的 clientFactory 对象中获取对应 serviceId 的负载均衡器的上下文 RibbonLoadBalancerContext 对象。然后根据 ServiceInstance 中的信息来构建具体服务实例信息的 Server 对象，并使用 RibbonLoadBalancerContext 对象的 reconstructURIWithServer 函数来构建服务实例的 URI。

为了帮助理解，简单介绍一下上面提到的 SpringClientFactory 和 RibbonLoadBalancerContext：

- SpringClientFactory 类是一个用来创建客户端负载均衡器的工厂类，该工厂类会为每一个不同名的 Ribbon 客户端生成不同的 Spring 上下文。
- RibbonLoadBalancerContext 类是 LoadBalancerContext 的子类，该类用于存储一些被负载均衡器使用的上下文内容和 API 操作（reconstructURIWithServer 就是其中之一）。

从 reconstructURIWithServer 的实现中我们可以看到，它同 reconstructURI 的定义类似。只是 reconstructURI 的第一个保存具体服务实例的参数使用了 Spring Cloud 定义的 ServiceInstance，而 reconstructURIWithServer 中使用了 Netflix

中定义的 Server，所以在 RibbonLoadBalancerClient 实现 reconstructURI 的时候，做了一次转换，使用 ServiceInstance 的 host 和 port 信息构建了一个 Server 对象来给 reconstructURIWithServer 使用。从 reconstructURIWithServer 的实现逻辑中，我们可以看到，它从 Server 对象中获取 host 和 port 信息，然后根据以服务名为 host 的 URI 对象 original 中获取其他请求信息，将两者内容进行拼接整合，形成最终要访问的服务实例的具体地址。

```
public class LoadBalancerContext implements IClientConfigAware {

    ...

    public URI reconstructURIWithServer(Server server, URI original) {
        String host = server.getHost();
        int port = server .getPort();
        if (host.equals(original.getHost())
                && port == original.getPort()) {
            return original;
        }
        String scheme = original.getScheme();
        if (scheme == null) {
            scheme = deriveSchemeAndPortFromPartialUri(original).first();
        }

        try {
            StringBuilder sb = new StringBuilder();
            sb.append(scheme).append("://");
            if (!Strings.isNullOrEmpty(original.getRawUserInfo())) {
                sb.append(original.getRawUserInfo()).append("@");
            }
            sb.append(host);
            if (port >= 0) {
                sb.append(":").append(port);
            }
            sb.append(original.getRawPath());
            if (!Strings.isNullOrEmpty(original.getRawQuery())) {
                sb.append("?").append(original.getRawQuery());
            }
            if (!Strings.isNullOrEmpty(original.getRawFragment())) {
                sb.append("#").append(original.getRawFragment());
            }
            URI newURI = new URI(sb.toString());
            return newURI;
        } catch (URISyntaxException e) {
```

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

    ...
}
```

另外，从 RibbonLoadBalancerClient 的 execute 函数逻辑中，我们还能看到在回调拦截器中，执行具体的请求之后，Ribbon 还通过 RibbonStatsRecorder 对象对服务的请求进行了跟踪记录，这里不再展开说明，有兴趣的读者可以继续研究。

分析到这里，我们已经可以大致理清 Spring Cloud Ribbon 中实现客户端负载均衡的基本脉络，了解了它是如何通过 LoadBalancerInterceptor 拦截器对 RestTemplate 的请求进行拦截，并利用 Spring Cloud 的负载均衡器 LoadBalancerClient 将以逻辑服务名为 host 的 URI 转换成具体的服务实例地址的过程。同时通过分析 LoadBalancerClient 的 Ribbon 实现 RibbonLoadBalancerClient，可以知道在使用 Ribbon 实现负载均衡器的时候，实际使用的还是 Ribbon 中定义的 ILoadBalancer 接口的实现，自动化配置会采用 ZoneAwareLoadBalancer 的实例来实现客户端负载均衡。

## 负载均衡器

通过之前的分析，我们已经对 Spring Cloud 如何使用 Ribbon 有了基本的了解。虽然 Spring Cloud 中定义了 LoadBalancerClient 作为负载均衡器的通用接口，并且针对 Ribbon 实现了 RibbonLoadBalancerClient，但是它在具体实现客户端负载均衡时，是通过 Ribbon 的 ILoadBalancer 接口实现的。在上一节进行分析时候，我们对该接口的实现结构已经做了一些简单的介绍，下面我们根据 ILoadBalancer 接口的实现类逐个看看它是如何实现客户端负载均衡的。

### AbstractLoadBalancer

AbstractLoadBalancer 是 ILoadBalancer 接口的抽象实现。在该抽象类中定义了一个关于服务实例的分组枚举类 ServerGroup，它包含以下三种不同类型。

- ALL：所有服务实例。
- STATUS_UP：正常服务的实例。
- STATUS_NOT_UP：停止服务的实例。

另外，还实现了一个 chooseServer() 函数，该函数通过调用接口中的 chooseServer(Object key) 实现，其中参数 key 为 null，表示在选择具体服务实例时忽略 key 的条件判断。

最后，还定义了两个抽象函数。

- getServerList(ServerGroup serverGroup)：定义了根据分组类型来获取不同的服务实例的列表。
- getLoadBalancerStats()：定义了获取 LoadBalancerStats 对象的方法，LoadBalancerStats 对象被用来存储负载均衡器中各个服务实例当前的属性和统计信息。这些信息非常有用，我们可以利用这些信息来观察负载均衡器的运行情况，同时这些信息也是用来制定负载均衡策略的重要依据。

```
public abstract class AbstractLoadBalancer implements ILoadBalancer {

    public enum ServerGroup{
        ALL,
        STATUS_UP,
        STATUS_NOT_UP
    }

    public Server chooseServer() {
        return chooseServer(null);
    }

    public abstract List<Server> getServerList(ServerGroup serverGroup);

    public abstract LoadBalancerStats getLoadBalancerStats();
}
```

### BaseLoadBalancer

BaseLoadBalancer 类是 Ribbon 负载均衡器的基础实现类，在该类中定义了很多关于负载均衡器相关的基础内容。

- 定义并维护了两个存储服务实例 Server 对象的列表。一个用于存储所有服务实例的清单，一个用于存储正常服务的实例清单。

```
@Monitor(name = PREFIX + "AllServerList", type = DataSourceType.INFORMATIONAL)
protected volatile List<Server> allServerList = Collections
    .synchronizedList(new ArrayList<Server>());
@Monitor(name = PREFIX + "UpServerList", type = DataSourceType.INFORMATIONAL)
protected volatile List<Server> upServerList = Collections
    .synchronizedList(new ArrayList<Server>());
```

- 定义了之前我们提到的用来存储负载均衡器各服务实例属性和统计信息的 LoadBalancerStats 对象。

- 定义了检查服务实例是否正常服务的 IPing 对象，在 BaseLoadBalancer 中默认为 null，需要在构造时注入它的具体实现。
- 定义了检查服务实例操作的执行策略对象 IPingStrategy，在 BaseLoadBalancer 中默认使用了该类中定义的静态内部类 SerialPingStrategy 实现。根据源码，我们可以看到该策略采用线性遍历 ping 服务实例的方式实现检查。该策略在当 IPing 的实现速度不理想，或是 Server 列表过大时，可能会影响系统性能，这时候需要通过实现 IPingStrategy 接口并重写 pingServers(IPing ping, Server[] servers)函数去扩展 ping 的执行策略。

```
private static class SerialPingStrategy implements IPingStrategy {
    @Override
    public boolean[] pingServers(IPing ping, Server[] servers) {
        int numCandidates = servers.length;
        boolean[] results = new boolean[numCandidates];

        if (logger.isDebugEnabled()) {
            logger.debug("LoadBalancer: PingTask executing ["
                    + numCandidates + "] servers configured");
        }

        for (int i = 0; i < numCandidates; i++) {
            results[i] = false;
            try {
                if (ping != null) {
                    results[i] = ping.isAlive(servers[i]);
                }
            } catch (Throwable t) {
                logger.error("Exception while pinging Server:"
                        + servers[i], t);
            }
        }
        return results;
    }
}
```

- 定义了负载均衡的处理规则 IRule 对象，从 BaseLoadBalancer 中 chooseServer(Object key)的实现源码，我们可以知道，负载均衡器实际将服务实例选择任务委托给了 IRule 实例中的 choose 函数来实现。而在这里，默认初始化了 RoundRobinRule 为 IRule 的实现对象。RoundRobinRule 实现了最基本且常用的线性负载均衡规则。

```
public Server chooseServer(Object key) {
    if (counter == null) {
```

```
        counter = createCounter();
    }
    counter.increment();
    if (rule == null) {
        return null;
    } else {
        try {
            return rule.choose(key);
        } catch (Throwable t) {
            return null;
        }
    }
}
```

- 启动 ping 任务：在 BaseLoadBalancer 的默认构造函数中，会直接启动一个用于定时检查 Server 是否健康的任务。该任务默认的执行间隔为 10 秒。
- 实现了 ILoadBalancer 接口定义的负载均衡器应具备以下一系列基本操作。
  - addServers(List newServers)：向负载均衡器中增加新的服务实例列表，该实现将原本已经维护着的所有服务实例清单 allServerList 和新传入的服务实例清单 newServers 都加入到 newList 中，然后通过调用 setServersList 函数对 newList 进行处理，在 BaseLoadBalancer 中实现的时候会使用新的列表覆盖旧的列表。而之后介绍的几个扩展实现类对于服务实例清单更新的优化都是通过对 setServersList 函数的重写来实现的。

```
public void addServers(List<Server> newServers) {
if (newServers != null && newServers.size() > 0) {
    try {
        ArrayList<Server> newList = new ArrayList<Server>();
        newList.addAll(allServerList);
        newList.addAll(newServers);
        setServersList(newList);
    } catch (Exception e) {
        logger.error("Exception while adding Servers", e);
    }
}
}
```

- chooseServer(Object key)：挑选一个具体的服务实例，在上面介绍 IRule 的时候，已经做了说明，这里不再赘述。
- markServerDown(Server server)：标记某个服务实例暂停服务。

```
public void markServerDown(Server server) {
    if (server == null) {
```

```
        return;
    }
    if (!server.isAlive()) {
        return;
    }
    logger.error("LoadBalancer: markServerDown called on ["
            + server.getId() + "]");
    server.setAlive(false);
    notifyServerStatusChangeListener(singleton(server));
}
```

- getReachableServers()：获取可用的服务实例列表。由于 BaseLoadBalancer 中单独维护了一个正常服务的实例清单，所以直接返回即可。

```
public List<Server> getReachableServers() {
    return Collections.unmodifiableList(upServerList);
}
```

- getAllServers()：获取所有的服务实例列表。由于 BaseLoadBalancer 中单独维护了一个所有服务的实例清单，所以也直接返回它即可。

```
public List<Server> getAllServers() {
    return Collections.unmodifiableList(allServerList);
}
```

### DynamicServerListLoadBalancer

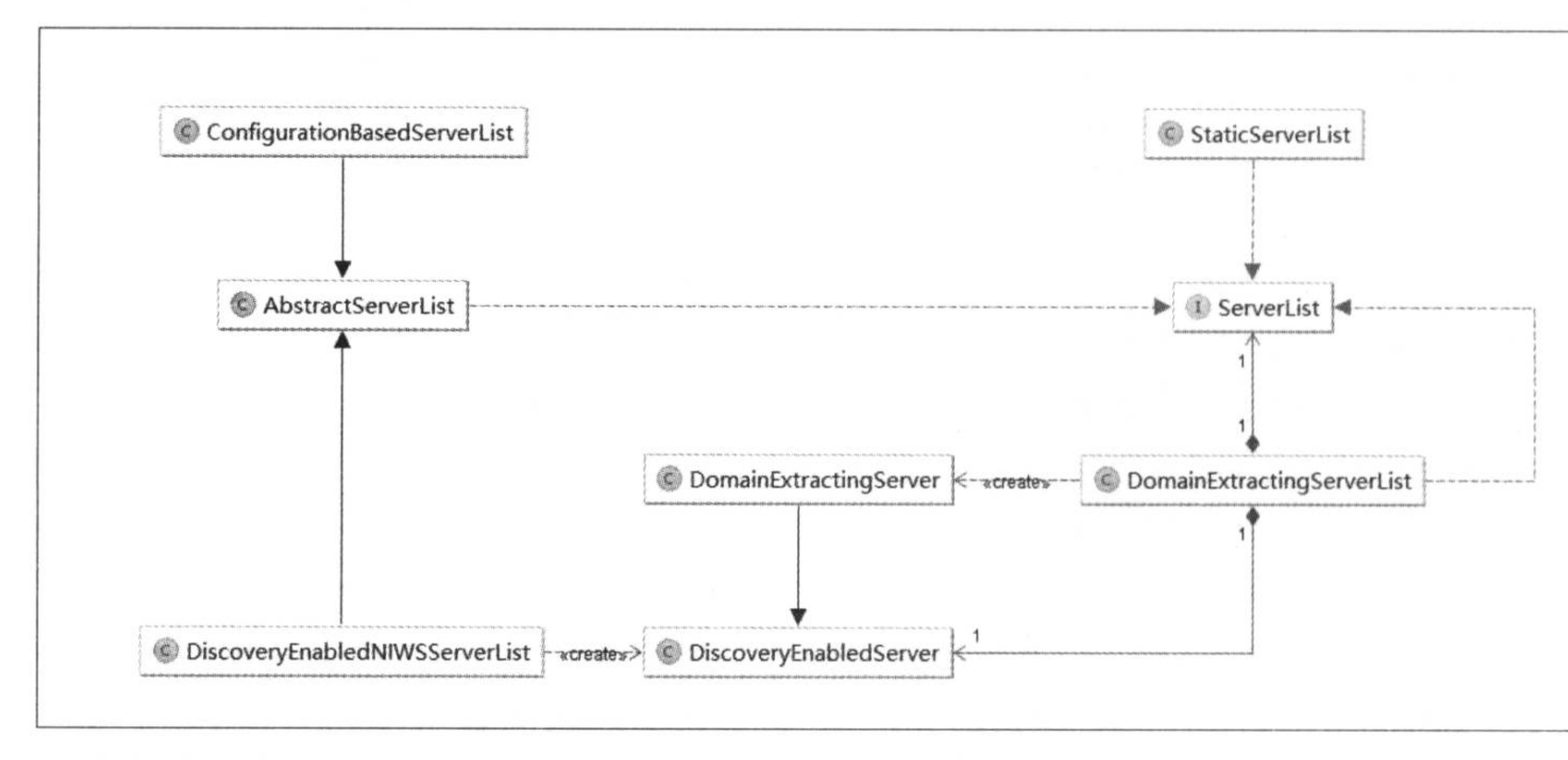
DynamicServerListLoadBalancer 类继承于 BaseLoadBalancer 类，它是对基础负载均衡器的扩展。在该负载均衡器中，实现了服务实例清单在运行期的动态更新能力；同时，它还具备了对服务实例清单的过滤功能，也就是说，我们可以通过过滤器来选择性地获取一批服务实例清单。下面我们具体来看看在该类中增加了一些什么内容。

#### ServerList

从 DynamicServerListLoadBalancer 的成员定义中，我们马上可以发现新增了一个关于服务列表的操作对象 ServerList<T> serverListImpl。其中泛型 T 从类名中对于 T 的限定 DynamicServerListLoadBalancer<T extends Server>可以获知它是一个 Server 的子类，即代表了一个具体的服务实例的扩展类。而 ServerList 接口定义如下所示：

```
public interface ServerList<T extends Server> {

    public List<T> getInitialListOfServers();

    public List<T> getUpdatedListOfServers();
}
```

它定义了两个抽象方法：getInitialListOfServers 用于获取初始化的服务实例清单，而 getUpdatedListOfServers 用于获取更新的服务实例清单。那么该接口的实现有哪些呢？通过搜索源码，我们可以整理出如下图所示的结构。




从上图中我们可以看到有多个 ServerList 的实现类，那么在 DynamicServerListLoadBalancer 中的 ServerList 默认配置到底使用了哪个具体实现呢？既然在该负载均衡器中需要实现服务实例的动态更新，那么势必需要 Ribbon 具备访问 Eureka 来获取服务实例的能力，所以我们从 Spring Cloud 整合 Ribbon 与 Eureka 的包 org.springframework.cloud.netflix.ribbon.eureka 下进行探索，可以找到配置类 EurekaRibbonClientConfiguration，在该类中可以找到如下创建 ServerList 实例的内容：

```
    @Bean
    @ConditionalOnMissingBean
    public ServerList<?> ribbonServerList(IClientConfig config) {
        DiscoveryEnabledNIWSServerList discoveryServerList = new
DiscoveryEnabledNIWSServerList(
                config);
        DomainExtractingServerList serverList = new DomainExtractingServerList(
                discoveryServerList, config, this.approximateZoneFromHostname);
        return serverList;
    }
```

可以看到，这里创建的是一个 DomainExtractingServerList 实例，从下面它的源码中我们可以看到，在它内部还定义了一个 ServerList list。同时，DomainExtractingServerList 类中对 getInitialListOfServers 和 getUpdatedListOfServers 的具体实现，其实委托给了内部定义的 ServerList list 对象，而该

对象是通过创建 DomainExtractingServerList 时，由构造函数传入的 DiscoveryEnabledNIWSServerList 实现的。

```
public class DomainExtractingServerList implements ServerList<DiscoveryEnabledServer> {

    private ServerList<DiscoveryEnabledServer> list;
    private IClientConfig clientConfig;
    private boolean approximateZoneFromHostname;

    public DomainExtractingServerList(ServerList<DiscoveryEnabledServer> list,
            IClientConfig clientConfig, boolean approximateZoneFromHostname) {
        this.list = list;
        this.clientConfig = clientConfig;
        this.approximateZoneFromHostname = approximateZoneFromHostname;
    }

    @Override
    public List<DiscoveryEnabledServer> getInitialListOfServers() {
        List<DiscoveryEnabledServer> servers = setZones(this.list
                .getInitialListOfServers());
        return servers;
    }

    @Override
    public List<DiscoveryEnabledServer> getUpdatedListOfServers() {
        List<DiscoveryEnabledServer> servers = setZones(this.list
                .getUpdatedListOfServers());
        return servers;
    }
    ...
}
```

那么 DiscoveryEnabledNIWSServerList 是如何实现这两个服务实例获取的呢？我们从源码中可以看到这两个方法都是通过该类中的一个私有函数 obtainServersViaDiscovery 通过服务发现机制来实现服务实例的获取的。

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

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

而obtainServersViaDiscovery的实现逻辑如下所示，主要依靠EurekaClient从服务注册中心中获取到具体的服务实例 InstanceInfo 列表（EurekaClient 的具体实现，我们在分析 Eureka 的源码时已经做了详细的介绍，这里传入的 vipAddress 可以理解为逻辑上的服务名，比如 USER-SERVICE）。接着，对这些服务实例进行遍历，将状态为 UP（正常服务）的实例转换成 DiscoveryEnabledServer 对象，最后将这些实例组织成列表返回。

```
    private List<DiscoveryEnabledServer> obtainServersViaDiscovery() {
        List<DiscoveryEnabledServer> serverList = new
ArrayList<DiscoveryEnabledServer>();

        if (eurekaClientProvider == null || eurekaClientProvider.get() == null) {
            logger.warn("EurekaClient has not been initialized yet, returning an empty
list");
            return new ArrayList<DiscoveryEnabledServer>();
        }

        EurekaClient eurekaClient = eurekaClientProvider.get();
        if (vipAddresses!=null){
            for (String vipAddress : vipAddresses.split(",")) {
                List<InstanceInfo> listOfInstanceInfo =
eurekaClient.getInstancesByVipAddress(
                        vipAddress, isSecure, targetRegion);
                for (InstanceInfo ii : listOfInstanceInfo) {
                    if (ii.getStatus().equals(InstanceStatus.UP)) {
                        // 省略了一些实例信息的加工逻辑
                        DiscoveryEnabledServer des = new DiscoveryEnabledServer(ii,
isSecure, shouldUseIpAddr);
                        des.setZone(DiscoveryClient.getZone(ii));
                        serverList.add(des);
                    }
                }
                if (serverList.size()>0 && prioritizeVipAddressBasedServers){
                    break;
                }
            }
        }
        return serverList;
    }
```

在 DiscoveryEnabledNIWSServerList 中通过 EurekaClient 从服务注册中心获取到最新的服务实例清单后，返回的 List 到了 DomainExtractingServerList 类中，将继续通过 setZones 函数进行处理。而这里的处理具体内容如下所示，主要完成将 DiscoveryEnabledNIWSServerList 返回的 List 列表中的元素，转换成内部定义的

DiscoveryEnabledServer 的子类对象 DomainExtractingServer，在该对象的构造函数中将为服务实例对象设置一些必要的属性，比如 id、zone、isAliveFlag、readyToServe 等信息。

```
    private List<DiscoveryEnabledServer> setZones(List<DiscoveryEnabledServer>
servers) {
        List<DiscoveryEnabledServer> result = new ArrayList<>();
        boolean isSecure = this.clientConfig.getPropertyAsBoolean(
                CommonClientConfigKey.IsSecure, Boolean.TRUE);
        boolean shouldUseIpAddr = this.clientConfig.getPropertyAsBoolean(
                CommonClientConfigKey.UseIPAddrForServer, Boolean.FALSE);
        for (DiscoveryEnabledServer server : servers) {
            result.add(new DomainExtractingServer(server, isSecure, shouldUseIpAddr,
                    this.approximateZoneFromHostname));
        }
        return result;
    }
```

ServerListUpdater

通过上面的分析我们已经知道了 Ribbon 与 Eureka 整合后，如何实现从 Eureka Server 中获取服务实例清单。那么它又是如何触发向 Eureka Server 去获取服务实例清单以及如何在获取到服务实例清单后更新本地的服务实例清单的呢？继续来看 DynamicServerListLoadBalancer 中的实现内容，我们可以很容易地找到下面定义的关于 ServerListUpdater 的内容：

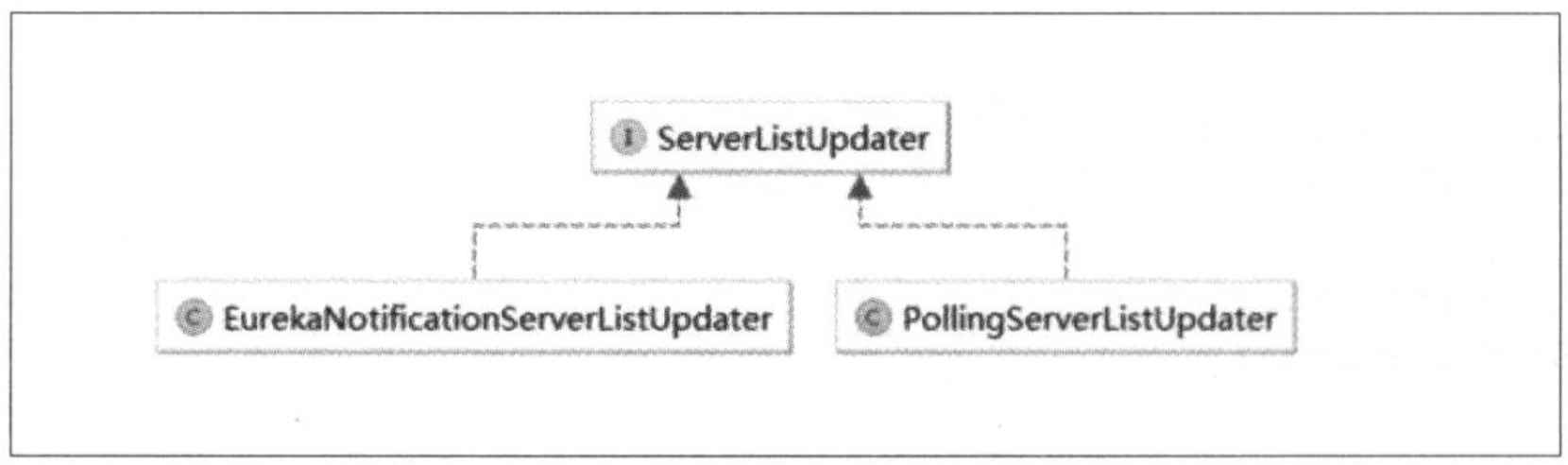
```
    protected final ServerListUpdater.UpdateAction updateAction = new
ServerListUpdater.UpdateAction() {
        @Override
        public void doUpdate() {
            updateListOfServers();
        }
    };

    protected volatile ServerListUpdater serverListUpdater;
```

根据该接口的命名，我们基本就能猜到，这个对象实现的是对 ServerList 的更新，所以可以称它为“服务更新器”。从下面的源码中可以看到，在 ServerListUpdater 内部还定义了一个 UpdateAction 接口，上面定义的 updateAction 对象就是以匿名内部类的方式创建了一个它的具体实现，其中 doUpdate 实现的内容就是对 ServerList 的具体更新操作。除此之外，“服务更新器”中还定义了一系列控制它和获取它的信息的操作。

```
    public interface ServerListUpdater {

        public interface UpdateAction {
```

```
    void doUpdate();
}

// 启动服务更新器，传入的 UpdateAction 对象为更新操作的具体实现。
void start(UpdateAction updateAction);

// 停止服务更新器
void stop();

// 获取最近的更新时间戳
String getLastUpdate();

// 获取上一次更新到现在的时间间隔，单位为毫秒
long getDurationSinceLastUpdateMs();

// 获取错过的更新周期数
int getNumberMissedCycles();

// 获取核心线程数
int getCoreThreads();
}
```

而 ServerListUpdater 的实现类不多，具体如下图所示。




根据两个类的注释，我们可以很容易地知道它们的作用。

- PollingServerListUpdater：动态服务列表更新的默认策略，也就是说，DynamicServerListLoadBalancer 负载均衡器中的默认实现就是它，它通过定时任务的方式进行服务列表的更新。
- EurekaNotificationServerListUpdater：该更新器也可服务于 DynamicServerListLoadBalancer 负载均衡器，但是它的触发机制与 PollingServerListUpdater 不同，它需要利用 Eureka 的事件监听器来驱动服务列表的更新操作。

下面我们来详细看看它默认实现的 PollingServerListUpdater。先从用于启动“服务更新器”的 start 函数源码看起，具体如下。我们可以看到 start 函数的实现内容

验证了之前提到的：以定时任务的方式进行服务列表的更新。它先创建了一个 Runnable 的线程实现，在该实现中调用了上面提到的具体更新服务实例列表的方法 updateAction.doUpdate()，最后再为这个 Runnable 线程实现启动了一个定时任务来执行。

```
    @Override
    public synchronized void start(final UpdateAction updateAction) {
        if (isActive.compareAndSet(false, true)) {
            final Runnable wrapperRunnable = new Runnable() {
                @Override
                public void run() {
                    if (!isActive.get()) {
                        if (scheduledFuture != null) {
                            scheduledFuture.cancel(true);
                        }
                        return;
                    }
                    try {
                        updateAction.doUpdate();
                        lastUpdated = System.currentTimeMillis();
                    } catch (Exception e) {
                        logger.warn("Failed one update cycle", e);
                    }
                }
            };

            scheduledFuture = getRefreshExecutor().scheduleWithFixedDelay(
                    wrapperRunnable,
                    initialDelayMs,
                    refreshIntervalMs,
                    TimeUnit.MILLISECONDS
            );
        } else {
            logger.info("Already active, no-op");
        }
    }
```

继续看 PollingServerListUpdater 的其他内容，我们可以找到用于启动定时任务的两个重要参数 initialDelayMs 和 refreshIntervalMs 的默认定义分别为 1000 和 30*1000，单位为毫秒。也就是说，更新服务实例在初始化之后延迟 1 秒后开始执行，并以 30 秒为周期重复执行。除了这些内容之外，还能看到它还会记录最后更新时间、是否存活等信息，同时也实现了 ServerListUpdater 中定义的一些其他操作内容，这些操作相对比较简单，这里不再具体说明，有兴趣的读者可以自己查看源码了解其实现原理。

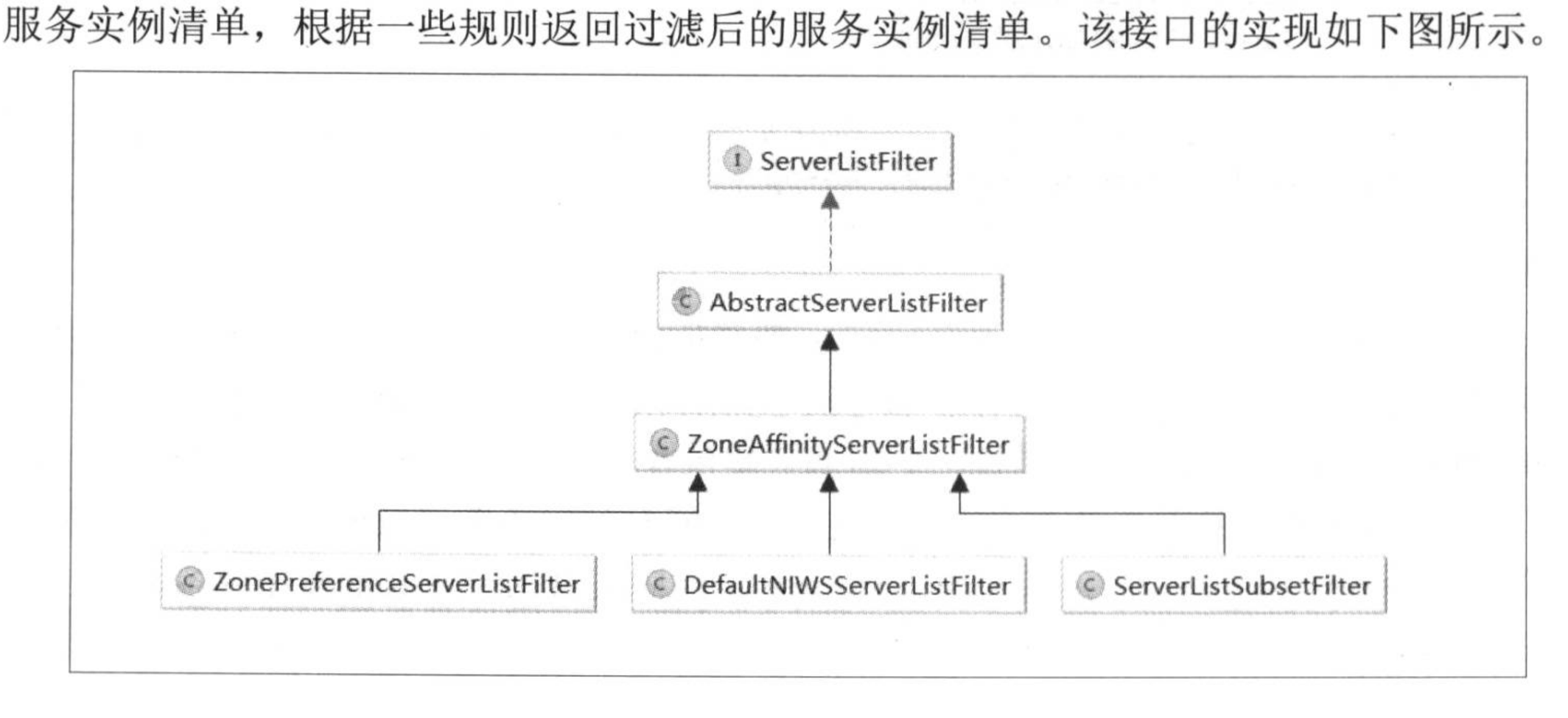
ServerListFilter

在了解了更新服务实例的定时任务是如何启动的之后，我们回到 updateAction.doUpdate()调用的具体实现位置，在 DynamicServerListLoadBalancer 中，它的实际实现委托给了 updateListOfServers 函数，具体实现如下：

```
public void updateListOfServers() {
    List<T> servers = new ArrayList<T>();
    if (serverListImpl != null) {
        servers = serverListImpl.getUpdatedListOfServers();
        LOGGER.debug("List of Servers for {} obtained from Discovery client: {}",
              getIdentifier(), servers);

        if (filter != null) {
            servers = filter.getFilteredListOfServers(servers);
            LOGGER.debug("Filtered List of Servers for {} obtained from Discovery
client: {}",
                  getIdentifier(), servers);
        }
    }
    updateAllServerList(servers);
}
```

可以看到，这里终于用到了之前提到的 ServerList 的 getUpdatedListOfServers()，通过之前的介绍已经知道这一步实现了从 Eureka Server 中获取服务可用实例的列表。在获得了服务实例列表之后，这里又将引入一个新的对象 filter，追溯该对象的定义，我们可以找到它是 ServerListFilter 定义的。

ServerListFilter 接口非常简单，该接口中定义了一个方法 List getFiltered-ListOfServers(List servers)，主要用于实现对服务实例列表的过滤，通过传入的服务实例清单，根据一些规则返回过滤后的服务实例清单。该接口的实现如下图所示。



其中，除了 ZonePreferenceServerListFilter 的实现是 Spring Cloud Ribbon 中对 Netflix Ribbon 的扩展实现外，其他均是 Netflix Ribbon 中的原生实现类。下面，我们可以分别看看这些过滤器实现都有什么特点。

- AbstractServerListFilter：这是一个抽象过滤器，在这里定义了过滤时需要的一个重要依据对象 LoadBalancerStats，我们在之前介绍过，该对象存储了关于负载均衡器的一些属性和统计信息等。

```
public abstract class AbstractServerListFilter<T extends Server> implements
ServerListFilter<T> {

    private volatile LoadBalancerStats stats;

    public void setLoadBalancerStats(LoadBalancerStats stats) {
        this.stats = stats;
    }

    public LoadBalancerStats getLoadBalancerStats() {
        return stats;
    }
}
```

- ZoneAffinityServerListFilter：该过滤器基于“区域感知（Zone Affinity）”的方式实现服务实例的过滤，也就是说，它会根据提供服务的实例所处的区域（Zone）与消费者自身的所处区域（Zone）进行比较，过滤掉那些不是同处一个区域的实例。

```
public List<T> getFilteredListOfServers(List<T> servers) {
    if (zone != null && (zoneAffinity || zoneExclusive) && servers !=null &&
servers.size() > 0){
        List<T> filteredServers = Lists.newArrayList(Iterables.filter(
                servers, this.zoneAffinityPredicate.getServerOnlyPredicate()));
        if (shouldEnableZoneAffinity(filteredServers)) {
            return filteredServers;
        } else if (zoneAffinity) {
            overrideCounter.increment();
        }
    }
    return servers;
}
```

从上面的源码中我们可以看到，对于服务实例列表的过滤是通过 Iterables.filter(servers,this.zoneAffinityPredicate.getServerOnlyPredicate()) 来实现的，其中判断依据由 ZoneAffinityPredicate 实现服务实例与消费者的 Zone

比较。而在过滤之后，这里并不会马上返回过滤的结果，而是通过 shouldEnableZoneAffinity 函数来判断是否要启用“区域感知”的功能。从下面 shouldEnableZoneAffinity 的实现中，我们可以看到，它使用了 LoadBalancerStats 的 getZoneSnapshot 方法来获取这些过滤后的同区域实例的基础指标（包含实例数量、断路器断开数、活动请求数、实例平均负载等），根据一系列的算法求出下面的几个评价值并与设置的阈值进行对比（下面的为默认值），若有一个条件符合，就不启用“区域感知”过滤的服务实例清单。这一算法实现为集群出现区域故障时，依然可以依靠其他区域的实例进行正常服务提供了完善的高可用保障。同时，通过这里的介绍，我们也可以关联着来理解之前介绍 Eureka 的时候提到的对于区域分配设计来保证跨区域故障的高可用问题。

- blackOutServerPercentage：故障实例百分比（断路器断开数/实例数量）>= 0.8。
- activeReqeustsPerServer：实例平均负载 >= 0.6。
- availableServers：可用实例数（实例数量 - 断路器断开数）< 2。

```
private boolean shouldEnableZoneAffinity(List<T> filtered) {
    if (!zoneAffinity && !zoneExclusive) {
        return false;
    }
    if (zoneExclusive) {
        return true;
    }
    LoadBalancerStats stats = getLoadBalancerStats();
    if (stats == null) {
        return zoneAffinity;
    } else {
        logger.debug("Determining if zone affinity should be enabled with given server
list: {}", filtered);
        ZoneSnapshot snapshot = stats.getZoneSnapshot(filtered);
        double loadPerServer = snapshot.getLoadPerServer();
        int instanceCount = snapshot.getInstanceCount();
        int circuitBreakerTrippedCount = snapshot.getCircuitTrippedCount();
        if (((double) circuitBreakerTrippedCount) / instanceCount >=
blackOutServerPercentageThreshold.get()
                || loadPerServer >= activeReqeustsPerServerThreshold.get()
                || (instanceCount - circuitBreakerTrippedCount) <
availableServersThreshold.get()) {
                logger.debug("zoneAffinity is overriden. blackOutServerPercentage: {},
activeReqeustsPerServer: {}, availableServers: {}",
                new Object[] {(double) circuitBreakerTrippedCount / instanceCount,
loadPerServer, instanceCount - circuitBreakerTrippedCount});
                return false;
```

```
        } else {
            return true;
        }
    }
}
```

- DefaultNIWSServerListFilter：该过滤器完全继承自 ZoneAffinity-ServerListFilter，是默认的 NIWS（Netflix Internal Web Service）过滤器。
- ServerListSubsetFilter：该过滤器也继承自 ZoneAffinityServer-ListFilter，它非常适用于拥有大规模服务器集群（上百或更多）的系统。因为它可以产生一个“区域感知”结果的子集列表，同时它还能够通过比较服务实例的通信失败数量和并发连接数来判定该服务是否健康来选择性地从服务实例列表中剔除那些相对不够健康的实例。该过滤器的实现主要分为以下三步：

  1. 获取“区域感知”的过滤结果，作为候选的服务实例清单。
  2. 从当前消费者维护的服务实例子集中剔除那些相对不够健康的实例（同时也将这些实例从候选清单中剔除，防止第三步的时候又被选入），不够健康的标准如下所示。

     a. 服务实例的并发连接数超过客户端配置的值，默认为 0，配置参数为 <clientName>.<nameSpace>.ServerListSubsetFilter.eliminationConnectionThresold。

     b. 服务实例的失败数超过客户端配置的值，默认为 0，配置参数为 <clientName>.<nameSpace>.ServerListSubsetFilter.eliminationFailureThresold。

     c. 如果按符合上面任一规则的服务实例剔除后，剔除比例小于客户端默认配置的百分比，默认为 0.1（10%），配置参数为<clientName>.<nameSpace>.ServerListSubsetFilter.forceEliminatePercent，那么就先对剩下的实例列表进行健康排序，再从最不健康的实例进行剔除，直到达到配置的剔除百分比。

  3. 在完成剔除后，清单已经少了至少 10%（默认值）的服务实例，最后通过随机的方式从候选清单中选出一批实例加入到清单中，以保持服务实例子集与原来的数量一致，而默认的实例子集数量为 20，其配置参数为<clientName>.<nameSpace>.ServerListSubsetFilter.size。

- ZonePreferenceServerListFilter：Spring Cloud 整合时新增的过滤器。若使用 Spring Cloud 整合 Eureka 和 Ribbon 时会默认使用该过滤器。它实现了通过配

置或者 Eureka 实例元数据的所属区域（Zone）来过滤出同区域的服务实例。如下面的源码所示，它的实现非常简单，首先通过父类 ZoneAffinityServerListFilter 的过滤器来获得“区域感知”的服务实例列表，然后遍历这个结果，取出根据消费者配置预设的区域 Zone 来进行过滤，如果过滤的结果是空就直接返回父类获取的结果，如果不为空就返回通过消费者配置的 Zone 过滤后的结果。

```
@Override
public List<Server> getFilteredListOfServers(List<Server> servers) {
    List<Server> output = super.getFilteredListOfServers(servers);
    if (this.zone != null && output.size() == servers.size()) {
        List<Server> local = new ArrayList<Server>();
        for (Server server : output) {
            if (this.zone.equalsIgnoreCase(server.getZone())) {
                local.add(server);
            }
        }
        if (!local.isEmpty()) {
            return local;
        }
    }
    return output;
}
```

### ZoneAwareLoadBalancer

ZoneAwareLoadBalancer 负载均衡器是对 DynamicServerListLoadBalancer 的扩展。在 DynamicServerListLoadBalancer 中，我们可以看到它并没有重写选择具体服务实例的 chooseServer 函数，所以它依然会采用在 BaseLoadBalancer 中实现的算法。使用 RoundRobinRule 规则，以线性轮询的方式来选择调用的服务实例，该算法实现简单并没有区域（Zone）的概念，所以它会把所有实例视为一个 Zone 下的节点来看待，这样就会周期性地产生跨区域（Zone）访问的情况，由于跨区域会产生更高的延迟，这些实例主要以防止区域性故障实现高可用为目的而不能作为常规访问的实例，所以在多区域部署的情况下会有一定的性能问题，而该负载均衡器则可以避免这样的问题。那么它是如何实现的呢？

首先，在 ZoneAwareLoadBalancer 中，我们可以发现，它并没有重写 setServersList，说明实现服务实例清单的更新主逻辑没有修改。但是我们可以发现它重写了这个函数 setServerListForZones(Map<String, List<Server>>zoneServersMap)。看到这里可能会有一些陌生，因为它并不是接口中定义的必备函数，所以我们不妨去父类 DynamicServerListLoadBalancer 中寻找一下该函数，我们可以找到下面的定义：

```
public void setServersList(List lsrv) {
    super.setServersList(lsrv);
    List<T> serverList = (List<T>) lsrv;
    Map<String, List<Server>> serversInZones = new HashMap<String, List<Server>>();
    ...
    setServerListForZones(serversInZones);
}

protected void setServerListForZones(Map<String, List<Server>> zoneServersMap) {
    LOGGER.debug("Setting server list for zones: {}", zoneServersMap);
    getLoadBalancerStats().updateZoneServerMapping(zoneServersMap);
}
```

setServerListForZones 函数的调用位于更新服务实例清单函数 setServersList 的最后，同时从其实现的内容来看，它在父类 DynamicServerListLoadBalancer 中的作用是根据按区域 Zone 分组的实例列表，为负载均衡器中的 LoadBalancerStats 对象创建 ZoneStats 并放入 Map zoneStatsMap 集合中，每一个区域 Zone 对应一个 ZoneStats，它用于存储每个 Zone 的一些状态和统计信息。

在 ZoneAwareLoadBalancer 中对 setServerListForZones 的重写如下：

```
protected void setServerListForZones(Map<String, List<Server>> zoneServersMap) {
    super.setServerListForZones(zoneServersMap);
    if (balancers == null) {
        balancers = new ConcurrentHashMap<String, BaseLoadBalancer>();
    }
    for (Map.Entry<String, List<Server>> entry: zoneServersMap.entrySet()) {
        String zone = entry.getKey().toLowerCase();
        getLoadBalancer(zone).setServersList(entry.getValue());
    }
    for (Map.Entry<String, BaseLoadBalancer> existingLBEntry: balancers.entrySet()) {
        if (!zoneServersMap.keySet().contains(existingLBEntry.getKey())) {
            existingLBEntry.getValue().setServersList(Collections.emptyList());
        }
    }
}
```

可以看到，在该实现中创建了一个 ConcurrentHashMap()类型的 balancers 对象，它将用来存储每个 Zone 区域对应的负载均衡器。而具体的负载均衡器的创建则是通过在下面的第一个循环中调用 getLoadBalancer 函数来完成，同时在创建负载均衡器的时候会创建它的规则（如果当前实现中没有 IRule 的实例，就创建一个 AvailabilityFilteringRule 规则；如果已经有具体实例，就克隆一个）。在创建完负载均衡器后又马上调用 setServersList 函数为其设置对应 Zone 区域的实例清单。而第二个循环则是对 Zone 区域中实例清单的检查，看看是否有 Zone 区域下已经没有实例了，是的话就将

balancers 中对应 Zone 区域的实例列表清空，该操作的作用是为了后续选择节点时，防止过时的 Zone 区域统计信息干扰具体实例的选择算法。

在了解了该负载均衡器是如何扩展服务实例清单的实现后，我们来具体看看它是如何挑选服务实例，来实现对区域的识别的：

```
public Server chooseServer(Object key) {
    if (!ENABLED.get() || getLoadBalancerStats().getAvailableZones().size() <= 1) {
        logger.debug("Zone aware logic disabled or there is only one zone");
        return super.chooseServer(key);
    }
    Server server = null;
    try {
        LoadBalancerStats lbStats = getLoadBalancerStats();
        Map<String, ZoneSnapshot> zoneSnapshot = ZoneAvoidanceRule.createSnapshot
(lbStats);
        logger.debug("Zone snapshots: {}", zoneSnapshot);
        ...
        Set<String> availableZones = ZoneAvoidanceRule.getAvailableZones
(zoneSnapshot, triggeringLoad.get(), triggeringBlackoutPercentage.get());
        logger.debug("Available zones: {}", availableZones);
        if (availableZones != null && availableZones.size() <
zoneSnapshot.keySet().size()) {
            String zone = ZoneAvoidanceRule.randomChooseZone(zoneSnapshot,
availableZones);
            logger.debug("Zone chosen: {}", zone);
            if (zone != null) {
                BaseLoadBalancer zoneLoadBalancer = getLoadBalancer(zone);
                server = zoneLoadBalancer.chooseServer(key);
            }
        }
    } catch (Throwable e) {
        logger.error("Unexpected exception when choosing server using zone aware
logic", e);
    }
    if (server != null) {
        return server;
    } else {
        logger.debug("Zone avoidance logic is not invoked.");
        return super.chooseServer(key);
    }
}
```

从源码中我们可以看到，只有当负载均衡器中维护的实例所属的 Zone 区域的个数大于 1 的时候才会执行这里的选择策略，否则还是将使用父类的实现。当 Zone 区域的个数大于 1 的时候，它的实现步骤如下所示。

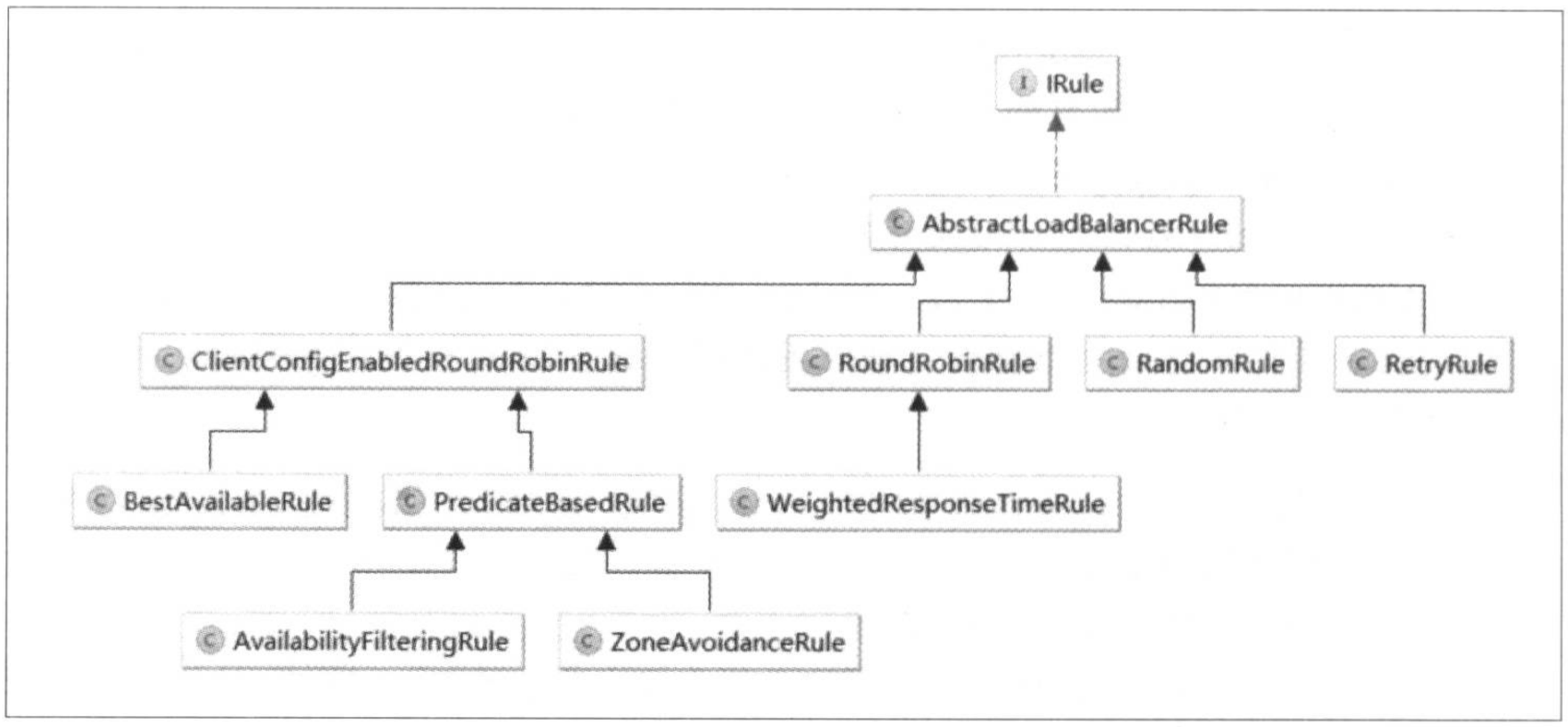
- 调用 ZoneAvoidanceRule 中的静态方法 createSnapshot(lbStats)，为当前负载均衡器中所有的 Zone 区域分别创建快照，保存在 Map zoneSnapshot 中，这些快照中的数据将用于后续的算法。
- 调用 ZoneAvoidanceRule 中的静态方法 getAvailableZones(zoneSnapshot, triggeringLoad.get(), triggeringBlackoutPercentage.get())，来获取可用的 Zone 区域集合，在该函数中会通过 Zone 区域快照中的统计数据来实现可用区的挑选。
  - 首先它会剔除符合这些规则的 Zone 区域：所属实例数为零的 Zone 区域；Zone 区域内实例的平均负载小于零，或者实例故障率（断路器断开次数/实例数）大于等于阈值（默认为 0.99999）。
  - 然后根据 Zone 区域的实例平均负载计算出最差的 Zone 区域，这里的最差指的是实例平均负载最高的 Zone 区域。
  - 如果在上面的过程中没有符合剔除要求的区域，同时实例最大平均负载小于阈值（默认为 20%），就直接返回所有 Zone 区域为可用区域。否则，从最坏 Zone 区域集合中随机选择一个，将它从可用 Zone 区域集合中剔除。
- 当获得的可用 Zone 区域集合不为空，并且个数小于 Zone 区域总数，就随机选择一个 Zone 区域。
- 在确定了某个 Zone 区域后，则获取了对应 Zone 区域的服务均衡器，并调用 chooseServer 来选择具体的服务实例，而在 chooseServer 中将使用 IRule 接口的 choose 函数来选择具体的服务实例。在这里，IRule 接口的实现会使用 ZoneAvoidanceRule 来挑选出具体的服务实例。

关于 ZoneAvoidanceRule 的策略以及其他一些还未提到的负载均衡策略，我们将在下一节做更加详细的解读。

### 负载均衡策略

通过上面的源码解读，我们已经对 Ribbon 实现的负载均衡器以及其中包含的服务实例过滤器、服务实例信息的存储对象、区域的信息快照等都有了深入的认识和理解，但是对于负载均衡器中的服务实例选择策略只是讲解了几个默认实现的内容，而对于 IRule 的其他实现还没有详细解读，下面我们来看看在 Ribbon 中都提供了哪些负载均衡的策略实现。

如下图所示，可以看到在 Ribbon 中实现了非常多的选择策略，其中也包含了我们在前面内容中提到过的 RoundRobinRule 和 ZoneAvoidanceRule。下面我们来详细解读一下 IRule 接口的各个实现。




### AbstractLoadBalancerRule

负载均衡策略的抽象类，在该抽象类中定义了负载均衡器 ILoadBalancer 对象，该对象能够在具体实现选择服务策略时，获取到一些负载均衡器中维护的信息来作为分配依据，并以此设计一些算法来实现针对特定场景的高效策略。

```
public abstract class AbstractLoadBalancerRule implements IRule, IClientConfigAware {

    private ILoadBalancer lb;

    @Override
    public void setLoadBalancer(ILoadBalancer lb){
        this.lb = lb;
    }

    @Override
    public ILoadBalancer getLoadBalancer(){
        return lb;
    }
}
```

### RandomRule

该策略实现了从服务实例清单中随机选择一个服务实例的功能。它的具体实现如下，可以看到 IRule 接口的 choose(Object key)函数实现，委托给了该类中的 choose(ILoadBalancer lb, Object key)，该方法增加了一个负载均衡器对象的参数。从具体的实现上看，它会使用传入的负载均衡器来获得可用实例列表 upList 和所有实例列表 allList，并通过 rand.nextInt(serverCount)函数来获取一个随机数，

并将该随机数作为 upList 的索引值来返回具体实例。同时，具体的选择逻辑在一个 while (server == null) 循环之内，而根据选择逻辑的实现，正常情况下每次选择都应该选出一个服务实例，如果出现死循环获取不到服务实例时，则很有可能存在并发的 Bug。

```
@Override
public Server choose(Object key) {
    return choose(getLoadBalancer(), key);
}

public Server choose(ILoadBalancer lb, Object key) {
    ...
    Server server = null;
    while (server == null) {
        if (Thread.interrupted()) {
            return null;
        }
        List<Server> upList = lb.getReachableServers();
        List<Server> allList = lb.getAllServers();
        int serverCount = allList.size();
        if (serverCount == 0) {
            return null;
        }
        int index = rand.nextInt(serverCount);
        server = upList.get(index);
        if (server == null) {
            Thread.yield();
            continue;
        }
        if (server.isAlive()) {
            return (server);
        }
        server = null;
        Thread.yield();
    }
    return server;
}
```

### RoundRobinRule

该策略实现了按照线性轮询的方式依次选择每个服务实例的功能。它的具体实现如下，其详细结构与 RandomRule 非常类似。除了循环条件不同外，就是从可用列表中获取所谓的逻辑不同。从循环条件中，我们可以看到增加了一个 count 计数变量，该变量会在每次循环之后累加，也就是说，如果一直选择不到 server 超过 10 次，那么就会结束尝试，并打印一个警告信息 No available alive servers after 10 tries from load balancer: ...。

而线性轮询的实现则是通过 AtomicInteger nextServerCyclicCounter 对象实现，每次进行实例选择时通过调用 incrementAndGetModulo 函数实现递增。

```
public Server choose(ILoadBalancer lb, Object key) {
    ...
    Server server = null;
    int count = 0;
    while (server == null && count++ < 10) {
        List<Server> reachableServers = lb.getReachableServers();
        List<Server> allServers = lb.getAllServers();
        int upCount = reachableServers.size();
        int serverCount = allServers.size();
        if ((upCount == 0) || (serverCount == 0)) {
            log.warn("No up servers available from load balancer: " + lb);
            return null;
        }
        int nextServerIndex = incrementAndGetModulo(serverCount);
        server = allServers.get(nextServerIndex);
        if (server == null) {
            Thread.yield();
            continue;
        }
        if (server.isAlive() && (server.isReadyToServe())) {
            return (server);
        }
        server = null;
    }
    if (count >= 10) {
        log.warn("No available alive servers after 10 tries from load balancer: "
                + lb);
    }
    return server;
}
```

### RetryRule

该策略实现了一个具备重试机制的实例选择功能。从下面的实现中我们可以看到，在其内部还定义了一个 IRule 对象，默认使用了 RoundRobinRule 实例。而在 choose 方法中则实现了对内部定义的策略进行反复尝试的策略，若期间能够选择到具体的服务实例就返回，若选择不到就根据设置的尝试结束时间为阈值（maxRetryMillis 参数定义的值 + choose 方法开始执行的时间戳），当超过该阈值后就返回 null。

```
public class RetryRule extends AbstractLoadBalancerRule {
    IRule subRule = new RoundRobinRule();
```

```
    long maxRetryMillis = 500;
    ...
    public Server choose(ILoadBalancer lb, Object key) {
        long requestTime = System.currentTimeMillis();
        long deadline = requestTime + maxRetryMillis;
        Server answer = null;
        answer = subRule.choose(key);
        if (((answer == null) || (!answer.isAlive()))
            && (System.currentTimeMillis() < deadline)) {
        InterruptTask task = new InterruptTask(deadline
                - System.currentTimeMillis());
        while (!Thread.interrupted()) {
            answer = subRule.choose(key);
            if (((answer == null) || (!answer.isAlive()))
                    && (System.currentTimeMillis() < deadline)) {
                Thread.yield();
            } else {
                break;
            }
        }
        task.cancel();
    }
        if ((answer == null) || (!answer.isAlive())) {
            return null;
        } else {
            return answer;
        }
    }
    ...
}
```

### WeightedResponseTimeRule

该策略是对 RoundRobinRule 的扩展，增加了根据实例的运行情况来计算权重，并根据权重来挑选实例，以达到更优的分配效果，它的实现主要有三个核心内容。

#### 定时任务

WeightedResponseTimeRule 策略在初始化的时候会通过 serverWeightTimer.schedule(new DynamicServerWeightTask(), 0, serverWeightTaskTimerInterval) 启动一个定时任务，用来为每个服务实例计算权重，该任务默认 30 秒执行一次。

```
class DynamicServerWeightTask extends TimerTask {
    public void run() {
        ServerWeight serverWeight = new ServerWeight();
```

```
            try {
                serverWeight.maintainWeights();
            } catch (Throwable t) {
                logger.error("Throwable caught while running DynamicServerWeightTask for
" + name, t);
            }
        }
    }
```

权重计算

在源码中我们可以轻松找到用于存储权重的对象 `List<Double> accumulatedWeights = new ArrayList<Double>()`，该 List 中每个权重值所处的位置对应了负载均衡器维护的服务实例清单中所有实例在清单中的位置。

维护实例权重的计算过程通过 `maintainWeights` 函数实现，具体如下面的代码所示：

```
public void maintainWeights() {
    ILoadBalancer lb = getLoadBalancer();
    ...
    try {
        logger.info("Weight adjusting job started");
        AbstractLoadBalancer nlb = (AbstractLoadBalancer) lb;
        LoadBalancerStats stats = nlb.getLoadBalancerStats();
        ...
        // 计算所有实例的平均响应时间的总和: totalResponseTime
        double totalResponseTime = 0;
        for (Server server : nlb.getAllServers()) {
            // 如果服务实例的状态快照不在缓存中，那么这里会进行自动加载
            ServerStats ss = stats.getSingleServerStat(server);
            totalResponseTime += ss.getResponseTimeAvg();
        }
        // 逐个计算每个实例的权重: weightSoFar + totalResponseTime - 实例的平均响应时间
        Double weightSoFar = 0.0;
        List<Double> finalWeights = new ArrayList<Double>();
        for (Server server : nlb.getAllServers()) {
            ServerStats ss = stats.getSingleServerStat(server);
            double weight = totalResponseTime - ss.getResponseTimeAvg();
            weightSoFar += weight;
            finalWeights.add(weightSoFar);
        }
        setWeights(finalWeights);
    } catch (Throwable t) {
        logger.error("Exception while dynamically calculating server weights", t);
    } finally {
```

```
        serverWeightAssignmentInProgress.set(false);
    }
}
```

该函数的实现主要分为两个步骤：

- 根据 LoadBalancerStats 中记录的每个实例的统计信息，累加所有实例的平均响应时间，得到总平均响应时间 totalResponseTime，该值会用于后续的计算。
- 为负载均衡器中维护的实例清单逐个计算权重（从第一个开始），计算规则为 weightSoFar+totalResponseTime－实例的平均响应时间，其中 weightSoFar 初始化为零，并且每计算好一个权重需要累加到 weightSoFar 上供下一次计算使用。

举个简单的例子来理解这个计算过程，假设有 4 个实例 A、B、C、D，它们的平均响应时间为 10、40、80、100，所以总响应时间是 10 + 40 + 80 + 100 = 230，每个实例的权重为总响应时间与实例自身的平均响应时间的差的累积所得，所以实例 A、B、C、D 的权重分别如下所示。

- 实例 A：230 - 10 = 220
- 实例 B：220 +（230 - 40）= 410
- 实例 C：410 +（230 - 80）= 560
- 实例 D：560 +（230 - 100）= 690

需要注意的是，这里的权重值只是表示了各实例权重区间的上限，并非某个实例的优先级，所以不是数值越大被选中的概率就越大。那么什么是权重区间呢？以上面例子的计算结果为例，它实际上是为这 4 个实例构建了 4 个不同的区间，每个实例的区间下限是上一个实例的区间上限，而每个实例的区间上限则是我们上面计算并存储于 List accumulatedWeights 中的权重值，其中第一个实例的下限默认为零。所以，根据上面示例的权重计算结果，我们可以得到每个实例的权重区间。

- 实例 A：[0, 220]
- 实例 B：(220, 410]
- 实例 C：(410, 560]
- 实例 D：(560，690)

不难发现，实际上每个区间的宽度就是：总的平均响应时间 - 实例的平均响应时间，所以实例的平均响应时间越短、权重区间的宽度越大，而权重区间的宽度越大被选中的概率就越高。可能很多读者会问，这些区间边界的开闭是如何确定的呢？为什么不那么规则？下面我们会通过实例选择算法的解读来解释。

实例选择

WeightedResponseTimeRule 选择实例的实现与之前介绍的算法结构类似，下面是它主体的算法（省略了循环体和一些判断等处理）：

```
public Server choose(ILoadBalancer lb, Object key) {
    ...
        List<Double> currentWeights = accumulatedWeights;
        ...
        List<Server> allList = lb.getAllServers();
        int serverCount = allList.size();
        if (serverCount == 0) {
            return null;
        }
        int serverIndex = 0;
        // 获取最后一个实例的权重
        double maxTotalWeight = currentWeights.size() == 0 ? 0 : currentWeights.get
(currentWeights.size() - 1);
        if (maxTotalWeight < 0.001d) {
            // 如果最后一个实例的权重值小于0.001，则采用父类实现的线性轮询的策略
            server =  super.choose(getLoadBalancer(), key);
            if(server == null) {
                return server;
            }
        } else {
            // 如果最后一个实例的权重值大于等于0.001，就产生一个[0, maxTotalWeight)的随机数
            double randomWeight = random.nextDouble() * maxTotalWeight;
            int n = 0;
            for (Double d : currentWeights) {
            // 遍历维护的权重清单，若权重大于等于随机得到的数值，就选择这个实例
                if (d >= randomWeight) {
                    serverIndex = n;
                    break;
                } else {
                    n++;
                }
            }
            server = allList.get(serverIndex);
        }
    ...
    return server;
}
```

从源码中我们可以看到，选择实例的核心过程就两步：

- 生成一个[0，最大权重值)区间内的随机数。
- 遍历权重列表，比较权重值与随机数的大小，如果权重值大于等于随机数，就拿当前权重列表的索引值去服务实例列表中获取具体的实例。这就是在上一节中提到的服务实例会根据权重区间挑选的原理，而权重区间边界的开闭原则根据算法，正常每个区间为(x，y]的形式，但是第一个实例和最后一个实例为什么不同呢？由于随机数的最小取值可以为 0，所以第一个实例的下限是闭区间，同时随机数的最大值取不到最大权重值，所以最后一个实例的上限是开区间。

若继续以上面的数据为例进行服务实例的选择，则该方法会从[0，690)区间中选出一个随机数，比如选出的随机数为 230，由于该值位于第二个区间，所以此时就会选择实例 B 来进行请求。

### ClientConfigEnabledRoundRobinRule

该策略较为特殊，我们一般不直接使用它。因为它本身并没有实现什么特殊的处理逻辑，正如下面的源码所示，在它的内部定义了一个 RoundRobinRule 策略，而 choose 函数的实现也正是使用了 RoundRobinRule 的线性轮询机制，所以它实现的功能实际上与 RoundRobinRule 相同，那么定义它有什么特殊的用处呢？

虽然我们不会直接使用该策略，但是通过继承该策略，默认的 choose 就实现了线性轮询机制，在子类中做一些高级策略时通常有可能会存在一些无法实施的情况，那么就可以用父类的实现作为备选。在后文中我们将继续介绍的高级策略均是基于 ClientConfigEnabledRoundRobinRule 的扩展。

```
public class ClientConfigEnabledRoundRobinRule extends AbstractLoadBalancerRule {

    RoundRobinRule roundRobinRule = new RoundRobinRule();
    ...
    @Override
    public Server choose(Object key) {
        if (roundRobinRule != null) {
            return roundRobinRule.choose(key);
        } else {
            throw new IllegalArgumentException(
                    "This class has not been initialized with the RoundRobinRule class");
        }
    }
}
```

### BestAvailableRule

该策略继承自 ClientConfigEnabledRoundRobinRule，在实现中它注入了负载

均衡器的统计对象 LoadBalancerStats，同时在具体的 choose 算法中利用 LoadBalancerStats 保存的实例统计信息来选择满足要求的实例。从如下源码中我们可以看到，它通过遍历负载均衡器中维护的所有服务实例，会过滤掉故障的实例，并找出并发请求数最小的一个，所以该策略的特性是可选出最空闲的实例。

```
public Server choose(Object key) {
    if (loadBalancerStats == null) {
        return super.choose(key);
    }
    List<Server> serverList = getLoadBalancer().getAllServers();
    int minimalConcurrentConnections = Integer.MAX_VALUE;
    long currentTime = System.currentTimeMillis();
    Server chosen = null;
    for (Server server: serverList) {
        ServerStats serverStats = loadBalancerStats.getSingleServerStat(server);
        if (!serverStats.isCircuitBreakerTripped(currentTime)) {
            int concurrentConnections = serverStats.getActiveRequestsCount(currentTime);
            if (concurrentConnections < minimalConcurrentConnections) {
                minimalConcurrentConnections = concurrentConnections;
                chosen = server;
            }
        }
    }
    if (chosen == null) {
        return super.choose(key);
    } else {
        return chosen;
    }
}
```

同时，由于该算法的核心依据是统计对象 loadBalancerStats，当其为空的时候，该策略是无法执行的。所以从源码中我们可以看到，当 loadBalancerStats 为空的时候，它会采用父类的线性轮询策略，正如我们在介绍 ClientConfigEnabledRoundRobinRule 时那样，它的子类在无法满足实现高级策略的时候，可以使用线性轮询策略的特性。后面将要介绍的策略因为也都继承自 ClientConfigEnabledRoundRobinRule，所以它们都会具有这样的特性。

### PredicateBasedRule

这是一个抽象策略，它也继承了 ClientConfigEnabledRoundRobinRule，从其命名中可以猜出这是一个基于 Predicate 实现的策略，Predicate 是 Google Guava Collection 工具对集合进行过滤的条件接口。

如下面的源码所示，它定义了一个抽象函数 getPredicate 来获取 AbstractServerPredicate 对象的实现，而在 choose 函数中，通过 AbstractServerPredicate 的 chooseRoundRobinAfterFiltering 函数来选出具体的服务实例。从该函数的命名我们也大致能猜出它的基础逻辑：先通过子类中实现的 Predicate 逻辑来过滤一部分服务实例，然后再以线性轮询的方式从过滤后的实例清单中选出一个。

```
public abstract class PredicateBasedRule extends ClientConfigEnabledRoundRobinRule {

    public abstract AbstractServerPredicate getPredicate();

    @Override
    public Server choose(Object key) {
        ILoadBalancer lb = getLoadBalancer();
        Optional<Server> server = getPredicate().chooseRoundRobinAfterFiltering
(lb.getAllServers(), key);
        if (server.isPresent()) {
            return server.get();
        } else {
            return null;
        }
    }
}
```

通过下面 AbstractServerPredicate 的源码片段，可以证实我们上面所做的猜测。在上面 choose 函数中调用的 chooseRoundRobinAfterFiltering 方法先通过内部定义的 getEligibleServers 函数来获取备选的实例清单（实现了过滤），如果返回的清单为空，则用 Optional.absent() 来表示不存在，反之则以线性轮询的方式从备选清单中获取一个实例。

```
public abstract class AbstractServerPredicate implements Predicate<PredicateKey> {

    ...

    public Optional<Server> chooseRoundRobinAfterFiltering(List<Server> servers,
Object loadBalancerKey) {
        List<Server> eligible = getEligibleServers(servers, loadBalancerKey);
        if (eligible.size() == 0) {
            return Optional.absent();
        }
        return Optional.of(eligible.get(nextIndex.getAndIncrement() % eligible.
size()));
    }

    public List<Server> getEligibleServers(List<Server> servers, Object
loadBalancerKey) {
```

```
        if (loadBalancerKey == null) {
            return ImmutableList.copyOf(Iterables.filter(servers, this.getServer-
OnlyPredicate()));
        } else {
            List<Server> results = Lists.newArrayList();
            for (Server server: servers) {
                if (this.apply(new PredicateKey(loadBalancerKey, server))) {
                    results.add(server);
                }
            }
            return results;
        }
    }
}
```

在了解了整体逻辑之后，我们来详细看看实现过滤功能的 getEligibleServers 函数。从源码上看，它的实现结构简单清晰，通过遍历服务清单，使用 this.apply 方法来判断实例是否需要保留，如果是就添加到结果列表中。

可能到这里，不熟悉 Google Guava Collections 集合工具的读者会感到困惑，这个 apply 在 AbstractServerPredicate 中找不到它的定义，那么它是如何实现过滤的呢？实际上，AbstractServerPredicate 实现了 com.google.common.base.Predicate 接口，而 apply 方法是该接口中的定义，主要用来实现过滤条件的判断逻辑，它输入的参数则是过滤条件需要用到的一些信息（比如源码中的 new PredicateKey (loadBalancerKey, server)），它传入了关于实例的统计信息和负载均衡器的选择算法传递过来的 key）。既然在 AbstractServerPredicate 中我们未能找到 apply 的实现，所以这里的 chooseRoundRobinAfterFiltering 函数只是定义了一个模板策略："先过滤清单，再轮询选择"。对于如何过滤，需要我们在 AbstractServerPredicate 的子类中实现 apply 方法来确定具体的过滤策略。

后面我们将要介绍的两个策略就是基于此抽象策略实现，只是它们使用了不同的 Predicate 实现来完成过滤逻辑以达到不同的实例选择效果。

> Google Guava Collections 是一个对 Java Collections Framework 增强和扩展的开源项目。虽然 Java Collections Framework 已经能够满足我们大多数情况下使用集合的要求，但是当遇到一些特殊的情况时我们的代码会比较冗长且容易出错。Guava Collections 可以帮助我们让集合操作代码更为简短精练并大大增强代码的可读性。

### AvailabilityFilteringRule

该策略继承自上面介绍的抽象策略 PredicateBasedRule，所以它也继承了"先过

滤清单，再轮询选择”的基本处理逻辑，其中过滤条件使用了 AvailabilityPredicate：

```
public class AvailabilityPredicate extends AbstractServerPredicate {

    ...

    public boolean apply(@Nullable PredicateKey input) {
        LoadBalancerStats stats = getLBStats();
        if (stats == null) {
            return true;
        }
        return !shouldSkipServer(stats.getSingleServerStat(input.getServer()));
    }

    private boolean shouldSkipServer(ServerStats stats) {
        if ((CIRCUIT_BREAKER_FILTERING.get() && stats.isCircuitBreakerTripped())
                || stats.getActiveRequestsCount() >= activeConnectionsLimit.get()) {
            return true;
        }
        return false;
    }
}
```

从上述源码中，我们可以知道它的主要过滤逻辑位于 shouldSkipServer 方法中，它主要判断服务实例的两项内容：

- 是否故障，即断路器是否生效已断开。
- 实例的并发请求数大于阈值，默认值为 $2^{32}-1$，该配置可通过参数<clientName>.<nameSpace>.ActiveConnectionsLimit 来修改。

这两项内容中只要有一个满足 apply 就返回 false（代表该节点可能存在故障或负载过高），都不满足就返回 true。

在该策略中，除了实现了上面的过滤方法之外，对于 choose 的策略也做了一些改进优化，所以父类的实现对于它来说只是一个备用选项，其具体实现如下所示：

```
public Server choose(Object key) {
    int count = 0;
    Server server = roundRobinRule.choose(key);
    while (count++ <= 10) {
        if (predicate.apply(new PredicateKey(server))) {
            return server;
        }
        server = roundRobinRule.choose(key);
    }
    return super.choose(key);
}
```

可以看到，它并没有像在父类中那样，先遍历所有的节点进行过滤，然后在过滤后的集合中选择实例。而是先以线性的方式选择一个实例，接着用过滤条件来判断该实例是否满足要求，若满足就直接使用该实例，若不满足要求就再选择下一个实例，并检查是否满足要求，如此循环进行，当这个过程重复了 10 次还是没有找到符合要求的实例，就采用父类的实现方案。

简单地说，该策略通过线性抽样的方式直接尝试寻找可用且较空闲的实例来使用，优化了父类每次都要遍历所有实例的开销。

### ZoneAvoidanceRule

该策略我们在介绍负载均衡器 ZoneAwareLoadBalancer 时已经提到过，它也是 PredicateBasedRule 的具体实现类。在之前的介绍中主要针对 ZoneAvoidanceRule 中用于选择 Zone 区域策略的一些静态函数，比如 createSnapshot、getAvailableZones。在这里我们将详细看看 ZoneAvoidanceRule 作为服务实例过滤条件的实现原理。从下面 ZoneAvoidanceRule 的源码片段中可以看到，它使用了 CompositePredicate 来进行服务实例清单的过滤。这是一个组合过滤条件，在其构造函数中，它以 ZoneAvoidancePredicate 为主过滤条件，AvailabilityPredicate 为次过滤条件初始化了组合过滤条件的实例。

```
public class ZoneAvoidanceRule extends PredicateBasedRule {

    ...
    private CompositePredicate compositePredicate;

    public ZoneAvoidanceRule() {
        super();
        ZoneAvoidancePredicate zonePredicate = new ZoneAvoidancePredicate(this);
        AvailabilityPredicate availabilityPredicate = new AvailabilityPredicate(this);
        compositePredicate = createCompositePredicate(zonePredicate,
availabilityPredicate);
    }
    ...
}
```

ZoneAvoidanceRule 在实现的时候并没有像 AvailabilityFilteringRule 那样重写 choose 函数来优化，所以它完全遵循了父类的过滤主逻辑："先过滤清单，再轮询选择"。其中过滤清单的条件就是我们上面提到的以 ZoneAvoidancePredicate 为主过滤条件、AvailabilityPredicate 为次过滤条件的组合过滤条件 CompositePredicate。从 CompositePredicate 的源码片段中，我们可以看到它定义了一个主过滤条件 AbstractServerPredicate delegate 以及一组次过滤条件列表 List

fallbacks，所以它的次过滤列表是可以拥有多个的，并且由于它采用了 List 存储所以次过滤条件是按顺序执行的。

```
public class CompositePredicate extends AbstractServerPredicate {

    private AbstractServerPredicate delegate;
    private List<AbstractServerPredicate> fallbacks = Lists.newArrayList();

    private int minimalFilteredServers = 1;
    private float minimalFilteredPercentage = 0;

    @Override
    public List<Server> getEligibleServers(List<Server> servers, Object
loadBalancerKey) {
        List<Server> result = super.getEligibleServers(servers, loadBalancerKey);
        Iterator<AbstractServerPredicate> i = fallbacks.iterator();
        while (!(result.size() >= minimalFilteredServers && result.size() > (int)
(servers.size() * minimalFilteredPercentage))
                && i.hasNext()) {
            AbstractServerPredicate predicate = i.next();
            result = predicate.getEligibleServers(servers, loadBalancerKey);
        }
        return result;
    }

}
```

在获取过滤结果的实现函数 getEligibleServers 中，它的处理逻辑如下所示。

- 使用主过滤条件对所有实例过滤并返回过滤后的实例清单。
- 依次使用次过滤条件列表中的过滤条件对主过滤条件的结果进行过滤。
- 每次过滤之后（包括主过滤条件和次过滤条件），都需要判断下面两个条件，只要有一个符合就不再进行过滤，将当前结果返回供线性轮询算法选择：
  - 过滤后的实例总数 >= 最小过滤实例数（minimalFilteredServers，默认为 1）。
  - 过滤后的实例比例 > 最小过滤百分比（minimalFilteredPercentage，默认为 0）。

## 配置详解

通过上一节对 Spring Cloud Ribbon 的源码分析，相信读者对于 Ribbon 的几个重要接口都已经有所了解。下面，我们将详细介绍 Ribbon 在使用时的各种配置方式。

## 自动化配置

由于 Ribbon 中定义的每一个接口都有多种不同的策略实现，同时这些接口之间又有一定的依赖关系，这使得第一次使用 Ribbon 的开发者很难上手，不知道如何选择具体的实现策略以及如何组织它们的关系。Spring Cloud Ribbon 中的自动化配置恰恰能够解决这样的痛点，在引入 Spring Cloud Ribbon 的依赖之后，就能够自动化构建下面这些接口的实现。

- IClientConfig：Ribbon 的客户端配置，默认采用 com.netflix.client.config.DefaultClientConfigImpl 实现。
- IRule：Ribbon 的负载均衡策略，默认采用 com.netflix.loadbalancer.ZoneAvoidanceRule 实现，该策略能够在多区域环境下选出最佳区域的实例进行访问。
- IPing：Ribbon 的实例检查策略，默认采用 com.netflix.loadbalancer.NoOpPing 实现，该检查策略是一个特殊的实现，实际上它并不会检查实例是否可用，而是始终返回 true，默认认为所有服务实例都是可用的。
- ServerList<Server>：服务实例清单的维护机制，默认采用 com.netflix.loadbalancer.ConfigurationBasedServerList 实现。
- ServerListFilter<Server>：服务实例清单过滤机制，默认采用 org.springframework.cloud.netflix.ribbon.ZonePreferenceServerListFilter 实现，该策略能够优先过滤出与请求调用方处于同区域的服务实例。
- ILoadBalancer：负载均衡器，默认采用 com.netflix.loadbalancer.ZoneAwareLoadBalancer 实现，它具备了区域感知的能力。

上面这些自动化配置内容仅在没有引入 Spring Cloud Eureka 等服务治理框架时如此，在同时引入 Eureka 和 Ribbon 依赖时，自动化配置会有一些不同，后续我们会做详细的介绍。

通过自动化配置的实现，我们可以轻松地实现客户端负载均衡。同时，针对一些个性化需求，我们也可以方便地替换上面的这些默认实现。只需在 Spring Boot 应用中创建对应的实现实例就能覆盖这些默认的配置实现。比如下面的配置内容，由于创建了 PingUrl 实例，所以默认的 NoOpPing 就不会被创建。

```
@Configuration
public class MyRibbonConfiguration {
    @Bean
    public IPing ribbonPing(IClientConfig config) {
        return new PingUrl();
    }
}
```

另外，也可以通过使用@RibbonClient 注解来实现更细粒度的客户端配置，比如下面的代码实现了为 hello-service 服务使用 HelloServiceConfiguration 中的配置。

```
@Configuration
@RibbonClient(name = "hello-service", configuration = HelloServiceConfiguration.
class)
public class RibbonConfiguration {
}
```

## Camden 版本对 RibbonClient 配置的优化

上面我们介绍了在 Brixton 版本中对 RibbonClient 的 IPing、IRule 等接口实现进行个性化定制的方法，主要通过独立创建一个 Configuration 类来定义 IPing、IRule 等接口的具体实现 Bean，然后在创建 RibbonClient 时指定要使用的具体 Configuration 类来覆盖自动化配置的默认实现。虽然这种方式已经能够实现个性化的定义，但是当有大量这类配置的时候，对于各个 RibbonClient 的指定配置信息都将分散在这些配置类的注解定义中，这使得管理和修改都变得非常不方便。所以，在 Camden 版本中，Spring Cloud Ribbon 对 RibbonClient 定义个性化配置的方法做了进一步优化。可以直接通过 <clientName>.ribbon.<key>=<value>的形式进行配置。比如我们要实现与上面例子一样的配置（将 hello-service 服务客户端的 IPing 接口实现替换为 PingUrl），只需在 application.properties 配置中增加下面的内容即可：

```
hello-service.ribbon.NFLoadBalancerPingClassName=com.netflix.loadbalancer.PingUrl
```

其中 hello-service 为服务名，NFLoadBalancerPingClassName 参数用来指定具体的 IPing 接口实现类。在 Camden 版本中，Spring Cloud Ribbon 新增了一个 org.springframework.cloud.netflix.ribbon.PropertiesFactory 类来动态地为 RibbonClient 创建这些接口实现。

```
public class PropertiesFactory {
    @Autowired
    private Environment environment;

    private Map<Class, String> classToProperty = new HashMap<>();

    public PropertiesFactory() {
        classToProperty.put(ILoadBalancer.class, "NFLoadBalancerClassName");
        classToProperty.put(IPing.class, "NFLoadBalancerPingClassName");
        classToProperty.put(IRule.class, "NFLoadBalancerRuleClassName");
        classToProperty.put(ServerList.class, "NIWSServerListClassName");
        classToProperty.put(ServerListFilter.class, "NIWSServerListFilterClassName");
    }
```

```
    public boolean isSet(Class clazz, String name) {
        return StringUtils.hasText(getClassName(clazz, name));
    }

    public String getClassName(Class clazz, String name) {
        if (this.classToProperty.containsKey(clazz)) {
            String classNameProperty = this.classToProperty.get(clazz);
            String className = environment.getProperty(name + "." + NAMESPACE + "."
+ classNameProperty);
            return className;
        }
        return null;
    }

    @SuppressWarnings("unchecked")
    public <C> C get(Class<C> clazz, IClientConfig config, String name) {
        String className = getClassName(clazz, name);
        if (StringUtils.hasText(className)) {
            try {
                Class<?> toInstantiate = Class.forName(className);
                return (C) instantiateWithConfig(toInstantiate, config);
            } catch (ClassNotFoundException e) {
                throw new IllegalArgumentException("Unknown class to load
"+className+" for class " + clazz + " named " + name);
            }
        }
        return null;
    }
}
```

从上述源码定义中可以看到，除了 NFLoadBalancerPingClassName 参数之外，还提供了其他几个接口的动态配置实现，具体如下所述。

- NFLoadBalancerClassName：配置 ILoadBalancer 接口的实现。
- NFLoadBalancerPingClassName：配置 IPing 接口的实现。
- NFLoadBalancerRuleClassName：配置 IRule 接口的实现。
- NIWSServerListClassName：配置 ServerList 接口的实现。
- NIWSServerListFilterClassName：配置 ServerListFilter 接口的实现。

所以，在 Camden 版本中我们可以通过配置的方式，更加方便地为 RibbonClient 指定 ILoadBalancer、IPing、IRule、ServerList 和 ServerListFilter 的定制化实现。

## 参数配置

对于 Ribbon 的参数配置通常有两种方式：全局配置以及指定客户端配置。

- 全局配置的方式很简单，只需使用 ribbon.<key>=<value>格式进行配置即可。其中，<key>代表了 Ribbon 客户端配置的参数名，<value>则代表了对应参数的值。比如，我们可以像下面这样全局配置 Ribbon 创建连接的超时时间：

```
ribbon.ConnectTimeout=250
```

  全局配置可以作为默认值进行设置，当指定客户端配置了相应 key 的值时，将覆盖全局配置的内容。
- 指定客户端的配置方式采用<client>.ribbon.<key>=<value>的格式进行配置。其中，<key>和<value>的含义同全局配置相同，而<client>代表了客户端的名称，如上文中我们在@RibbonClient 中指定的名称，也可以将它理解为是一个服务名。为了方便理解这种配置方式，我们举一个具体的例子：假设，有一个服务消费者通过 RestTemplate 来访问 hello-service 服务的/hello 接口，这时我们会这样调用 restTemplate.getForEntity("http://hello-service/hello", String.class).getBody();。如果没有服务治理框架的帮助，我们需要为该客户端指定具体的实例清单，可以指定服务名来做详细的配置，具体如下：

```
hello-service.ribbon.listOfServers=localhost:8001,localhost:8002, localhost:8003
```

对于 Ribbon 参数的 key 以及 value 类型的定义，可以通过查看 com.netflix.client.config.CommonClientConfigKey 类获得更为详细的配置内容，在本书中不进行详细介绍。

## 与 Eureka 结合

当在 Spring Cloud 的应用中同时引入 Spring Cloud Ribbon 和 Spring Cloud Eureka 依赖时，会触发 Eureka 中实现的对 Ribbon 的自动化配置。这时 ServerList 的维护机制实现将被 com.netflix.niws.loadbalancer.DiscoveryEnabledNIWSServerList 的实例所覆盖，该实现会将服务清单列表交给 Eureka 的服务治理机制来进行维护；IPing 的实现将被 com.netflix.niws.loadbalancer.NIWSDiscoveryPing 的实例所覆盖，该实现也将实例检查的任务交给了服务治理框架来进行维护。默认情况下，用于获取实例请求的 ServerList 接口实现将采用 Spring Cloud Eureka 中封装的 org.springframework.cloud.netflix.ribbon.eureka.DomainExtractingServerList，其目的是为了让实例维护策略更加通用，所以将使用物理元数据来进行负载均衡，而不是使用原生的 AWS AMI 元数据。

在与 Spring Cloud Eureka 结合使用的时候，我们的配置将会变得更加简单。不再需要通过类似 hello-service.ribbon.listOfServers 的参数来指定具体的服务实例清单，因为 Eureka 将会为我们维护所有服务的实例清单。而对于 Ribbon 的参数配置，我们依然可以采用之前的两种配置方式来实现，而指定客户端的配置方式可以直接使用 Eureka 中的服务名作为<client>来完成针对各个微服务的个性化配置。

此外，由于 Spring Cloud Ribbon 默认实现了区域亲和策略，所以，我们可以通过 Eureka 实例的元数据配置来实现区域化的实例配置方案。比如，可以将处于不同机房的实例配置成不同的区域值，以作为跨区域的容错机制实现。而实现的方式非常简单，只需在服务实例的元数据中增加 zone 参数来指定自己所在的区域，比如：

```
eureka.instance.metadataMap.zone=shanghai
```

在 Spring Cloud Ribbon 与 Spring Cloud Eureka 结合的工程中，我们也可以通过参数配置的方式来禁用 Eureka 对 Ribbon 服务实例的维护实现。只需在配置文件中加入如下参数，这时我们对于服务实例的维护就又将回归到使用<client>.ribbon.listOfServers 参数配置的方式来实现了。

```
ribbon.eureka.enabled=false
```

## 重试机制

由于 Spring Cloud Eureka 实现的服务治理机制强调了 CAP 原理中的 AP，即可用性与可靠性，它与 ZooKeeper 这类强调 CP（一致性、可靠性）的服务治理框架最大的区别就是，Eureka 为了实现更高的服务可用性，牺牲了一定的一致性，在极端情况下它宁愿接受故障实例也不要丢掉“健康”实例，比如，当服务注册中心的网络发生故障断开时，由于所有的服务实例无法维持续约心跳，在强调 AP 的服务治理中将会把所有服务实例都剔除掉，而 Eureka 则会因为超过 85%的实例丢失心跳而会触发保护机制，注册中心将会保留此时的所有节点，以实现服务间依然可以进行互相调用的场景，即使其中有部分故障节点，但这样做可以继续保障大多数的服务正常消费。

由于 Spring Cloud Eureka 在可用性与一致性上的取舍，不论是由于触发了保护机制还是服务剔除的延迟，引起服务调用到故障实例的时候，我们还是希望能够增强对这类问题的容错。所以，我们在实现服务调用的时候通常会加入一些重试机制。在目前我们使用的 Brixton 版本中，对于重试机制的实现需要我们自己来扩展完成。而从 Camden SR2 版本开始，Spring Cloud 整合了 Spring Retry 来增强 RestTemplate 的重试能力，对于开发者来说只需通过简单的配置，原来那些通过 RestTemplate 实现的服务访问就会自动根据配置来实现重试策略。