目录

本电子书全部文章，均节选自《疯狂 Spring Cloud 微服务架构实战》一书，作者杨恩雄。
作者博客：https://my.oschina.net/JavaLaw/blog

本电子书全部文章，均节选自《疯狂 Spring Cloud 微服务架构实战》一书，作者杨恩雄。
作者博客：https://my.oschina.net/JavaLaw/blog

1 Spring Cloud 概述

本章要点

- 传统应用的问题
- 微服务与 Spring Cloud
- 本书介绍

本章将会简述 Spring Cloud 的功能，描述什么是 Spring Cloud，它能为我们带来什么，为后面学习该框架的知识打下理论的基础。

1.1 传统的应用

1.1.1 单体应用

在此之前，笔者所在公司开发 Java 程序，大都使用 Struts、Spring、Hibernate（MyBatis）等技术框架，每一个项目都会发布一个单体应用。例如开发一个进销存系统，将会开发一个 war 包部署到 Tomcat 中，每一次需要开发新的模块或添加新功能时，都会在原来的基础上不断的添加。若干年后，这个 war 包不断的膨胀，程序员在进行调试时，服务器也可能需要启动半天，维护这个系统的效率极为低下。这样一个 war 包，涵盖了库存、销售、会员、报表等模块，如图 1-1。

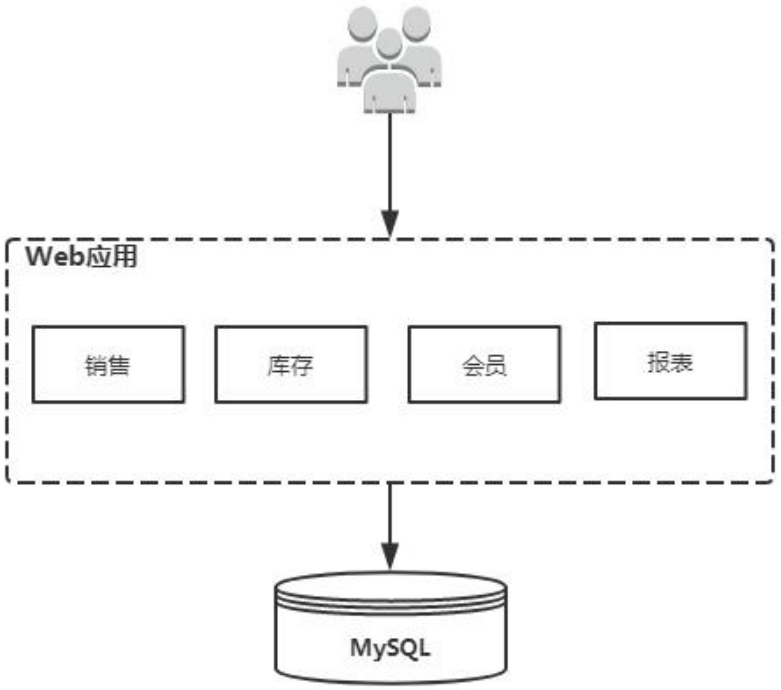

图 1-1 单体应用

这样的单体应用隐患非常多，任何的一个 bug，都有可能导致整个系统宕机。笔者印象最深刻的是，曾经有一客户在高峰期，导出一张销售明细报表（数据量较大），最终造成整个系统瘫痪，前台的销售人员无法售卖。维护这样一个系统，不仅效率极低，而且充满风险，项目组的各个成员惶惶不可终日，我们需要本质上的改变。

1.1.2 架构演进

针对以上的单体应用的问题，我们参考 SOA 架构，将各个模块划分独立的服务模块（war），并且使用了数据库的读写分离，架构如图 1-2。

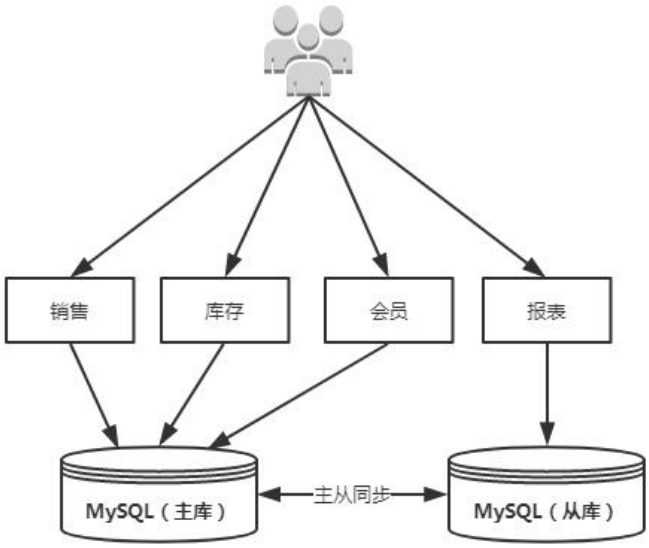

图 1-2 架构演进

各个模块之间会存在相互调用的依赖关系，例如销售模块会调用会员模块的接口，为了减少各个模块之间的耦合，我们加入了企业服务总线（ESB），各模块与 ESB 之间的架构如图 1-3 所示。

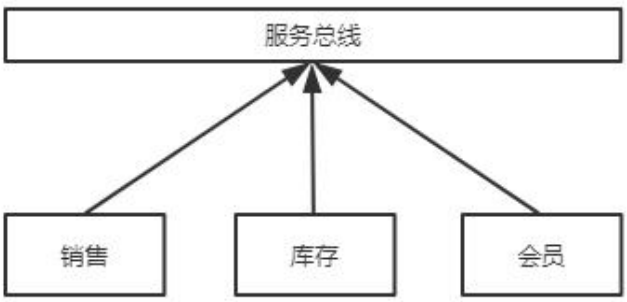

图 1-3 ESB

加入 ESB 后，各个模块将服务发布到 ESB 中，它们与 ESB 之间使用 SOAP 协议进行通信。图 1-2 与图 1-3 的架构实现后，整个系统的性能有了明显的提升，各个模块的耦合度也降低了。运行了一段日子后，又出现了新的问题，由于销售终端数量的增多，销售模块明

显超过其承受能力，为了保证销售前端的正常运行，我们使用了 Nginx 做负载均衡，请见图 1-4。

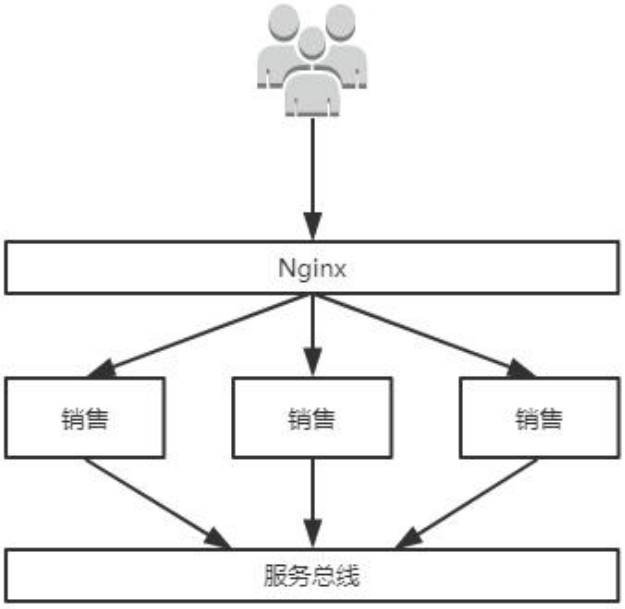

图 1-4 使用 Nginx

随着销售模块的增多，带来了许多问题，例如管理这些模块，对于运维工程师来说，是一项艰巨的任务，一旦销售模块有所修改，他们将通宵达旦进行升级。另外，企业服务总线也有可能成为性能的瓶颈，虽然目前仍未出现该问题，但我们需要未雨绸缪。

1.1.3 架构要求

从前面的架构演进可知，应用中的每一个点，都有可能成为系统的问题点。随着互联网应用的普及，在大数据、高并发的环境下，我们的系统架构需要面对更为严苛的挑战，我们需要一套新的架构，它起码能满足以下要求：

- 高性能：这是应用程序的基本要求。
- 独立性：其中一个模块出现 bug 或者其他问题，不可以影响其他模块或者整个应用。
- 容易扩展：应用中的每一个节点，都可以根据实际需要进行扩展。
- 便于管理：对于各个模块的资源，可以轻松进行管理、升级，减少维护成本。
- 状态监控与警报：对整个应用程序进行监控，当某一个节点出现问题时，能及时发出警报。

为了能解决遇到的问题、达到以上的架构要求，我们开始研究 Spring Cloud。

1.2 微服务与 Spring Cloud

1.2.1 什么是微服务

微服务一词来源 Martin Fowler 的“Microservices”一文，微服务是一种架构风格，将单体应用划分为小型的服务单元，微服务之间使用 HTTP 的 API 进行资源访问与操作。

在对单体应用的划分上，微服务与前面的 SOA 架构有点类似，但是 SOA 架构侧重于将每个单体应用的服务集成到 ESB 上，而微服务做得更加彻底，强调将整个模块变成服务组件，微服务对模块的划分粒度可能会更细。以我们前面的销售、会员模块为例，在 SOA 架构中，只需要将相应的服务发布到 ESB 容器就可以了，而在微服务架构中，这两个模块本身，将会变为一个或多个的服务组件。SOA 架构与微服务架构，请见图 1-5 与图 1-6。

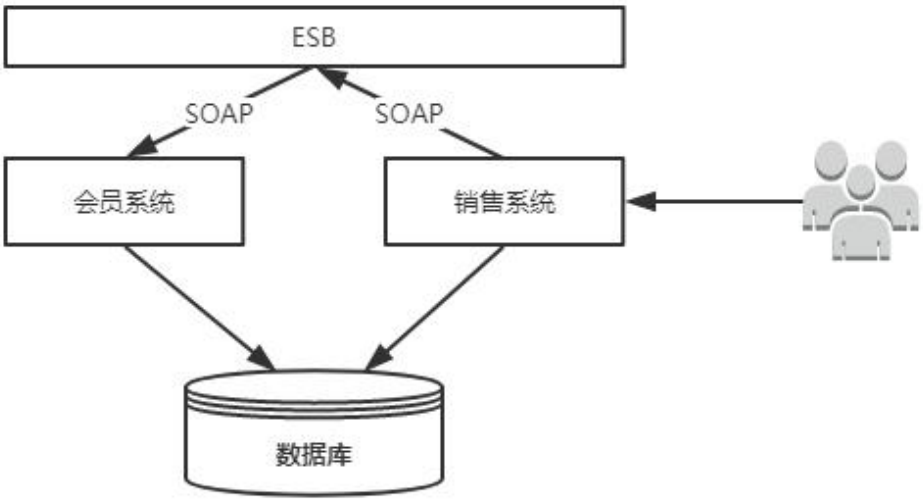

图 1-5 SOA 架构

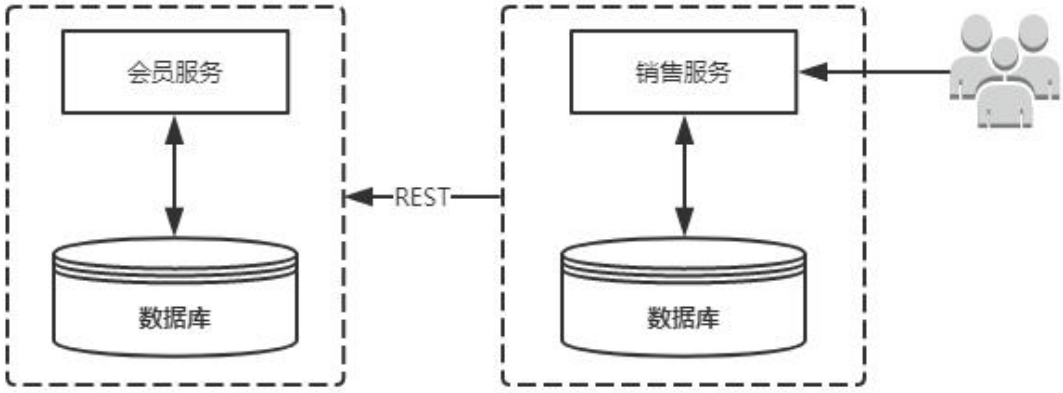

图 1-6 微服务架构

在微服务的架构上，Martin Fowler 的文章肯定了 Netflix 的贡献，接下来，我们了解一下 Netflix OSS。

1.2.2 关于 Netflix OSS

Netflix 是一个互联网影片提供商，在几年前，Netflix 公司成立了自己的开源中心，名称为 Netflix Open Source Software Center，简称 Netflix OSS。这个开源组织专注于大数据、云计算方面的技术，提供了多个开源框架，这些框架包括大数据工具、构建工具、基于云平台的服务工具等。Netflix 所提供的这些框架，很好的遵循微服务所推崇的理念，实现了去中心化的服务管理、服务容错等机制。

1.2.3 Spring Cloud 与 Netflix

Spring Cloud 并不是一个具体的框架，大家可以把它理解为一个工具箱，它提供的各类工具，可以帮助我们快速的构建分布式系统。

Spring Cloud 的各个项目基于 Spring Boot，将 Netflix 的多个框架进行封装，并且通过自动配置的方式将这些框架绑定到 Spring 的环境中，从而简化了这些框架的使用。由于 Spring Boot 的简便，使得我们在使用 Spring Cloud 时，很容易的将 Netflix 各个框架整合进我们的项目中。Spring Cloud 下的“Spring Cloud Netflix”模块，主要封装了 Netflix 的以下项目：

- Eureka：基于 REST 服务的分布式中间件，主要用于服务管理。
- Hystrix：容错框架，通过添加延迟阀值以及容错的逻辑，来帮助我们控制分布式系统间组件的交互。
- Feign：一个 REST 客户端，目的是为了简化 Web Service 客户端的开发
- Ribbon：负载均衡框架，在微服务集群中为各个客户端的通信提供支持，它主要实现中间层应用程序的负载均衡
- Zuul：为微服务集群提供过代理、过滤、路由等功能。

1.2.4 Spring Cloud 的主要模块

除了 Spring Cloud Netflix 模块外，Spring Cloud 还包括以下几个重要的模块：

- Spring Cloud Config：为分布式系统提供了配置服务器和配置客户端，通过对它们的配置，可以很好的管理集群中的配置文件。
- Spring Cloud Sleuth：服务跟踪框架，可以与 Zipkin、Apache HTrace 和 ELK 等数据分析、服务跟踪系统进行整合，为服务跟踪、解决问题提供了便利。
- Spring Cloud Stream：用于构建消息驱动微服务的框架，该框架在 Spring Boot 的基础上，整合了“Spring Integration”来连接消息代理中间件。
- Spring Cloud Bus：连接 RabbitMQ、Kafka 等消息代理的集群消息总线。

1.3 关于本书

1.3.1 下载本书的软件及源码

读者可以到以下的地址下载本书的软件以及源码：http://pan.baidu.com/s/1nvbZSeP，该地址下有一个“Spring Cloud”目录，下面有一个 codes.zip 的压缩包，下载解压后即可拿到 codes 目录。本书所的软件，基本上都可以在 soft 目录下找到。

如果读者无法从以上的渠道获取本书的软件及源码，欢迎发邮件与笔者联系，邮箱地址：yangenxiong@163.com ，也可以直接访问笔者博客直接交流：https://my.oschina.net/JavaLaw/blog。

1.3.2 导入本书的案例

在 Eclipse 中，选择导入已存在的 Maven 项目，再选择 codes 下的具体目录即可。由于每个案例都会包含多个 Maven 项目，因此建议以小节为单位导入，导入界面请见图 1-7。

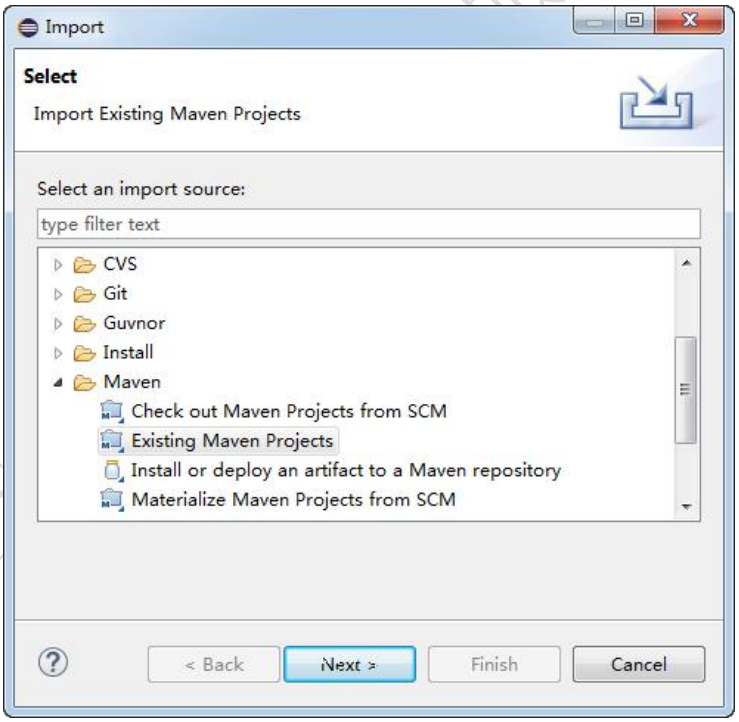

图 1-7 导入本章案例

在导入案例前，请先按照第 2 章，把开发环境搭建好。

1.4 本章小结

本章的 1.1 小节，对传统的单体应用、SOA 架构做了一个简单的总结，在此过程中分析我们所遇到的问题。在 1.2 小节，简单介绍了微服务与 Spring Cloud。接下来，我们正式开始讲述本书的知识点。

2 开发环境搭建

本章要点

- 安装与配置 Maven
- 安装 Eclipse
- 本书涉及的技术版本
- Spring Boot 使用

工欲善其事，必先利其器。在讲述本书的技术内容前，先将开发环境搭建好，本书所涉及基础环境将在本章准备，包括 Eclipse、Maven 等。如果读者对 Maven、Eclipse、Spring Boot 等项目较为熟悉，可以直接跳过本章的安装过程。

笔者建议读者在查阅本书过程中，使用与本书相同的工具以及版本。本章使用的 Java 版本为 1.8，图 2-1 为“java –version”命令的输出，Java 安装与配置较为简单，本书不再赘述。

图 2-1 Java 版本

注：本书全部的案例均在 Windows7 下开发和运行。

2.1 安装与配置 Maven

2.1.1 关于 Maven

Maven 是 Apache 下的一个开源项目，用于项目的构建。使用 Maven 可以对项目的依赖包进行管理，支持构建脚本的继承，对于一些模块（子项目）较多的项目来说，Maven 是更好的选择，子项目可以继承父项目的构建脚本，减少了构建脚本的冗余。

除此之外，Maven 本身的插件机制让其更加强大和灵活，使用者可以配置各种 Maven 插件来完成自己的事，如果感觉官方或者第三方提供的 Maven 插件不够用，还可以自行编写符合自己要求的 Maven 插件。Maven 为使用者提供了一个统一的依赖仓库，各种开源项目的发布包可以在上面找到，在一间公司或者一个项目组内部，甚至可以搭建私有的 Maven 仓库，将自己项目的包放到私有仓库中，供其他项目组或者开发者使用。

Maven 的众多特性中，最为重要的是它对依赖包的管理，Maven 将项目所使用的依赖包的信息放到 pom.xml 的 dependencies 节点。例如我们需要使用 spring-core 模块的 jar 包，只需在 pom.xml 配置该模块的依赖信息，Maven 会自动将 spring-beans 等模块引入到我们项目的环境变量中。Spring Cloud 项目基于 Spring Boot 搭建，正是由于依赖管理的特性，使得 Maven 与 Spring Boot 更加相得益彰，可以让我们更快速的搭建一个可用的开发环境。

2.1.2 下载与安装 Maven

本书所使用的 Maven 版本为 3.5，可以到 Maven 官方网站下载：http://maven.apache.org/。下载并解压后得到 apache-maven-3.5.0 目录，将主目录下的的 bin 目录加入到系统的环境变量中，如图 2-2 所示。

图 2-2 修改环境变量

配置完后，打开 cmd 命令行，输入“mvn –v”，可以看到输出的 Maven 版本信息。Maven 下载的依赖包会存放到本地仓库中，默认路径为：C:\Users\用户名\.m2\repository。

2.1.3 配置远程仓库

如果不进行仓库配置，默认情况下，会到 apache 官方的仓库下载依赖包，由于 Apache 官方的仓库位于国外，下载速度较慢，会降低开发效率，笔者建议使用国内的 Maven 仓库或者搭建自己的私服，本书重点不是 Maven，因此直接使用了由阿里云提供的 Maven 仓库。修改 apache-maven-3.5.0/conf 目录下的 setting.xml，在 mirrors 节点下加入以下配置：

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

配置完后，以后在使用过程中，Maven 会先到阿里云的仓库中下载依赖包。另外，需要注意的是，本书的大部分案例，都没有使用 Maven 的继承特性，每一个 Maven 项目都可以独立引入。

2.2 安装 Eclipse

2.2.1 Eclipse 版本

本书使用 Eclipse 作为开发工具，使用版本为 Luna（4.4），大家可以从以下的地址得到该版本的 Eclipse：http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/lunasr2，也可以在本书所附的 soft 目录下找到该版本的 Eclipse。目前 Eclipse 已经发展到 4.7 版本，本书主要在 Eclipse 中使用 Maven 插件。

2.2.2 在 Eclipse 配置 Maven

Luna 版本的 Eclipse 自带了 Maven 插件，默认使用的是 Maven3.2，由于我们前面安装的是 Maven3.5 版本，因此需要在 Eclipse 中指定 Maven 版本以及配置文件。指定 Maven 的配置如图 2-3 所示，指定配置文件如图 2-4 所示。

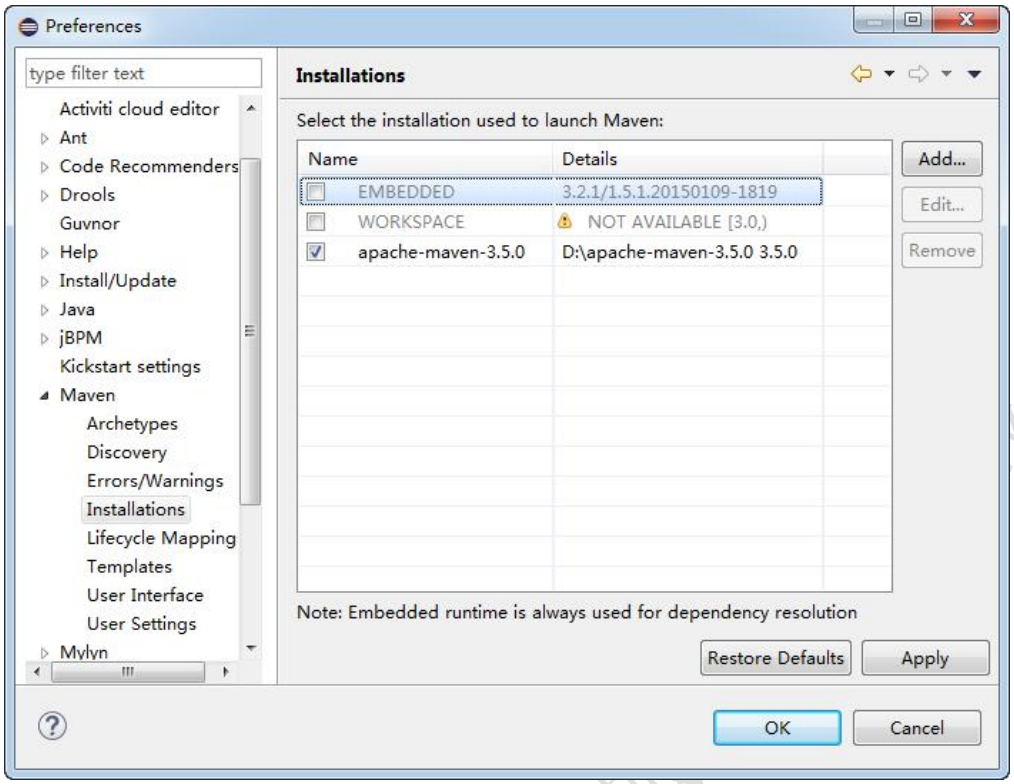

图 2-3 Eclipse 指定 Maven 版本

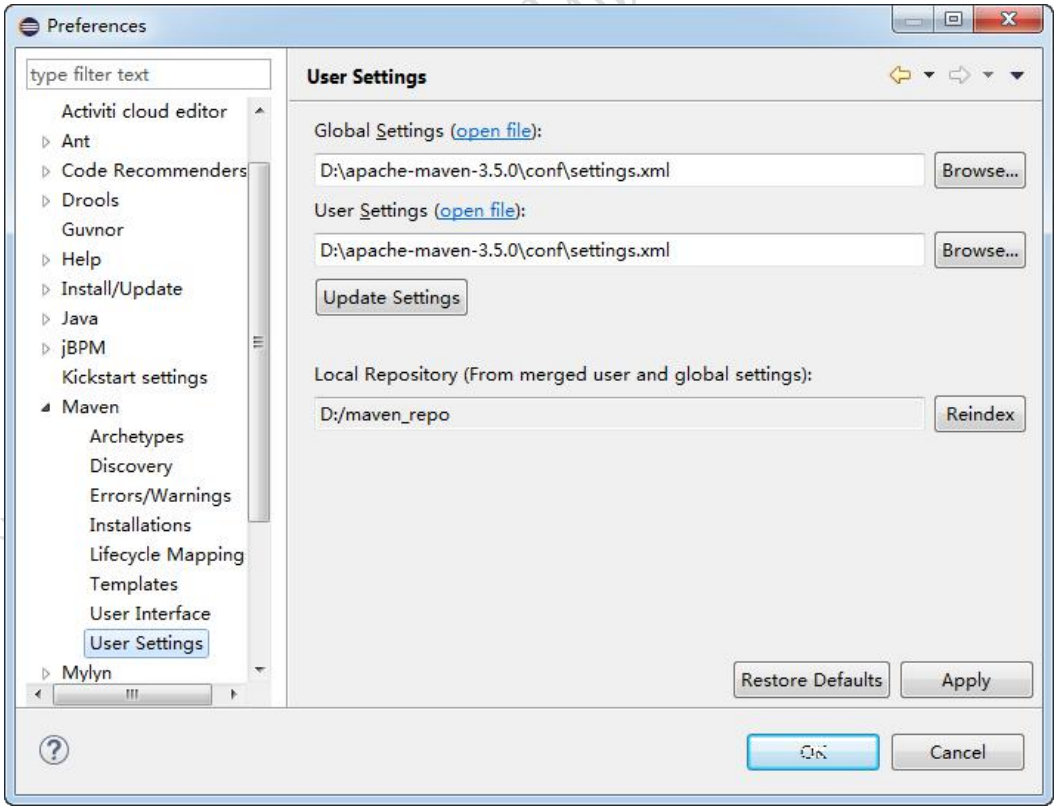

图 2-4 指定 Maven 配置文件

注意：本书的案例，如无特别说明均以 Maven 项目的形式导入。

3 Spring Boot 简介与配置

本章要点

- Spring Boot 简介
- Spring Boot 使用

3.1 Spring Boot

Spring Cloud 基于 Spring Boot 搭建，本小节将对 Spring Boot 作一个大致的讲解，读者知道 Spring Boot 作用即可。

3.1.1 Spring Boot 简介

开发一个全新的项目，需要先进行开发环境的搭建，例如要确定技术框架以及版本，还要考虑各个框架之间的版本兼容问题，完成这些繁琐的工作后，还要对新项目进行配置，测试能否正常运行，最后才将搭建好的环境提交给项目组的其他成员使用。经常出现的情形是，表面上已经成功运行，但部分项目组成员仍然无法运行，项目初期浪费大量的时间做这些工作，几乎每个项目都会投入部分工作量来做这些固定的事情。

受 Ruby On Rails、Node.js 等技术的影响，JavaEE 领域需要一种更为简便的开发方式，来取代这些繁琐的项目搭建工作。在此背景下，Spring 推出了 Spring Boot 项目，该项目可以让使用者更快速的搭建项目，使用者可以更专注、快速的投入到业务系统开发中。系统配置、基础代码、项目依赖的 jar 包，甚至是开发时所用到的应用服务器等，Spring Boot 已经帮我们准备好，只要在建立项目时，使用构建工具加入相应的 Spring Boot 依赖包，项目即可运行，使用者无需关心版本兼容等问题。

Spring Boot 支持 Maven 和 Gradle 这两款构建工具。Gradle 使用 Groovy 语言进行构建脚本的编写，与 Maven、Ant 等构建工具有良好的兼容性。鉴于笔者使用 Maven 较多，因此本书使用 Maven 作为项目构建工具。笔者成书时，Spring Boot 最新的正式版本为 1.5.4，要求 Maven 版本为 3.2 或以上。

3.1.2 新建 Maven 项目

在新建菜单中选择新建“Maven Project”，填写的项目信息如图 2-5 所示。

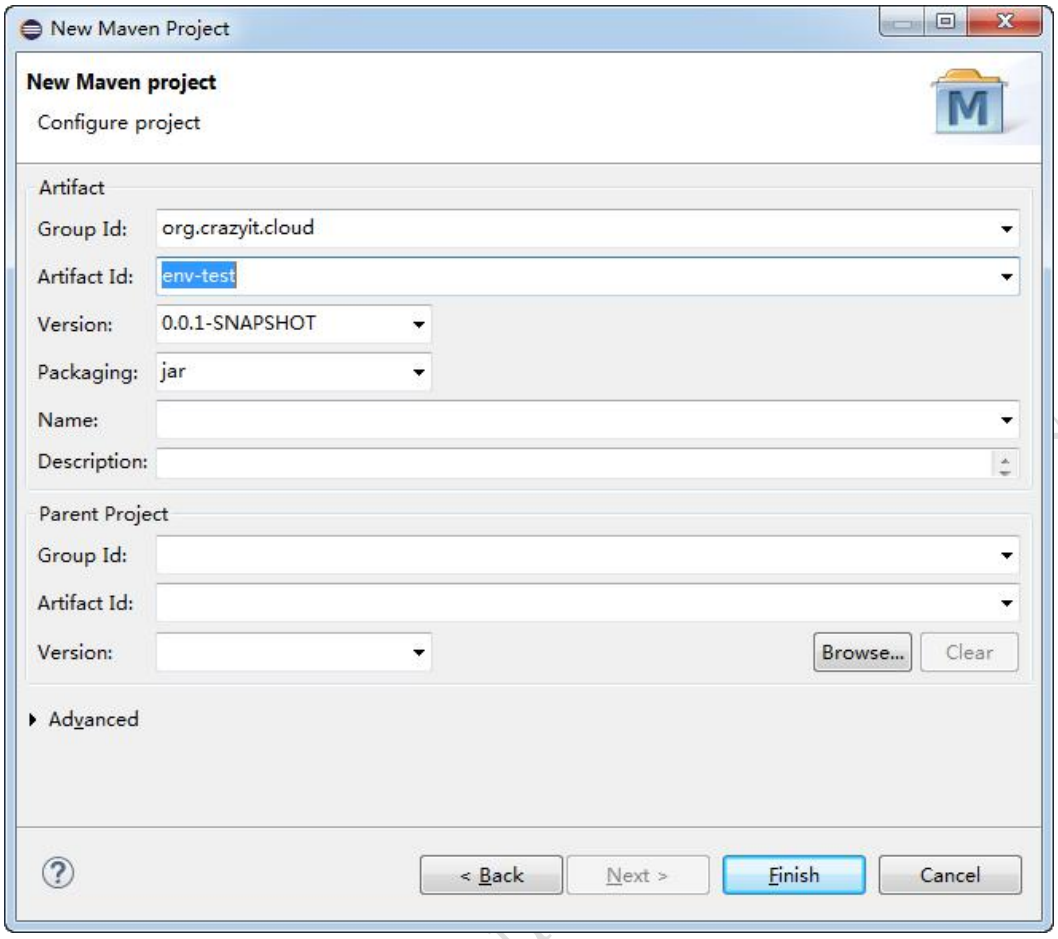

图 2-5 新建 Maven 项目

为了测试项目的可用性，加入 Spring Boot 的 web 启动模块，让该项目具有 Web 容器的功能，pom.xml 文件内容如代码清单 2-1 所示。

代码清单 2-1：codes\02\env-test\pom.xml

```
<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>org.crazyit.cloud</groupId>
    <artifactId>env-test</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <version>1.5.4.RELEASE</version>
        </dependency>
    </dependencies>
</project>
```

配置完依赖后，该依赖会自动帮我们的项目加上其他的 Spring 模块以及所依赖的第三方包，例如 spring-core、sprin-beans、spring-mvc 等，除了这些模块外，还加入了嵌入式

的 Tomcat。

3.1.3 编写启动类

加入了依赖后，只需要编写一个简单的启动类，即可启动 Web 服务，启动类如代码清单 2-2 所示。

代码清单 2-2：codes\02\env-test\src\main\java\org\crazyit\cloud\MyApplication.java

```
@SpringBootApplication
public class MyApplication {

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

MyApplication 类使用了@SpringBootApplication 注解，该注解声明了该类是一个 Spring Boot 应用，该注解具有“@SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan”等注解的功能。直接运行 MyApplication 的 main 方法，看到以下输出信息后，证明成功启动：

```
2017-08-02 20:53:05.327 INFO 1976 --- [ main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2017-08-02 20:53:05.530 INFO 1976 --- [ main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup
2017-08-02 20:53:05.878 INFO 1976 --- [ main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-08-02 20:53:05.885 INFO 1976 --- [ main] org.crazyit.cloud.MyApplication : Started MyApplication in 5.758 seconds (JVM running for 6.426)
```

根据输出信息可知，启动的 Tomcat 端口为 8080，打开浏览器访问：http://localhost:8080，可以看到错误页面，表示应用已经成功启动。

3.1.4 编写控制器

在前面小节加入的 spring-boot-starter-web 模块，默认集成了 SpringMVC，因此只需要编写一个 Controller，即可实现一个最简单的 HelloWord 程序，代码清单 2-3 为控制器。

代码清单 2-3：codes\02\env-test\src\main\java\org\crazyit\cloud\MyController.java

```
@Controller
public class MyController {

    @GetMapping("/hello")
    @ResponseBody
    public String hello() {
        return "Hello World";
    }
}
```

代码清单 2-3 中使用了@Controller 注解来修饰 MyController，由于启动类中使用了

@SpringBootApplication 注解，该注解含有@ComponentScan 的功能，因此@Controller 会被扫描并注册。在 hello 方法中使用了@GetMapping 与@ResponseBody 注解，声明 hello 方法的访问地址以及返回内容。重新运行启动类，打开浏览器并访问以下地址：http://localhost:8080/hello，可以看到控制器的返回。

3.1.5 发布 REST WebService

Spring MVC 支持直接发布 REST 风格的 WebService，新建测试的对象 Person，如代码清单 2-4 所示。

代码清单 2-4：codes\02\env-test\src\main\java\org\crazyit\cloud\Person.java

```
public class Person {

    private Integer id;

    private String name;

    private Integer age;

    ...省略 setter 和 getter 方法
}
```

修改控制器类，修改后如代码清单 2-5。

代码清单 2-5：codes\02\env-test\src\main\java\org\crazyit\cloud\MyController.java

```
@RestController
public class MyController {

    @GetMapping("/hello")
    @ResponseBody
    public String hello() {
        return "Hello World";
    }

    @RequestMapping(value = "/person/{personId}", method = RequestMethod.GET, produces =
MediaType.APPLICATION_JSON_VALUE)
    public Person findPerson(@PathVariable("personId") Integer personId) {
        Person p = new Person();
        p.setId(personId);
        p.setName("Crazyit");
        p.setAge(30);
        return p;
    }
}
```

MyController 类中，将原来的@Controller 注解修改为@RestController，新建 findPerson 方法，该方法将会根据参数 id 来创建一个 Person 实例并返回，访问该方法将会得到 JSON 字符串。运行启动类，在浏览器中输入：http://localhost:8080/person/1，可看到接口返回以下 JSON 字符串：

```
{"id":1,"name":"Crazyit","age":30}
```

调用 REST 服务的方式有很多，此部分内容将在后面章节中讲述。

3.2 Spring Boot 配置文件

Spring Cloud 基于 Spring Boot 构建，很多模块的配置均放在 Spring Boot 的配置文件中，因此有必要了解一下 Spring Boot 的配置文件规则，为学习后面的章节打下基础。

3.2.1 默认配置文件

Spring Boot 会按顺序读取各种配置，例如命令行参数、系统参数等，本章只讲述配置文件的参数读取。默认情况下，Spring Boot 会按顺序到以下目录读取 application.properties 或者 application.yml 文件：

- 项目根目录的 config 目录。
- 项目根目录。
- 项目 classpath 下的 config 目录。
- 项目 classpath 根目录。

如对以上描述有疑问，可参看图 2-6。

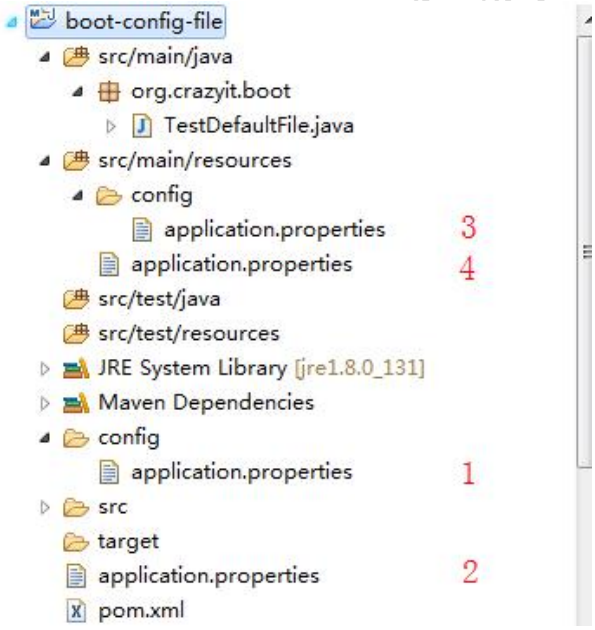

图 2-6 配置文件读取顺序

图 2-6 中的数字为文件的读取顺序，本小节使用的 boot-config-file 项目依赖了 spring-boot-starter-web 项目，为 pom.xml 加入以下依赖：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>1.5.4.RELEASE</version>
    </dependency>
</dependencies>
```

3.2.2 指定配置文件位置

如果想自己指定配置文件，可以在 Spring 容器的启动命令中加入参数，例子如代码清单 2-6 所示。

代码清单 2-6：codes\02\boot-config-file\src\main\java\org\crazyit\boot\TestDefaultFile.java

```
ConfigurableApplicationContext context = new SpringApplicationBuilder(
        TestDefaultFile.class)
        .properties(
                "spring.config.location=classpath:/test-folder/my-config.properties")
        .run(args);
```

TestDefaultFile 类，在使用 SpringApplicationBuilder 时，配置了 spring.config.location 属性来设定需要读取的配置文件。

3.2.3 yml 文件

YAML 语言使用一种方便的格式的来进行数据配置，通过配置分层、缩进，在很大程度上增强了配置文件的可读性，使用 YAML 语言的配置文件以“.yml”作为后缀。代码清单 2-7 为一份 yml 配置文件。

代码清单 2-7：codes\02\boot-config-file\src\main\resources\my-config.yml

```
jdbc:
  user:
    root
  passwd:
    123456
  driver:
    com.mysql.jdbc.Driver
```

在此，需要注意的是，每一行配置的缩进要使用空格，不要使用 tab 键进行缩进。代码清单 2-7 对应的 properties 文件内容如下：

```
jdbc.user=root
jdbc.passwd=123456
jdbc.driver=com.mysql.jdbc.Driver
```

3.2.4 运行时指定 profiles 配置

如果在不同的环境下激活不同的配置，可以使用 profiles，代码清单 2-8 中配置了两个 profiles。

代码清单 2-8：codes\02\boot-config-file\src\main\resources\test-profiles.yml

```
spring:
  profiles: mysql
jdbc:
  driver:
    com.mysql.jdbc.Driver
---
spring:
```

```
  profiles: oracle
jdbc:
  driver:
    oracle.jdbc.driver.OracleDriver
```

定义了 mysql 与 oracle 两个 profiles，profiels 间使用“---”进行分隔，在 Spring 容器启动时，使用 spring.profiles.active 来指定激活哪个 profiles，如代码清单 2-9 所示。

代码清单 2-9：codes\02\boot-config-file\src\main\java\org\crazyit\boot\TestProfiles.java

```
        ConfigurableApplicationContext context = new SpringApplicationBuilder(
                TestProfiles.class)
                .properties(
                        "spring.config.location=classpath:/test-profiles.yml")
                .properties("spring.profiles.active=oracle").run(args);
```

对 Spring Boot 的配置文件有一定了解后，对后面章节 Spring Cloud 的配置内容就不会陌生。

3.2.5 热部署

每次修改 Java 后，都需要重新运行 Main 方法才能生效，这样的会降低开发效果，我们可以使用 Spring Boot 提供的开发工具来实现热部署，为项目加上以下依赖：

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

当 Java 文件修改后，容器会重新加载本项目的 Java 类。

3.3 小结

本文主要讲述了本书基础环境的搭建，读者主要掌握 Maven 的使用，本书的案例几乎都是 Maven 项目。Spring Cloud 项目以 Spring Boot 作为基础进行构建，本书的大部分案例也是基于 Spring Boot，本章对 Spring Boot 作了大致的讲解，并且配合一个 Hello World 例子来演示 Spring Boot 的便捷，学习完本章后，读者知道 Spring Boot 的大致功能，即可达到目标。

4 微服务发布与调用

要点

- 认识 Eureka 框架
- 运行 Eureka 服务器
- 发布微服务
- 调用微服务

本章将讲述 Spring Cloud 中 Eureka 的使用，包括在 Eureka 服务器上发布、调用微服务，Eureka 的配置以及 Eureka 集群等内容。

4.1 Eureka 介绍

Spring Cloud 集成了 Netflix OSS 的多个项目，形成了 spring-cloud-netflix 项目，该项目包含了多个子模块，这些子模块对集成的 Netflix 旗下框架进行了封装，本小节将讲述其中一个较为重要的服务管理框架：Eureka。

4.1.1 关于 Eureka

Eureka 提供基于 REST 的服务，在集群中主要用于服务管理。Eureka 提供了基于 Java 语言的客户端组件，客户端组件实现了负载均衡的功能，为业务组件的集群部署创造了条件。使用该框架，可以将业务组件注册到 Eureka 容器中，进行集群部署，Eureka 提供的服务调用功能，可以发布容器中的服务并进行调用。

4.1.2 Eureka 架构

一个简单的 Eureka 集群，需要有一个 Eureka 服务器、若干个服务提供者。我们可以将业务组件注册到 Eureka 服务器中，其他客户端组件可以向服务器获取服务并且进行远程调用。图 3-1 为 Eureka 的架构图。

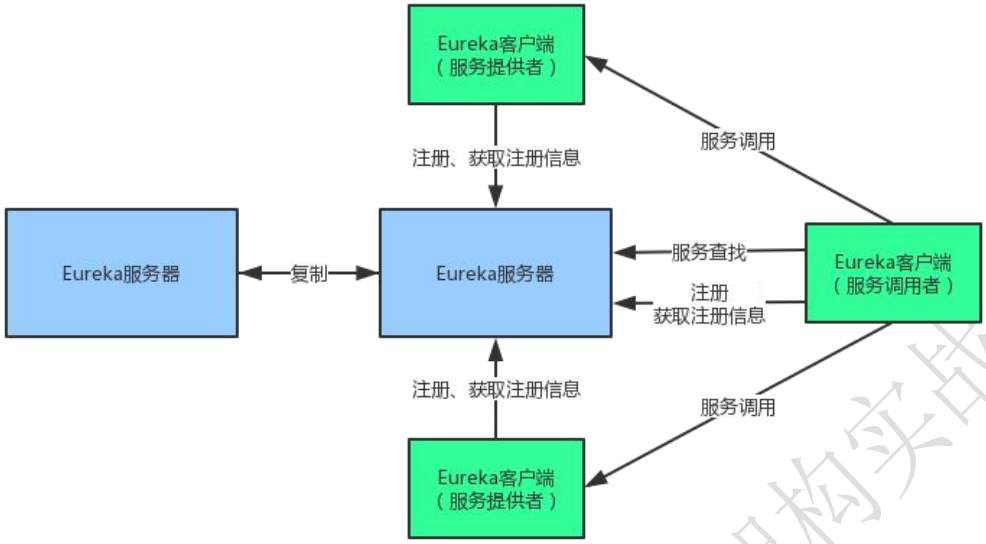

图 3-1 Eureka 架构图

图 3-1 中有两个服务器，服务器支持集群部署，每个服务器也可以作为对方服务器的客户端进行相互注册与复制。图 3-1 的三个 Eureka 客户端，两个用于发布服务，其中一个用于调用服务。不管服务器还是客户端，都可以部署多个实例，如此一来，就很容易构建高可用的服务集群。

4.1.3 服务器端

对于注册到服务器端的服务组件，Eureka 服务器并没有提供后台的存储，这些注册的服务实例被保存在内存的注册中心，它们通过心跳来保持其最新状态，这些操作都可以在内存中完成。客户端存在着相同的机制，同样在内存中保存了注册表信息，这样的机制提升了 Eureka 组件的性能，每次服务的请求都不必经过服务器端的注册中心。

4.1.4 服务提供者

作为 Eureka 客户端存在的服务提供者，主要进行以下工作：第一、向服务器注册服务；第二、发送心跳给服务器；第三、向服务器端获取注册列表。当客户端注册到服务器时，它将会提供一些关于它自己的信息给服务器端，例如自己的主机、端口、健康检测连接等。

4.1.5 服务调用者

对于发布到 Eureka 服务器的服务，使用调用者可对其进行服务查找与调用，服务调用者也是作为客户端存在，但其职责主要是发现与调用服务。在实际情况中，有可能出现本身既是服务提供者，也是服务调用者的情况，例如传统的企业应用三层架构中，服务层会调用数据访问层的接口进行数据操作，它本身也会提供服务给控制层使用。

本小节对 Eureka 作了介绍，读者大概了解 Eureka 架构及各个角色的作用即可，说一千道一万，还不如来个案例实际，下一小节将以一个案例来展示 Eureka 的作用。

4.2 第一个 Eureka 应用

本小节将编写一个 Hello Wrold 小程序，来演示 Eureka 作用，程序中将会包含服务器、服务提供者以及服务调用者。

4.2.1 构建服务器

先创建一个名称为 first-ek-server 的 Maven 项目作为服务器，在 pom.xml 文件中加入 Spring Cloud 的依赖，如代码清单 3-1 所示。

代码清单 3-1：codes\03\first-ek-server\pom.xml

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Dalston.SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

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

加入的 spring-cloud-starter-eureka-server，会自动引入 spring-boot-starter-web，因此只需要加入该依赖，我们的项目就具有 Web 容器的功能。接下来，编写一个最简单的启动类，启动我们的 Eureka 服务器，启动类如代码清单 3-2 所示。

代码清单 3-2：codes\03\first-ek-server\src\main\java\org\crazyit\cloud\FirstServer.java

```
@SpringBootApplication
@EnableEurekaServer
public class FirstServer {

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

}
```

启动类几乎与前面章节的 Spring Boot 项目一致，只是加入了@EnableEurekaServer，声明这是一个 Eureka 服务器。直接运行 FirstServer 即可启动 Eureka 服务器，需要注意的

是，本例中并没有配置服务器端口，因此默认端口为 8080，我们将端口配置为 8761，在 src/main/resources 目录下创建 application.yml 配置文件，内容如下：

```
server:
  port: 8761
```

本书的大部分案例使用 yml 文件进行配置，以上的配置片断声明服务器的 HTTP 端口为 8761，该配置是 Spring Boot 的公共配置，Sprin Boot 有近千个公共配置，如想了解这些配置，可参看 Spring Boot 的文档。运行 FirstServer 的 main 方法后，可以看到控制台输出如下：

```
2017-08-04 15:35:58.900 INFO 4028 --- [ main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8761 (http)
2017-08-04 15:35:58.901 INFO 4028 --- [ main] .s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 8761
2017-08-04 15:35:58.906 INFO 4028 --- [ main] org.crazyit.cloud.FirstServer : Started FirstServer in 12.361 seconds (JVM running for 12.891)
2017-08-04 15:35:59.488 INFO 4028 --- [nio-8761-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring FrameworkServlet 'dispatcherServlet'
```

在启动过程中会出现部分异常信息，暂时不需要进行处理，将在后面章节讲解如何避免出现这些异常。成功启动后，打开浏览器，输入：http://localhost:8761，可以看到 Eureka 服务器控制台，如图 3-2 所示。

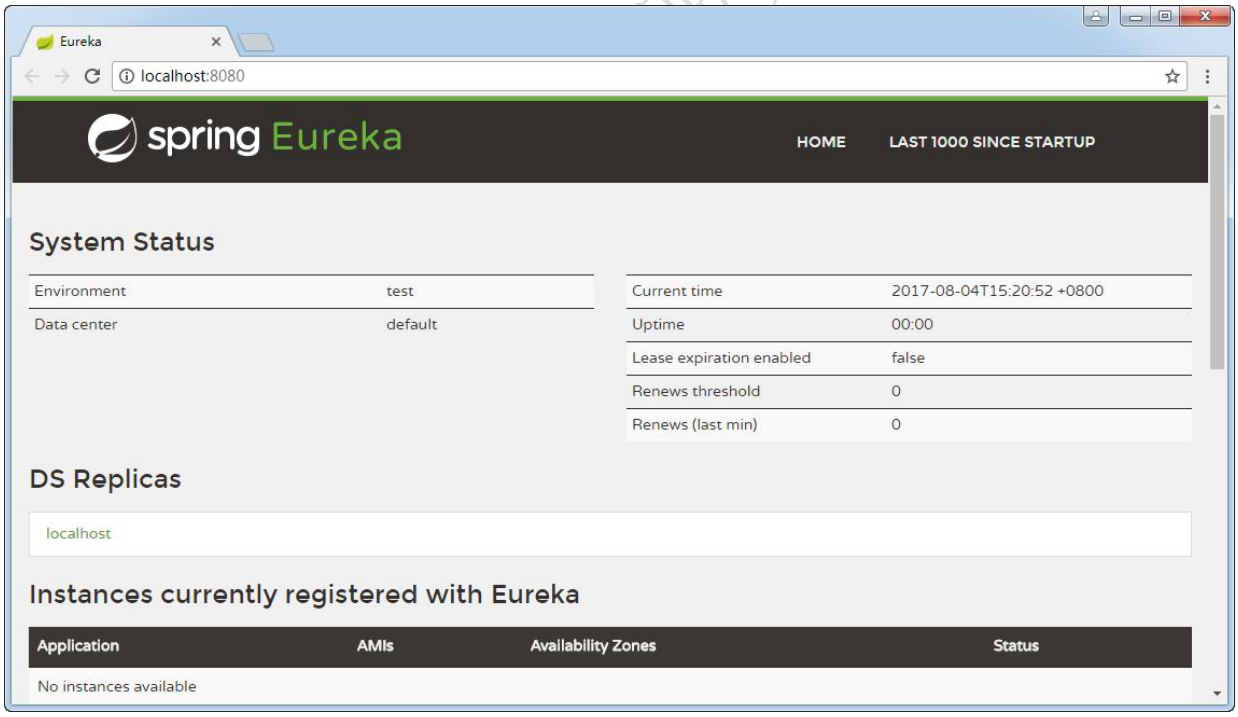

图 3-2 Eureka 控制台

在图 3-2 的下方，可以看到服务的实例列表，目前我们并没有注册服务，因此列表为空。

4.2.2 服务器注册开关

在启动 Eureka 服务器时，会在控制台看到以下两个异常信息：

```
java.net.ConnectException: Connection refused: connect
```

```
com.netflix.discovery.shared.transport.TransportException: Cannot execute request on any known server
```

这是由于在服务器启动时，服务器会把自己当作一个客户端，注册去 Eureka 服务器，并且会到 Eureka 服务器抓取注册信息，它自己本身只是一个服务器，而不是服务的提供者（客户端），因此可以修改 application.yml 文件，修改以下两个配置：

```
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
```

以上配置中的 eureka.client.registerWithEureka 属性，声明是否将自己的信息注册到 Eureka 服务器，默认值为 true。属性 eureka.client.fetchRegistry 则表示，是否到 Eureka 服务器中抓取注册信息。将这两个属性设置为 false，则启动时不会出现异常信息。

4.2.3 编写服务提供者

在前面搭建环境章节，我们使用了 Spring Boot 来建立了一个简单的 Web 工程，并且在里面编写了一个 REST 服务，本例中的服务提供者，与该案例类似。建立名称为“first-ek-service-invoker”的项目，pom.xml 中加入依赖，如代码清单 3-2 所示。

代码清单 3-2：codes\03\3.2\first-ek-service-provider\pom.xml

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

在 src/main/resources 目录中建立 application.yml 配置文件，文件内容如代码清单 3-3 所示。

代码清单 3-3：codes\03\3.2\first-ek-service-provider\src\main\resources\application.yml

```
spring:
  application:
    name: first-service-provider
eureka:
  instance:
    hostname: localhost
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

以上配置中，将应用名称配置为“first-service-provider”，该服务将会被注册到端口为 8761 的 Ereka 服务器，也就是本小节前面所构建的服务器。另外，还使用了 eureka.instance.hostname 来配置该服务实例的主机名称。编写一个 Controller 类，并提供一个最简单的 REST 服务，如代码清单 3-4。

代码清单 3-4：

codes\03\3.2\first-ek-service-provider\src\main\java\org\crazyit\cloud\FirstController.java

```
@RestController
public class FirstController {

    @RequestMapping(value = "/person/{personId}", method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON_VALUE)
    public Person findPerson(@PathVariable("personId") Integer personId) {
        Person person = new Person(personId, "Crazyit", 30);
        return person;
    }
}
```

编写启动类，请见代码清单 3-5。

代码清单 3-5：

codes\03\3.2\first-ek-service-provider\src\main\java\org\crazyit\cloud\FirstServiceProvider.java

```
@SpringBootApplication
@EnableEurekaClient
public class FirstServiceProvider {

    public static void main(String[] args) {
        new SpringApplicationBuilder(FirstServiceProvider.class).run(args);
    }
}
```

在启动类中，使用了@EnableEurekaClient 注解，声明该应用是一个 Eureka 客户端。配置完成后，运行服务器项目“first-ek-server”的启动类 FirstServer，再运行代码清单 3-5 的 FirstServiceProvider，的浏览器中访问 Eureka：http://localhost:8761/，可以看到服务列表如图 3-3 所示。

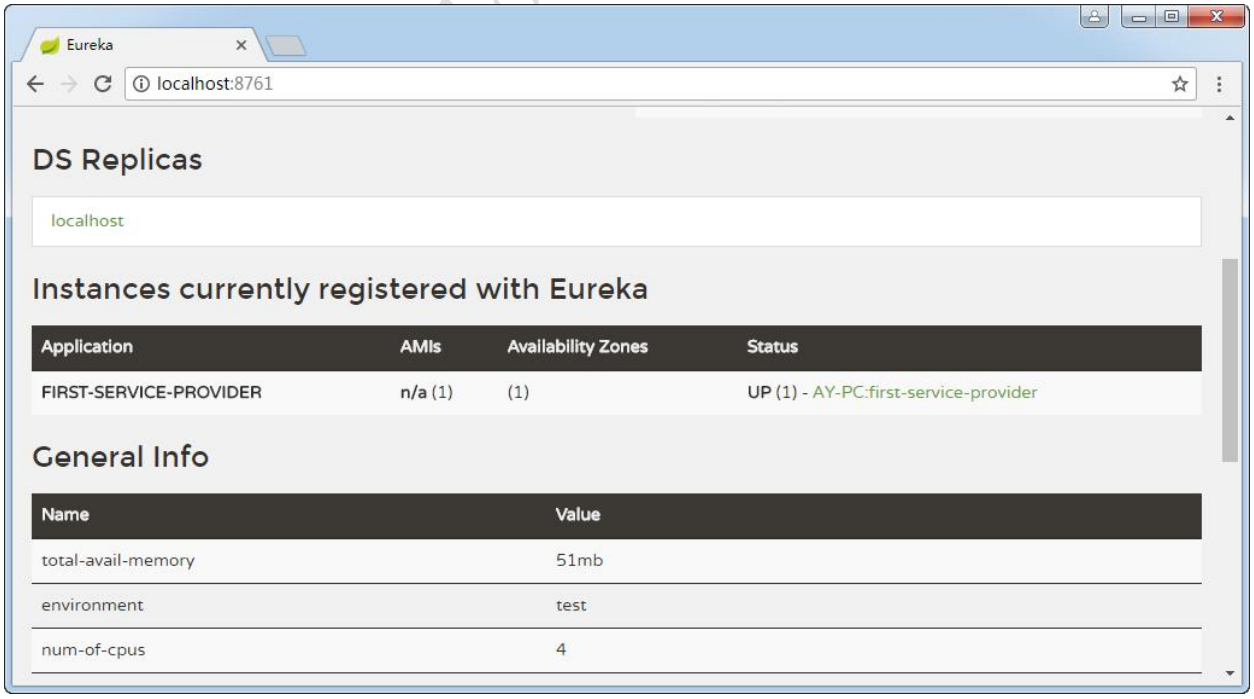

图 3-3 查看发布的服务

如图 3-3，可以看到当前注册的服务列表，只有我们编写的“first-service-provider”。服务注册成功后，接下来编写服务调用者。

4.2.4 编写服务调用者

服务被注册、发布到 Eureka 服务器后，就需要有程序去发现它，并且进行调用。此处所说的调用者，是指同样注册到 Eureka 的客户端，来调用其他客户端发布的服务，简单的说，就是 Eureak 内部调用。由于同一个服务，可能会部署多个实例，调用过程可能涉及负载均衡、服务器查找等问题，Netflix 的项目已经帮我们解决，并且 Spring Cloud 已经封装了一次，我们仅需编写少量代码，就可以实现服务调用。

新建名称为“first-ek-service-invoker”的项目，在 pom.xml 文件中加入依赖，如代码清单 3-6 所示。

代码清单 3-6：codes\03\3.2\first-ek-service-invoker\pom.xml

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

建立配置文件 application.yml，内容如代码清单 3-7。

代码清单 3-7：codes\03\3.2\first-ek-service-invoker\src\main\resources\application.yml

```
server:
  port: 9000
spring:
  application:
    name: first-service-invoker
eureka:
  instance:
    hostname: localhost
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

在配置文件中，配置了应用名称为“first-service-invoker”，这个调用者的访问端口为 9000，需要注意的，这个调用本身也可以对外提供服务。与提供者一样，使用 eureka 的配置，将调用者注册到“first-ek-server”上面。下面编写一个控制器，让调用者对外提供一个测试的服务，代码清单 3-8 为控制器的代码实现。

代码清单 3-8：

codes\03\3.2\first-ek-service-invoker\src\main\java\org\crazyit\cloud\InvokerController.java

```
@RestController
@Configuration
public class InvokerController {

    @Bean
```

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

    @RequestMapping(value = "/router", method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON_VALUE)
    public String router() {
        RestTemplate restTpl = getRestTemplate();
        // 根据应用名称调用服务
        String json = restTpl.getForObject(
                "http://first-service-provider/person/1", String.class);
        return json;
    }
}
```

在控制器中，配置了 RestTemplate 的 bean，RestTemplate 本来是 spring-web 模块下面的类，主要用来调用 REST 服务，本身并不具备调用分布式服务的能力，但是 RestTemplate 的 bean 被@LoadBalanced 注解修饰后，这个 RestTemplate 实例就具有访问分布式服务的能力，关于该类的一些机制，我们将放到负载均衡一章中讲解。

在控制器中，新建了一个 router 的测试方法，用来对外发布 REST 服务，该方法只是一个路由作用，实际上使用 RestTemplate 来调用“first-ek-service-provider”（服务提供者）的服务。需要注意的是，调用服务时，仅仅是通过服务名称来进行调用。接下来编写启动类，如代码清单 3-9 所示。

代码清单 3-9：

codes\03\3.2\first-ek-service-invoker\src\main\java\org\crazyit\cloud\FirstInvoker.java

```
@SpringBootApplication
@EnableDiscoveryClient
public class FirstInvoker {

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

在启动类中，使用了@EnableDiscoveryClient 注解来修改启动类，该注解使得服务调用者，有能力去 Eureka 中发现服务，需要注意的是@EnableEurekaClient 注解已经包含了@EnableDiscoveryClient 的功能，也就是说，一个 Eureka 客户端，本身就具有发现服务的能力。配置完成后，依次执行以下操作：

- 启动服务器（first-ek-server）
- 启动服务提供者（first-ek-service-provider）
- 启动服务调用者（first-ek-service-invoker）

使用浏览器访问 Eureka，可看到注册的客户信息，如图 3-4。

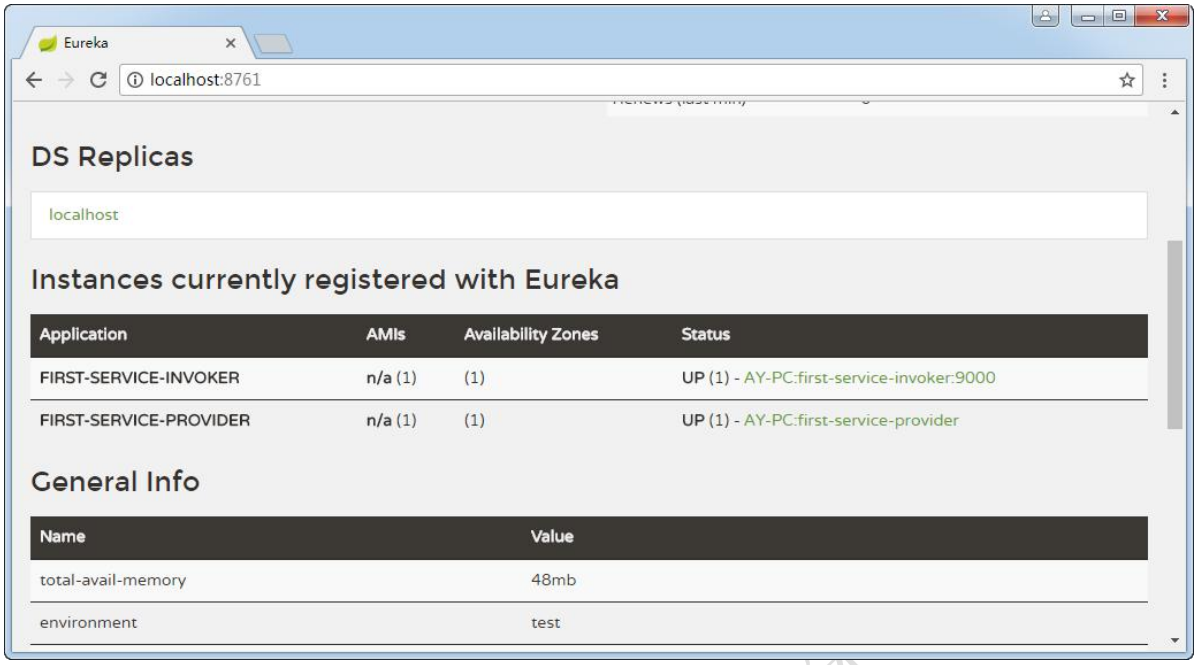

图 3-4 服务列表

全部成功启动后，在浏览器中访问服务调用者发布的“router”服务：http://localhost:9000/router，可以看到在浏览器输出如下：

```
{"id":1,"name":"Crazyit","age":30}
```

根据输出可知，实际上调用了服务提供者的/person/1 服务，第一个 Eureka 应用到此结束，下面对这个应用程序的结构作一个简单描述。

4.2.5 程序结构

本案例新建了三个项目，如果读者对程序的结构不太清晰，可以参看图 3-5。

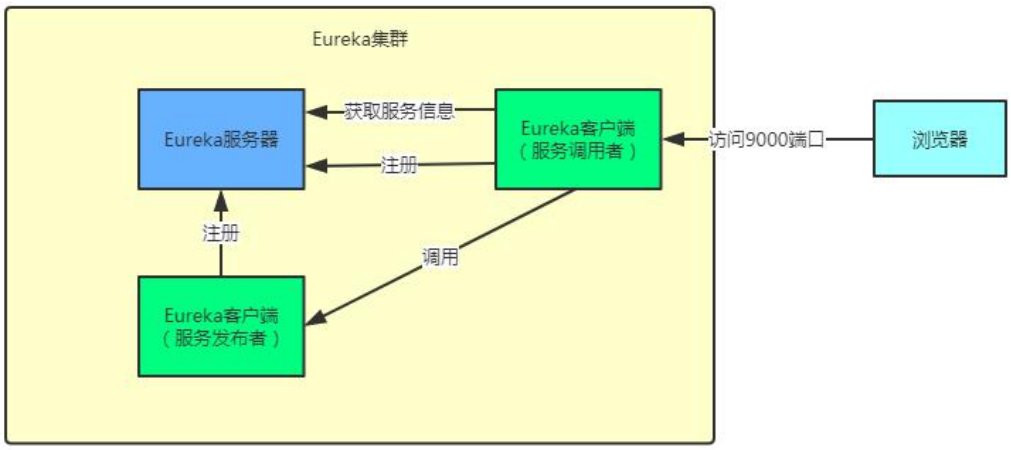

图 3-5 本案例的结构

如图 3-5，Eureka 服务为本例的“first-ek-server”，服务发布者为“first-ek-service-provider”，而调用者为“first-ek-service-invoker”，用户通过浏览器访问调用者的 9000 端口的 router 服务，router 服务中查找服务提供者的服务并进行调用。在本

例中，服务调用有点像路由器的角色。

5 Eureka 集群搭建

要点

- 搭建 Eureka 集群

5.1 Eureka 集群搭建

在运行第一个 Eureka 应用时，服务器实例、服务提供者实例都只是启动了一个，并没有体现高可用的特性，本小节将对前面的 Eureka 应用进行改造，使其可以进行集群部署。

5.1.1 本例集群结构图

本例将会运行两个服务器实例、两个服务提供者实例，然后服务调用者请求服务，集群结构如图 3-6 所示。

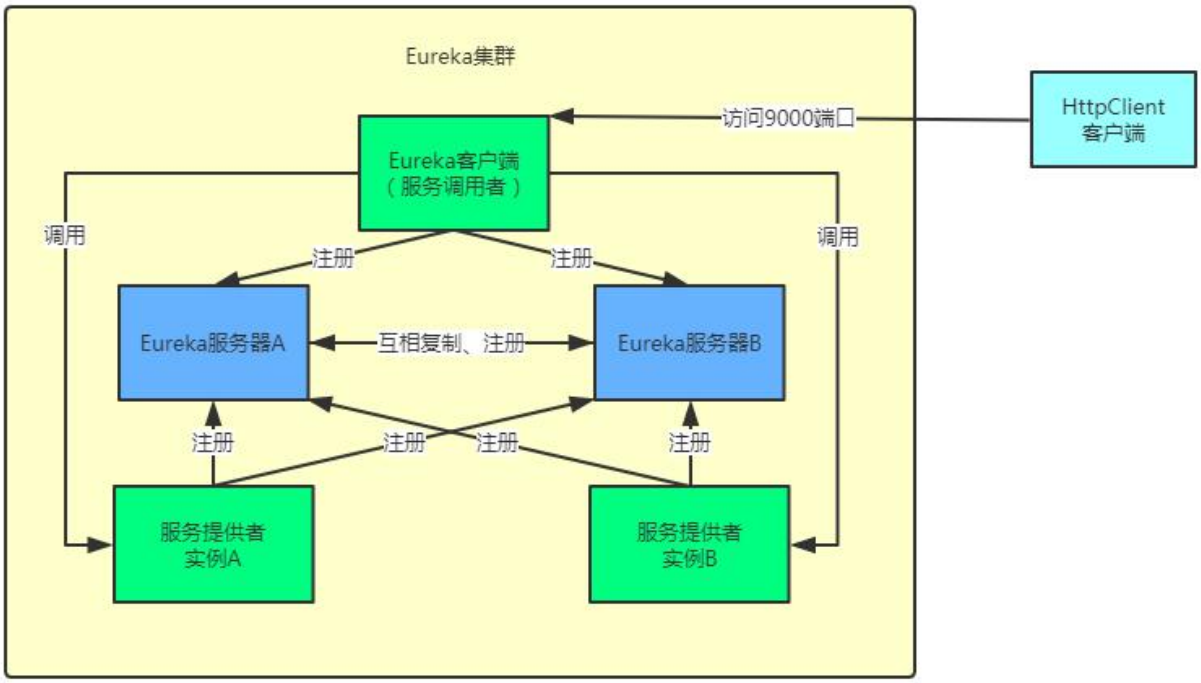

图 3-6 集群结构

第一个 Eureka 应用，使用的是浏览器访问 Eureka 的服务调用者，而改造后，为了能看到负载均衡的效果，会编写一个 HttpClient 的 REST 客户端访问服务调用者发布的服务。

由于本书的开发环境只有一台电脑，操作系统为 Windows，如果要构建集群，需要修改 hosts 文件，为其添加主机名的映射。修改 C:\Windows\System32\drivers\etc\hosts 文件，添加以下内容：

```
127.0.0.1 slave1 slave2
```

5.1.2 改造服务器端

新建项目“first-cloud-server”，使用 Maven 配置与 3.2 章节的服务器一致，由于需要对同一个应用程序启动两次，因此需要在配置文件中使用 profiles（关于 profiles 已经在第 2 章中讲述过）。服务器配置文件请见代码清单 3-10。

代码清单 3-10：codes\03\3.3\first-cloud-server\src\main\resources\application.yml

```
server:
  port: 8761
spring:
  application:
    name: first-cloud-server
  profiles: slave1
eureka:
  instance:
    hostname: slave1
  client:
    serviceUrl:
      defaultZone: http://slave2:8762/eureka/
---
server:
  port: 8762
spring:
  application:
    name: first-cloud-server
  profiles: slave2
eureka:
  instance:
    hostname: slave2
  client:
    serviceUrl:
      defaultZone: http://slave1:8761/eureka/
```

代码清单 3-10 中配置了两个 profiles，名称分别为 slave1 和 slave2。在 slave1 中，配置了应用端口为 8761，主机名为 slave1，当使用 salve1 这个 profiles 来启动服务器时，将会向 http://slave2:8762/eureka/ 注册自己。使用 salve2 来启动服务器，会向 http://slave1:8761/eureka/注册自己。**简单点说，就是两个服务器启动后，它们会互相注册。**

修改启动类，让类在启动时，读取控制台的输入，决定使用哪个 profiles 来启动服务器，请见代码清单 3-11。

代码清单 3-11：codes\03\3.3\first-cloud-server\src\main\java\org\crazyit\cloud\FirstServer.java

```
@SpringBootApplication
@EnableEurekaServer
```

```
public class FirstServer {

    public static void main(String[] args) {
        // 读取控制台输入，决定使用哪个 profiles
        Scanner scan = new Scanner(System.in);
        String profiles = scan.nextLine();
        new SpringApplicationBuilder(FirstServer.class).profiles(profiles).run(args);
    }
}
```

启动类中，先读取控制的输入，再调用 profiles 方法来设置启动的 profles。需要注意的是，第一个启动的服务器会抛出异常，异常原因我们前已经讲述，抛出的异常可不必理会。

5.1.3 改造服务提供者

服务提供者也需要启动两个实例，服务提供者的改造与服务端类似，将 3.2 章节中的“first-ek-service-provider”复制出来，并改名为“first-cloud-provider”。修改配置文件，将服务提供者注册到两个服务器中，配置文件请见代码清单 3-12。

代码清单 3-12：codes\03\3.3\first-cloud-provider\src\main\resources\application.yml

```
spring:
  application:
    name: first-cloud-provider
eureka:
  instance:
    hostname: localhost
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/,http://localhost:8762/eureka/
```

再修改启动类，为了避免端口决定，启动时读取控制台输出，决定使用哪个端口来启动，启动类如代码清单 3-13 所示。

代码清单 3-13：

codes\03\3.3\first-cloud-provider\src\main\java\org\crazyit\cloud\FirstServiceProvider.java

```
        // 读取控制台输入的端口，避免端口冲突
        Scanner scan = new Scanner(System.in);
        String port = scan.nextLine();
        new SpringApplicationBuilder(FirstServiceProvider.class).properties(
                "server.port=" + port).run(args);
```

启动类中使用了 properties 方法来设置启动端口。为了能看到效果，还需要改造控制器，将服务调用者请求的 URL 保存起来并返回，修改后的控制器请见代码清单 3-14。

代码清单 3-14：

codes\03\3.3\first-cloud-provider\src\main\java\org\crazyit\cloud\FirstController.java

```
@RestController
public class FirstController {

    @RequestMapping(value = "/person/{personId}", method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON_VALUE)
    public Person findPerson(@PathVariable("personId") Integer personId, HttpServletRequest
request) {
```

```
        Person person = new Person(personId, "Crazyit", 30);
        // 为了查看结果，将请求的 URL 设置到 Person 实例中
        person.setMessage(request.getRequestURL().toString());
        return person;
    }
}
```

控制器的 findPerson 方法，将请求的 URL 保存到 Person 实例的 message 属性中，调用服务后，可以通过 message 属性来查看请求的 URL。

5.1.4 改造服务调用者

将 3.2 章节中的“first-ek-service-invoker”复制并改名为“first-cloud-invoker”。本例中的服务调用者只需启动一个实例，因此修改下配置文件即可使用，请见代码清单 3-15。

代码清单 3-15：codes\03\3.3\first-cloud-invoker\src\main\resources\application.yml

```
server:
  port: 9000
spring:
  application:
    name: first-cloud-invoker
eureka:
  instance:
    hostname: localhost
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/,http:// localhost:8761/eureka/
```

修改的配置，将服务调用注册到两个服务器上。

5.1.5 编写 REST 客户端进行测试

本例使用的是 HttpClient，HttpClient 是 Apache 提供的一个 HTTP 工具包。新建名称为“first-cloud-rest-client”的项目，在 pom.xml 中加入以下依赖：

```
        <dependency>
            <groupId>org.apache.httpcomponents</groupId>
            <artifactId>httpclient</artifactId>
            <version>4.5.2</version>
        </dependency>
```

新建启动类，在 main 方法中编写调用 REST 服务的代码，如代码清单 3-16 所示。

代码清单 3-16：03\3.3\first-cloud-rest-client\src\main\java\org\crazyit\cloud\TestHttpClient.java

```
        // 创建默认的 HttpClient
        CloseableHttpClient httpclient = HttpClients.createDefault();
        // 调用 6 次服务并输出结果
        for(int i = 0; i < 6; i++) {
            // 调用 GET 方法请求服务
            HttpGet httpget = new HttpGet("http://localhost:9000/router");
            // 获取响应
            HttpResponse response = httpclient.execute(httpget);
```

```
            // 根据 响应解析出字符串
            System.out.println(EntityUtils.toString(response.getEntity()));
        }
```

在 main 方法，调用了 6 次 9000 端口的 router 服务并输出结果。完成编写后，按以下顺序启动各个组件：

- 启动两个服务器端，控制台中分别输入“slave1”和“slave2”。
- 启动两个服务提供者，控制台分别输入 8081 与 8082。
- 启动服务调用者。

启动了整个集群后，运行 TestHttpClient，可以看到输出如下：

```
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8081/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8082/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8081/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8082/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8081/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8082/person/1"}
```

根据输出结果可知，8081 与 8082 端口分别被请求了 3 次，可见已经达到负载均衡的目的，关于负载均衡更详细的内容，将在后面章节中讲解。

6 负载均衡框架 Ribbon 介绍

本文要点

- 认识 Ribbon
- 第一个 Ribbon 程序

负载均衡是分布式架构的重点，负载均衡机制将决定着整个服务集群的性能与稳定。根据前面章节可知，Eureka 服务实例可以进行集群部署，每个实例都均衡处理服务请求，那么这些请求是如何被分摊到各个服务实例中的？本章将讲解 Netflix 的负载均衡项目 Ribbon。

6.1 Ribbon 介绍

6.1.1 Ribbon 简介

Ribbon 是 Netflix 下的负载均衡项目，它在集群中为各个客户端的通信提供了支持，它主要实现中间层应用程序的负载均衡。Ribbon 提供以下特性：

- 负载均衡器，可支持插拔式的负载均衡规则。
- 对多种协议提供支持，例如 HTTP、TCP、UDP。

- 集成了负载均衡功能的客户端。

同为 Netflix 项目，Ribbon 可以与 Eureka 整合使用，Ribbon 同样被集成到 Spring Cloud 中，作为 spring-cloud-netflix 项目中的子模块。Spring Cloud 将 Ribbon 的 API 进行了封装，使用者可以使用封装后的 API 来实现负载均衡，也可以直接使用 Ribbon 的原生 API。

6.1.2 Ribbon 子模块

Ribbon 主要有以下三大子模块：

- ribbon-core：该模块为 Ribbon 项目的核心，主要包括负载均衡器接口定义、客户端接口定义，内置的负载均衡实现等 API。
- ribbon-eureka：为 Eureka 客户端提供的负载均衡实现类。
- ribbon-httpclient：对 Apache 的 HttpClient 进行封装，该模块提供了含有负载均衡功能的 REST 客户端。

6.1.3 负载均衡器组件

Ribbon 的负载均衡器主要与集群中的各个服务器进行通信，负载均衡器需要提供以下基础功能：

- 维护服务器的 IP、DNS 名称等信息。
- 根据特定的逻辑在服务器列表中循环。

为了实现负载均衡的基础功能，Ribbon 的负载均衡器有以下三大子模块：

- Rule：一个逻辑组件，这些逻辑将会决定，从服务器列表中返回哪个服务器实例。
- Ping：该组件主要使用定时器，来确保服务器网络可以连接。
- ServerList：服务器列表，可以通过静态的配置确定负载的服务器，也可以动态指定服务器列表。如果动态指定服务器列表，则会有后台的线程来刷新该列表。

本章关于 Ribbon 的知识，主要围绕负载均衡器组件进行。接下来，先编写一个 Ribbon 程序，让大家对 Ribbon 有一个初步的认识。

6.2 第一个 Ribbon 程序

关于整合 Spring Cloud 的内容，会在后面章节讲述。本小节将以一个简单的 Hello World 程序，来展示 Ribbon API 的使用。本例的程序结构如图 4-1 所示。

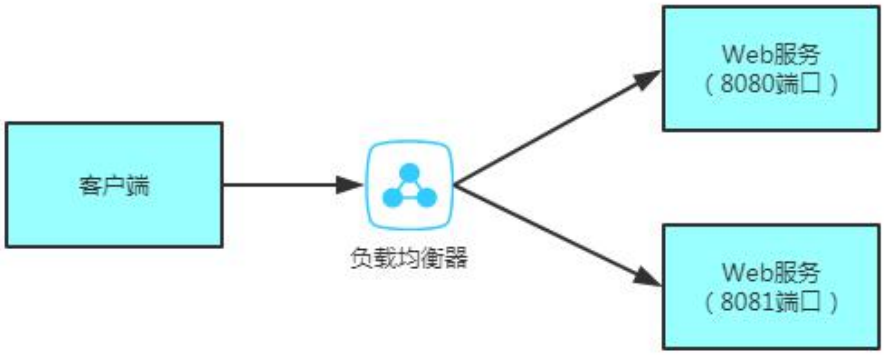

图 4-1 本例程序结构

本书所使用的 Spring Cloud，默认集成的 Ribbon 版本为 2.2.2，因此本书也使用该版本的 Ribbon。

6.2.1 编写服务

为了能查看负载均衡效果，先编写一个简单的 REST 服务，通过指定不同的端口，让服务可以启动多个实例。本例的请求服务器，仅仅是一个基于 Spring Boot 的 Web 应用，与书 2.3 章节的应用类似，如果读者熟悉建立过程，可跳过部分创建过程，本小节最终目的是发布两个 REST 服务。

新建名称为“first-ribbon-server”的 Maven 项目，加入以下依赖：

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <version>1.5.4.RELEASE</version>
        </dependency>
```

建立 Spring Boot 启动类，如代码清单 4-1 所示。

代码清单 4-1：

codes\04\4.2\first-ribbon-server\src\main\java\org\crazyit\cloud\FirstServerApplication.java

```
@SpringBootApplication
public class FirstServerApplication {

    public static void main(String[] args) {
        // 读取控制台输入作为端口参数
        Scanner scan = new Scanner(System.in);
        String port = scan.nextLine();
        // 设置启动的服务器端口
        new SpringApplicationBuilder(FirstServerApplication.class).properties(
                "server.port=" + port).run(args);
    }
}
```

运行 main 方法，并在控制台输入端口号，即可启动 Web 服务器。接下来编写控制器，添加一个 REST 服务，请见代码清单 4-2。

代码清单 4-2：

codes\04\4.2\first-ribbon-server\src\main\java\org\crazyit\cloud\MyController.java

```
@RestController
public class MyController {

    @RequestMapping(value = "/person/{personId}", method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON_VALUE)
    public Person findPerson(@PathVariable("personId") Integer personId,
            HttpServletRequest request) {
        Person p = new Person();
        p.setId(personId);
        p.setName("Crazyit");
        p.setAge(30);
        p.setMessage(request.getRequestURL().toString());
        return p;
    }

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String hello() {
        return "hello";
    }
}
```

在控制器中，发布了两个的 REST 服务。其中，调用地址为“/person/personId”的服务后，会返回一个 Person 实例的 JSON 字符串，为了看到请求的 URL，为 Person 的 message 属性设置了请求的 URL。

6.2.2 编写请求客户端

新建名称为“first-ribbon-client”的 Maven 项目，加入以下依赖：

```
<dependency>
    <groupId>com.netflix.ribbon</groupId>
    <artifactId>ribbon</artifactId>
    <version>2.2.2</version>
</dependency>
<dependency>
    <groupId>com.netflix.ribbon</groupId>
    <artifactId>ribbon-httpclient</artifactId>
    <version>2.2.2</version>
</dependency>
```

接下来，使用 Ribbon 的客户端发送请求，请见代码清单 4-3。

代码清单 4-3：

codes\04\4.2\first-ribbon-client\src\main\java\org\crazyit\cloud\TestRestClient.java

```
public class TestRestClient {

    public static void main(String[] args) throws Exception {
        // 设置请求的服务器
        ConfigurationManager.getConfigInstance().setProperty(
                "my-client.ribbon.listOfServers",
                "localhost:8080,localhost:8081");
        // 获取 REST 请求客户端
        RestClient client = (RestClient) ClientFactory
```

```
                .getNamedClient("my-client");
        // 创建请求实例
        HttpRequest request = HttpRequest.newBuilder().uri("/person/1").build();
        // 发 送 6 次请求到服务器中
        for (int i = 0; i < 6; i++) {
            HttpResponse response = client.executeWithLoadBalancer(request);
            String result = response.getEntity(String.class);
            System.out.println(result);
        }
    }
}
```

代码清单 4-3 中，使用了 ConfigurationManager 类来配置了请求的服务器列表，为“localhost:8080”与“localhost:8081”，再使用 RestClient 对象，向“/person/1”地址发送 6 次请求。

启动两次服务器类 FirstServerApplication，并在控制台分别输入 8080 和 8081 端口。启动服务器后，运行客户端，输出结果如下：

```
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8081/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8080/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8081/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8080/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8081/person/1"}
{"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8080/person/1"}
```

根据输出结果可知，RestClient 轮流向 8080 与 8081 端口发送请求，可见 RestClient 中已经帮我们实现了负载均衡的功能。

6.2.3 Ribbon 配置

在编写客户端时，使用了 ConfigurationManager 来设置配置项，除了在代码中指定配置项外，还可以将配置放到“.properties”文件中，ConfigurationManager 的 loadPropertiesFromResources 方法可以指定 properties 文件的位置，配置格式如下：

```
<client>.<nameSpace>.<property>=<value>
```

其中<client>为客户的名称，声明该配置属于哪一个客户端，在使用 ClientFactory 时可传入客户端的名称，即可返回对应的“请求客户端”实例。<nameSpace>为该配置的命名空间，默认为“ribbon”，<property>为属性名，<value>为属性值。如果想对全部的客户端生效，可以将客户端名称去掉，直接以“<namespace>.<property>”的格式进行配置。以下的配置，为客户端指定服务器列表：

```
my-client.ribbon.listOfServers=localhost:8080,localhost:8081
```

Ribbon 的配置，同样可以使用在 Spring Cloud 的配置文件（即 application.yml）中使用。

7 Ribbon 负载均衡器

本文要点

- Ribbon 负载均衡器

7.1 Ribbon 负载均衡器

Ribbon 提供了几个负载均衡的组件，其目的就是为了让请求转给合适的服务器处理，因此，如何选择合适的服务器，便成为负载均衡机制的核心。本小节将围绕 Ribbon 负载均衡器的组件，向大家展示 Ribbon 负载均衡的实现机制。

7.1.1 负载均衡器

Ribbon 的负载均衡器接口，定义了服务器的操作，主要是用于进行服务器选择。前面的例子中，客户端使用了 RestClient 类，在发送请求时，会使用负载均衡器（ILoadBalancer）接口，根据特定的逻辑来选择服务器，服务器列表可使用 listOfServers 进行配置，也可以使用动态更新机制。代码清单 4-4 使用负载均衡器来选择服务器。

代码清单 4-4：

codes\04\4.2\first-ribbon-client\src\main\java\org\crazyit\cloud\ChoseServerTest.java

```
// 创建负载均衡器
ILoadBalancer lb = new BaseLoadBalancer();
// 添加服务器
List<Server> servers = new ArrayList<Server>();
servers.add(new Server("localhost", 8080));
servers.add(new Server("localhost", 8081));
lb.addServers(servers);
// 进行 6 次服务器选择
for(int i = 0; i < 6; i++) {
    Server s = lb.chooseServer(null);
    System.out.println(s);
}
```

代码中使用了 BaseLoadBalancer 这个负载均衡器，将两个服务器对象加入到负载均衡器中，再调用 6 次 chooseServer 方法，可以看到输出如下：

```
localhost:8081
localhost:8080
localhost:8081
localhost:8080
localhost:8081
localhost:8080
```

根据结果可知，最终选择的服务器与前面章节一致，可以判定本例第一个 Ribbon 例子，选择服务器的逻辑是一致的，在默认情况下，会使用 RoundRobinRule 的规则逻辑。

7.1.2 自定义负载规则

根据前一小节可知，选择哪个服务器进行请求处理，由 ILoadBalancer 接口的 chooserServer 方法决定，而在 BaseLoadBalancer 类中，则使用 IRule 接口的 choose 方法来决定选择哪一个服务器对象。如果想自定义负载均衡规定，可以编写一个 IRule 接口的实现类。代码清单 4-5 实现了自己的负载规定。

代码清单 4-5：codes\04\4.2\first-ribbon-client\src\main\java\org\crazyit\cloud\MyRule.java

```
public class MyRule implements IRule {

    ILoadBalancer lb;

    public MyRule() {
    }

    public MyRule(ILoadBalancer lb) {
        this.lb = lb;
    }

    public Server choose(Object key) {
        // 获取全部的服务器
        List<Server> servers = lb.getAllServers();
        // 只返回第一个 Server 对象
        return servers.get(0);
    }

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

    public ILoadBalancer getLoadBalancer() {
        return this.lb;
    }
}
```

在自定义规则类中，实现的 choose 方法，调用了 ILoadBalancer 的 getAllServers 方法返回全部的服务器，为了简单起见，本例只返回第一个服务器。为了能在负载均衡器中使用自定义的规则，需要修改选择服务器的代码，请见代码清单 4-6。

代码清单 4-6：codes\04\4.2\first-ribbon-client\src\main\java\org\crazyit\cloud\TestMyRule.java

```
        // 创建负载均衡器
        BaseLoadBalancer lb = new BaseLoadBalancer();
        // 设置自定义的负载规则
        lb.setRule(new MyRule(lb));
        // 添加服务器
        List<Server> servers = new ArrayList<Server>();
        servers.add(new Server("localhost", 8080));
        servers.add(new Server("localhost", 8081));
        lb.addServers(servers);
        // 进行 6 次服务器选择
```

```
for(int i = 0; i < 6; i++) {
    Server s = lb.chooseServer(null);
    System.out.println(s);
}
```

运行代码清单 4-6，可以看到，请求 6 次所得到的服务器均为“localhost:8080”。以上是直接使用编码方式来设置负载规则，可以使用配置的方式来完成这些工作。修改 Ribbon 的配置，让请求的客户端，使用我们定义的负载规则，请见代码清单 4-7。

代码清单 4-7：

codes\04\4.2\first-ribbon-client\src\main\java\org\crazyit\cloud\TestMyRuleConfig.java

```
// 设置请求的服务器
ConfigurationManager.getConfigInstance().setProperty(
        "my-client.ribbon.listOfServers",
        "localhost:8080,localhost:8081");
// 配置规则处理类
ConfigurationManager.getConfigInstance().setProperty(
        "my-client.ribbon.NFLoadBalancerRuleClassName",
        MyRule.class.getName());
// 获取 REST 请求客户端
RestClient client = (RestClient) ClientFactory
        .getNamedClient("my-client");
// 创建请求实例
HttpRequest request = HttpRequest.newBuilder().uri("/person/1").build();
// 发 送 10 次请求到服务器中
for (int i = 0; i < 6; i++) {
    HttpResponse response = client.executeWithLoadBalancer(request);
    String result = response.getEntity(String.class);
    System.out.println(result);
}
```

请求客户端中，与前面章节的客户端基本一致，只是加入了“my-clent.ribbon.NFLoadBalancerRuleClassName”属性，设置了自定义规则处理类为 MyRule，这个配置项同样可以在配置文件中使用，包括 Spring Cloud 的配置文件（application.yml 等）。

启动前面章节的服务器端两次，分别设置 8080 与 8081 端口，再运行代码清单 4-7，可以看到输出了 6 次 {"id":1,"name":"Crazyit","age":30,"message":"http://localhost:8080/person/1"}，根据结果可知，我们的自定义规则生效，请求只让 8080 端口处理。

在实际环境中，如果要实现自定义的负载规则，可能还需要结合各种因素，例如考虑具体业务的发生时间、服务器性能等，实现中还可能还涉及使用计算器、数据库等技术，具体情形会更为复杂，本例的负载规则较为简单，目的是让读者了解负载均衡的原理。

7.1.3 Ribbon 自带的负载规则

Ribbon 提供了若干个内置的负载规则，使用者完全可以直接使用，主要有以下内置的负载规则：

- RoundRobinRule：系统默认的规则，通过简单的轮询服务列表来选择服务器，其他的规则在很多情况下，仍然使用 RoundRobinRule。

- AvailabilityFilteringRule：该规则会忽略以下服务器：
  - 无法连接的服务器：在默认情况下，如果 3 次连接失败，该服务器将会被置为“短路”的状态，该状态将持续 30 秒，如果再次连接失败，“短路”状态的持续时间将会以几何级增加。可以通过修改 niws.loadbalancer.<clientName>.connectionFailureCountThreshold 属性，来配置连接失败的次数。
  - 并发数过高的服务器：如果连接到该服务器的并发数过高，也会被这个规则忽略，可以通过修改<clientName>.ribbon.ActiveConnectionsLimit 属性来设定最高并发数。
- WeightedResponseTimeRule：为每个服务器赋予一个权重值，服务器的响应时间越长，该权重值就是越少，这个规则会随机选择服务器，这个权重值有可能会决定服务器的选择。
- ZoneAvoidanceRule：该规则以区域、可用服务器为基础，进行服务器选择。使用 Zone 对服务器进行分类，可以理解为机架或者机房。
- BestAvailableRule：忽略“短路”的服务器，并选择并发数较低的服务器。
- RandomRule：顾名思义，随机选择可用的服务器。
- RetryRule：含有重试的选择逻辑，如果使用 RoundRobinRule 选择服务器无法连接，那么将会重新选择服务器。

以上提供的负载规则，基本可以满足大部分的需求，如果有更为复杂的要求，建议实现自定义负载规则。

7.1.4 Ping 机制

在负载均衡器中，提供了 Ping 的机制，每隔一段时间，会去 Ping 服务器，判断服务器是否存活。该工作由 IPing 接口的实现类负责，如果单独使用 Ribbon，在默认情况下，不会激活 Ping 机制，默认的实现类为 DummyPing。代码清单 4-8，使用另外一个 IPing 实现类 PingUrl。

代码清单 4-8：codes\04\4.2\first-ribbon-client\src\main\java\org\crazyit\cloud\TestPingUrl.java

```
// 创建负载均衡器
BaseLoadBalancer lb = new BaseLoadBalancer();
// 添加服务器
List<Server> servers = new ArrayList<Server>();
// 8080 端口连接正常
servers.add(new Server("localhost", 8080));
// 一个不存在的端口
servers.add(new Server("localhost", 8888));
lb.addServers(servers);
// 设置 IPing 实现类
lb.setPing(new PingUrl());
// 设置 Ping 时间间隔为 2 秒
lb.setPingInterval(2);
Thread.sleep(6000);
for(Server s : lb.getAllServers()) {
    System.out.println(s.getHostPort() + " 状态：" + s.isAlive());
}
```

代码清单 4-8，使用了代码的方法来设置负载均衡器使用 PingUrl，设置了每隔 2 秒，就向两个服务器请求，PingUrl 实际是使用的是 HttpClient，以上例子中，实际上会请求“http://localhost:8080”与“http://localhost:8888”这两个地址，在运行前先以 8080 端口启动前面章节的服务器，最终效果为 8080 的服务器状态正常，而 8888 的服务器则无法连接，运行代码清单 4-8，可以看到输出如下：

```
localhost:8080 状态：true
localhost:8888 状态：false
```

除了在代码中配置使用 IPing 类外，还可以在配置中设置 IPing 实现类，请见代码清单 4-9。

代码清单 4-9：

codes\04\4.2\first-ribbon-client\src\main\java\org\crazyit\cloud\TestPingUrlConfig.java

```
// 设置请求的服务器
ConfigurationManager.getConfigInstance().setProperty(
        "my-client.ribbon.listOfServers",
        "localhost:8080,localhost:8888");
// 配置 Ping 处理类
ConfigurationManager.getConfigInstance().setProperty(
        "my-client.ribbon.NFLoadBalancerPingClassName",
        PingUrl.class.getName());
// 配置 Ping 时间间隔
ConfigurationManager.getConfigInstance().setProperty(
        "my-client.ribbon.NFLoadBalancerPingInterval",
        2);
// 获取 REST 请求客户端
RestClient client = (RestClient) ClientFactory
        .getNamedClient("my-client");
Thread.sleep(6000);
// 获取全部服务器
List<Server> servers = client.getLoadBalancer().getAllServers();
System.out.println(servers.size());
// 输出状态
for(Server s : servers) {
    System.out.println(s.getHostPort() + " 状态：" + s.isAlive());
}
```

注意代码中的以下两个配置：

- my-client.ribbon.NFLoadBalancerPingClassName：配置 IPing 的实现类。
- my-client.ribbon.NFLoadBalancerPingInterval：配置 Ping 操作的时间间隔。

以上两个配置，同样可以使用在配置文件中。

7.1.5 自定义 Ping

通过前面章节的案例可知，实现自定义 Ping 较为简单，先实现 IPing 接口，然后再通过配置来设定具体的 Ping 实现类，代码清单 4-10 为自定义的 Ping 类。

代码清单 4-10：

codes\04\4.2\first-ribbon-client\src\main\java\org\crazyit\cloud\MyPing.java

```
public class MyPing implements IPing {

    public boolean isAlive(Server server) {
```

```
            System.out.println("这是自定义 Ping 实现类：" + server.getHostPort());
            return true;
        }
    }
```

要使用自定义的 Ping 类，通过修改<client>.<nameSpace>.NFLoadBalancerPingClassName 配置即可，在此不再赘述。

7.1.6 其他配置

本小节主要介绍了 Ribbon 负载均衡器的负载规则以及 Ping，这两部分可以通过配置来实现逻辑的改变，除了这两部分外，还可以使用以下的配置，来改变负载均衡器的其他行为：

- NFLoadBalancerClassName：指定负载均衡器的实现类，可利用该配置，实现自己的负载均衡器。
- NIWSServerListClassName：服务器列表处理类，用来维护服务器列表，Ribbon 已经实现动态服务器列表。
- NIWSServerListFilterClassName：用于处理服务器列表拦截。

8 Spring Cloud 与 RibbonRibbon

本章要点

- Spring Cloud 中使用 Ribbon

Spring Cloud 集成了 Ribbon，结合 Eureka，可实现客户端的负载均衡。我们前面章节所使用的 RestTemplate（被@LoadBalanced 修饰）、还有后面章节的 Feign，都已经拥有负载均衡功能。本小节将以 RestTemplate 为基础，讲述及测试 Eureka 中的 Ribbon 配置。

8.1 准备工作

为了本小节的测试做准备，按顺序进行以下工作：

- 新建 Eureka 服务器端项目，命名为“cloud-server”，端口 8761，代码目录 codes\04\4.4\cloud-server。
- 新建 Eureka 服务提供者项目，命名为“cloud-provider”，代码目录 codes\04\4.4\cloud-provider，该项目主要进行以下工作：
  - 在控制器里面，发布一个 REST 服务，地址为“/person/{personId}”，请求后返回 Person 实例，其中 Person 的 message 为 HTTP 请求的 URL。
  - 服务提供者需要启动两次，因此在控制台中需要输入启动端口。
- 新建 Eureka 服务调用者项目，命名为“cloud-invoker”，对外端口为 9000，代码

目录 codes\04\4.4\cloud-invoker。本例的负载均衡配置主要针对服务调用者。以上项目准备完成并启动后，结构如图 4-2 所示。

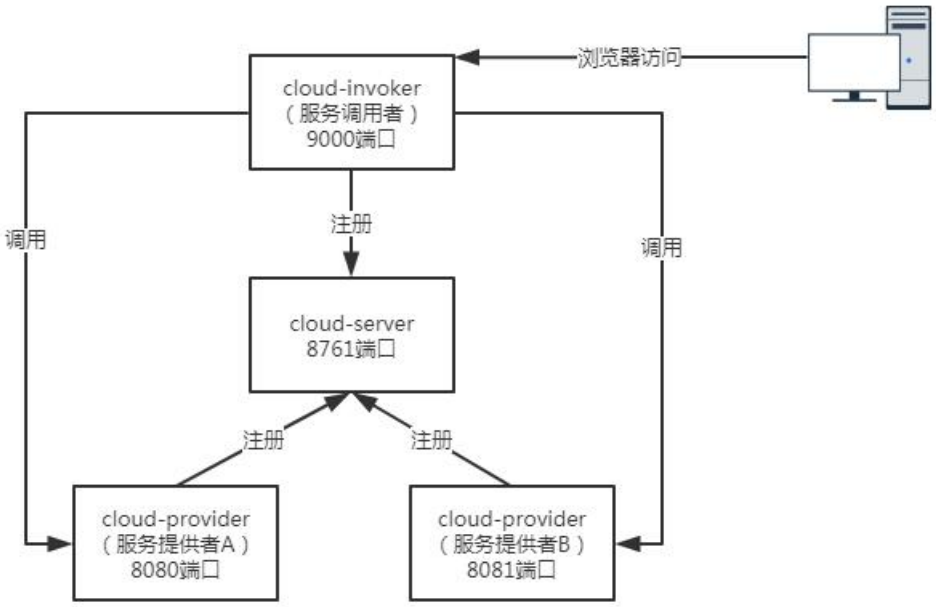

图 4-2 准备的项目结构图

注意：Eureka 相关项目的建立，可参见本书的 3.2 章节。

8.2 使用代码配置 Ribbon

在前面章节讲述了负载规则以及 Ping，在 Spring Cloud 中，可将自定义的负载规则以及 Ping 类，放到服务调用者中，查看效果。新建自定义的 IRule 与 IPing，两个实现类请见代码清单 4-11。

代码清单 4-11：

codes\04\4.4\cloud-invoker\src\main\java\org\crazyit\cloud\MyRule.java

codes\04\4.4\cloud-invoker\src\main\java\org\crazyit\cloud\MyPing.java

```
public class MyRule implements IRule {

    private ILoadBalancer lb;

    public Server choose(Object key) {
        List<Server> servers = lb.getAllServers();
        System.out.println("这是自定义服务器定规则类，输出服务器信息：");
        for(Server s : servers) {
            System.out.println("            " + s.getHostPort());
        }
        return servers.get(0);
    }
    ...省略 setter 和 getter 方法
}
public class MyPing implements IPing {
```

```
    public boolean isAlive(Server server) {
        System.out.println("自定义 Ping 类，服务器信息：" + server.getHostPort());
        return true;
    }
}
```

根据两个自定义的 IRule 和 IPing 类可知，实际上跟 4.3 章节中的自定义实现类似，服务器选择规则中只返回集合中的第一个实例，IPing 实现仅仅是控制输入服务器信息。接下来，新建配置类，返回规则与 Ping 的 Bean，请见代码清单 4-12。

代码清单 4-12：

codes\04\4.4\cloud-invoker\src\main\java\org\crazyit\cloud\config\MyConfig.java

04\4.4\cloud-invoker\src\main\java\org\crazyit\cloud\config\CloudProviderConfig.java

```
public class MyConfig {
    @Bean
    public IRule getRule() {
        return new MyRule();
    }
    @Bean
    public IPing getPing() {
        return new MyPing();
    }
}

@RibbonClient(name="cloud-provider", configuration=MyConfig.class)
public class CloudProviderConfig {

}
```

代码清单 4-12 中，CloudProviderConfig 配置类，使用了@RibbonClient 注解，配置了 RibbonClient 的名称为“cloud-provider”，对应的配置类为“MyConfig”，也就是名称为“cloud-provider”的客户端，将使用 MyRule 与 MyPing 两个类。在服务调用者的控制器中，加入对外服务，服务中调用 RestTemplate，如代码清单 4-13。

代码清单 4-13：

codes\04\4.4\cloud-invoker\src\main\java\org\crazyit\cloud\InvokerController.java

```
@RestController
@Configuration
public class InvokerController {

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

    @RequestMapping(value = "/router", method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON_VALUE)
    public String router() {
        RestTemplate restTpl = getRestTemplate();
        // 根据名称调用服务
        String json = restTpl.getForObject(
                "http://cloud-provider/person/1", String.class);
```

```
        return json;
    }
}
```

以上的控制器中，为 RestTemplate 加入了@LoadBalanced 修饰，与第 3 章类似，在此不再赘述，关于 RestTemplate 的原理，将在本章后面章节讲述。进行以下操作，查看本例效果：

- 启动一个 Eureka 服务器（cloud-server）。
- 启动两次 Eureka 服务提供者（cloud-provider），分别输入 8080 与 8081 端口。
- 启动一个 Eureka 服务调用者（cloud-invoker）。
- 打开浏览器访问 http://localhost:9000/router，可以看到调用服务后返回的 JSON 字符串，不管刷新多少次，最终都只会访问其中一个端口。

8.3 使用配置文件设置 Ribbon

在前面使用 Ribbon 时，可以通过配置来定义各个属性，在使用 Spring Cloud 时，这些属性同样可以配置到 application.yml 中，以下的配置同样生效：

```
cloud-provider:
  ribbon:
    NFLoadBalancerRuleClassName: org.crazyit.cloud.MyRule
    NFLoadBalancerPingClassName: org.crazyit.cloud.MyPing
    listOfServers: http://localhost:8080/,http://localhost:8081/
```

为 cloud-provider 这个客户端，配置了规则处理类、Ping 类以及服务器列表，以同样的方式运行本小节例子，可看到同样的效果，在此不再赘述。

代码与配置文件的方式进行配置，两种方式的效果一致，但对比起来，明显是配置文件的方式更加简便。

注意：本案例的 cloud-invoker 模块中，默认使用了代码的方式来配置 Ribbon，配置文件中的配置已被注释。

8.4 Spring 使用 Ribbon 的 API

Spring Cloud 对 Ribbon 进行封装，例如像负载客户端、负载均衡器等，我们可以直接使用 Spring 的 LoadBalancerClient 来处理请求以及服务选择。代码清单 4-14，在服务器调用者的控制器中使用 LoadBalancerClient。

代码清单 4-14：codes\04\4.4\cloud-invoker\src\main\java\org\crazyit\cloud\InvokerController.java

```
    @Autowired
    private LoadBalancerClient loadBalancer;

    @RequestMapping(value = "/uselb", method = RequestMethod.GET,
                produces = MediaType.APPLICATION_JSON_VALUE)
    public ServiceInstance uselb() {
        // 查找服务器实例
        ServiceInstance si = loadBalancer.choose("cloud-provider");
        return si;
    }
```

除了使用 Spring 封装的负载客户端外，还可以直接使用 Ribbon 的 API，代码 4-15，直接获取 Spring Cloud 默认环境中，各个 Ribbon 的实现类。

代码清单 4-15：codes\04\4.4\cloud-invoker\src\main\java\org\crazyit\cloud\InvokerController.java

```
@Autowired
private SpringClientFactory factory;

@RequestMapping(value = "/defaultValue", method = RequestMethod.GET,
        produces = MediaType.APPLICATION_JSON_VALUE)
public String defaultValue() {
    System.out.println("==== 输出默认配置：");
    // 获取默认的配置
    ZoneAwareLoadBalancer alb = (ZoneAwareLoadBalancer) factory
            .getLoadBalancer("default");
    System.out.println("      IClientConfig: "
            + factory.getLoadBalancer("default").getClass()
                    .getName());
    System.out.println("      IRule: " + alb.getRule().getClass().getName());
    System.out.println("      IPing: " + alb.getPing().getClass().getName());
    System.out.println("      ServerList: "
            + alb.getServerListImpl().getClass().getName());
    System.out.println("      ServerListFilter: "
            + alb.getFilter().getClass().getName());
    System.out.println("      ILoadBalancer: " + alb.getClass().getName());
    System.out.println("      PingInterval: " + alb.getPingInterval());
    System.out.println("==== 输出 cloud-provider 配置：");
    // 获取 cloud-provider 的配置
    ZoneAwareLoadBalancer alb2 = (ZoneAwareLoadBalancer) factory
            .getLoadBalancer("cloud-provider");
    System.out.println("      IClientConfig: "
            + factory.getLoadBalancer("cloud-provider").getClass()
                    .getName());
    System.out.println("      IRule: " + alb2.getRule().getClass().getName());
    System.out.println("      IPing: " + alb2.getPing().getClass().getName());
    System.out.println("      ServerList: "
            + alb2.getServerListImpl().getClass().getName());
    System.out.println("      ServerListFilter: "
            + alb2.getFilter().getClass().getName());
    System.out.println("      ILoadBalancer: " + alb2.getClass().getName());
    System.out.println("      PingInterval: " + alb2.getPingInterval());
    return "";
}
```

代码中使用了 SpringClientFactory，通过该实例，可获取各个默认的实现类以及配置，分别输出了默认配置以及“cloud-provider”配置。运行代码清单 4-15，浏览器中访问地址 http://localhost:8080/defaultValue，可看到控制台输出如下：

```
==== 输出默认配置：
    IClientConfig: com.netflix.loadbalancer.ZoneAwareLoadBalancer
    IRule: com.netflix.loadbalancer.ZoneAvoidanceRule
    IPing: com.netflix.niws.loadbalancer.NIWSDiscoveryPing
    ServerList: org.springframework.cloud.netflix.ribbon.eureka.DomainExtractingServerList
    ServerListFilter: org.springframework.cloud.netflix.ribbon.ZonePreferenceServerListFilter
```

```
    ILoadBalancer: com.netflix.loadbalancer.ZoneAwareLoadBalancer
    PingInterval: 30
==== 输出 cloud-provider 配置：
    IClientConfig: com.netflix.loadbalancer.ZoneAwareLoadBalancer
    IRule: org.crazyit.cloud.MyRule
    IPing: org.crazyit.cloud.MyPing
    ServerList: org.springframework.cloud.netflix.ribbon.eureka.DomainExtractingServerList
    ServerListFilter: org.springframework.cloud.netflix.ribbon.ZonePreferenceServerListFilter
    ILoadBalancer: com.netflix.loadbalancer.ZoneAwareLoadBalancer
    PingInterval: 30
```

根据输出可知，cloud-provider 客户端使用的负载规则类以及 Ping 类，是我们自定义的实现类。

一般情况下，Spring 已经帮我们封装好了 Ribbon，我们只需要直接调用 RestTemplate 等 API 来访问服务即可。

9 RestTemplate 负载均衡原理

本文要点

- RestTemplate 的负载均衡原理

9.1 @LoadBalanced 注解概述

RestTemplate 本是 spring-web 项目中的一个 REST 客户端访问类，它遵循 REST 的设计原则，提供简单的 API 让调用去访问 HTTP 服务器。RestTemplate 本身不具有负载均衡的功能，该类也与 Spring Cloud 没有关系，但为何加入@LoadBalanced 注解后，一个 RestTemplate 实例就具有负载均衡的功能呢？实际上这要得益于 RestTemplate 的拦截器功能。

在 Spring Cloud 中，使用@LoadBalanced 修饰的 RestTemplate，在 Spring 容器启动时，会为这些被修饰过的 RestTemplate 添加拦截器，拦截器中使用了 LoadBalancerClient 来处理请求，LoadBalancerClient 本来就是 Spring 封装的负载均衡客户端，通过这样间接处理，使得 RestTemplate 就拥有了负载均衡的功能。

本小节将模仿拦截器机制，带领大家实现一个简单的 RestTemplate，以便让大家更了解 @LoadBalanced 以及 RestTemplate 的原理。本小节的案例只依赖了 spring-boot-starter-web 模块：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>1.5.4.RELEASE</version>
</dependency>
```

9.2 编写自定义注解以及拦截器

先模仿@LoadBalanced 注解，编写一个自定义注解，请见代码清单 4-16。

代码清单 4-1：

\codes\04\4.5\rest-template-test\src\main\java\org\crazyit\cloud\MyLoadBalanced.java

```
@Target({ ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface MyLoadBalanced {

}
```

注意 MyLoadBalanced 注解中使用了@Qualifier 限定注解，接下来编写自定义的拦截器，请见代码清单 4-17。

代码清单 4-17：

codes\04\4.5\rest-template-test\src\main\java\org\crazyit\cloud\MyInterceptor.java

```
public class MyInterceptor implements ClientHttpRequestInterceptor {

    public ClientHttpResponse intercept(HttpRequest request, byte[] body,
            ClientHttpRequestExecution execution) throws IOException {
        System.out.println("==========  这是自定义拦截器实现");
        System.out.println("                原来的 URI：" + request.getURI());
        // 换成新的请求对象（更换 URI）
        MyHttpRequest newRequest = new MyHttpRequest(request);
        System.out.println("                拦截后新的 URI：" + request.getURI());
        return execution.execute(newRequest, body);
    }
}
```

在自定义拦截器 MyInterceptor 中，实现了 intercept 方法，该方法会将原来的 HttpRequest 对象，转换为我们自定义的 MyHttpRequest，MyHttpRequest 是一个自定义的请求类，实现如下请见代码清单 4-18。

代码清单 4-18：

codes\04\4.5\rest-template-test\src\main\java\org\crazyit\cloud\MyHttpRequest.java

```
public class MyHttpRequest implements HttpRequest {

    private HttpRequest sourceRequest;

    public MyHttpRequest(HttpRequest sourceRequest) {
        this.sourceRequest = sourceRequest;
    }

    public HttpHeaders getHeaders() {
        return sourceRequest.getHeaders();
    }

    public HttpMethod getMethod() {
        return sourceRequest.getMethod();
    }
```

```
    /**
     * 将 URI 转换
     */
    public URI getURI() {
        try {
            String oldUri = sourceRequest.getURI().toString();
            URI newUri = new URI("http://localhost:8080/hello");
            return newUri;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return sourceRequest.getURI();
    }
}
```

MyHttpRequest 类中，会将原来请求的 URI 进行改写，只要使用了这个对象，所有的请求都会被转发到 http://localhost:8080/hello 这个地址。**Spring Cloud 在对 RestTemplate 进行拦截的时候，也做了同样的事情，只不过并没有像我们这样固定了 URI，而是对“源请求”进行了更加灵活的处理。**接下来使用自定义注解以及拦截器。

9.3 使用自定义拦截器以及注解

编写一个 Spring 的配置类，在初始化的 bean 中为容器中的 RestTemplate 实例设置自定义拦截器，本例的 Spring 自动配置类请见代码清单 4-19。

代码清单 4-19：

codes\04\4.5\rest-template-test\src\main\java\org\crazyit\cloud\MyAutoConfiguration.java

```
@Configuration
public class MyAutoConfiguration {

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

    @Bean
    public SmartInitializingSingleton myLoadBalancedRestTemplateInitializer() {
        System.out.println("====   这个 Bean 将在容器初始化时创建      =====");
        return new SmartInitializingSingleton() {

            public void afterSingletonsInstantiated() {
                for(RestTemplate tpl : myTemplates) {
                    // 创建一个自定义的拦截器实例
                    MyInterceptor mi = new MyInterceptor();
                    // 获取 RestTemplate 原来的拦截器
                    List list = new ArrayList(tpl.getInterceptors());
                    // 添加到拦截器集合
                    list.add(mi);
                    // 将新的拦截器集合设置到 RestTemplate 实例
                    tpl.setInterceptors(list);
                }
```

```
                }
            };
        }
    }
```

配置类中，定义了 RestTemplate 实例的集合，并且使用了@MyLoadBalanced 以及@Autowired 注解进行修饰，@MyLoadBalanced 中含有@Qualifier 注解，简单来说，就是 Spring 容器中，使用了@MyLoadBalanced 修饰的 RestTemplate 实例，将会被加入到配置类的 RestTemplate 集合中。

在容器初始化时，会调用 myLoadBalancedRestTemplateInitializer 方法来创建 Bean，该 Bean 在初始化完成后，会遍历 RestTemplate 集合并为它们设置“自定义拦截器”，请见代码清单 4-19 中的粗体代码。下面在控制器中使用@MyLoadBalanced 来修饰调用者的 RestTemplate。

9.4 控制器中使用 RestTemplate

控制器代码请见代码清单 4-20。

代码清单 4-20：

codes\04\4.5\rest-template-test\src\main\java\org\crazyit\cloud\InvokerController.java

```
@RestController
@Configuration
public class InvokerController {

    @Bean
    @MyLoadBalanced
    public RestTemplate getMyRestTemplate() {
        return new RestTemplate();
    }

    /**
     * 浏览器访问的请求
     */
    @RequestMapping(value = "/router", method = RequestMethod.GET,
                produces = MediaType.APPLICATION_JSON_VALUE)
    public String router() {
        RestTemplate restTpl = getMyRestTemplate();
        // 根据名称来调用服务，这个 URI 会被拦截器所置换
        String json = restTpl.getForObject("http://my-server/hello", String.class);
        return json;
    }

    /**
     * 最终的请求都会转到这个服务
     */
    @RequestMapping(value = "/hello", method = RequestMethod.GET)public String hello() {
        return "Hello World";
    }
}
```

注意控制器的 hello 方法，前面实现的拦截器，会将全部请求都转到这个服务中。控制器的 RestTemplate，使用了@MyLoadBalanced 注解进行修饰，熟悉前面使用 RestTemplate

的读者可发现，我们实现的注解，与 Spring 提供的@LoadBalanced 注解使用方法一致。控制器的 router 方法中，使用这个被拦截过的 RestTemplate 发送请求。

打开浏览器，访问 http://localhost:8080/router，可以看到实际上调用了 hello 服务，在访问该地址时，控制台输出如下：

```
==========  这是自定义拦截器实现
            原来的 URI：http://my-server/hello
            拦截后新的 URI：http://localhost:8080/hello
```

Spring Cloud 对 RestTemplate 的拦截实现更加复杂，并且在拦截器中，使用 LoadBalancerClient 来实现请求的负载均衡功能，我们在实际环境中，并不需要实现自定义注解以及拦截器，用 Spring 提供的现成 API 即可，本小节目的是展示 RestTemplate 的原理。

本文要点

- REST 客户端

Spring Cloud 集群中，各个角色的通信基于 REST 服务，因此在调用服务时，就不可避免的需要使用 REST 服务的请求客户端。前面的章节中使用了 Spring 自带的 RestTemplate，RestTemplate 使用的是 HttpClient 发送请求。本章中，将介绍另一个 REST 客户端：Feign。

10 REST 客户端 Feign 介绍

在学习 Feign 前，先了解 REST 客户端，本小节将简单地讲述 Apache CXF 与 Restlet 这两款 Web Service 框架，并使用这两个框架来编写 REST 客户端，最后再编写一个 Feign 的 Hello World 例子。通过此过程，让大家可以对 Feign 有一个初步的印象。如已经掌握这两个 REST 框架，可直接到后面章节学习 Feign。

本章的各个客户端，将会访问 8080 端口的“/person/{personId}”和“/hello”这两个服务中的一个，服务端项目使用“spring-boot-starter-web”进行搭建，本小节对应的服务端项目目录为：codes\05\5.1\rest-server。

10.1 使用 CXF 调用 REST 服务

CXF 是目前一个较为流行的 Web Service 框架，是 Apache 下的一个开源项目。使用 CXF 可以发布和调用各种协议的服务，包括 SOAP 协议、XML/HTTP 等，当前 CXF 已经对 REST 风格的 Web Service 提供支持，可以发布或调用 REST 风格的 Web Service。由于 CXF 可以与 Spring 进行整合使用并且配置简单，因此得到许多开发者的青睐，而笔者以往所在公司的大部分项目，均使用 CXF 来发布和调用 Web Service，本章所使用的 CXF 版本为 3.1.10，Maven 中加入以下依赖：

```
        <dependency>
```

```
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-core</artifactId>
            <version>3.1.10</version>
        </dependency>
        <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-rs-client</artifactId>
            <version>3.1.10</version>
        </dependency>
```

编写代码请求/person/{personId}服务，请见代码清单 5-1。

代码清单 5-1：codes\05\5.1\rest-client\src\main\java\org\crazyit\cloud\CxfClient.java

```
public class CxfClient {

    public static void main(String[] args) throws Exception {
        // 创建 WebClient
        WebClient client = WebClient.create("http://localhost:8080/person/1");
        // 获取响应
        Response response = client.get();
        // 获取响应内容
        InputStream ent = (InputStream) response.getEntity();
        String content = IOUtils.readStringFromStream(ent);
        // 输出字符串
        System.out.println(content);
    }
}
```

客户端中，使用了 WebClient 类发送请求，获取响应后读取输入流，获取服务返回的 JSON 字符串。，运行代码清单 5-1，可看到返回的信息。

10.2 使用 Restlet 调用 REST 服务

Restlet 是一个轻量级的 REST 框架，使用它可以发布和调用 REST 风格的 Web Service。本小节例子所使用的版本为 2.3.10，Maven 依赖如下：

```
        <dependency>
            <groupId>org.restlet.jee</groupId>
            <artifactId>org.restlet</artifactId>
            <version>2.3.10</version>
        </dependency>
        <dependency>
            <groupId>org.restlet.jee</groupId>
            <artifactId>org.restlet.ext.jackson</artifactId>
            <version>2.3.10</version>
        </dependency>
```

客户端实现请见代码清单 5-2。

代码清单 5-2：codes\05\5.1\rest-client\src\main\java\org\crazyit\cloud\RestletClient.java

```
public class RestletClient {

    public static void main(String[] args) throws Exception {
        ClientResource client = new ClientResource(
```

```
                "http://localhost:8080/person/1");
        // 调用 get 方法，服务端发布的是 GET
        Representation response = client.get(MediaType.APPLICATION_JSON);
        // 创建 JacksonRepresentation 实例，将响应转换为 Map
        JacksonRepresentation jr = new JacksonRepresentation(response,
                HashMap.class);
        // 获取转换后的 Map 对象
        Map result = (HashMap) jr.getObject();
        // 输出结果
        System.out.println(result.get("id") + "-" + result.get("name") + "-"
                + result.get("age") + "-" + result.get("message"));
    }
}
```

代码清单 5-2 中使用 Restlet 的 API 较为简单，在此不过多赘述，但需要注意的是，在 Maven 中使用 Restlet，要额外配置仓库地址，笔者成书时 Apache 官方仓库中，并没有 Restlet 的包。在项目的 pom.xml 文件中增加以下配置：

```
<repositories>
    <repository>
        <id>maven-restlet</id>
        <name>Restlet repository</name>
        <url>http://maven.restlet.org</url>
    </repository>
</repositories>
```

10.3 Feign 框架介绍

Feign 是一个 Github 上一个开源项目，目的是为了简化 Web Service 客户端的开发。在使用 Feign 时，可以使用注解来修饰接口，被注解修饰的接口具有访问 Web Service 的能力，这些注解中既包括了 Feign 自带的注解，也支持使用第三方的注解。除此之外，Feign 还支持插件式的编码器和解码器，使用者可以通过该特性，对请求和响应进行不同的封装与解析。

Spring Cloud 将 Feign 集成到 netflix 项目中，当与 Eureka、Ribbon 集成时，Feign 就具有负载均衡的功能。Feign 本身在使用上的简便性，加上与 Spring Cloud 的高度整合，使用该框架在 Spring Cloud 中调用集群服务，将会大大降低开发的工作量。

10.4 第一个 Feign 程序

先使用 Feign 编写一个 Hello World 的客户端，访问服务端的“/hello”服务，得到返回的字符串。当前 Spring Cloud 所依赖的 Feign 版本为 9.5.0，本章案例中的 Feign 也使用该版本。建立名称为“feign-client”的 Maven 项目，加入以下依赖：

```
        <dependency>
            <groupId>io.github.openfeign</groupId>
            <artifactId>feign-core</artifactId>
            <version>9.5.0</version>
        </dependency>
        <dependency>
            <groupId>io.github.openfeign</groupId>
```

```
            <artifactId>feign-gson</artifactId>
            <version>9.5.0</version>
        </dependency>
```

新建接口 HelloClient，请见代码清单 5-3。

代码清单 5-3：codes\05\5.1\feign-client\src\main\java\org\crazyit\cloud\HelloClient.java

```
public interface HelloClient {

    @RequestLine("GET /hello")
    String sayHello();
}
```

HelloClient 表示一个服务接口，接口的“sayHello”方法中，使用了@RequestLine 注解，表示使用 GET 方法，向“/hello”发送请求。接下来编写客户端的运行类，请见代码清单 5-4。

代码清单 5-4：codes\05\5.1\feign-client\src\main\java\org\crazyit\cloud\HelloMain.java

```
public class HelloMain {

    public static void main(String[] args) {
        // 调用 Hello 接口
        HelloClient hello = Feign.builder().target(HelloClient.class,
                "http://localhost:8080/");
        System.out.println(hello.sayHello());
    }
}
```

运行类中，使用 Feign 创建 HelloClient 接口的实例，最后调用接口定义的方法。运行代码清单 5-4，可以看到返回的“Hello World”字符串，可见接口已经被调用。熟悉 AOP 的朋友大概已经猜到，Feign 实际上会帮我们动态生成代理类。Feign 使用的是 JDK 的动态代理，生成的代理类，会将请求的信息封装，交给 feign.Client 接口发送请求，而该接口的默认实现类，最终会使用 java.net.HttpURLConnection 来发送 HTTP 请求。

10.5 请求参数与返回对象

本案例中有两个服务，另外一个地址为“/person/{personId}”，需要传入参数并且返回 JSON 字符串，编写第二个 Feign 客户端，调用该服务。新建 PersonClient 服务类，定义调用接口并添加注解，请见代码清单 5-5。

代码清单 5-5：codes\05\5.1\feign-client\src\main\java\org\crazyit\cloud\PersonClient.java

```
public interface PersonClient {

    @RequestLine("GET /person/{personId}")
    Person findById(@Param("personId") Integer personId);

    @Data // 为所有属性加上 setter 和 getter 等方法
    class Person {
        Integer id;
        String name;
        Integer age;
        String message;
```

```
    }
}
```

定义的接口名称为“findById”，参数为“personId”。需要注意的是，由于会返回 Person 实例，我们在接口中定义了一个 Person 的类，为了减少代码量，使用了 Lombok 项目，使用了该项目的@Data 注解。要使用 Lombok，需要添加以下 Maven 依赖：

```
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.16.18</version>
        </dependency>
```

准备好提供服务的客户端类后，再编写运行类。运行类基本上与前面的 HelloWorld 类似，请见代码清单 5-6。

代码清单 5-6：codes\05\5.1\feign-client\src\main\java\org\crazyit\cloud\PersonMain.java

```
public class PersonMain {

    public static void main(String[] args) {
        PersonClient personService = Feign.builder()
                .decoder(new GsonDecoder())
                .target(PersonClient.class, "http://localhost:8080/");
        Person person = personService.findById(2);
        System.out.println(person.id);
        System.out.println(person.name);
        System.out.println(person.age);
        System.out.println(person.message);
    }
}
```

调用 Person 服务的运行类中，添加了解码器的配置，GsonDecoder 会将返回的 JSON 字符串，转换为接口方法返回的对象，关于解码器等内容，将在后面章节中讲述。运行代码清单 5-6，可以看到最终的输出。

本小节使用了 CXF、Restlet、Feign 来编写 REST 客户端，在编写客户端的过程中，可以看到 Feign 的代码更加“面向对象”，至于是否更加简洁，则见仁见智。下面的章节，将深入了解 Feign 的各项功能。

本文要点

- Feign 编码器与解码器

11 Feign 的编码器与解码器

本小节所有的案例都是单独使用 Feign，Feign 在 Spring Cloud 的使用将在后面章节讲述，请读者注意该细节。

5.2.1 编码器

向服务发送请求的过程中，有些情况需要对请求的内容进行处理。例如服务端发布的服务接收的是 JSON 格式参数，而客户端使用的是对象，这种情况就可以使用编码器，将对象转换为 JSON 字符串。

为服务端编写一个 REST 服务，处理 POST 请求，请见代码清单 5-7。

代码清单 5-7：codes\05\5.1\rest-server\src\main\java\org\crazyit\cloud\MyController.java

```
@RequestMapping(value = "/person/create", method = RequestMethod.POST,
        consumes = MediaType.APPLICATION_JSON_VALUE)
public String createPerson(@RequestBody Person person) {
    System.out.println(person.getName() + "-" + person.getAge());
    return "Success, Person Id: " + person.getId();
}
```

控制器中，发布了一个“/person/create”的服务，需要传入 JSON 格式的请求参数。在客户端中，要调用该服务，先编写接口，再使用注解进行修饰，请见代码清单 5-8。

代码清单 5-8：codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\PersonClient.java

```
public interface PersonClient {

    @RequestLine("POST /person/create")
    @Headers("Content-Type: application/json")
    String createPerson(Person person);

    @Data
    class Person {
        Integer id;
        String name;
        Integer age;
        String message;
    }
}
```

注意在客户端的服务接口中，使用了@Headers 注解，声明请求的内容类型为 JSON，接下来再编写运行类，如代码清单 5-9。

代码清单 5-9：codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\EncoderTest.java

```
public class EncoderTest {

    public static void main(String[] args) {
        // 获取服务接口
        PersonClient personClient = Feign.builder()
                .encoder(new GsonEncoder())
                .target(PersonClient.class, "http://localhost:8080/");
        // 创建参数的实例
        Person person = new Person();
        person.id = 1;
        person.name = "Angus";
        person.age = 30;
```

```
        String response = personClient.createPerson(person);
        System.out.println(response);
    }
}
```

运行类中，在创建服务接口实例时，使用了 encoder 方法来指定编码器，本案例使用了 Feign 提供的 GsonEncoder 类，该类会在发送请求过程中，将请求的对象转换为 JSON 字符串。Feign 支持插件式的编码器，如果 Feign 提供的编码器无法满足要求，还可以使用自定义的编码器，这部分内容在后面章节讲述。启动服务，运行代码清单 5-9，可看到服务已经调用成功，运行后输出如下：

```
Success, Person Id: 1
```

5.2.2 解码器

编码器是对请求的内容进行处理，解码器则会对服务响应的内容进行处理，例如解析响应的 JSON 或者 XML 字符串，转换为我们所需要的对象，在代码中通过以下的代码片断设置解码器：

```
        PersonClient personService = Feign.builder()
                .decoder(new GsonDecoder())
                .target(PersonClient.class, "http://localhost:8080/");
```

在前面章节中，我们已经使用过 GsonDecoder 解码器，在此不再作赘述。

5.2.3 XML 的编码与解码

除了支持 JSON 的处理外，Feign 还为 XML 的处理了提供编码器与解码器，可以使用 JAXBEncoder 与 JAXBDecoder 来进行编码与解码。为服务端添加发布 XML 的接口，请见代码清单 5-10。

代码清单 5-10：codes\05\5.1\rest-server\src\main\java\org\crazyit\cloud\MyController.java

```
    @RequestMapping(value = "/person/createXML", method = RequestMethod.POST,
            consumes = MediaType.APPLICATION_XML_VALUE,
            produces = MediaType.APPLICATION_XML_VALUE)    public String createXMLPerson(@RequestBody Person person) {
        System.out.println(person.getName() + "-" + person.getId());
        return "<result><message>success</message></result>";
    }
```

在服务端发布的服务方法中，声明了传入的参数为 XML。需要注意的是，服务端项目“rest-server”使用的是“spring-boot-starter-web”进行构建，默认情况下不支持 XML 接口，调用接口时会得到以下异常信息：

```
{"timestamp":1502705981406,"status":415,"error":"Unsupported Media Type","exception":"org.springframework.web.HttpMediaTypeNotSupportedException","message":"Content type 'application/xml;charset=UTF-8' not supported","path":"/person/createXML"}
```

为服务端的 pom.xml 加入以下依赖即可解决该问题：

```
        <dependency>
            <groupId>com.fasterxml.jackson.jaxrs</groupId>
            <artifactId>jackson-jaxrs-xml-provider</artifactId>
            <version>2.9.0</version>
```

```
        </dependency>
```

编写客户端时，先定义好服务接口以及对象，接口请见代码清单 5-11。

代码清单 5-11：codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\PersonClient.java

```
public interface PersonClient {

    @RequestLine("POST /person/createXML")
    @Headers("Content-Type: application/xml")
    Result createPersonXML(Person person);

    @Data
    @XmlRootElement
    class Person {
        @XmlElement
        Integer id;
        @XmlElement
        String name;
        @XmlElement
        Integer age;
        @XmlElement
        String message;
    }

    @Data
    @XmlRootElement
    class Result {
        @XmlElement
        String message;
    }
}
```

在接口中，定义了“Content-Type”为 XML，使用了 JAXB 的相关注解来修饰 Person 与 Result。接下来，只需要调用 createPersonXML 方法即可请求服务，请见代码清单 5-12。

代码清单 5-12：codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\XMLTest.java

```
public class XMLTest {

    public static void main(String[] args) {
        JAXBContextFactory jaxbFactory = new JAXBContextFactory.Builder().build();
        // 获取服务接口
        PersonClient personClient = Feign.builder()
                .encoder(new JAXBEncoder(jaxbFactory))
                .decoder(new JAXBDecoder(jaxbFactory))
                .target(PersonClient.class, "http://localhost:8080/");
        // 构建参数
        Person person = new Person();
        person.id = 1;
        person.name = "Angus";
        person.age = 30;
        // 调用接口并返回结果
        Result result = personClient.createPersonXML(person);
        System.out.println(result.message);
    }
}
```

本小节的请求有一点特殊，请求服务时传入参数为 XML、返回的结果也是 XML，目的是为了能使编码与解码一起使用。开启服务，运行代码清单 5-12，可以看到服务端与客户端的输出。

5.2.4 自定义编码器与解码器

根据前面的两小节可知，Feign 的插件式编码器与解码器，可以对请求以及结果进行处理，如果对于一些特殊的要求，可以使用自定义的编码器与解码器。实现自定义编码器，需要实现 Encoder 接口的 encode 方法，而对于解码器，则要实现 Decoder 接口的 decode 方法，例如以下的代码片断：

```
public class MyEncoder implements Encoder {

    public void encode(Object object, Type bodyType, RequestTemplate template)
            throws EncodeException {
        // 实现自己的 Encode 逻辑
    }
}
```

在使用时，调用 Feign 的 API 来设置编码器或者解码器即可，实现较为简单，在此不再赘述。

12 自定义 Feign 客户端

Feign 使用一个 Client 接口来发送请求，默认情况下，使用 HttpURLConnection 连接 HTTP 服务。与前面的编码器类似，客户端也采用了插件式设计，也就是说，我们可以实现自己的客户端。本小节将使用 HttpClient 来实现一个简单的 Feign 客户端。为 pom.xml 加入 HttpClient 的依赖：

```
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.5.2</version>
</dependency>
```

新建 feign.Client 接口的实现类，具体实现请见代码清单 5-13。

代码清单 5-13：codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\MyFeignClient.java

```
public class MyFeignClient implements Client {

    public Response execute(Request request, Options options)
            throws IOException {
        System.out.println("====  这是自定义的 Feign 客户端");
        try {
            // 创建一个默认的客户端
            CloseableHttpClient httpclient = HttpClients.createDefault();
            // 获取调用的 HTTP 方法
```

```
                final String method = request.method();
                // 创建一个 HttpClient 的 HttpRequest
                HttpRequestBase httpRequest = new HttpRequestBase() {
                    public String getMethod() {
                        return method;
                    }
                };
                // 设置请求地址
                httpRequest.setURI(new URI(request.url()));
                // 执行请求，获取响应
                HttpResponse httpResponse = httpclient.execute(httpRequest);
                // 获取响应的主体内容
                byte[] body = EntityUtils.toByteArray(httpResponse.getEntity());
                // 将 HttpClient 的响应对象转换为 Feign 的 Response
                Response response = Response.builder()
                        .body(body)
                        .headers(new HashMap<String, Collection<String>>())
                        .status(httpResponse.getStatusLine().getStatusCode())
                        .build();
                return response;
            } catch (Exception e) {
                throw new IOException(e);
            }
        }
    }
```

简单讲一下自定义 Feign 客户端的实现过程，在实现 execute 方法时，将 Feign 的 Request 实例，转换为 HttpClient 的 HttpRequestBase，再使用 CloseableHttpClient 来执行请求，得到响应的 HttpResponse 实例后，再转换为 Feign 的 Reponse 实例返回。不仅我们实现的客户端，包括 Feign 自定的客户端以及其他扩展的客户端，实际上就是一个对象转换的过程。在运行类中直接使用我们的自定义客户端，请见代码清单 5-14。

代码清单 5-14：codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\MyClientTest.java

```
public class MyClientTest {

    public static void main(String[] args) {
        // 获取服务接口
        PersonClient personClient = Feign.builder()
                .encoder(new GsonEncoder())
                .client(new MyFeignClient())
                .target(PersonClient.class, "http://localhost:8080/");
        // 请求 Hello World 接口
        String result = personClient.sayHello();
        System.out.println("      接口响应内容：" + result);
    }
}
```

运行代码清单 5-14，输出如下：

```
====  这是自定义的 Feign 客户端
    接口响应内容：Hello World
```

注意：在本例的实现中，笔者简化了实现，自定义的客户端中并没有转换请求头等信息，因此使用本例的客户端，无法请求其他格式的服务。

虽然 Feign 也有 HttpClient 的实现，但本例的目的主要是向大家展示 Feign 客户的原理。

举一反三，如果我们实现一个客户端，在实现中调用 Ribbon 的 API，来实现负载均衡的功能，是完全可以实现的。幸运的是，Feign 已经帮我们实现了 RibbonClient，可以直接使用，更进一步，Spring Cloud 也实现自己的 Client，我们将在后面章节中讲述。

13 Feign 第三方注解与注解翻译器

使用第三方注解

根据前面章节可知，通过注解修改的接口方法，可以让接口方法获得访问服务的能力。除了 Feign 自带的方法外，还可以使用第三方的注解。如果想使用 JAXRS 规范的注解，可以使用 Feign 的“feign-jaxrs”模块，在 pom.xml 中加入以下依赖即可：

```
<!-- Feign 对 JAXRS 的支持 -->
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-jaxrs</artifactId>
    <version>9.5.0</version>
</dependency>
<!-- JAXRS -->
<dependency>
    <groupId>javax.ws.rs</groupId>
    <artifactId>jsr311-api</artifactId>
    <version>1.1.1</version>
</dependency>
```

在使用注解修饰接口时，可以直接使用@GET、@Path 等注解，例如想要使用 GET 方法调用“/hello”服务，可以定义以下接口：

```
@GET @Path("/hello")
String rsHello();
```

以上修饰接口的，实际上就等价于“@RequestLine("GET /hello")”。为了让 Feign 知道这些注解的作用，需要在创建服务客户端时，调用 contract 方法来设置 JAXRS 注解的解析类，请见以下代码：

```
RSClient rsClient = Feign.builder()
        .contract(new JAXRSContract())
        .target(RSClient.class, "http://localhost:8080/");
```

设置了 JAXRSContract 类后，Feign 就知道如何处理 JAXRS 的相关注解，下一小节，将讲解 Feign 是如何处理第三方注解的。

Feign 解析第三方注解

根据前一小节可知，设置了 JAXRSContract 后，Feign 就知道如何处理接口中的 JAXRS 注解。JAXRSContract 继承了 BaseContract 类，BaseContract 类实现了 Contract 接口，简单的说，一个 Contract 就相当于一个**翻译器**，Feign 本身并不知道这些第三方注解的含义，

而通过实现一个翻译器（Contract）来告诉 Feign，这些注解是做什么的。

为了让读者能够了解其中的原理，本小节将使用一个自定义注解，并且翻译给 Feign，让其去使用。代码清单 5-15 为自定义注解以及客户端接口的代码。

代码清单 5-15：

codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\contract\MyUrl.java

codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\contract\HelloClient.java

```
@Target(METHOD)
@Retention(RUNTIME)
public @interface MyUrl {

    // 定义 url 与 method 属性
    String url();
    String method();
}

public interface HelloClient {

    @MyUrl(method = "GET", url = "/hello")
    String myHello();
}
```

接下来，就要将 MyRrl 注解的作用告诉 Feign，新建 Contract 继承 BaseContract 类，实现请见代码清单 5-16。

代码清单 5-16：

codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\contract\MyContract.java

```
public class MyContract extends Contract.BaseContract {

    @Override
    protected void processAnnotationOnClass(MethodMetadata data, Class<?> clz) {

    }

    /**
     * 用于处理方法级的注解
     */
    protected void processAnnotationOnMethod(MethodMetadata data,
            Annotation annotation, Method method) {
        // 是 MyUrl 注解才进行处理
        if(MyUrl.class.isInstance(annotation)) {
            // 获取注解的实例
            MyUrl myUrlAnn = method.getAnnotation(MyUrl.class);
            // 获取配置的 HTTP 方法
            String httpMethod = myUrlAnn.method();
            // 获取服务的 url
            String url = myUrlAnn.url();
            // 将值设置到模板中
            data.template().method(httpMethod);
            data.template().append(url);
        }
    }

    @Override
```

```
    protected boolean processAnnotationsOnParameter(MethodMetadata data,
            Annotation[] annotations, int paramIndex) {
        return false;
    }
}
```

在 MyContract 类中，需要实现三个方法，分别是处理类注解、处理方法注解、处理参数注解的方法，由于我们只定了一个方法注解 @MyRul，因此实现 processAnnotationOnMethod 即可。

在 processAnnotationOnMethod 方法中，通过 Method 的 getAnnotation 获取 MyUrl 的实例，将 MyUrl 的 url、method 属性分别设置到 Feign 的模板中。在创建客户端时，再调用 contract 方法即可，请见代码清单 5-17。

代码清单 5-17：codes\05\5.2\feign-use\src\main\java\org\crazyit\feign\contract\ContractTest.java

```
public class ContractTest {

    public static void main(String[] args) {
        // 获取服务接口
        HelloClient helloClient = Feign.builder()
                .contract(new MyContract())
                .target(HelloClient.class, "http://localhost:8080/");
        // 请求 Hello World 接口
        String result = helloClient.myHello();
        System.out.println("      接口响应内容：" + result);
    }
}
```

运行代码清单 5-18，可看到控制台输出如下：

```
接口响应内容：Hello World
```

由本例可知，一个 Contract 实际上承担的是一个翻译的作用，将第三方（或者自定义）注解的作用告诉 Feign。在 Spring Cloud 中，也实现了 Spring 的 Contract，可以在接口中使用 @RequestMapping 注解，读者在学习 Spring Cloud 整合 Feign 时，见到使用 @RequestMapping 修饰的接口，就可以明白其中的原理。

14 Spring Cloud 整合 Feign

前面讲解了 Feign 的使用，在了解如何单独使用 Feign 后，再学习 Spring Cloud 中使用 Feign，将会有非常大的帮助。虽然 Spring Cloud 对 Feign 进行了封装，但万变不离其宗，只要了解其内在原理，使用起来就可以得心应手。

在开始本小节前，先准备 Spring Cloud 的测试项目。测试案例主要有以下三个项目：

- spring-feign-server：Eureka 服务器端项目，端口为 8761，代码目录为 codes\05\5.3\spring-feign-server。
- spring-feign-provider：服务提供者，代码目录为 codes\05\5.3\spring-feign-provider，该项目可以在控制台中根据输入的端口号启动

多个实例，本小节启动 8080 与 8081 这两个端口，该项目提供以下两个 REST 服务：

- 第一个地址为“/person/{personId}”，请求后返回 Person 实例，Person 的 message 属性为 HTTP 请求的 URL。
- 第二个地址为“/hello”的服务，返回“Hello World”字符串。

- spring-feign-invoker：服务调用者项目，对外端口为 9000，代码目录 codes\05\5.3\spring-feign-invoker，本小节例子主要在该项目下使用 Feign。

Spring Cloud 整合 Feign

为服务调用者（spring-feign-invoker）的 pom.xml 文件加入以下依赖：

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

在服务调用者的启动类中，打开 Feign 开关，请见代码清单 5-18。

代码清单 5-18：

codes\05\5.3\spring-feign-invoker\src\main\java\org\crazyit\cloud\InvokerApplication.java

```
@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients
public class InvokerApplication {

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

接下来，编写客户端接口，与直接使用 Feign 类似，代码清单 5-19 为服务端接口。

代码清单 5-19：

codes\05\5.3\spring-feign-invoker\src\main\java\org\crazyit\cloud\PersonClient.java

```
@FeignClient("spring-feign-provider") //声明调用的服务名称
public interface PersonClient {

    @RequestMapping(method = RequestMethod.GET, value = "/hello")
    String hello();
}
```

与单独使用 Fiegn 不同的是，接口使用了@FeignClient 注解来修饰，并且声明了需要调用的服务名称，本例的服务提供者名称为“spring-feign-provider”。另外，接口方法使用了@RequestMapping 来修饰，根据 5.2.7 章节可知，通过编写“翻译器（Contract）”，可以让 Feign 知道第三方注解的含义，Spring Cloud 也提供翻译器，会将@RequestMapping 注解的含义告知 Feign，因此我们的服务接口就可以直接使用该注解。

除了方法的@RequestMapping 注解外，默认还支持@RequestParam、@RequestHeader、@PathVariable 这 3 个参数注解，也就是说，在定义方法时，可以使用方式定义参数：

```
@RequestMapping(method = RequestMethod.GET, value = "/hello/{name}")
```

```
    String hello(@PathVariable("name") String name);
```

需要注意的是，使用了 Spring Cloud 的“翻译器”后，将不能再使用 Feign 的默认注解。接下来，在控制器中调用接口方法，请见代码清单是 5-20。

代码清单 5-20：

codes\05\5.3\spring-feign-invoker\src\main\java\org\crazyit\cloud\InvokerController.java

```
@RestController
@Configuration
public class InvokerController {

    @Autowired
    private PersonClient personClient;

    @RequestMapping(value = "/invokeHello", method = RequestMethod.GET)
    public String invokeHello() {
        return personClient.hello();
    }
}
```

在控制器中，为其注入了 PersonClient 的 bean，不难看出，客户端实例的创建及维护，Spring 容器都帮我们实现了。查看本例的效果，请按以下步骤操作：

- 启动 Eureka 服务器（spring-feign-server）。
- 启动两个服务提供者（spring-feign-provider），控制台中分别输入 8080 与 8081 端口。
- 启动一个服务调用者（spring-feign-invoker），端口为 9000。
- 在浏览器中输入：http://localhost:9000/invokerHello，可以看到服务提供者的“/hello”服务被调用。

Feign 负载均衡

在前面章节，我们尝试过编写自定义的 Feign 客户端，在 Spring Cloud 中，同样提供了自定义的 Feign 客户端。可能大家已经猜到，如果结合 Ribbon 使用，Spring Cloud 所提供的客户端，会拥有负载均衡的功能。

Spring Cloud 实现的 Feign 客户端，类名为 LoadBalancerFeignClient，在该类中，维护着与 SpringClientFactory 相关的实例，通过 SpringClientFactory 可以获取负载均衡器，负载均衡器会根据一定的规则来选取处理请求的服务器，最终实现负载均衡的功能。接下来，调用“服务提供者”的“/person/{personId}”服务来测试负载均衡。为客户端接口添加内容，请见代码清单 5-21。

代码清单 5-21：

codes\05\5.3\spring-feign-invoker\src\main\java\org\crazyit\cloud\PersonClient.java

```
    @RequestMapping(method = RequestMethod.GET, value = "/person/{personId}")
    Person getPerson(@PathVariable("personId") Integer personId);
```

为“服务调用者”的控制器添加方法，请见代码清单 5-22。

代码清单 5-22：

```
    @RequestMapping(value = "/router", method = RequestMethod.GET,
                produces = MediaType.APPLICATION_JSON_VALUE)
    public String router() {
```

```
        // 调用服务提供者的接口
        Person p = personClient.getPerson(2);
        return p.getMessage();
    }
```

运行“服务调用者”，在浏览器中输入：http://localhost:9000/router，进行刷新，可以看到 8080 与 8081 端口被循环调用。

默认配置

Spring Cloud 为 Feign 的使用提供了各种默认属性，例如前面讲到的“注解翻译器（Contract）”、Feign 客户端，默认情况下，Spring 将会为 Feign 的属性提供了以下的 Bean：

- 解码器（Decoder）：bean 名称为 feignDecoder，ResponseEntityDecoder 类，
- 编码器（Encoder）：bean 名称为 feignEecoder，SpringEncoder 类。
- 日志（Logger）： bean 名称为 feignLogger，Slf4jLogger 类。
- 注解翻译器（Contract）： bean 名称为 feignContract，SpringMvcContract 类。
- Feign 实例的创建者（Feign.Builder）：bean 名称为 feignBuilder，HystrixFeign.Builder 类。Hystrix 框架将在后面章节中讲述。
- Feign 客户端（Client）：bean 名称为 feignClient，LoadBalancerFeignClient 类。

一般情况下，Spring 提供的这些 Bean 已经足够我们使用，如果有些更特殊的需求，可以实现自己的 Bean，请见后面章节。

15 第一个 Hystrix 程序

先编写一个简单的 Hello World 程序，展示 Hystrix 的基本作用。

准备工作

使用 Spring Boot 的 spring-boot-starter-web 项目，建立一个普通的 Web 项目，发布两个测试服务用于测试，控制器的代码请见代码清单 6-1。

代码清单 6-1：codes\06\6.2\first-hystrix-server\src\main\java\org\crazyit\cloud\MyController.java

```
@RestController
public class MyController {

    @GetMapping("/normalHello")
    public String normalHello(HttpServletRequest request) {
        return "Hello World";
    }

    @GetMapping("/errorHello")
    public String errorHello(HttpServletRequest request) throws Exception {
```

```
            // 模拟需要处理 10 秒
            Thread.sleep(10000);
            return "Error Hello World";
        }
    }
```

一个正常的服务，另外一个服务则需要等待 10 秒才有返回。本例的 Web 项目对应的代码目录为 codes\06\6.2\first-hystrix-server，启动类是 ServerApplication。

客户端使用 Hystrix

结合 Hystrix 来请求 Web 服务，可能与原来的方式不太一样。新建项目“first-hystrix-client”，在 pom.xml 中加入以下依赖：

```
            <dependency>
                <groupId>com.netflix.hystrix</groupId>
                <artifactId>hystrix-core</artifactId>
                <version>1.5.12</version>
            </dependency>
            <dependency>
                <groupId>org.slf4j</groupId>
                <version>1.7.25</version>
                <artifactId>slf4j-log4j12</artifactId>
            </dependency>
            <dependency>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
                <version>1.2</version>
            </dependency>
            <dependency>
                <groupId>org.apache.httpcomponents</groupId>
                <artifactId>httpclient</artifactId>
                <version>4.5.2</version>
            </dependency>
```

本书 Spring Cloud 所使用的 Hystrix 版本为 1.5.12，我们也使用与其一致的版本。客户端项目除了要使用 Hystrix 外，还会使用 HttpClient 模块访问 Web 服务，因此要加入相应的依赖。新建一个命令类，实现请见代码清单 6-2。

代码清单 6-2：

codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\HelloCommand.java

```
public class HelloCommand extends HystrixCommand<String> {

    private String url;

    CloseableHttpClient httpclient;

    public HelloCommand(String url) {
        // 调用父类的构造器，设置命令组的 key，默认用来作为线程池的 key
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        // 创建 HttpClient 客户端
        this.httpclient = HttpClients.createDefault();
        this.url = url;
```

```
    }

    protected String run() throws Exception {
        try {
            // 调用 GET 方法请求服务
            HttpGet httpget = new HttpGet(url);
            // 得到服务响应
            HttpResponse response = httpclient.execute(httpget);
            // 解析并返回命令执行结果
            return EntityUtils.toString(response.getEntity());
        } catch (Exception e) {
            e.printStackTrace();
        }
        return "";
    }
}
```

新建运行类，执行 HelloCommand，如代码清单 6-3 所示。

代码清单 6-3：codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\HelloMain.java

```
public class HelloMain {

    public static void main(String[] args) {
        // 请求正常的服务
        String normalUrl = "http://localhost:8080/normalHello";
        HelloCommand command = new HelloCommand(normalUrl);
        String result = command.execute();
        System.out.println("请求正常的服务，结果：" + result);
    }
}
```

正常情况下，直接调用 HttpClient 的 API 来请求 Web 服务，而前面的命令类与运行类，则通过命令来执行调用的工作。在命令类 HelloCommand 中，实现了父类的 run 方法，使用 HttpClient 调用服务的过程，都放到了该方法中。运行 HelloMain 类，可以看到，结果与平常调用 Web 服务无异。接下来，测试使用 Hystrix 的情况下调用有问题的服务。

调用错误服务

假设我们所调用的 Hello 服务发生故障，导致无法正常访问，那么对于客户端来说，如何自保呢？本例将调用延时的服务，为客户端设置回退方法。修改 HelloCommand 类，加入回退方法，请见代码清单 6-4。

代码清单 6-4：codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\HelloCommand.java

```
    protected String getFallback() {
        System.out.println("执行 HelloCommand 的回退方法");
        return "error";
    }
```

在运行类中，调用发生故障的服务，请见代码清单 6-5。

代码清单 6-5：codes\06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\HelloErrorMain.java

```
public class HelloErrorMain {

    public static void main(String[] args) {
        // 请求异常的服务
        String normalUrl = "http://localhost:8080/errorHello";
        HelloCommand command = new HelloCommand(normalUrl);
        String result = command.execute();
        System.out.println("请求异常的服务，结果：" + result);
    }
}
```

运行 HelloErrorMain 类，输出如下：

```
执行 HelloCommand 的回退方法
请求异常的服务，结果：error
```

根据结果可知，回退方法被执行。本例中调用的“errorHello”服务，会阻塞 10 秒才有返回。默认情况下，如果调用的 Web 服务无法在 1 秒内完成，那么将会触发回退。

回退更像是一个备胎，当请求的服务无法正常返回时，就调用该“备胎”的实现。这样做，可以很好的保护客户端，服务端所提供的服务受网络等条件的制约，如果有服务真的需要 10 秒才能返回结果，而客户端又没有容错机制，后果就是，客户端将一直等待返回，直到网络超时或者服务有响应，而外界会一直不停地发送请求给客户端，最终导致的结果就是，客户端因请求过多而瘫痪。

16 Hystrix 运作流程

在前面的例子中，使用 Hystrix 时仅仅创建命令并予以执行，看似简单，实际上，Hystrix 有一套较为复杂的执行逻辑，为了能让大家大致了解该执行过程，笔者将整个流程作了简化。Hystrix 的运作流程请见图 6-3。

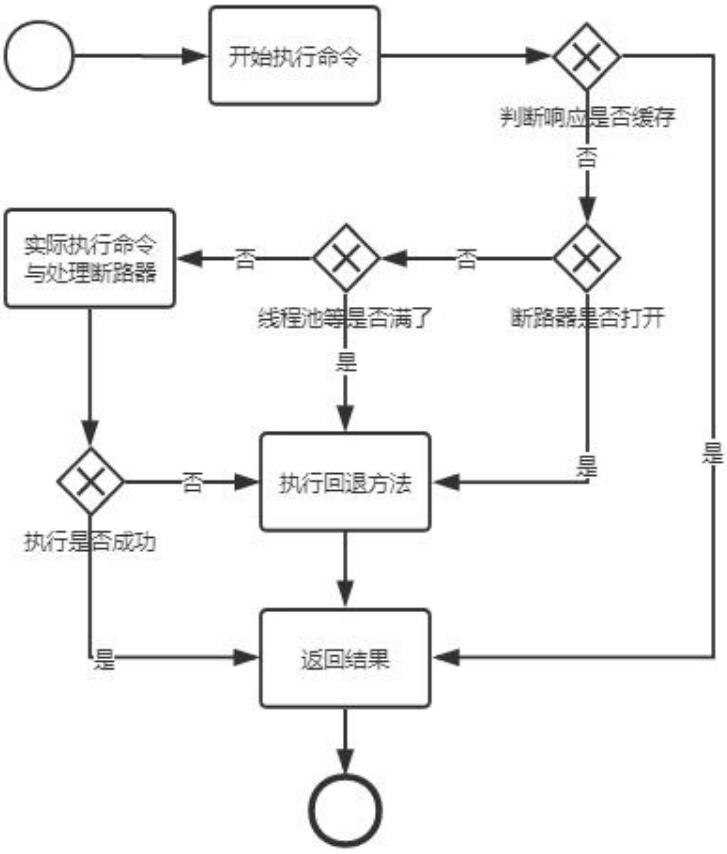

图 6-3 Hystrix 的运作流程图

简单说明一下运作流程：

- 第一步：在命令开始执行时，会做一些准备工作，例如为命令创建相应的线程池（后面章节讲述）等。
- 第二步：判断是否打开了缓存，打开了缓存就直接查找缓存并返回结果。
- 第三步：判断断路器是否打开，如果打开了，就表示链路不可用，直接执行回退方法。结合本章开头的例子，可理解为“基础服务”模块不可用，“服务 A”模块直接执行回退，响应用户请求。
- 第四步：判断线程池、信号量（计数器）等条件，例如像线程池超负荷，则执行回退方法，否则，就去执行命令的内容（例如前面例子中的调用服务）。
- 第五步：执行命令，计算是否要对断路器进行处理，执行完成后，如果满足一定条件，则需要开启断路器。如果执行成功，则返回结果，反之则执行回退。

整个流程最主要的点，就在于断路器是否被打开，后面会讲解断路器的相关内容。我们的客户端在使用 Hystrix 时，表面上只是创建了一个命令来执行，实际上 Hystrix 已经为客户端添加了几层的保护。

图 6-3 的流程图对 Hystrix 的运作流程做了最简单的描述，对于部分的细节，在此不进行赘述，读者大致了解运作流程即可。

17 Hystrix 属性配置与回退

属性配置

使用 Hystrix 时，可以为命令设置属性，以下的代码片断，为一个命令设置了执行的超时时间：

```
public MyCommand(boolean isTimeout) {
    super(

Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"))

            .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
                    .withExecutionTimeoutInMilliseconds(500))
        );
}
```

以上的配置仅对该命令生效，设置了命令的超时时间为 500 毫秒，该配置项的默认值为 1 秒，如果想对全局生效，可以使用以下的代码片断：

```
ConfigurationManager
        .getConfigInstance()
        .setProperty(
        "hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds",
        500);
```

以上的代码片断，同样设置了命令超时时间为 500 毫秒，但对全局有效。除了超时的配置外，还需要了解一下命令的相关名称，可以为命令设置以下名称：

- 命令组名称（GroupKey）：必须提供命令组名称，默认情况下，全局维护的线程池 Map 以该值作为 key，该 Map 的 value 为执行命令的线程池。
- 命令名称（CommandKey）：可选参数。
- 线程池名称（ThreadPoolKey）：指定了线程的 key 后，全局维护的线程池 Map 将以该值作为 key。

以下的代码片断，分别设置以上的 3 个 Key：

```
public RunCommand(String msg) {
    super(
            Setter.withGroupKey(
                    HystrixCommandGroupKey.Factory.asKey("group-key"))

.andCommandKey(HystrixCommandKey.Factory.asKey("command-key"))

.andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("pool-key"))

            );
}
```

Hystrix 的配置众多，后面章节的案例会涉及部分的配置，读者如果想了解更多的配置，可到以下的地址查看：https://github.com/Netflix/Hystrix/wiki/Configuration

回退

根据前面章节的流程图可知，至少会有 3 种情况触发回退（fallback）：

- 断路器被打开。
- 线程池、队列、信号量满载。
- 实际执行命令失败。

在命令中，实现父类（HystrixCommand）的 getFallback()方法，即可实现回退，当以上的情况发生时，将会执行回退方法。前面的例子中，已经展示了“执行命令失败”的回退，下面测试一下断路器被打开时的回退，详细请见代码清单 6-7。

代码清单 6-7：06\6.2\first-hnystrix-client\src\main\java\org\crazyit\cloud\fallback\FallbackTest.java

```
public class FallbackTest {

    public static void main(String[] args) {
        // 断路器被强制打开
        ConfigurationManager.getConfigInstance().setProperty(
                "hystrix.command.default.circuitBreaker.forceOpen", "true");
        FallbackCommand c = new FallbackCommand();
        c.execute();
        // 创建第二个命令，断路器关闭
        ConfigurationManager.getConfigInstance().setProperty(
                "hystrix.command.default.circuitBreaker.forceOpen", "false");
        FallbackCommand c2 = new FallbackCommand();
        c2.execute();
    }

    static class FallbackCommand extends HystrixCommand<String> {
        public FallbackCommand() {
            super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        }

        /**
         * 断路器被强制打开，该方法不会执行
         */
        protected String run() throws Exception {
            System.out.println("命令执行");
            return "";
        }

        /**
         * 回退方法，断路器打开后会执行回退
         */
        protected String getFallback() {
            System.out.println("执行回退方法");
            return "fallback";
        }
    }
}
```

如果让断路器打开，需要符合一定的条件，本例为了简单起见，在代码清单中，使用了

配置管理类（ConfigurationManager）将断路器强制打开与关闭，在打开断路器后，FallbackCommand 总会执行回退（getFallback）方法，将断路器关闭，命令执行正常。如果断路器被打开，而命令中没有提供回退方法，将抛出以下异常：

```
com.netflix.hystrix.exception.HystrixRuntimeException: FallbackCommand short-circuited and no fallback available.
```

另外，需要注意的是，命令执行后，不管是否会触发回退，都会去计算整个链路的健康状况，根据健康状况来判断是否要打开断路器，如果命令仅仅失败了一次，是不足以打开断路器的，关于断路器的逻辑将在后面章节讲述。

回退的模式

Hystrix 的回退机制比较灵活，你可以在 A 命令的回退方法中执行 B 命令，如果 B 命令也执行失败，同样也会触发 B 命令的回退，这样就形成一种链式的命令执行，例如以下代码片断：

```
static class CommandA extends HystrixCommand<String> {
    …省略其他代码
    protected String run() throws Exception {
        throw new RuntimeException();
    }

    protected String getFallback() {
        return new CommandB().execute();
    }
}
```

还有其他较为复杂的例子，例如银行转账，假设一个转账命令包含调用 A 银行扣款、B 银行加款两个命令，其中一个命令失败后，再执行转账命令的回退，如图 6-4 所示。

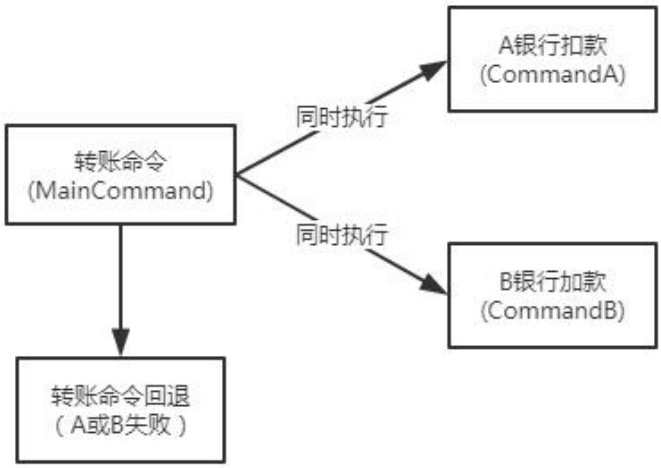

图 6-4 多命令回退

要做到图 6-4 的多命令只执行一次回退的效果，CommandA 与 CommandB，不能有回退方法，如果 CommandA 命令执行失败，并且该命令有回退方法，此时将不会执行“MainCommand”的回退方法。除了上面所提到的链式的回退以及多命令回退，读者还可以根据实际情况来设计回退。