第 1 章

通过 Spring Boot 入门微服务

通过微服务，架构师能有效地降低企业级应用里各模块的耦合度，从而能给企业带来切实的实惠。基于这一点（当前还有其他好处），在架构级别，微服务得到了广泛的重视。对于开发者来说，一旦具备微服务方面的开发和设计的能力，不仅能让自己有更多的工作机会，更能让自己在架构方面更加资深，从而让自己更有价值。

由于涉及架构，因此在开发微服务架构时，大家不仅要“写代码”，还要会设置一些配置 “分布式服务组件”的配置信息。听上去并不容易，不过本章将会通过简单易懂的文字让大家无障碍地通过 Spring Boot 入门“微服务”，并以此为起点，向大家展示企业级开发中“微服务架构”的常用组件。

1.1 Spring Boot、Spring Cloud 与微服务架构

和传统的 Spring MVC 框架相比，通过使用基于 Spring Boot 的开发模式，我们可以简化搭建框架时配置文件的数量，从而提升系统的可维护性。而且在 Spring Boot 框架里，我们还能更方便地引入 Spring Cloud 的诸如安全和负载均衡方面的组件。可以这样说，Spring Boot 架构是微服务的基础，在这个架构里，我们可以引入 Spring Cloud 的诸多组件，从而搭建基于微服务的系统。

搞明白这 3 个相关概念的关系后，我们能知道在微服务方面“该学什么”以及“该怎么学”，否则大家可能无法把微服务的知识点有效地整合成知识体系。

1.1.1 通过和传统架构的对比了解微服务的优势

从图 1.1 中，我们能看到一个传统的在线购物网站的基本架构。

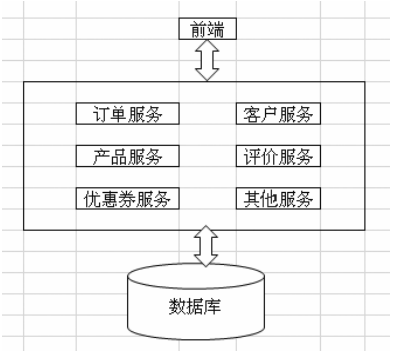

图 1.1　传统在线购物网站的基本架构

用户在前端页面上的操作，会被转化成一个个发向后端各模块的请求，当对应的模块处理请求时会和数据库交互。比如用户在前端页面输入关键字搜索商品，这个请求会被定位到“产品服务”模块里，该模块会和数据库交互，找到合适的商品结果后返回。

在实际项目里，为了应付高并发的访问请求（大家可以想象一下双十一的场景），往往会做分布式部署，如图 1.2 所示。在这种框架里，从前端页面里发到后端的大量请求会被负载均衡服务器（比如 Nginx 或 Ribbon）分发到不同的服务器处理，而在每个服务器里，都会有一套如图 1.1 所示的服务模块。如果再有必要，还可以把数据库做成集群，用多台数据库服务器分担高并发的压力。

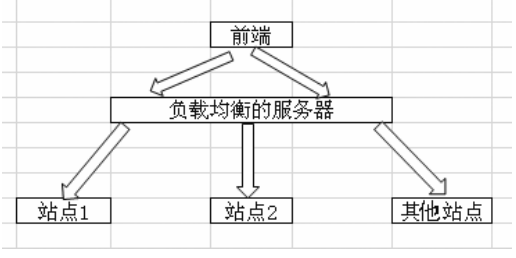

图 1.2　基于分布式的在线购物网站的架构

从实际效果上来看，如果采用图 1.2 的分布式架构，用多台业务处理服务器和数据库服务器，确实能满足高并发的需求。不过根据实践经验，上述架构一般会存在如下问题。

- 第一，各功能模块之间的调用关系会比较复杂，用专业的话来说就是耦合度比较高，一个模块的修改往往会影响到其他多个模块，也就是说代码比较难维护。
- 第二，由于在具体的每台机器上是集中式部署，因此稳定性不强，往往一个问题会导致整个系统崩溃。即使采用基于分布式的主从冗余等措施，这个问题也无法得到根本解决。
- 第三，可扩展性不强。假设当前的并发量是每秒 100 次请求，目前用 2 台服务器即可，当业务量上升后，每秒的并发量上升到 1000 次后，就需要再扩展服务器了，这时很不便利。

和上述架构相比，微服务（Microservice）的体系结构如图 1.3 所示。

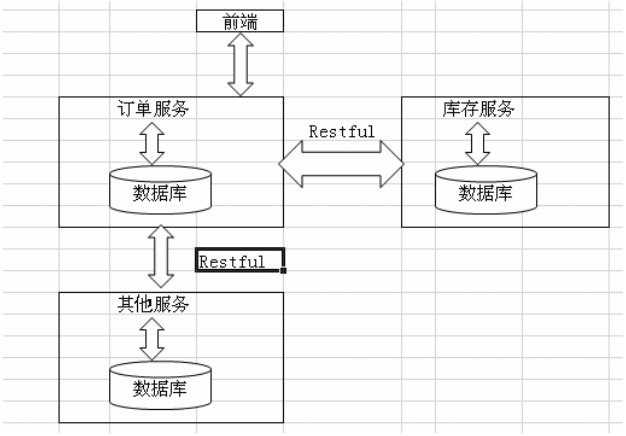

图 1.3　微服务架构

从图 1.3 我们能看到，微服务模块之间一般会通过 Restful 格式的请求通信，换句话说，模块间的耦合度比较低，这样就很便于在任何模块里变更业务需求。

而且，每个模块都具有自己的数据库，也就是说，每个模块都能独立运行，整个系统的扩展性比较强，比如能用比较小的代价来扩展新的功能模块。

1.1.2　Spring Boot、Spring Cloud 和微服务三者的关系

微服务是体系架构，或者说是模块的组织形式，说得再通俗点，如果我们用“微服务架构”的方式组装业务模块，那么整个系统就能具有如上文所述的“高扩展性”和“模块间低耦合度”的特性。

注意，微服务是一个抽象的概念，它有不同的实现方式，而基于 Spring Boot 的 Spring Cloud 是当前比较流行的一种实现微服务的方式。

由于 Spring 具备 IOC 的特性，因此通过 Spring 开发出来的模块，它们之间的耦合度非常低，这同微服务的要求非常相似。在之前 Spring 版本的基础上，Pivotal 团队提供了一套全新的 Spring Boot 框架。

在这套框架里，开发者可以嵌入 Web 服务器，比如 Tomcat，无须像之前那样把项目文件打包（假设打包成 War 文件）并部署到 Web 服务器上，而且 Spring Boot 还具备自动配置的功能，更为便利的是，通过定义配置文件，开发者还能 “自动监控健康”基于 Spring Boot 框架模块的各项运行时的性能指标。总之，大家可以这样理解，Spring Boot 是之前 Spring 框架的升级版，通过之后基于代码的叙述，我们更能详细地体会到 Spring Boot 框架的优势。

我们可以通过 Spring Boot 在单台机器上搭建实现业务功能的模块，但事实上实现高并发的网站项目一般有“负载均衡”“路由代理”“消息服务”和“流量过高断路”等需求，而这些需求能很好地通过 Spring Cloud 这套框架提供的组件得到解决。

讲完三者的含义后，我们就能清晰地理顺这三者的关系了。

- 第一，微服务架构能给项目带来便于扩展和维护的优势，从而能给公司带来实惠，这也是微服务比较热门的原因。（有好处了别人才会用。）

- 第二，通过 Spring Boot 能开发微服务“单机版”的功能模块。即使是单机版的，由于在其中能嵌入 Web 服务器（当然还有其他升级点），因此和传统的 Spring 架构相比，也能给开发人员带来实际的便利。
- 第三，通过基于 Spring Cloud 框架的实现“负载均衡”等功能的组件，我们能有效地把各“单机版”的功能模块整合到一起组成架构。在这套架构里，我们不仅能得到微服务架构所带来的好处，由于引入了 Spring Cloud 组件，因此我们更能满足“高并发访问量”的需求。

1.1.3 基于 Netflix OSS 的 Spring Cloud 的常用组件

提到 Spring Cloud，我们就不得不先提一下 Netflix。这家公司组织成立了一个开源社区，名为 Netflix Open Source Software Center（Netflix OSS）。经过很多大神级别的开发者共同努力，这个社区推出了架构，Spring Cloud 就是其中之一。

大家也可以这样理解，Spring Cloud 是各种支持分布式微服务的组件的集合，通过配套使用其中的各项组件，开发者能以“微服务”的方式构建基于分布式部署的系统。在表 1.1 里，我们能看到 Spring Cloud 中的常用组件。

表 1.1 Spring Cloud 常用组件归纳表

组件名	功能	在项目中的作用
Eureka	服务治理组件	能很好地管理提供微服务的各项模块，比如通过 Eureka，系统能有效地发现新注册的组件，并把它加入到集群中
Ribbon	实现负载均衡的组件	能把高并发的请求有效地分发到已注册的各服务节点上
Hystrix	能提供容错保护功能	像保险丝，一旦请求多到会让系统崩溃，Hystrix 就会自动熔断，这样请求就不会再发到系统里，从而能保护系统
Zuul	能提供路由功能	比如能过滤掉一些非法请求，也能提供智能路由功能
RabbitMQ 或 Kafka	消息中间件	通过诸如这类的消息中间件，各模块间能有效地发送消息
Feign	能优化调用服务的框架	在微服务框架里，模块间一般是通过 Rest 的格式来通信，通过 Feign，模块间能更便捷地调用 Rest 服务
Steuth	能跟踪微服务的调用过程	在企业级应用里，一般会包含多个模块，而一个请求往往会调用多个服务模块。通过 Steuth，开发者能方便地看到服务调用的流程，从而能很方便地定位问题
Spring Cloud Config	服务配置管理工具	通过它能很好地管理微服务框架（或是集群）中的诸多配置文件

表 1.1 讲述的一些组件，比如 Ribbon 或 Hystrix 不只是能被用在微服务领域，在其他的高并发场景下也能用到。由此我们能体会到，上述组件构成了能搭建基于 Spring Cloud 微服务的全家桶，开发者能根据实际需求选用其中的一个或多个组件。

1.2　通过 Maven 开发第一个 Spring Boot 项目

用传统 Spring 框架开发项目，虽然各项目的业务功能点不同，但是它们会有不少相同的配置文件。新创建一个 Spring 项目时，我们不得不复制这些配置文件，对架构师（或高级开发）来说，这种“代码粘贴”动作是需要尽量避免的。Spring Boot 能有效地解决这类问题。

Spring Boot 没有“颠覆性”地改变 Spring 框架，而是通过引入 Maven 和“自动化配置”等方式来简化配置文件。它不仅能让开发者在新建项目时减少配置文件方面的工作量，还能进一步降低项目中类和 jar 包之间的依赖关系，它的价值在于“能减轻程序员在开发和配置项目中的工作量”。

1.2.1　Maven 是什么，能带来什么帮助

我们在用 Eclipse 开发项目时，一定会引入支持特定功能的 jar 包，比如从图 1.4 中，我们能看到这个项目需要引入支持 mysql 的 jar 包。

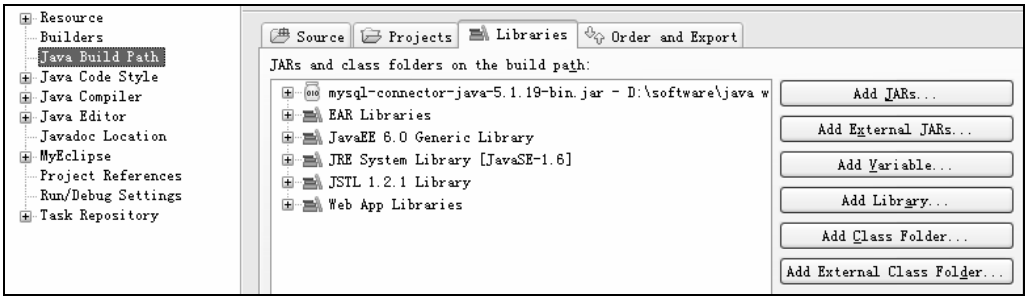

图 1.4　在项目里引入 jar 包的示意图

从图 1.4 中我们能看到，支持 mysql 的 jar 包是放在本地路径里的，这样在本地运行时自然是没问题的，但要把这个项目发布到服务器上就会有问题了，因为在这个项目的.classpath 文件已经指定 mysql 的 jar 包在本地 D 盘下的某个路径中，如图 1.5 所示。

```
<classpathentry kind="lib" path="D:/software/java web/mysql-connector-java-5.1.19-bin.jar"/>
<classpathentry kind="output" path="WebRoot/WEB-INF/classes"/>
```

图 1.5　指定 jar 路径的 classpath 文件的片段

一旦发布到服务器上，项目依然会根据.classpath 的配置从 D 盘下的这个路径去找，事实上服务器上是不可能有这样的路径和 jar 包的。

我们也可以通过在.classpath 里指定相对路径来解决这个问题，在下面的代码里，我们可以指定本项目将引入“本项目路径/WebRoot/lib”目录里的 jar 包。

```
    <classpathentry kind="lib" path="WebRoot/lib/jar 包名.jar"/>
```

这样做，发布到服务器时，由于会把整个项目路径里的文件都上传，因此不会出错。但这样依然会给我们带来不便。比如这个服务器上我们部署了 5 个项目，它们都会用到这个 mysql 支持包，

这样我们就不得不把这个 jar 包上传 5 次。再扩展一下，如果 5 个项目里会用到 20 个相同的 jar 包，那么我们还真就不得不做多次复制。如果我们要升级其中的一个 jar 包，那么还真就得做很多重复的复制粘贴动作。

期望中的工作模式应该是，有一个“仓库”统一放置所有的 jar 包，在开发项目时，可以通过配置文件引入必要的包，而不是把包复制到本项目里。这就是 Maven 的做法。

用通俗的话来讲，Maven 是一套 Eclipse 的插件，它的核心价值是能理顺项目间的依赖关系，具体来讲，能通过其中的 pom.xml 配置文件来统一管理本项目所要用到的 jar 包，在项目里引入 Maven 插件后，开发者就不必手动添加 jar 包了，这样也能避免因此来带来的一系列问题。

1.2.2 通过 Maven 开发 Spring Boot 的 HelloWorld 程序

在这个案例中，大家不仅可以理解如何开发 Spring Boot 的程序，更能理解 Maven 的一般用法。

代码位置	视频位置
代码\第 1 章\MyFirstSpringBoot	视频\第 1 章\通过 Maven 开发 Spring Boot 的 HelloWorld 程序

第一步，创建 Maven 项目。本书使用 MyEclipse 作为开发环境，在其中已经引入了 Maven 插件，所以我们可以通过“File”→“New”菜单，直接创建 Maven 项目，如图 1.6 所示。

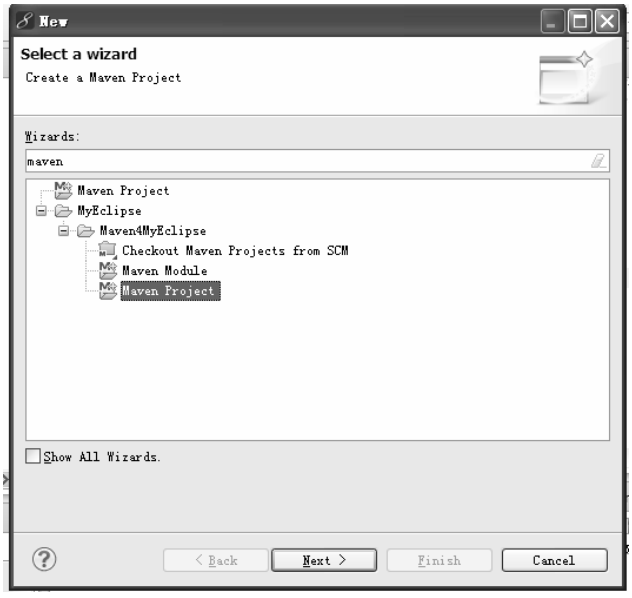

图 1.6 在 MyEclipse 里创建 Maven 项目的示意图

在图 1.6 中，单击“Next”按钮后，会见到如图 1.7 所示的界面，在其中我们可以设置 Group Id 等属性。

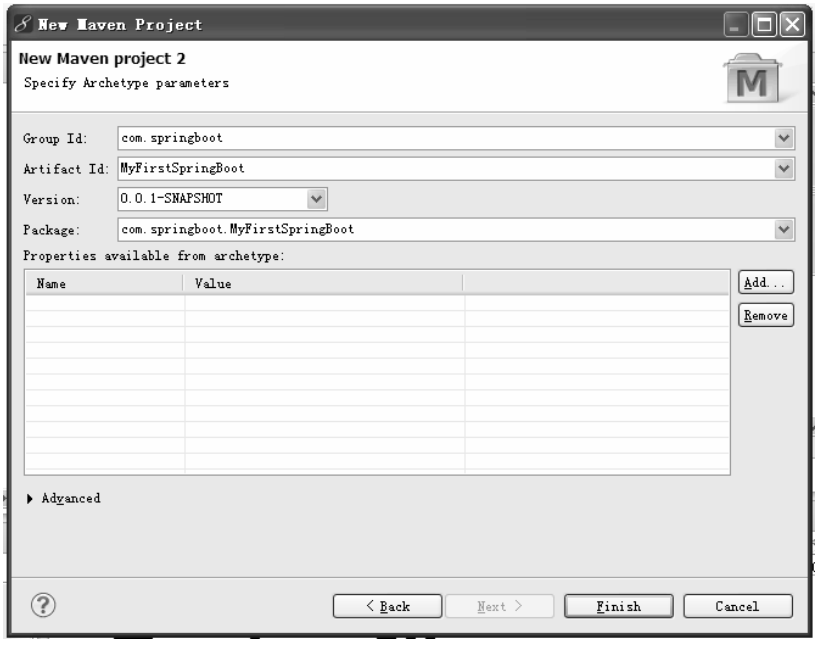

图 1.7　设置 Maven 各属性的示意图

其中，Group Id 代表公司名（也叫组织名），这里设置成“com.springBoot”；Artifact Id 是项目名；Version 和 Packag 采用默认值。一般来说，通过 Group Id、Artifact Id 和 Version 就能定位到唯一的 jar 包。完成设置后，能看到新建的项目 MyFirstSpringBoot，如图 1.8 所示。

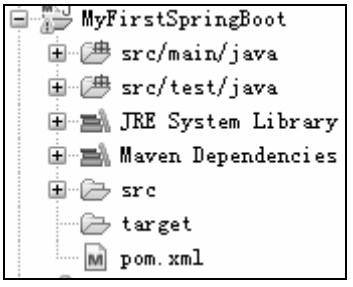

图 1.8　创建好的 Maven 项目示意图

第二步，改写 pom.xml。当我们创建好 Maven 项目后，在其中能看到 pom.xml 文件。在 Maven 项目里一般是通过 pom.xml 来指定本项目的基本信息以及需要引入的 jar 依赖包，关键代码如下：

```
<groupId>com.springboot</groupId>
<artifactId>MyFirstSpringBoot</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
<name>MyFirstSpringBoot</name>
<url>http://maven.apache.org</url>
<dependencies>
 <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>1.5.4.RELEASE</version>
```

```
12    </dependency>
13    <dependency>
14      <groupId>junit</groupId>
15      <artifactId>junit</artifactId>
16      <version>3.8.1</version>
17      <scope>test</scope>
18    </dependency>
19  </dependencies>
```

其中，第 1~4 行的代码是自动生成的，用来指定本项目的基本信息，这和我们在之前创建 Maven 项目时所填的信息是一致的。

从第 7~19 行的 dependencies 属性里，我们可以指定本项目所用到的 jar 包，在第 8 和第 13 行分别通过两个 dependency 来指定该引入两类 jar 包。其中，第 8~12 行指定了需要引入用以开发 Spring Boot 项目的名为 spring-boot-starter-web 的 jar 的集合，第 13~18 行指定了需要引入用以单元测试的 junit 包。

从上述代码中，我们能见到通过 Maven 管理项目依赖文件的一般方式。比如在下面的代码片段里，通过第 2~4 行的代码说明需要引入 org.springframework.boot 这个公司组织（发布 Spring Boot jar 包的组织）发布的名为 spring-boot-starter-web 的一套支持 Spring Boot 的 jar 包，而且通过第 4 行指定了引入包的版本号是 1.5.4.RELEASE。

```
1  <dependency>
2        <groupId> org.springframework.boot </groupId>
3        <artifactId>spring-boot-starter-web</artifactId>
4        <version>1.5.4.RELEASE</version>
5  </dependency>
```

这样一来，在本项目里，我们就不用再手动地添加 jar 包，这些包实际上是存放在本地的 jar 包仓库里的，也就是说，在项目里是通过 pom.xml 的配置来指定需要引入这些包。

第三步，改写 App.java。在创建项目时，指定的 package 是 com.springboot.MyFirstSpringBoot，在其中会有一个 App.java，我们把这个文件改写成如下样式。

```
1  package com.springboot.MyFirstSpringBoot;
2  import org.springframework.boot.SpringApplication;
3  import org.springframework.boot.autoconfigure.SpringBootApplication;
4  import org.springframework.web.bind.annotation.RequestMapping;
5  import org.springframework.web.bind.annotation.RestController;
6
7  @RestController
8  @SpringBootApplication
9  public class App {
10     @RequestMapping("/HelloWorld")
11     public String sayHello() {
12         return "Hello World!";
13     }
14     public static void main(String[] args) {
15         SpringApplication.run(App.class, args);
16     }
17 }
18
```

由于是第一次使用 Maven，我们在这里再强调一次，虽然我们没有在项目里手动地引入 jar，

但是在 pom.xml 里指定了待引入的依赖包，具体而言就是需要依赖 org.springframework.boot 组织所提供的 spring-boot-starter-web，所以在代码的第 2~5 行里，我们可以通过 import 语句，使用 spring-boot-starter-web（也就是 Spring Boot）的类库。

在第 8 行里，我们引入了@SpringBootApplication 注解，以此声明该类是一个基于 Spring Boot 的应用。

在第 10~13 行的代码里，我们通过@RequestMapping 指定了用于处理/HelloWorld 请求的 sayHello 方法，在第 14 行的 main 函数里，我们通过第 15 行的代码启动了 Web 服务。

至此，我们完成了代码编写工作。启动 MyFirstSpringBoot 项目里的 App.java，在浏览器里输入“http://localhost:8080/HelloWorld”。

由于/HelloWorld 请求能被第 11~13 行的 sayHello 方法的@RequestMapping 对应上，所以会通过 sayHello 方法输出“Hello World!”的内容，如图 1.9 所示。

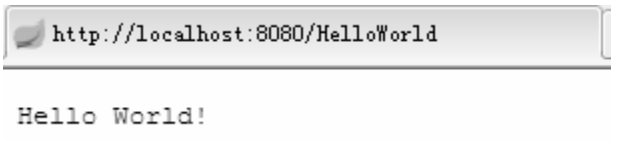

图 1.9　HelloWorld 程序运行效果图

从这个程序里，我们能体会到开发 Spring Boot 和传统 Spring 程序的不同。

第一，在之前的 Spring MVC 框架里，我们不得不在 web.xml 中定义采用 Spring 的监听器，而且为了采用@Controller 控制器类，我们还得加上一大堆配置，但在 Spring Boot 里，我们只需要添加一个@SpringBootApplication 注解。

第二，我们往往需要把传统的 Spring MVC 项目发布到诸如 Tomcat 的 Web 服务器上，启动 Web 服务器后我们才能在浏览器里输入请求查看运行的效果；这里我们只需启动 App.java 这个类即可达到类似的效果。

1.2.3　Controller 类里处理 Restful 格式的请求

之前我们已经提到过，微服务模块间一般是通过 Restful 格式的请求来交互，在表 1.2 里，我们能看到各种 Restful 请求的格式。

表 1.2　常用 Restful 格式请求的功能归纳表

请求类型	url	功能说明
Get	/accounts	以 HTTP 里的 get 协议查询所有的 account 对象
Post	/accounts	以 HTTP 里的 post 协议查询所有的 account 对象
Get	/accounts/id	返回指定 id 的账户，相当于“查指定对象”
Put	/accounts/id	更新指定 id 的账户，相当于“改”
Delete	/accounts/id	删除指定 id 的账户，相当于“删”

其中，Get 等都是基于 HTTP 协议的请求。具体而言，如果我们指定请求类型是 Get，并设置请求 url 是/accounts/123，那么我们就能得到 id 是 123 的账户信息，如果发的是 Get 类型的/accounts，

就返回所有的账户。

在 SpringBootRestfulDemo 案例中，我们将向大家演示在 Spring Boot 里编写支持 Restful 格式请求的服务类的一般方法，同样，这里我们用 Maven 来创建项目。

代码位置	视频位置
代码\第 1 章\SpringBootRestfulDemo	视频\第 1 章\Spring Boot Restful 效果演示

在这个项目里，我们用和刚才 MyFirstSpringBoot 一样的方法创建 Maven 项目，只是这里的 artifactId 需要填写成本项目的名字 SpringBootRestfulDemo。这个项目的 pom.xml 和 MyFirstSpringBoot 项目里的一致，同样是引入 Spring Boot 的依赖包。在这个项目的 App.java 的 main 函数里，我们同样加入了启动代码，如下所示。

```
1  //省略必要的 package 和 import 代码
2  //同样通过@SpringBootApplication 注解来说明本类是启动类
3  @SpringBootApplication
4  public class App {
5      public static void main(String[] args) {
6          SpringApplication.run(App.class, args);
7      }
8  }
```

在这个项目中，我们需要定义描述账户信息的 Account 类，代码如下所示。

```
1  package com.springboot.SpringBootRestfulDemo;
2  public class Account {
3     private int id;
4     private String accountName;
5      //省略针对 id 和 accountName 这两个属性的 get 和 set 方法
6  }
```

在 RestfulController.java 里，我们将定义处理各种 Restful 格式请求的方法，代码如下所示。

```
1  //省略必要的 package 和 import 方法
2  //通过这个注解说明本控制器可以处理 Restful 格式的请求
3  @RestController
4  public class RestfulController {
5     //正式场景里，应当在数据表里存储账户信息，这里我们用 HashMap 演示
6      static Map<Integer, Account> accounts = new HashMap<
7          Integer, Account>();
8      //如果是 Get 请求，而且请求格式是/account，则将调用这个方法
9      @RequestMapping(value = "/account", method = RequestMethod.GET)
10     List<Account> getAccountList() //返回所有的账户信息
11     { return new ArrayList<Account>(accounts.values()); }
12     //如果是 POST 请求，而且请求格式是/account，则将调用这个方法
13     @RequestMapping(value = "/account", method = RequestMethod.POST)
       //插入一条数据，并返回 OK
14     String postAccount(@ModelAttribute Account account){
15      accounts.put(account.getId(), account);
16      return "OK";
17     }
18     //如果是 GET 请求，而且请求时带 id 参数，则将调用这个方法
19     @RequestMapping(value = "/account/{id}", method = RequestMethod.GET)
20     Account getAccount(@PathVariable Integer id){
```

```
21      //return accounts.get(id);
22      //在项目中，一般会如 21 行所示从数据源里得到数据并返回
23      //但这里，由于没有数据源，所以这里造个数据返回
24      Account account = new Account();
25      account.setId(id);
26      account.setAccountName("Tom");
27      return account;
28      }
29      //如果是 PUT 请求，而且请求时带 id 参数，则将调用这个方法
30      @RequestMapping(value="/account/{id}", method=RequestMethod.PUT)
31      String putAccount(@PathVariable Integer id, @ModelAttribute
        Account account){
32      //向数据源插入一条数据并返回
33      accounts.put(id, account);
34      return "OK";
35      }
36      //如果是 Delete 请求，而且请求时带 id 参数，则将调用这个方法
37      @RequestMapping(value="/account/{id}", method=RequestMethod.DELETE)
38      String deleteUser(@PathVariable Integer id){
39      //从数据源里删除这条 id 所指向的账号信息
40      accounts.remove(id);
41      return "OK";
42      }
43  }
```

在上述代码里，我们在每个方法的@RequestMapping 注解里，不仅指定了触发该方法的 url 请求格式，还指定了能触发该方法的请求类型。

在正式的项目里，我们是从数据源（比如 Account 数据表）里获取数据，这里我们用 HashMap 来代替数据库，所有的增、删、改、查都是针对上文第 6 行定义的 accounts 对象。

这里我们通过 url 的形式简易演示一下“Get”形式请求的运行效果。启动 App.java 后，在浏览器里输入“http://localhost:8080/account/1”，我们能看到 Json 格式的返回效果，如图 1.10 所示。

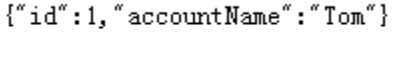

图 1.10　Get 请求返回的效果图

这里的请求其实是触发了第 20 行的 getAccount 方法，至于 Post 等其他格式的请求，无法通过浏览器的形式简单地调用，所以这里只给出实现代码，在后文里，我们将详细地给出调用方法。

1.2.4　@SpringBootApplication 注解等价于其他 3 个注解

Spring Boot 和传统的 Spring 框架一样，是通过注解来降低类（以及模块）之间的耦合，在其中，@SpringBootApplication 这个注解用得比较多，因为我们一般用它来启动应用项目。

事实上它是一个复合注解，等价于@ComponentScan、@SpringBootConfiguration 和@EnableAutoConfiguration。

- @ComponentScan 继承于@Configuration，用来表示程序启动时将自动扫描当前包及子包下的所有类。

- @SpringBootConfiguration 表示将会把本类里声明的一个或多个以@Bean 注解标记的实例纳入 Spring 容器中。
- @EnableAutoConfiguration 用来表示程序启动时将自动地装载 springboot 默认配置文件。

1.2.5 通过配置文件实现热部署

如果我们每次在修改完 Spring Boot 里的 Java 或配置文件后都需要重启诸如 App.java 这样的启动类才能生效，那么这样的开发效率未免太低。在实际的开发过程中，我们可以通过修改 pom.xml 的方式来实现热部署。

以刚才的 SpringBootRestfulDemo 项目为例，为了实现热部署，我们需要把 pom.xml 修改如下：

```
1   <dependencies>
2       其他代码不变，只需添加一个 dependency 元素
3       <dependency>
4           <groupId>org.springframework.boot</groupId>
5           <artifactId>spring-boot-devtools</artifactId>
6           <version>1.5.4.RELEASE</version>
7       </dependency>
8       其他代码不变
9     </dependencies>
```

当我们在 pom.xml 添加完第 3~7 行的代码后，启动 App.java，这时我们能看到如下输出。

```
1   {"id":1,"accountName":"Tom"}
```

注意，此时别停服务，直接修改 getAccount 方法，把第 6 行参数修改成“Peter”，如下所示。

```
1   @RequestMapping(value = "/account/{id}", method = RequestMethod.GET)
2       Account getAccount(@PathVariable Integer id)    {
3        //return accounts.get(id);
4        Account account = new Account();
5        account.setId(id);
6        account.setAccountName("Peter");
7        return account;
8        }
```

此时如果我们再往浏览器里输入 http://localhost:8080/account/1，那么输出就变成“Peter”了，也就是说，无须重启 App 启动类，即能看到修改后的效果。

```
1   {"id":1,"accountName":"Peter"}
```

1.3 通过 Actuator 监控 Spring Boot 运行情况

当我们把 Spring Boot 部署到服务器之后，一般需要监控微服务的运行情况：一方面，我们可以据此分析和排查问题；另一方面，我们能以此为依据优化代码。

Spring Boot 里提供了 spring-boot-starter-actuator 模块，引入该模块后，我们能实时地监控微服务的部署和运行情况，从而能减少程序员编写监控系统模块所用的工作量。这里我们将着重讲一下

常用的监控指标。

1.3.1　准备待监控的项目

新建一个基于 Maven 的名为 SpringBootActuatorDemo 的项目，启动后，再通过 actuator 来监控它所在站点的实时情况。

代码位置	视频位置
代码\第 1 章\SpringBootActuatorDemo	视频\第 1 章\通过 Actuator 监控项目

步骤01 在 pom.xml 加入 Spring Boot 和 actuator 的依赖包，关键代码如下：

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

其中，第 2~6 行引入的是 Spring Boot 的依赖包，第 7~11 行引入的是 actuator 的依赖包，其他代码不变。

步骤02 在 App.java 的 main 函数里，同样编写启动 Spring Boot 的代码。

```
//省略必要的 package 和 import 代码
@SpringBootApplication
public class App{
    public static void main( String[] args ){
    //启动 Spring Boot
     SpringApplication.run(App.class, args);
    }
}
```

步骤03 在 src 目录下，编写包含配置信息的 application.properties 文件。在 Spring Boot 的项目里，我们一般把配置文件放在这个目录，如图 1.11 所示。

图 1.11　application.properties 文件的一般位置

application.properties 里的代码如下所示。

```
management.security.enabled=false
```

```
2   info.build.artifact=org.springframework.boot
3   info.build.name=SpringBootActuatorDemo
4   info.build.description=DemoActuator
5   info.build.version=1.0
```

其中，第 1 行的代码用来指定本站点（运行本项目的站点，也叫节点）无须验证，这样我们就能通过浏览器看到一些 actuator 给出的监控信息，第 2~5 行的代码用来指定本站点的信息。

编写完成后，通过 App.java 启动 Spring Boot，随后，我们就能通过 actuator 查看监控信息。

1.3.2 通过/info 查看本站点的自定义信息

在确保启动 SpringBootActuatorDemo 的情况下，在浏览器里输入“http://localhost:8080/info”，能看到如下输出信息：

```
1   {"build":
2   {"description":"DemoActuator","name":"SpringBootActuatorDemo",
     "version":"1.0","artifact":"org.springframework.boot"}
3   }
```

其中，第 2 行的输出信息和我们在 application.properties 里配置的站点信息是一致的。

1.3.3 通过/health 查看本站点的健康信息

输入“http://localhost:8080/health”，能看到如下关于本站点健康信息的输出：

```
1   {"status":"UP",
2   "diskSpace":{"status":"UP","total":143893012480,"free":73405607936,
    "threshold":10485760}
3   }
```

在第 1 行里，能看到本站点的状态是“UP”，也就是启动状态；在第 2 行里，能看到关于磁盘使用量的情况，总体来说，状态也是“UP”。

1.3.4 通过/metrics 查看本站点的各项指标信息

输入“http://localhost:8080/metrics”，我们能看到关于本站点内存使用量、线程使用情况以及垃圾回收等信息，大致输出如下：

```
1   {
2   "mem":54530,
3   "mem.free":7435,
4   "processors":2,
5   "instance.uptime":8862204,
6   省略其他信息
7    }
```

比如在上述第 3 行里，我们能看到空闲内存的值。这里的指标数很多，我们就不一一列出了，大家可以自己看一下。总结起来，/metrics 将返回如下种类的信息：

- mem.*：描述内存使用量的信息。
- heap.*：描述虚拟机堆内存的信息。
- threads.*：描述线程使用情况的信息。
- classes.*：描述类加载和卸载的信息。
- gc.*：用来描述垃圾回收的信息。

此外，我们还能通过具体的指标名查看对应的值，比如输入“http://localhost:8080/metrics/gc.*”，就能看到垃圾回收相关指标的信息，输出如下：

```
1   {"gc.copy.count":60,"gc.copy.time":206,"gc.marksweepcompact.count":
    2, "gc.marksweepcompact.time":97}
```

1.3.5　actuator 在项目里的实际用法

除了刚才给出的用法外，我们还能通过/env 查看当前站点的环境信息，能通过/mappings 来查看当前站点的 Spring MVC 控制器的映射关系，能通过/beans 来查看当前站点中的 bean 信息。

不过在项目里，我们一般不是通过浏览器来查看，而是会通过代码来定时检测，再进一步，一旦当检测到的数据低于预期就自动发警告邮件。在本书的后继部分，将给出这种做法的实际案例。

1.4　本 章 小 结

这章我们不仅讲述了微服务和传统体系架构的差别，还通过了一些基本的 Spring Boot 案例让大家感性地认识了微服务。通过这些案例，大家不仅可以了解到 Spring Boot 的基本语法，还能掌握实际项目中和 Spring Boot 密切相关的一些技能，比如热启动、如何在控制器类里处理 Restful 格式的请求和通过 actuator 监控微服务站点的方法等。

通过本章，大家能对 Spring Boot 有一个初步的了解，这也是大家继续通过本书后继章节了解 Spring Cloud 微服务的基础。请大家注意，微服务是一个框架，所以大家在后继学习时，不仅要专注具体的实现代码，务必还要关注微服务的框架本身，比如微服务模块间如何实现“负载均衡”以及多个微服务模块构建成集群的方式。

第 2 章

用 Spring Data 框架连接数据库

和 JDBC 一样，通过 Spring Data 框架里的 JPA 组件，我们也能用比较相似的方法无差别地访问不同类型数据库。

这种“屏蔽”的便利性和 Spring 里“解耦合”的理念是一脉相承的，具体来说，通过 Spring Data 框架，我们能轻易地解耦合业务逻辑和底层的数据库实现逻辑，这种“解耦合”的特性能从很大程度上提升系统的扩展性与可维护性，使得我们能用很小的代码更换系统的数据存储容器。

而且，JPA 组件也能起到 ORM 里映射的效果，也就是说，通过它，我们还能比较容易地实现业务中“一对一”“一对多”和“多对多”的效果。

2.1 Spring Data 框架概述

Spring Data 是一个能简化数据库访问的开源框架，通过该框架里的 ORM 特性，我们能比较快捷地编写对数据库层的访问逻辑。由于它也是 Spring 家族的，因此它和 Spring Boot 乃至 Spring Cloud 有着天然的亲近性。

从图 2.1 中，我们能看到 Spring Data 框架在项目里所起到的作用，通过它，程序员能更关注于企业的核心价值——业务实现，从而可以不必过多地关注业务数据在数据库层的存储和读取细节，这种解耦合的便利性无疑将提升系统代码的可维护性。

在表 2.1 中，我们归纳了一些常见的子项目以及所对应的功能。不过在实际项目里，我们用得比较多的还是 JPA 组件。

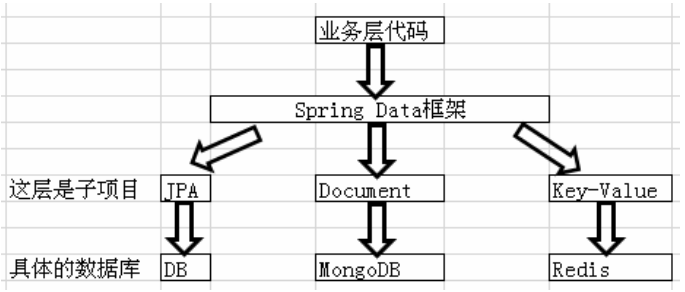

图 2.1　Spring Data 框架在项目里的示意图

表 2.1　Spring Data 常用子项目功能归纳表

子项目名	功能
JPA	支持对传统数据库的连接操作
Document	能支持 NoSQL，比如 MongoDB
Key-Value	能支持 Key-Value 类型的数据库，比如 Redis
Hadoop	能支持 Hadoop 的 MapReduce 特性
Graph	能支持 Neo4j 图形数据库

2.2　Spring Data 通过 JPA 连接 MySQL

JPA（Java Persistence API）是一套数据库持久层映射的规范，我们比较熟悉的 Hibernate 框架就是基于这套规范实现的，也就是说，它们两者的语法和开发方式非常相似。

这里，我们将通过 Spring Data 里的 JPA 实现组件来开发针对 MySQL 数据库的各种操作。

2.2.1　连接 MySQL 的案例分析

这里我们将实现通过 JPA 连接并访问 MySQL 数据库的整个流程。

代码位置	视频位置
代码\第 2 章\SpringBootJPAMySQLDemo	视频\第 2 章\Spring Boot 连接 MySQL 数据库

1．创建数据表，构建 Maven 项目

我们在 MySQL 里创建一个名为 springboot 的数据库，在其中创建一个名为 student 的表，结构如表 2.2 所示。

表 2.2　student 表结构的说明

字段名	类型	含义
id	varchar	主键，学号
name	varchar	姓名
age	varchar	年龄
score	float	成绩

创建完表之后，我们再创建一个名为 SpringBootJPAMySQLDemo 的 Maven 类型的项目。

2．在 pom.xml 里配置要用到的包

本项目中 pom.xml 的关键代码如下，在其中将指定本项目要用到的 jar 包。

```
1    //省略描述项目名部分的配置代码
2    <parent>
3         <groupId>org.springframework.boot</groupId>
4         <artifactId>spring-boot-starter-parent</artifactId>
5         <version>1.5.6.RELEASE</version>
6     </parent>
7    <dependencies>
8     <dependency>
9            <groupId>org.springframework.boot</groupId>
10           <artifactId>spring-boot-starter-web</artifactId>
11    </dependency>
12    <dependency>
13          <groupId>org.springframework.boot</groupId>
             <artifactId>spring-boot-starter-data-jpa</artifactId>
14     </dependency>
15     <dependency>
16         <groupId>mysql</groupId>
17         <artifactId>mysql-connector-java</artifactId>
18         <version>5.1.3</version>
19     </dependency>
20     //省略描述 junit 依赖包的代码
21   </dependencies>
```

在第 2~6 行中，我们用 parent 标签来配置各子模块将要依赖的通用依赖包，也就是各子模块都要用到的 jar 包。注意，这里的版本是 1.5.6.RELEASE。

在我们引入了第 8~11 行的依赖包后，我们就可以把本项目配置成 Spring MVC 了，比如通过 @RestController 来定义控制器。注意，在第 5 行里，我们已经定义了父类依赖包的版本号，这里就不必再重复定义了。

在第 12~14 行中，我们引入了 Spring data jpa 所必需的依赖包，其实就是所必需的 jar 文件。在第 15 到 19 行中，引入了 mysql 的驱动包。

在本项目里用到的 jar 包都存在于本地 Maven 仓库里，一旦在本项目的 pom.xml 里指定了要用到哪些 jar 包，就将根据具体指定的 groupId 和 artifactId 引用本地仓库里对应的包。

比如本机的 maven 本地仓库的路径是 C:\Documents and Settings\Administrator\.m2\repository，而在 pom.xml 里配置 mysql 依赖包的代码如下：

```
1   <groupId>mysql</groupId>
2   <artifactId>mysql-connector-java</artifactId>
3   <version>5.1.3</version>
```

那么本项目就会引用 Maven 本地仓库路径\mysql\mysql-connector-java\5.1.3 目录下的 jar 包，如图 2.2 所示。其中，路径中的 mysql 和 groupId 相一致，mysql-connector-java 和 artifactId 相一致，5.1.3 和 version 相一致。

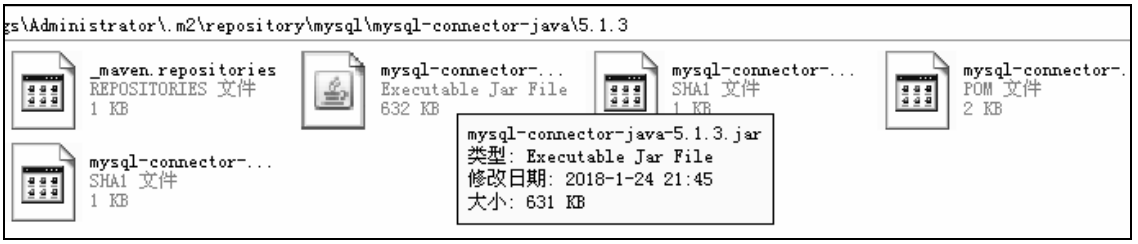

图 2.2　Maven 里被引用的 jar 实际位置示意图

同理，大家可以找到本项目引用到的 jpa 包的实际位置。

如果在本地仓库里找不到所需要的 jar 包，那么 Maven 会自动到远端仓库去下载 jar 包放置到本地仓库，比如本项目里用到的 spring-boot-starter-web 版本是 1.5.6.RELEASE，如果本地没有，大家还能看到从远端仓库（一般是一个能提供各种 Maven 包的网站）下载的这个过程。

3．编写启动程序和控制器类

DataServerApp.java 的代码如下，在其中的第 5 行里，我们编写了启动代码。不过请注意，它是放在 jpademo 这个 package 里的。

```
1   @SpringBootApplication
2   public class DataServerApp{
3       public static void main( String[] args )
4       {
5           SpringApplication.run(DataServerApp.class, args);
6       }
7   }
```

在 studentController 里，我们放置了控制器部分的代码，在其中我们通过@RequestMapping 注解来指定 request 请求和待调用方法的对应关系。

```
1   @RestController //用这个注解说明该类是控制器
2   @RequestMapping(value = "/students")//指定基础路径
3   public class studentController {
4       @Autowired //将自动引入 studentService
5       private StudentService studentService;
6       @RequestMapping(value = "/find/name/{name}")
7       public List<Student> getStudentByName(@PathVariable String name)    {
8           List<Student> students = studentService.findByName(name);
9           return students;
10      }
11      @RequestMapping(value = "/nameAndscore/{name}/{score}" )
12      public List<Student> findByNameAndScore(@PathVariable String name,
        @PathVariable float score) {
13      List<Student> students = studentService.findByNameAndScore(name,
         score);
14          return students;
15      }
16  }
```

在第 7~10 行里，定义了将被/find/name/{name}格式 url 触发的 getStudentByName 方法，其中是调用 service 类的方法，返回指定 name 的学生信息。

在第 11~15 行里，我们定义了可以被“/nameAndscore/{name}/{score}”这种 url 格式触发的 findByNameAndScore 方法，在其中，同样是通过调用 service 层的方法返回指定 name 和 score 的学生成绩。

在前文里已经提到，@SpringBootApplication 注解包含了@ComponentScan，通过后者这个注解，我们能设置 Spring 容器的扫描范围。如果不设置，默认的扫描范围是本包（也就是 jpademo）以及它的子目录。

这里我们需要让容器扫描带有@RestController 的 studentController 类并把它设置成控制器类，如果把控制器类和 App.java 类设置成平级，那么容器会无法识别这个控制器，这就是为什么把控制器类包含在 jpademo 子目录里的原因。

同理，后面将要讲述的 StudentService.java 类，由于出现了@Autowired 注解，因此也希望被容器扫描到，所以我们同样需要把该类放在 jpademo 的子目录里。

4．编写 Service 类

在 StudentService.java 里，我们编写提供业务服务的代码，上文里已经提到，为了也能让容器扫描到它，需要把它放在 jpademo.servcie 包（处于 jpademo 的子目录中）里，代码如下：

```
1   package jpademo.service;
2   省略必要的 import 代码
3   @Service//自动注册到容器里
4   public class StudentService {
5       @Autowired //自动引入 Repository 类
6       private StudentRepository stuRepository;
7       //根据 name 查找
8       public List<Student> findByName(String name){
9          //调用 stuRepository 里的对应方法
10         return stuRepository.findByName(name);
11      }
12      //根据 name 和 score 查找
13      public List<Student> findByNameAndScore(String name,float score){
14          //同样也是调用 stuRepository 里的对应方法
15          return stuRepository.findByNameAndScore(name, score);
16      }
17  }
```

这个类里提供了两种服务方法，第 8 行的 findByName 方法实现了根据名字搜索的功能，第 13 行的 findByNameAndScore 方法实现了根据名字和分数搜索的功能。在这两个方法里，都是调用 StudentRepository 类型的 stuRepository 对象里的方法来实现功能的。

5．编写 Repository 类

在 JPA 里，一般是在 Repository 类放置连接数据库的业务代码，它的作用有些类似 DAO。这里我们将在 StudentRepository 类里实现在刚才 service 层里调动的两个操作数据库的方法。

```
1   package jpademo.repository;//同样放入 jpademo 的子目录
2   省略必要的 import 代码
3   @Component
4   //注意这是一个继承 Repository 的接口
5   public interface StudentRepository extends Repository<Student, Long>{
```

```
    //通过 Query 注解定义查询语句
    @Query(value = "from Student s where s.name=:name")
    List<Student> findByName(@Param("name") String name);
    //JPA 将根据这个方法自动拼装查询语句
    List<Student> findByNameAndScore(String name, float score);
}
```

这里大家会看到一个比较有意思的现象，我们在第 8 行和第 10 行定义的两种方法都没有方法体。事实上在 JPA 的底层实现里将会根据方法名以及注解自动地执行查询语句并返回结果。

具体而言，在第 8 行的 findByName 方法里，将会执行第 7 行@Query 注解所带的基于 Student 表的查询语句，并以 List<Student>的形式返回结果。在第 10 行的 findByNameAndScore 方法里，JPA 底层将解析方法名，以 Name 和 Score 这两个字段为条件查询 Student 表，同样以 List<Student>的形式返回结果。

我们这里只给出了常用的通过 equals 查询的例子，在表 2.3 里，我们能看到 JPA 支持的其他常用关键字。

表 2.3　JPA 里支持的常用关键字列表

关键字	方法名示例	等价的 where 条件
Equals	findBy 字段名 Equals	where 字段名=参数
And	findBy 字段 1And 字段 2	where 字段 1=参数 1 and 字段 2=参数 2
Or	findBy 字段 1Or 字段 2	where 字段 1=参数 1 or 字段 2=参数 2
Between	findBy 字段名 Between	where 字段 between 参数 1 and 参数 2
GreaterThan	findBy 字段名 GreaterThan	where 字段名>参数
LessThan	findBy 字段名 LessThan	where 字段名<参数

除此之外，JPA 还支持 isnull、like 和 OrderBy 等其他查询关键字，但在项目里，简单查询的 SQL 语句毕竟是少数，在大多数查询语句里，往往会带 3 个以上关键字，比如：

```
    select * from student where name=xxx and score>xxx and id in (xxx,xxx) order
by id asc
```

在类似复杂的场景里，就无法直接使用上述“字段名+关键字”形式的方法了，这时就可以通过@Query 引入较为复杂的 SQL 语句。注意，需要把 nativeQuery 设成 true。具体代码如下：

```
1   @Query(value = 复杂的 sql 语句, nativeQuery = true)
2   List<Student> findStudent(String name,float score,String ids);
```

6．在配置文件里设置连接数据库的参数

在 application.properties 文件里，我们配置了 MySQL 数据库的各项连接参数，代码如下：

```
1   spring.jpa.show-sql = true
2   spring.jpa.hibernate.ddl-auto=update
3   spring.datasource.url=jdbc:mysql://localhost:3306/springboot
4   spring.datasource.username=root
5   spring.datasource.password=123456
6   spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

在第 2 行里，我们设置了数据表的创建方式，这里是 update，在启动本项目时，Spring 容器会把本地的映射文件和数据表做个比较，如果有差别，就用本地映射文件里的定义更新数据表结构，

如果无差别，就什么也不做。这里如果没有特殊情况，不要用 create，因为 create 的含义是“删除后再创建”，这样会导致数据表的数据丢失。

在第 3~6 行中，我们定义了连接 url、用户名、密码和连接驱动等属性。

7．编写本地映射文件

由于 Spring data JPA 属于一种数据持久化映射技术，因此我们需要在本地开发一个能和 Student 数据表关联的 Model 对象，代码如下：

```
1   package jpademo.model;//为了被扫描到，同样是处于 jpademo 的子目录
2   //省略必要的 import 方法
3   @Entity
4   @Table(name="Student") //和 Student 数据表关联
5   public class Student {
6       @Id //通过@Id 定义主键
7       private String ID;
8       @Column(name = "Name")
9       private String name;
10      @Column(name = "Age")
11      private String age;
12      @Column(name = "Score")
13      private float score;
14      //省略必要的 get 和 set 方法
15  }
```

其中，我们通过第 3 行和第 4 行的注解来说明本类是用来映射 Student 表的；通过第 6 行的@Id 注解，我们指定了第 7 行的 ID 属性是用来映射表里的主键 ID 的；通过类似于第 8 行的@Column 注解，后面我们一一指定了本类里属性和 Student 数据表里的对应关系。

8．查看运行结果

至此，代码编写完成。运行前，我们需要到 student 表里插入一条 name 是 tom、score 是 100.0 的数据。通过 DataServerApp.java 启动 web 服务后，在浏览器里输入“http://localhost:8080/students/find/name/tom”，就会触发 Controller 层里的 getStudentByName，在浏览器里能看到如下所示的结果。

[{"name":"tom","age":"12","score":100.0,"id":"1"}]

如果输入“http://localhost:8080/students/nameAndscore/tom/100”，就调用 findByNameAndScore 方法，也能看到同样的结果。

2.2.2 使用 yml 格式的配置文件

在刚才的例子里，我们是把配置文件写在.properties 文件里，在项目里，我们还可以使用扩展名是 yml 的 YAML 文件来存放配置信息。

和传统的配置文件相比，yml 文件结构性比较强，比较容易被理解，在企业级系统里也被广泛应用。

这里我们将在刚才 SpringBootJPAMySQLDemo 项目的基础上稍做修改，在其中将会用到 yml 文件来存放数据库的连接信息。

代码位置	视频位置
代码\第 2 章\SpringBootJPAYMLDemo	视频\第 2 章\YML 配置文件演示

在这个项目里，需要去掉 application.properties 文件，在相同的位置添加一个 application.yml 文件，代码如下：

```
spring:
  jpa:
    show-sql: true
    hibernate:
      dll-auto: update
  datasource:
    url: jdbc:mysql://localhost:3306/springboot
    username: root
    password: 123456
    driver-class-name: com.mysql.jdbc.Driver
```

在上述文件里，我们能看到 yml 是用缩进来定义层级关系的。其中，第 1~3 行的代码等价于 spring.jpa.show-sql = true，其他的配置信息以此类推。而且，建议在定义属性的冒号后面空一格再定义属性的值。

2.2.3　通过 profile 文件映射到不同的运行环境

我们在项目里经常会根据不同的运行环境使用不同的配置信息，比如在测试环境里连接测试数据库，在生产环境里连接生产库，又如，在测试和生产环境里往不同的位置输出日志信息。通过 profile，我们能轻易地实现这种效果。

代码位置	视频位置
代码\第 2 章\SpringBootJPAProfileDemo	视频\第 2 章\通过 profile 文件映射到不同环境

这个项目是在 2.2.2 小节的 SpringBootJPAYMLDemo 项目基础上修改而成的，这里我们将为 QA 和 PROD 环境配置不同的数据库连接参数。

修改点 1，在 application.yml 里设置 QA 和 PROD 两个环境的配置信息，代码如下：

```
spring:
  profiles: QA
  jpa:
    show-sql: true
    hibernate:
      dll-auto: create
  datasource:
    url: jdbc:mysql://localhost:3306/springboot
    username: root
    password: 123456
    driver-class-name: com.mysql.jdbc.Driver
---
spring:
  profiles: PROD
  jpa:
    show-sql: false
```

```
17      hibernate:
18        dll-auto: update
19    datasource:
20      url: jdbc:mysql://localhost:3306/springboot
21      username: root
22      password: 123456
23      driver-class-name: com.mysql.jdbc.Driver
```

其中，第 1~11 行配置的是 QA 环境的信息，第 13~23 行配置的是 PROD，中间用第 12 行的横线分隔，这个分隔符纯粹是为了提升可读性，开发中可以不加这个内容。上述代码的关键是在第 2 行和第 14 行里，用 spring.profiles = XX 的形式来指定该段代码的作用域。

修改点 2，在启动文件 App.java 里，修改代码如下：

```
1   //省略必要的 package 和 import 代码
2   @SpringBootApplication
3   public class App
4   {
5       public static void main( String[] args ){
6           ConfigurableApplicationContext context =
            new SpringApplicationBuilder(App.class).properties(
            "spring.config.location=classpath:/application.yml")
            .properties("spring.profiles.active=QA").run(args);
7       }
8   }
```

这里通过第 6 行的代码以.properties("spring.profiles.active=XX")的形式指定该以 QA 或 PROD 模式启动服务，从而指定本程序读取的是测试还是生产环境的数据库连接参数。

2.3 通过 JPA 实现各种关联关系

在实际项目里，我们会关联查询多张数据表，从中获得必要的业务数据，对应地，我们也可以通过 JPA 把基于多表的各种关联关系映射到 Model 类里。

具体而言，表之间的关联关系可以是一对一、一对多或多对多，通过 JPA，我们能用比较简单的方式来实现这些关联关系。

2.3.1 一对一关联

代码位置	视频位置
代码\第 2 章\SpringBootJPAOne2OneDemo	视频\第 2 章\JPA 一对一关联演示

在这个业务场景里，我们让一个学生（Student）只能拥有一张银行卡（Card），具体而言，学生和银行卡之间是一对一关联。

步骤01 创建学生和银行卡这两张数据表。学生表的结构如表 2.4 所示，其中用 cardID 来表示该学生所拥有的银行卡号。

表 2.4　一对一关联里的 Student 表结构

字段名	类型	含义
id	varchar	主键，学号
name	varchar	姓名
age	varchar	年龄
score	float	成绩
cardID	varchar	对对应的银行卡号

描述银行卡的 Card 表结构如表 2.5 所示。

表 2.5　一对一关联里的 Card 表结构

字段名	类型	含义
cardID	varchar	卡号，与 Student 表里的 cardID 关联
balance	float	余额

步骤02 在 pom.xml 里描述本项目的依赖包。在这个项目里，我们将和之前的项目一样，依赖 JPA、Spring Boot 以及 MySQL 的 jar 包，所以就不再给出详细的代码了。

步骤03 在 application.yml 里配置 jpa 以及 mysql 数据库连接的信息，代码如下：

```
spring:
  jpa:
    show-sql: true
    hibernate:
      dll-auto: update
  datasource:
    url: jdbc:mysql://localhost:3306/springboot
    username: root
    password: 123456
    driver-class-name: com.mysql.jdbc.Driver
```

这里同样要注意缩进，而且这里代码的具体含义在之前的项目介绍里都解释过，所以就不再额外解释了。

步骤04 编写用来映射数据表的学生和银行卡的 Model 类，其中 Student.java 的代码如下：

```
//省略必要的 package 和 import 代码
@Entity
@Table(name="Student") //映射到 MySQL 里的 Student 表
public class Student {
    @Id //主键
    private String ID;
    @Column(name = "Name")//通过@Column 指定映射的列名
    private String name;
    @Column(name = "Age")
    private String age;
    @Column(name = "Score")
    private float score;
    //通过@OneToOne 来指定和 Card 的一对一关联关系
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "cardID", unique=true)
```

```
16      private Card card;
17      //省略必要的 get 和 set 方法
18  }
```

在上述代码的第 14~16 行中，通过@OneToOne 的注解指定了 Student 和 Card 的一对一关联，其中通过第 15 行的@JoinColumn 来表示是通过 cardID 来关联到 Card 表的。

Card.java 代码如下，这个类比较简单，通过第 2 行和第 3 行的@Entity 和 Table 注解来指定待关联的数据表名，通过第 5 行的@Id 来指定主键，通过第 7 行的@Column 来指定对应的列名。

```
1   //省略必要的 package 和 import 代码
2   @Entity
3   @Table(name="Card")//指定关联到 Card 表
4   public class Card {
5       @Id
6       private String cardID;//指定主键
7       @Column(name = "balance")//指定映射的列名
8       private float balance;
9       //省略必要的 get 和 set 方法
10  }
```

步骤05 编写控制器类 StudentController.java，具体代码如下：

```
1   //省略必要的 package 和 import 代码
2   @RestController //指定本类是控制器类
3   @RequestMapping(value = "/students")
4   public class StudentController {
5       @Autowired
6       private StudentService studentService;
7       @RequestMapping(value = "/one2oneDemo")
8       public void one2oneDemo() {
9           studentService.one2oneDemo();
10      }
11  }
```

在上述代码的第 7 行和第 8 行里，我们能看到，/one2oneDemo 格式的请求将触发 one2oneDemo 方法，在这个方法里，将调用 service 层的对应方法。

步骤06 编写实现 Service 层功能的 StudentService.java，代码如下：

```
1   //省略必要的 package 和 import 代码
2   @Service
3   public class StudentService {
4       @Autowired
5       private StudentRepository stuRepository;
6       public void one2oneDemo() {
7           //创建一个学生
8           Student s = new Student();
9           s.setID("1");
10          s.setName("Peter");
11          s.setScore(100);
12          s.setAge("12");
13          //创建一张卡
14          Card card = new Card();
15          card.setCardID("card1");
```

```
16          card.setBalance(200);
17          s.setCard(card);
18          //保存学生信息
19          stuRepository.save(s);
20          //通过学生找到卡，并打印卡信息
21          Student peter = stuRepository.findByName("Peter").get(0);
22          System.out.println(peter.getCard().getCardID());
23          System.out.println(peter.getCard().getBalance());
24           //删除学生后，卡信息也会一并被删除
25           stuRepository.delete(s);
26      }
27  }
```

在上述代码里，我们能看到学生和银行卡之间的关联关系。具体而言，当我们在第 19 行 save 学生信息后，能在第 21 行通过 name 找到该学生所对应的卡，在第 22 行和第 23 行里，能打印出对应的卡信息。

由于之前设置的学生和银行卡之间的级联关系（CascadeType）是 ALL，其中也包含“删除”，因此在第 25 行里，当我们通过 delete 语句删除学生信息后，就能发现 card 表里和该学生对应的银行卡记录也会被删除。

步骤07 实现 StudentRepository 接口，在其中实现针对数据库的操作，具体代码如下：

```
1   //省略必要的 package 和 import 代码
2   @Component
3   public interface StudentRepository extends JpaRepository<Student, Long>{
4       @Query(value = "from Student s where s.name=:name")
5       List<Student> findByName(@Param("name") String name);
6
7   }
```

我们在第 4 行和第 5 行的代码里，实现了根据 name 查找 Student 对象的功能，至于在 Service 层里调用的 save 和 delete 方法，则是封装在 JpaRepository 类里的，我们无须编写。

最后，我们还得在 App.java 里实现 SpringBoot 的启动代码，这块我们之前已经提到过，所以就不再解释了。

```
1   @SpringBootApplication
2   public class App{
3       public static void main( String[] args )
4       {
5           SpringApplication.run(App.class, args);
6       }
7   }
```

至此，当我们通过 App.java 启动 Spring Boot 时，就能通过在浏览器里输入如下 url 来查看效果了。

```
1       http://localhost:8080/students/one2oneDemo
```

根据 Controller 层的定义，该 url 请求会触发 Service 层里的 one2oneDemo 方法，大家如果查看数据库，就能看到“插入学生后对应的银行卡信息也能自动插入”以及“删除学生后对应的卡也会自动删除”的级联操作效果。

2.3.2 一对多关联

代码位置	视频位置
代码\第 2 章\SpringBootJPAOne2ManyDemo	视频\第 2 章\JPA 一对多关联

这里，我们将实现一个用户（User）拥有多辆汽车（Car）的业务场景。其中，用户表的结构如表 2.6 所示，描述汽车的 Car 表结构如表 2.7 所示。

表 2.6 一对多关联里的 User 表结构

字段名	类型	含义
userID	Int	用户 ID，主键，自增长
Name	varchar	用户姓名

表 2.7 一对多关联里的 Car 表结构

字段名	类型	含义
carID	int	汽车 ID，主键，自增长
price	float	汽车价格
userID	int	用户 ID，外键，与 User 表关联

在创建完 Maven 类型的 SpringBootJPAOne2ManyDemo 项目后，在其中的 pom.xml 里，我们将和之前的项目一样，同样引入 JPA、Spring Boot 以及 MySQL 的 jar 包。

由于这里连接的数据库和之前“2.3.1”小节中的一致，因此 application.yml 用的是和之前一样的代码。

在 User.java 和 Car.java 这两个 Model 类里，我们将定义一对多关联关系，其中 User.java 的代码如下：

```
1   //省略必要的 package 和 import 代码
2   @Entity
3   @Table(name="User") //指定关联到 User 表
4   public class User {
5       @Id
6       @Column(name="userID") //定义主键
7       @GeneratedValue(strategy = GenerationType.IDENTITY)
8       private int userID;
9       @Column(name = "name")
10      private String name;
11      //通过@OneToMany 定义一对多关联
12      @OneToMany(cascade = CascadeType.ALL,mappedBy = "user")
13      private Set<Car> cars;
14      //省略必要的 get 和 set 方法
15  }
```

在第 13 行里，我们通过 Set 类来存放一个用户拥有的多辆汽车。在第 12 行里，我们通过@OneToMany 注解定义了“一个用户拥有多辆车”的关系。这里 cascade 的级联关系是 ALL，也就是说，一旦从数据表里删除这个用户，那么对应的汽车也会从数据表里被删除；mappedBy 的取值是 user，也就是说，在 Car 类里使用过这个属性来指定车的主人。

描述汽车类的 Car.java 的代码如下：

```
1   //省略必要的 package 和 import 代码
2   @Entity
3   @Table(name="Car") //和 Car 表相关联
4   public class Car {
5       @Id
6       @Column(name="carID") /主键
7       @GeneratedValue(strategy = GenerationType.IDENTITY)
8       private int carID;
9       @Column(name = "price")
10      private float price;
11      @ManyToOne(cascade = CascadeType.ALL)
12      @JoinColumn(name="userID")
13      private User user;
14      //省略必要的 get 和 set 方法
15  }
```

在这里的第 11~13 行里，通过@ManyToOne 的注解来定义汽车和用户的关联关系，其中用第 12 行的@JoinColumn 来指定 Car 类是通过 userID 这个属性和 User 类关联的，第 13 行定义的 user 类则指定了这个 Car 的主人。

在 userController.java 里，我们定义了这个 Spring Boot 项目的“控制器类”，具体代码如下：

```
1   //省略必要的 package 和 import 代码
2   @RestController //指定该类是控制器类
3   @RequestMapping(value = "/users")
4   public class userController {
5      @Autowired
6      private UserService userService;
7      @RequestMapping(value = "/one2manyDemo")
8       public void one2manyDemo() {
9           userService.one2manyDemo();
10     }
11  }
```

在第 7 行里，我们通过@RequestMapping 注解定义了触发该方法的 url 格式，在第 8 行的 one2manyDemo 方法里，调用了 service 层里的 one2manyDemo 方法。下面我们来看一下 UserService.java 这段代码。

```
1   //省略必要的 package 和 import 代码
2   @Service //指定本类是 Service
3   public class UserService {
4      @Autowired
5      private UserRepository userRepository;
6      public void one2manyDemo(){
7          //创建两个 Car 对象
8          Car car1 = new Car();
9          car1.setPrice(100);
10         Car car2 = new Car();
11         car2.setPrice(200);
12         //创建一个用户
13         User user = new User();
14         user.setName("Peter");
15         //设置两辆车的主人是 Peter
16         car1.setUser(user);
```

```
        car2.setUser(user);
        //定义一个 Set，放入两辆车
        Set<Car> cars = new HashSet<Car>();
        cars.add(car1);
        cars.add(car2);
        //给用户指定两辆车
        user.setCars(cars);
        //通过 save 方法存入用户
        userRepository.save(user);
        //先注释掉这行代码
        //userRepository.delete(user);
    }
}
```

在上述代码的第 8~23 行里，我们定义了一个用户和两辆车，并设置了“Peter”拥有两辆车的一对多关系。当我们在第 25 行通过 save 方法存入用户时，不仅能在 User 表里看到对应的用户信息，还能在 Car 表里看到关联的两辆车也被插入了。

如果我们打开第 27 行的注释，就会发现虽然我们只是通过 delete 方法删除了用户，但由于这里一对多的级联关系是 ALL，因此这个用户所对应的两辆车也会被从 Car 数据表里删除。

在上述 UserService.java 里，我们事实上是调用了 UserRepository 这个和 JPA 有关类里的方法，在这个 Repository 接口里，我们只是继承了 JpaRepository，在其中什么都没做，具体代码如下：

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

也就是说，在 Service 层里，我们使用了 JpaRepository 里自带的 save 和 delete 方法。

最后，我们还得编写该 Spring Boot 的启动类 App.java，代码如下：

```
//省略必要的 pacage 和 import 代码
@SpringBootApplication
public class App{
    public static void main( String[] args )
    {
      SpringApplication.run(App.class, args);
    }
}
```

当我们启动上述 App.java，并在浏览器里输入“http://localhost:8080/users/one2manyDemo”后，就会触发 UserService 类里的 one2manyDemo 方法，从而看到本案例的演示效果。

2.3.3 多对多关联

代码位置	视频位置
代码\第 2 章\SpringBootJPAMany2ManyDemo	视频\第 2 章\JPA 多对多关联

这里，我们将实现多本图书（Book）和多名作者（Author）之间的多对多关系，具体而言，一本书可以有多名作者，同一作者可以写多本书。

在表 2.8 中，我们定义了描述图书的 Book 表。

表 2.8　多对多关联里的 Book 表结构

字段名	类型	含义
bookID	int	ID，主键
Name	varchar	图书的名字

描述作者的 Author 表结构如表 2.9 所示。

表 2.9　多对多关联里的 Author 表结构

字段名	类型	含义
authorID	int	ID，主键
name	varchar	作者的姓名

同时，我们还需要创建 book_author 表来描述书和作者的多对多关联，结构如表 2.10 所示。

表 2.10　多对多关联里的 book_author 表结构

字段名	类型	含义
authorID	int	作者 ID
bookID	int	图书 ID

在创建完 Maven 类型的 SpringBootJPAMany2ManyDemo 项目后，在其中的 pom.xml 里，我们将和之前的项目一样，同样引入 JPA、Spring Boot 以及 MySQL 的 jar 包。

在 Book.java 和 Author.java 这两个 Model 类里，我们将定义多对多关联关系。其中，Book.java 的代码如下：

```
1   //省略必要的 package 和 import 代码
2   @Entity
3   @Table(name="Book")//和 Book 表相关联
4   public class Book {
5       @Id  //主键
6       private int bookID;
7       @Column(name = "name")
8       private String name;
9       //定义多对多关联
10      @ManyToMany(cascade = CascadeType.ALL)
11      @JoinTable(name = "book_author", joinColumns = {
12      @JoinColumn(name = "bookID", referencedColumnName = "bookID")},
        inverseJoinColumns = {
13      @JoinColumn(name = "authorID", referencedColumnName = "authorID")})
14      private Set<Author> authors;
15      //省略对应的 get 和 set 方法
16  }
```

在第 10 行中，我们定义了图书和作者的多对多关联；在第 11~13 行中，定义了 book_author 表里分别用 bookID 和 authorID 来描述双方的多对多关系；在第 14 行中，通过 Set 来描述这本图书里的多名作者信息。

描述作者类的 Author.java 的代码如下，其中通过第 10 行的@ManyToMany 注解来定义作者和图书的多对多关联，通过第 11 行定义 Set 类型的 books 属性来存放作者所写的多本书。

```
1  //省略必要的 package 和 import 代码
2  @Entity
3  @Table(name="Author") //指定该类和 Author 表相关联
4  public class Author {
5      @Id  //主键
6      private int authorID;
7      @Column(name = "name")
8      private String name;
9      //指定 Author 和 Book 的多对多关联
10     @ManyToMany(mappedBy = "authors")
11     private Set<Book> books;
12     //省略必要的 get 和 set 方法
13 }
```

在 Controller.java 里，我们定义“控制器类”，具体代码如下：

```
1  //省略必要的 package 和 import 代码
2  @RestController //定义控制器类
3  @RequestMapping(value = "/books")
4  public class Controller {
5      @Autowired
6      private BookService bookService;
7      @RequestMapping(value = "/many2manyDemo")
8      public void many2manyDemo() {
9          bookService.many2manyDemo();
10     }
11 }
```

其中，在第 7 行中，我们通过@RequestMapping 注解定义了触发该方法的 url 格式；在第 8 行的 many2manyDemo 方法中，调用了 service 层里的对应方法。下面我们来看一下 bookService.java 代码。

```
1  //省略必要的 package 和 import 代码
2  @Service
3  public class BookService {
4      @Autowired
5      private BookRepository bookRepository;
6      @Autowired
7      private AuthorRepository authorRepository;
8      public void many2manyDemo()
9      {
10         //定义三位作者
11         Author author1 = new Author();
12         author1.setAuthorID(1);
13         author1.setName("Peter");
14         Author author2 = new Author();
15         author2.setAuthorID(2);
16         author2.setName("Tom");
17         Author author3 = new Author();
18         author3.setAuthorID(3);
19         author3.setName("Ben");
20         //创建两本书
21         Book javaBook = new Book();
22         javaBook.setBookID(1);
```

```
23          javaBook.setName("Java");
24          Book dbBook = new Book();
25          dbBook.setBookID(2);
26          dbBook.setName("Oracle");
27          //通过两个 set 存放 Java 书和 DB 书的作者
28          Set<Author> javaAuthors = new HashSet<Author>();
29          javaAuthors.add(author1);
30          javaAuthors.add(author3);
31          Set<Author> dbAuthors = new HashSet<Author>();
32          dbAuthors.add(author2);
33          dbAuthors.add(author3);
34          //设置 Java 书和 DB 书的作者
35          javaBook.setAuthors(javaAuthors);
36          dbBook.setAuthors(dbAuthors);
37          //保存 java 书和 DB 书
38          bookRepository.save(javaBook);
39          bookRepository.save(dbBook);
40      }
41  }
```

在上述代码的第 10~36 行里，我们完成了如下动作。

第一，定义了 3 名作者信息。
第二，创建了 java 和 DB 两本书的信息。
第三，定义了两个 Set，在其中存放了两本书的作者信息。
第四，给两本书设置了对应 Set，以此指定两本书的作者。

在第 38~39 行中，我们通过 save 方法保存了两本书，此时我们能看到如下效果。

第一，在 Book 表里能看到 Java 和 DB 图书的信息。
第二，在 Author 表里，能看到 3 名作者的信息。
第三，在 book_author 表里，能看到图书和作者的对应关系。

在上述的 Service 类里，我们事实上是调用了 BookRepository 和 AuthorRepository 这两个和 JPA 有关的类中的方法。同样地，在这两个类里我们只是继承了 JpaRepository 这个接口，在其中什么都没做。BookRepository 类的具体代码如下：

```
1   @Component
2   public interface BookRepository extends JpaRepository<Book, Long>
3   { }
```

AuthorRepository 类的代码如下：

```
1   @Component
2   public interface AuthorRepository extends JpaRepository<Author, Long>
3   { }
```

也就是说，在 Service 层里，我们也是使用了 JpaRepository 里自带的 save 方法。
最后，我们还得编写该 Spring Boot 的启动类 App.java，代码如下：

```
1   //省略必要的 pacage 和 import 代码
2   @SpringBootApplication
3   public class App{
```

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

当我们启动上述 App.java，并在浏览器里输入“http://localhost:8080/books/many2manyDemo”后，就会触发 BookService 类里的 many2manyDemo 方法，从而看到本案例的演示效果。

2.4 本 章 小 结

通过本章的学习，大家能发现通过 JPA 能比较方便地开发基于 MySQL 的数据库业务代码，事实上，只要我们在.yml 文件里修改对应的连接驱动、连接 URL、数据库用户名和密码，就能用非常相似的代码来访问 Oracle 或 SQL Server 等其他数据库。

在本章中，还讲述了通过 JPA 实现一对一、一对多和多对多等关联场景的方式。在真实的项目里，可能业务场景要比这复杂，但开发步骤是一致的。换句话说，在学完本章后，大家能用同样的方法很快地实现各类真实的“关联”业务。

第 3 章

服务治理框架：Eureka

在微服务项目里，我们需要关注能带来实际价值的业务功能，但同时还得考虑“微服务如何发布”以及“如何让客户发现并调用微服务”这类面向基础设施的问题。

我们固然可以自己开发一套“管理微服务”的框架，但这样势必会增加项目的开发周期和成本，事实上 Eureka 框架已经提供了上述功能。具体而言，在服务器端，我们能通过 Eureka 服务治理框架发布和注册服务；在客户端，我们可以用此发现并调用微服务。

不仅如此，在高并发的场景里，我们还可以配置 Eureka 集群，即在多台机器上配置 Eureka，以此来适应常见的“负载均衡”和“故障转移”需求。

3.1 了解 Eureka 框架

Eureka 是 Spring Cloud Netflix 全家桶中的一个组件，在有些资料里，它也被称为“服务发现框架”。不管叫什么名字，我们都可以通过它来注册、发布、发现和调用服务。

3.1.1 Eureka 能干什么

在项目里，一般存在“服务提供者”和“服务调用者”两种角色，为了调到服务，服务提供者需要服务调用者知道“服务所在的 IP 地址、端口号和提供服务的方法名”这些关键信息，如果服务比较多，那么该如何维护这些服务信息呢？

比较直观的解决方案是“用静态的方式来管理服务列表”，比如在一个配置文件里放入所有的服务清单，包括刚才提到的 IP 地址、端口号和方法名，但这未必是一种好的选项。

一方面，如果系统里服务提供模块的数量很多，那么这类配置文件就会很长，这样可读性就

会很差，从而导致该文件很难维护。另一方面，随着项目的不断深入，服务提供模块一定是会不断变更的，在配置文件中的服务列表信息也需要随之不断更改。这不仅增加了系统的维护难度，还会提升诸如命名冲突这类问题的风险。

Eureka 组件为此提供了一套较好的解决方案。

第一，服务提供者可以向 Eureka 注册中心注册本模块可以提供的服务。

第二，服务调用者能从 Eureka 注册中心查找（也就是发现）和调用所需的服务。

第三，大家可以把 Eureka 理解成第三方的服务管理平台。一旦有新的服务生成或有旧的服务失效，Eureka 能做到自动更新服务列表，这就降低了因服务不断变更而导致的项目维护成本。

3.1.2 Eureka 的框架图

从图 3.1 中，我们能看到 Eureka 的基本架构。

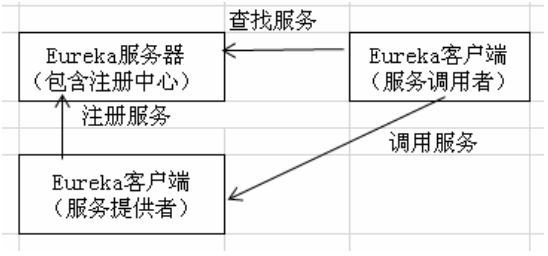

图 3.1 Eureka 的基本框架

在 Eureka 的服务器里，包含着记录当前所有服务列表的注册中心，而服务提供者和调用者所在的机器均被称为“Eureka 客户端”。

服务提供者会和服务器进行如下交互：第一，注册本身能提供的服务；第二，定时发送心跳，以此证明本服务处于生效状态。服务调用者一般会从服务器查找服务，并根据找到的结果从服务提供者端调用服务。

3.2 构建基本的 Eureka 应用

在这一部分，我们将编写 Eureka 的服务器、服务提供者和调用者的代码，并通过它们之间的交互来向大家演示 Eureka 的开发步骤和工作流程。

3.2.1 搭建 Eureka 服务器

这里我们将在 EurekaBasicDemo-Server 项目里编写 Eureka 服务器的代码。

代码位置	视频位置
代码\第 3 章\EurekaBasicDemo-Server	视频\第 3 章\搭建 Eureka 服务器

第一步，当我们创建完 Maven 类型的项目后，需要在 pom.xml 里编写该项目所需要的依赖包，关键代码如下：

```
1    <dependencyManagement>
2       <dependencies>
3           <dependency>
4               <groupId>org.springframework.cloud</groupId>
5               <artifactId>spring-cloud-dependencies</artifactId>
6               <version>Brixton.SR5</version>
7               <type>pom</type>
8               <scope>import</scope>
9           </dependency>
10      </dependencies>
11   </dependencyManagement>
12   <dependencies>
13     <dependency>
14        <groupId>org.springframework.cloud</groupId>
15        <artifactId>spring-cloud-starter-eureka-server</artifactId>
16     </dependency>
17 </project>
```

在第 1~11 行的代码中，我们引入了版本号是 Brixton.SR5 的 Spring Cloud 包，这个包里包含着 Eureka 的支持包，在第 13~16 行的代码中，引入了 Eureka Server 端的支持包，引入后，我们才能在项目的 java 文件里使用 Eureka 组件的特性。

第二步，在 application.yml 里，需要配置 Eureka 服务端的信息，代码如下：

```
1   server:
2     port: 8888
3   eureka:
4     instance:
5       hostname: localhost
6     client:
7       register-with-eureka: false
8       fetch-registry: false
9       serviceUrl:
10        defaultZone: http://localhost:8888/eureka/
```

在第 2 行和第 5 行的代码中，我们指定了 Eureka 服务端使用的主机地址和端口号，这里分别是 localhost 和 8888，也就是说让服务端运行在本地的 8888 号端口。在第 10 行中，我们指定了服务端所在的 url 地址。

由于这已经是服务器端，因此我们通过第 7 行的代码指定无须向 Eureka 注册中心注册自己，同理，服务器端的职责是维护服务列表而不是调用服务，所以通过第 8 行的代码指定本端无须检索服务。

第三步，在 RegisterCenterApp.java 里编写 Eureka 启动代码。

```
1   //省略必要的 package 和 import 代码
2   @EnableEurekaServer //指定本项目是 Eureka 服务端
3   @SpringBootApplication
4   public class RegisterCenterApp
5   {
6       public static void main( String[] args )
7       {
```

```
8           SpringApplication.run(RegisterCenterApp.class, args);
9       }
10  }
```

在第 6 行的 main 函数里，我们还是通过 run 方法启动 Eureka 服务。

运行 App.java 启动 Eureka 服务器端后，在浏览器里输入“localhost:8888”后，可以看到如图 3.2 所示的 Eureka 服务器端的信息面板，其中 Instances currently registered with Eureka 目前是空的，说明尚未有服务注册到本服务器的注册中心。

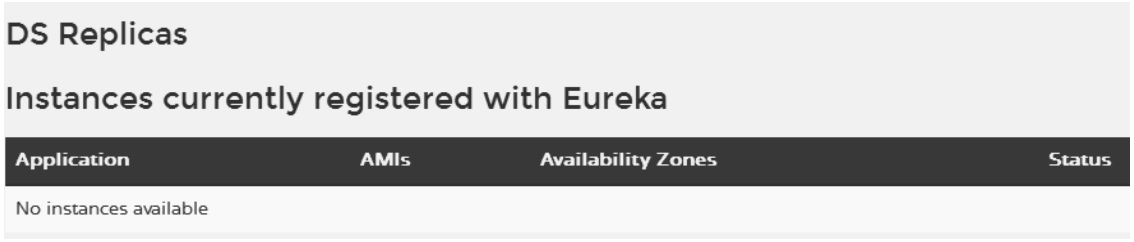

图 3.2　Eureka 服务器端的信息面板示意图

3.2.2　编写作为服务提供者的 Eureka 客户端

这里我们将在 EurekaBasicDemo-ServerProvider 项目里编写 Eureka 客户端的代码。在这个项目里，我们将提供一个 SayHello 的服务。

代码位置	视频位置
代码\第 3 章\EurekaBasicDemo-ServerProvider	视频\第 3 章\搭建提供服务的 Eureka 客户端

第一步，创建完 Maven 类型的项目后，我们需要在 pom.xml 里写入本项目的依赖包，关键代码如下。本项目所用到的依赖包之前都用过，所以这里就不展开讲了。

```
1   <dependencyManagement>
2       <dependencies>
3           <dependency>
4               <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
5               <version>Brixton.SR5</version>
6               <type>pom</type>
7               <scope>import</scope>
8           </dependency>
9       </dependencies>
10    </dependencyManagement>
11    <dependencies>
12      <dependency>
13            <groupId>org.springframework.boot</groupId>
14            <artifactId>spring-boot-starter-web</artifactId>
15            <version>1.5.4.RELEASE</version>
16      </dependency>
17      <dependency>
18          <groupId>org.springframework.cloud</groupId>
19          <artifactId>spring-cloud-starter-eureka</artifactId>
20      </dependency>
```

```
21    </dependencies>
```

第二步，在 application.yml 里编写针对服务提供者的配置信息，代码如下：

```
1   server:
2     port: 1111
3   spring:
4     application:
5       name: sayHello
6   eureka:
7     client:
8       serviceUrl:
9         defaultZone: http://localhost:8888/eureka/
```

从第 2 行里，我们能看到本服务将启用 1111 号端口；在第 5 行中，我们指定了本服务的名字，叫 sayHello；在第 9 行中，我们把本服务注册到了 Eureka 服务端，也就是注册中心里。

第三步，在 Controller.java 里编写控制器部分的代码，在其中实现对外的服务。

```
1   //省略必要的 package 和 import 代码
2   @RestController //说明这是一个控制器
3   public class Controller {
4       @Autowired //描述 Eureka 客户端信息的类
5       private DiscoveryClient eurekaClient;
6       @RequestMapping(value = "/hello/{username}",
         method = RequestMethod.GET )
7       public String hello(@PathVariable("username") String username) {
8           ServiceInstance inst = eurekaClient.getLocalServiceInstance();
9           //输出服务相关的信息
10          System.out.println("host is:" + inst.getHost());
11          System.out.println("port is:" + inst.getPort());
12          System.out.println("ServiceID is:" + inst.getServiceId() );
13          System.out.println("url path is:" + inst.getUri().getPath());
14          //返回字符串
15          return "hello " + username;
16      }
17  }
```

我们通过第 6 行和第 7 行的代码指定了能触发 hello 方法的 url 格式，在这个方法里，我们首先通过第 10~13 行的代码输出了主机名、端口号和 ServiceID 等信息，并在第 15 行里返回了一个字符串。

第四步，编写 Spring Boot 的启动类 ServiceProviderApp.java，代码如下：

```
1   //省略必要的 package 和 import 代码
2   @SpringBootApplication
3   @EnableEurekaClient
4   public class ServiceProviderApp {
5      public static void main( String[] args )
6      {
7        SpringApplication.run(ServiceProviderApp.class, args);
8      }
9   }
```

由于这是处于 Eureka 的客户端，因此加入第 3 行所示的注解，在 main 函数里，我们依然是通

过 run 方法启动 Spring Boot 服务。

3.2.3 编写服务调用者的代码

启动 Eureka 服务器端的 RegisterApp.java 和服务提供者端的 ServiceProviderApp.java，在浏览器里输入“http://localhost:8888/”后，在 Eureka 的信息面板里能看到 SayHello 服务，如图 3.3 所示。

图 3.3 在 Eureka 信息面板里能看到 SayHello 服务

这时在浏览器里输入“http://localhost:1111/hello/Mike”，就能直接调用服务，同时能在浏览器中看到“hello Mike”的输出。

不过在大多数的场景里，我们一般是在程序里调用服务，而不是简单地通过浏览器调用，在下面的 EurekaBasicDemo-ServiceCaller 项目里，我们将演示在 Eureka 客户端调用服务的步骤。

代码位置	视频位置
代码\第 3 章\EurekaBasicDemo-ServerCaller	视频\第 3 章\Eureka 服务调用端

第一步，在这个 Maven 项目里，编写如下的 pom.xml 配置，关键代码如下：

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>1.5.4.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-ribbon</artifactId>
    </dependency>
</dependencies>
```

请大家注意，从第 21~24 行的代码里，我们需要引入 ribbon 的依赖包，通过它我们可以实现负载均衡，在后继章节里，我们将详细讲述负载均衡的实现方式。其他的依赖包，我们之前都已经见过，所以就不再解释了。

第二步，在 application.yml 里，编写针对本项目的配置信息，代码如下：

```
1   spring:
2     application:
3       name: callHello
4   server:
5     port: 8080
6   eureka:
7     client:
8       serviceUrl:
9         defaultZone: http://localhost:8888/eureka/
```

在第 3 行里，我们指定了本服务的名字叫 callHello。在第 5 行里，我们指定了本服务是运行在 8080 端口。在第 9 行里，我们把本服务注册到 Eureka 服务器上。

第三步，编写提供服务的控制器类，在其中调用服务提供者提供的服务，代码如下：

```
1   //省略必要的 package 和 import 代码
2   @RestController
3   @Configuration
4   public class Controller {
5       @Bean
6       @LoadBalanced
7       public RestTemplate getRestTemplate()
8       {
9         return new RestTemplate();
10      }
11      @RequestMapping(value = "/hello", method = RequestMethod.GET   )
12      public String hello() {
13          RestTemplate template = getRestTemplate();
14          String retVal = template.getForEntity("http://sayHello/hello/
             Eureka", String.class).getBody();
15          return "In Caller, " + retVal;
16      }
17  }
```

在第 7 行的 getRestTemplate 方法上，我们启动了@LoadBalanced（负载均衡）的注解。

关于负载均衡的细节将在后面章节里详细描述，这里我们引入@LoadBalanced 注解的原因是，RestTemplate 类型的对象本身不具备调用远程服务的能力，如果我们去掉这个注解，程序未必能跑通。只有当我们引入该注解，该方法所返回的对象才能具备调用远程服务的能力。

在提供服务的第 12~16 行的 hello 方法里，我们通过第 14 行的代码，用 RestTemplate 类型对象的 getForEntity 方法调用服务提供者 sayHello 提供的 hello 方法。这里我们通过 http://sayHello/hello/Eureka 这个 url 去发现并调用对应的服务。在这个 url 里，只包含了服务名 sayHello，并没有包含服务所在的主机名和端口号。从中我们能看出，该 url 其实是通过注册中心定位到 sayHello 服务的物理位置的。至于这个 url 和该服务物理位置的绑定关系，是在 Eureka 内部实现的，这也是 Eureka 可以被称作“服务发现框架”的原因。

第四步，在 ServiceCallerApp.java 方法里，编写启动本服务的代码。这我们已经很熟悉了，所

以就不再讲述了。

```
1   //省略必要的 package 和 import 代码
2   @EnableDiscoveryClient
3   @SpringBootApplication
4   public class ServiceCallerApp
5   {
6       public static void main( String[] args )
7       {
8          SpringApplication.run(ServiceCallerApp.class, args);
9       }
10  }
```

3.2.4 通过服务调用者调用服务

当我们依次启动 Eureka 服务器（也就是注册中心）、服务提供者和服务调用者的 Spring Boot 启动程序后，在浏览器里输入“http://localhost:8888/”后，能在信息面板里看到两个服务，分别是服务提供者 sayHello 和服务调用者 callHello，如图 3.4 所示。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CALLHELLO	n/a (1)	(1)	UP (1) - 192.168.42.1:callHello:8080
SAYHELLO	n/a (1)	(1)	UP (1) - 192.168.42.1:sayHello:1111

图 3.4 在 Eureka 信息面板里能看到两个服务

由于服务调用者运行在 8080 端口上，如果我们在浏览器里输入“http://localhost:8080/hello”，能看到在浏览器中输出“In Caller, hello Eureka”，就说明它确实已经调用了服务提供者 sayHello 里的 hello 方法。

此外，我们还能在服务提供者所在的控制台里看到 host、port 和 ServiceID 的输出，如图 3.5 所示。这能进一步验证服务提供者控制器类里的 hello 方法被服务调用者调用了。

```
ServiceProviderApp (14) [Java App
2018-02-18 13:41:10.125
2018-02-18 13:46:10.140
2018-02-18 13:51:10.140
2018-02-18 13:56:10.140
host is:192.168.42.1
port is:1111
ServiceID is:sayHello
2018-02-18 14:01:10.156
```

图 3.5 服务提供者代码的部分输出截图

3.3　实现高可用的 Eureka 集群

在上文里，我们演示了 Eureka 客户端调用服务的整个流程，这里我们将在架构上有所改进。大家可以想象一下，在上文的案例中，Eureka 注册中心只部署在一台机器上，这样一旦它出现问题，就会导致整个服务调用系统的崩溃，如果这种情况发生在生产环境上，后果是不堪设想的。

大家别以为这是危言耸听，在高并发的场景下（比如双十一的并发环境），这种情况发生的可能性不低。针对这种场景，这里我们将部署两台 Eureka 注册中心，彼此相互注册，以此搭建一个可用性比较高的 Eureka 集群。

代码位置	视频位置
代码\第 3 章\ ek-cluster-server 代码\第 3 章\ ek-cluster-server-backup 代码\第 3 章\ ek-cluster-ServiceProvider 代码\第 3 章\ ek-cluster-ServiceCaller	视频\第 3 章\搭建高可用的 Eureka 集群

3.3.1　集群的示意图

在这个集群里，我们将配置 2 台相互注册的 Eureka 服务器，这样一来，每台服务器都包含着对方的服务注册信息，相当于双机热备，同时服务提供者只需向其中的一个注册服务，如图 3.6 所示。

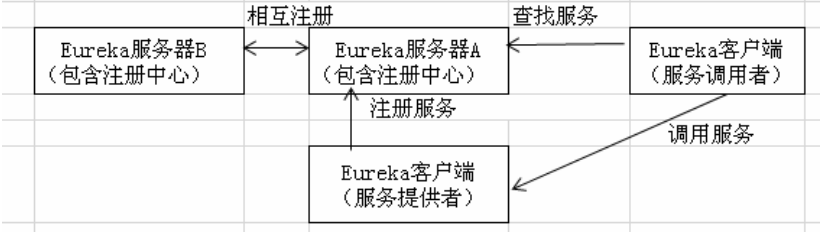

图 3.6　高可用 Eureka 集群示意图

这样，如果服务器 A 或 B 宕机，那么另一台服务器依然可以向外部提供服务列表，服务调用者依然可以据此调用服务。

在并发要求更高的环境里，我们甚至可以搭建 2 台以上的服务器，不过事实上，双机热备的集群能满足大多数的场景：一方面，不是每个系统的并发量都很高，所以双机热备足以满足大多数的并发需求；另一方面，毕竟两台服务器同时宕机的可能性也不大。

3.3.2　编写相互注册的服务器端代码

这里为了演示方便，我们在一台机器上模拟双服务器的场景，在真实项目里，我们一般是把两个相互注册的服务器安装在两台主机上，因为如果只安装在一台上，那么该服务器发生故障，两个服务器都会失效。具体的实现步骤如下。

步骤01 到 C:\WINDOWS\system32\drivers\etc 目录里，找到 hosts 文件，在其中加入两个机器名（其实都是指向本机），代码如下。修改后，需要重启机器。

```
127.0.0.1       ekServer1
127.0.0.1       ekServer2
```

步骤02 创建 ek-cluster-server 项目，其实是根据 3.2.1 小节里的 EurekaBasicDemo-Server 项目改写而来。和之前的项目相比，我们只改动了 application.yml 文件，代码如下：

```
server:
  port: 8888
spring:
  application:
    name: ekServer1
eureka:
  instance:
    hostname: ekServer1
  client:
    serviceUrl:
      defaultZone: http://ekServer2:8889/eureka/
```

这里的端口号没变，依然是 8888，但我们在第 5 行把项目名修改成了 ekServer1；在第 8 行里，把提供服务的主机名也修改成 ekServer1；在第 11 行里，我们指定了本服务所在的 url，这里请注意，我们把 ekServer1 所在的 serverUrl 指定到 ekServer2 的 8889 端口上，也就是说，这里我们指定 ekServer1 向 ekServer2 注册。

步骤03 在真实项目里，我们一般会在两台主机上启动两个 Eureka 服务，所以这里我们再创建一个 Maven 类型的项目 ek-cluster-server-backup，和之前的 ek-cluster-server 相比，它们的差别还是在于 application.yml，代码如下：

```
server:
  port: 8889
spring:
  application:
    name: ekServer2
eureka:
  instance:
    hostname: ekServer2
  client:
    serviceUrl:
      defaultZone: http://ekServer1:8888/eureka/
```

这里的配置信息其实和刚才的是对偶的，这里的 application 名和主机名都叫 ekServer2，不过请注意第 11 行，这里的 serviceUrl 是注册到 ekServer1 的 8888 端口上，这里我们同样指定 ekServer2 向 ekServer1 注册。结合上文，至此我们实现了双服务器之间的相互注册。

3.3.3 服务提供者只需向其中一台服务器注册

虽然在集群里搭建了两台服务器，但是服务提供者只需向其中的一台注册即可，否则高可用

的便利性就会以牺牲代码可维护性为代价了。

这里我们是在 ek-cluster-ServiceProvider 项目编写服务提供程序，它是根据上文 3.2.2 小节里的项目 EurekaBasicDemo-ServerProvider 改写而来的，其中只修改了 application.yml 部分的代码。

```
server:
  port: 1111
spring:
  application:
    name: sayHello
eureka:
  client:
    serviceUrl:
      defaultZone: http://ekServer1:8888/eureka/
```

我们只改动了第 9 行的代码，这说明本服务是向 ekServer1 的 8888 号端口注册。

由于这里两个 Eureka 服务器是相互注册的，因此本服务提供者无须同时向两个服务器注册，一旦向 ekServer1 注册后，该服务器就会自动把服务提供者的信息复制到 ekServer2 上。

3.3.4　修改服务调用者的代码

我们把服务调用者的代码放入 ek-cluster-ServiceCaller 这个 Maven 项目里，这是根据之前 3.2.3 小节里的 EurekaBasicDemo-ServerCaller 项目改写而来的。其中，我们也只修改 application.yml 代码。

```
spring:
  application:
    name: callHello
server:
  port: 8080
eureka:
  client:
    serviceUrl:
      defaultZone: http://ekServer1:8888/eureka/
```

改动点还是在第 9 行上，这里是向 ekServer1 服务器的 8888 号端口注册。同理，这里也无须向另外一个机器（ekServer2）注册。

3.3.5　正常场景下的运行效果

按如下次序启动 4 个项目的 Spring Boot 服务。

第一，ek-cluster-server（第一个 Eureka 服务器）。
第二，ek-cluster-server-backup（第二个 Eureka 服务器）。
第三，ek-cluster-ServiceProvider（服务提供者）。
第四，ek-cluster-ServiceCaller（服务调用者）。

随后，大家能在 http://ekserver1:8888/和 http://ekserver2:8889/两个浏览器上看到如图 3.7 所示

的 4 个可用服务。由于是相互注册，因此它们的内容是一样的。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CALLHELLO	n/a (1)	(1)	UP (1) - 192.168.42.1:callHello:8080
EKSERVER1	n/a (1)	(1)	UP (1) - 192.168.42.1:ekServer1:8888
EKSERVER2	n/a (1)	(1)	UP (1) - 192.168.42.1:ekServer2:8889
SAYHELLO	n/a (1)	(1)	UP (1) - 192.168.42.1:sayHello:1111

图 3.7　集群运行后的效果图

虽然这里我们也可以通过 http://localhost:8888/和 http://localhost:8889/看到相同的效果，但是不推荐。这是因为，在真实的项目里，Eureka 的服务器应该是和开发机器分开的，也就是说它们应该被部署在其他机器上，只不过这里我们为了演示方便，把它们都放在了本机中。

当我们确认服务启动后，可以在浏览器里输入“http://ekserver1:8080/hello”来查看服务调用的效果，这里其实触发了 ek-cluster-ServiceCaller 中 Controller 里的 hello 方法。

和之前一样，这里的输出还是“In Caller, hello Eureka”，这说明双机热备的 Eureka 架构至少不会影响基本的功能。同样，这里不建议通过 http://localhost:8080/hello 来查看运行效果。

3.3.6　一台服务器宕机后的运行效果

这里我们可以故意关闭 ek-cluster-server 服务，以此来模拟一台服务器宕机的情况。

关闭后，我们在浏览器里输入“http://ekserver1:8080/hello”，虽然我们在服务提供者和服务调用者的 application.yml 里指定的 serviceUrl.defaultZone 都是 http://ekServer1:8888/eureka/，但是在一台 Eureka 服务器失效的情况下，我们依然能看到正确的结果，如图 3.8 所示。

图 3.8　一台 Eureka 服务器宕机后的效果图

如果在刚才关闭的是 ek-cluster-server-backup，让 ek-cluster-server 运行，这里我们还是能看到同样的效果。也就是说，在这个 Eureka 双服务器的集群里，一台服务器宕机后，整个服务体系依然可用，这就大大提升了系统的可用性。

3.4　Eureka 的常用配置信息

这里我们将讲述查看 Eureka 客户端和服务器端配置信息的方法，并在此基础上讲述一些项目里常用的配置参数的用法。

3.4.1　查看客户端和服务器端的配置信息

在作者的机器上，本地仓库在 C:\Documents and Settings\Administrator\.m2\中，所以之后的叙述就以此为准，大家可以对应地找到自己 Maven 的本地仓库。

在 ~/.m2\repository\org\springframework\cloud\spring-cloud-netflix-eureka-client\1.3.1.RELEASE 这个目录里，可以看到 spring-cloud-netflix-eureka-client-1.3.1.RELEASE.jar 文件，在大家的机器上，也能找到版本号相同或不同的这个 jar 包。

解开这个 jar 文件，能在 META-INF 目录里找到 spring-configuration-metadata.json，在其中就用 json 格式的文件记录了所有的 Eureka 客户端的配置信息，我们来看一下部分代码。

```
1   {
2         "sourceType": "org.springframework.cloud.netflix.eureka.
           EurekaClientConfigBean",
3         "defaultValue": false,
4         "name": "eureka.client.allow-redirects",
5         "description": 省略关于该属性的描述,
6         "type": "java.lang.Boolean"
7       }
```

在第 4 行中，我们能看到该属性的名字，即 eureka.client.allow-redirects；第 2 行代码定义了该属性所在的类名；在第 3 行代码定义了该属性的默认值；第 6 行定义了该属性的类型。

同样的，在作者机器上也存在着 spring-cloud-netflix-eureka-server-1.3.1.RELEASE.jar 这个文件，解开它之后，在 META-INF 目录里能看到 spring-configuration-metadata.json，在其中包含着服务器端的所有配置信息。

3.4.2　设置心跳检测的时间周期

Eureka 客户端会定时向服务器端发送心跳，以此证明该站点可用，这个值默认是 30 秒，在实际应用里，我们可以通过修改 eureka.instance.lease-renewal-interval-in-seconds 属性来改变这个值。具体的做法是，在客户端的 application.yml 里，添加如下部分的代码。

```
1   eureka:
2     instance:
3       lease-renewal-interval-in-seconds: 60
```

关于心跳，还有另外一个 lease-expiration-duration-in-seconds 属性，默认是 90 秒，这说明如果服务器端有 90 秒没收到客户端的心跳，就会把它从服务列表里删除。

3.4.3　设置自我保护模式

在 Eureka 服务器端，我们能看到 eureka.server.enable-self-preservation 参数，用它可以指定是否启动保护模式，默认值是 true。

从上文中我们已经知道，如果 Eureka 服务端在一定的时间段内没有接收到某个客户端服务提供者实例的心跳，那么 Eureka 服务端将注销该实例，这个时间段的默认值是 90 秒。

这样做能避免因服务不可用而导致的“错误扩展”，从而能把错误的影响控制在一个较小的范围内。但现实中可能会发生这种情况：服务器端和客户端之间联系不上不是因为客户端服务不可用，而是因为当前网络确实有故障（服务提供者本身没问题），这时就不应当注销服务了。

Eureka 服务器能通过“自我保护模式”来处理这类问题，根据官方文档，如果在 15 分钟内，超过 85%的客户端实例都没有发来正常的心跳，那么 Eureka 服务器就认为出现了网络故障，这时就会进入自我保护模式。

进入该模式后，Eureka Server 就会保护服务注册表中的信息，不再继续删除注册中心里的服务列表数据。当网络故障恢复后，就会自动退出自我保护模式。

从上述描述来看，自我保护模式能提升 Eureka 集群的健壮性，所以如果没有特殊的情况，不建议通过把 eureka.server.enable-self-preservation 设置成 false 来关闭自我保护模式。

3.4.4 其他常用配置信息

在表 3.1 里，我们归纳了在客户端常用的一些配置信息，它们一般是配置在 Eureka 客户端。

表 3.1 Eureka 客户端常用配置信息归纳表

参数值	描述
eureka.client.instance-info-replication-interval-seconds	实例变更后复制到 Eureka 服务器所需的时间间隔，默认为 30 秒
eureka.client.eureka-server-total-connections	Eureka 客户端所允许的所有 Eureka 服务器连接的总数，默认是 200。注意，这里指的是所有服务器
eureka.client.eureka-server-total-connections-per-host	Eureka 客户端所允许的 Eureka 单台服务器连接的总数，默认是 50。注意，这里指的是单台
eureka.client.register-with-eureka	该实例是否需要在 eureka 服务器上注册，默认是 true
eureka.client.eureka-connection-idle-timeout-seconds	申请服务的 HTTP 请求的最长等待时间，默认为 30 秒，如果 30 秒内该 HTTP 请求没有任何动作，就会自动断开连接
eureka.client.fetch-registry	是否获取 Eureka 服务器注册中心上的所有服务的注册信息，默认为 true

在表 3.2 里，我们归纳了在服务器端常用的一些配置信息，这些参数的影响面都比较大，所以没有特殊理由，不建议修改。同样的，服务端的参数一般都会用默认值，没事不会轻易修改。

表 3.2 Eureka 服务器端常用配置信息归纳表

参数值	描述
eureka.server.renewal-percent-threshold	开启自我保护模式的阈值因子，默认是 0.85，比如在 15 分钟内，有 85%的客户端服务不可用，则开启自我保护模式
eureka.server.renewal-threshold-update-interval-ms	关于自我保护模式中阈值更新的时间间隔，单位为毫秒
eureka.server.response-cache-auto-expiration-in-seconds	当 Eureka 服务器的注册中心里的服务信息发生时，其被保存在缓存中不失效的时间，默认为 180 秒

3.5　本 章 小 结

在这个章节里，我们不仅学习了 Eureka 各部分组件的基本用法，更了解了在项目里搭建 Eureka 架构的基本步骤。从本章给出的架构案例中，我们能看到，高可用的架构确实能降低系统因宕机而造成的风险。换句话说，大家从这个章节里已经开始接触架构师所需要的技能点。

这只是一个开始，在本书的后继章节里，还将进一步讲解其他诸如“负载均衡”之类的和架构相关的知识点。大家在后继的学习中，除了得了解其他相关组件的用法之外，还得留意“集群”和“架构”方面的知识点。当然，遇到这样比较“值钱”的知识点，作者会给出提示，以引起大家的重视。

第 4 章

负载均衡组件：Ribbon

在一些高并发的分布式系统里，往往会用多台服务器搭建成集群，以此来均衡系统访问量，对此大家可以参考第 3 章 Eureka 的例子。但即使搭建集群，如果不做任何配置，系统依然无法把流量有效地分摊到各服务器上。

负载均衡可以在硬件层和软件层实现，对于一些有分布式需求但并发量不是特别高的系统而言，用软件层的即可。在 Spring Cloud 的诸多组件里，Ribbon 能提供负载均衡的功能，事实上，它足以满足大多数系统的负载均衡需求。

4.1 网络协议和负载均衡

虽然说 Spring Cloud 里的 Ribbon 组件向大家屏蔽了在网络协议层面的实现细节，但如果大家了解了这些细节，就将会更好地了解负载均衡的实现原理。而且，对于资深架构师而言，可能不仅仅限于现有的负载均衡组件（比如 Ribbon），更得结合诸多组件的优势创建一套适合本项目的实现方案，这就更得对底层的网络协议有深刻的了解了。

4.1.1 基于 4 层和 7 层的负载均衡策略

所有的负载均衡硬件或软件都是作用在网络通信协议之上的，当前我们的网络是运行在如图 4.1 所示的 OSI 七层网络协议之上的。

从上往下分别是应用层、表示层、会话层、传输层、网络层、数据链路层和物理层，其中，我们比较熟悉的 TCP/IP 或 UDP 协议工作在第 4 层，即传输层；而 HTTP、FTP、Telnet 和 SMTP 等常用的协议则是在第 7 层，即物理层。

图 4.1　OSI 七层网络协议结构图

一般来讲，负载均衡主要有 4 层交换（L4 switch）和 7 层交换（L7 switch）之分，这些术语是针对上述网络协议而言的，具体而言，是指在进行负载均衡时是在第 4 层还是在第 7 层转发请求。

针对 4 层的负载均衡策略是基于 TCP/IP 协议来实现的，比如可以根据 IP 地址和端口号决定转发的规则；针对 7 层的策略可以根据请求中基于 HTTP 协议的信息来转发，比如可以根据 HTTP 信息头里包含的操作系统类型来进行转发。

从应用角度来看，基于 4 层和 7 层的负载均衡技术最大的差别在于功能与效率。基于 4 层的方案由于无须解析基于应用层的消息内容，因此简单高效；基于 7 层的方案则能根据具体的业务场景来进行分发，所以功能比较强大，但代价比较昂贵。所以在选用时，其实没有优劣之分，还得根据具体的需求场景来综合考虑。

4.1.2　硬件层和软件层的负载均衡方案比较

硬件方面的解决方案一般是指在服务器和网络之间配置负载均衡器，让专门的硬件设备完成分发流量的任务。在这种解决方案里，我们可以给负载均衡器配置针对项目的专有负载均衡策略，从而达到很好的效果。相比软件层的解决方案，负载均衡器效果较好，但价格比较高。

基于软件的负载均衡是指在一台或多台服务器上安装专门的软件来实现均衡流量的效果。一般来说，它的成本比较低廉，所以能根据实际需求增加或更改负载均衡机器的数量，但同能的情况下，效果没有硬件来的好。

在大多数的应用场景里，对负载均衡要求不会特别高，所以说，基于软件的方案足以满足大多数的需求。

常见的硬件负载均衡器有思科和 BIG-IP 系列产品。常见的负载均衡软件有 LVS 和 Nginx，其中 LVS 工作在 4 层，而 Nginx 则可以工作在 7 层。

4.1.3　常见的软件负载均衡策略

一般来说，负载均衡软件会用如下 4 种策略来把请求分派到集群中的服务器上。

第一，轮询策略。这种策略的原理非常简单，把请求依次派发到服务器节点上，这适用于各

个服务节处理请求的能力都相同的场景。

第二，随机策略。这与轮询相似，只是不需要对每个请求进行编号，每次随机取一个。同样地，该策略也将每个服务器节点视为等同的。

第三，最小响应时间策略。在这种策略里，将计算出每个服务节点的平均响应时间，以此来选择响应时间最小的服务器。该策略能较好地根据服务器的情况做动态调整，但时间上会有些延后，可能无法更好地适应高并发流量的场景。

第四，最小并发数策略。在这种策略里，记录了当前时刻每个节点正在处理的请求数量，然后选择并发数最小的节点。该策略实现起来较为复杂，但能合理地分配负载。

4.1.4 Ribbon 组件基本介绍

Ribbon 是 Spring Cloud Netflix 全家桶中负责负载均衡的组件，是一组类库的集合。通过 Ribbon，程序员能在不涉及具体实现细节的基础上“透明”地用到负载均衡，而不必在项目里过多地编写实现负载均衡的代码。

比如，在某个包含 Eureka 和 Ribbon 的集群中，某个服务（可以理解成一个 jar 包）被部署在多台服务器上，当多个服务使用者同时调用该服务时，这些并发的请求能被用一种合理的策略转发到各台服务器上。

事实上，在使用 Spring Cloud 的其他各种组件时，我们都能看到 Ribbon 的痕迹，比如 Eureka 能和 Ribbon 整合，而在后文里将提到的提供网关功能 Zuul 组件在转发请求时，也可以整合 Ribbon，从而达到负载均衡的效果。

从代码层面来看，Ribbon 有如下 3 个比较重要的接口。

第一，IloadBalancer。这也叫负载均衡器，通过它，我们能在项目里根据特定的规则合理地转发请求。ILoadBalancer 是一个接口，常见的实现类有 BaseLoadBalancer。

第二，IRule。这个接口有多个实现类，比如 RandomRule 和 RoundRobinRule，这些实现类具体地定义了诸如“随机”和“轮询”等的负载均衡策略。此外，我们还能重写该接口里的方法来自定义负载均衡的策略。

第三，IPing 接口。通过该接口，我们能获取到当前哪些服务器是可用的，也能通过重写该接口里的方法来自定义判断服务器是否可用的规则。

当然，还有 ServerList、ServerListFilter、ServerListUpdate 和 DynamicServerListLoadbalancer 等其他接口和类，不能说它们不重要，但它们比较偏重于底层实现，在一般项目里出现的概率不高，所以在本章中不会详细讲述它们。

4.2 编写基本的负载均衡程序

虽然在实际项目里，Ribbon 经常是和其他组件配套使用的，但在这里，为了让大家感性地体会到负载均衡的实际效果和开发方式，所以在这里将基于 Ribbon 独立地实现负载均衡功能。

4.2.1　编写服务器端的代码

在 3.3 节里，我们编写了高可用的 Eureka 集群，启动后能以如下两个不同的 url 形式向外界提供服务。

```
1   http://ekserver1:8080/hello
2   http://ekserver2:8080/hello
```

大家可以把它们想象成是两个不同的服务器，当外部的请求较频繁时，我们可以把请求分发到这两台服务器上，以提升系统处理高并发请求的能力，如图 4.2 所示。

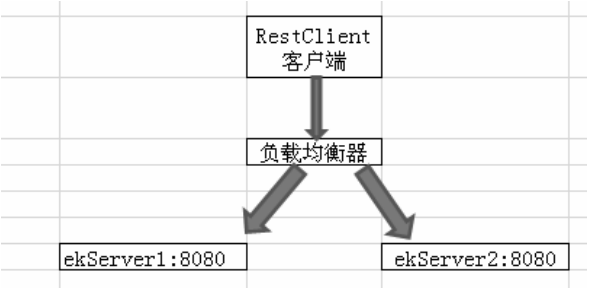

图 4.2　第一个负载均衡代码的示意图

4.2.2　编写客户端调用的代码

代码位置	视频位置
代码\第 4 章\RabbionBasicDemo	视频\第 4 章\负载均衡基础案例分析

步骤01 创建名为 RabbionBasicDemo 的 Maven 项目，在其中的 pom.xml 里，编写 Ribbon 的依赖包，关键代码如下：

```
1   <groupId>com.springboot</groupId>
2     <artifactId>RabbionBasicDemo</artifactId>
3     <version>0.0.1-SNAPSHOT</version>
4     <packaging>jar</packaging>
5     <dependencies>
6       <dependency>
7             <groupId>com.netflix.ribbon</groupId>
8             <artifactId>ribbon-core</artifactId>
9             <version>2.2.0</version>
10      </dependency>
11      <dependency>
12            <groupId>com.netflix.ribbon</groupId>
13            <artifactId>ribbon-httpclient</artifactId>
14            <version>2.2.0</version>
15      </dependency>
16      省略非关键的代码
17    </dependencies>
```

在第 2~4 行里，我们定义了该项目的名字、所用版本号以及打包方式等关键信息。在第 6~10 行里，我们引入了 Ribbon-Core 模块，在其中包含了负载均衡器等关键接口和 API 的定义。在第

11~15 行中，我们引入了 ribbon-httpclient 模块，在该模块里提供了包含负载均衡功能的 HTTP 客户端的调用方法。

步骤02 编写基于负载均衡策略的客户端调用类 RibbonDemo.java，代码如下。

```
1   //省略必要的 package 和 import 方法
2   public class RibbonDemo{
3      public static void main( String[] args ) throws Exception
4      {
5           //定义基于 Rest 的客户端
6          RestClient client = (RestClient)ClientFactory.getNamedClient
            ("RibbonDemo");
7           HttpRequest request = HttpRequest.newBuilder().uri(new
             URI("/hello")).build();
8          //设置负载均衡的属性
9         ConfigurationManager.getConfigInstance().setProperty("RibbonDemo.
           ribbon.listOfServers", "ekserver1:8080,ekserver2:8080");
10          //为了避免频繁访问而导致的服务失效，睡眠 3 秒
11          Thread.sleep(5000);
12          //向 2 个服务器发出调用 10 次的请求
13          for(int i = 0; i < 10; i ++) {
14             HttpResponse response =
                client.executeWithLoadBalancer(request);
15             //输出每次访问的状态，其中包含访问的服务器
16             System.out.println("Status for URI:" + response.getRequestedURI()
                + " is :" + response.getStatus());
17             //输出返回结果
18             System.out.println("Result is:" +
                response.getEntity(String.class));
19          }
20     }
21  }
```

在第 6 行里，我们通过工厂模式生成了一个 RestClient 类型的 client 类，通过这个类提供的 executeWithLoadBalancer 方法，我们可以把请求平摊到两台服务器上。在第 7 行里，我们创建了一个 HttpRequest 类型的 request 请求，通过 request 对象定义 url 请求的后缀是“/hello”。

在第 9 行里，我们通过 RibbonDemo.ribbon.listOfServers 这个属性设置了两台可供负载均衡选择的服务器，它们分别指向了注册到 Eureka 服务器能提供服务的两个地址。在实际的项目里，可以把这两台服务器的地址写入配置文件里。

为了更好地演示负载均衡的效果，我们在第 11 行里让线程睡眠 5 秒。在第 13~19 行的 for 循环里，我们发起了 10 次请求调用。具体而言，在第 14 行，通过 client 的 executeWithLoadBalancer 方法，以负载均衡的方式向 ekserver1:8080/hello 和 ekserver2:8080/hello 发起请求，并用 response 对象来接收返回结果。之后，在第 16 行，我们输出了返回状态，均是 200，表示正常访问，在第 18 行，我们输出了调用服务的结果。

如果大家在自己机器上运行这段代码，就会发现 for 循环里的请求被分摊到两个服务器上，而不是让某台服务器过多地承担。如果把循环次数修改成 100 次，那么能更清晰地看到这个效果。

4.3　Ribbon 中重要组件的用法

在之前的案例中，我们用 RestClient 对象的 executeWithLoadBalancer 方法实现了基本的负载均衡功能。事实上，通过 Ribbon 提供的组件，我们能更好地实现负载均衡的效果。这里我们将依次讲述它的各种重要组件。

4.3.1　ILoadBalancer：负载均衡器接口

在前文里，我们通过 RestClient 类型对象的 executeWithLoadBalancer 方法来实现基于负载均衡的请求的调用，在 Ribbon 里，我们还可以通过 ILoadBalancer 这个接口以基于特定的负载均衡策略来选择服务器。

通过下面的 ILoadBalancerDemo.java，我们来看一下这个接口的基本用法。这个类放在 4.2 节创建的 RabbionBasicDemo 项目里，代码如下：

```
1   //省略必要的 package 和 import 代码
2   public class ILoadBalancerDemo {
3      public static void main(String[] args){
4          //创建 ILoadBalancer 的对象
5           ILoadBalancer loadBalancer = new BaseLoadBalancer();
6          //定义一个服务器列表
7           List<Server> myServers = new ArrayList<Server>();
8          //创建两个 Server 对象
9           Server s1 = new Server("ekserver1",8080);
10          Server s2 = new Server("ekserver2",8080);
11         //两个 server 对象放入 List 类型的 myServers 对象里
12          myServers.add(s1);
13          myServers.add(s2);
14          //把 myServers 放入负载均衡器
15          loadBalancer.addServers(myServers);
16          //在 for 循环里发起 10 次调用
17          for(int i=0;i<10;i++){
18            //用基于默认的负载均衡规则获得 Server 类型的对象
19            Server s = loadBalancer.chooseServer("default");
20            //输出 IP 地址和端口号
21            System.out.println(s.getHost() + ":" + s.getPort());
22          }
23     }
24  }
```

在第 5 行里，我们创建了 BaseLoadBalancer 类型的 loadBalancer 对象，而 BaseLoadBalancer 是负载均衡器 ILoadBalancer 接口的实现类。

在第 6~13 行里，我们创建了两个 Server 类型的对象，并把它们放入 myServers 里。在第 15 行里，我们把 List 类型的 myServers 对象放入了负载均衡器里。

在第 17~22 行的 for 循环里，我们通过负载均衡器模拟了 10 次选择服务器的动作。具体而言，

在第 19 行里，通过 loadBalancer 的 chooseServer 方法以默认的负载均衡规则选择服务器；在第 21 行里，用“打印”这个动作来模拟实际的“使用 Server 对象处理请求”的动作。

上述代码的运行结果如下所示，loadBalancer 这个负载均衡器把 10 次请求均摊到了两台服务器上，从中确实能看到 “负载均衡”的效果。

```
ekserver2:8080
ekserver1:8080
ekserver2:8080
ekserver1:8080
ekserver2:8080
ekserver1:8080
ekserver2:8080
ekserver1:8080
ekserver2:8080
ekserver1:8080
```

4.3.2 IRule：定义负载均衡规则的接口

在上文里我们提到了负载均衡的一些规则，在 Ribbon 里，我们可以通过定义 IRule 接口的实现类来给负载均衡器设置相应的规则。在表 4.1 里，我们能看到 IRule 接口的一些常用的实现类。

表 4.1 IRule 的实现类归纳表

实现类的名字	负载均衡的规则
RandomRule	采用随机选择的策略
RoundRobinRule	采用轮询策略
RetryRule	采用该策略时，会包含重试动作
AvailabilityFilterRule	会过滤一些多次连接失败和请求并发数过高的服务器
WeightedResponseTimeRule	根据平均响应时间为每个服务器设置一个权重，根据该权重值优先选择平均响应时间较小的服务器
ZoneAvoidanceRule	优先把请求分配到和该请求具有相同区域（Zone）的服务器上

在下面的 IRuleDemo.java 的程序里，我们来看一下 IRule 的基本用法。同样，这个类是放在项目里的。

```
//省略必要的 package 和 import 代码
public class IRuleDemo {
    public static void main(String[] args){
        //请注意这时用到的是 BaseLoadBalancer，而不是 ILoadBalancer 接口
         BaseLoadBalancer loadBalancer = new BaseLoadBalancer();
         //声明基于轮询的负载均衡策略
         IRule rule = new RoundRobinRule();
        //在负载均衡器里设置策略
        loadBalancer.setRule(rule);
         //如下定义 3 个 Server，并把它们放入 List 类型的集合中
         List<Server> myServers = new ArrayList<Server>();
         Server s1 = new Server("ekserver1",8080);
         Server s2 = new Server("ekserver2",8080);
         Server s3 = new Server("ekserver3",8080);
```

```
15          myServers.add(s1);
16          myServers.add(s2);
17          myServers.add(s3);
18          //在负载均衡器里设置服务器的 List
19          loadBalancer.addServers(myServers);
20          //输出负载均衡的结果
21          for(int i=0;i<10;i++){
22            Server s = loadBalancer.chooseServer(null);
23            System.out.println(s.getHost() + ":" + s.getPort());
24          }
25      }
26  }
```

这段代码和上文里的 ILoadBalancerDemo.java 很相似，但有如下的差别点。

（1）在第 5 行里，我们是通过 BaseLoadBalancer 这个类而不是接口来定义负载均衡器，原因是该类包含 setRule 方法。

（2）在第 7 行中，定义了一个基于轮询规则的 rule 对象，并在第 9 行里把它设置进负载均衡器。

（3）在第 19 行里，我们是把包含 3 个 Server 的 List 对象放入负载均衡器，而不是之前的两个。由于这里存粹是为了演示效果，因此我们放入了一个根本不存在的“ekserver3”服务器。

运行该程序后，我们可以看到有 10 次输出，而且确实是按“轮询”的规则有顺序地输出 3 个服务器的名字。如果我们把第 7 行改成如下代码，就会看到“随机”地输出服务器名。

```
1       IRule rule = new RandomRule();
```

4.3.3　IPing：判断服务器是否可用的接口

在项目里，我们一般会让 ILoadBalancer 接口自动地判断服务器是否可用（这些业务都封装在 Ribbon 的底层代码里）。此外，我们还可以用 Ribbon 组件里的 IPing 接口来实现这个功能。

在下面的 IRuleDemo.java 代码里，我们将演示 IPing 接口的一般用法。同样，这段代码也是在 RibbonBasisDemo 这个项目里。

```
1   //省略必要的 package 和 import 代码
2   class MyPing implements IPing {
3       public boolean isAlive(Server server) {
4           //如果服务器名是 ekserver2，则返回 false
5           if (server.getHost().equals("ekserver2")) {
6               return false;
7           }
8           return true;
9       }
10  }
```

第 2 行定义的 MyPing 类实现了 IPing 接口，并在第 3 行重写了其中的 isAlive 方法。

在这个方法里，我们根据服务器名来判断。具体而言，如果名字是 ekserver2，就返回 false，表示该服务器不可用；否则，返回 true，表示当前服务器可用。

```
11  public class IRuleDemo {
```

```
12      public static void main(String[] args) {
13          BaseLoadBalancer loadBalancer = new BaseLoadBalancer();
14          //定义 IPing 类型的 myPing 对象
15          IPing myPing = new MyPing();
16          //在负载均衡器里使用 myPing 对象
17          loadBalancer.setPing(myPing);
18          //同样是创建三个 Server 对象并放入负载均衡器
19          List<Server> myServers = new ArrayList<Server>();
20          Server s1 = new Server("ekserver1", 8080);
21          Server s2 = new Server("ekserver2", 8080);
22          Server s3 = new Server("ekserver3", 8080);
23          myServers.add(s1);
24          myServers.add(s2);
25          myServers.add(s3);
26          loadBalancer.addServers(myServers);
27          //通过 for 循环多次请求服务器
28          for (int i = 0; i < 10; i++) {
29              Server s = loadBalancer.chooseServer(null);
30              System.out.println(s.getHost() + ":" + s.getPort());
31          }
32      }
33  }
```

在第 12 行的 main 函数里，我们在第 15 行创建了 IPing 类型的 myPing 对象，并在第 17 行把这个对象放入了负载均衡器。通过第 18~26 行的代码，我们创建了 3 个服务器，并把它们也放入负载均衡器。

在第 28 行的 for 循环里，我们依然是请求并输出服务器名。由于这里的负载均衡器 loadBalancer 中包含了一个 IPing 类型的对象，因此在根据策略得到服务器后，会根据 myPing 里的 isActive 方法来判断该服务器是否可用。

由于在这个方法里我们定义了 ekServer2 这台服务器不可用，因此负载均衡器 loadBalancer 对象始终不会把请求发送到该服务器上，也就是说，在输出结果中，我们不会看到“ekserver2:8080”的输出。

从中我们能看到 IPing 接口的一般用法，我们可以通过重写其中的 isAlive 方法来定义“判断服务器是否可用”的逻辑。在实际项目里，判断的依据无非是“服务器响应是否时间过长”或“发往该服务器的请求数是否过多”，而这些判断方法都封装在 IRule 接口以及它的实现类里，所以在一般的场景中我们会用到 IPing 接口。

4.4 Ribbon 整合 Eureka 组件

在上文里，我们分别讲述了 Ribbon 里实现负载均衡功能的相关重要组件，事实上，Ribbon 一般不会单独出现，往往是嵌在其他架构中。这里我们将演示 Ribbon 和 Eureka 配套使用的开发方式。

4.4.1　整体框架的说明

在第 3 章给出的 Eureka 的高可用案例中，我们就已经用到了 LoadBalanced 注解。回顾一下如图 4.3 所示的示意图，在这个案例中，我们配置了两台相互注册的 Eureka 服务器，但服务提供者只是配置在一台机器上，而不是用多台能提供服务的机器来分摊流量。

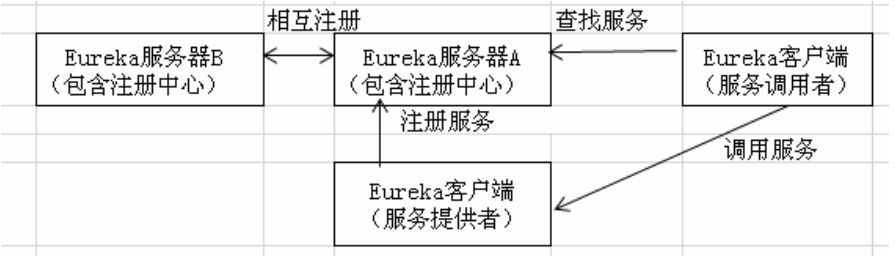

图 4.3　回顾第 3 章给出的高可用的 Eureka 框架的示意图

当时我们引入@LoadBalanced 注解的原因是，RestTemplate 类型的对象本身不具备调用远程服务的能力，也就是说，引入该注解的目的存粹是为了让代码跑通。

在本案例的框架里，我们将配置一个 Eureka 服务器，搭建 3 个提供相同服务的 Eureka 服务提供者，同时在 Eureka 服务调用者里引入 Ribbon 组件。这样，当有多个 url 向服务调用者发起调用请求时，整个框架能按配置在 IRule 和 IPing 中的“负载均衡策略”和“判断服务器是否可用的策略”把这些 url 请求合理地分摊到多台机器上。

从图 4.4 中，我们能看到本系统的结构图。其中，3 个服务提供者向 Eureka 服务器注册服务，而基于 Ribbon 的负载均衡器能有效地把请求分摊到不同的服务器上。

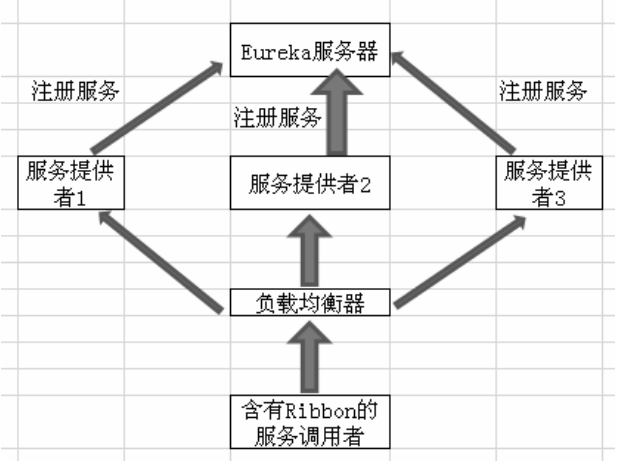

图 4.4　Eureka 和 Ribbon 整合后的结构图

为了让大家更方便地跑通这个案例，我们将讲解全部的服务器、服务提供者和服务调用者部分的代码。在表 4.2 中，列出了本架构中的所有项目。

表 4.2　Ribbon 整合 Eureka 案例中的项目列表

项目名	说明
代码\第 4 章\EurekaRibbonDemo-Server	Eureka 服务器
代码\第 4 章\EurekaRibbonDemo-ServiceProviderOne 代码\第 4 章\EurekaRibbonDemo-ServiceProviderTwo 代码\第 4 章\EurekaRibbonDemo-ServiceProviderThree	在这 3 个项目里，分别部署着一个相同的服务提供者
代码\第 4 章\EurekaRibbonDemo-ServiceCaller	服务调用者

代码位置	视频位置
见表 4.2	视频\第 4 章\Ribbon 和 Eureka 整合的案例

4.4.2　编写 Eureka 服务器

这部分的代码其实是沿用第 3 章 EurekaBasicDemo-Server 这个项目的，只是把项目名改成了 EurekaRibbonDemo-Server。在本书附带资料的相关位置中，大家能看到完整的代码。

步骤01 在 pom.xml 里编写本项目需要用到的依赖包，其中通过如下代码引入了 Eureka 服务器所必需的包。

```
1    <dependencies>
2      <dependency>
3          <groupId>org.springframework.cloud</groupId>
4          <artifactId>spring-cloud-starter-eureka-server</artifactId>
5      </dependency>
```

步骤02 在 application.yml 这个文件里，指定了针对 Eureka 服务器的配置，关键代码如下：

```
1  server:
2    port: 8888
3  eureka:
4    instance:
5      hostname: localhost
6    client:
7      serviceUrl:
8        defaultZone: http://localhost:8888/eureka/
```

在第 2 行和第 5 行里，指定了本服务器所在的主机地址和端口号是 localhost:8888。在第 8 行里，指定了默认的 url 是 http://localhost:8888/eureka/。

步骤03 在 RegisterCenterApp 这个服务启动程序里编写启动代码。

```
1  //省略必要的 package 和 import 代码
2  @EnableEurekaServer
3  @SpringBootApplication
4  public class RegisterCenterApp
5  {
6     public static void main( String[] args )
7     {
8        SpringApplication.run(RegisterCenterApp.class, args);
```

```
9       }
10  }
```

启动该程序后，能在 http://localhost:8888/看到该服务器的相关信息。

4.4.3　编写 Eureka 服务提供者

这里有 3 个服务提供者，它们均是根据第 3 章里的 EurekaBasicDemo-ServiceProvider 改写而来。我们就拿 EurekaRibbonDemo-ServiceProviderOne 来举例，看一下其中包含的关键要素。

第一，同样是在 pom.xml 里，引入了服务提供者程序所需的 jar 包，不过在其中需要适当地修改项目名。

第二，同样是在 ServiceProviderApp.java 里，编写了启动程序，代码不变。

第三，在 application.yml 里，编写了针对这个服务提供者的配置信息。关键代码如下：

```
1   server:
2     port: 1111
3   spring:
4     application:
5       name: sayHello
6   eureka:
7     client:
8       serviceUrl:
9         defaultZone: http://localhost:8888/eureka/
```

在第 2 行里，指定了本服务是运行在 1111 端口上，在另外的两个服务提供者程序里，我们分别指定了它们的工作端口是 2222 和 3333。在第 5 行里，我们指定了服务提供者的名字是 sayHello，另外两个服务器提供者的名字同样是 sayHello，正因为它们的名字都一样，所以服务调用者在请求服务时，负载均衡组件才能有效地分摊流量。

第四，在 Controller 这个控制器类里，编写了处理 url 请求的逻辑。关键代码如下：

```
1   //省略了必要的 package 和 import 的代码
2   @RestController
3   public class Controller {
4       @RequestMapping(value = "/sayHello/{username}", method =
         RequestMethod.GET)
5       public String hello(@PathVariable("username") String username) {
6           System.out.println("This is ServerProvider1");
7           return "Hello Ribbon, this is Server1, my name is:" + username;
8       }
9   }
```

在第 2 行里，我们通过@RestController 注解来说明本类承担着“控制器”的角色。在第 4 行里，我们定义了触发 hello 方法的 url 格式和 HTTP 请求的方式。在第 5~8 行的 hello 方法里我们返回了一个字符串。请大家注意，在第 6 行和第 7 行的代码里，我们能明显地看出输出和返回信息来自于 1 号服务提供者。

EurekaRibbonDemo-ServiceProviderTwo 和 EurekaRibbonDemo-ServiceProviderOne 项目很相似，改动点有如下 3 个。

第一，在 pom.xml 里，把项目名修改成 EurekaRibbonDemo-ServiceProviderTwo。

第二，在 application.yml 里，把端口号修改成 2222。关键代码如下：

```
server:
  port: 2222
```

第三，在 Controller.java 的 hello 方法里，在输出和返回信息里打上“Server2”的标记。关键代码如下：

```
@RequestMapping(value = "/sayHello/{username}", method =
 RequestMethod.GET  )
    public String hello(@PathVariable("username") String username) {
        System.out.println("This is ServerProvider2");
        return "Hello Ribbon, this is Server2, my name is:" + username;
    }
```

在 EurekaRibbonDemo-ServiceProviderThree 里，同样在 EurekaRibbonDemo-ServiceProviderOne 的基础上做上述 3 个改动。这里需要在 application.yml 里把端口号修改成 3333，在 Controller 类中需要在输出和返回信息中打上“Server3”的标记。大家可以到本书附带资料的相关位置查看本项目的全部代码。

4.4.4 在 Eureka 服务调用者里引入 Ribbon

EurekaRibbonDemo-ServiceCaller 项目是根据第 3 章的 EurekaBasicDemo-ServiceCaller 改写而来，其中的关键信息如下。

第一，在 pom.xml 里，只是适当地修改项目名字，没有修改其他代码。

第二，没有修改启动类 ServiceCallerApp.java 里的代码。

第三，在 application.yml 里，添加描述服务器列表的 listOfServers 属性，代码如下：

```
spring:
  application:
    name: callHello
server:
  port: 8080
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8888/eureka/
sayHello:
  ribbon:
    listOfServers:
http://localhost:1111/,http://localhost:2222/,http://localhost:3333
```

在第 3 行中，我们指定了服务调用者本身的服务名是 callHello，在第 5 行里，指定了这个微服务器运行在 8080 端口上。由于服务调用者本身也能对外界提供服务，因此外部程序能根据这个服务名和端口号以 url 的形式调用其中的 hello 方法。

这里的关键是第 12~13 行，我们通过 ribbon.listOfServers 指定了该服务调用者能获得服务的 3 个 url 地址。注意，这里的 3 个地址和上文里服务提供者发布服务的 3 个地址是一致的。

第四，在控制器类里，用 RestTemplate 对象，以负载均衡的方式调用服务，代码如下：

```
1   //省略必要的 package 和 import 的代码
2   @RestController
3   @Configuration
4   public class Controller {
5       @Bean
6       @LoadBalanced
7       public RestTemplate getRestTemplate()
8       { return new RestTemplate();  }
9       //提供服务的 hello 方法
10      @RequestMapping(value = "/hello", method = RequestMethod.GET   )
11      public String hello() {
12          RestTemplate template = getRestTemplate();
13          String retVal = template.getForEntity(
             "http://sayHello/sayHello/Eureka", String.class).getBody();
14          return "In Caller, " + retVal;
15      }
16  }
```

在这个控制器类的第 7 行里，我们通过 getRestTemplate 方法返回一个 RestTemplate 类型对象。RestTemplate 是 Spring 提供的能以 Rest 形式访问服务的对象，本身不具备负载均衡的能力，所以我们需要在第 6 行通过@LoadBalanced 注解赋予它这个能力。

在第 11~15 行的 hello 方法里，我们首先在第 12 行通过 getRestTemplate 方法得到了 template 对象，随后通过第 13 行的代码用 template 对象提供的 getForEntity 方法访问之前 Eureka 服务提供者提供的“http://sayHello/sayHello/Eureka”服务，并得到 String 类型的结果，最后在第 14 行根据调用结果返回一个字符串。由于在框架里我们模拟了在 3 台机器上部署服务的场景，而在上述服务调用者的代码里我们又在 template 对象上加入了@LoadBalanced 注解，因此在第 13 行代码里发起的请求会被均摊到 3 台服务器上。

需要注意的是，这里我们没有重写 IRule 和 IPing 接口，所以这里采用的是默认的 RoundRobbin（也就是轮询）的访问策略，同时将默认所有的服务器都处于可用状态。

依次启动本框架中的 Eureka 服务器、3 台服务提供者和服务器调用者的服务之后，在浏览器里输入“http://localhost:8888/”，我们能看到如图 4.5 所示的效果。其中，有 3 个提供服务的 SAYHELLO 应用实例，它们分别运行在 1111、2222 和 3333 端口上，同时服务调用者 CALLHELLO 运行在 8080 端口上。

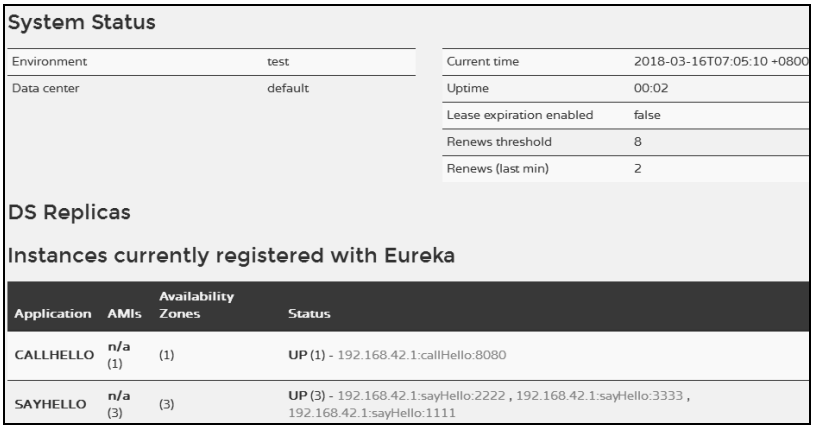

图 4.5　启动所有服务后在控制台中的效果图

如果我们不断在浏览器里输入“http://localhost:8080/hello”，就能依次看到如下所示的输出。

```
1   In Caller, Hello Ribbon, this is Server2, my name is:Eureka
2   In Caller, Hello Ribbon, this is Server1, my name is:Eureka
3   In Caller, Hello Ribbon, this is Server3, my name is:Eureka
4   In Caller, Hello Ribbon, this is Server2, my name is:Eureka
5   In Caller, Hello Ribbon, this is Server1, my name is:Eureka
6   In Caller, Hello Ribbon, this is Server3, my name is:Eureka
7   …
```

从上述输出来看，请求是以 Server2、Server1 和 Server3 的次序被均摊到 3 台服务器上。在每次启动服务后，可能承接请求的服务器次序会有所变化，可能下次是按 Server1、Server2 和 Server3 的次序，但每次都能看到“负载均衡”的效果。

4.4.5 重写 IRule 和 IPing 接口

这里，我们将在上述案例的基础上重写 IRule 和 IPing 接口里的方法，从而实现自定义负载均衡和判断服务器是否可用的规则。

注 意

由于我们是在客户端，也就是 EurekaRibbonDemo-ServiceCaller 这个项目调用服务，因此本部分的所有代码都是写在这个项目里的。

步骤01 编写包含负载均衡规则的 MyRule.java，代码如下：

```
1   package com.controller; //请注意这个 package 路径
2   //省略必要的 import 语句
3   public class MyRule implements IRule {//实现 IRule 类
4       private ILoadBalancer lb;
5       //必须要重写这个 choose 方法
6       public Server choose(Object key) {
7          //得到 0 到 3 的一个随机数，但不包括 3
8          int number = (int)(Math.random() * 3);
9          System.out.println("Choose the number is:" + number);
10         //得到所有的服务器对象
11          List<Server> servers = lb.getAllServers();
12          //根据随机数返回一个服务器
13          return servers.get(number);
14      }
15      //省略必要的 get 和 set 方法
16  }
```

在上述代码的第 3 行里，我们实现了 IRule 类，并在其中的第 6 行里重写了 choose 方法。在这个方法里，我们在第 8 行通过 Math.random 方法得到了 0 到 3 之间的一个随机数，包括 0，但不包括 3，并用这个随机数在第 13 行返回了一个 Server 对象，以此实现随机选择的效果。在实际的项目里，还可以根据具体的业务逻辑 choose 方法来实现其他“选择服务器”的策略。

步骤02 编写判断服务器是否可用的 MyPing.java，代码如下：

```
1   package com.controller; //也请注意这个 package 的路径
2   //省略 import 语句
3   public class MyPing implements IPing { //这里是实现 IPing 类
4      //重写了判断服务器是否可用的 isAlive 方法
5      public boolean isAlive(Server server) {
6         //这里是生成一个随机数，以此来判断该服务器是否可用
7         //还可以根据服务器的响应时间等依据判断服务器是否可用
8          double data = Math.random();
9          if (data > 0.6) {
10             System.out.println("Current Server is available, Name: " +
                server.getHost() + ", Port is:" + server.getHostPort());
11             return true;
12         } else {
13             System.out.println("Current Server is not available, Name: "
                + server.getHost() + ", Port is:" + server.getHostPort());
14             return false;
15         }
16     }
17  }
```

在第 3 行里，我们是实现了 IPing 接口，并在第 5 行重写了其中的 isAlive 方法。

在这个方法里，我们根据一个随机数来判断该服务器是否可用，如果可用，就返回 true，反之则返回 false。注意，这仅仅是一个演示的案例，在实际项目里，我们基本上是不会重写 isAlive 方法的。

步骤03 改写 application.yml，在其中添加关于 MyPing 和 MyRule 的配置，代码如下：

```
1   spring:
2     application:
3       name: callHello
4   server:
5     port: 8080
6   eureka:
7     client:
8       serviceUrl:
9         defaultZone: http://localhost:8888/eureka/
10  sayHello:
11    ribbon:
12      NFLoadBalancerRuleClassName: com.controller.MyRule
13      NFLoadBalancerPingClassName: com.controller.MyPing
14      listOfServers:
15  http://localhost:1111/,http://localhost:2222/,http://localhost:3333
```

改动点是第 10~13 行，注意这里的 SayHello 需要和服务提供者给出的“服务名”一致。在第 12 行、第 13 行里，分别定义了本程序（也就是服务调用者）所用到的 MyRule 和 MyPing 类，配置时需要包含包名和文件名。

步骤04 改写 Controller.java 和这个控制器类，代码如下。

```
1   //省略必要的 package 和 import 代码
2   @RestController
3   @Configuration
4   public class Controller {
```

```
5       //以 Autowired 的方式引入 loadBalancerClient 对象
6       @Autowired
7       private LoadBalancerClient loadBalancerClient;
8       //给 RestTemplate 对象加入@LoadBalanced 注解
9       //以此赋予该对象负载均衡的能力
10      @Bean
11      @LoadBalanced
12      public RestTemplate getRestTemplate()
13      { return new RestTemplate();   }
14      @Bean //引入 MyRule
15      public IRule ribbonRule()
16      { return new MyRule();}
17      @Bean //引入 MyPing
18      public IPing ribbonpIng()
19      {  return new MyPing();}
20      //编写提供服务的 hello 方法
21      @RequestMapping(value = "/hello", method = RequestMethod.GET   )
22      public String hello() {
23          //引入策略，这里的 sayHello 需要和 application.yml
24          //第 10 行的 sayHello 一致，这样才能引入 MyPing 和 MyRule
25          loadBalancerClient.choose("sayHello");
26          RestTemplate template = getRestTemplate();
27          String retVal = template.getForEntity(
             "http://sayHello/sayHello/Eureka", String.class).getBody();
28          return "In Caller, " + retVal;
29      }
30  }
```

和之前的代码相比，我们添加了第 15 行和第 18 行的两个方法，以此引入自定义的 MyRule 和 MyPing 两个方法。

而且，在 hello 方法的第 15 行里，我们通过 choose 方法为 loadBalancerClient 这个负载均衡对象选择了 MyRule 和 MyPing 这两个规则。

如果依次启动 Eureka 服务器，注册在 Eureka 里的 3 个服务提供者和服务调用者之后，在浏览器里输入“http://localhost:8080/hello”，就能在 EurekaRibbonDemo-ServiceCaller 的控制台里看到类似于如下的输出。

```
1   Choose the number is:1
2   Choose the number is:0
3   Current Server is not available, Name: 192.168.42.1,
     Port is:192.168.42.1:2222
4   Current Server is available, Name: 192.168.42.1, Port is:192.168.42.1:3333
5   Current Server is not available, Name: 192.168.42.1,
     Port is:192.168.42.1:1111
```

第 1 行和第 2 行是 MyRule 里的输出，第 3~5 行是 MyPing 里的输出，由于这些输出和随机数有关，因此每次输出的内容未必一致，但至少能说明我们在 MyRule 和 MyPing 里配置的相关策略是生效的，服务调用者（EurekaRibbonDemo-ServiceCaller）的多次请求在以“负载均衡”的方式分发到各服务提供者时会引入我们定义在上述两个类里的策略。

4.4.6　实现双服务器多服务提供者的高可用效果

代码位置	视频位置
代码\第 4 章\RabbionBasicDemo	视频\第 4 章\负载均衡高可用案例

这里我们把相同的服务提供模块部署在 3 台服务器上，除了能得到“负载均衡”的便利之外，还达到了“高可用”的效果，比如 3 台服务器中的某台失效了，系统就会把请求发送到其他机器上。

这种“高可用”的特性是互联网项目（尤其是高并发互联网项目）的必备需求，不过这里依然有一个隐患：如果 Eureka 服务器失效了，那么即使 3 台提供服务的机器都可用，服务调用者也还是无法得到服务。

在第 3 章里，我们配置了两个相互注册的 Eureka 服务器，这里我们将在当前“多服务提供者”的基础上引入“双 Eureka 服务器”的效果，以此实现更高程度的“高可用”效果。整个系统的架构如图 4.6 所示。

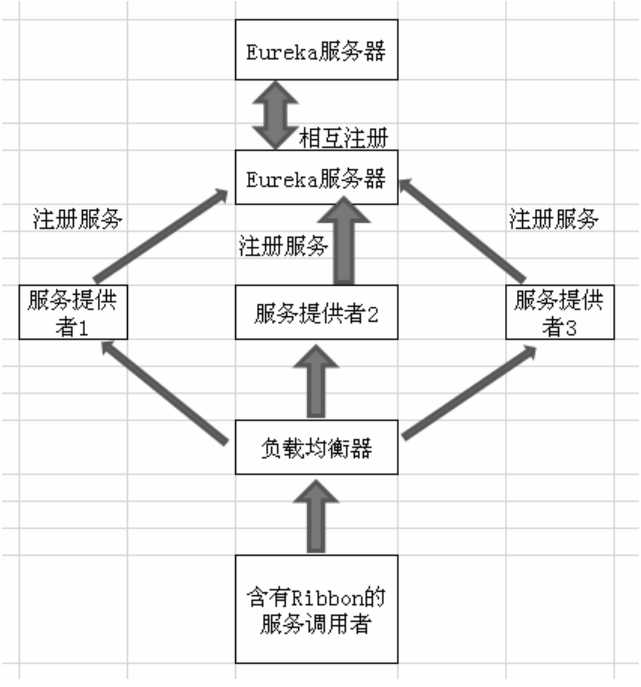

图 4.6　双服务器多服务提供者的高可用架构示意图

从图 4.6 中我们能看到，只有当两台 Eureka 服务器都宕机，或者所有提供服务的机器都宕机，整个系统才无法对外提供服务，但事实上发生这些情况的概率非常低。更何况在真实项目里我们会时刻监听每台服务器的状态，只要发生不可用，就会立即发送邮件等推送信息，这样维护人员就能立即介入修复。

为了实现上述效果，我们需要在 4.4.5 小节代码的基础上做如下修改。

步骤01 在 EurekaRibbonDemo-Server 项目里，修改 application.yml，代码如下：

```
1   spring:
2     application:
```

```
3       name: ekServer1
4   server:
5     port: 8888
6   eureka:
7     instance:
8       hostname: ekServer1
9     client:
10      serviceUrl:
11        defaultZone: http://ekServer2:8889/eureka/
```

这里同样需要在 hosts 文件里添加 ekServer1 和 ekServer2，具体做法请参照 3.3 节的说明。请注意在第 11 行里，本服务器是向另外一台 ekServer2:8889 注册。

步骤02 创建名为 EurekaRibbonDemo-backup-Server 的项目，代码和 EurekaRibbonDemo-Server 大多一致，但需要修改其中的 application.yml，代码如下：

```
1   spring:
2     application:
3       name: ekServer2
4   server:
5     port: 8889
6   eureka:
7     instance:
8       hostname: ekServer2
9     client:
10      serviceUrl:
11        defaultZone: http://ekServer1:8888/eureka/
```

这个 ekServer1 里的配置是对偶的，在第 11 行里，指定本服务是向 ekServer1 的 8888 端口注册。结合刚才 ekServer1 的配置，我们能看到这两台服务器（ekServer1 和 ekServe2）是相互注册的，以此实现“热备冗余”的效果。

步骤03 在 3 个服务提供者和一个服务调用者的项目里，修改它们的 application.yml，其中需要修改的部分如下：

```
1   eureka:
2     client:
3       serviceUrl:
4         #defaultZone: http://localhost:8888/eureka/
5         defaultZone: http://ekServer1:8888/eureka/
```

原来采用第 4 行的代码是向 localhost:8888 注册，现在是向 ekServer1:8888 注册。

修改完成后，先启动两个包含 Eureka 服务器的程序，再启动 3 个包含服务提供者的程序，最后启动服务调用者的程序。启动完成后，我们可以通过 http://ekserver1:8080/hello 来查看调用 hello 服务的效果，这里的效果和 4.4.5 小节中运行的效果一致，所以就不再额外给出了。

同样，如果我们故意停止一个包含 Eureka 服务器的程序（比如 EurekaRibbonDemo-Server 程序），以此来模拟一台服务器失效的效果，由于这里实现了双服务器相互注册，所以如果再次在浏览器里输入“http://ekserver1:8080/hello”，那么依然可以看到调用服务后的输出效果。

4.5　配置 Ribbon 的常用参数

在上文里，我们是在 application.yml 里配置 Ribbon 诸如负载均衡策略等信息，在这部分里我们将归纳其他常用参数。

代码位置	视频位置
代码\第 4 章\RabbionBasicDemo	视频\第 4 章\常用的 Ribbon 参数

4.5.1　参数的影响范围

在 EurekaRibbonDemo-ServiceCaller 项目的 application.yml 里，我们采用 sayHello.ribbon 的形式配置参数，格式如下：

```
1   sayHello:
2     ribbon:
3       NFLoadBalancerRuleClassName: com.controller.MyRule
```

上述格式的参数是针对 sayHello 服务的。此外，我们还可以如下形式配置全局性的参数：

```
1     ribbon:
2       NFLoadBalancerRuleClassName: com.controller.MyRule
```

这里参数的作用范围是全局，也就是说，在 MyRule 中定义的负载均衡规则将作用在所有的服务上，而不仅仅是 sayHello 这个服务上。

4.5.2　归纳常用的参数

在 EurekaRibbonDemo-ServiceCaller 项目的 application.yml 里，我们通过如下代码配置了 Rule 规则、Ping 规则和可用服务器的列表，其中第 1 行的 sayHello 是服务名，说明这些配置不是全局性的，而是仅仅针对 sayHello 这个服务。

```
1   sayHello:
2     ribbon:
3       NFLoadBalancerRuleClassName: com.controller.MyRule
4       NFLoadBalancerPingClassName: com.controller.MyPing
5       listOfServers: http://localhost:1111/,http://localhost:2222/,
         http://localhost:3333
```

这里我们再给出一些其他的常用配置，具体的含义请看注释。

```
1   sayHello:
2     ribbon:
3       ConnectionTimeout: 200 #连接的超时时间
4       RealTimeout: 1000      #连接外带处理的超时时间
5       MaxAutoRetries: 5      #对当前请求实例的重试次数
6       MaxHttpConnectionsPerHost:5  #对每个主机每次最多的 HTTP 请求数
```

```
7       EnableConnectionPool:true  #是否启用连接池来管理连接
8       #只有第 7 行的值是 true，如下相关池的属性才能生效
9       PoolMaxThreads:10       #池中最大线程数
10      PoolMinThreads: 2 #池中最小线程数
11      PoolKeepAliveTime: 10   #线程的等待时间
12      PoolKeepAliveTimeUnits:SECONDS #等待时间的范围
```

4.5.3 在类里设置 Ribbon 参数

除了能在 application.yml 里设置外，我们还可以在 Java 类里编写针对 Ribbon 的配置参数。这里我们在 EurekaRibbonDemo-ServiceCaller 的基础上，重新编写一个服务调用者项目，命名为 EurekaRibbonConfigDemo-ServiceCaller，在其中演示通过类设置 Ribbon 参数的做法。

这个项目和 EurekaRibbonDemo-ServiceCaller 非常相似，但有如下差别。

差别 1，在 application.yml 里，去掉针对 IRule 和 IPing 实现类的配置，关键代码如下，其中我们能看到注释掉了第 3 行和第 4 行的代码。

```
1   sayHello:
2     ribbon:
3       # NFLoadBalancerRuleClassName: com.controller.MyRule
4       # NFLoadBalancerPingClassName: com.controller.MyPing
5       listOfServers: http://localhost:1111/,
6   http://localhost:2222/, http://localhost:3333
```

差别 2，新建 ConfigRibbon.java，在其中引入 MyRule 和 MyPing 类，代码如下。

```
1   省略必要的 package 和 import 代码
2   @Configuration
3   public class ConfigRibbon{
4       @Bean
5       public IRule getRule()
6       { return new MyRule(); }
7       @Bean
8       public IPing getPing()
9       { return new MyPing();}
10  }
```

在第 2 行里，我们通过@Configuration 这个注解说明本类是配置类。在第 5 行和第 8 行里，我们提供了 getRule 和 getPing 这两个方法，由于它们被@Bean 这个注解修饰，因此能被 Spring 容器自动注入。

差别 3，改写 Controller 部分的代码。

```
1   省略必要的 package 和 import 代码
2   @RestController
3   @Configuration
4   public class Controller {
5       //提供被@LoadBalanced 修饰的 RestTemplate 对象
6       @Bean
7       @LoadBalanced
8       public RestTemplate getRestTemplate()
9       { return new RestTemplate();   }
```

```
    //在 hello 方法里，无需再从配置文件里获得参数
    @RequestMapping(value = "/hello", method = RequestMethod.GET  )
    public String hello() {
        RestTemplate template = getRestTemplate();
        String retVal = template.getForEntity(
         "http://sayHello/sayHello/Eureka", String.class).getBody();
        return "In Caller, " + retVal;
    }
}
```

由于我们在 ConfigRibbon 类里已经把 MyRule 和 MyPing 通过@Bean 注解放入了容器，同时 hello 方法里的 RestTemplate 对象又被@LoadBalanced 注解修饰，因此通过 RestTemplate 实现负载均衡时，会自动地调用封装在 MyRule 和 MyPing 里的方法。

在 ConfigRibbon 类里定义的 Ribbon 配置是全局性的。此外，我们还可以通过@RibbonClinet 注解让配置参数只作用在单个服务上，具体的做法是新建一个名为 ConfigRibbonSayHello 的类，代码如下：

```
1  省略必要的 package 和 import 代码
2  @Configuration
3  @RibbonClient(name="sayHello",configuration=ConfigRibbon.class)
4  public class ConfigRibbonSayHello { }
```

其中，在类里可以不用放任何代码，但需要用类似第 3 行的注解来修饰这个类。

在定义@RibbonClient 注解时，需要用 configuration 来指定包含配置信息的类名，需要用 name 来指定这个配置所作用的服务名。

改写完成后，我们可以依次启动 Eureka 服务器、3 个服务提供者和基于配置文件的服务调用者，随后在浏览器里输入“http://localhost:8080/hello”，同样能在控制台里看到定义在 MyRule 和 MyPing 里的输出，这说明基于类的配置参数成功生效。

4.6　本 章 小 结

在本章里，我们首先介绍了目前比较常见的基于软件和硬件实现负载均衡的解决方案，并在此基础上介绍了 Spring Cloud 全家桶里实现负载均衡的重要组件：Ribbon。随后，我们在代码层面介绍了 Ribbon 各重要组件的用法，以及在 Eureka 框架里整合 Eureka 的各种做法。最后，我们还讲述了通过配置文件和配置类在 Eureka 框架里引入 Ribbon 参数的常见开发方式。