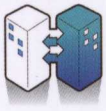
附赠
视频教学

以项目开发为主线
涵盖Spring Boot整合众多热门技术的开发案例
CSDN博客专家专业奉献
源代码下载

黄文毅 著

一步一步学 Spring Boot 2



微服务项目实战

清華大學出版社

作/者/简/介

黄文毅 CSDN博客专家，CSDN学院、网易学院和腾讯学院等网络学院讲师。曾就职于厦门星耀蓝图科技有限公司（为浦发银行、兴业银行、南京银行、湖南农信银行等银行开发系统）和上海美团（从事美团和大众点评后端开发工作），目前就职于厦门美图科技有限公司，从事美图秀秀和美拍后端开发工作。

一步一步学 Spring Boot 2 微服务项目实战

黄文毅 著

清華大學出版社
北京

内 容 简 介

本书以项目实战为主线，循序渐进地介绍了Spring Boot 2.0整合众多流行技术及在Web应用开发方面的各项技能。第1章由零开始引导读者快速搭建Spring Boot开发环境。第2章、第3章、第10章和第13章介绍Spring Boot数据访问应用，包括Spring Boot集成Druid、Spring Data JPA和MyBatis，快速访问MySQL和Mongo DB数据库。第4章至第6章重点介绍Spring Boot集成Thymeleaf模板引擎、事务使用以及拦截器和监听器的应用。第7章至第9章主要介绍Spring Boot使用Redis缓存和Quartz定时器、集成Log4j日志框架和发送Email邮件。第11、12章主要介绍Spring Boot集成ActiveMQ和异步调用、全局异常使用。第14、15章主要介绍Spring Boot应用监控和应用安全Security。第16、17章介绍Spring boot微服务在Zookeeper注册和Dubbo的使用、多环境配置和使用以及在Tomcat上的部署应用。第18章主要探索Spring Boot背后的原理和执行流程。为帮助读者快速掌握，编者还录制了与本书内容相关的教学视频，读者下载后即可观看学习。

本书适合Java开发人员、Spring Boot开发人员以及计算机专业的学生使用。

图书在版编目（CIP）数据

一步一步学Spring Boot 2：微服务项目实战/黄文毅著. —北京：清华大学出版社，2018
ISBN 978-7-302-50329-3

Ⅰ. ①一… Ⅱ. ①黄… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字（2018）第114912号

责任编辑： 王金柱
封面设计： 王　翔
责任校对： 王　叶
责任印制： 杨　艳
出版发行： 清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦A座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者： 北京嘉实印刷有限公司
经　　销： 全国新华书店
开　　本： 180mm×230mm　　**印　　张：** 13.75　　**字　　数：** 308千字
版　　次： 2018年8月第1版　　**印　　次：** 2018年8月第1次印刷
印　　数： 1~3500
定　　价： 59.00元

产品编号： 078582-01

前　言

Spring Boot 是近几年非常流行的微服务框架，相对于其他开发框架，Spring Boot 不但使用更加简单，而且功能更加丰富、性能更加稳定和健壮。Spring Boot 是在 Spring 框架基础上创建的一个全新的框架，其设计目的是简化 Spring 应用的搭建和开发过程，使得开发人员不仅能提高开发速度和生产效率，而且能够增强系统的稳定性和扩展性。

本书通过完整的项目实例带领大家一步一步学习 Spring Boot。通过实战项目学习 Spring Boot 的基础知识、使用技巧以及技术原理，最终达到融会贯通。

本书内容安排

本书是一本 Spring Boot 项目实战书籍，从内容结构上可以分为两部分，第 1~17 章是项目实战篇，主要介绍如何使用 Spring Boot、如何通过 Spring Boot 整合其他热门技术、如何通过 Spring Boot 开发完整的项目。第 18 章是原理篇，主要介绍 Spring Boot 背后的原理和执行的流程。

下面是各章的内容概要。

第 1 章介绍开始学习 Spring Boot 之前的环境准备、如何一分钟快速搭建 Spring Boot、Spring Boot 文件目录以及 Maven Helper 插件的安装和使用等。

第 2 章主要介绍如何安装和使用 MySQL、Spring Boot 集成 MySQL 数据库、Spring Boot 集成 Druid 以及通过实例讲解 Spring Boot 具体的运用。

第 3 章主要介绍 Spring Data JPA 核心接口及继承关系、在 Spring Boot 中集成 Spring Data JPA 以及如何通过 Spring Data JPA 实现增删改查及自定义查询等。

第 4 章主要介绍 Thymeleaf 模板引擎、Thymeleaf 模板引擎标签和函数、在 Spring Boot 中使用 Thymeleaf、集成测试以及 Rest Client 工具的使用。

第 5 章主要介绍 Spring 声明式事务、Spring 注解事务行为以及在 Spring Boot 中如何使用方法级别事务和类级别事务等。

第 6 章主要介绍如何在 Spring Boot 中使用过滤器 Filter 和监听器 Listener。

第 7 章主要介绍如何安装 Redis 缓存、Redis 缓存 5 种基本数据类型的增删改查、在 Spring Boot 中如何集成 Redis 缓存以及如何使用 Redis 缓存用户数据等。

第 8 章主要介绍 Log4j 基础知识、在 Spring Boot 中集成 Log4j、Log4j 在 Spring Boot 中的运用以及如何把日志打印到控制台和记录到日志文件中。

第 9 章主要介绍在 Spring Boot 中使用 XML 配置和 Java 注解两种方式定义和使用 Quartz 定时器以及如何在 Spring Boot 中通过 JavaMailSender 接口给用户发送广告邮件等。

第 10 章主要介绍如何在 Spring Boot 中集成 MyBatis 框架、通过 MyBatis 框架实现查询等功能以及如何使用 MyBatisCodeHelper 插件快速生成增删改查代码。

第 11 章主要介绍 ActiveMQ 的安装与使用、Spring Boot 集成 ActiveMQ、利用 ActiveMQ 实现异步发表微信说说以及 Spring Boot 异步调用 @Async 等。

第 12 章主要介绍 Srping Boot 全局异常使用、自定义错误页面、全局异常类开发、Retry 重试机制等。

第 13 章主要介绍如何安装和使用 MongoDB 数据库、NoSQL Manager for MongoDB 客户端的安装与使用以及在 Spring Boot 中集成 MongoDB 数据库开发简单的功能等。

第 14 章主要介绍 Spring Security 的基础知识、Spring Boot 如何集成 Spring Security、利用 Spring Security 实现授权登录以及利用 Spring Boot 实现数据库数据授权登录等。

第 15 章主要介绍如何通过 Spring Boot 监控和管理应用、自定义监控端点以及自定义 HealthIndicator 等。

第 16 章主要介绍如何安装并运行 Zookeeper、Spring Boot 集成 Dubbo、my-spring-boot 项目的服务拆分和实践、正式版 API 如何发布、服务注册等。

第 17 章主要介绍 Spring Boot 多环境配置及使用、Spring Boot 如何打包成 War 包并部署到外部 Tomcat 服务器上等。

第 18 章主要回顾 MySpringApplication 入口类的注解和 run 方法的原理，梳理 Spring Boot 启动执行的流程并简单分析 spring-boot-starter 起步依赖原理等。

学习本书的预备知识

Java 基础

读者需要掌握 J2SE 基础知识，这是最基本的也是最重要的。

Java Web 开发技术

在项目实战中需要用到 Java Web 的相关技术，比如 Spring、HTML、Tomcat、MyBatis 等技术。

数据库基础

读者需要掌握主流数据库基本知识，比如 MySQL 等，掌握基本的 SQL 语法以及常用数据库的安装。

本书使用的软件版本

本书项目实战开发环境为 Windows 10，开发工具使用 IntelliJ IDEA 2016.2，JDK 使用 1.8 版本，Tomcat 使用 1.8 版本，Spring Boot 使用 2.0.0.RC1 版本。

读者对象

本书适合所有 Java 编程语言开发人员、所有对 Spring Boot 感兴趣并希望使用 Spring Boot 开发框架进行开发的人员、缺少 Spring Boot 项目实战经验以及对 Spring Boot 内部原理感兴趣的开发人员学习。

源代码和教学视频下载

GitHub 源代码下载地址：git@github.com:huangwenyi10/stepbystep-learn-springboot.git

教学视频下载地址：https://pan.baidu.com/s/1A5xjEcvE5A2T6I5bmAmkYQ

也可以扫描二维码下载。

如果你在下载过程中遇到问题，可发送邮件至 booksaga@126.com 获得帮助，邮件标题为“一步一步学 Spring Boot：微服务项目实战”。

勘误与交流

限于笔者水平和写作时间，书中出现疏漏之处在所难免，欢迎大家通过电子邮件等方式批评指正。

笔者的邮箱：huangwenyi10@163.com

笔者的博客：http://blog.csdn.net/huangwenyi1010

致谢

本书能够顺利出版，首先要感谢清华大学出版社王金柱编辑给笔者一次和大家分享技术、交流学习的机会，感谢王金柱编辑在本书出版过程中的辛勤付出。

感谢厦门星耀蓝图科技有限公司，笔者的Spring Boot知识是在贵公司积累沉淀的，书中很多知识点和项目实战经验都来源于贵公司，感谢公司总经理杨小雄、主管林良昆、架构师高志强、同事陈明元和林腾亚对笔者的关心和帮助。

感谢上海美团科技有限公司的同事，感谢公司主管王纪伟、导师叶永林、组长文慧、同事杨伟勤对笔者的栽培与帮助。

感谢笔者的家人和何庆华学长，他们对笔者生活的照顾使得笔者没有后顾之忧，全身心投入本书的写作当中。

最后，感谢你选择了本书，让我们开始 Spring Boot 的探险之旅吧！

黄文毅

2018.2.1

目　录

第1章

第一个 Spring Boot 项目

本章主要介绍学习 Spring Boot 之前的环境准备，包括如何一分钟快速搭建 Spring Boot、Spring Boot 文件目录的简单介绍以及 Maven Helper 插件的安装和使用等。

1.1 Spring Boot 简单介绍

Spring Boot 是目前流行的微服务框架，倡导“约定优先于配置”，其设计目的是用来简化新 Spring 应用的初始化搭建以及开发过程。Spring Boot 提供了很多核心的功能，比如自动化配置、starter 简化 Maven 配置、内嵌 Servlet 容器、应用监控等功能，让我们可以快速构建企业级应用程序。本书是一本实战教程，不会浪费太多笔墨来介绍 Spring Boot 原理，但会通过具体的项目实例一步步揭开 Spring Boot 神秘的面纱。

1.2 Spring Boot 环境准备

本节将介绍如何安装 JDK、IntelliJ IDEA 以及 Apache Maven。在开始学习 Spring Boot 之前，我们需要准备好开发环境，本书主要以 Window 操作系统为例进行介绍。如果电脑中已经安装 JDK、IntelliJ IDEA 或者 Apache Maven，可以跳过本节的内容。

1.2.1 安装 JDK

JDK（Java SE Development Kit）建议使用 1.8 及以上的版本，官方下载路径为 http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html。大家可以根据自己 Windows 操作系统的配置选择合适的 JDK 1.8 安装包，这里不过多描述。

软件下载完成之后，双击安装文件，出现安装界面，如图 1-1 所示。一路单击【下一步】按钮，即可完成安装。这里把 JDK 安装在路径：C:\Program Files\Java\jdk1.8.0_77 下。

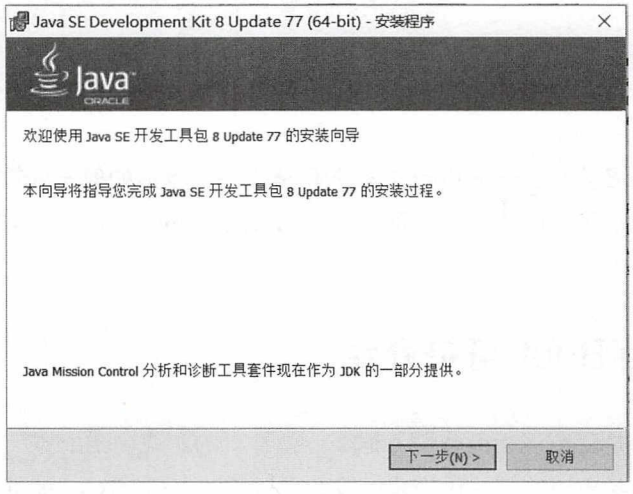


图 1-1 JDK 安装界面

安装完成后，需要配置环境变量 JAVA_HOME，步骤如下：

步骤 01 在电脑桌面上右击【我的电脑】→【属性】→【高级系统设置】→【环境变量】→【系统变量 (S)】→【新建】，出现新建环境变量窗口，如图 1-2 所示。

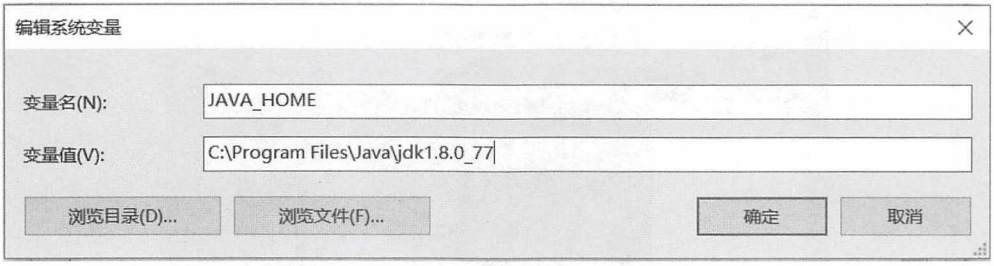


图 1-2 新建环境变量窗口

步骤 02 在【变量名】和【变量值】中分别填入 JAVA_HOME 和 C:\Program Files\Java\jdk1.8.0_77，单击【确定】按钮。

步骤 03 JAVA_HOME 配置好之后，将 %JAVA_HOME%\bin 加入【系统变量】的 path 中。完成后，打开命令行窗口，输入命令 java -version。出现如图 1-3 所示的提示，即表示安装成功。

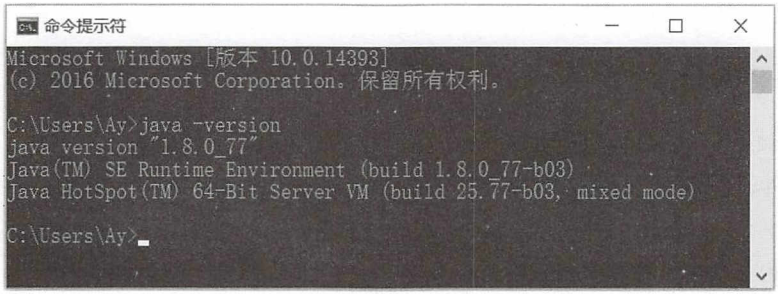


图 1-3 安装成功

JDK 安装路径最好不要出现中文，否则会出现意想不到的结果。

1.2.2 安装 IntelliJ IDEA

在 IntelliJ IDEA 的官方网站（http://www.jetbrains.com/idea/）可以免费下载 IDEA。下载完 IDEA 后，运行安装程序，按提示安装即可。本书使用的是 IntelliJ IDEA 2016.2 版本，当然大家也可以使用其他版本的 IDEA，只要版本不要过低即可。安装成功之后，打开软件，界面如图 1-4 所示。

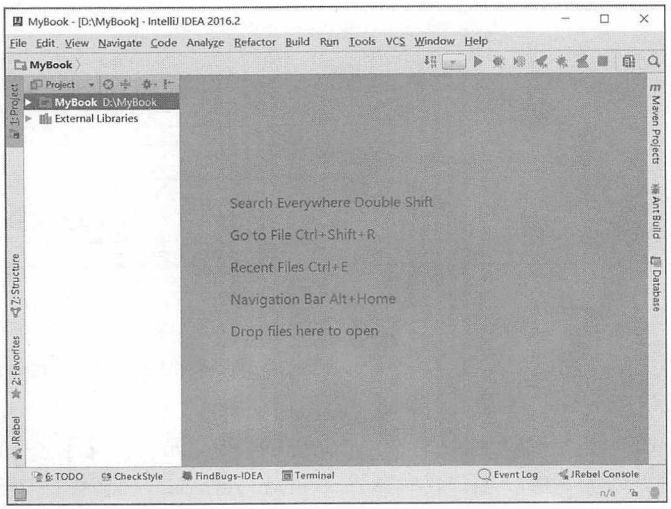


图 1-4 IntelliJ IDEA 软件界面

1.2.3 安装 Apache Maven

Apache Maven 是目前流行的项目管理和构建自动化工具。虽然 IDEA 已经包含 Maven 插件，但是还是希望大家在工作中能够安装自己的 Maven 插件，方便以后项目配置。大家可以通过 Maven 的官方网站（http://maven.apache.org/download.cgi）下载最新版的 Maven，本书的 Maven 版本为 apache-maven-3.5.0。

下载完后解压缩即可，例如解压到 D 盘，然后将 Maven 的安装路径 D:\apache-maven-3.5.0\bin 加入 Windows 的环境变量 path 中。安装完成后，在命令行窗口执行命令：mvn -v，如果输出如图 1-5 所示的内容，表示 Maven 安装成功。

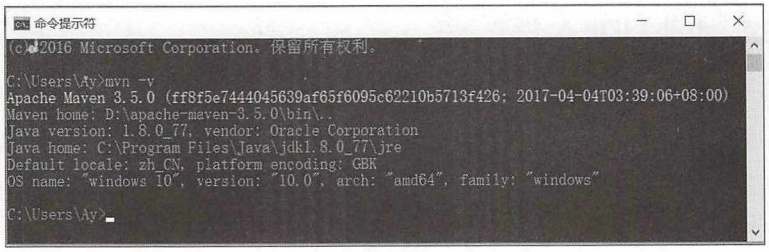


图 1-5 Maven 安装成功

接下来，我们要在 IntelliJ IDEA 下配置 Maven，具体步骤如下：

步骤 01　在 Maven 安装目录（D:\apache-maven-3.5.0）下新建文件夹 repository，用来作为本地仓库。

步骤 02　在 IntelliJ IDEA 界面中选择【File】→【Settings】，在出现的窗口中找到 Maven 选项，分别把【Maven home directory】【User settings file】【Local repository】设置为我们自己 Maven 的相关目录，如图 1-6 所示。

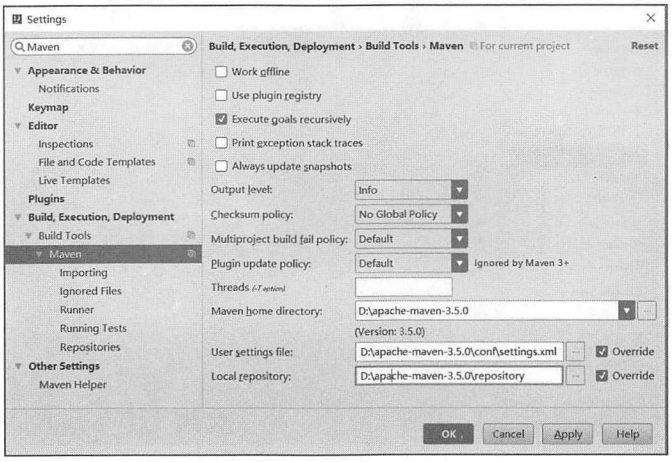


图 1-6　Maven 设置窗口

步骤 03　设置完成后，单击【Apply】→【OK】。到此，Maven 在 IntelliJ IDEA 的配置完成。

这里需要注意的是，之所以把 Maven 默认仓库（C:\${user.home}\.m2\repository）的路径改为我们自己的目录（D:\apache-maven-3.5.0\repository），是因为 repository 仓库到时候会存放很多 jar 包，放在 C 盘影响电脑的性能。

1.3　一分钟快速搭建 Spring Boot 项目

1.3.1　使用 Spring Initializr 新建项目

使用 IntelliJ IDEA 创建 Spring Boot 项目有多种方式，比如 Maven 和 Spring Initializr。这里只介绍 Spring Initializr 这种方式，因为这种方式不但可以生成完整的目录结构，还可以生成一个默认的主程序，节省时间。我们的目的是掌握 Spring Boot 相关的知识，而不是学一堆花样。具体步骤如下：

步骤 01 在 IntelliJ IDEA 界面中，单击【File】→【New】→【Product】，在弹出的窗口中选择【Spring Initializr】选项，在【Product SDK】下拉框中选择 JDK 的安装路径，如果没有，就新建一个，单击【Next】按钮，如图 1-7 所示。

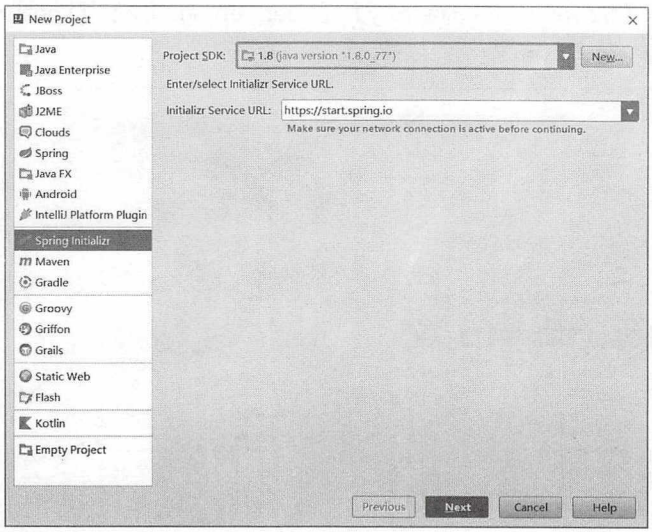


图 1-7 新建 Spring Boot 项目

步骤 02 选择【Spring Boot Version】，这里按默认版本（本书使用的 Spring Boot 版本为 2.0.0.RELEASE）即可。勾选【Web】选项，然后单击【Next】按钮，如图 1-8 所示。

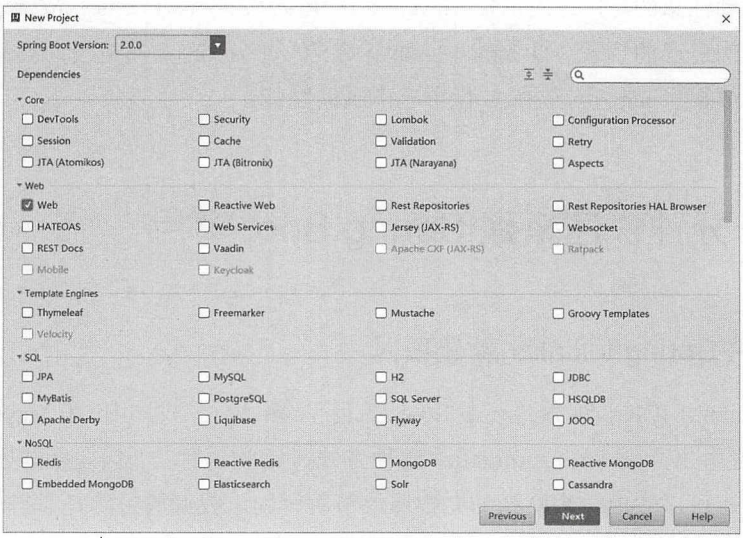


图 1-8 选择版本和组件

步骤 03 填写项目名称【my-spring-boot】，其他选项保持默认即可。然后单击【Finish】按钮，至此，一个完整的 Spring Boot 创建完成，如图 1-9 所示。

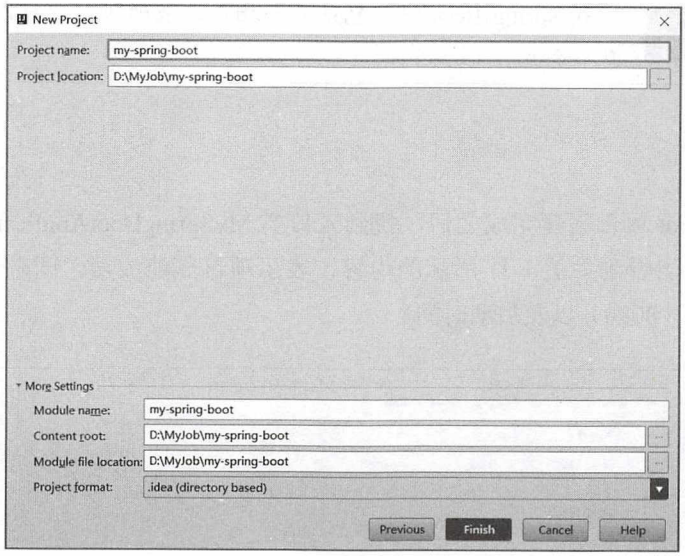


图 1-9 填写项目名称窗口

步骤 04 在 IDEA 开发工具上，找到刷新依赖的按钮（Reimport All Maven Projects），下载相关的依赖包，这时开发工具开始下载 Spring Boot 项目所需依赖包，如图 1-10 所示。

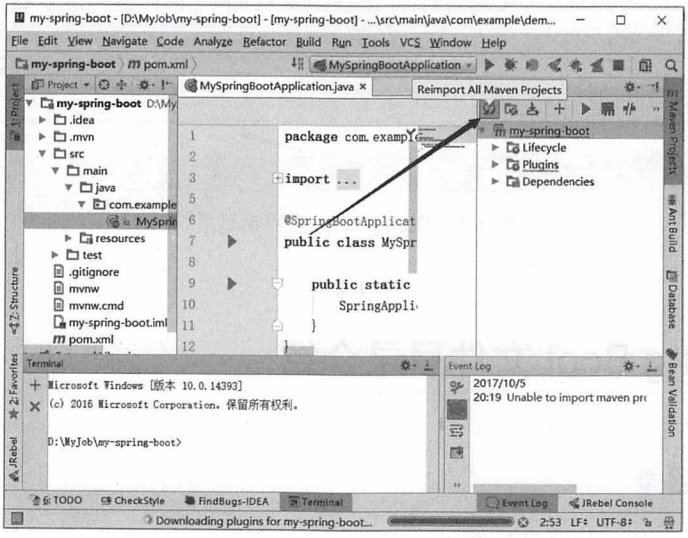


图 1-10 刷新依赖窗口

至此，Spring Boot 项目创建完成。

下载 Spring Boot 依赖包是一个相对漫长的过程，可以去喝杯茶休息一会儿。

1.3.2 测试

Spring Boot 项目创建完成之后，找到入口类 MySpringBootApplication 中的 main 方法并运行。当看到如图 1-11 所示的内容，表示项目启动成功。同时还可以看出项目启动的端口（8080）以及启动时间。

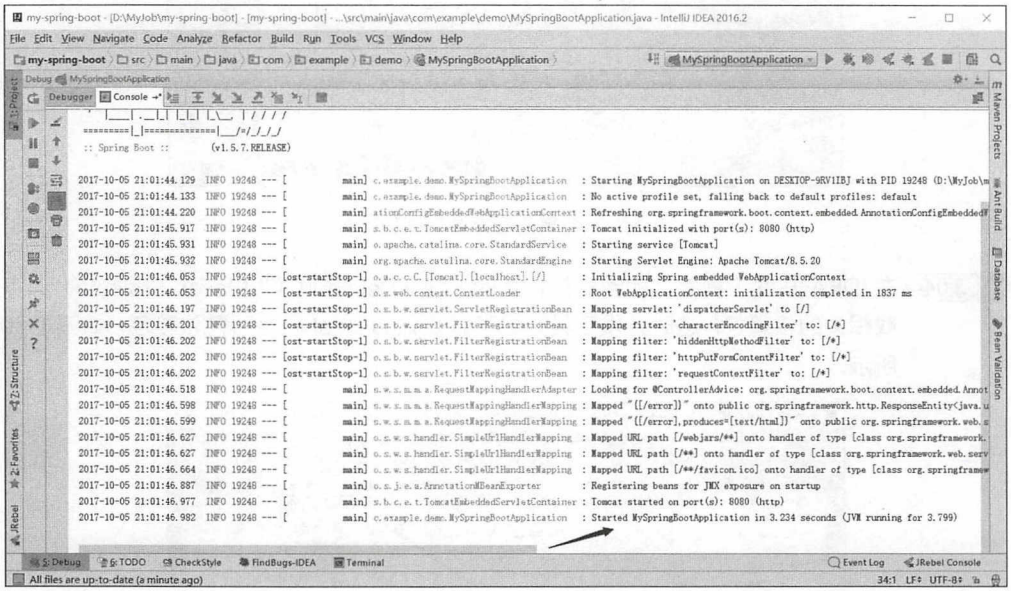


图 1-11 Spring Boot 项目启动成功

1.4 Spring Boot 文件目录介绍

1.4.1 工程目录

Spring Boot 的工程目录如图 1-12 所示。

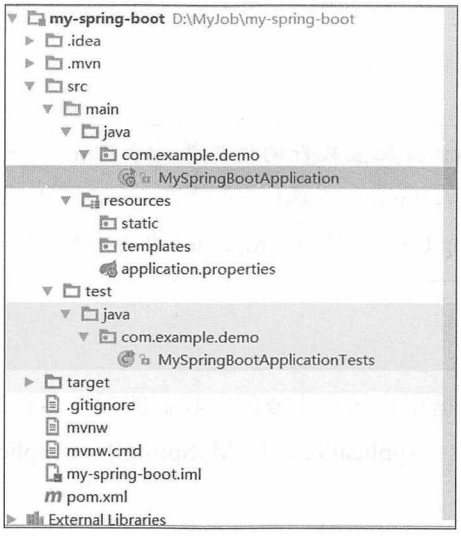


图 1-12 Spring Boot 工程目录

- /src/main/java：目录下放置所有的 Java 文件（源代码文件）。
- /src/main/resources：用于存放所有的资源文件，包括静态资源文件、配置文件、页面文件等。
- /src/main/resources/static：用于存放各类静态资源。
- /src/main/resources/application.properties：配置文件，这个文件非常重要。Spring Boot 默认支持两种配置文件类型，即 .properties 和 .yml。
- /src/main/resources/templates：用于存放模板文件，如 Thymeleaf（这个技术不懂不用着急，以后会介绍）模板文件。
- /src/test/java：放置单元测试类 Java 代码。
- /target：放置编译后的 .class 文件、配置文件等。

Spring Boot 将很多配置文件进行了统一管理，且配置了默认值。Spring Boot 会自动在 /src/main/resources 目录下找 application.properties 或者 application.yml 配置文件。找到后将运用此配置文件中的配置，否则使用默认配置。这两种类型的配置文件有其一即可，也可以两者并存。两者区别如下：

```
application.properties:
server.port = 8080
```

```
application.yml:
server:
    port:8080
```

properties 配置文件的优先级高于 .yml 。在 .properties 文件中配置了 server.port = 8080，同时在 .yml 中配置了 server.port = 8090。Spring Boot 将使用 .properties 中的 8080 端口。

1.4.2 入口类

入口类的类名是根据项目名称生成的，我们的项目名称是 my-spring-boot，故入口类的类名是项目名称 + Application，即 MySpringBootApplication.java。入口类的代码很简单，代码如下：

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class MySpringBootApplication {

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

- @SpringBootApplication：是一个组合注解，包含 @EnableAutoConfiguration、@ComponentScan 和 @SpringBootConfiguration 三个注解，是项目启动注解。如果我们使用这三个注解，项目依旧可以启动起来，只是过于烦琐。因此，用 @SpringBootApplication 进行简化。
- @SpringApplication.run：应用程序开始运行的方法。

入口类需要放置在包的最外层，以便能够扫描到所有子包中的类。

1.4.3 测试类

Spring Boot 的测试类主要放置在 /src/test/java 目录下面。项目创建完成后，Spring Boot 会自动为我们生成测试类 MySpringBootApplicationTests.java。其类名也是

根据项目名称 + ApplicationTests 生成的。测试类的代码如下：

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class MySpringBootApplicationTests {
    @Test
    public void contextLoads() {
    }
}
```

- @RunWith（SpringRunner.class）：@RunWith(Parameterized.class) 是参数化运行器，配合 @Parameters 使用 Junit 的参数化功能。查源代码可知，SpringRunner 类继承自 SpringJUnit4ClassRunner 类，此处表明使用 SpringJUnit4ClassRunner 执行器，此执行器集成了 Spring 的一些功能。如果只是简单的 Junit 单元测试，该注解可以去掉。
- @SpringBootTest：此注解能够测试我们的 SpringApplication，因为 Spring Boot 程序的入口是 SpringApplication，所以基本上所有配置都会通过入口类去加载，而该注解可以引用入口类的配置。
- @Test：Junit 单元测试的注解，在方法上，注解表示一个测试方法。

当我们右击执行 MySpringBootApplicationTests.java 中的 contextLoads 方法的时候，大家可以看到控制台打印的信息和执行入口类中的 SpringApplication.run() 方法，打印的信息是一致的。由此便知，@SpringBootTest 引入了入口类的配置。

1.4.4 pom 文件

Spring Boot 项目下的 pom.xml 文件主要用来存放依赖信息。具体代码如下（部分代码已省略）：

```
<parent>
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.RC1</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

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

- spring-boot-starter-parent：是一个特殊的 starter，用来提供相关的 Maven 默认依赖，使用它之后，常用的包依赖可以省去 version 标签。
- spring-boot-starter-web：只要将其加入项目的 Maven 依赖中，我们就得到了一个可执行的 Web 应用。该依赖中包含许多常用的依赖包，比如 spring-web、spring-webmvc 等。我们不需要做任何 Web 配置，便能获得相关 Web 服务。
- spring-boot-starter-test：这个依赖和测试相关，只要引入它，就会把所有与测试相关的包全部引入。
- spring-boot-maven-plugin：这是一个 Maven 插件，能够以 Maven 的方式为应用提供 Spring Boot 的支持，即为 Spring Boot 应用提供了执行 Maven 操作的可能。能够将 Spring Boot 应用打包为可执行的 jar 或 war 文件。

1.5 Maven Helper 插件的安装和使用

1.5.1 Maven Helper 插件介绍

Maven Helper 是一款可以方便查看 Maven 依赖树的插件，可以在 IntelliJ IDEA 上安装使用它。Maven Helper 支持多种视图来查看 Maven 依赖，同时可以帮助我们分析 pom 文件中的依赖是否存在冲突，方便快速定位错误。

1.5.2 Maven Helper 插件的安装

安装 Maven Helper 插件很简单，在菜单栏选择【File】→【Settings】→【Plugins】，在搜索框中输入【Maven Helper】，然后单击【Install】安装即可。安装成功之后，重启 IntelliJ IDEA 即可使用。

1.5.3 Maven Helper 插件的使用

插件安装成功之后，如图 1-13 所示，打开 pom 文件，除了 Text 视图外，多了一个 Dependency Analyzer 视图。

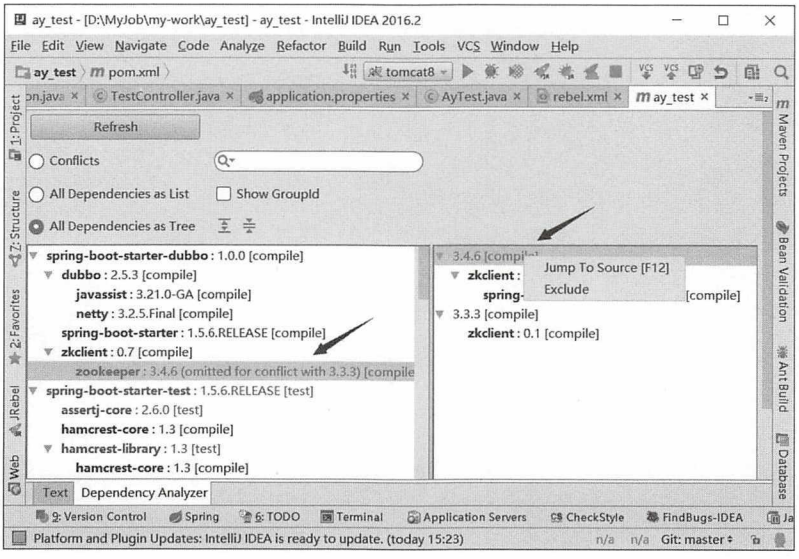


图 1-13 Maven Helper 插件

- All Dependencies as List：以列表方式显示所有依赖包。
- All Dependencies as Tree：以树形方式显示所有依赖包。
- Conflicts：查看所有有冲突的依赖包，如果存在冲突，就会显示红色，同时会显示在右侧视图。我们可以单击有冲突的包，右击【Exclude】来排除冲突，通过右击【Jump To Source[F12]】可以跳转到源代码。

第2章

集成 MySQL 数据库

本章将介绍 MySQL 的安装和使用、Spring Boot 集成 MySQL 数据库、Spring Boot 集成 Druid 以及通过实例讲解 Spring Boot 具体的运用。

2.1 MySQL 介绍与安装

数据库类型有很多，比如有 MySQL、Oracle 这样的关系型数据库，又有 MongoDB、NoSQL 这样的非关系型数据库。本节主要讲解目前项目中运用广泛的关系型数据库 MySQL。

2.1.1 MySQL 概述

MySQL 是目前项目中运用广泛的关系型数据库。无论在什么样的公司，普通公

司也好，互联网公司也好，都运用甚广。MySQL 所使用的 SQL 语言是用于访问数据库的最常用的标准化语言。MySQL 软件由于其体积小、速度快、总体拥有成本低，尤其是开放源码这一特点，一般中小型网站的开发都选择 MySQL 作为网站数据库。

2.1.2 MySQL 的安装

MySQL 的安装很简单，安装方式也有多种。大家可以到 MySQL 的官方网站（https://dev.mysql.com/downloads/mysql/）下载 MySQL 安装软件，按照提示一步一步安装即可。如果电脑中已经安装 MySQL，可略过此节，本书使用的 MySQL 版本为 5.7.17。

安装完成之后，我们需要检验 MySQL 是否安装成功。具体步骤如下：

步骤 01 打开命令行窗口，进入 MySQL 安装目录，笔者的 MySQL 安装目录是 C:\Program Files\MySQL\MySQL Server 5.7\bin。

步骤 02 在命令行窗口中输入命令 mysql -uroot -p 和密码登录 MySQL，然后输入命令 status，出现如图 2-1 所示的信息，表示安装成功。

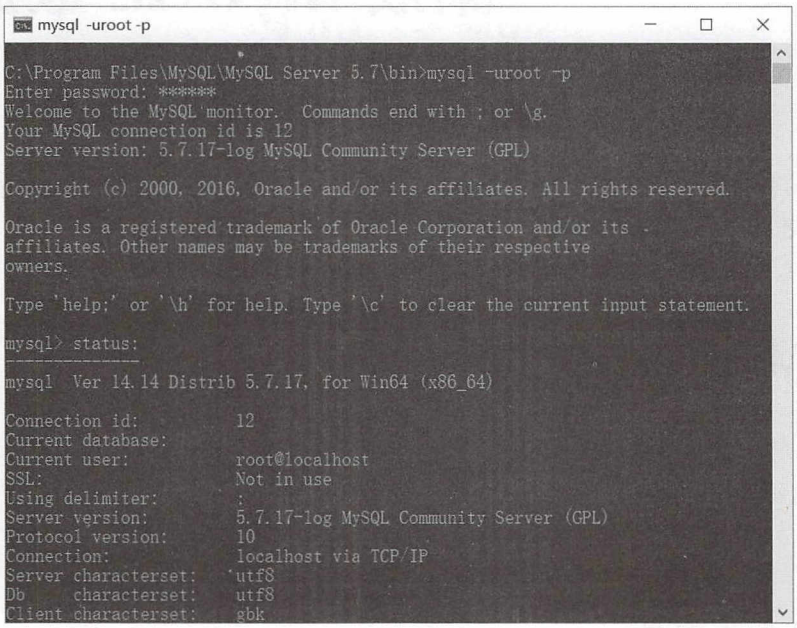


图 2-1 MySQL 安装状态

2.2　集成 MySQL 数据库

Spring Boot 集成 MySQL 非常简单，因为 Spring Boot 包含一个功能强大的资源库，借助于 Spring Boot 框架，我们可以不用编写原始的访问数据库的代码，也不用调用 JDBC（Java Data Base Connectivity）或者连接池等被称为底层的代码，我们将在更高层次上访问数据库。

2.2.1　引入依赖

集成 MySQL 数据库之前，我们需要在项目的 pom 文件中添加 MySQL 所需的依赖，具体代码如下：

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

- mysql-connector-java：MySQL 连接 Java 的驱动程序。
- spring-boot-starter-jdbc：支持通过 JDBC 连接数据库。

2.2.2　添加数据库配置

在 pom 文件中引入 MySQL 所需的 Maven 依赖之后，我们需要在 application.properties 文件中添加如下的配置信息：

```
### mysql 连接信息
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/test
### 用户名
spring.datasource.username=root
### 密码
spring.datasource.password=123456
### 驱动
```

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

2.2.3 设计表和实体

配置信息添加完成之后，在 MySQL 数据库中创建一张表。MySQL 安装成功之后，默认有一个 test 数据库，在 test 数据库里新建表 ay_user。具体建表的 SQL 语句如下：

```
-- ----------------------------
-- 用户表
-- Table structure for ay_user
-- ----------------------------
DROP TABLE IF EXISTS 'ay_user';
CREATE TABLE 'ay_user' (
    'id' varchar(32) NOT NULL COMMENT '主键',
    'name' varchar(10) DEFAULT NULL COMMENT '用户名',
    'password' varchar(32) DEFAULT NULL COMMENT '密码'
);
```

数据库表 ay_user 字段很简单，包括主键 id、用户名 name 和密码 password。ay_user 表创建好之后，我们往数据库表 ay_user 中插入两条数据，具体插入数据的 SQL 语句如下：

```
INSERT INTO 'ay_user' ('id', 'name', 'password') VALUES ('1', '阿毅', '123456');
INSERT INTO 'ay_user' ('id', 'name', 'password') VALUES ('2', '阿兰', '123456');
```

除了使用 SQL 语句插入之外，还可以使用 Navicat for MySQL 客户端插入数据，2.3 节会详细介绍。数据插入成功之后，可在 MySQL 客户端查询到两条数据，具体如图 2-2 和图 2-3 所示。

图 2-2 创建表 ay_user

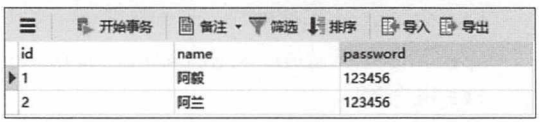


图 2-3 插入两条数据

表和数据准备好之后，在项目的目录下（/src/main/java/com.example.demo.model）新建实体类，具体代码如下：

```
/**
 * 描述：用户表
 * @Author 阿毅
 * @date   2017/10/8.
 */
public class AyUser {

    //主键
    private String id;
    //用户名
    private String name;
    //密码
    private String password;

    //此处省略 set 、get 方法
}
```

至此，数据库表、数据、实体已经全部准备好了。接下来就开始使用开发测试用例进行测试。

2.3　集成测试

2.3.1　测试用例开发

在项目的单元测试类 MySpringBootApplicationTests.java 中添加如下代码：

```
@Resource
private JdbcTemplate jdbcTemplate;
/**
 * MySQL 集成 Spring Boot 简单测试
 */
@Test
public void mySqlTest(){
    String sql = "select id,name,password from ay_user";
    List<AyUser> userList =(List<AyUser>) jdbcTemplate.query
```

```
        (sql, new RowMapper<AyUser>(){
            @Override
            public AyUser mapRow(ResultSet rs, int rowNum) throws
            SQLException {
                AyUser user = new AyUser();
                user.setId(rs.getString("id"));
                user.setName(rs.getString("name"));
                user.setPassword(rs.getString("password"));
                return user;
            }
        });
        System.out.println("查询成功：");
        for(AyUser user:userList){
            System.out.println("【id】:" + user.getId() + ";【name】:"
            + user.getName());
        }
    }
```

- JdbcTemplate：是一个通过 JDBC 连接数据库的工具类。2.2 节引入了依赖 spring-boot-starter-jdbc 中包含的 spring-jdbc 包，我们可以通过这个工具类对数据库进行增、删、改、查等操作。
- @Resource：自动注入，通过这个注解，在项目启动之后，Spring Boot 会帮助我们实例化一个 JdbcTemplate 对象，省去初始化工作。
- query() 方法：JdbcTemplate 对象中的查询方法，通过传入 SQL 语句和 RowMapper 对象可以查询出数据库中的数据。
- RowMapper 对象：RowMapper 对象可以将查询出的每一行数据封装成用户定义的类，在上面的代码中，通过调用 RowMapper 中的 mapRow 方法，将数据库中的每一行数据封装成 AyUser 对象，并返回。

SQL 语句要么全部大写，要么全部小写，不要大小写混用。

2.3.2 测试

单元测试方法开发完成之后，双击方法 mySqlTest，右击执行一下单元测试，这时可以在控制台看到打印信息，具体如下：

查询成功：

【id】：1；【name】：阿毅

【id】：2；【name】：阿兰

至此，Spring Boot集成MySQL数据库大功告成，这一节的内容简单但是非常重要，之后的章节都是在本节的基础上进行开发的。

2.3.3 Navicat for MySQL 客户端安装与使用

Navicat for MySQL 是连接 MySQL 数据库的客户端工具，通过使用该客户端工具方便对数据库进行操作，比如创建数据库表、添加数据等。如果大家已经安装其他的 MySQL 客户端，可以略过本节。

Navicat for MySQL 的安装也非常简单，大家到网上下载安装即可。安装完成之后，打开软件，界面如图 2-4 所示。

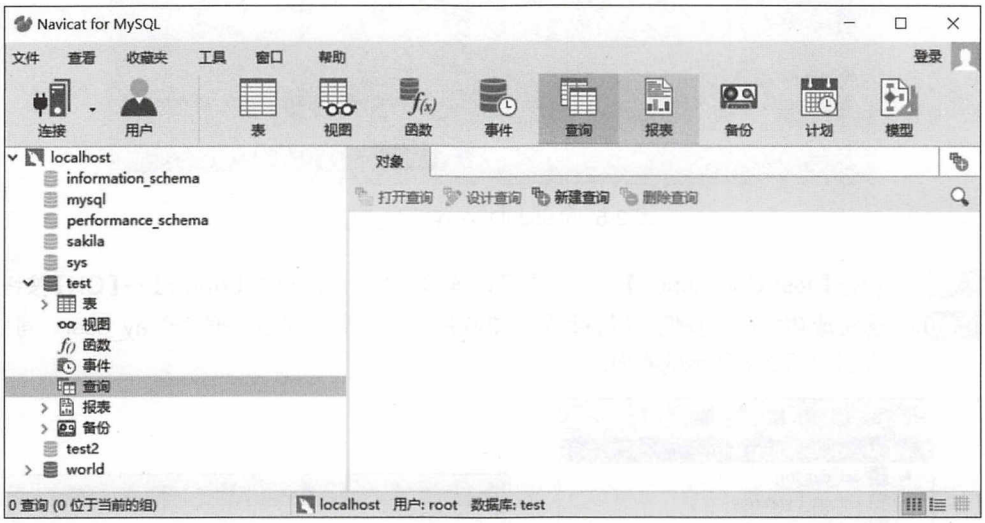


图 2-4 Navicat for MySQL 界面

我们可以通过【查询】→【新建查询】在弹出的窗口中编写相关的查询语句来查询数据。当然，还有很多操作，大家可以自己去使用和掌握它，这里就不一一描述了。

2.3.4 IntelliJ IDEA 连接 MySQL

除了通过 Navicat for MySQL 客户端连接数据库之外，如果不喜欢在自己的电脑中安装一堆软件，我们还可以通过 IntelliJ IDEA 来连接 MySQL 数据库。具体步骤如下：

步骤 01 在 IntelliJ IDEA 界面中，单击右侧的【Database】→【New（加号）】→【Data Source】→【MySQL】，在弹出的窗口中输入主机、用户名、密码、端口等信息，如图 2-5 所示。

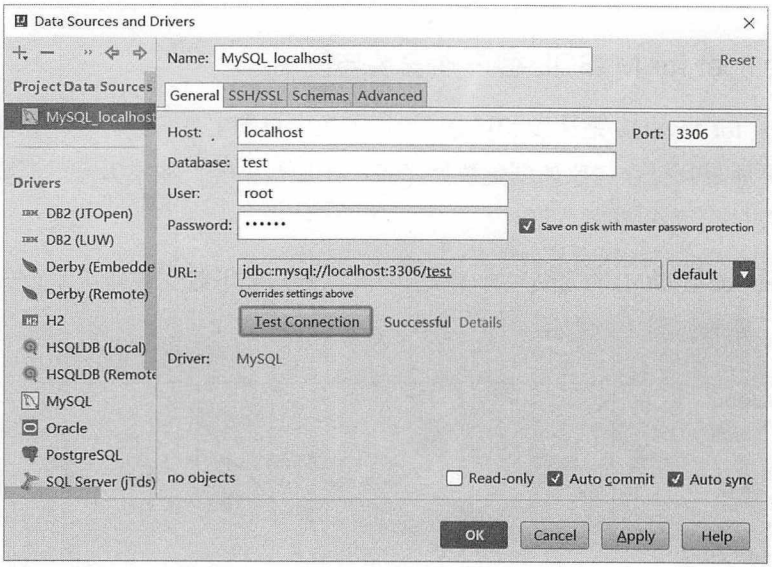


图 2-5 IntelliJ IDEA 连接 MySQL

步骤 02 单击【Test Connection】测试是否可以连接成功，然后单击【Apply】→【OK】按钮。

步骤 03 连接成功之后，我们可以看到如图 2-6 所示的界面，双击数据库表 ay_user，可以看到如图 2-7 所示的界面。

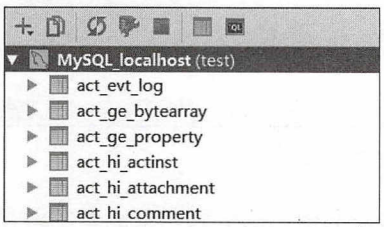


图 2-6 连接 MySQL 成功界面

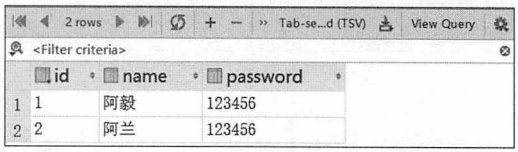


图 2-7 双击表 ay_user 界面

成功连接 MySQL 数据库之后，我们可以在图 2-6 中看到停止数据库、刷新数据库、

命令行窗口等按钮，通过这些按钮可以停止和刷新数据库，或者打开命令行窗口编写 SQL 语句。在图 2-7 中，我们可以查询某张表的数据，单击“+”号、“-”号按钮进行数据的添加和删除，还可以在 Filter criteria 输入框中编写过滤条件，搜索出我们所需要的数据。比如在 Filter criteria 输入框中输入 id = '1' and name = ' 阿毅 ' 或者 name like '% 兰 %' and id = '2'，查询结果如图 2-8 和图 2-9 所示。

图 2-8　查询结果 1

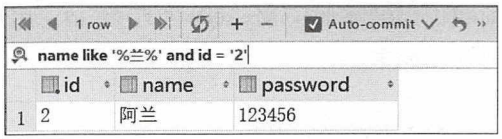


图 2-9　查询结果 2

2.4　集成 Druid

2.4.1　Druid 概述

Druid 是阿里巴巴开源项目中的一个数据库连接池。Druid 是一个 JDBC 组件，包括三部分：① DruidDriver 代理 Driver，能够提供基于 Filter-Chain 模式的插件体系；② DruidDataSource 高效可管理的数据库连接池；③ SQLParser，支持所有 JDBC 兼容的数据库，包括 Oracle、MySQL、SQL Server 等。Durid 在监控、可扩展、稳定性和性能方面具有明显的优势，通过其提供的监控功能可以观察数据库连接池和 SQL 查询的工作情况，使用 Druid 连接池可以提高数据库的访问性能。

2.4.2　引入依赖

我们在项目的 pom 文件中继续添加 druid 的依赖，具体代码如下：

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.4</version>
</dependency>
```

在这里，笔者使用的是 1.1.4 版本，添加完依赖之后，IntelliJ IDEA 会自动帮助我们下载依赖包，只要刷新一下依赖即可。

2.4.3 Druid 配置

依赖添加完成之后，在 application.properties 配置文件中继续添加 Druid 配置，之前我们已经添加了 MySQL 的连接 url、用户名、密码等配置，application.properties 完整代码如下：

```
### MySQL 连接信息
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/test
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
###   数据源类别
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
### 初始化大小，最小，最大
spring.datasource.initialSize=5
spring.datasource.minIdle=5
spring.datasource.maxActive=20
### 配置获取连接等待超时的时间，单位是毫秒
spring.datasource.maxWait=60000
### 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒
spring.datasource.timeBetweenEvictionRunsMillis=60000
### 配置一个连接在池中最小生存的时间，单位是毫秒
spring.datasource.minEvictableIdleTimeMillis=300000
spring.datasource.validationQuery=SELECT 1 FROM DUAL
spring.datasource.testWhileIdle=true
spring.datasource.testOnBorrow=false
spring.datasource.testOnReturn=false
### 打开 PSCache，并且指定每个连接上 PSCache 的大小
spring.datasource.poolPreparedStatements=true
spring.datasource.maxPoolPreparedStatementPerConnectionSize=20
### 配置监控统计拦截的 filters，去掉后监控界面的 SQL 无法统计，'wall' 用于防火墙
spring.datasource.filters=stat,wall,log4j
### 通过 connectProperties 属性来打开 mergeSql 功能，慢 SQL 记录
```

```
    spring.datasource.connectionProperties=druid.stat.
mergeSql=true;druid.stat.slowSqlMillis=5000
    ### 合并多个 DruidDataSource 的监控数据
    #spring.datasource.useGlobalDataSourceStat=true
```

上面每一个配置的含义都有相关的注释，这里就不再过多介绍。这里要注意的是，在 .properties 配置文件中，# 字符是注释符号。

2.4.4 开启监控功能

开启监控功能的方式有多种：① 使用原生的 Servlet、Filter 方式，然后通过 @ServletComponentScan 启动扫描包进行处理；② 使用代码注册 Servlet 和 Filter 的方式处理。这里我们选择 Spring Boot 推荐的第二种方式实现，在项目的 Java 目录下（/src/main/java/com.example.demo.filter）新建一个配置类 DruidConfiguration.java，具体代码如下：

```
@Configuration
public class DruidConfiguration {

    @Bean
    public ServletRegistrationBean druidStatViewServle(){
        //ServletRegistrationBean 提供类的进行注册 .
        ServletRegistrationBean servletRegistrationBean
            = new ServletRegistrationBean(new StatViewServlet(),
            "/druid/*");
        // 添加初始化参数：initParams
        // 白名单：
        servletRegistrationBean.addInitParameter("allow", "127.0.0.1");
        //IP 黑名单 （存在共同时，deny 优先于 allow)
        // 如果满足 deny，就提示 :Sorry, you are not permitted to
        view this page.servletRegistrationBean.addInitParameter("deny",
        "192.168.1.73");
        // 登录查看信息的账号和密码
        servletRegistrationBean.addInitParameter("loginUsername","admin");
        servletRegistrationBean.addInitParameter("loginPassword", "123456");
```

```
        //是否能够重置数据
        servletRegistrationBean.addInitParameter("resetEnable", "false");
        return servletRegistrationBean;
    }

    @Bean
    public FilterRegistrationBean druidStatFilter(){
        FilterRegistrationBean filterRegistrationBean
            = new FilterRegistrationBean(new WebStatFilter());
        //添加过滤规则
        filterRegistrationBean.addUrlPatterns("/*");
        //添加需要忽略的格式信息
        filterRegistrationBean.addInitParameter("exclusions",
            "*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*");
        return filterRegistrationBean;
    }
}
```

- @Configuration：Spring 中有很多 XML 配置文件，文件中会配置很多 bean。在类上添加 @Configuration 注解，大家可以理解为该类变成一个 XML 配置文件。
- @Bean：等同于 XML 配置文件中的 <bean> 配置。Spring Boot 会把加上该注解方法的返回值装载进 Spring IoC 容器，方法的名称对应 <bean> 标签的 id 属性值。具体代码如下：

```
@Bean
public FilterRegistrationBean druidStatFilter(){
       FilterRegistrationBean filterRegistrationBean
           = new FilterRegistrationBean(new WebStatFilter());
      return filterRegistrationBean;
}
```

等同于：

```
<bean id="druidStatFilter" class=
    "org.springframework.boot.web.servlet.ServletRegistrationBean">
</bean>
```

类 ServletRegistrationBean 和 FilterRegistrationBean：在 DruidConfiguration.java 配置文件中，我们配置了两个类，即 druidStatViewServlet 和 druidStatFilter，并且通过注册类 ServletRegistrationBean 和 FilterRegistrationBean 实现 Servlet 和 Filter 类的注册。

在 druidStatViewServlet 类中，我们设定了访问数据库的白名单、黑名单、登录用户名和密码等信息。在 druidStatFilter 类中，我们设定了过滤的规则和需要忽略的格式。至此，配置类开发完成。

2.4.5 测试

在 DruidConfiguration.java 类开发完成之后，重新启动一下项目，然后通过访问网址 http://localhost:8080/druid/index.html 打开监控的登录界面，如图 2-10 所示。在登录界面中输入用户名：admin 和密码：123456 即可登录成功，如图 2-11 所示。

图 2-10　Druid 监控登录界面

图 2-11　Druid 登录成功界面

在 Druid 的监控界面中，我们可以对数据源、SQL、Web 应用等进行监控。

第3章

集成 Spring Data JPA

本章主要介绍 Spring Data JPA 核心接口及继承关系、在 Spring Boot 中集成 Spring Data JPA 以及如何通过 Spring Data JPA 实现增删改查及自定义查询等。

3.1 Spring Data JPA 介绍

本节主要介绍 Spring Data JPA 是什么、Spring Data JPA 核心接口 Repository、核心接口间的继承关系图。

3.1.1 Spring Data JPA 介绍

JPA（Java Persistence API）是 Sun 官方提出的 Java 持久化规范。所谓规范，即只定义标准规则，不提供实现。而 JPA 的主要实现有 Hibernate、EclipseLink、OpenJPA 等。

JPA 是一套规范，不是一套产品。Hibernate 是一套产品，如果这些产品实现了 JPA 规范，那么我们可以称其为 JPA 的实现产品。

Spring Data JPA 是 Spring Data 的一个子项目，通过提供基于 JPA 的 Respository 极大地减少了 JPA 作为数据访问方案的代码量。通过 Spring Data JPA 框架，开发者可以省略实现持久层业务逻辑的工作，唯一要做的就是声明持久层的接口，其他都交给 Spring Data JPA 来完成。

3.1.2 核心接口 Repository

Spring Data JPA 最顶层的接口是 Repository，该接口是所有 Repository 类的父类。具体代码如下：

```
package org.springframework.data.repository;
import java.io.Serializable;
public interface Repository<T, ID extends Serializable> {

}
```

Repository 类下没有任何接口，只是一个空类。Repository 接口的子类有 CrudRepository、PagingAndSortingRepository、JpaRepository 等。其中，CrudRepository 类提供了基本的增删改查等接口，PagingAndSortingRepository 类提供了基本的分页和排序等接口，而 JpaRepository 是 CrudRepository 和 PagingAndSortingRepository 的子类，继承了它们的所有接口。所以在真实的项目当中，我们都是通过实现 JpaRepository 或者其子类进行基本的数据库操作。JpaRepository 的具体代码如下：

```
@NoRepositoryBean
public interface JpaRepository extends PagingAndSortingRepository<T,
ID> {
    List<T> findAll();
    List<T> findAll(Sort var1);
    List<T> findAll(Iterable<ID> var1);
    <S extends T> List<S> save(Iterable<S> var1);
    void flush();
    <S extends T> S saveAndFlush(S var1);
    void deleteInBatch(Iterable<T> var1);
```

```
    void deleteAllInBatch();
    T getOne(ID var1);
    <S extends T> List<S> findAll(Example<S> var1);
    <S extends T> List<S> findAll(Example<S> var1, Sort var2);
}
```

- @NoRepositoryBean：使用该注解标明此接口不是一个 Repository Bean。

3.1.3 接口继承关系图

Repository 接口间的继承关系如图 3-1 所示。通过该继承图可以清楚地知道接口间的集成关系。在项目中，我们一般都是实现 JapRepository 类，加上自己定义的业务方法来完成业务开发。

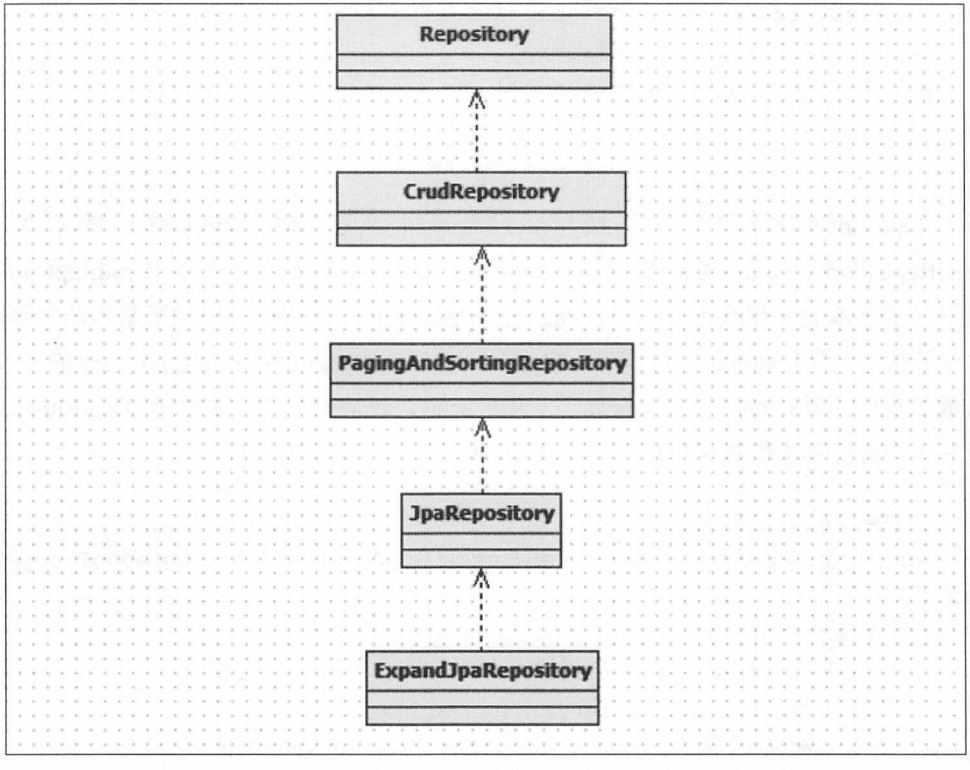


图 3-1 Repository 接口间的继承关系

3.2　集成 Spring Data JPA

本节主要介绍如何在 Spring Boot 中集成 Spring Data JPA、服务层类开发，如何通过 Spring Data JPA 实现基本的增删改查功能，以及自定义查询方法等内容。

3.2.1 引入依赖

在 Spring Boot 中集成 Spring Data JPA，首先需要在 pom.xml 文件中引入所需的依赖，具体代码如下：

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

在之前的章节中，我们已经在开发工具中安装好 Maven Helper 插件，所以大家可以通过该插件查看目前引入的所有依赖，如图 3-2 所示。

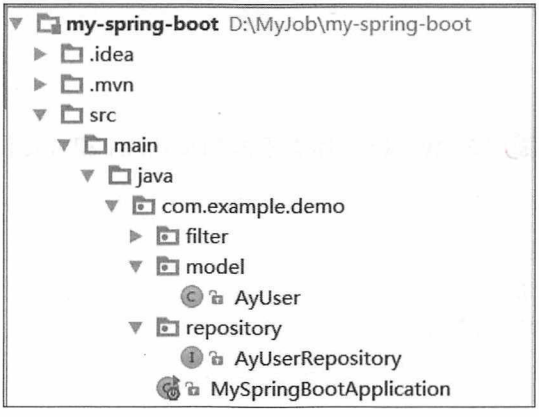


图 3-2　my-spring-boot 项目目录

3.2.2 继承 JpaRepository

在 pom.xml 文件中引入依赖之后，我们在目录 /src/main/java/com.example.demo.repository 下开发一个 AyUserRepository 类，如图 3-3 所示，具体代码如下：

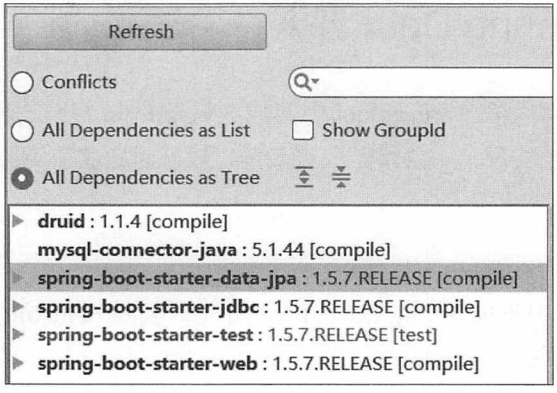


图 3-3 Maven Helper 查看 pom 依赖包

```
/**
 * 描述：用户 Repository
 * @author 阿毅
 * @date   2017/10/14.
 */
public interface AyUserRepository extends JpaRepository<AyUser,
String>{

}
```

与此同时，我们需要在 AyUser 实体类下添加 @Entity 和 @Id 注解，具体代码如下：

```
/**
 * 描述：用户表
 * @Author 阿毅
 * @date   2017/10/8.
 */
@Entity
@Table(name = "ay_user")
public class AyUser {
    //主键
    @Id
    private String id;
    //用户名
    private String name;
```

```
    //密码
    private String password;
}
```

- @Entity：每个持久化 POJO 类都是一个实体 Bean，通过在类的定义中使用 @Entity 注解来进行声明。
- @Table：声明此对象映射到数据库的数据表。该注释不是必需的，如果没有，系统就会使用默认值（实体的短类名）。
- @Id：指定表的主键。

3.2.3 服务层类实现

我们在 my-spring-boot 项目下继续开发服务层接口类和实现类：AyUserService 和 AyUserServiceImpl 类，分别存放在目录 /src/main/java/com.example.demo.service 和 /src/main/java/com.example.demo.service.impl 下。具体代码如下：

```
/**
 * 描述：用户服务层接口
 * @author 阿毅
 * @date   2017/10/14
 */
public interface AyUserService {
    AyUser findById(String id);
    List<AyUser> findAll();
    AyUser save(AyUser ayUser);
    void delete(String id);
}
```

接口类 AyUserService 定义了 4 个接口，findById 和 findAll 用来查询单个和所有数据，delete 用来删除数据，save 同时具备保存和更新数据的功能。接口实现类 AyUserServiceImpl 的代码如下：

```
/**
 * 描述：用户服务层实现类
 * @author 阿毅
 * @date   2017/10/14
 */
```

```
@Service
public class AyUserServiceImpl implements AyUserService{

    @Resource
    private AyUserRepository ayUserRepository;

    @Override
    public AyUser findById(String id){
        return ayUserRepository.findOne(id);
    }

    @Override
    public List<AyUser> findAll() {
        return ayUserRepository.findById(id).get();
    }

    @Override
    public AyUser save(AyUser ayUser) {
        return ayUserRepository.save(ayUser);
    }

    @Override
    public void delete(String id) {
        ayUserRepository.deleteById(id);
    }
}
```

- @ Service：Spring Boot 会自动扫描到 @Component 注解的类，并把这些类纳入 Spring 容器中管理。也可以用 @Component 注解，只是 @Service 注解更能表明该类是服务层类。
- @Component：泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。
- @Repository：持久层组件，用于标注数据访问组件，即 DAO 组件 。
- @Resource：这个注解属于 J2EE，默认按照名称进行装配，名称可以通过 name 属性进行指定。如果没有指定 name 属性，当注解写在字段上时，就默认取字段名进行查找。如果注解写在 setter 方法上，就默认取属性名进行装

配。当找不到与名称匹配的 bean 时，才按照类型进行装配。但需要注意的是，name 属性一旦指定，就只会按照名称进行装配。具体代码如下：

```
@Resource(name = "ayUserRepository")
private AyUserRepository ayUserRepository;
```

- @Autowired：这个注解属于 Spring，默认按类型装配。默认情况下，要求依赖对象必须存在，如果要允许 null 值，那么可以设置它的 required 属性为 false，如 @Autowired(required=false)；如果想使用名称装配，那么可以结合 @Qualifier 注解使用。具体代码如下：

```
@Autowired
@Qualifier("ayUserRepository")
private AyUserRepository ayUserRepository;
```

3.2.4 增删改查分页简单实现

前面已经在服务层类 AyUserService 开发完增删改查方法，这一节将继续在类中添加分页接口，具体代码如下：

```
/**
 * 描述：用户服务层接口
 * @author 阿毅
 * @date   2017/10/14
 */
public interface AyUserService {
    AyUser findById(String id);
    List<AyUser> findAll();
    AyUser save(AyUser ayUser);
    void delete(String id);
    // 分页
    Page<AyUser> findAll(Pageable pageable);
}
```

- Pageable：这是一个分页接口，查询时只需要传入一个 Pageable 接口的实现类，指定 pageNumber 和 pageSize 即可。pageNumber 为第几页，而 pageSize 为每页的大小。

- Page：分页查询结果会封装在该类中，Page 接口实现 Slice 接口，通过查看其源代码可知。我们通过调用 getTotalPages 和 getContent 等方法可以很方便地获得总页数和查询的记录。Page 接口和 Slice 接口源代码如下：

```
public interface Page<T> extends Slice<T> {
    int getTotalPages();
    long getTotalElements();
    <S> Page<S> map(Converter<? super T, ? extends S> var1);
}

public interface Slice<T> extends Iterable<T> {
    int getNumber();
    int getSize();
    int getNumberOfElements();
    List<T> getContent();
    boolean hasContent();
    Sort getSort();
    boolean isFirst();
    boolean isLast();
    boolean hasNext();
    boolean hasPrevious();
    Pageable nextPageable();
    Pageable previousPageable();
    <S> Slice<S> map(Converter<? super T, ? extends S> var1);
}
```

分页方法定义好之后，在类 AyUserServiceImpl 中实现该方法，具体代码如下：

```
@Override
public Page<AyUser> findAll(Pageable pageable) {
    return ayUserRepository.findAll(pageable);
}
```

3.2.5 自定义查询方法

我们除了使用 JpaRepository 接口提供的增删改查分页等方法之外，还可以自定义查询方法。在 AyUserRepository 类中添加几个自定义查询方法，具体代码如下：

```
/**
 * 描述：用户 Repository
 * @author 阿毅
 * @date   2017/10/14.
 */
public interface AyUserRepository extends JpaRepository<AyUser,
String>{

    /**
     * 描述：通过名字相等查询，参数为 name
     * 相当于：select u from ay_user u where u.name = ?1
     */
    List<AyUser> findByName(String name);

    /**
     * 描述：通过名字 like 查询，参数为 name
     * 相当于：select u from ay_user u where u.name like ?1
     */
    List<AyUser> findByNameLike(String name);

    /**
     * 描述：通过主键 id 集合查询，参数为 id 集合
     * 相当于：select u from ay_user u where id in(?,?,?)
     * @param ids
     */
    List<AyUser> findByIdIn(Collection<String> ids);
}
```

在 AyUserRepository 中自定义了 3 个查询的方法。从代码可以看出，Spring Data JPA 为我们约定了一系列规范，只要按照规范编写代码，Spring Data JPA 就会根据代码翻译成相关的 SQL 语句，进行数据库查询。比如可以使用 findBy、Like、In 等关键字，其中 findBy 可以用 read、readBy、query、queryBy、get、getBy 来代替。关于查询关键字的更多内容，大家可以到官方网站（https://docs.spring.io/spring-data/data-jpa/docs/current/reference/html/）查看，里面有详细的内容介绍，这里就不一一列举了。

AyUserRepository 类中的自定义查询方法开发完成之后，分别在类 AyUserService 和类 AyUserServiceImpl 中调用它们。

AyUserService 继续添加这 3 个方法，具体代码如下：

```
List<AyUser> findByName(String name);
List<AyUser> findByNameLike(String name);
List<AyUser> findByIdIn(Collection<String> ids);
```

AyUserServiceImpl 类添加这 3 个方法，具体代码如下：

```
@Override
public List<AyUser> findByName(String name){
    return ayUserRepository.findByName(name);
}
@Override
public List<AyUser> findByNameLike(String name){
    return ayUserRepository.findByNameLike(name);
}
@Override
public List<AyUser> findByIdIn(Collection<String> ids){
    return ayUserRepository.findByIdIn(ids);
}
```

@Override 注解不可以去掉，它可以帮助我们校验接口方法是否被误改。

3.3 集成测试

3.3.1 测试用例开发

在测试类 MySpringBootApplicationTests 中添加如下代码：

```
@Resource
private AyUserService ayUserService;

@Test
public void testRepository(){
    // 查询所有数据
```

```
        List<AyUser> userList =  ayUserService.findAll();
        System.out.println("findAll() :" + userList.size());
        // 通过 name 查询数据
        List<AyUser> userList2 = ayUserService.findByName("阿毅");
        System.out.println("findByName() :" + userList2.size());
        Assert.isTrue(userList2.get(0).getName().equals("阿毅"),
        "data error!");
        // 通过 name 模糊查询数据
        List<AyUser> userList3 = ayUserService.findByNameLike("%毅%");
        System.out.println("findByNameLike() :" + userList3.size());
        Assert.isTrue(userList3.get(0).getName().equals("阿毅"),
        "data error!");
        // 通过 id 列表查询数据
        List<String> ids = new ArrayList<String>();
        ids.add("1");
        ids.add("2");
        List<AyUser> userList4 = ayUserService.findByIdIn(ids);
        System.out.println("findByIdIn() :" + userList4.size());
        // 分页查询数据
        PageRequest pageRequest = new PageRequest(0,10);
        Page<AyUser> userList5 =  ayUserService.findAll(pageRequest);
        System.out.println("page findAll():" + userList5.getTotalPages() +
        "/" + userList5.getSize());
        // 新增数据
        AyUser ayUser = new AyUser();
        ayUser.setId("3");
        ayUser.setName("test");
        ayUser.setPassword("123");
        ayUserService.save(ayUser);
        // 删除数据
        ayUserService.delete("3");
    }
```

- Assert：添加 Assert 断言在软件开发中是一种常用的调试方式。从理论上来说，通过 Assert 断言方式可以证明程序的正确性，现在在项目中被广泛使用，这是大家需要掌握的基本知识。Assert 提供了很多好用的方法，比如 isNull、isTrue 等。

3.3.2 测试

通过运行 3.3.1 节中的单元测试用例，我们可以在控制台看到如下打印信息：

```
findAll() :2
findByName() :1
findByNameLike() :1
findByIdIn() :2
page findAll():1/10
```

通过上面的打印信息可以看出，Spring Boot 集成 Spring Data JPA 已经成功，同时代码中的所有 Assert 断言全部通过，说明增删改查分页以及自定义查询方法都可以正常运行。

第4章

使用 Thymeleaf 模板引擎

本章主要介绍 Thymeleaf 模板引擎、Thymeleaf 模板引擎标签和函数、Spring Boot 中如何使用 Thymeleaf、集成测试以及 Rest Client 工具的使用。

4.1 Thymeleaf 模板引擎介绍

Thymeleaf 是一个优秀的、面向 Java 的 XML/XHTML/HTML5 页面模板，具有丰富的标签语言和函数。因此，在使用 Spring Boot 框架进行页面设计时，一般都会选择 Thymeleaf 模板。下面简单列举一下 Thymeleaf 常用的表达式、标签和函数。

常用表达式：

- ${...} 变量表达式。
- *{...} 选择表达式。
- #{...} 消息文字表达式。
- @{...} 链接 url 表达式。
- #maps 工具对象表达式。

常用标签：

- th:action 定义后台控制器路径。
- th:each 循环语句。
- th:field 表单字段绑定。
- th:href 定义超链接。
- th:id div 标签中的 ID 声明，类似 HTML 标签中的 ID 属性。
- th:if 条件判断语句。
- th:include 布局标签，替换内容到引入文件。
- th:fragment 布局标签，定义一个代码片段，方便其他地方引用。
- th:object 替换对象。
- th:src 图片类地址引入。
- th:text 显示文本。
- th:value 属性赋值。

常用函数：

- #dates 日期函数。
- #lists 列表函数。
- #arrays 数组函数。
- #strings 字符串函数。
- #numbers 数字函数。
- #calendars 日历函数。

- #objects 对象函数。
- #bools 逻辑函数。

关于 Thymeleaf 表达式、标签、函数等更多内容，大家可以到官方网站（http://www.thymeleaf.org/）参考学习，这里不过多描述。

4.2　使用 Thymeleaf 模板引擎

4.2.1　引入依赖

要使用 Thymeleaf模板引擎，我们需要在 pom.xml 文件中引入依赖。依赖引入之后，记得刷新依赖。具体代码如下：

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

同时，我们还需要在 application.properties 文件中添加 Thymeleaf 配置，具体代码如下：

```
#thymeleaf 配置
# 模板的模式，支持 HTML、XML、TEXT、JAVASCRIPT 等
spring.thymeleaf.mode=HTML5
# 编码，可不用配置
spring.thymeleaf.encoding=UTF-8
# 内容类别，可不用配置
spring.thymeleaf.content-type=text/html
# 开发配置为 false，避免修改模板还要重启服务器
spring.thymeleaf.cache=false
# 配置模板路径，默认是 templates，可以不用配置
#spring.thymeleaf.prefix=classpath:/templates/
```

这里要注意的是，Thymeleaf 模板引擎默认会读取 my-spring-boot 项目资源文件夹 resource 下的 templates 目录，这个目录是用来存放 HTML 文件的。如果我们添加

了 Thymeleaf 依赖，而没有进行任何配置，或者添加默认目录，启动应用时就会报错。

4.2.2 控制层开发

在 my-spring-boot 项目目录 /src/main/java/com.example.demo.controller 下开发控制层类 AyUserController.java，同时把 AyUserService 服务注入控制层类当中。具体代码如下：

```
@Controller
@RequestMapping("/ayUser")
public class AyUserController {

    @Resource
    private AyUserService ayUserService;

    @RequestMapping("/test")
    public String test(Model model) {
       // 查询数据库所有用户
        List<AyUser> ayUser = ayUserService.findAll();
        model.addAttribute("users",ayUser);
        return "ayUser";
    }
}
```

- @Controller：标注此类为一个控制层类，同时让 Spring Boot 容器管理起来。
- @RequestMapping：是一个用来处理请求地址映射的注解，可用于类或者方法。用于类，表示类中所有响应请求的方法都是以该地址作为父路径的。@RequestMapping 注解有 value、method 等属性，value 属性可以默认不写。“/ayUser”就是 value 属性的值。value 属性的值就是请求的实际地址。
- Model 对象：一个接口，我们可以把数据库查询出来的数据设置到该类中，前端会从该对象获取数据。其实现类为 ExtendedModelMap，具体可看源代码：

```
public class ExtendedModelMap extends ModelMap implements Model {
    public ExtendedModelMap() {
    }
}
```

4.2.3 Thymeleaf 模板页面开发

控制层类 AyUserController.java 开发完成之后，我们继续在 /src/main/resources/templates 目录下开发 ayUser.html 页面，具体代码如下：

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>hello</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
</head>
<body>
<table>
    <tr>
        <td> 用户名 </td>
        <td> 密码 </td>
    </tr>
    <tr th:each="user:${users}">
        <td th:text="${user.name}"></td>
        <td th:text="${user.password}"></td>
    </tr>
</table>
</body>
</html>
```

<html xmlns:th="http://www.thymeleaf.org"> 是 Thymeleaf 命名空间，通过引入该命名空间就可以在 HTML 文件中使用 Thymeleaf 标签语言，用关键字“th”来标注。下面看几个简单的例子：

```
//th:text 用于显示文本 Hello、Thymeleaf
<p th:text=" 'Hello,Thymeleaf' "></p>
//${} 关键字用于获取内存变量为 name 的值
<p th:text="${name}"></p>
//th:src 用于设定 <img> 图片文件的链接地址，@{} 为超链接 url 表达式
<img th:src="@{/image/a.jpg}"/>
```

4.3 集成测试

4.3.1 测试

在 4.2 节中，我们已经简单开发好控制层类和前端 HTML 页面。现在重新运行入口类 MySpringBootApplication 的 main 方法，然后在浏览器中访问 http://localhost:8080/ayUser/test，出现如图 4-1 所示的页面，说明 Spring Boot 集成 Thymeleaf 成功，同时也说明我们开发的前端页面没有问题。

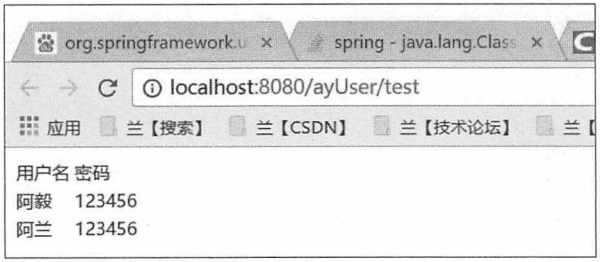


图 4-1 Spring Boot 集成 Thymeleaf 测试

4.3.2 Rest Client 工具介绍

Rest Client 是一个用于测试 RESTful Web Service 的 Java 客户端，非常小巧，界面非常简单。IntelliJ IDEA 软件已经集成该插件，方便我们进行调试。我们可以在 IntelliJ IDEA 功能菜单中选择【Tools】→【Test RESTful web service】来打开插件，具体如图 4-2 和图 4-3 所示。

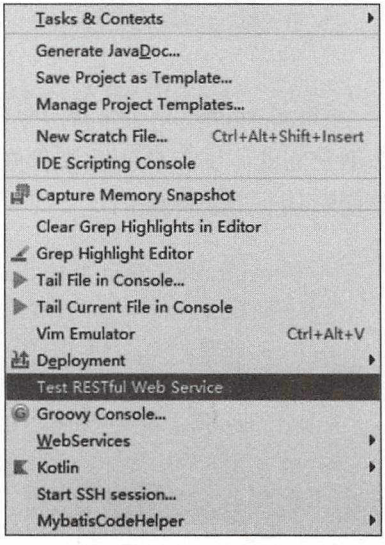


图 4-2 进入 Rest Client 选项

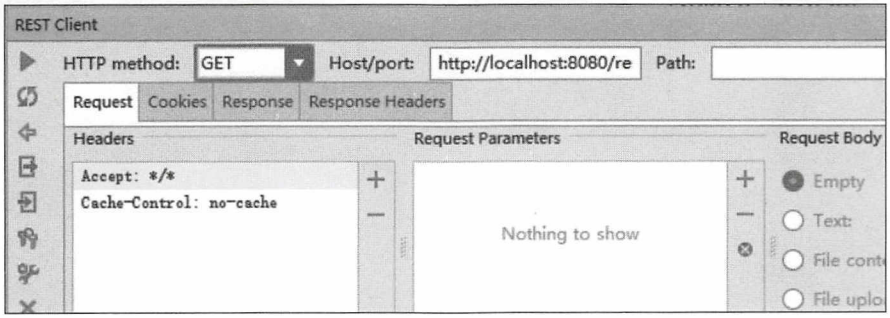


图 4-3　Rest Client 展示页面

在 Rest Client 插件中，我们可以在 HTTP method 中选择请求方式：Post 方式或者 Get 方式等。在 Host/port：中填入需要访问的主机 host 和端口 port，在 Path 中填入访问的映射路径，最后单击右上角的 run 按钮，就可以访问后端代码。同时，可以在 Request、Cookies、Response、Response Headers 中查看请求信息、Cookies 信息、响应信息、请求头信息等。

4.3.3　使用 Rest Client 测试

接下来，我们使用 Rest Clinet 介绍控制层类代码。在 Rest Client 界面中，在 HTTP method 中选择 Get 方式，在 Host/port 中填入 http://localhost:8080，在 Path 中填入 ayUser/test，具体如图 4-4 所示。然后单击 run 按钮运行之后，当我们看到如图 4-5 所示的 Response 响应界面时，说明代码执行成功。

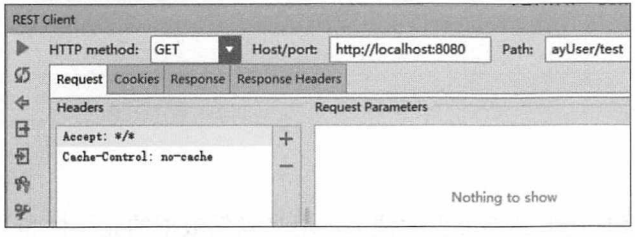


图 4-4　Rest Client 测试界面

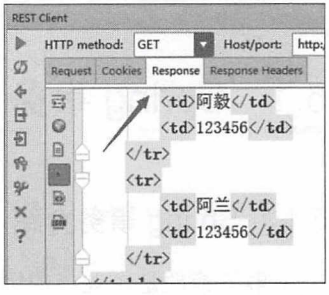


图 4-5　Response 响应页面

第5章

Spring Boot 事务支持

本章主要介绍 Spring 声明式事务、Spring 注解事务行为以及在 Spring Boot 中如何使用方法级别事务和类级别事务等。

5.1 Spring 事务

5.1.1 Spring 事务介绍

事务管理是企业级应用程序开发中必不可少的技术，用来确保数据的完整性和一致性。事务有四大特性（ACID）：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）。作为企业级应用程序框架，Spring 在不同的事务管理 API 上定义了一个抽象层，而应用程序开发人员不必了解底层的事务管理 API，就可以使用 Spring 的事务管理机制。

Spring 既支持编程式事务管理（也称编码式事务），又支持声明式事务管理。编程式事务管理是指将事务管理代码嵌入业务方法中来控制事务的提交和回滚。在编程式事务中，必须在每个业务操作中包含额外的事务管理代码。声明式事务管理是指将事务管理代码从业务方法中分离出来，以声明的方式来实现事务管理。在大多数情况下，声明式事务管理比编程式事务管理更好用。Spring 通过 Spring AOP 框架支持声明式事务管理。

数据访问的技术很多，如 JDBC、JPA、Hibernate、分布式事务等。面对众多的数据访问技术，Spring 在不同的事务管理 API 上定义了一个抽象层 PlatformTransactionManager，应用程序开发人员不必了解底层的事务管理 API 就可以使用 Spring 的事务管理机制。

Spring 并不直接管理事务，而是提供了许多内置事务管理器实现，常用的有 DataSourceTransactionManager、JdoTransactionManager、JpaTransactionManager 以及 HibernateTransactionManager 等。

5.1.2 Spring 声名式事务

Spring 配置文件中关于事务配置总是由三部分组成，分别是 DataSource、TransactionManager 和代理机制。无论哪种配置方式，一般变化的只是代理机制部分，DataSource 和 TransactionManager 这两部分只会根据数据访问方式有所变化，比如使用 Hibernate 进行数据访问时，DataSource 实现为 SessionFactory，TransactionManager 的实现为 HibernateTransactionManager。

Spring 声明式事务配置提供 5 种方式，而基于 Annotation 的注解方式目前比较流行，所以这里只简单介绍基于注解方式配置 Spring 声明式事务。我们可以使用 @Transactional 注解在类或者方法中表明该类或者方法需要事务支持，被注解的类或者方法被调用时，Spring 开启一个新的事务，当方法正常运行时，Spring 会提交这个事务。具体例子如下：

```
@Transactional
public AyUser updateUser() {
    // 执行数据库操作
}
```

这里需要注意的是，@Transactional 注解来自 org.springframework.transaction.annotation。Spring 提供了 @EnableTransactionManagement 注解在配置类上来开启声明式事务的支持。使用 @EnableTransactionManagement 后，Spring 容器会自动扫描注解 @Transactional 的方法和类。

5.1.3 Spring 注解事务行为

当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如，方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。事务的传播行为可以在 @Transactional 的属性中指定，Spring 定义了 7 种传播行为，具体如表 5-1 所示。

表5-1 Spring传播行为

传播行为	含义
PROPAGATION_REQUIRED	如果当前没有事务，就新建一个事务；如果已经存在一个事务，就加入这个事务中
PROPAGATION_SUPPORTS	支持当前事务，如果当前没有事务，就以非事务方式执行
PROPAGATION_MANDATORY	使用当前的事务，如果当前没有事务，就抛出异常
PROPAGATION_REQUIRES_NEW	新建事务，如果当前存在事务，就把当前事务挂起
PROPAGATION_NOT_SUPPORTED	以非事务方式执行操作，如果当前存在事务，就把当前事务挂起
PROPAGATION_NEVER	以非事务方式执行，如果当前存在事务，就抛出异常
PROPAGATION_NESTED	如果当前存在事务，就在嵌套事务内执行；如果当前没有事务，就执行与PROPAGATION_REQUIRED类似的操作

隔离级别定义了一个事务可能受其他并发事务影响的程度。在典型的应用程序中，多个事务并发运行经常会操作相同的数据来完成各自的任务。并发虽然是必需的，可是会导致许多问题，并发事务所导致的问题可以分为以下三类。

（1）脏读（Dirty Read）：脏读发生在一个事务读取了另一个事务改写但尚未提交的数据。如果改写在稍后被回滚了，那么第一个事务获取的数据就是无效的。

（2）不可重复读（Nonrepeatable Read）：不可重复读发生在一个事务执行相同的查询两次或两次以上，但是每次都得到不同的数据时。通常是因为另一个并发事务在两次查询期间更新了数据。

（3）幻读（Phantom Read）：幻读与不可重复读类似，发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录。

针对这些问题，Spring 提供了 5 种事务的隔离级别，具体如表 5-2 所示。

表5-2 Spring隔离级别

隔离级别	含义
ISOLATION_DEFAULT	使用数据库默认的事务隔离级别，另外4个与JDBC的隔离级别相对应
ISOLATION_READ_UNCOMMITTED	事务最低的隔离级别，允许另一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读、不可重复读和幻读
ISOLATION_READ_COMMITTED	使用当前的事务，如果当前没有事务，就抛出异常
ISOLATION_REPEATABLE_READ	新建事务，如果当前存在事务，就把当前事务挂起
ISOLATION_SERIALIZABLE	以非事务方式执行操作，如果当前存在事务，就把当前事务挂起

@Transactional 可以通过 propagation 属性定义事务行为，属性值分别为 REQUIRED、SUPPORTS、MANDATORY、REQUIRES_NEW、NOT_SUPPORTED、NEVER 以及 NESTED，分别对应表 5-1 中的内容。可以通过 isolation 属性定义隔离级别，属性值分别为 DEFAULT、READ_UNCOMMITTED、READ_COMMITTED、REPEATABLE_READ 以及 SERIALIZABLE。

还可以通过 timeout 属性设置事务过期时间，通过 readOnly 指定当前事务是否是只读事务，通过 RollbackFor（noRollbackFor）指定哪个或者哪些异常可以引起（或不可以引起）事务回滚。

5.2 Spring Boot 事务的使用

5.2.1 Spring Boot 事务介绍

Spring Boot 开启事务很简单，只需要一个注解 @Transactional 就可以了，因为在 Spring Boot 中已经默认对 JPA、JDBC、Mybatis 开启了事务，引入它们依赖的时候，

事物就默认开启。当然，如果你需要用其他的 ORM 框架，比如 BeatlSQL，就需要自己配置相关的事物管理器。

Spring Boot 用于配置事务的类为 TransactionAutoConfiguration，此配置类依赖于 JtaAutoConfiguration 和 DataSourceTransactionManagerAutoConfiguration，具体查看源代码可知，而 DataSourceTransactionManagerAutoConfiguration 开启了对声明式事务的支持，所以在 Spring Boot 中，无须显示开启使用 @EnableTransactionManagement。

5.2.2 类级别事务

在第 3 章中，我们已经在 Spring Boot 中集成了 Spring Data JPA，同时开发了 AyUserRepository 类实现 JpaRepository 接口，JpaRepository 接口是不开启事务的，而 SimpleJapRepository 默认是开启事务的，所以我们需要手工给 AyUserRepository 添加事务。AyUserRepository 类中的方法是在服务层类 AyUserServiceImpl 中被使用，而事务一般都是加在服务层，因此我们可以在 AyUserServiceImpl 类上添加 @Transactional 注解来开启事务。AyUserServiceImpl 类开启事务的代码如下：

```
/**
 * 描述：用户服务层实现类
 * @author 阿毅
 * @date   2017/10/14
 */
@Transactional
@Service
public class AyUserServiceImpl implements AyUserService {

    @Resource(name = "ayUserRepository")
    private AyUserRepository ayUserRepository;

    // 省略代码
}
```

@Transactional 注解在类上，意味着此类的所有 public 方法都是开启事务的。

5.2.3 方法级别事务

@Transactional 除了可以注解在类上，还可以注解到方法上面。当注解在类

上时，意味着此类的所有 public 方法都是开启事务的。如果类级别和方法级别同时使用了 @Transactional 注解，就使用方法级别注解覆盖类级别注解。我们可以给 AyUserServiceImpl 类中的 save() 方法添加事务，同时在 save 完成之后抛 NullPointException 异常，查看数据是否可以回滚，具体代码如下：

```
/**
 * 描述：用户服务层实现类
 * @author 阿毅
 * @date   2017/10/14
 */
//注解在类上
@Transactional
@Service
public class AyUserServiceImpl implements AyUserService {

    @Resource(name = "ayUserRepository")
    private AyUserRepository ayUserRepository;

    //注解在方法上
    @Transactional
    @Override
    public AyUser save(AyUser ayUser) {
        AyUser saveUser =  ayUserRepository.save(ayUser);
        //出现空指针异常
        String error = null;
        error.split("/");
        return saveUser;
    }

}
```

5.2.4 测试

5.2.1 节和 5.2.2 节的代码开发完成之后，我们在测试类 MySpringBootApplicationTests 中添加测试方法，具体代码如下：

```
@Test
public void testTransaction(){
    AyUser ayUser = new AyUser();
    ayUser.setId("3");
    ayUser.setName("阿华");
    ayUser.setPassword("123");
    ayUserService.save(ayUser);
}
```

运行 testTransaction() 单元测试用例，当代码执行完成后，由于方法 save 保持数据时出现空指针，数据会回滚，因此在数据库中查询不到保存的数据。现在把 AyUserServiceImpl 类上的 @ Transactional 注解和 save 方法上的 @ Transactional 注解全部注释掉，再次执行 testTransaction() 单元测试用例，查询数据库，发现数据库多了一条数据，如图 5-1 所示。

图 5-1 将数据插入数据库

第6章

使用过滤器和监听器

本章主要介绍如何在 Spring Boot 中使用过滤器 Filter 和监听器 Listener。

6.1 Spring Boot 使用过滤器 Filter

6.1.1 过滤器 Filter 介绍

Filter 也称为过滤器，是处于客户端与服务器资源文件之间的一道过滤网，它是 Servlet 技术中最激动人心的技术之一。Web 开发人员通过 Filter 技术管理 Web 服务器的所有资源，例如对 JSP、Servlet、静态图片文件或静态 HTML 文件等进行拦截，从而实现一些特殊的功能，如实现 URL 级别的权限访问控制、过滤敏感词汇、压缩响应信息等一些高级功能。

Filter 接口源代码如下：

```
public interface Filter {
    void init(FilterConfig var1) throws ServletException;
    void doFilter(ServletRequest var1, ServletResponse var2,
    FilterChain var3) throws IOException, ServletException;
    void destroy();
}
```

Filter 的创建和销毁由 Web 服务器负责。Web 应用程序启动时，Web 服务器将创建 Filter 的实例对象，并调用其 init 方法，读取 web.xml 配置，完成对象的初始化功能，从而为后续的用户请求做好拦截的准备工作（Filter 对象只会创建一次，init 方法也只会执行一次）。开发人员通过 init 方法的参数可获得代表当前 filter 配置信息的 FilterConfig 对象。

当客户请求访问与过滤器关联的 URL 时，过滤器将先执行 doFilter 方法。FilterChain 参数用于访问后续过滤器。Filter 对象创建后会驻留在内存中，当 Web 应用移除或服务器停止时才销毁。在 Web 容器卸载 Filter 对象之前，destroy 被调用。该方法在 Filter 的生命周期中仅执行一次。在这个方法中，可以释放过滤器使用的资源。

Filter 可以有很多个，一个个 Filter 组合起来就形成一个 FilterChain，也就是我们所说的过滤链，如图 6-1 所示。

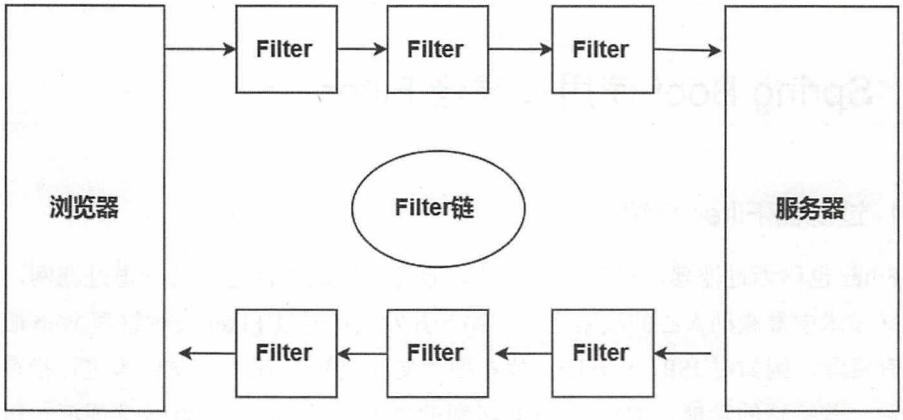


图 6-1 Filter 链

FilterChain 的执行顺序遵循先进后出的原则：当 Web 客户端发送一个 Request 请求时，这个 Request 请求会先经过 FilterChain，由它利用 dofilter() 方法调用各个子 Filter，至于子 filter 的执行顺序如何，则看客户端是如何制定规则的。当 Request 请求被第一个 Filter 处理之后，又通过 dofilte() 往下传送，被第二个、第三个……Filter 截获处理。当 Request 请求被所有的 Filter 处理之后，返回的顺序是从最后一个开始返回，直到返回给客户端。

6.1.2 过滤器 Filter 的使用

在 Spring Boot 中使用 Filter 过滤器很简单。首先在项目 my-spring-boot 的目录 /src/main/java/com.example.demo.filter 下新建 AyUserFilter.java 类，具体代码如下：

```
/**
 * 过滤器
 * @author Ay
 * @date   2017/11/2.
 */
@WebFilter(filterName = "ayUserFilter", urlPatterns = "/*")
public class AyUserFilter implements Filter{

    @Override
    public void init(FilterConfig filterConfig) throws
    ServletException {
        System.out.println("---------->>> init");
    }

    @Override
    public void doFilter(ServletRequest servletRequest,
        ServletResponse servletResponse,FilterChain filterChain)
        throws IOException, ServletException {
        System.out.println("---------->>> doFilter");
        filterChain.doFilter(servletRequest,servletResponse);
    }
```

```
    @Override
    public void destroy() {
        System.out.println("---------->>> destory");
    }
}
```

- @WebFilter：用于将一个类声明为过滤器，该注解将会在应用部署时被容器处理，容器根据具体的属性配置将相应的类部署为过滤器。这样我们在 Web 应用中使用监听器时，不需要在 web.xml 文件中配置监听器的相关描述信息。该注解的常用属性有 filterName、urlPatterns、value 等。filterName 属性用于指定过滤器的 name，等价于 XML 配置文件中的 <filter-name> 标签。urlPatterns 属性用于指定一组过滤器的 URL 匹配模式，等价于 XML 配置文件中的 <url-pattern> 标签。value 属性等价于 urlPatterns 属性，但是两者不可以同时使用。

AyUserFilter.java 类开发完成之后，我们需要在入口类 MySpringBootApplication.java 中添加注解 @ServletComponentScan，具体代码如下：

```
@SpringBootApplication
@ServletComponentScan
public class MySpringBootApplication {

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

- @ServletComponentScan：使用该注解后，Servlet、Filter、Listener 可以直接通过 @WebServlet、@WebFilter、@WebListener 注解自动注册，无须其他代码。

事实上，在 Spring Boot 中添加自己的 Servlet、Filter 和 Listener 有两种方法，即代码注册和注解自动注册。上面我们已经讲解了如何用注解自动注册，而代码注册可以通过 ServletRegistrationBean、FilterRegistrationBean 和 ServletListenerRegistrationBean 注册 Bean。虽然条条大路通罗马，但是希望大家先掌握一种方式，一路走到底，而不是纠结于会有几种写法。

6.1.3 测试

6.1.2 节的代码开发完成之后，重新启动运行 my-spring-boot 项目时，Web 容器会初始化 AyUserFilter 对象，并调用 init 方法，可以在 IntelliJ IDEA 控制台看到打印信息，如图 6-2 所示。在浏览器中输入 http://localhost:8080/ayUser/test 访问应用时，AyUserFilter 拦截器会拦截本次的请求，并调用 doFilter 方法，同时会在控制台打印信息，如图 6-3 所示。

```
2017-11-04 14:38:58.602  INFO 20744 --- [ost-startStop-1] o.s.b.w.ser
2017-11-04 14:38:58.607  INFO 20744 --- [ost-startStop-1] o.s.b.w.ser
2017-11-04 14:38:58.608  INFO 20744 --- [ost-startStop-1] o.s.b.w.ser
2017-11-04 14:38:58.608  INFO 20744 --- [ost-startStop-1] o.s.b.w.ser
2017-11-04 14:38:58.608  INFO 20744 --- [ost-startStop-1] o.s.b.w.ser
2017-11-04 14:38:58.608  INFO 20744 --- [ost-startStop-1] o.s.b.w.ser
---------->>> init
2017-11-04 14:38:58.864  INFO 20744 --- [           main] com.alibaba
```

图 6-2 初始化调用 init 方法打印信息

```
2017-11-04 14:39:01.782  INFO 20744 --- [           main] o.s.j.e.a.
2017-11-04 14:39:01.784  INFO 20744 --- [           main] o.s.j.e.a.
2017-11-04 14:39:01.789  INFO 20744 --- [           main] o.s.j.e.a.
2017-11-04 14:39:01.860  INFO 20744 --- [           main] s.b.c.e.t.
2017-11-04 14:39:01.864  INFO 20744 --- [           main] c.example.
2017-11-04 14:46:00.515  INFO 20744 --- [nio-8080-exec-1] o.a.c.c.C.
2017-11-04 14:46:00.515  INFO 20744 --- [nio-8080-exec-1] o.s.web.se
2017-11-04 14:46:00.535  INFO 20744 --- [nio-8080-exec-1] o.s.web.se
---------->>> doFilter
2017-11-04 14:46:03.255  INFO 20744 --- [nio-8080-exec-1] o.h.h.i.Qu
```

图 6-3 拦截调用 doFilter 方法打印信息

6.2 Spring Boot 使用监听器 Listener

6.2.1 监听器 Listener 介绍

监听器也叫 Listener，是 Servlet 的监听器，可以用于监听 Web 应用中某些对象、信息的创建、销毁、增加、修改、删除等动作的发生，然后做出相应的响应处理。当范围对象的状态发生变化时，服务器自动调用监听器对象中的方法，常用于统计在线人数和在线用户，系统加载时进行信息初始化、统计网站的访问量等。

根据监听对象可以把监听器分为 3 类：ServletContext（对应 application）、HttpSession（对应 session）、ServletRequest（对应 request）。Application 在整个 Web 服务中只有一个，在 Web 服务关闭时销毁。Session 对应每个会话，在会话起始时创建，一端关闭会话时销毁。Request 对象是客户发送请求时创建的（一同创建的还有 Response），用于封装请求数据，在一次请求处理完毕时销毁。

根据监听的事件，可把监听器分为以下 3 类。

（1）监听对象创建与销毁，如 ServletContextListener。

（2）监听对象域中属性的增加和删除，如 HttpSessionListener 和 ServletRequestListener。

（3）监听绑定到 Session 上的某个对象的状态，如 ServletContextAttributeListener、HttpSessionAttributeListener、ServletRequestAttributeListener 等。

6.2.2 监听器 Listener 的使用

在 Spring Boot 中使用 Listener 监听器和 Filter 基本一样。首先我们在项目 my-spring-boot 的目录 /src/main/java/com.example.demo.listener 下新建 AyUserListener.java 类，具体代码如下：

```
/**
 * 描述：监听器
 * @author Ay
 * @date   2017/11/4
 */
@WebListener
public class AyUserListener implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent
    servletContextEvent) {
        System.out.println("ServletContext 上下文初始化 ");
    }

    @Override
    public void contextDestroyed(ServletContextEvent
    servletContextEvent) {
        System.out.println("ServletContext 上下文销毁 ");
    }
}
```

- @WebListener：用于将一个类声明为监听器，该注解将会在应用部署时被容器处理，容器根据具体的属性配置将相应的类部署为监听器。这样我们在 Web 应用中使用监听器时，不需要在 web.xml 文件中配置监听器的相关描述信息。

- ServletContextListener 类：能够监听 ServletContext 对象的生命周期，实际上就是监听 Web 应用的生命周期。当 Servlet 容器启动或终止 Web 应用时，会触发 ServletContextEvent 事件，该事件由 ServletContextListener 类来处理。在 ServletContextListener 接口中定义了处理 ServletContextEvent 事件的两个方法：contextInitialized 和 contextDestroyed。
  - contextInitialized：当 Servlet 容器启动 Web 应用时调用该方法。在调用完该方法之后，容器再对 Filter 初始化，并且对那些在 Web 应用启动时就需要被初始化的 Servlet 进行初始化。
  - contextDestroyed：当 Servlet 容器终止 Web 应用时调用该方法。在调用该方法之前，容器会先销毁所有的 Servlet 和 Filter 过滤器。

我们可以在 contextInitialized 方法中查询所有的用户，利用缓存技术把用户数据存放到缓存中。在第 7 章中会具体讲解如何利用监听器和 Redis 缓存技术来缓存用户数据，提高系统性能。

6.2.3 测试

6.2.2 节的代码开发完成之后，重新启动运行 my-spring-boot 项目时，Web 容器会初始化 AyUserListener 对象，并调用 contextInitialized 方法，可以在 IntelliJ IDEA 控制台看到打印信息，如图 6-4 所示。当我们销毁容器时，会调用 contextDestroyed 方法，并在控制台打印信息。这里需要注意的是，在 IDEA 开发工具中，直接终止容器或者关闭进程是不会执行销毁方法 contextDestroyed 的。

```
2017-11-04 18:40:41.875  INFO 23484 --- [ost-startStop-1] o.s.b.w.servlet.
2017-11-04 18:40:41.879  INFO 23484 --- [ost-startStop-1] o.s.b.w.servlet.
2017-11-04 18:40:41.880  INFO 23484 --- [ost-startStop-1] o.s.b.w.servlet.
2017-11-04 18:40:41.880  INFO 23484 --- [ost-startStop-1] o.s.b.w.servlet.
2017-11-04 18:40:41.880  INFO 23484 --- [ost-startStop-1] o.s.b.w.servlet.
2017-11-04 18:40:41.880  INFO 23484 --- [ost-startStop-1] o.s.b.w.servlet.
2017-11-04 18:40:41.880  INFO 23484 --- [ost-startStop-1] o.s.b.w.servlet.
ServletContext上下文初始化
```

图 6-4　容器启动监听打印信息

第7章

集成 Redis 缓存

本章主要介绍如何安装 Redis 缓存、Redis 缓存 5 种基本数据类型的增删改查、Spring Boot 中如何集成 Redis 缓存以及如何使用 Redis 缓存用户数据等。

7.1 Redis 缓存介绍

7.1.1 Redis 概述

Redis 是一个基于内存的单线程高性能 key-value 型数据库，读写性能优异。与 Memcached 缓存相比，Redis 支持丰富的数据类型，包括 string（字符串）、list（链表）、set（集合）、zset（sorted set 有序集合）和 hash（哈希类型）。因此，Redis 在企业中被广泛使用。

7.1.2 Redis 服务器的安装

Redis 项目本身不支持 Windows，但是 Microsoft 开放技术小组开发和维护这个 Windows 端口（针对 Win64），所以我们可以在网络上下载 Redis 的 Windows 版本。具体步骤如下：

步骤 01　打开官方网站（http://redis.io/），单击 Download，如图 7-1 所示。

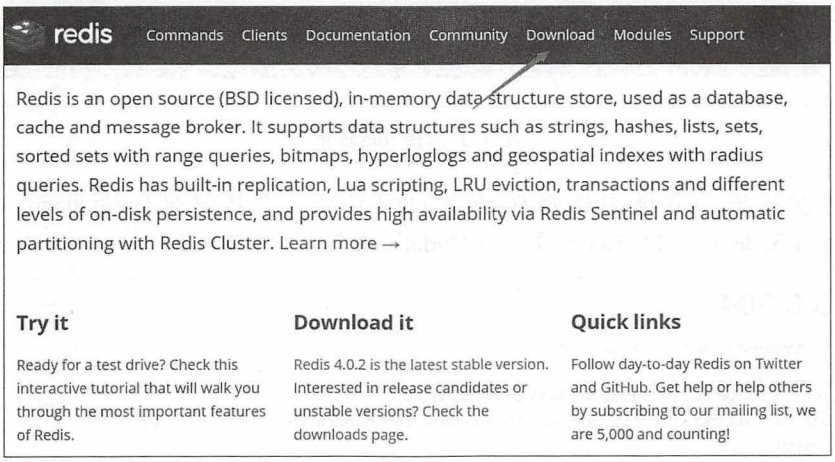


图 7-1　Redis 下载首页

步骤 02　在弹出的页面中找到 Learn more 链接，并单击进入，如图 7-2 所示。

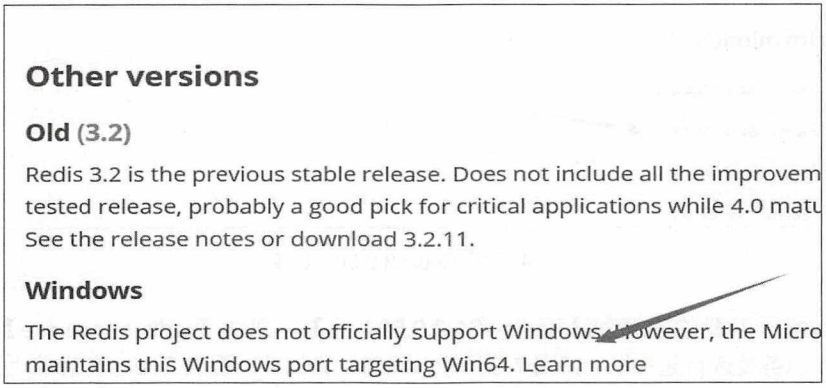


图 7-2　单击 Learn more 链接

步骤 03　在弹出的页面中选择【releases】选项，如图 7-3 所示。

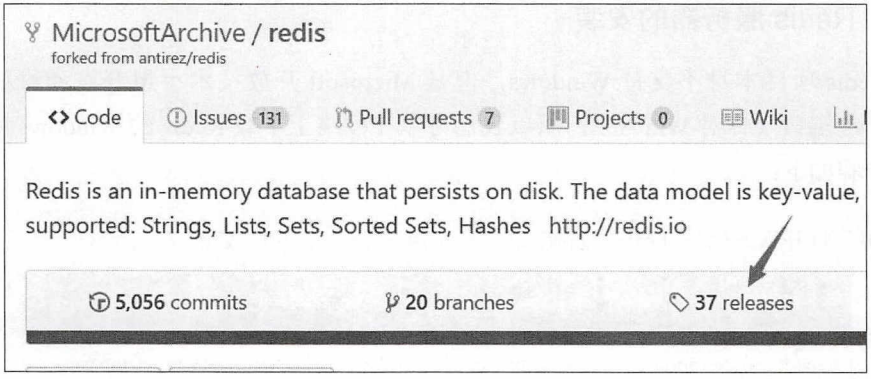


图 7-3 选择 releases

步骤 04 在弹出的界面中选择 Redis 3.0.504 版本，选择其他版本也可以，单击【Redis-x64-3.0.504.zip】下载 Redis 安装包，如图 7-4 所示。

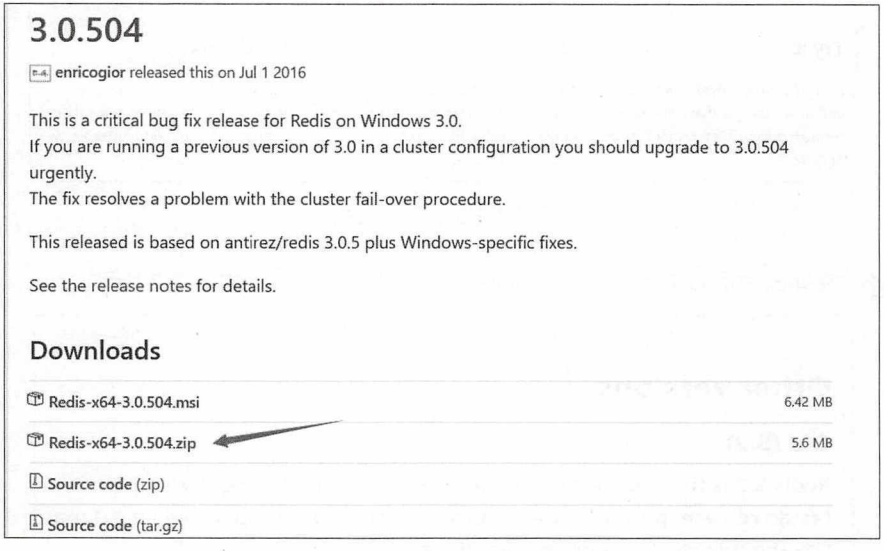
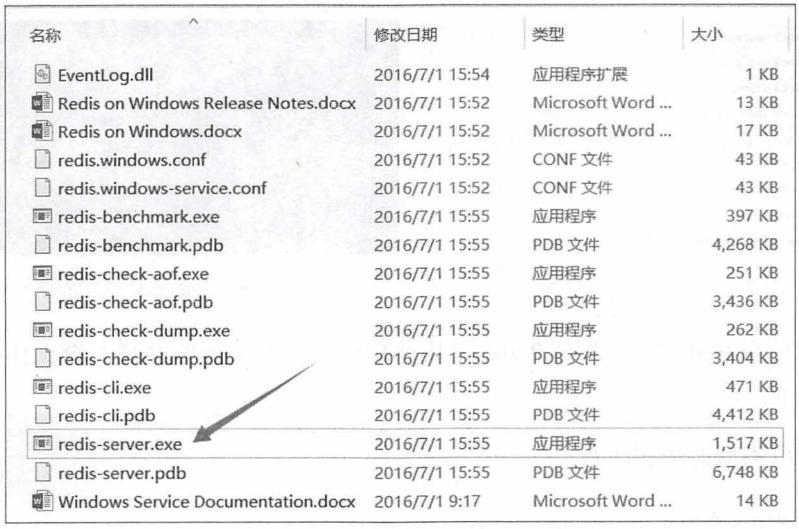


图 7-4 下载 Redis 3.0.504 安装包

步骤 05 解压下载的安装包【Redis-x64-3.0.504.zip】，双击【redis-server.exe】，Redis 服务就运行起来了，如图 7-5 所示。同时，我们可以看到 Redis 启动成功的界面，如图 7-6 所示。



名称	修改日期	类型	大小
EventLog.dll	2016/7/1 15:54	应用程序扩展	1 KB
Redis on Windows Release Notes.docx	2016/7/1 15:52	Microsoft Word ...	13 KB
Redis on Windows.docx	2016/7/1 15:52	Microsoft Word ...	17 KB
redis.windows.conf	2016/7/1 15:52	CONF 文件	43 KB
redis.windows-service.conf	2016/7/1 15:52	CONF 文件	43 KB
redis-benchmark.exe	2016/7/1 15:55	应用程序	397 KB
redis-benchmark.pdb	2016/7/1 15:55	PDB 文件	4,268 KB
redis-check-aof.exe	2016/7/1 15:55	应用程序	251 KB
redis-check-aof.pdb	2016/7/1 15:55	PDB 文件	3,436 KB
redis-check-dump.exe	2016/7/1 15:55	应用程序	262 KB
redis-check-dump.pdb	2016/7/1 15:55	PDB 文件	3,404 KB
redis-cli.exe	2016/7/1 15:55	应用程序	471 KB
redis-cli.pdb	2016/7/1 15:55	PDB 文件	4,412 KB
redis-server.exe	2016/7/1 15:55	应用程序	1,517 KB
redis-server.pdb	2016/7/1 15:55	PDB 文件	6,748 KB
Windows Service Documentation.docx	2016/7/1 9:17	Microsoft Word ...	14 KB

图 7-5　启动 redis 服务器

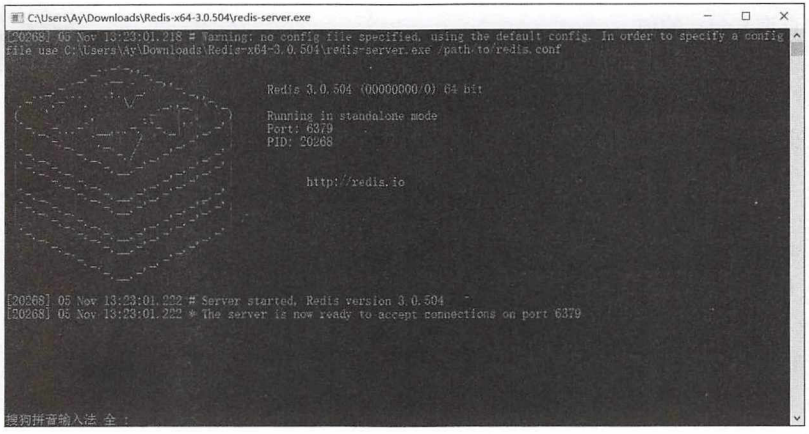


图 7-6　redis 启动成功界面

7.1.3　Redis 缓存测试

Redis 安装成功之后，我们可以在安装包里找到 Redis 客户端程序 redis-cli.exe，如图 7-7 所示。双击 redis-cli.exe，打开 Redis 客户端界面，如果 7-8 所示。

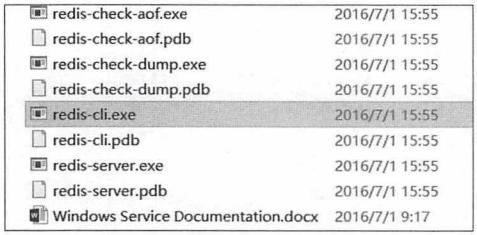


图 7-7 启动 redis 客户端

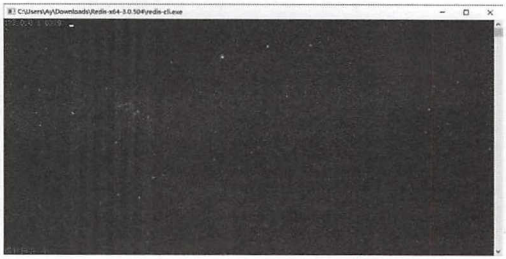


图 7-8 redis 启动成功界面

下面使用 Redis 客户端对 Redis 的几种数据类型做基本的增删改查操作练习，具体代码如下：

字符串类型的增删改查：

```
### 增加一个值 key 为 name，value 为 ay
127.0.0.1:6379> set name 'ay'
OK
### 查询 name 的值
127.0.0.1:6379> get name
"ay"
### 更新 name 的值为 al
127.0.0.1:6379> set name 'al'
OK
### 查询 name 的值
127.0.0.1:6379> get name
"al"
### 删除 name 的值
127.0.0.1:6379> del name
(integer) 1
### 查询是否存在 name，0 代表不存在
127.0.0.1:6379> exists name
(integer) 0
127.0.0.1:6379>
```

List 集合的增删改查：

```
### 添加 key 为 user_list，value 为 'ay','al' 的 list 集合
127.0.0.1:6379> lpush user_list 'ay' 'al'
```

```
(integer) 2
### 查询 key 为 user_list 的集合
127.0.0.1:6379> lrange user_list 0 -1
1) "al"
2) "ay"
### 往 list 尾部添加 love 元素
127.0.0.1:6379> rpush user_list 'love'
(integer) 3
### 往 list 头部添加 hope 元素
127.0.0.1:6379> lpush user_list 'hope'
(integer) 4
### 查询 key 为 user_list 的集合
127.0.0.1:6379> lrange user_list 0 -1
1) "hope"
2) "al"
3) "ay"
4) "love"
### 更新 index 为 0 的值
127.0.0.1:6379> lset user_list 0 'wish'
OK
### 查询 key 为 user_list 的集合
127.0.0.1:6379> lrange user_list 0 -1
1) "wish"
2) "al"
3) "ay"
4) "love"
### 删除 index 为 0 的值
127.0.0.1:6379> lrem user_list 0 'wish'
(integer) 1
### 查询 key 为 user_list 的集合
127.0.0.1:6379> lrange user_list 0 -1
1) "al"
2) "ay"
3) "love"
127.0.0.1:6379>
```

Set 集合的增删改查：

```
### 添加 key 为 user_set,value 为 "ay" "al"  "love" 的集合
127.0.0.1:6379> sadd user_set "ay" "al"  "love"
(integer) 3
### 查询 key 为 user_set 的集合
127.0.0.1:6379> smembers user_set
1) "al"
2) "ay"
3) "love"
### 删除 value 为 love，返回 1 表示删除成功，0 表示失败
127.0.0.1:6379> srem user_set 'love'
(integer) 1
### 查询 set 集合所有值
127.0.0.1:6379> smembers user_set
1) "al"
2) "ay"
### 添加 love 元素，set 集合是没有顺序的，所以无法判断添加到哪个位置
127.0.0.1:6379> sadd user_set 'love'
(integer) 1
### 查询 set 集合所有的值，发现添加到第二个位置
127.0.0.1:6379> smembers user_set
1) "al"
2) "love"
3) "ay"
### 添加 love 元素，set 集合已经存在，返回 0 代表添加不成功，但是不会报错
127.0.0.1:6379> sadd user_set 'love'
(integer) 0
```

Hash 集合的增删改查：

```
### 清除数据库
127.0.0.1:6379> flushdb
OK
### 创建 hash，key 为 user_hset，字段为 user1，值为 ay
127.0.0.1:6379> hset user_hset "user1"  "ay"
(integer) 1
### 往 key 为 user_hset 的哈希集合添加字段为 user2，值为 al
```

```
127.0.0.1:6379> hset user_hset "user2"  "al"
(integer) 1
### 查询 user_hset 字段长度
127.0.0.1:6379> hlen user_hset
(integer) 2
### 查询 user_hset 所有字段
127.0.0.1:6379> hkeys user_hset
1) "user1"
2) "user2"
### 查询 user_hset 所有值
127.0.0.1:6379> hvals user_hset
1) "ay"
2) "al"
### 查询字段 user1 的值
127.0.0.1:6379> hget user_hset "user1"
"ay"
### 获取 key 为 user_hset 的哈希集合的所有字段和值
127.0.0.1:6379> hgetall user_hset
1) "user1"
2) "ay"
3) "user2"
4) "al"
### 更新字段 user1 的值为 new_ay
127.0.0.1:6379> hset user_hset "user1" "new_ay"
(integer) 0
### 更新字段 user2 的值为 new_al
127.0.0.1:6379> hset user_hset "user2" "new_al"
(integer) 0
### 获取 key 为 user_hset 的哈希集合的所有字段和值
127.0.0.1:6379> hgetall user_hset
1) "user1"
2) "new_ay"
3) "user2"
4) "new_al"
### 删除字段 user1 和值
127.0.0.1:6379> hdel user_hset user1
(integer) 1
```

```
### 获取 key 为 user_hset 的哈希集合的所有字段和值
127.0.0.1:6379> hgetall user_hset
1) "user2"
2) "new_al"
127.0.0.1:6379>
```

SortedSet 集合的增删改查：

```
### 清除数据库
127.0.0.1:6379> flushdb
OK
### 为 SortedSet 集合添加 ay 元素，分数为 1
127.0.0.1:6379> zadd user_zset  1 "ay"
(integer) 1
### 为 SortedSet 集合添加 al 元素，分数为 2
127.0.0.1:6379> zadd user_zset  2 "al"
(integer) 1
### 为 SortedSet 集合添加 love 元素，分数为 3
127.0.0.1:6379> zadd user_zset  3 "love"
(integer) 1
### 按照分数由小到大查询 user_zset 集合的元素
127.0.0.1:6379> zrange user_zset 0 -1
1) "ay"
2) "al"
3) "love"
### 按照分数由大到小查询 user_zset 集合的元素
127.0.0.1:6379> zrevrange user_zset 0 -1
1) "love"
2) "al"
3) "ay"
### 查询元素 ay 的分数值
127.0.0.1:6379> zscore user_zset "ay"
"1"
### 查询元素 love 的分数值
127.0.0.1:6379> zscore user_zset "love"
"3"
```

7.2 Spring Boot 集成 Redis 缓存

7.2.1 Spring Boot 缓存支持

在 Spring Boot 中提供了强大的基于注解的缓存支持，可以通过注解配置方式低侵入地给原有 Spring 应用增加缓存功能，提高数据访问性能。Spring Boot 配置了多个 CacheManager 的实现，我们可以根据具体的项目要求使用相应的缓存技术，如图 7-9 所示。

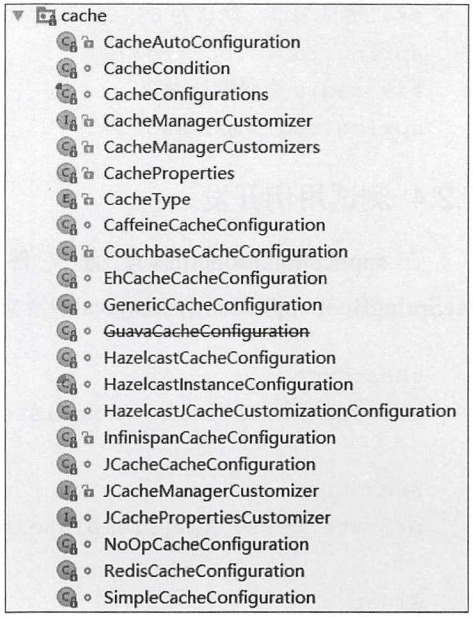


图 7-9 Spring Boot 缓存配置类

从图 7-9 可知，Spring Boot 支持许多类型的缓存，比如 EhCache、JCache、Redis 等。在不添加任何额外配置的情况下，Spring Boot 默认使用 SimpleCacheConfiguration，考虑到 Redis 缓存在企业中被广泛使用，故选择用 Redis 缓存来进行讲解。

7.2.2 引入依赖

在 Spring Boot 中集成 Redis，首先需要在 pom.xml 文件中引入所需的依赖，具体代码如下：

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

7.2.3 添加缓存配置

在 pom 文件中引入 Redis 所需的依赖之后，我们需要在 application.properties 文件中添加如下配置信息：

```
### redis 缓存配置
### 默认 redis 数据库为 db0
spring.redis.database=0
### 服务器地址，默认为 localhost
spring.redis.host=localhost
### 链接端口，默认为 6379
spring.redis.port=6379
### redis 密码默认为空
spring.redis.password=
```

7.2.4 测试用例开发

在 application.properties 配置文件中添加完 Redis 配置之后，在测试类 MySpringBootApplicationTests.java 中继续添加如下代码：

```
@Resource
private RedisTemplate redisTemplate;

@Resource
private StringRedisTemplate stringRedisTemplate;

@Test
public void testRedis(){
    //增 key: name, value: ay
    redisTemplate.opsForValue().set("name","ay");
    String name = (String)redisTemplate.opsForValue().get("name");
    System.out.println(name);
    //删除
    redisTemplate.delete("name");
    //更新
    redisTemplate.opsForValue().set("name","al");
    //查询
    name = stringRedisTemplate.opsForValue().get("name");
    System.out.println(name);
}
```

RedisTemplate 和 StringRedisTemplate 都是 Spring Data Redis 为我们提供的模板类，用来对数据进行操作，其中 StringRedisTemplate 只针对键值是字符串的数据进行操作。

在应用启动的时候，Spring 会为我们初始化这两个模板类，通过 @Resource 注解注入即可使用。

RedisTemplate 和 StringRedisTemplate 除了提供 opsForValue 方法用来操作简单属性数据之外，还提供了以下数据访问方法。

（1）opsForList：操作含有 list 的数据。

（2）opsForSet：操作含有 set 的数据。

（3）opsForZSet：操作含有 ZSet（有序 set）的数据。

（4）opsForHash：操作含有 hash 的数据。

当我们的数据存放到 Redis 的时候，键（key）和值（value）都是通过 Spring 提供的 Serializer 序列化到数据库的。RedisTemplate 默认使用 JdkSerializationRedisSerializer，而 StringRedisTemplate 默认使用 StringRedisSerializer。

7.2.5 测试

7.2.4 节测试用例代码开发完成之后，运行单元测试用例，除了可以在控制台查看打印结果信息和在 redis 客户端查看数据之外，我们还可以使用 RedisClient 客户端工具查看 Redis 缓存数据库中的数据。大家可以到网络上下载 RedisClient 客户端软件并安装到自己的操作系统中。安装完成之后，可以看到如图 7-10 所示的界面。

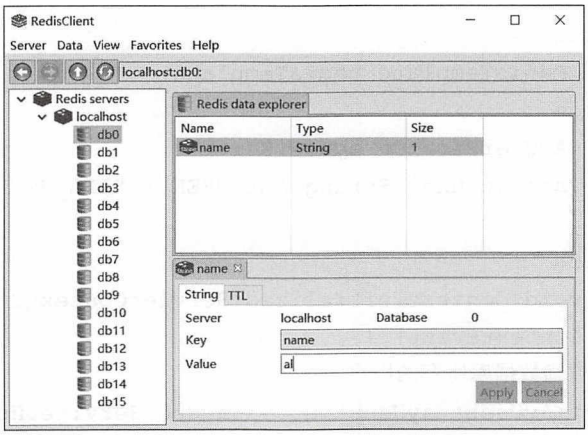


图 7-10　RedisClient 界面

在图 7-10 中，我们可以看到 Redis 默认有 16 个数据库，客户端与 Redis 建立连接后会自动选择 0 号数据库。通过该客户端，我们可以清楚地查看 Redis 数据库中存放的数据情况，同时可以在客户端中对数据进行增删改查等操作，使用起来非常方便。

7.3 Redis 缓存在 Spring Boot 中使用

7.3.1 监听器 Listener 的开发

在 6.2.2 节中，我们已经简单开发好 AyUserListener 监听器类，并在上下文启动时打印信息。本节在上下文初始化的方法中加载数据库中的所有用户数据，并存放到 Redis 缓存中。之所以要把用户数据存放到缓存中，是因为用户的数据属于变动不大的数据，适合存放到缓存中，在应用需要获取用户数据时，可以直接到 Redis 缓存中获取，不用到数据库中获取数据库连接查询数据，提高数据的访问速度。具体代码如下：

```
/**
 * 描述：监听器
 * @author Ay
 * @date   2017/11/4
 */
@WebListener
public class AyUserListener implements ServletContextListener {

    @Resource
    private RedisTemplate redisTemplate;
    @Resource
    private AyUserService ayUserService;
    private static final String ALL_USER = "ALL_USER_LIST";

    @Override
    public void contextInitialized(ServletContextEvent
    servletContextEvent) {
        // 查询数据库所有的用户
        List<AyUser> ayUserList =  ayUserService.findAll();
        // 清除缓存中的用户数据
```

```
            redisTemplate.delete(ALL_USER);
            //将数据存放到 Redis 缓存中
            redisTemplate.opsForList().leftPushAll(ALL_USER, ayUserList);
            //在真实项目中需要注释掉，查询所有的用户数据
            List<AyUser> queryUserList = redisTemplate.opsForList().
                range(ALL_USER, 0, -1);
            System.out.println("缓存中目前的用户数有：" + queryUserList.
                size() + " 人");
            System.out.println("ServletContext 上下文初始化");
        }

        @Override
        public void contextDestroyed(ServletContextEvent
        servletContextEvent) {
            System.out.println("ServletContext 上下文销毁");
        }
    }
```

redisTemplate.opsForList().leftPushAll：查询缓存中所有的用户数据，若 ALL_USER 键不存在，则会创建该键及与其关联的 List，之后再将参数中的 ayUserList 从左到右依次插入。

redisTemplate.opsForList().range：取链表中的全部元素，其中 0 表示第一个元素，-1 表示最后一个元素。

在 7.2.4 节中已经提到，当我们的数据存放到 Redis 的时候，键（key）和值（value）都是通过 Spring 提供的 Serializer 序列化到数据库的。RedisTemplate 默认使用 JdkSerializationRedisSerializer，而 StringRedisTemplate 默认使用 StringRedisSerializer。所以我们需要让用户类 AyUser（/src/main/java/com.example.demo.model）实现序列化接口 Serializable，具体代码如下：

```
/**
 * 描述：用户表
 * @Author 阿毅
 * @date   2017/10/8.
 */
@Entity
```

```
@Table(name = "ay_user")
public class AyUser implements Serializable{
    // 省略代码
}
```

7.3.2 项目启动缓存数据

在 7.3.1 节中，我们已经开发好 AyUserListener 监听器类和 AyUser 用户类，重新启动项目，这时数据库表 ay_user 中的所有数据都会加载到 Redis 缓存中。在 contextInitialized 方法中断点调试，出现如图 7-9 所示的界面，此时代码数据已经成功被加载到缓存中。同时，我们也可以用 Redis Client 客户端软件来查看用户数据是否存放到缓存中。

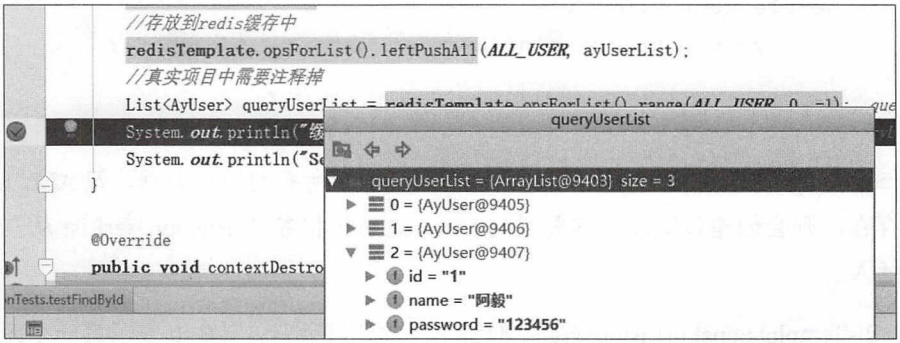


图 7-9 Redis 断点调试界面

7.3.3 更新缓存数据

项目启动并加载所有用户数据到缓存之后，我们需要修改 AyUserServiceImpl 中的接口，比如 findById、save、delete 等方法。如果在 Redis 缓存中查询不到数据，我们就需要到数据库查询，如果能够在数据库中查询到数据，除了返回数据之外，还需要把数据更新到缓存中。这样再次查询数据时，就不需要到数据库中查询数据了。这里主要对方法 findById 进行修改，AyUserServiceImpl 具体需要修改的代码如下：

```
// 省略代码
@Service
public class AyUserServiceImpl implements AyUserService {
```

```
    @Resource(name = "ayUserRepository")
    private AyUserRepository ayUserRepository;

    @Resource
    private RedisTemplate redisTemplate;

    private static final String ALL_USER = "ALL_USER_LIST";
    @Override
    public AyUser findById(String id){
        //step.1  查询 Redis 缓存中的所有数据
        List<AyUser> ayUserList = redisTemplate.opsForList().
            range(ALL_USER, 0, -1);
        if(ayUserList != null && ayUserList.size() > 0){
            for(AyUser user : ayUserList){
                if (user.getId().equals(id)){
                    return user;
                }
            }
        }
        //step.2  查询数据库中的数据
        AyUser ayUser = ayUserRepository.findOne(id);
        if(ayUser != null){
            //step.3 将数据插入 Redis 缓存中
            redisTemplate.opsForList().leftPush(ALL_USER, ayUser);
        }
        return ayUser;
    }
    // 省略

}
```

对于 save、delete 等方法的修改，思路是一样的，这里就不一一重复叙述了，读者可自己实现。虽然引入 Redis 缓存用户数据可以提高访问性能，但是带来的代码复杂度也是可想而知的。所以在以后的工作中，在性能和代码复杂度的权衡下，大家要根据具体的业务场景加以选择，不可滥用缓存，这是要跟大家提醒的重点。

7.3.4 测试

7.3.3 节代码开发完成之后，我们在测试类 MySpringBootApplicationTests 下继续添加如下测试方法：

```
@Test
    public void testFindById(){
        Long redisUserSize = 0L;
        // 查询 id = 1 的数据，该数据存在于 Redis 缓存中
        AyUser ayUser = ayUserService.findById("1");
        redisUserSize = redisTemplate.opsForList().
        size("ALL_USER_LIST");
        System.out.println("目前缓存中的用户数量为: " + redisUserSize);
        System.out.println("--->>> id: " + ayUser.getId() + " name:"
        + ayUser.getName());
        // 查询 id = 2 的数据，该数据存在于 Redis 缓存中
        AyUser ayUser1 = ayUserService.findById("2");
        redisUserSize = redisTemplate.opsForList().
        size("ALL_USER_LIST");
        System.out.println("目前缓存中的用户数量为: "+ redisUserSize);
        System.out.println("--->>> id: " + ayUser1.getId() +
        "name:" + ayUser1.getName());
        // 查询 id = 4 的数据，不存在于 Redis 缓存中，存在于数据库中，
        // 所以会把在数据库中查询的数据更新到缓存中
        AyUser ayUser3 = ayUserService.findById("4");
        System.out.println("--->>> id: " + ayUser3.getId() +
        "name:" + ayUser3.getName());
        redisUserSize = redisTemplate.opsForList().
        size("ALL_USER_LIST");
        System.out.println("目前缓存中的用户数量为: " + redisUserSize);

    }
```

代码开发完成之后，重新启动项目，数据库中的 3 条数据会重新被添加到 Redis 缓存中，如图 7-10 所示。项目启动成功之后，我们往数据库表 ay_user 中添加 id 为 4 的第 4 条数据，如图 7-11 所示。

图 7-10　数据库中存在 3 条数据

图 7-11　插入 id 为 4 的数据

最后执行单元测试方法 testFindById()，在 IntelliJ IDEA 的控制台中会打印如下信息：

```
目前缓存中的用户数量为：3
--->>> id: 1 name:阿毅
目前缓存中的用户数量为：3
--->>> id: 2 name:阿兰
--->>> id: 4 name:test
目前缓存中的用户数量为：4
```

第8章

集成 Log4j 日志

本章主要介绍 Log4j 基础知识、在 Spring Boot 中集成 Log4j、Log4j 在 Spring Boot 中的运用以及如何把日志打印到控制台并记录到日志文件中。

8.1 Log4j 介绍

Log4j 是 Apache 下的一个开源项目，通过使用 Log4j 可以将日志信息打印到控制台、文件等。我们也可以控制每一条日志的输出格式，通过定义每一条日志信息的级别能够更加细致地控制日志的生成过程。

在应用程序中添加日志记录有三个目的：

（1）监视代码中变量的变化情况，周期性地记录到文件中，供其他应用进行统计分析工作。

（2）跟踪代码运行时的轨迹，作为日后审计的依据。

（3）担当集成开发环境中调试器的作用，向文件或控制台打印代码的调试信息。

Log4j 中有三个主要的组件，分别是 Loggers（记录器），Appenders （输出源）和 Layouts（布局），这三个组件可以简单地理解为日志类别、日志要输出的地方和日志以哪种形式输出。Log4j 的原理图如图 8-1 所示。

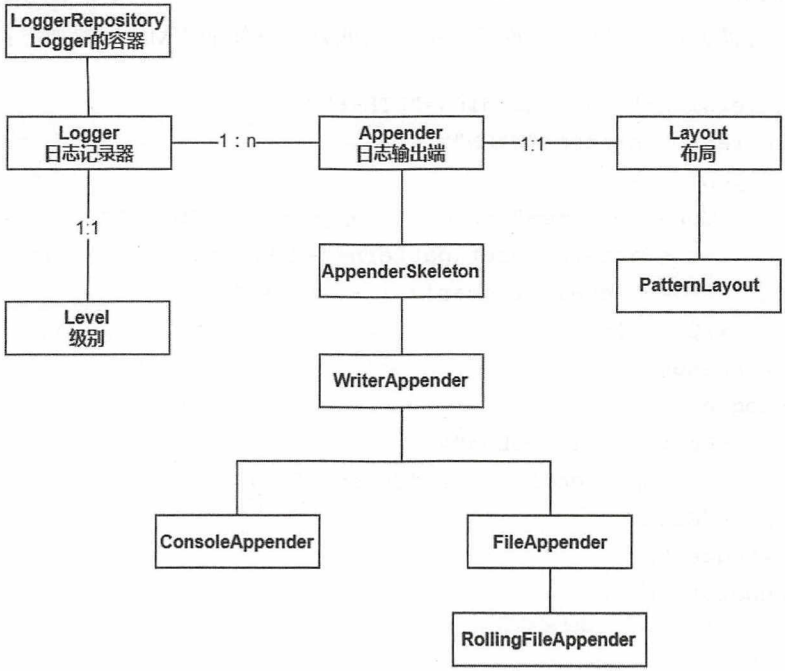


图 8-1　Log4j 日志框架简单原理图

- Loggers（记录器）：Loggers 组件被分为 7 个级别：all、debug、info、warn、error、fatal、off。这 7 个级别是有优先级的：all<debug< info< warn< error< fatal<off，分别用来指定这条日志信息的重要程度。Log4j 有一个规则：只输出级别不低于设定级别的日志信息。假设 Loggers 级别设定为 info，则 info、warn、error 和 fatal 级别的日志信息都会输出，而级别比 info 低的 debug 则不会输出。Log4j 允许开发人员定义多个 Logger，每个 Logger 拥有自己的名字，Logger 之间通过名字来表明隶属关系。

- Appenders（输出源）：Log4j 日志系统允许把日志输出到不同的地方，如控制台（Console）、文件（Files）等，可以根据天数或者文件大小产生新的文件，可以以流的形式发送到其他地方等。
- Layouts（布局）：Layout 的作用是控制 Log 信息的输出方式，也就是格式化输出的信息。

Log4j 支持两种配置文件格式，一种是 XML 格式的文件，一种是 Java 特性文件 log4j2.properties （键 = 值）。properties 文件简单易读，而 XML 文件可以配置更多的功能（比如过滤），没有好坏，能够融会贯通就好。具体的 XML 配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t]
            %-5level %logger{36} - %msg%n" />
        </Console>
    </Appenders>
    <Loggers>
        <Root level="debug">
            <AppenderRef ref="Console" />
        </Root>
    </Loggers>
</Configuration>
```

8.2 集成 Log4j2

8.2.1 引入依赖

在 Spring Boot 中集成 Log4j2，首先需要在 pom.xml 文件中引入所需的依赖，具体代码如下：

```
!-- log4j2 -->
<dependency>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

Spring Boot 默认使用 Logback 日志框架来记录日志，并用 INFO 级别输出到控制台，所以我们在引入 Log4j2 之前，需要先排除该包的依赖，再引入 Log4j2 的依赖。具体做法就是找到 pom.xml 文件中的 spring-boot-starter-web 依赖，使用 exclusion 标签排除 Logback，具体排除 Logback 依赖的代码如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
            <!-- 排查 Spring Boot 默认日志 -->
            <exclusion>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-logging</artifactId>
            </exclusion>
    </exclusions>
</dependency>
```

8.2.2 添加 Log4j 配置

在 8.1 节中已经讲过，Log4j2 支持两种配置文件格式，一种是 XML 格式的文件，一种是 properties 格式的文件。这里我们使用 XML 格式配置 Log4j2，properties 格式大家可以自学。使用 XML 格式配置很简单，只需要在 application.properties 文件中添加如下配置信息即可：

```
###log4j 配置
logging.config=classpath:log4j2.xml
```

配置完成之后，Spring Boot 会帮我们在 classpath 路径下查找 log4j2.xml 文件，所以最后一步只需要配置好 log4j2.xml 文件即可。

8.2.3 创建 log4j2.xml 文件

application.properties 配置完成之后，在目录 /src/main/resources 下新建空的日志配置文件 log4j2.xml。具体代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <appenders>

    </appenders>
    <loggers>
        <root level="all">

        </root>
    </loggers>
</Configuration>
```

8.3 使用 Log4j 记录日志

8.3.1 打印到控制台

现在我们把日志打印到控制台，需要往 log4j2.xml 配置文件中添加相关的配置，具体代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <!-- 指定日志的输出格式 -->
            <PatternLayout pattern="[%d{HH:mm:ss:SSS}] [%p]
            - %l - %m%n"/>
        </Console>
    </appenders>
    <loggers>
        <root level="all">
            <!-- 控制台输出 -->
```

```
            <appender-ref ref="Console"/>
        </root>
    </loggers>
</Configuration>
```

- <Console/>：指定控制台输出。
- <PatternLayout/>：控制日志的输出格式。

在 6.2.2 和 7.3.1 节中，我们已经开发好 AyUserListener 监听器，不过是使用 System.out.println 来打印信息，这是一种非常不合理的方式，现在我们把 Logger 类引入 AyUserListener.java 监听器中，同时把 System.out.println 相关代码注释掉，改成用日志方式记录信息。这样，在项目启动过程中，调用上下文初始化和销毁方法的时候，就会把日志记录到开发工具控制台或者日志文件中。AyUserListener 的具体代码如下：

```
/**
 * 描述：监听器
 * @author Ay
 * @date   2017/11/4
 */
@WebListener
public class AyUserListener implements ServletContextListener {
    // 省略代码
    // 需要添加的代码
    Logger logger = LogManager.getLogger(this.getClass());
    @Override
    public void contextInitialized(ServletContextEvent
    servletContextEvent) {
        // 查询数据库所有的用户
        List<AyUser> ayUserList =  ayUserService.findAll();
        // 清除缓存中的用户数据
        redisTemplate.delete(ALL_USER);
        // 存放到 Redis 缓存中
        redisTemplate.opsForList().leftPushAll(ALL_USER, ayUserList);
        // 在真实项目中需要注释掉
        List<AyUser> queryUserList = redisTemplate.opsForList().
        range(ALL_USER, 0, -1);
```

```
            //System.out.println("缓存中目前的用户数有：" +
              queryUserList.size() + " 人");
            //System.out.println("ServletContext 上下文初始化");
            logger.info("ServletContext 上下文初始化");
            logger.info("缓存中目前的用户数有：" + queryUserList.size() +
                        " 人");
        }

        @Override
        public void contextDestroyed(ServletContextEvent
        servletContextEvent) {
            //System.out.println("ServletContext 上下文销毁");
            logger.info("ServletContext 上下文销毁");
        }
    }
```

8.3.2 记录到文件

在 8.3.1 节中，日志只是被打印到控制台中，当项目真正被上线之后，是没有控制台这个概念的，在上线环境中，项目的日志都是被记录到文件中的。所以我们继续在 log4j2.xml 配置文件中添加相关配置，使日志可以被打印到文件中，具体代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <!-- 设置日志输出的格式 -->
            <PatternLayout pattern="[%d{HH:mm:ss:SSS}]
            [%p] - %l - %m%n"/>
        </Console>
        <RollingFile name="RollingFileInfo" fileName="D:/info.log"
                     filePattern="D:/$${date:yyyy-MM}/info-
                     %d{yyyy-MM-dd}-%i.log">
            <Filters>
                <ThresholdFilter level="INFO"/>
            </Filters>
            <PatternLayout pattern="[%d{HH:mm:ss:SSS}] [%p] - %l
            - %m%n"/>
```

```
            <Policies>
                <TimeBasedTriggeringPolicy/>
                <SizeBasedTriggeringPolicy size="100 MB"/>
            </Policies>
        </RollingFile>
    </appenders>

    <loggers>
        <root level="all">
            <appender-ref ref="Console"/>
            <appender-ref ref="RollingFileInfo"/>
        </root>
    </loggers>

</Configuration>
```

- <RollingFile> 标签：fileName 用于定义日志的数据路径，如 D:/info.log；filePattern 用于定义日志的匹配方式。
- <Filters> 标签：日志过滤策略，<ThresholdFilter> 标签用于指定日志信息的最低输出级别，默认为 DEBUG。

现在我们修改 3.2.3 节中 AyUserServiceImpl 类的删除方法 delete，希望删除用户这个操作可以被记录到日志文件中，AyUserServiceImpl 类代码具体的修改如下：

```
/**
 * 描述：用户服务层实现类
 * @author 阿毅
 * @date   2017/10/14
 */
//@Transactional
@Service
public class AyUserServiceImpl implements AyUserService {
    // 省略代码
    // 需要添加的代码
    Logger logger = LogManager.getLogger(this.getClass());

    @Override
```

```
    public void delete(String id) {
        ayUserRepository.delete(id);
         // 需要添加的代码
        logger.info("userId:" + id + "用户被删除");
    }

    // 省略代码
}
```

8.3.3 测试

代码开发完成之后，接下来就是测试工作了。重新启动项目，启动之前，记得打开 Redis 服务器，因为之前我们已经在 Spring Boot 中整合了 Redis。在项目重启的过程中，我们可以在 IntelliJ IDEA 控制台中看到如图 8-2 所示的信息。同时，可以到 D 盘查看日志文件 info.log，在日志文件中按 Ctrl + F 快捷键，可以查询到和图 8-2 所示一样的信息。

```
[DEBUG] - org.springframework.data.redis.core.RedisConnectionUtils.doGetConnection(RedisConnectionUtils.java:126) - Openin
[DEBUG] - org.springframework.data.redis.core.RedisConnectionUtils.releaseConnection(RedisConnectionUtils.java:210) - Clos
[INFO] - com.example.demo.listener.AyUserListener.contextInitialized(AyUserListener.java:42) - ServletContext上下文初始化
[INFO] - com.example.demo.listener.AyUserListener.contextInitialized(AyUserListener.java:43) - 缓存中目前的用户数有：3 人
[DEBUG] - org.springframework.web.filter.GenericFilterBean.init(GenericFilterBean.java:208) - Initializing filter 'request
[DEBUG] - org.springframework.web.filter.GenericFilterBean.init(GenericFilterBean.java:239) - Filter 'requestContextFilter
[DEBUG] - org.springframework.web.filter.GenericFilterBean.init(GenericFilterBean.java:208) - Initializing filter 'httpPut
[DEBUG] - org.springframework.web.filter.GenericFilterBean.init(GenericFilterBean.java:239) - Filter 'httpPutFormContentFi
[DEBUG] - org.springframework.web.filter.GenericFilterBean.init(GenericFilterBean.java:208) - Initializing filter 'hiddenH
```

图 8-3 Redis 断点调试界面

接着再测试一下在删除用户的时候日志是否可以打印到控制台或者记录到日志文件中。在测试类 MySpringBootApplicationTests 下添加测试用例，具体代码如下：

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MySpringBootApplicationTests {
    // 省略代码
     Logger logger = LogManager.getLogger(this.getClass());

       @Test
       public void testLog4j(){
```

```
            ayUserService.delete("4");
            logger.info("delete success!!!");
        }
    }
```

数据库 ay_test 表中存在 4 条数据，如图 8-3 所示。运行单元测试方法 testLog4j，如果同样可以在 IntelliJ IDEA 控制台或者 D 盘的 info.log 文件中打印如图 8-4 所示的信息，证明 Spring Boot 整合 Log4j 以及在 Spring Boot 中运用 Log4j 成功。

id	name	password
1	阿毅	123456
2	阿兰	123456
3	阿华	123
4	test	12345

图 8-3 ay_test 表中的 4 条数据

```
nternal.ResourceRegistryStandardImpl.releaseResources(ResourceRegistryStandardImpl.ja
nternal.LogicalConnectionManagedImpl.close(LogicalConnectionManagedImpl.java:213) - C
nternal.ResourceRegistryStandardImpl.releaseResources(ResourceRegistryStandardImpl.ja
nternal.LogicalConnectionManagedImpl.close(LogicalConnectionManagedImpl.java:220) - L
n.support.TransactionSynchronizationManager.doUnbindResource(TransactionSynchronizat
ServiceImpl.delete(AyUserServiceImpl.java:77) - userId:4用户被删除
plicationTests.testLog4j(MySpringBootApplicationTests.java:152) - delete success!!!
```

图 8-4 Redis 断点调试界面

启动项目的时候，记得启动 Redis 服务器，否则会报错。

第9章

Quartz 定时器和发送 Email

本章主要介绍在 Spring Boot 中使用 XML 配置和 Java 注解两种方式定义和使用 Quartz 定时器，以及如何在 Spring Boot 中通过 JavaMailSender 接口给用户发送广告邮件等。

9.1 使用 Quartz 定时器

9.1.1 Quartz 概述

Quartz 是一个完全由 Java 编写的开源任务调度框架，通过触发器设置作业定时运行规则、控制作业的运行时间。Quartz 定时器作用很多，比如定时发送信息、定时生成报表等。

Quartz 框架主要核心组件包括调度器、触发器、作业。调度器作为作业的总指挥，触发器作为作业的操作者，作业为应用的功能模块。其关系如图 9-1 所示。

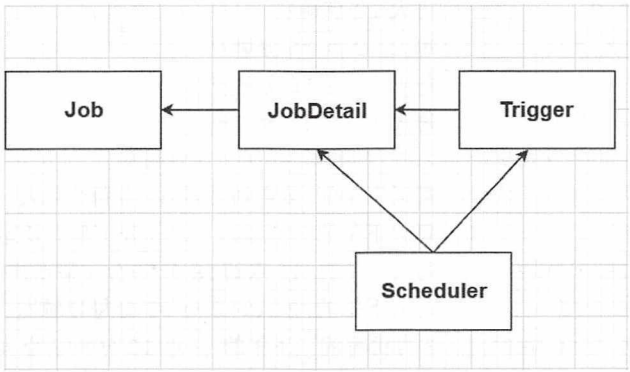


图 9-1　Quartz 各个组件的关系

Job 是一个接口，该接口只有一个方法 execute，被调度的作业（类）需实现该接口中的 execute() 方法，JobExecutionContext 类提供了调度上下文的各种信息。每次执行该 Job 均重新创建一个 Job 实例。Job 的源代码如下：

```
public interface Job {
    void execute(JobExecutionContext var1) throws
JobExecutionException;
}
```

Quartz 在每次执行 Job 时，都重新创建一个 Job 实例，所以它不直接接收一个 Job 实例，相反它接收一个 Job 实现类，以便运行时通过 newInstance() 的反射机制实例化 Job。因此需要通过一个类来描述 Job 的实现类及其他相关的静态信息，如 Job 名字、描述、关联监听器等信息，JobDetail 承担了这一角色。JobDetail 用来保存作业的详细信息。一个 JobDetail 可以有多个 Trigger，但是一个 Trigger 只能对应一个 JobDetail。

Trigger 触发器描述触发 Job 的执行规则，主要有 SimpleTrigger 和 CronTrigger 两个子类。当仅需触发一次或者以固定时间间隔周期执行时，SimpleTrigger 是最适合的选择；而 CronTrigger 则可以通过 Cron 表达式定义出各种复杂时间规则的调度方案，如每早晨 9:00 执行，周一、周三、周五下午 5:00 执行等。Cron 表达式定义如下：

CronTrigger 配置格式：

```
格式：[秒] [分] [小时] [日] [月] [周] [年]
0 0 12 * * ?              每天 12 点触发
0 15 10 ? * *             每天 10 点 15 分触发
0 15 10 * * ?             每天 10 点 15 分触发
0 15 10 * * ? *           每天 10 点 15 分触发
0 15 10 * * ? 2005        2005 年每天 10 点 15 分触发
0 * 14 * * ?              每天下午的 2 点到 2 点 59 分每分触发
0 0/5 14 * * ?            每天下午的 2 点到 2 点 59 分（整点开始，每隔 5 分触发）
0 0/5 14,18 * * ?         每天下午的 18 点到 18 点 59 分（整点开始，每隔 5 分触发）
0 0-5 14 * * ?            每天下午的 2 点到 2 点 05 分每分触发
0 10,44 14 ? 3 WED        3 月份每周三下午的 2 点 10 分和 2 点 44 分触发
0 15 10 ? * MON-FRI       从周一到周五每天上午的 10 点 15 分触发
0 15 10 15 * ?            每月 15 号上午 10 点 15 分触发
0 15 10 L * ?             每月最后一天的 10 点 15 分触发
0 15 10 ? * 6L            每月最后一周的星期五的 10 点 15 分触发
0 15 10 ? * 6L 2002-2005      从 2002 年到 2005 年每月最后一周的星期五的
                              10 点 15 分触发
0 15 10 ? * 6#3           每月的第三周的星期五开始触发
0 0 12 1/5 * ?            每月的第一个中午开始每隔 5 天触发一次
0 11 11 11 11 ?           每年的 11 月 11 号 11 点 11 分触发（光棍节）
```

Scheduler 负责管理 Quartz 的运行环境，Quartz 是基于多线程架构的，启动的时候会初始化一套线程，这套线程用来执行一些预置的作业。Trigger 和 JobDetail 可以注册到 Scheduler 中。Scheduler 可以将 Trigger 绑定到某一 JobDetail 中，这样当 Trigger 触发时，对应的 Job 就会被执行。Scheduler 拥有一个 SchedulerContext，类似于 ServletContext，保存着 Scheduler 上下文信息，Job 和 Trigger 都可以访问 SchedulerContext 内的信息。Scheduler 使用一个线程池作为任务运行的基础设施，任务通过共享线程池中的线程提高运行效率。

9.1.2 引入依赖

在 Spring Boot 中集成 Quartz，首先需要在 pom.xml 文件中引入所需的依赖，具体代码如下：

```
<!-- quartz 定时器 -->
<dependency>
    <groupId>org.quartz-scheduler</groupId>
    <artifactId>quartz</artifactId>
    <version>2.2.3</version>
</dependency>
```

9.1.3 定时器配置文件

创建定时器的方法有两种：① 使用 XML 配置文件的方式；②使用注解的方式。注解的方式不需要任何配置文件且简单高效，这两种方式都会讲到。我们先来讲第一种方式，也就是配置文件的方式。首先在 /src/main/resources 目录下新建配置文件 spring-mvc.xml，具体代码如下：

```
<beans xmlns= "http://www.springframework.org/schema/beans"
       xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context= "http://www.springframework.org/schema
                      /context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:task="http://www.springframework.org/schema/task"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context
       /spring-context-4.2.xsd
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop.xsd
       http://www.springframework.org/schema/task
       http://www.springframework.org/schema/task/spring-task.xsd">

    <context:annotation-config/>
    <!-- 利用 import 引入定时器的文件 -->
    <import resource="spring-quartz.xml"/>
```

```
</beans>
```

- <import> 标签：利用 import 标签导入定时器的配置文件，该标签可以根据具体业务分离配置文件。然后在 /src/main/resources 目录下新建 spring-quartz.xml 配置文件，具体代码如下：

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema
       /beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <!--  定义 Job 对象 -->
    <bean id="taskJob" class="com.example.demo.quartz.TestTask"/>
    <!--  定义 JobDetail 对象 -->
    <bean id="jobDetail"
    class="org.springframework.scheduling.quartz.
           MethodInvokingJobDetailFactoryBean">
        <!--  目标对象 taskJob -->
        <property name="targetObject">
            <ref bean="taskJob"/>
        </property>
        <!--  目标方法 -->
        <property name="targetMethod">
            <value>run</value>
        </property>
    </bean>

    <!--  调度触发器 -->
    <bean id="myTrigger"
        class="org.springframework.scheduling.quartz.
        CronTriggerFactoryBean">
        <!--  指定使用 jobDetail -->
        <property name="jobDetail">
            <ref bean="jobDetail" />
```

```
        </property>
        <!--  定义触发规则，每 10 秒执行一次 -->
        <property name="cronExpression">
            <value>0/10 * * * * ?</value>
        </property>
    </bean>

    <!-- 调度工厂 -->
    <bean id="scheduler"
          class="org.springframework.scheduling.quartz.
                 SchedulerFactoryBean">
        <!-- 注册触发器，可注册多个 -->
        <property name="triggers">
            <list>
                <ref bean="myTrigger"/>
            </list>
        </property>
    </bean>
</beans>
```

在 spring-quartz.xml 配置文件中分别定义了 Job、JobDetail、Trigger 以及 Scheduler，并配置了它们之间的关系。

9.1.4 创建定时器类

定时器的依赖以及配置文件开发完成之后，在 /src/main/java/com.example.demo.quartz 目录下新建定时器类 TestTask.java，具体代码如下：

```
/**
 * 描述：定时器类
 * @author Ay
 * @date   2017/11/18
 */
public class TestTask {
```

```
    // 日志对象
    private static final Logger logger = LogManager.
        getLogger(TestTask.class);

    public void run() {
        logger.info("定时器运行了 !!!");
    }
}
```

使用第二种创建定时器的方法更加简单，只需要创建一个定时器类，加上相关的注解就搞定了。比如，我们可以在 /src/main/java/com.example.demo.quartz 目录下创建 SendMailQuartz 定时器类，具体代码如下：

```
/**
 * 描述：定时器类
 * @author Ay
 * @date   2017/11/18
 */
@Component
@Configurable
@EnableScheduling
public class SendMailQuartz {

    // 日志对象
    private static final Logger logger = LogManager.
    getLogger(SendMailQuartz.class);

    // 每 5 秒执行一次
    @Scheduled(cron = "*/5 * *  * * * ")
    public void reportCurrentByCron(){
        logger.info("定时器运行了 !!!");
    }

}
```

- @Configurable：加上此注解的类相当于 XML 配置文件，可以被 Spring Boot 扫描初始化。

- @EnableScheduling：通过在配置类注解 @EnableScheduling 来开启对计划任务的支持，然后在要执行计划任务的方法上注解 @Scheduled，声明这是一个计划任务。
- @Scheduled：注解为定时任务，在 cron 表达式里写执行的时机。

9.1.5 Spring Boot 扫描配置文件

在 9.1.3 节中，我们已经开发好 spring-mvc.xml 配置文件，但是想让 Spring Boot 扫描到该配置文件，还需要在入口类 MySpringBootApplication 中添加 @ImportResource 注解，具体代码如下：

```
@SpringBootApplication
@ServletComponentScan
@ImportResource(locations={"classpath:spring-mvc.xml"})
public class MySpringBootApplication {
    // 省略代码
}
```

- @ImportResource：导入资源配置文件，让 Spring Boot 可以读取到，类似于 XML 配置文件中的 <import> 标签。

9.1.6 测试

代码开发完成之后，重新启动项目，在 IntelliJ IDEA 控制台中可以看到如图 9-2 所示的信息，证明在 Spring Boot 中整合 Quartz 定时器成功。

```
org.springframework.transaction.support.TransactionSynchronizationManager.doUnbindResource(TransactionSynchro
org.quartz.core.JobRunShell.run(JobRunShell.java:201) - Calling execute on job DEFAULT.jobDetail1
org.quartz.core.QuartzSchedulerThread.run(QuartzSchedulerThread.java:276) - batch acquisition of 1 triggers
com.example.demo.quartz.TestTask.run(TestTask.java:17) - 定时器运行了!!!
com.example.demo.quartz.SendMailQuartz.reportCurrentByCron(SendMailQuartz.java:40) - 定时器运行了!!!
org.springframework.transaction.support.TransactionSynchronizationManager.bindResource(TransactionSynchroniza
org.springframework.transaction.support.AbstractPlatformTransactionManager.getTransaction(AbstractPlatformTra
```

图 9-2　Quartz 定时器打印信息

9.2 Spring Boot 发送 Email

9.2.1 Email 介绍

邮件服务在互联网早期就已经出现，如今已成为人们互联网生活中必不可少的一项服务。邮件发送与接收的过程如下：

（1）发件人使用 SMTP 协议传输邮件到邮件服务器 A。

（2）邮件服务器 A 根据邮件中指定的接收者投送邮件至相应的邮件服务器 B。

（3）收件人使用 POP3 协议从邮件服务器 B 接收邮件。

SMTP（Simple Mail Transfer Protocol）是电子邮件（Email）传输的互联网标准，定义在 RFC 5321，默认使用 25 端口。

POP3（Post Office Protocol - Version 3）主要用于支持使用客户端远程管理在服务器上的电子邮件，定义在 RFC 1939，为 POP 协议的第三版（最新版）。

这两个协议均属于 TCP/IP 协议族的应用层协议，运行在 TCP 层。

发送邮件的需求比较常见，如找回密码、事件通知、向用户发送广告邮件等。SUN 公司给广大 Java 开发人员提供了一款邮件发送和接收的开源类库 JavaMail，支持常用的邮件协议，如 SMTP、POP3、IMAP 等。开发人员使用 JavaMail 编写邮件程序时，不再需要考虑底层的通信细节（如 Socket），而是关注逻辑层面。JavaMail 可以发送各种复杂 MIME 格式的邮件内容，注意 JavaMail 仅支持 JDK4 及以上版本。虽然 JavaMail 是 JDK 的 API，但它并没有直接加入 JDK 中，所以我们需要另外添加依赖。

Spring 提供了非常好用的 JavaMailSender 接口实现邮件发送，在 Spring Boot 的 Starter 模块中已为此提供了自动化配置。

9.2.2 引入依赖

要在 Spring Boot 中集成 Mail，首先需要在 pom.xml 文件中引入所需的依赖，具体代码如下：

```
<!-- mail start -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mail</artifactId>
</dependency>
```

9.2.3 添加 Email 配置

在 pom 文件中引入 Mail 所需的依赖之后，我们需要在 application.properties 文件中添加如下配置信息：

```
###Mail 邮件配置
### 邮箱主机
spring.mail.host=smtp.163.com
### 用户名
spring.mail.username=ay_test@163.com
### 设置的授权码
spring.mail.password=ay12345
### 默认编码
spring.mail.default-encoding=UTF-8
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
spring.mail.properties.mail.smtp.starttls.required=true
```

9.2.4 在定时器中发送邮件

在 Spring Boot 中添加完依赖和配置之后，我们在项目的目录 /src/main/java/com.example.demo.mail 下新建邮件服务接口类 SendJunkMailService，具体代码如下：

```
/**
 * 描述：发送用户邮件服务
 * @author Ay
 * @date   2017/11/19
 */
public interface SendJunkMailService {

    boolean sendJunkMail(List<AyUser> ayUser);
}
```

然后继续在项目的目录 /src/main/java/com.example.demo.mail.impl 下新建接口类的实现类 SendJunkMailServiceImpl.java，具体代码如下：

```
/**
 * 描述：发送用户邮件服务
 * @author Ay
 * @date   2017/11/19
 */
@Service
public class SendJunkMailServiceImpl implements SendJunkMailService {

    @Autowired
    JavaMailSender mailSender;
    @Resource
    private AyUserService ayUserService;
    @Value("${spring.mail.username}")
    private String from;
          public static final Logger logger =
              LogManager.getLogger(SendJunkMailServiceImpl.class);

    @Override
    public boolean sendJunkMail(List<AyUser> ayUserList) {

        try{
            if(ayUserList == null || ayUserList.size() <= 0 )
            return Boolean.FALSE;
            for(AyUser ayUser: ayUserList){
                MimeMessage mimeMessage = this.mailSender.
                createMimeMessage();
                MimeMessageHelper message = new
                MimeMessageHelper(mimeMessage);
                //邮件发送方
                message.setFrom(from);
                //邮件主题
                message.setSubject("地瓜今日特卖");
                //邮件接收方
                message.setTo("al_test@163.com");
```

```
                //邮件内容
                message.setText(ayUser.getName() +" ,
                你知道吗？厦门地瓜今日特卖，一斤只要 9 元 ");
                //发送邮件
                this.mailSender.send(mimeMessage);
            }
        }catch(Exception ex){
            logger.error("sendJunkMail error and ayUser=%s",
            ayUserList, ex);
            return Boolean.FALSE;
        }
        return Boolean.TRUE;
    }
}
```

- @Value：可以将 application.properties 配置文件中的配置设置到属性中。如上面的代码中，将 spring.mail.username 的值 huangwenyi10@163.com 设置给 from 属性。

- JavaMailSender：邮件发送接口。在 Spring Boot 的 Starter 模块中已为此提供了自动化配置，我们只需要通过注解 @Autowired 注入进来即可使用。

在 9.1.4 节中已经开发了 SendMailQuartz 定时器类，现在我们重新修改该类，让定时器类能够每隔一段时间给数据库的用户发送广告邮件，SendMailQuartz 类具体的修改如下：

```
/**
 * 描述：定时器类
 * @author Ay
 * @date   2017/11/18
 */
@Component
@Configurable
@EnableScheduling
public class SendMailQuartz {

    //日志对象
```

```
    private static final Logger logger = LogManager.
    getLogger(SendMailQuartz.class);

    @Resource
    private SendJunkMailService sendJunkMailService;
    @Resource
    private AyUserService ayUserService;

    //每 5 秒执行一次
    @Scheduled(cron = "*/5 * *  * * * ")
    public void reportCurrentByCron(){
        List<AyUser> userList = ayUserService.findAll();
        if (userList == null || userList.size() <= 0) return;
        //发送邮件
        sendJunkMailService.sendJunkMail(userList);
        logger.info(" 定时器运行了 !!!");
    }

}
```

9.2.5 测试

代码全部开发完成之后，重新启动项目，发送邮件定时器类 SendMailQuartz，每隔 5 秒（真实项目会设置得比较长，比如 1 天、2 天等）会查询数据库表 ay_test 中的所有用户，并发送广告邮件给对应的用户。我们登录 al_test@163.com 邮箱便可以查看到如图 9-1 和图 9-2 所示的信息。

图 9-1 163 邮箱界面

图 9-2 163 邮件内容

第 10 章

集成 MyBatis

本章主要介绍如何在 Spring Boot 中集成 MyBatis 框架、通过 MyBatis 框架实现查询等功能以及如何使用 MyBatisCodeHelper 插件快速生成增删改查代码。

10.1 MyBatis 介绍

10.1.1 MyBatis 概述

MyBatis 是一款优秀的持久层框架，支持定制化 SQL、存储过程以及高级映射。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。MyBatis 可以使用简单的 XML 或注解来配置和映射原生信息，将接口和 Java 的 POJOs（Plain Old Java Objects，普通的 Java 对象）映射成数据库中的记录。

10.2 集成 MyBatis

10.2.1 引入依赖

在 Spring Boot 中集成 Mybatis，首先需要在 pom.xml 文件中引入所需的依赖，具体代码如下：

```
<!-- mybatis start -->
    <dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.1</version>
</dependency>
```

10.2.2 添加 MyBatis 配置

在 pom 文件中添加 Mybatis 所需的依赖之后，我们需要在 application.properties 文件中添加如下配置信息：

```
### mybatis 配置
mybatis.mapper-locations=classpath:/mappers/*Mapper.xml
mybatis.type-aliases-package=com.example.demo.dao
```

- mybatis.mapper-locations：Mapper 资源文件存放的路径。
- mybatis.type-aliases-package：Dao 接口文件存放的目录。

10.2.3 Dao 层和 Mapper 文件开发

application.properties 配置添加完成之后，我们需要根据 MyBatis 配置添加对应的文件夹。首先，在 /src/main/java/com.example.demo.dao 目录下新建 AyUserDao 接口，这样 Spring Boot 启动时，就可以根据 application.properties 配置 mybatis.type-aliases-package，找到 AyUserDao 接口。AyUserDao 具体代码如下：

```
/**
 * 描述：用户 DAO
 * @author Ay
```

```
 * @date   2017/11/20.
 */
@Mapper
public interface AyUserDao {

    /**
     *  描述：通过用户名和密码查询用户
     * @param name
     * @param password
     */
    AyUser findByNameAndPassword(@Param("name") String name,
                                 @Param("password") String password);

}
```

- @Mapper：重要注解， MyBatis 根据接口定义与 Mapper 文件中的 SQL 语句动态创建接口实现。
- @Param：注解参数，在 Mapper.xml 配置文件中，可以采用 #{} 的方式对 @Param 注解括号内的参数进行引用。
- findByNameAndPassword：该方法可以通过用户名和密码查询用户。

然后在 /src/main/resources 目录下新建 AyUserMapper.xml 文件，Spring Boot 在项目启动时，会根据 application.properties 配置 mybatis.mapper-locations 找到该文件。AyUserMapper 具体代码如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.example.demo.dao.AyUserDao" >

    <resultMap id="UserResultMap" type="com.example.demo.model.AyUser">
        <id column="id" property="id" jdbcType="VARCHAR"/>
        <result column="name" property="name" jdbcType="VARCHAR"/>
        <result column="password" property="password"
         jdbcType= "VARCHAR"/>
        <result column="mail" property="mail" jdbcType="VARCHAR"/>
    </resultMap>
```

```
    <select id="findByNameAndPassword" resultMap="UserResultMap"
     parameterType="String">
           select * from ay_user u
         <where>
             u.name = #{name}
             and u.password = #{password}
         </where>
     </select>

</mapper>
```

- mapper> 标签：该标签的 namespace 属性用于绑定 Dao 接口。
- <select> 标签：用来编写 select 语句，映射查询语句。select 标签有几个重要的属性，比如 resultMap。
- <resultMap>：映射管理器 resultMap，是 Mybatis 中最强大的工具，描述了如何将数据库查询的结果集映射到 Java 对象，并管理结果和实体类之间的映射关系。

AyUserDao 代码开发完成之后，在之前开发好的 AyUserService 接口类中添加接口 findByNameAndPassword，具体代码如下：

```
/**
 * 描述：用户服务层接口
 * @author 阿毅
 * @date   2017/10/14
 */
public interface AyUserService {
    // 此处省略代码

    AyUser findByNameAndPassword(String name, String password);

}
```

然后，在 AyUserServiceImpl 类中实现 findByNameAndPassword 接口，具体代码如下：

```
/**
 * 描述：用户服务层实现类
 * @author 阿毅
 * @date   2017/10/14
 */
//@Transactional
@Service
public class AyUserServiceImpl implements AyUserService {

    //此处省略代码

    @Resource
    private AyUserDao ayUserDao;

    @Override
    public AyUser findByNameAndPassword(String name,
    String password) {
        return ayUserDao.findByNameAndPassword(name, password);
    }
}
```

10.2.4 测试

代码开发完成之后，在 MySpringBootApplicationTests 类下添加测试方法，具体代码如下：

```
@Resource
private AyUserService ayUserService;

@Test
public void testMybatis(){
    AyUser ayUser = ayUserService.findByNameAndPassword("阿毅",
                    "123456");
    logger.info(ayUser.getId() + ayUser.getName());

}
```

执行测试用例，在 IntelliJ IDEA 控制台可以看到相应的打印信息。

第 11 章

异步消息与异步调用

本章主要介绍 ActiveMQ 的安装与使用、Spring Boot 集成 ActiveMQ、利用 ActiveMQ 实现异步发表微信说说以及 Spring Boot 异步调用 @Async 等。

11.1 JMS 消息介绍

JMS（Java Message Service，Java 消息服务）是一组 Java 应用程序接口，提供消息的创建、发送、读取等一系列服务。JMS 提供了一组公共应用程序接口和响应的语法，类似于 Java 数据库的统一访问接口 JDBC，是一种与厂商无关的 API，使得 Java 程序能够与不同厂商的消息组件很好地进行通信。

JMS 支持两种消息发送和接收模型。一种称为 P2P（Ponit to Point）模型，即采

用点对点的方式发送消息。P2P 模型是基于队列的，消息生产者（Producer）发送消息到队列，消息消费者（Consumer）从队列中接收消息，队列的存在使得消息的异步传输成为可能。P2P 模式图如图 11-1 所示。

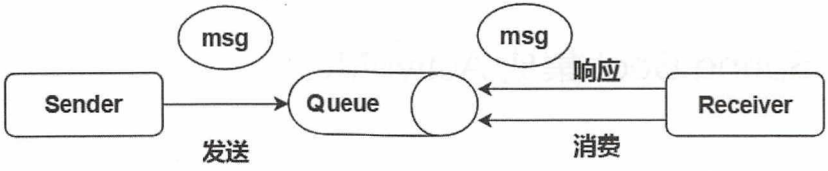


图 11-1　P2P 模式图

P2P 的特点是每个消息只有一个消费者（一旦被消费，消息就不在消息队列中），发送者和接收者之间在时间上没有依赖性，也就是说当发送者发送了消息之后，无论接收者有没有正在运行，都不会影响消息被发送到队列中，接收者在成功接收消息之后需向队列应答成功。

另一种称为 Pub / Sub（Publish / Subscribe，发布 - 订阅）模型，发布 - 订阅模型定义了如何向一个内容节点发布和订阅消息，这个内容节点称为 Topic（主题）。主题可以认为是消息传递的中介，消息发布者将消息发布到某个主题，而消息订阅者则从主题订阅消息。主题使得消息的订阅者与消息的发布者互相保持独立，不需要进行接触即可保证消息的传递，发布 - 订阅模型在消息的一对多广播时采用。

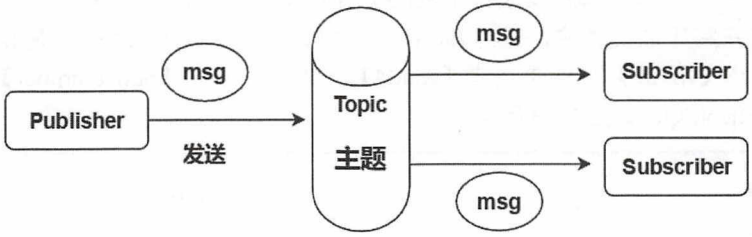


图 11-2　Pub / Sub 模式图

Pub/Sub 的特点是每个消息可以有多个消费者，发布者和订阅者之间有时间上的依赖性。针对某个主题（Topic）的订阅者，必须创建一个订阅者之后，才能消费发布者的消息，而且为了消费消息，订阅者必须保持运行的状态。为了缓和这样严格的时

间相关性，JMS 允许订阅者创建一个可持久化的订阅。这样，即使订阅者没有被激活（运行），也能接收到发布者的消息。如果你希望发送的消息可以不做任何处理、被一个消息者处理或者可以被多个消费者处理，那么可以采用 Pub/Sub 模型。

11.2 Spring Boot 集成 ActiveMQ

11.2.1 ActiveMQ 概述

MQ 全称为 MessageQueue，中文名为消息队列，是一个消息的接收和转发的容器，可用于消息推送。ActiveMQ 是 Apache 提供的一个开源消息系统，完全采用 Java 来实现，因此能够很好地支持 J2EE 提出的 JMS（Java Message Service，Java 消息服务）规范。

11.2.2 ActiveMQ 的安装

安装 ActiveMQ 之前，我们需要到官方网站（http://activemq.apache.org/activemq-5150-release.html）下载，本书使用 apache-activemq-5.15.0 版本进行讲解。ActiveMQ 具体安装步骤如下：

步骤 01 将从官方网站下载的安装包 apache-activemq-5.15.0-bin.zip 解压。

步骤 02 打开解压的文件夹，进入 bin 目录，根据操作系统是 32 位还是 64 位，选择进入【win32】文件夹或者【win64】文件夹。双击【activemq.bat】即可启动 ActiveMQ，如图 11-3 所示。

名称	修改日期
activemq.bat	2017/6/27 13:48
InstallService.bat	2017/6/27 13:48
UninstallService.bat	2017/6/27 13:48
wrapper.conf	2017/6/27 13:48
wrapper.dll	2017/6/27 13:43
wrapper.exe	2017/6/27 13:43

图 11-3 ActiveMQ 安装文件

步骤 03　当看到如图 11-4 所示的启动信息时，代表 ActiveMQ 安装成功。从图 11-4 中可以看出，ActiveMQ 默认启动到 8161 端口。

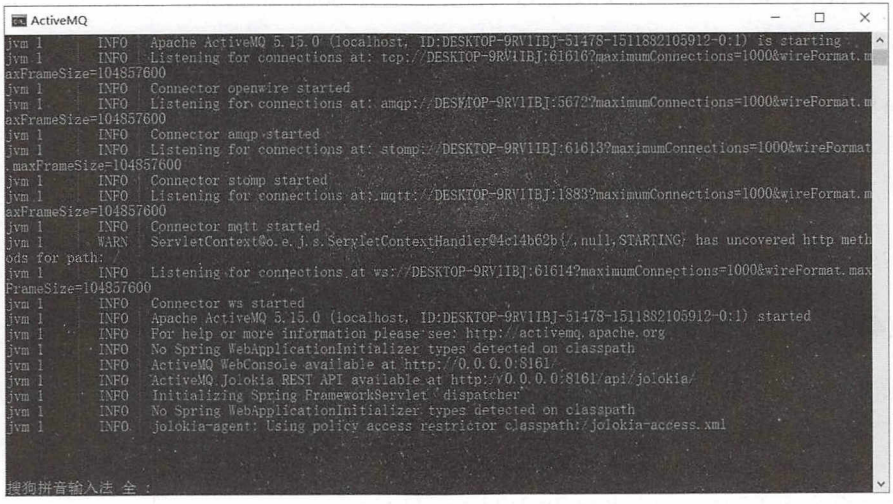


图 11-4　ActiveMQ 启动成功界面

步骤 04　安装成功之后，在浏览器中输入 http://localhost:8161/admin 链接访问，第一次访问需要输入用户名 admin 和密码 admin 进行登录，登录成功之后，可以看到 ActiveMQ 的首页，如图 11-5 所示。

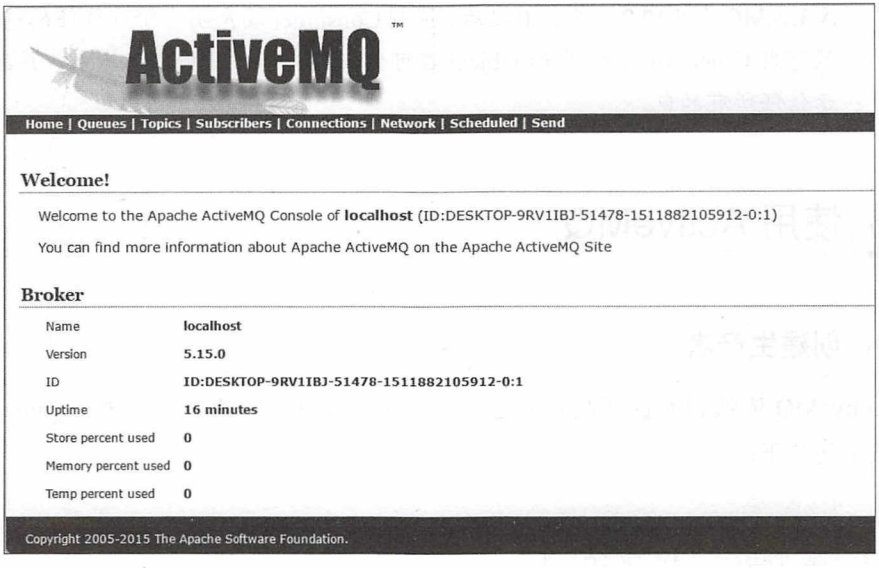


图 11-5　ActiveMQ 首页

11.2.3 引入依赖

在 Spring Boot 中集成 ActiveMQ，首先需要在 pom.xml 文件中引入所需的依赖，具体代码如下：

```
<!-- activemq start -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
```

11.2.4 添加 ActiveMQ 配置

依赖添加完成之后，我们需要在 application.properties 配置文件中添加 ActiveMQ 配置，具体代码如下：

```
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.in-memory=true
spring.activemq.pool.enabled=false
spring.activemq.packages.trust-all=true
```

- spring.activemq.packages.trust-all：ObjectMessage 的使用机制是不安全的，ActiveMQ 自 5.12.2 和 5.13.0 之后，强制 Consumer 端声明一份可信任的包列表，只有当 ObjectMessage 中的 Object 在可信任包内，才能被提取出来。该配置表示信任所有的包。

11.3 使用 ActiveMQ

11.3.1 创建生产者

ActiveMQ 依赖和配置开发完成之后，首先在数据库中建立说说表 ay_mood。具体建表语句如下：

```
DROP TABLE IF EXISTS 'ay_mood';
CREATE TABLE 'ay_mood' (
  'id' varchar(32) NOT NULL,
```

```
  'content' varchar(256) DEFAULT NULL,
  'user_id' varchar(32) DEFAULT NULL,
  'praise_num' int(11) DEFAULT NULL,
  'publish_time' datetime DEFAULT NULL,
  PRIMARY KEY ('id'),
  KEY 'mood_user_id_index' ('user_id') USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

数据库表建好之后，生成对应的 Java Bean 对象，具体代码如下：

```
/**
 * 描述：微信说说实体
 * @author Ay
 * @date   2017/11/28.
 */
@Entity
@Table(name = "ay_mood")
public class AyMood implements Serializable {
    //主键
    @Id
    private String id;
    //说说内容
    private String content;
    //用户 id
    private String userId;
    //点赞数量
    private Integer praiseNum;
    //发表时间
    private Date publishTime;

    //省略 set、get 方法
}
```

AyMood 实体对象开发完成之后，开发对应的 AyMoodRepository 接口，具体代码如下：

```
/**
 * 描述：说说 repository
```

```
 * @author Ay
 * @date   2017/12/02
 */
public interface AyMoodRepository extends JpaRepository
<AyMood,String> {

}
```

Repository 接口开发完成之后，开发对应的说说服务层接口 AyMoodService 和相应的实现类 AyMoodServiceImpl。AyMoodService 具体代码如下：

```
/**
 * 描述：微信说说服务层
 * @author Ay
 * @date   2017/12/2.
 */
public interface AyMoodService {

    AyMood save(AyMood ayMood);
}
```

AyMoodService 代码很简单，只有一个保存说说的方法 save()，AyMoodService 开发完成之后，实现该接口，具体代码如下：

```
/**
 * 描述：微信说说服务层
 * @author Ay
 * @date   2017/12/2
 */
@Service
public class AyMoodServiceImpl implements AyMoodService{

    @Resource
    private AyMoodRepository ayMoodRepository;

    @Override
    public AyMood save(AyMood ayMood) {
        return ayMoodRepository.save(ayMood);
    }
```

```
}
```

在实现类 AyMoodServiceImpl 中注入 AyMoodRepository 接口，并调用其提供的 sava() 方法，保存说说到数据库。代码开发完成之后，在测试类 MySpringBootApplicationTests 下添加测试方法：

```
@Resource
private AyMoodService ayMoodService;

@Test
public void testAyMood(){
    AyMood ayMood = new AyMood();
    ayMood.setId("1");
    //用户阿毅 id 为 1
    ayMood.setUserId("1");
    ayMood.setPraiseNum(0);
    //说说内容
    ayMood.setContent("这是我的第一条微信说说！！！");
    ayMood.setPublishTime(new Date());
    //往数据库保存一条数据，代码用户阿毅发表了一条说说
    AyMood mood = ayMoodService.save(ayMood);
}
```

测试用例 testAyMood 开发完成之后，我们运行它。运行成功之后，可以在数据库表 ay_mood 中看到一条记录，如图 11-6 所示。

开始事务　备注　筛选　排序　导入　导出

id	content	user_id	praise_num	publish_time
1	这是我的第一天微信说说！	1	0	2017-12-02 13:46:42

图 11-6　说说表记录

用户发表说说的类虽然开发完成了，但是有一个问题，我们知道微信的用户量极大，每天都有几亿的用户发表不同的说说，如果按照上面的做法，用户每发一条说说，后端就单独开一个线程，将该说说的内容实时保存到数据库中。而后端服务系统的线程数和数据库线程池中的线程数量都是固定而且宝贵的，这样必然为后端服务和数据库带来极大的压力。所以我们使用 ActiveMQ 做异步消费来抵抗用户量大及发表说说而产生的压力，提高系统整体的性能。

下面我们来开发生产者和消费者。生产者 AyMoodProducer 的代码如下：

```
/**
 * 生产者
 * @author Ay
 * @date 2017/11/30
 */
@Service
public class AyMoodProducer{

    @Resource
    private JmsMessagingTemplate jmsMessagingTemplate;

    public void sendMessage(Destination destination,
    final String message) {
        jmsMessagingTemplate.convertAndSend(destination, message);
    }

}
```

- JmsMessagingTemplate：发消息的工具类，也可以注入 JmsTemplate，JmsMessagingTemplate 对 JmsTemplate 进行了封装。参数 destination 是发送到队列的，message 是待发送的消息。

11.3.2 创建消费者

生产者 AyMoodProducer 开发完成之后，我们来开发消费者 AyMoodConsumer，具体代码如下：

```
/**
 * 消费者
 * @author Ay
 * @date   2017/11/30
 */
@Component
public class AyMoodConsumer{

    @JmsListener(destination = "ay.queue")
```

```
    public void receiveQueue(String text) {
        System.out.println("用户发表说说【" + text + "】成功");
    }
}
```

- @JmsListener：使用 JmsListener 配置消费者监听的队列 ay.queue，其中 text 是接收到的消息。

11.3.3 测试

生产者和消费者开发完成之后，我们在测试类 MySpringBootApplicationTests 下开发测试方法 testAyMood()，具体代码如下：

```
@Resource
private AyMoodProducer ayMoodProducer;

@Test
public void testActiveMQ() {

    Destination destination = new ActiveMQQueue("ay.queue");
    ayMoodProducer.sendMessage(destination, "hello,mq!!!");
}
```

测试方法开发完成之后，运行测试方法，我们可以在控制台看到打印的信息，如图 11-7 所示。同时，可以在浏览器中访问 http://localhost:8161/admin/，查看队列 ay.queue 的消费情况，如图 11-8 所示。

```
[14:59:02:375] [TRACE] - org.springframework.transaction.support.TransactionSy
[14:59:02:375] [DEBUG] - org.springframework.jms.listener.adapter.MessagingMes
[14:59:02:375] [TRACE] - org.springframework.messaging.handler.invocation.Invo
用户发表说说【hello,mq!!!】成功
[14:59:02:375] [TRACE] - org.springframework.messaging.handler.invocation.Invo
[14:59:02:375] [TRACE] - org.springframework.jms.listener.adapter.MessagingMes
[14:59:02:375] [TRACE] - org.springframework.transaction.support.TransactionSy
```

图 11-7　控制台打印的信息

Name ↑	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequ
ay	0	0	0	0
ay.queue	0	0	10	10
myQueue	0	0	0	0
mytest.queue	10	0	0	0

图 11-8 ay.queue 的消费情况

生产者和消费者开发完成之后，现在我们把用户发表说说改成异步消费模式。首先在 AyMoodService 类下添加异步保存接口 asynSave()，具体代码如下：

```
/**
 * 描述：微信说说服务层
 * @author Ay
 * @date   2017/12/2.
 */
public interface AyMoodService {
    AyMood save(AyMood ayMood);
    String asynSave(AyMood ayMood);
}
```

然后在类 AyMoodServiceImpl 下实现 asynSave 方法，asynSave 方法并不保存说说记录，而是调用 AyMoodProducer 类的 sendMessage 推送消息，完整代码如下：

```
/**
 * 描述：微信说说服务层
 * @author Ay
 * @date   2017/12/2
 */
@Service
public class AyMoodServiceImpl implements AyMoodService{

    @Resource
    private AyMoodRepository ayMoodRepository;
    @Override
```

```
    public AyMood save(AyMood ayMood) {
        return ayMoodRepository.save(ayMood);
    }
    //队列
    private static Destination destination = new ActiveMQQueue
    ("ay.queue.asyn.save");

    @Resource
    private AyMoodProducer ayMoodProducer;
    @Override
    public String asynSave(AyMood ayMood){
        //往队列 ay.queue.asyn.save 推送消息，消息内容为说说实体
        ayMoodProducer.sendMessage(destination, ayMood);
        return "success";
    }
}
```

其次，在 AyMoodProducer 生产者类下添加 sendMessage(Destination destination, final AyMood ayMood) 方法，消息内容是 ayMood 实体对象。AyMoodProducer 生产者的完整代码如下：

```
/**
 * 生产者
 * @author Ay
 * @date 2017/11/30
 */
@Service
public class AyMoodProducer {

    @Resource
    private JmsMessagingTemplate jmsMessagingTemplate;

    public void sendMessage(Destination destination,
    final String message) {
        jmsMessagingTemplate.convertAndSend(destination, message);
    }
```

```
    public void sendMessage(Destination destination,
    final AyMood ayMood) {
        jmsMessagingTemplate.convertAndSend(destination, ayMood);
    }
}
```

最后，修改 AyMoodConsumer 消费者，在 receiveQueue 方法中保持说说记录，完整代码如下：

```
/**
 * 消费者
 * @author Ay
 * @date   2017/11/30
 */
@Component
public class AyMoodConsumer {

    @JmsListener(destination = "ay.queue")
    public void receiveQueue(String text) {
        System.out.println("用户发表说说【" + text + "】成功");
    }

    @Resource
    private AyMoodService ayMoodService;

    @JmsListener(destination = "ay.queue.asyn.save")
    public void receiveQueue(AyMood ayMood){
        ayMoodService.save(ayMood);
    }
}
```

用户发表说说，异步保存所有代码开发完成之后，在测试类 MySpringBootApplicationTests 下添加 testActiveMQAsynSave 测试方法，具体代码如下：

```
@Test
public void testActiveMQAsynSave() {
    AyMood ayMood = new AyMood();
    ayMood.setId("2");
```

```
        ayMood.setUserId("2");
        ayMood.setPraiseNum(0);
        ayMood.setContent("这是我的第一条微信说说！！！");
        ayMood.setPublishTime(new Date());
        String msg = ayMoodService.asynSave(ayMood);
        System.out.println("异步发表说说 :" + msg);

    }
```

运行测试方法 testActiveMQAsynSave，成功之后，我们可以在数据库表 ay_mood 中查询到用户（id 为 2）发表的记录，具体如图 11-9 所示。

开始事务　备注　筛选　排序　导入　导出

id	content	user_id	praise_num	publish_time
1	这是我的第一天微信说说！	1	0	2017-12-02 13:46:42
2	这是我的第一条微信说说！	2	0	2017-12-02 15:27:30

图 11-9　ay_mood 表记录

11.4　Spring Boot 异步调用

11.4.1　异步调用介绍

异步调用是相对于同步调用而言的，同步调用是指程序按预定顺序一步步执行，每一步必须等到上一步执行完成之后才能执行，而异步调用则无须等待上一步程序执行完成即可执行。在日常开发的项目中，当访问的接口较慢或者做耗时任务时，不想程序一直卡在耗时任务上，想让程序能够并行执行，我们除了可以使用多线程来并行地处理任务，也可以使用 Spring Boot 提供的异步处理方式 @Async 来处理。在 Spring Boot 框架中，只要提过 @Async 注解就能将普通的同步任务改为异步调用任务。

11.4.2　@Async 的使用

使用 @Async 注解之前，我们需要在入口类添加注解 @EnableAsync 开启异步调

用，具体代码如下：

```
@SpringBootApplication
@ServletComponentScan
@ImportResource(locations={"classpath:spring-mvc.xml"})
@EnableAsync
public class MySpringBootApplication {

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

然后，修改 AyUserServiceImpl 类的 findAll 方法，使它能够记录代码执行完成所花费的时间，具体代码如下：

```
@Override
public List<AyUser> findAll() {
    try{
        System.out.println("开始做任务");
        long start = System.currentTimeMillis();
        List<AyUser> ayUserList = ayUserRepository.findAll();
        long end = System.currentTimeMillis();
        System.out.println("完成任务,耗时: " + (end - start) + "毫秒");
        return ayUserList;
    }catch (Exception e){
        logger.error("method [findAll] error",e);
        return Collections.EMPTY_LIST;
    }
}
```

11.4.3 测试

AyUserServiceImpl 类的方法 findAll() 开发完成之后，在 MySpringBootApplicationTests 测试类下开发测试方法 testAsync()，该方法调用 3 次 findAll()，并记录总共消耗的时间，由于现在是同步调用，因此代码按照顺序一步一步执行。testAsync 方法的具体代码如下：

```
@Test
public void testAsync(){
    long startTime = System.currentTimeMillis();
    System.out.println(" 第一次查询所有用户！ ");
    List<AyUser> ayUserList = ayUserService.findAll();
    System.out.println(" 第二次查询所有用户！ ");
    List<AyUser> ayUserList2 = ayUserService.findAll();
    System.out.println(" 第三次查询所有用户！ ");
    List<AyUser> ayUserList3 = ayUserService.findAll();
    long endTime = System.currentTimeMillis();
    System.out.println("总共消耗 " + (endTime - startTime) + "毫秒");
}
```

测试方法 testAsync() 开发完成之后，我们运行它，运行成功之后，可以在控制台看到如下打印信息：

```
第一次查询所有用户！
开始做任务
完成任务，耗时：371 毫秒
第二次查询所有用户！
开始做任务
完成任务，耗时：34 毫秒
第三次查询所有用户！
开始做任务
完成任务，耗时：32 毫秒
总共消耗：438 毫秒
```

从打印结果可以看出，调用 3 次 findAll() 总共消耗 438 毫秒。现在我们在 AyUserService 接口中添加异步查询方法 findAsynAll()，并在 AyUserServiceImpl 类中实现该方法，具体代码如下：

```
/**
 * 描述：用户服务层接口
 * @author 阿毅
 * @date   2017/10/14
 */
public interface AyUserService {
```

```
    //省略大量代码

    List<AyUser> findAll();
    //异步查询
    Future<List<AyUser>> findAsynAll();
}
```

在 AyUserServiceImpl 类中实现 findAsynAll() 方法，并在方法中添加异步调用注解 @Async，具体代码如下：

```
@Override
@Async
public Future<List<AyUser>> findAsynAll() {
    try{
        System.out.println("开始做任务");
        long start = System.currentTimeMillis();
        List<AyUser> ayUserList = ayUserRepository.findAll();
        long end = System.currentTimeMillis();
        System.out.println("完成任务,耗时: " + (end - start) + "毫秒");
        return new AsyncResult<List<AyUser>>(ayUserList) ;
    }catch (Exception e){
        logger.error("method [findAll] error",e);
        return new AsyncResult<List<AyUser>>(null);
    }
}
```

findAsynAll() 方法开发完成之后，在 MySpringBootApplicationTests 测试类下开发测试方法 testAsync2()，具体代码如下：

```
@Test
public void testAsync2()throws Exception{
    long startTime = System.currentTimeMillis();
    System.out.println("第一次查询所有用户！");
    Future<List<AyUser>> ayUserList = ayUserService.findAsynAll();
    System.out.println("第二次查询所有用户！");
    Future<List<AyUser>> ayUserList2 = ayUserService.findAsynAll();
    System.out.println("第三次查询所有用户！");
```

```
        Future<List<AyUser>> ayUserList3 = ayUserService.findAsynAll();
        while (true){
            if(ayUserList.isDone() && ayUserList2.isDone() &&
            ayUserList3.isDone()){
                break;
            }else {
                Thread.sleep(10);
            }
        }
        long endTime = System.currentTimeMillis();
        System.out.println("总共消耗 " + (endTime - startTime) + "毫秒");
    }
```

测试方法 testAsync2() 开发完成之后，我们运行它，运行成功之后，可以在控制台看到如下打印信息：

```
第一次查询所有用户！
第二次查询所有用户！
第三次查询所有用户！
开始做任务
开始做任务
开始做任务
完成任务，耗时：316 毫秒
完成任务，耗时：316 毫秒
完成任务，耗时：315 毫秒
总共消耗：334 毫秒
```

从上面的打印结果可以看出，testAsync2 方法的执行速度比 testAsync 方法快 104（438 — 334）毫秒。由此说明，异步调用的速度比同步调用快。

第 12 章

全局异常处理与 Retry 重试

本章主要介绍 Srping Boot 全局异常使用、自定义错误页面、全局异常类开发、Retry 重试机制的介绍与使用等。

12.1 全局异常介绍

使用 Web 应用时，在请求处理过程中发生错误是非常常见的情况。Spring Boot 为我们提供了一个默认的映射：/error，当处理中抛出异常之后，会转到该请求中处理，并且该请求有一个全局的错误页面用来展示异常内容。比如现在启动 my-spring-boot 项目（启动项目之前，记得启动 Redis 服务和 ActiveMQ 服务），项目启动完成之后，在浏览器中随便输入一个访问地址，比如 http://localhost:8080/ayUser/testdddd，由于地址不存在，Spring Boot 会跳转到错误页面，如图 12-1 所示。

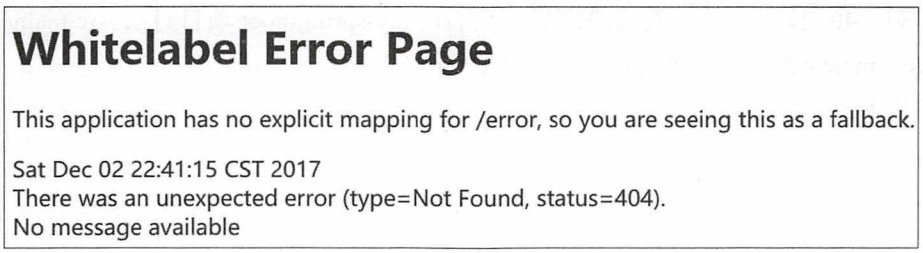


图 12-1　错误页面

虽然 Spring Boot 为我们提供了默认的错误页面映射，但是在实际应用中，图 12-1 的错误页面对用户来说并不友好，我们需要自己来实现异常提示。

12.2 Spring Boot 全局异常使用

12.2.1 自定义错误页面

12.1 节中，我们已经知道 Spring Boot 的错误提示页面的用户体验不好，这一节将自己实现错误提示页面。首先，在 my-spring-boot 项目目录 /src/main/resources/static 下新建自定义错误页面 404.html，具体的代码如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<div class="text" style=" text-align:center;">
    主人，我累了，让我休息一会 !!!
</div>
</body>
</html>
```

错误页面的内容很简单，就是当访问路径不存在时，在页面中间显示一句话：“主人，我累了，让我休息一会 !!!”。当然，在真正的项目中，该错误页面的样式会更

加美观。404 错误页面开发完成之后，我们在 my-spring-boot 项目目录 /src/main/java/com.example.demo 下新建包 error，并在 error 包下新建 ErrorPageConfig 配置类，具体代码如下：

```
/**
 * 描述：自定义错误页面
 * @author Ay
 * @date   2017/12/02
 */
@Configuration
public class ErrorPageConfig {

    @Bean
    public EmbeddedServletContainerCustomizer containerCustomizer() {

        return new EmbeddedServletContainerCustomizer() {
            @Override
            public void customize(ConfigurableEmbeddedServletContainer
            container) {
              ErrorPage error404Page = new ErrorPage(HttpStatus.
              NOT_FOUND, "/404.html");
              container.addErrorPages(error404Page);
            }
        };
    }

}
```

- EmbeddedServletContainerCustomizer：Spring Boot 的自动配置有一个特性就是能够通过代码来修改配置，这样可以很方便地修改配置，而我们只需要实现 Spring Boot 定义的接口即可。这里需要注册一个实现 EmbeddedServletContainerCustomizer 的 Bean，在 ErrorPageConfig 类中，我们使用匿名类来实现 EmbeddedServletContainerCustomizer 接口，同时实现该接口唯一的方法 customize，并在 customize 方法中自定义 401、404、500 等错误页面。

12.2.2 测试

404.html 错误页面和 ErrorPageConfig 类开发完成之后，重新启动 my-spring-boot 项目（项目启动之前，记得启动 Redis 缓存服务和 ActiveMQ 服务，否则项目会报错，之后不再提示），在浏览器中输入链接 http://localhost:8080/ayUser/testdddd，由于该链接不存在，因此会出现如图 12-2 所示的自定义错误页面。

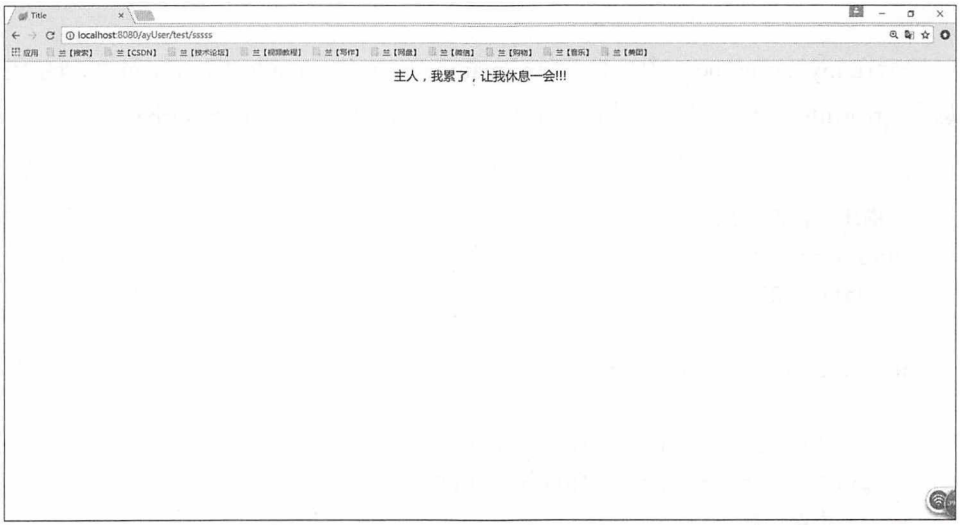


图 12-2　自定义 404 错误界面

12.2.3 全局异常类开发

在项目中，我们会遇到各种各样的业务异常，业务异常是指正常的业务处理时，由于某些业务的特殊要求而导致处理不能继续而抛出异常。我们希望这些业务异常能够被统一处理，因此使用 Spring Boot 进行全局异常处理就变得很方便。首先，统一封装自定义业务异常 BusinessException，该类继承自 RuntimeException 异常类，并提供带有异常信息的构造方法，具体代码如下：

```
/**
 * 描述：业务异常
 * @author Ay
 * @date   2017/12/3
 */
```

```
public class BusinessException extends RuntimeException{

    public BusinessException(){}

    public BusinessException(String message) {
        super(message);
    }
}
```

然后在 my-spring-boot 项目目录 /src/main/java/com.example.demo.error 下新建错误信息类 ErrorInfo，该类用于封装错误信息，包括错误码，具体代码如下：

```
/**
 * 描述：错误信息类
 *@author Ay
 * @date  2017/12/3
 */
public class ErrorInfo<T> {

    public static final Integer SUCCESS = 200;
    public static final Integer ERROR = 100;
    //错误信息
    private Integer code;
    //错误码
    private String message;
    private String url;
    private T data;
    //省略 set、get 方法
}
```

其次，在 my-spring-boot 项目目录 /src/main/java/com.example.demo.error 下新建统一异常处理类 GlobalDefaultExceptionHandler，具体代码如下：

```
/**
 * 描述：统一业务异常处理类
 * @author Ay
 * @date   2017/12/3
 */
```

```
@ControllerAdvice(basePackages={"com.example.demo",})
public class GlobalDefaultExceptionHandler {

    @ExceptionHandler({BusinessException.class})
    // 如果返回的为 json 数据或其他对象，就添加该注解
    @ResponseBody
    public ErrorInfo defaultErrorHandler(HttpServletRequest req,
    Exception e)
                                              throws Exception {
        ErrorInfo errorInfo = new ErrorInfo();
        errorInfo.setMessage(e.getMessage());
        errorInfo.setUrl(req.getRequestURI());
        errorInfo.setCode(ErrorInfo.SUCCESS);
        return errorInfo;
    }

}
```

- @ControllerAdvice：定义统一的异常处理类，basePackages 属性用于定义扫描哪些包，默认可不设置。
- @ExceptionHandler：用来定义函数针对的异常类型，可以传入多个需要捕获的异常类。
- @ResponseBody：如果返回的为 json 数据或其他对象，就添加该注解。

最后，在 AyUserController 类下添加控制层方法 findAll，并在方法里抛出 BusinessException，该异常会被全局异常类捕获到，具体代码如下：

```
/**
 * 描述：用户控制层
 * @author Ay
 * @date   2017/10/28.
 */
@Controller
@RequestMapping("/ayUser")
public class AyUserController {

    @Resource
```

```
    private AyUserService ayUserService;

    @RequestMapping("/findAll")
    public String findAll(Model model) {
        List<AyUser> ayUser = ayUserService.findAll();
        model.addAttribute("users",ayUser);
        throw new BusinessException(" 业务异常 ");
    }

    // 省略大量代码

}
```

12.2.4 测试

代码开发完成之后，重新启动 my-spring-boot 项目，项目启动成功之后，在浏览器中输入访问地址：http://localhost:8080/ayUser/findAll，可以看到后端返回的 json 信息，具体信息如下：

```
{"code":200,"message":" 业务异常 ","url":"/ayUser/findAll", "data":null}
```

12.3 Retry 重试机制

12.3.1 Retry 重试介绍

当我们调用一个接口时，可能由于网络等原因造成第一次失败，再去尝试就成功了，这就是重试机制。重试的解决方案有很多，比如利用 try-catch-redo 简单重试模式，通过判断返回结果或监听异常来判断是否重试，具体可以看如下的例子：

```
public void testRetry() throws Exception{
    boolean result = false;
    try{
        result = load();
        if(!result){
            load();// 一次重试
```

```
        }
    }catch (Exception e){
        load();// 一次重试
    }
}
```

try-catch-redo 重试模式还有可能出现重试无效，解决这个问题的方法是尝试增加重试次数 retrycount 以及重试间隔周期 interval，达到增加重试有效的可能性，因此我们可以利用 try-catch-redo-retry strategy 策略重试模式。具体代码如下：

```
public void testRetry2() throws Exception{
    boolean result = false;
    try{
        result = load();
        if(!result){
            // 延迟 3s，重试 3 次
            reLoad(3000L,3);// 延迟多次重试
        }
    }catch (Exception e){
        // 延迟 3s，重试 3 次
        reLoad(3000L,3);// 延迟多次重试
    }
}
```

但是这两种策略有一个共同的问题就是：正常逻辑和重试逻辑强耦合。基于这些问题，对于 Spring-Retry 规范正常逻辑和重试逻辑，将正常逻辑和重试逻辑解耦。Spring-Retry 是一个开源工具包，该工具把重试操作模板定制化，可以设置重试策略和回退策略。同时，重试执行实例保证线程安全。Spring-Retry 重试可以用 Java 代码方式实现，也可以用注解 @Retryable 方式实现，这里 Spring-Retry 提倡以注解的方式对方法进行重试。

12.3.2 Retry 重试机制的使用

使用 Spring 提供的重试策略之前，首先需要在 pom.xml 文件中引入所需的依赖，具体代码如下：

```
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
</dependency>
```

依赖添加完成之后，我们需要在入口类 MySpringBootApplication 中添加注解 @EnableRetry 开启 Retry 重试。完整代码如下：

```
@SpringBootApplication
@ServletComponentScan
@ImportResource(locations={"classpath:spring-mvc.xml"})
@EnableAsync
// 开启 retry 重试
@EnableRetry
public class MySpringBootApplication {

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

然后，在 AyUserService 类下添加新接口 findByNameAndPasswordRetry，具体代码如下：

```
AyUser findByNameAndPasswordRetry(String name, String password);
```

接口 findByNameAndPasswordRetry 添加完成之后，我们在 AyUserServiceImpl 类下实现接口 findByNameAndPasswordRetry，并在方法中故意抛出业务异常 BusinessException，具体代码如下：

```
@Override
@Retryable(value= {BusinessException.class},maxAttempts = 5,
backoff = @Backoff(delay = 5000,multiplier = 2))
```

```
public AyUser findByNameAndPasswordRetry(String name,
String password) {
    System.out.println("[findByNameAndPasswordRetry] 方法失败重试了！ ");
    throw new BusinessException();
}
```

- @Retryable：value 属性表示当出现哪些异常的时候触发重试，maxAttempts 表示最大重试次数默认为 3，delay 表示重试的延迟时间，multiplier 表示上一次延时时间是这一次的倍数。

最后，我们在 AyUserController 类下添加控制层方法 findByNameAndPasswordRetry，在该方法中调用服务层 AyUserServiceImpl 的方法 findByNameAndPasswordRetry。具体代码如下：

```
@RequestMapping("/findByNameAndPasswordRetry")
public String findByNameAndPasswordRetry(Model model) {
    AyUser ayUser = ayUserService.findByNameAndPasswordRetry
    ("阿毅","123456");
    model.addAttribute("users",ayUser);
    return "success";
}
```

12.3.3 测试

代码开发完成之后，重新启动 my-spring-boot 项目，项目运行成功之后，在浏览器中输入访问地址：http://localhost:8080/ayUser/findByNameAndPasswordRetry，由于方法 findByNameAndPasswordRetry 会抛出 BusinessException 异常，故 Retry 重试机制会检测到：进行第 2 次重试，重试成功，方法执行完成，重试失败，按照配置延迟 delay 时间，依次进行第 3 次、第 4 次、第 5 次重试，直到重试成功或者达到最大重试次数，重试策略终止。我们可以在 IDEA 的控制台多次看到如下打印信息：

```
[findByNameAndPasswordRetry] 方法失败重试了！
```

第 13 章

集成 MongoDB 数据库

本章主要介绍如何安装和使用 MongoDB 数据库、NoSQL Manager for MongoDB 客户端的安装与使用以及在 Spring Boot 中集成 MongoDB 数据库开发简单的功能等。

13.1 MongoDB 数据库介绍

13.1.1 MongoDB 概述

MongoDB 是一个高性能、开源、无模式的文档型数据库，是当前 NoSQL 数据库中比较热门的一种，在企业中被广泛使用。其主要功能特性有：面向集合存储、易存储对象类型的数据、支持动态查询、文件存储格式为 BSON（一种 JSON 的扩展）、支持复制和故障恢复等。MongoDB 非常适合实时地插入、更新与查询，并具备网站

实时数据存储所需的复制及高度伸缩性。由于性能很高，因此 MongoDB 也适合作为信息基础设施的缓存层。在系统重启之后，由 MongoDB 搭建的持久化缓存层可以避免下层的数据源过载。由于高伸缩性，因此 MongoDB 也非常适合由数十或数百台服务器组成的数据库。MongoDB 的路线图中已经包含对 MapReduce 引擎的内置支持，用于对象及 JSON 数据的存储方式，MongoDB 的 BSON 数据格式非常适合文档化格式的存储及查询。MongoDB 有很多优点，但缺点也是明显的，比如不能建立实体关系、没有事务管理机制等。

13.1.2 MongoDB 的安装

MongoDB 提供有 Windows、Linux、OSX、Solaris 等操作系统的安装包。本书主要针对 Windwos 操作系统进行讲解，具体安装步骤如下：

步骤 01　在官方网站（https://www.mongodb.com/download-center#community）根据操作系统的位数下载对应的安装包，如图 13-1 所示。

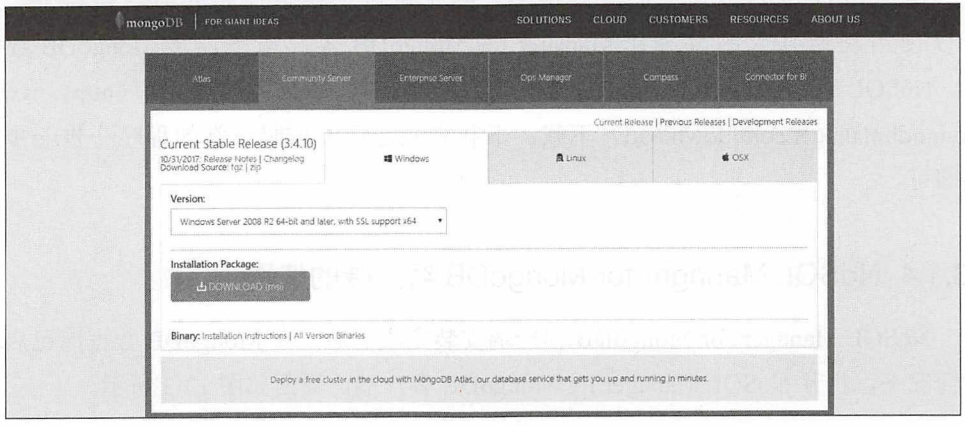


图 13-1　MongoDB 官方网站

步骤 02　双击下载的安装包进行安装。这里安装在 C 盘。

步骤 03　找到安装目录的 bin 路径，将其配置在 Windows 的环境变量 path 中，如 C:\Program Files\MongoDB\Server\3.4\bin。

步骤 04　创建一个保存数据库的目录，如 C:\mongodb\data，然后打开一个命令窗口，输入命令：mongod - - dbpath = C:\mongodb\data，启动 MongoDB 服务。

步骤 05　我们可以在命令行看到如图 13-2 所示的信息，代表 MongoDB 安装成功。

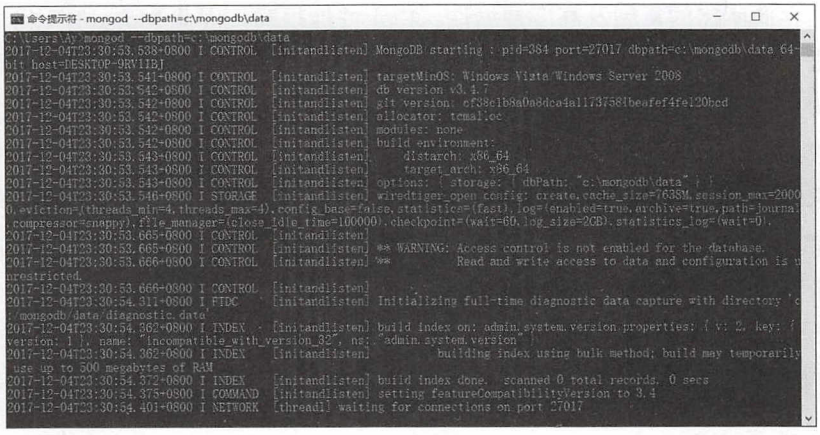


图 13-2 MongoDB 安装成功信息

13.1.3 NoSQL Manager for MongoDB 客户端介绍

连接 MongoDB 数据库的方式很多，除了可以使用最原始的命令窗口外，还可以使用功能强大的 NoSQL Manager for MongoDB 客户端来连接 MongoDB 数据库。NoSQL Manager for MongoDB 客户端安装包可以到官方网站（https://www.mongodbmanager.com/download）下载，下载完成之后，按照正常的程序一步一步安装即可。

13.1.4 NoSQL Manager for MongoDB 客户端的使用

NoSQL Manager for MongoDB 客户端安装完成之后，在 MongoDB 数据库已启动的情况下，打开 NoSQL Manager for MongoDB 客户端，界面如图 13-3 所示。

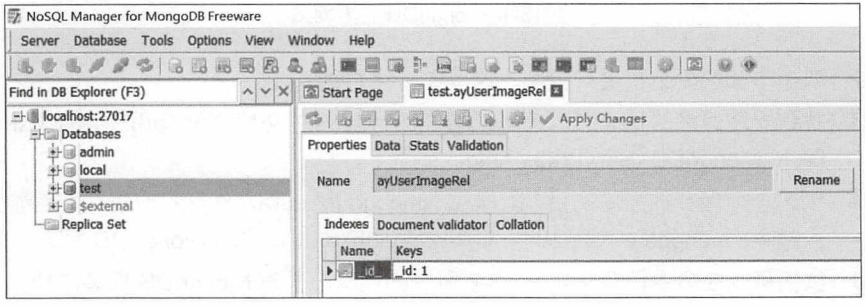


图 13-3 NoSQL Manager for MongoDB 界面

从图 13-3 中可以看出，MongoDB 数据库安装完成之后，默认创建了 3 个数据库，分别为 admin、local、test。单击 test 数据库，然后单击菜单栏的 shell 按钮，可以打开 shell 窗口，在 shell 窗口中可以编写相关的 SQL 语句，如图 13-4 所示。

在 shell 窗口中输入 SQL 语句：show dbs，单击执行按钮，可以在 Result 结果界面看到 SQL 语句执行的结果，如图 13-5 所示。

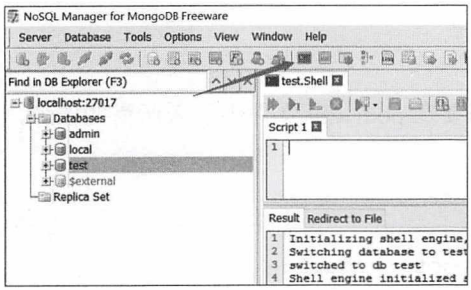


图 13-4　单击 shell 按钮

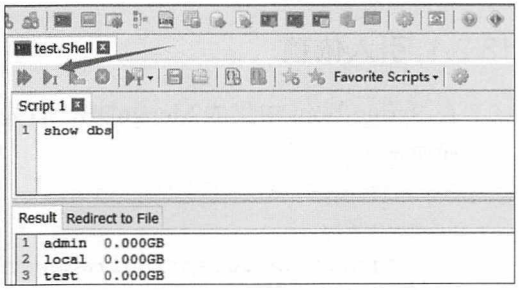


图 13-5　SQL 语句执行结果界面

我们还可以在 shell 窗口中编写命令练习 MongoDB 的一些 SQL 语法，具体代码如下：

```
// 显示所有的数据库
> show dbs;
// 切换到 test 数据库
> use test;
// 创建一个名为 user 的集合
> db.user.insert({"id":"1","name":"ay","age":"26"});
// 查询 user 集合
> db.user.find();
// 将集合 user 名称为 ay 的记录更新为名称 al
> db.user.update({"id":"1"},{"id":"1","name":"al","age":"26"});
// 查询 user 集合
> db.user.find();
// 显示所有集合
> show collections;
// 删除集合 user 名称为 al 的记录
> db.user.remove({"name":"al"});
// 查询 user 集合
> db.user.find();
```

上面的代码只是一个简单的 SQL 语句练习，更多关于 MongoDB 的 SQL 语句练习可在网上查阅学习。

13.2 集成 MongoDB

13.2.1 引入依赖

在 Spring Boot 中集成 MongoDB，首先需要在 pom.xml 文件中引入所需的依赖，具体代码如下：

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

13.2.2 添加 MongoDB 配置

在 pom 文件中引入 MongoDB 所需的依赖之后，我们需要在 application.properties 文件中添加如下配置信息：

```
###mongodb 配置
###host 地址
spring.data.mongodb.host=localhost
### 默认数据库端口 27017
spring.data.mongodb.port=27017
### 连接数据库名 test
spring.data.mongodb.database=test
```

13.2.3 连接 MongoDB

首先，在 my-spring-boot 项目目录 /src/main/java/com.example.demo.model 下新建用户附件类 AyUserAttachmentRel，具体代码如下：

```
/**
 * 描述：用户头像关联表
 * @author  Ay
```

```
 * @date    2017/12/4.
 */
public class AyUserAttachmentRel {

    @Id
    private String id;
    private String userId;
    private String fileName;
    // 省略 set、get 方法
}
```

用户附件类 AyUserAttachmentRel 开发完成之后，我们开发 AyUserAttachmentRelRepository 类，该类继承自 MongoRepository 类，MongoRepository 类在 spring-data-mongodb 包下，类似于第 3 章讲的 Spring Data JPA。追溯 MongoRepository 源代码可以看出，MongoRepository 最顶级的父类就是 Repository 接口。AyUserAttachmentRelRepository 类的具体代码如下：

```
/**
 * 描述：用户附件 Repository
 * @author Ay
 * @date   2017/12/4.
 */
public interface AyUserAttachmentRelRepository
        extends MongoRepository<AyUserAttachmentRel,String> {

}
```

AyUserAttachmentRelRepository 类很简单，只是纯粹的继承 MongoRepository，就能继承 MongoRepository 为我们提供的增删改查等方法。AyUserAttachmentRelRepository 开发完成之后，我们开发服务层接口 AyUserAttachmentRelService 类，在 AyUserAttachmentRelService 类中声明 save 方法，用来简单保存数据，具体代码如下：

```
/**
 * 描述：用户附件服务层
 * @author Ay
 * @date   2017/12/4.
 */
```

```
public interface AyUserAttachmentRelService {

    AyUserAttachmentRel save(AyUserAttachmentRel ayUserAttachmentRel);

}
```

接口 AyUserAttachmentRelService 类开发完成之后，接下来开发其对应的实现类 AyUserAttachmentRelServiceImpl，在 AyUserAttachmentRelServiceImpl 中实现接口层方法 save，注入 AyUserAttachmentRelRepository 类，并调用 AyUserAttachmentRelRepository 的 save 方法将数据保存到 MongoDB 数据库中。具体代码如下：

```
/**
 * 描述：用户附件实现层
 * @author  Ay
 * @date    2017/12/4.
 */
@Service
public class AyUserAttachmentRelServiceImpl implements
AyUserAttachmentRelService {

    @Resource
    private AyUserAttachmentRelRepository
    ayUserAttachmentRelRepository;

    public AyUserAttachmentRel save(AyUserAttachmentRel
    ayUserAttachmentRel){
        return ayUserAttachmentRelRepository.
        save(ayUserAttachmentRel);
    }
}
```

13.2.4 测试

所有代码开发完成之后，我们在测试类 MySpringBootApplicationTests 下添加测试方法 testMongoDB，在方法中创建 AyUserAttachmentRel 实体，并调用 ayUserAttachmentRelService 类中的 save 方法，将数据存储到 MongoDB 中，具体代码如下：

```
@Resource
private AyUserAttachmentRelService ayUserAttachmentRelService;

@Test
public void testMongoDB(){
    AyUserAttachmentRel ayUserAttachmentRel =
    new AyUserAttachmentRel();
    ayUserAttachmentRel.setId("1");
    ayUserAttachmentRel.setUserId("1");
    ayUserAttachmentRel.setFileName(" 个人简历 .doc");
    ayUserAttachmentRelService.save(ayUserAttachmentRel);
    System.out.println(" 保存成功 ");
}
```

运行测试方法 testMongoDB，运行前记得启动 MongoDB 数据库，测试方法运行成功之后，我们可以在 MongoDB 数据库中查询到数据。具体查询的 SQL 语句如下：

```
> use test;
> db.ayUserImageRel.find();
```

查询结果：

```
{ "_id" : "1", "_class" : "com.example.demo.model.
AyUserImageRel", "userId" : "1", "fileName" : " 个人简历 .doc" }
```

第14章

集成 Spring Security

本章主要介绍 Spring Security 基础知识、Spring Boot 如何集成 Spring Security、利用 Spring Security 实现授权登录以及利用 Spring Boot 实现数据库数据授权登录等。

14.1 Spring Security 介绍

在 Web 应用开发中，安全是非常重要的。因为安全属于应用的非功能性需求，大部分企业会把更多资源投入应用开发中，所以应用安全很容易被忽略。安全应该在应用开发的初期就考虑进来，如果在应用开发的后期才考虑安全问题，就可能陷入两难的境地：一方面，应用存在严重的安全漏洞，无法满足用户的要求，并可能造成用户的隐私数据被攻击者窃取；另一方面，应用的基本架构已经确定，要修复安全漏洞，可能需要对系统的架构做出比较重大的调整，因而需要更多开发时间，影响应用的发

布进程。因此，从应用开发的第一天就应该把安全相关的因素考虑进来。

市场上开源的安全框架很多，比如 Apache Shiro 安全框架、Spring Security 安全框架等。虽然 Spring Security 安全框架比 Apache Shiro 安全框架“重”，但是如果我们用心去了解 Spring Security 安全框架，就会发现其实 Spring Security 安全框架是非常优秀的。本书选择在 Spring Boot 中集成 Spring Security 安全框架。

Spring Security 安全框架除了包含基本的认证和授权功能外，还提供了加密解密、统一登录等一系列支持。Spring Security 安全框架简单的实现原理如图 14-1 所示。

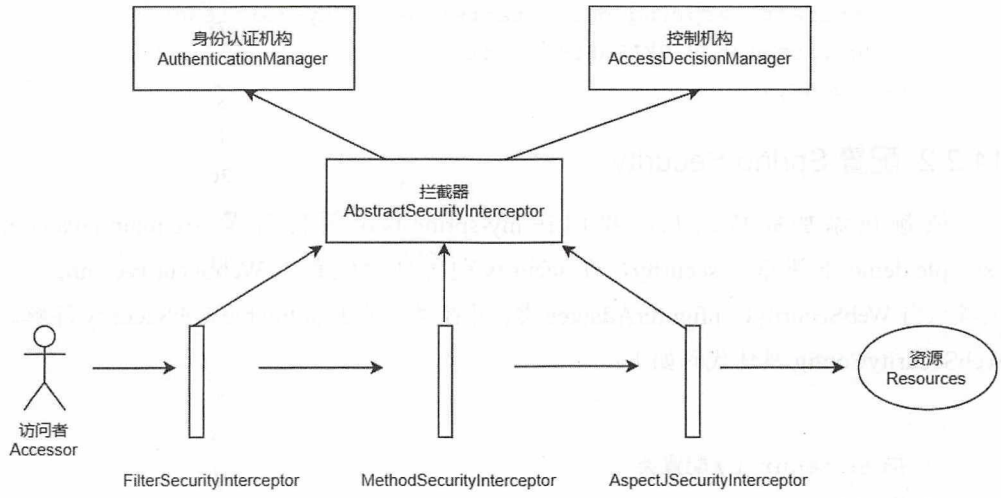


图 14-1　Spring Security 简单原理

在图 14-1 中，Accessor 是资源的访问者，在访问过程中，需要经过一系列拦截器 Interceptor 的拦截，比如 FilterSecurityInterceptor、MethodSecurityInterceptor、AspectJSecurityInterceptor 等。这些拦截器是统一的抽象类 AbstractSecurityInterceptor 的具体实现。“控制机构”AccessDecisionManager 决定谁可以访问资源，而“身份认证机构”AuthenticationManager 就是定义那个“谁”，解决的是访问者身份认证问题，只有确定注册类，才可以给予授权访问。“控制机构”AccessDecisionManager 和“身份认证机构”AuthenticationManager 负责制订规则，AbstractSecurityInterceptor 负责执行。

14.2 集成 Spring Security

14.2.1 引入依赖

在 Spring Boot 中集成 Spring Security，首先需要在 pom.xml 文件中引入所需的依赖，具体代码如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
    <version>1.5.9.RELEASE</version>
</dependency>
```

14.2.2 配置 Spring Security

依赖包添加完成之后，我们在 my-spring-boot 项目目录 /src/main/java/com.example.demo 下新建包 security，在 security 包下新建配置类 WebSecurityConfig，该类继承自 WebSecurityConfigurerAdapter 类，并在类上添加 @EnableWebSecurity 注解。WebSecurityConfig 具体代码如下：

```
/**
 * 描述：security 配置类
 * @author ay
 * @date   2017/12/10.
 */
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        // 路由策略和访问权限的简单配置
        http
                .formLogin()                    // 启用默认登录页面
                .failureUrl("/login?error")
```

```
                    // 登录失败返回 URL:/login?error
                .defaultSuccessUrl("/ayUser/test ")
                    // 登录成功跳转 URL，这里跳转到用户首页
                .permitAll();                    // 登录页面全部权限可访问
        super.configure(http);
    }

    /**
     * 配置内存用户
     */
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder
    auth) throws Exception {
        auth
            .inMemoryAuthentication()
            .withUser("阿毅").password("123456").roles("ADMIN")
            .and()
            .withUser("阿兰").password("123456").roles("USER");
    }
}
```

- @EnableWebSecurity：开启 Security 安全框架。
- configure 方法：WebSecurityConfig 继承 WebSecurityConfigurerAdapter 类需要重写 configure 方法，通过 formLogin 方法配置启用默认登录页面，通过 failureUrl 方法配置登录失败返回 URL，通过 defaultSuccessUrl 配置登录成功跳转 URL，这里调整到用户首页，通过 permitAll 方法设置登录页面全部权限可访问等。
- configureGlobal 方法：参数 AuthenticationManagerBuilder 类的方法 inMemoryAuthentication 可添加内存中的用户，并可给用户指定角色权限。比如上面的代码给用户阿毅分配了 ADMIN 权限，而给用户阿兰分配了 USER 权限。

14.2.3 测试

WebSecurityConfig 配置类开发完成后，重新启动 my-spring-boot 项目，项目启动成功后，在浏览器中输入访问链接：http://localhost:8080/ayUser/test，该访问请求会被

Security 框架拦截并跳转到默认的登录界面，如图 14-2 所示。在输入框中输入错误的用户名和密码：admin 和 123456，由于用户名和密码错误，Security 框架会跳转到之前在 WebSecurityConfig 类中配置的错误 URL:login?error 页面去，如图 14-3 所示。

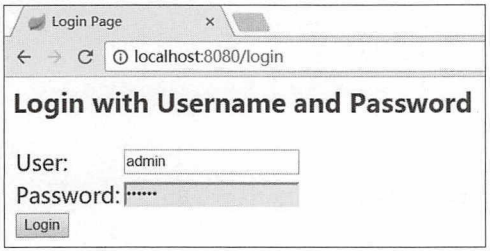


图 14-2 Security 登录界面

图 14-3 登录错误页面

14.2.4 数据库用户授权登录

在 14.2.2 节中，用户登录系统的用户名、密码以及角色都是“写死”在代码中，显然是不符合正常逻辑的。真正的项目都是通过查询数据库的用户名和密码进行用户认证和授权登录的。首先，我们需要在数据库中建立角色表 ay_role 和用户角色关联表 ay_user_role_rel，具体的建表语句如下：

```
-- 角色表
DROP TABLE IF EXISTS 'ay_role';
CREATE TABLE 'ay_role' (
  'id' varchar(255) DEFAULT NULL,
  'name' varchar(255) DEFAULT NULL
);
-- 用户角色关联表
DROP TABLE IF EXISTS 'ay_user_role_rel';
CREATE TABLE 'ay_user_role_rel' (
  'user_id' varchar(255) DEFAULT NULL,
  'role_id' varchar(255) DEFAULT NULL
);
```

角色表 ay_role 和用户角色关联表 ay_user_role_rel 建立完成之后，往表中插入数据：

```
-- 角色表 ay_role 数据
INSERT INTO 'ay_role' VALUES ('1', 'ADMIN');
INSERT INTO 'ay_role' VALUES ('2', 'USER');
-- 用户角色关联表 ay_user_role_rel
INSERT INTO 'ay_user_role_rel' VALUES ('1', '1');
INSERT INTO 'ay_user_role_rel' VALUES ('2', '2');
```

表 ay_role 和表 ay_user_role_rel 中的数据很简单，id 为 1 的用户拥有 ADMIN 角色，id 为 2 的用户拥有 USER 角色。数据插入完成之后，生成对应的实体类：AyRole 和 AyUserRoleRel，具体代码如下：

```
/**
 * 描述：用户角色实体
 * @author Ay
 * @date   2017/12/10.
 */
@Entity
@Table(name = "ay_role")
public class AyRole {

    @Id
    private String id;
    private String name;
    // 省略 set、get 方法
}

/**
 * 描述：用户角色关联
 * @author   Ay
 * @date     2017/12/10.
 */
@Entity
@Table(name = "ay_user_role_rel")
public class AyUserRoleRel {
    @Id
    private String userId;
    private String roleId;
```

```
    // 省略 set、get 方法
}
```

实体类 AyRole 和 AyUserRoleRel 开发完成之后，我们同样利用 JPA 生成对应的 Repository 接口：AyRoleRepository 和 AyUserRoleRelRepository，具体代码如下：

```
/**
 * 描述：用户角色 Repository
 * @author Ay
 * @date   2017/12/10.
 */
public interface AyRoleRepository extends JpaRepository<AyRole,
String> {

}

/**
 * 描述：用户角色关联 Repository
 * @author Ay
 * @date   2017/10/14.
 */
public interface AyUserRoleRelRepository extends JpaRepository
<AyUserRoleRel,String> {

    List<AyUserRoleRel> findByUserId(@Param("userId")String userID);
}
```

AyUserRoleRelRepository 类提供 findByUserId 方法用来查询用户关联的角色数据。Repository 接口 AyRoleRepository 和 AyUserRoleRelRepository 开发完成之后，生成对应的 Service 接口：AyRoleService 和 AyUserRoleRelService。具体代码如下：

```
/**
 * 描述：用户角色 Service
 * @author 阿毅
 * @date   2017/10/14
 */
public interface AyRoleService {
    AyRole find(String id);
}
```

AyRoleService 类提供 find 接口用来查询 AyRole 实体。

```
/**
 * 描述：用户角色关联 Service
 * @author 阿毅
 * @date   2017/10/14
 */
public interface AyUserRoleRelService {

    List<AyUserRoleRel> findByUserId(String userId);
}
```

AyUserRoleRelService 类提供 findByUserId 接口用来查询用户关联角色实体。Service 接口 AyRoleService 和 AyUserRoleRelService 开发完成之后，我们开发对应的实现类：AyRoleServiceImpl 和 AyUserRoleRelServiceImpl，在实现类中实现 Service 接口，并注入对应的 Repository 接口，具体代码如下：

```
/**
 * 描述：用户角色 Service
 * @author Ay
 * @date   2017/12/2
 */
@Service
public class AyRoleServiceImpl implements AyRoleService {

    @Resource
    private AyRoleRepository ayRoleRepository;

    @Override
    public AyRole find(String id){
        return ayRoleRepository.findById(id).get();
    }
}
```

```
/**
 * 描述：用户角色关联 Service
 * @author    Ay
 * @date      2017/12/10.
 */
@Service
public class AyUserRoleServiceImpl implements AyUserRoleRelService{

    @Resource
    private AyUserRoleRelRepository ayUserRoleRelRepository;

    @Override
    public List<AyUserRoleRel> findByUserId(String userId) {
        return ayUserRoleRelRepository.findByUserId(userId);
    }
}
```

实现类 AyRoleServiceImpl 和 AyUserRoleRelServiceImpl 开发完成之后，我们需要开发 CustomUserService 类并实现 UserDetailsService 接口，UserDetailsService 接口是 Spring Security 框架提供的，CustomUserService 的具体代码如下：

```
/**
 * 描述：自定义用户服务类
 * @author Ay
 * @date   2017/12/10.
 */
@Service
public class CustomUserService implements UserDetailsService{

    @Resource
    private AyUserService ayUserService;

    @Resource
    private AyUserRoleRelService ayUserRoleRelService;

    @Resource
    private AyRoleService ayRoleService;
```

```
    @Override
    public UserDetails loadUserByUsername(String name)
throws UsernameNotFoundException {
        AyUser ayUser = ayUserService.findByUserName(name);
        if(ayUser == null){
            throw new BusinessException("用户不存在");
        }
        // 获取用户所有的关联角色
        List<AyUserRoleRel> ayRoleList
        = ayUserRoleRelService.findByUserId(ayUser.getId());
        List<GrantedAuthority> authorityList = new
        ArrayList<GrantedAuthority>();
        if(ayRoleList != null && ayRoleList.size() > 0){
            for(AyUserRoleRel rel:ayRoleList){
                // 获取用户关联角色名称
                String roleName = ayRoleService.find(rel.
                getRoleId()).getName();
                authorityList.add(new SimpleGrantedAuthority
                (roleName));
            }
        }
        return new User(ayUser.getName(), ayUser.getPassword(),
        authorityList);
    }
}
```

在 CustomUserService 类中注入 AyUserService 服务类，并在 loadUserByUsername 方法中通过用户名查询用户，如果用户不存在，就抛出业务异常 BusinessException，该异常类在第 12 章中已经开发完成，如果用户存在，就继续根据用户 ID 查询用户关联的角色。最后在 loadUserByUsername 方法中返回 User 类，User 对象来自 org.springframework.security.core.userdetails.User，用于实现 UserDetails 接口，User 类具体源代码如下：

```
public class User implements UserDetails, CredentialsContainer {
    private static final long serialVersionUID = 420L;
    private String password;
```

```
    private final String username;

public User(String username, String password,
Collection<? extends GrantedAuthority> authorities) {
        this(username, password, true, true, true, true,
        authorities);
    }
    // 省略大量代码
}
```

CustomUserService 类开发完成之后，我们需要在 WebSecurityConfig 类中注册，WebSecurityConfig 具体代码如下：

```
/**
 * 描述：security 配置类
 * @author ay
 * @date   2017/12/10.
 */
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Bean
    public CustomUserService customUserService(){
        return new CustomUserService();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        // 路由策略和访问权限的简单配置
        http
                .formLogin()                    // 启用默认登录页面
                .failureUrl("/login?error")
                      // 登录失败返回 URL:/login?error
                .defaultSuccessUrl("/ayUser/test")
                      // 登录成功跳转 URL，这里调整到用户首页
```

```
                .permitAll();          // 登录页面全部权限可访问
        super.configure(http);
    }
    /**
     * 配置内存用户
     */
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth)
    throws Exception {
        auth
                .userDetailsService(customUserService());
//              .inMemoryAuthentication()
//              .withUser("阿毅").password("123456").roles("ADMIN")
//              .and()
//              .withUser("阿兰").password("123456").roles("USER");
    }
}
```

在 WebSecurityConfig 配置类中，通过注解 @Bean 将 CustomUserService 装进 Spring 容器，并在 configureGlobal 方法中注册 CustomUserService 类。

14.2.5 测试

代码开发完成之后，重新启动 my-spring-boot 项目，项目启动成功之后，在浏览器中输入访问链接：http://localhost:8080/login，在登录页面中填入用户名和密码：阿毅 /123456，单击 Login 按钮，便可以成功登录。关闭浏览器或者清除浏览器的 Cookie 和缓存信息，重新访问链接：http://localhost:8080/login，在登录页面中填入用户名和密码：阿兰 /123456，单击 Login 按钮，同样可以成功登录。

第15章

Spring Boot 应用监控

本章主要介绍如何通过 Spring Boot 监控和管理应用、自定义监控端点以及自定义 HealthIndicator 等。

15.1 应用监控介绍

Spring Boot 大部分模块都是用于开发业务功能或者连接外部资源。除此之外，Spring Boot 还为我们提供了 spring-boot-starter-actuator 模块，该模块主要用于管理和监控应用，是一个暴露自身信息的模块。spring-boot-starter-actuator 模块可以有效地减少监控系统在采集应用指标时的开发量。

spring-boot-starter-actuator 模块提供了监控和管理端点以及一些常用的扩展和配置方式，如表 15-1 所示。

表15-1 监控和管理端点

路径（端点名）	描述	鉴权
/health	显示应用监控指标	false
/beans	查看bean及其关系列表	true
/info	查看应用信息	false
/trace	查看基本追踪信息	true
/env	查看所有环境变量	true
/env/{name}	查看具体变量值	true
/mappings	查看所有url映射	true
/autoconfig	查看当前应用的所有自动配置	true
/configprops	查看应用所有配置属性	true
/shutdown	关闭应用（默认关闭）	true
/metrics	查看应用基本指标	true
/metrics/{name}	查看应用具体指标	true
/dump	打印线程栈	true

15.2 使用应用监控

15.2.1 引入依赖

在 Spring Boot 中使用监控，首先需要在 pom.xml 文件中引入所需的依赖 spring-boot-starter-actuator，具体代码如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
    <version>1.5.10.RELEASE</version>
</dependency>
```

15.2.2 添加配置

在 pom.xml 文件中引入 spring-boot-starter-actuator 依赖包之后，我们需要在 application.properties 文件中添加如下配置信息：

```
### 应用监控配置
# 指定访问这些监控方法的端口
# management.port
# 指定地址，比如只能通过本机监控，可以设置 management.address = 127.0.0.1
# management.address=127.0.0.1
# 敏感信息访问限制
# endpoints.bean.sensitive=false
# 设置关闭安全限制
management.security.enabled=false
```

management.port 用于指定访问这些监控方法的端口；management.address 用于指定地址，比如只能通过本机监控，可以设置 management.address = 127.0.0.1；endpoints.bean.sensitive 用于敏感信息访问限制。在表 15-1 中，鉴权为 false 的，表示不敏感，可以随意访问，否则就做了一些保护，不能随意访问。比如 beans 端点，我们可以设置 endpoints.bean.sensitive=false，这样就可以访问 beans 的所有信息。每一个端点都设置 sensitive=false 比较麻烦，因此也可以设置关闭安全限制：management.security.enabled=false。

15.2.3 测试

spring-boot-starter-actuator 依赖和配置都添加成功之后，重新启动 my-spring-boot 项目，项目启动成功之后，在浏览器中测试各个端点。比如在浏览器中输入 http://localhost:8080/health，可以看到如图 15-1 所示的应用健康信息。

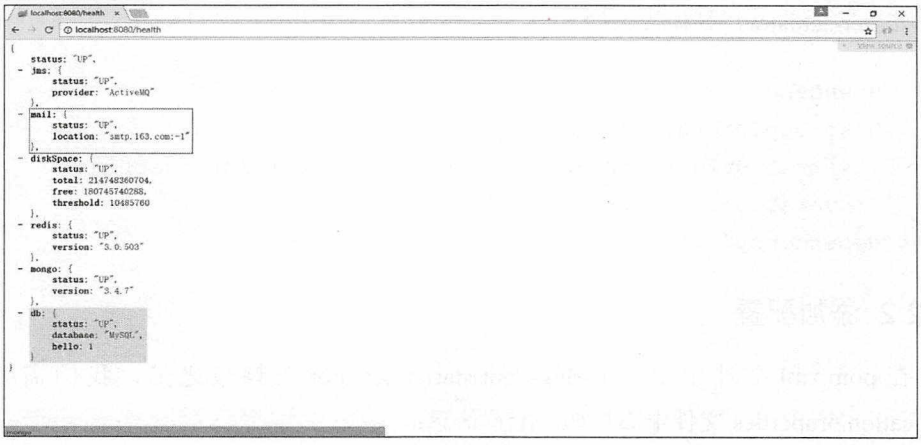


图 15-1 应用健康信息

在浏览器中输入 http://localhost:8080/metrics，可以看到如图 15-2 所示的各项指标信息。

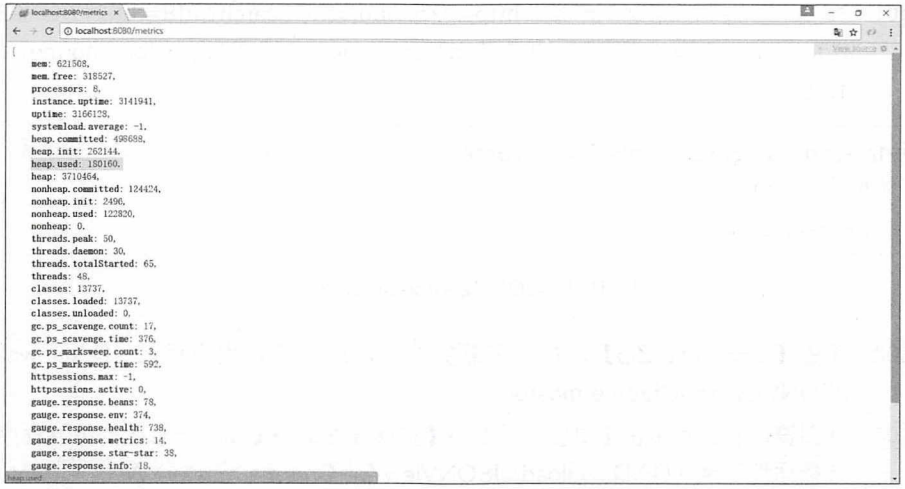


图 15-2　指标信息

其他端点测试可以按照表 15-2 所示的访问路径依次访问测试。

表15-2　端点访问路径

路径（端点名）	描述
http://localhost:8080/health	显示应用监控指标
http://localhost:8080/beans	查看bean及其关系列表
http://localhost:8080/info	查看应用信息
http://localhost:8080/trace	查看基本追踪信息
http://localhost:8080/env	查看所有环境变量
http://localhost:8080/env/{name}	查看具体变量值
http://localhost:8080/mappings	查看所有url映射
http://localhost:8080/autoconfig	查看当前应用的所有自动配置
http://localhost:8080/configprops	查看应用所有配置属性
http://localhost:8080/shutdown	关闭应用（默认关闭）
http://localhost:8080/metrics	查看应用基本指标
http://localhost:8080/metrics/{name}	查看应用具体指标
http://localhost:8080/dump	打印线程栈

在浏览器中可以把返回的数据格式化成 json 格式，这是因为在 Google 浏览器中安装了 JsonView 插件，具体安装步骤如下：

步骤 01 在浏览器中输入链接：https://github.com/search?utf8=%E2%9C%93&q=jsonview，在弹出的页面中单击 gildas-lormeau/JSONView-for-Chorme，如图 15-3 所示。

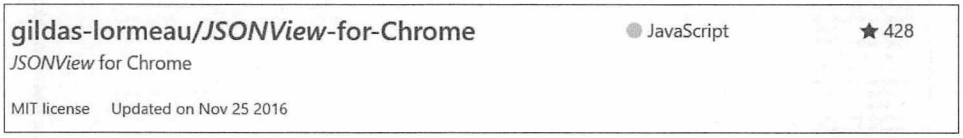


图 15-3 JSONView-for-Chrome 界面

步骤 02 单击【Download Zip】，插件下载完成后，解压缩到相应目录（D:\Download\JSONView-for-Chrome-master）。

步骤 03 在浏览器右上角单击【更多工具】→【扩展程序】→【加载已解压的扩展程序】，选择插件目录（D:\Download\JSONView-for-Chrome-master\WebContent）。

步骤 04 安装完成后，重新启动浏览器（快捷键：Ctrl+R）。

15.2.4 定制端点

除了 spring-boot-starter-actuator 提供的默认端点外，我们还可以定制端点，定制端点一般通过 endpoints + 端点名 + 属性名来设置。比如，我们可以在配置文件 application.properties 中把端点名 health 修改为 myhealth，具体代码如下：

```
endpoints.health.id=myhealth
```

配置添加完成之后，重新启动 my-spring-boot 项目，在浏览器中输入访问地址：http://localhost:8080/myhealth，一样可以获得应用健康相关的信息。如果我们想关闭端点 beans，可以在配置文件 application.properties 中添加如下代码：

```
endpoints.beans.enabled=false
```

配置添加完成之后，重新启动 my-pring-boot 项目，在浏览器中输入访问地址：http://localhost:8080/beans，查询不到 bean 相关的信息，而是跳转到错误界面。还可以通过关闭所有的端点，再开启所需端点来实现我们的需求，具体代码如下：

```
### 关闭所有的端点
endpoints.enabled=false
```

```
### 开启具体端点，如 beans
endpoints.beans.enabled=true
```

配置添加完成之后，重新启动 my-spring-boot 项目，在浏览器中输入访问地址：http://localhost:8080/beans，可以查询到 bean 相关信息，而输入访问地址：http://localhost:8080/health，查询不到应用健康相关的信息。除了 beans 端点外，输入其他端点的访问地址同样查询不到任何信息。

端点的访问路径默认在根路径下，如 http://localhost:8080/beans。我们可以配置修改端点访问的上下文路径，具体代码如下：

```
### 配置端点访问的上下文路径
management.context-path=/manage
```

配置添加完成之后，重新启动 my-pring-boot 项目，在浏览器中输入访问地址：http://localhost:8080/manage/beans，可以查询到 bean 相关的信息。端点访问的端口默认是项目访问端口，我们还可以通过配置修改端点访问端口。具体代码如下：

```
management.port=8081
```

配置添加完成之后，重新启动 my-pring-boot 项目，在浏览器中输入访问地址：http://localhost:8081/manage/beans，可以查询到 bean 相关的信息。

15.3 自定义端点

15.3.1 自定义端点 EndPoint

spring-boot-starter-actuator 模块中已经提供了许多原生端点。根据端点的作用，我们可以把原生端点分为以下三大类。

（1）应用配置类：获取应用程序中加载的应用配置、环境变量、自动化配置报告等与 Spring Boot 应用密切相关的配置类信息。

（2）度量指标类：获取应用程序运行过程中用于监控的度量指标，比如内存信息、线程池信息、HTTP 请求统计等。

（3）操作控制类：提供对应用的关闭等操作类功能。

如果 spring-boot-starter-actuator 模块提供的这些原生端点无法满足需求，我们还可以自定义端点。自定义端点时，只要继承抽象类 AbstractEndpoint 即可。这里在 my-spring-boot 目录 /src/main/java/com.example.demo 下新建 actuator 包，在 actuator 包下新建自定义端点类 AyUserEndPoint，AyUserEndPoint 主要用来监控数据库用户信息情况，比如用户总数量、被删除用户数量、活跃用户数量等。自定义端点 AyUserEndPoint 类代码如下：

```
/**
 * 描述：自定义端点
 * @author  Ay
 * @date    2017/12/9.
 */
@ConfigurationProperties(prefix = "endpoints.userEndPoints")
public class AyUserEndPoint extends AbstractEndpoint<Map
<String,Object>> {

    @Resource
    private AyUserService ayUserService;

    public AyUserEndPoint() {
        super("userEndPoints ");
    }

    @Override
    public Map<String, Object> invoke() {
        Map<String, Object> map = new HashMap<String, Object>();
        // 当前时间
        map.put("current_time",new Date());
        // 用户总数量
        map.put("user_num",ayUserService.findUserTotalNum());
        return map;
    }
}
```

- @ConfigurationProperties：该注解等同于在 application.properties 中添加配置：endpoints.userEndPoints。

- AbstractEndpoint：AbstractEndpoint 是 Endpoint 接口的抽象实现，实现 AbstractEndpoint 抽象类需要重写 invoke 方法。

在 AyUserEndPoint 类中注入 AyUserService 接口，并在 invoke 方法中调用 findUserTotalNum 方法，查询当前数据库总的用户数。所以需要在 AyUserService 接口中添加方法 findUserTotalNum，具体代码如下：

```
// 查询用户数量
Long findUserTotalNum();
```

同时，在 AyUserServiceImpl 类中实现方法 findUserTotalNum，具体代码如下：

```
@Override
public Long findUserTotalNum() {
    return ayUserRepository.count();
}
```

AyUserService 类与 AyUserServiceImpl 类开发完成之后，就可以在 invoke 方法中使用了。在 invoke 方法中定义 Map 集合，并往 Map 集合存放当前时间 current_time 和数据库用户总数 user_num。

自定义端点类 AyUserEndPoint 开发完成之后，还需要把自定义端点类注册到 Spring Boot 中，所以我们在项目 my-spring-boot 目录 /src/main/java/com.example.demo.actuator 下新建端点配置类 EndPointConfig，具体代码如下：

```
/**
 * 描述：自定义端点配置类
 * @author  Ay
 * @date    2017/12/9.
 */
@Configuration
public class EndPointConfig {
    @Bean
    public Endpoint<Map<String,Object>> AyUserEndPoint(){
        return new AyUserEndPoint();
    }
}
```

利用注解 @Configuration 把 EndPointConfig 定义成配置文件，在配置文件中，通过 @Bean 注解将 AyUserEndPoint 定义成可以让 Spring Boot 管理的 bean 对象。

15.3.2 测试

代码开发完成之后，我们要注释掉 application.properties 配置文件中所有关于应用监控的配置，只留下配置：management.security.enabled=false。配置文件修改完成之后，重新启动 my-spring-boot 项目，在浏览器中输入访问地址：http://localhost:8080/userEndPoints，便可以看到请求到的数据，具体数据如下：

```
{"user_num":3,"current": 当前时间 }
```

从返回数据中可以看出，当前数据库总共有 3 个用户，同时显示当前的具体时间（毫秒）。

15.3.3 自定义 HealthIndicator

默认端点 Health 的信息是从 HealthIndicator 的 bean 中收集的，Spring 中内置了一些 HealthIndicator，如表 15-3 所示。

表15-3 Spring内置的HealthIndicator

名称	描述
CassandraHealthIndicator	检测Cassandra数据库是否运行
DiskSpaceHealthIndicator	检测磁盘空间
DataSourceHealthIndicator	检测DataSource连接是否能获得
ElasticsearchHealthIndicator	检测Elasticsearch集群是否在运行
JmsHealthIndicator	检测JMS消息代理是否在运行
MailHealthIndicator	检测邮箱服务器是否在运行
MongoHealthIndicator	检测Mongo是否在运行
RabbitHealthIndicator	检测RabbitMQ是否在运行
RedisHealthIndicator	检测Redis是否在运行
SolrHealthIndicator	检测Solr是否在运行

启动项目 my-spring-boot，在浏览器中输入访问链接：http://localhost:8080/health，可以看到返回的 Spring Boot 应用健康数据只有：

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

这样简单的数据很明显是不够的，因此需要自定义 HealthIndicator 来获得更多应用健康的信息。我们在 my-spring-boot 项目目录 /src/main/java/com.example.actuator 下新建 MyHealthIndicator 类，该类实现 HealthIndicator 接口并重写 health 方法，MyHealthIndicator 类具体代码如下：

```
/**
 * 描述：自定义健康监控类
 * @author Ay
 * @date   2017/12/9
 */
@Component
public class MyHealthIndicator implements HealthIndicator{

    @Override
    public Health health() {
        Long totalSpace = checkTocalSpace();
        Long free = checkFree();
        String status = checkStatus();
        checkFree();
        return new Health.Builder()
            .up()
            .withDetail("status",status)
            .withDetail("total",totalSpace)
            .withDetail("free",free)
            .build();
    }
    private String checkStatus(){
        // 结合真实项目获取相关参数
        return "UP";
    }
    private Long checkTocalSpace(){
        // 结合真实项目获取相关参数
        return 10000L;
```

```
    }
    private Long checkFree(){
        // 结合真实项目获取相关参数
        return 5000L;
    }
}
```

15.3.4 测试

代码开发完成之后，重新启动 my-spring-boot 项目，项目启动成功之后，在浏览器中输入访问链接：http://localhost:8080/health，可以获得自定义健康类 MyHealthIndicator 返回的结果，具体结果信息如下：

```
my: {
status: "UP",
total: 10000,
free: 5000
}
// 省略其他健康数据
```

从上面返回的 json 结果信息可以看出，json 结果信息的 key：my 也就是英文 MyHealthIndicator 去掉 HealthIndicator。如果自定义健康类取名为 MyDefineHealthIndicator，返回结果信息将会变成：

```
myDefine: {
status: "UP",
total: 10000,
free: 5000
}
// 省略其他健康数据
```

15.4 保护 Actuator 端点

Actuator 端点发布的信息很多都涉及敏感信息和高危操作，比如 /shutdown 端点可以直接关闭应用程序，如果随便某个人都有权限访问该端点，那么是非常危险的。因此，我们有必要控制 Actuator 端点的访问权限来保护 Actuator 端点被非法访问。想

要保护 Actuator 端点，我们可以像保护其他 URL 路径一样，通过使用 Spring Security 来控制 URL 路径的授权访问。

在第 14 章中，我们已经在 Spring Boot 中集成 Spring Security，并且开发了 WebSecurityConfig 配置类对用户登录进行授权访问，现在改造该类，具体代码如下：

```
/**
 * 描述：security 配置类
 * @author ay
 * @date   2017/12/10.
 */
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    // 省略代码

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        // 路由策略和访问权限的简单配置
        http
                .authorizeRequests()
                // 要求有管理员的权限
                .antMatchers("/shutdown")
                .access("hasRole('ADMIN')")
                .antMatchers("/**").permitAll()
                .and()
                .formLogin()                          // 启用默认登录页面
                .failureUrl("/login?error")
                     // 登录失败，返回 URL:/login?error
                .defaultSuccessUrl("/ayUser/test")
                     // 登录成功，跳转 URL，这里调整到用户首页
                .permitAll();              // 登录页面全部权限可访问
        super.configure(http);
    }
}
```

我们通过使用 antMatchers("/shutdown").access("hasRole('READER')") 方法对 /shutdown 进行授权访问，/shutdown 端点现在仅允许拥有 ADMIN 权限的用户进行访问。

端点 /shutdown 已经被保护起来了，假如现在想保护其他端点，例如 /metrics、/health 等，只需要传入到 antMatchers() 的输入参数即可。具体代码如下：

```
.authorizeRequests()
// 要求有管理员的权限
.antMatchers("/shutdown","/metrics","/health")
.access("hasRole('READER')")
```

如果觉得每次添加一个端点的访问权限都得在 antMatchers() 方法中修改很麻烦，那么可以在 application.properties 配置文件中配置端点访问的上下文，具体配置如下：

```
### 配置端点访问的上下文路径
management.context-path=/manage
```

此时，在为 Actuator 端点赋予 ADMIN 权限的时候，就能借助这个上下文 /manage：

```
// 要求有管理员的权限
.antMatchers("/manage/**") .access("hasRole('READER')")
```

第16章

集成 Dubbo 和 Zookeeper

本章主要介绍如何安装并运行 Zookeeper、Spring Boot 集成 Dubbo、my-spring-boot 项目的服务拆分和实践、正式版 API 如何发布、服务注册等。

16.1 Zookeeper 介绍与安装

16.1.1 Zookeeper 概述

ZooKeeper 是一个开源的分布式应用程序协调服务，提供的功能包括命名服务、配置管理、集群管理、分布式锁等。

（1）命名服务。可以简单理解为电话簿。电话号码不好记，但是人名好记。要打谁的电话，直接查人名就好了。在分布式环境下，经常需要对应用 / 服务进行

统一命名，便于识别不同服务。类似于域名与 IP 之间的对应关系，域名容易记住。ZooKeeper 通过名称来获取资源或服务的地址、提供者等信息。

（2）配置管理。分布式系统都有大量服务器，比如在搭建 Hadoop 的 HDFS 的时候，需要在一台 Master 主机器上配置好 HDFS 需要的各种配置文件，然后通过 scp 命令把这些配置文件复制到其他节点上，这样各个机器拿到的配置信息是一致的，才能成功运行 HDFS 服务。Zookeeper 提供了这样的服务：一种集中管理配置的方法，我们在这个集中的地方修改了配置，所有对这个配置感兴趣的都可以获得变更。这样就省去手动复制配置，还保证了可靠和一致性。

（3）集群管理。集群管理包含两点：是否有机器退出和加入、选举 Master。在分布式集群中，经常会由于各种原因（比如硬件故障、软件故障、网络问题等），有些新的节点会加入进来，也有老的节点会退出集群。这个时候，集群中有些机器（比如 Master 节点）需要感知到这种变化，然后根据这种变化做出对应的决策。Zookeeper 集群管理就是感知变化，做出对应的策略。

（4）分布式锁。Zookeeper 的一致性文件系统使得锁的问题变得容易。锁服务可以分为两类，一类是保持独占，另一类是控制时序。单机程序的各个进程对互斥资源进行访问时需要加锁，分布式程序分布在各个主机上的进程对互斥资源进行访问时也需要加锁。很多分布式系统有多个可服务的窗口，但是在某个时刻只让一个服务去干活，当这台服务出问题的时候锁释放，立即 fail over 到另外的服务。在很多分布式系统中都是这么做的，这种设计有一个更好听的名字 Leader Election（Leader 选举）。举个通俗点的例子，比如去银行取钱，有多个窗口，但是只能有一个窗口对你服务。如果正在对你服务的窗口的柜员突然有急事走了，怎么办呢？找大堂经理（Zookeeper），大堂经理会指定另外的窗口继续为你服务。

Zookeeper的一个最常用的使用场景是担任服务生产者和服务消费者的注册中心，这也是接下来的章节中会使用到的。服务生产者将自己提供的服务注册到 Zookeeper 中心，服务消费者在进行服务调用的时候先到 Zookeeper 中查找服务，获取服务生产者的详细信息之后，再去调用服务生产者的内容与数据，具体如图 16-1 所示。

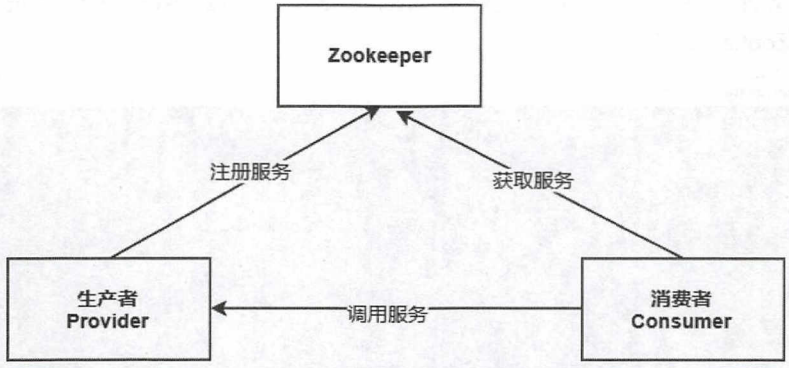


图 16-1　Zookeeper 服务注册简单原理

本书毕竟不是一本专门讲 Zookeeper 的书，所以本节中只是对 Zookeeper 进行简单的功能介绍。

16.1.2 Zookeeper 的安装与启动

Zookeeper 的安装与启动非常简单，具体步骤如下：

步骤 01　在官方网站（https://archive.apache.org/dist/zookeeper/）下载对应安装包，这里选用 zookeeper-3.3.6 版本。

步骤 02　将下载好的安装包解压到 D 盘目录下。在解压后的目录中找到配置文件 zoo_sample.cfg（存放在 D:\zookeeper-3.3.6\zookeeper-3.3.6\conf\ 目录下），该配置文件是 Zookeeper 为我们提供的最简单的配置文件，将其复制一份并重新命名为 zoo.cfg，Zookeeper 启动时会默认在 conf 目录下读取 zoo.cfg 配置文件，如图 16-2 所示。

名称	修改日期	类型	大小
configuration.xsl	2012/7/29 14:23	XSL 样式表	1 KB
log4j.properties	2012/7/29 14:23	PROPERTIES 文件	2 KB
zoo.cfg	2012/7/29 14:23	CFG 文件	1 KB
zoo_sample.cfg	2012/7/29 14:23	CFG 文件	1 KB

图 16-2　生成 zoo.cfg 配置文件

步骤 03　zoo.cfg 配置文件生成之后，在目录 D: \zookeeper-3.3.6\zookeeper-3.3.6\bin 下双击 zkServer.cmd 文件启动 Zookeeper，如果是 Linux 环境，就执行 zkServer.sh

文件。在命令行窗口中看到如图 16-3 所示的 Zookeeper 启动信息，代表 Zookeeper 安装成功。

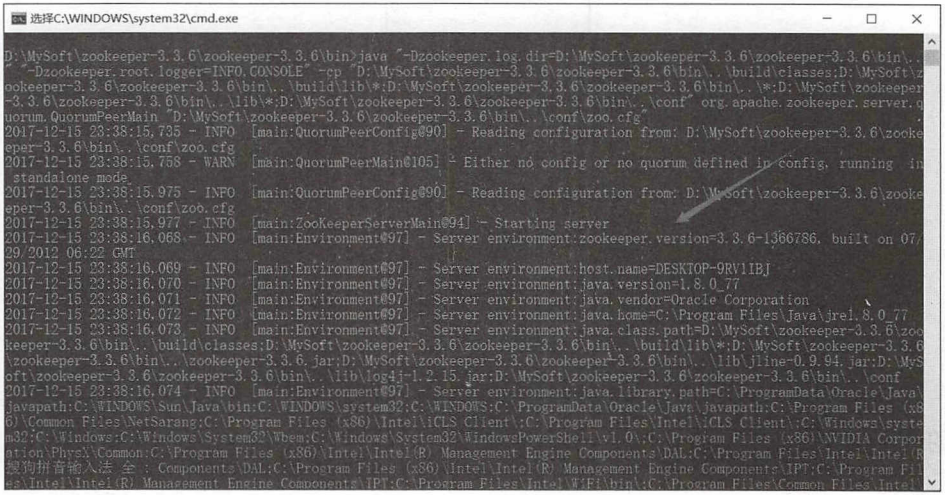


图 16-3 Zookeeper 安装成功信息

16.2 Spring Boot 集成 Dubbo

16.2.1 Dubbo 概述

Dubbo 是阿里巴巴 B2B 平台技术部开发的一款 Java 服务平台框架以及 SOA 治理方案。其功能主要包括：高性能 NIO 通信及多协议集成、服务动态寻址与路由、软负载均衡与容错、依赖分析与降级等。Dubbo 简单的底层框架如图 16-4 所示。

Registry 是服务注册与发现的注册中心， Provider 是暴露服务的服务提供方，Consumer 是调用远程服务的服务消费方，Monitor 是统计服务的调用次数和调用时间的监控中心，Container 是服务运行容器。Dubbo 简单的调用关系如下：

（1）服务容器 Container 负责启动、加载、运行服务提供者 Provider。

（2）服务提供者 Provider 在启动时，向注册中心 Registry 注册自己提供的服务。

（3）服务消费者 Consumer 在启动时，向注册中心 Registry 订阅自己所需的服务。

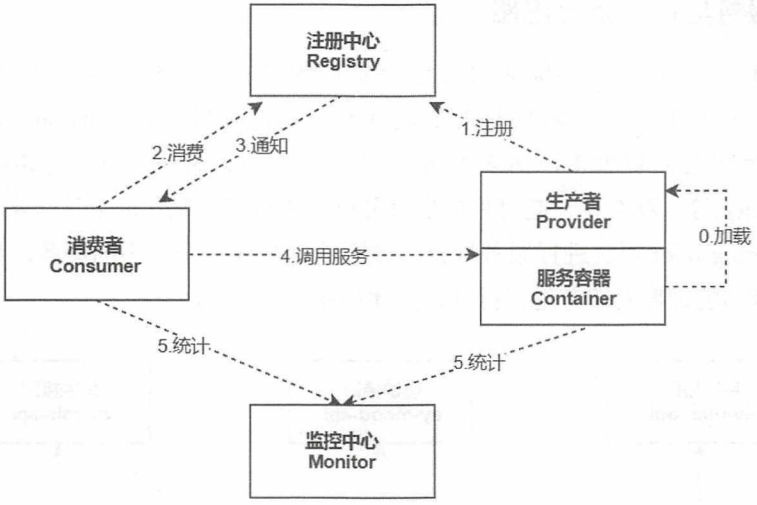


图 16-4　Dubbo 底层框架原理

（4）注册中心 Registry 返回服务提供者地址列表给消费者 Provider，如果有变更，注册中心 Registry 将基于长连接推送，变更数据给消费者 Consumer。

（5）服务消费者 Consumer 从提供者地址列表中基于软负载均衡算法选一台提供者进行调用，如果调用失败，就再选另一台调用。

（6）服务消费者 Consumer 和提供者 Provider 在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心 Monitor。

Dubbo 将注册 Registry 中心进行抽象，使得它可以外接不同的存储媒介给注册中心 Registry 提供服务，可以作为存储媒介的有 ZooKeeper、Memcached、Redis 等。引入 ZooKeeper 作为存储媒介，也就是把 ZooKeeper 的特性引进来。首先是负载均衡，单注册中心的承载能力是有限的，在流量达到一定程度的时候就需要分流，负载均衡就是为了分流而存在的。一个 ZooKeeper 群配合相应的 Web 应用就可以很容易达到负载均衡；然后是资源同步，单单有负载均衡还不够，节点之间的数据和资源需要同步，ZooKeeper 集群就天然具备这样的功能；最后是命名服务，将树状结构用于维护全局的服务地址列表，服务提供者在启动的时候，向 ZooKeeper 的指定节点目录写入自己的 URL 地址，这个操作就完成了服务的发布。ZooKeeper 其他特性还有 Mast 选举、分布式锁等。ZooKeeper 的知识在 16.1 节中已经简单地介绍了，这里就不再重复叙述。

16.2.2 服务与接口拆分思路

截至 16 章，my-spring-boot 项目已经集成很多技术，也定义了很多接口。但是对于真实的项目来说，特别是对于互联网公司的项目来说，my-spring-boot 这个大的服务承载的内容太多，诸多服务接口（比如 AyUserService、AyMoodService、AyRoleService 等）糅合在一起对外提供服务已经违背了微服务理念。因此，我们有必要对 my-spring-boot 项目进行服务拆分，使它被拆分成一个个小的服务，我们可以安装业务或者功能维度对服务进行拆分，具体如图 16-5 所示。

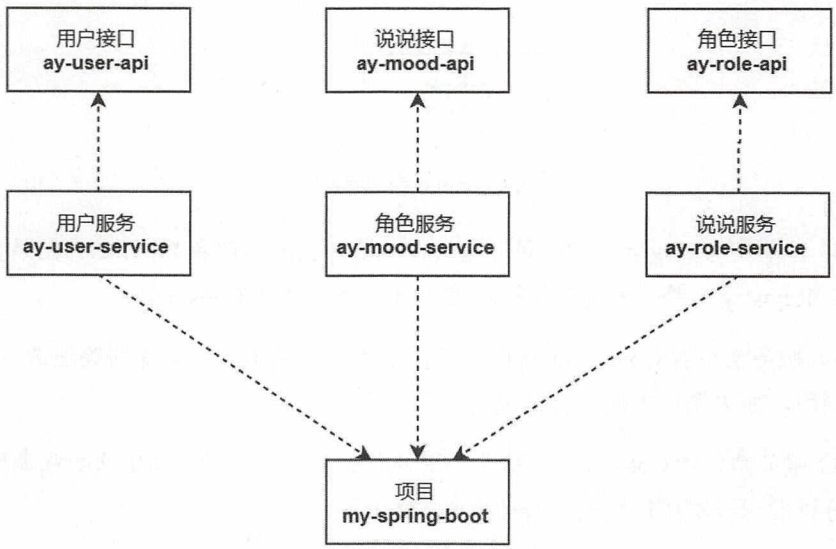


图 16-5 my-spring-boot 服务拆分

在图 16-5 中，my-spring-boot 项目被拆分为用户服务、角色服务、说说服务等。my-spring-boot 项目依赖于这些底层的服务为其提供相应的功能，而用户服务、角色服务和说说服务是面向接口 API 编程的，符合基本的编程原则。通过服务的拆分和面向接口编程，对于项目扩展和团队分工都有莫大的好处。

16.2.3 服务与接口拆分实践

我们已经清楚服务拆分的原因和好处，本节就要在项目中实践它。首先，我们在 my-spring-boot 项目下添加 ay-user-api、ay-mood-api、ay-role-api 接口模块。具体步骤如下：

步骤 01　选择 my-spring-boot 并右击，在弹出的快捷菜单中单击【New】→【Module】，在弹出的窗口中选择【Spring Initializr】，然后单击【Next】按钮，如图 16-6 所示。

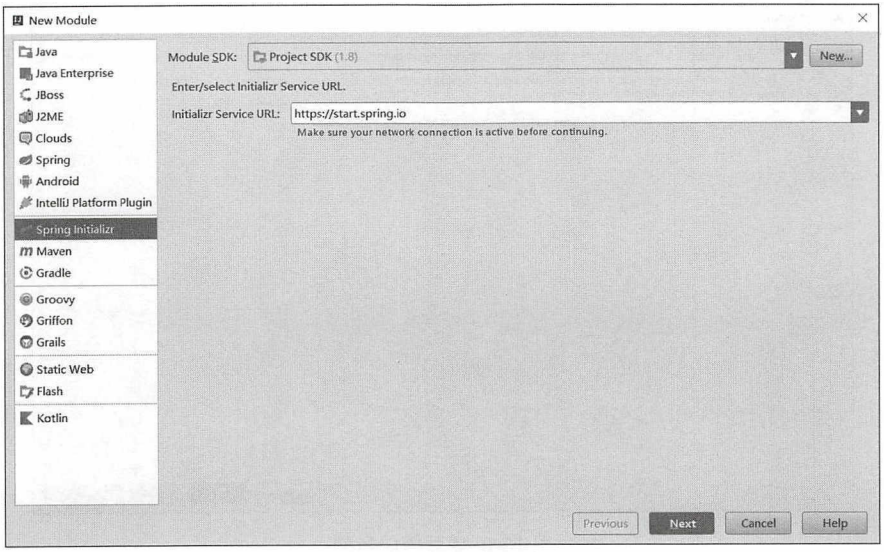


图 16-6　New Module 窗口

步骤 02　在 Name 输入框中输入模块的名称 ay-user-api，其他选项保持默认即可。 然后单击【Next】按钮，如图 16-7 所示。

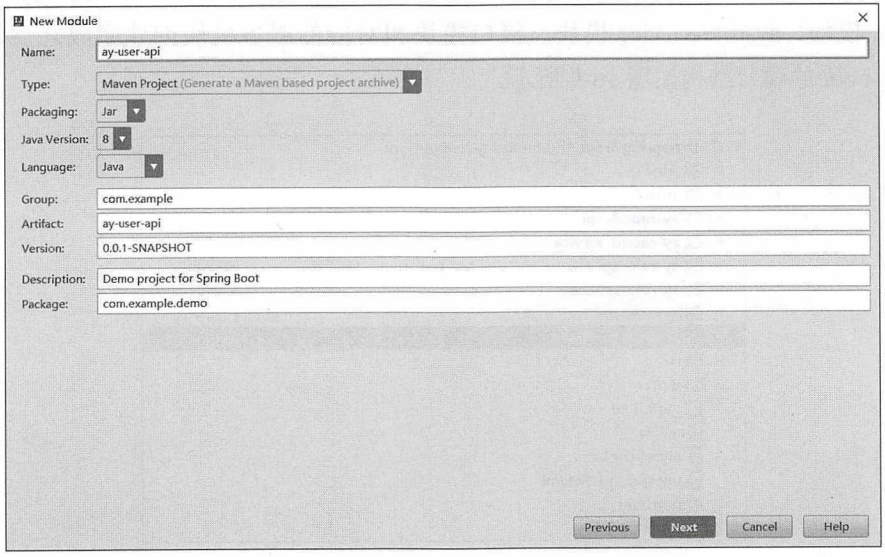


图 16-7　填写模块名称

步骤 03 一路单击【Next】按钮，在 Module name 输入框中填入模块的名称 ay-user-api，然后单击【Finish】按钮，如图 16-8 所示。

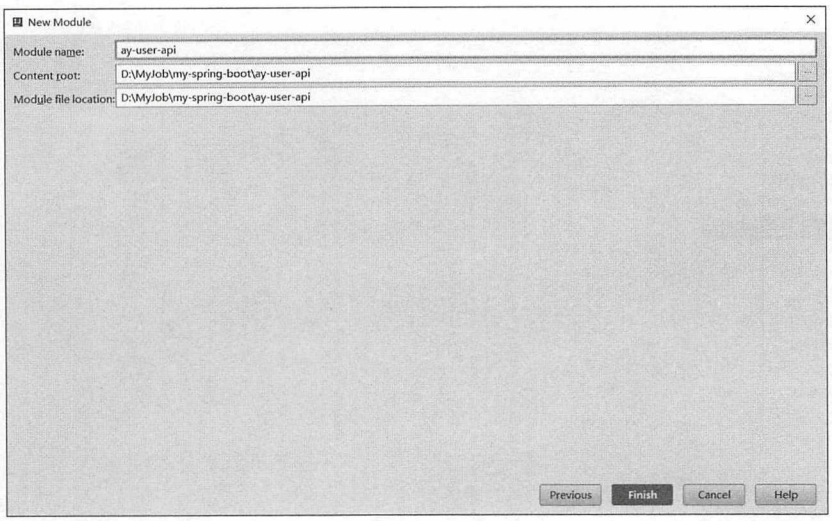


图 16-8 填写模块名称

按照上面的步骤就建立好 ay-user-api 模块了。按照相同的步骤，依次在 my-spring-boot 项目下创建接口模块：ay-mood-api 模块、ay-role-api 模块。接口模块创建完成之后，按照相同的步骤创建接口对应的服务模块：ay-user-service 模块、ay-mood-service 模块、ay-role-service 模块。接口模块和对应的服务模块创建完成之后，my-spring-boot 的项目结构如图 16-9 所示。

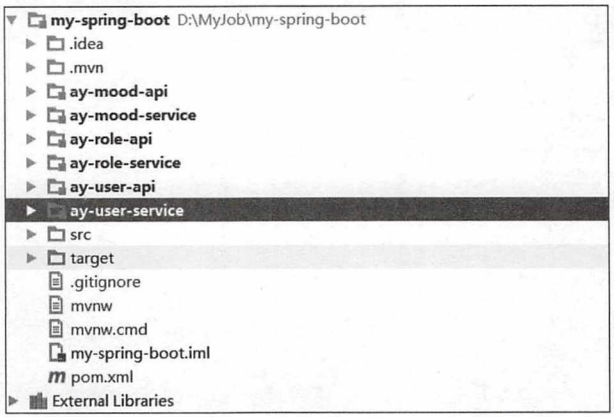


图 16-9 my-spring-boot 模块目录结构

为了方便，笔者把所有的模块都放在 my-spring-boot 项目下，在真实的项目中并不是这样的。在真实的项目中，我们会为接口和对应的服务单独建立一个项目，比如为 ay-user-api 和 ay-user-service 建立一个项目，为 ay-role-api 和 ay-role-service 建立一个项目，为 ay-mood-api 和 ay-mood-service 建立一个项目。这样不同的开发人员单独负责不同的项目，分工合作，提高开发效率。

所有的模块都建立好之后，我们可以把 my-spring-boot 项目中的接口移动到对应的接口模块。比如把 my-spring-boot 项目中的 AyUserService 接口移动到 ay-user-api，把 AyMoodService 接口移动到 ay-mood-api，等等。同时，把实现类 AyUserServiceImpl 移动到 ay-user-service，把实现类 AyMoodServiceImpl 移动到 ay-mood-service，等等。这里以用户模块为例讲解整个开发过程。

首先，ay-user-api 模块创建完成之后，该模块就是一个 spring-boot 微服务项目，享有 spring-boot 为我们默认生成的各种“福利”。在 ay-user-api 包下创建 api 包和 domain 包，分别用来存放接口类和实体类，在 api 包下存放所有 my-spring-boot 项目移动过来的用户接口。ay-user-api 模块的目录如图 16-10 所示。

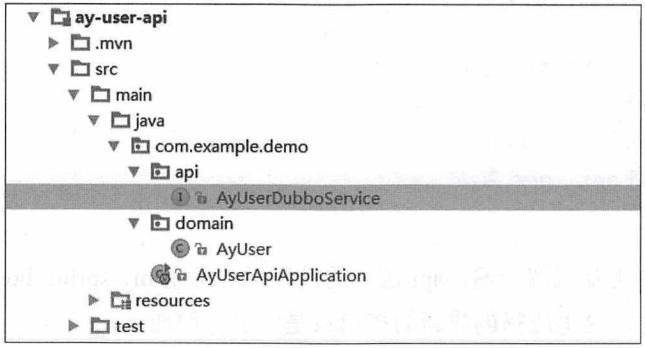


图 16-10　ay-user-api 模块目录结构

在图 16-10 中，为了方便，在 api 包下创建 AyUserDubboService 接口，该接口用来提供与用户相关的服务，比如增删改查等功能。AyUserDubboService 类的具体代码如下：

```
/**
 * 描述：用户接口
 * @author  Ay
 * @date    2017/12/16.
 */
```

```
public interface AyUserDubboService {

    AyUser findByUserNameAndPassword(String name, String password);
}
```

在 AyUserDubboService 类中，只有一个通过用户名和密码查询用户的接口 findByUserNameAndPassword。domain 包下的 AyUser 类具体代码如下：

```
/**
 * 描述：用户实体类
 * @author  Ay
 * @date    2017/12/16.
 */
public class AyUser {

    //主键
    private String id;
    //用户名
    private String name;
    //密码
    private String password;
    //邮箱
    private String mail;
    //省略 set、get 方法
}
```

这里还需要重复强调一遍，api 包下的接口理论上是 my-spring-boot 项目下的用户接口移动过来的，这里选择创建新的接口只是为了方便而已。

16.2.4 正式版发布

ay-user-api 模块接口开发完成之后，我们就可以将其发布成正式版 jar 包。jar 包有快照版 SNAPSHOT 和正式版。比如我们创建的 ay-user-api 模块默认就是快照版 0.0.1-SNAPSHOT。快照版的模块是可以重复修改的，而正式版的模块不可以修改。在开发过程中，模块基本都是 SNAPSHOT 版。当模块开发并测试完成后，会升级为正式版，而且在项目上线的时候，依赖的模块都必须是正式版，否则会出现意想不到

的错误。我们可以在 pom 文件中修改模块的版本，比如把 ay-user-api 修改成 0.0.1 正式版，具体步骤如下：

步骤 01　在 ay-user-api 模块中，把 pom 文件 jar 包版本 version 升级为正式版 0.0.1，如图 16-11 所示。

步骤 02　版本修改之后，在 IntelliJ IDEA 开发工具右侧单击【clean】【install】，将 ay-user-api 的 jar 包发布到本地 Maven 仓库，如图 16-12 所示。

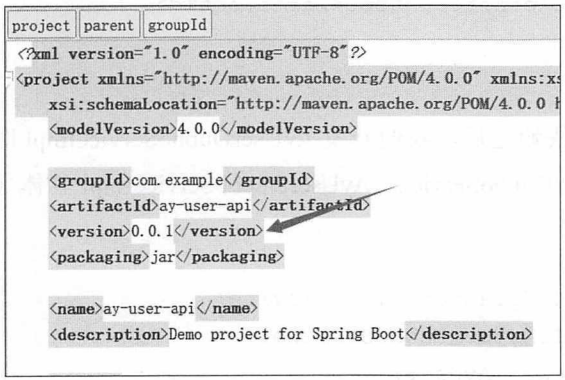


图 16-11　修改 pom 文件

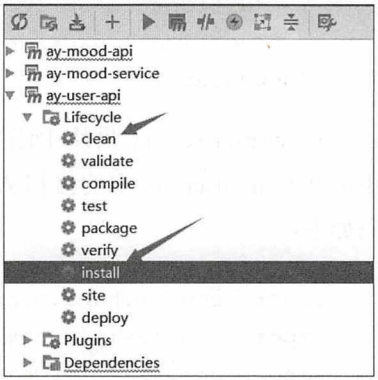


图 16-12　clean、install 命令窗口

步骤 03　ay-user-api 正式版的 jar 包发布成功之后，其他项目模块就可以使用它，同时我们可以到 Maven 仓库查看改 jar 包的具体信息，如图 16-13 所示。

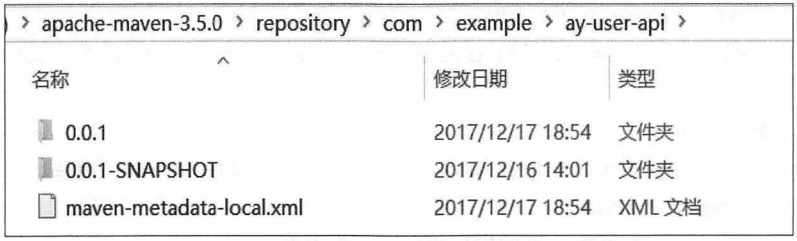


图 16-13　ay-user-api 的 Maven 仓库信息

16.2.5　Service 服务端开发

ay-user-api 接口模块发布为正式版之后，继续开发 ay-user-service 模块。首先在 ay-user-service 模块中引入 ay-user-api 的依赖和 Dubbo 依赖。添加 ay-user-api 的依赖是因为 ay-user-service 模块是面向接口编程的，而添加 Dubbo 依赖是因为服务开发完成之后，要把该服务注册到 Zookeeper 中，具体代码如下：

```
<dependency>
    <groupId>com.example</groupId>
    <artifactId>ay-user-api</artifactId>
    <version>0.0.1</version>
</dependency>
<dependency>
    <groupId>io.dubbo.springboot</groupId>
    <artifactId>spring-boot-starter-dubbo</artifactId>
    <version>1.0.0</version>
</dependency>
```

在 ay-user-service 模块中添加完依赖之后，我们开发 AyUserDubboServiceImpl 服务来实现 ay-user-api 中的接口 AyUserDubboService，AyUserDubboServiceImpl 具体代码如下：

```
import com.alibaba.dubbo.config.annotation.Service;
import com.example.demo.api.AyUserDubboService;
import com.example.demo.domain.AyUser;
/**
 * 描述：对外提供用户服务类
 * Created by Ay on 2017/12/16.
 */
@Service(version = "1.0")
public class AyUserDubboServiceImpl implements AyUserDubboService {

    @Override
    public AyUser findByUserNameAndPassword(String name,
    String password) {
        // 连接数据库，查询用户数据，此处省略
        AyUser ayUser = new AyUser();
        ayUser.setName("阿毅");
        ayUser.setPassword("123456");
        return ayUser;
    }
}
```

- @Service：这个注解不是 Spring 提供的，而是在 com.alibaba.dubbo.config.annotation.Service 包下面的，这一点要特别注意。在 AyUserDubboServiceImpl

类上添加 @Service 注解就可以把 AyUserDubboServiceImpl 类注册到 Zookeeper 服务中心，对外提供服务。version 属性用于指定服务的版本，这里服务版本为 1.0。当一个接口出现不兼容升级时，可以用版本号 version 过渡，版本号不同的服务相互间不引用。

- findByUserNameAndPassword 方法：理论上应该在该方法中通过 JPA Repository 或者 MyBatis 查询用户数据，这里为了方便，简单创建 AyUser 对象返回。

16.2.6 Service 服务注册

ay-user-service 模块开发完成之后，由于 ay-user-service 本身是一个 Spring Boot 微服务项目，因此我们可以单独运行它。找到 ay-user-service 模块的入口类 AyUserServiceApplication（确保 Zookeeper 是启动状态），执行入口类的 main 方法便可以启动 ay-user-service 服务。ay-user-service 服务启动完成之后，可以在 IntelliJ IDEA 的控制台中查看服务在 Zookeeper 的注册信息，具体信息如图 16-14 所示。

```
Session establishment complete on server 127.0.0.1/127.0.0.1:2181, sessionid = 0x16064c4eef00000, negotiated timeout = 1
zookeeper state changed (SyncConnected)
 [DUBBO] Register: dubbo://192.168.141.1:20880/com.example.demo.api.AyUserDubboService?anyhost=true&application=provider
 [DUBBO] Subscribe: provider://192.168.141.1:20880/com.example.demo.api.AyUserDubboService?anyhost=true&application=prov
 [DUBBO] Notify urls for subscribe url provider://192.168.141.1:20880/com.example.demo.api.AyUserDubboService?anyhost=t
Registering beans for JMX exposure on startup
Started AyUserServiceApplication in 3.673 seconds (JVM running for 5.405)
```

图 16-14　AyUserDubboService 注册信息

16.2.7 Client 客户端开发

ay-user-api 接口模块和 ay-user-service 服务模块开发完成之后，接下来开始开发客户端。所谓客户端，就是所有 ay-user-service 服务的对象。现在我们把 my-spring-boot 作为客户端，my-spring-boot 本身也是一个微服务。在 my-spring-boot 项目调用 ay-user-service 模块提供的服务，需要在 my-spring-boot 项目的 pom 文件中添加 ay-user-api 依赖和 Dubbo 依赖，具体代码如下：

```
<dependency>
    <groupId>com.example</groupId>
    <artifactId>ay-user-api</artifactId>
```

```
        <version>0.0.1</version>
    </dependency>
    <!-- dubbo start -->
    <dependency>
        <groupId>io.dubbo.springboot</groupId>
        <artifactId>spring-boot-starter-dubbo</artifactId>
        <version>1.0.0</version>
    </dependency>
```

ay-user-api 依赖和 Dubbo 依赖添加完成之后，我们就可以在代码中通过 @Reference 注解将 ay-user-service 模块提供的 AyUserDubboService 服务注入进来。具体代码如下：

```
@Reference(version = "1.0")
public AyUserDubboService ayUserDubboService;
```

@Reference 注解也是 Dubbo 框架提供的，在 com.alibaba.dubbo.config.annotation.Reference 包下。version 是 @Reference 注解的属性，version 的版本需要和 @Service 注解的 version 版本保持一致，否则服务将无法注入，这一点是需要特别注意的。

第17章

多环境配置与部署

本章主要介绍 Spring Boot 多环境配置及使用、Spring Boot 如何打成 War 包并部署到外部 Tomcat 服务器上等。

17.1 多环境配置介绍

在项目开发过程中，项目不同的角色会使用不同的环境。例如，开发人员会使用开发环境、测试人员会使用测试环境、性能测试会使用性能测试环境、项目开发完成之后会把项目部署到线上环境等，不同的环境往往会连接不同的 MySQL 数据库、Redis 缓存、MQ 消息中间件等，环境之间相互独立与隔离才不会相互影响，隔离的环境便于部署，提高工作效率。具体环境隔离如图 17-1 所示。

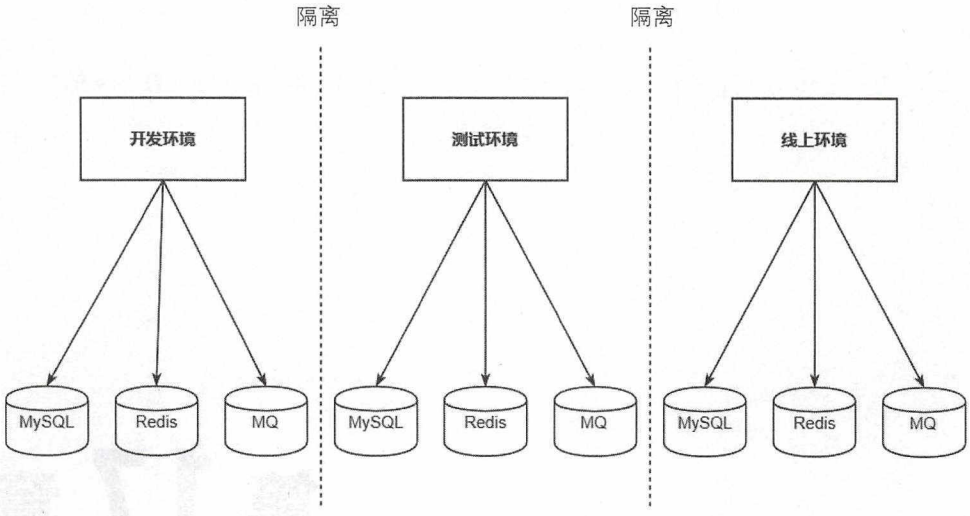


图 17-1 环境隔离

17.2 多环境配置使用

17.2.1 添加多个配置文件

假如项目 my-spring-boot 需要 3 个环境：开发环境、测试环境、性能测试环境。我们复制 my-spring-boot 项目配置文件 application.properties，分别取名为 application-dev.properties、application-test.properties、application-perform.properties，作为开发环境、测试环境、性能测试环境，具体如图 17-2 所示。

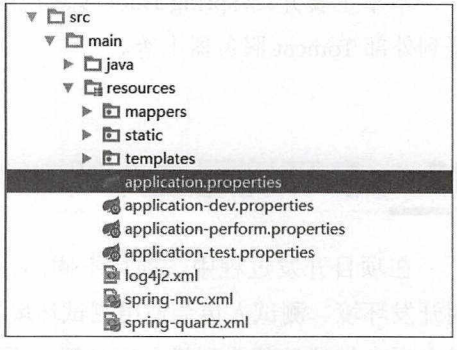


图 17-2 多环境配置文件

每个配置文件对应的 MySQL 数据库、Redis 缓存、ActiveMQ 消息队列等配置参数都不相同。

17.2.2 配置激活选项

多环境的配置文件开发完成之后，我们在 my-spring-boot 的配置文件 application.properties 中添加配置激活选项，具体代码如下：

```
### 激活开发环境配置
spring.profiles.active=dev
```

如果想激活测试环境的配置，可修改为：

```
### 激活测试环境配置
spring.profiles.active=test
```

如果想激活性能测试环境的配置，可修改为：

```
### 激活性能测试环境配置
spring.profiles.active=test
```

17.2.3 测试

多环境配置文件和配置激活选项开发完成之后，修改 application-dev.properties、application-test.properties、application-perform.properties 配置文件的数据库连接，具体代码如下。

开发环境配置文件 application-dev.properties 的具体代码修改如下：

```
### 开发环境 mysql 连接信息
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/test
```

测试环境配置文件 application-test.properties 的具体代码修改如下：

```
### 测试环境 mysql 连接信息
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/test2
```

性能测试环境配置文件 application-perform.properties 的具体代码修改如下：

```
### 性能测试环境 mysql 连接信息
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/test3
```

开发环境 MySQL 的 test 数据库已经存在，现在我们在 MySQL 数据库中创建 test2、test3 数据库作为测试环境和性能测试环境的数据库，并把 test 数据库中的数据

导入 test2、test3 数据库。我们可以利用 2.3.3 节介绍的 Navicat for MySQL 客户端完成数据库数据的导入、导出工作，具体步骤如下：

步骤 01 在 Navicat for MySQL 客户端中找到 test 数据库并右击，在打开的快捷菜单中单击【转储 SQL 文件】→【结构和数据】，将 test 数据库的数据存到指定的目录，如图 17-3 和图 17-4 所示。

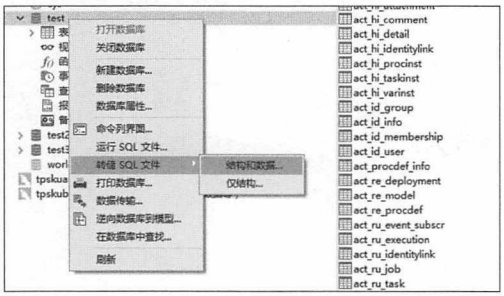


图 17-3 test 数据库导出操作

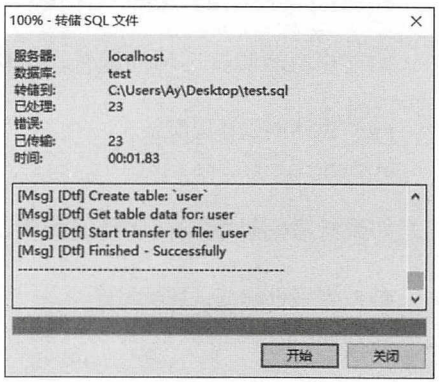


图 17-4 数据库数据导出成功

步骤 02 test 数据库数据导出成功之后，在 MySQL 数据库中新建 test2 和 test3 数据库。

步骤 03 将步骤 1 中的数据导入 test2 和 test3 数据库，右击后，在打开的快捷菜中单击【运行 SQL 文件】，选择 test.sql 存放的目录，单击【开始】按钮进行数据导入，如图 17-5 和图 17-6 所示。

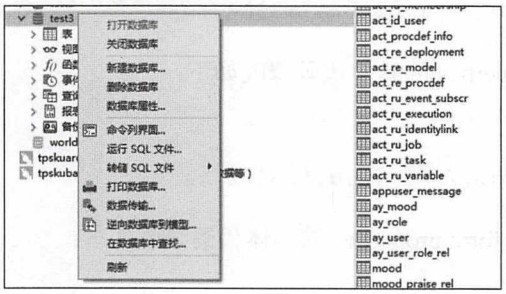


图 17-5 选择版本和组件

图 17-6 选择版本和组件

测试环境数据库 test2 和性能测试环境数据库 test3 创建和数据导入成功之后，重新启动 my-spring-boot 项目，项目成功启动之后，在浏览器中输入访问地址：

http://localhost:8080/ayUser/test，便可以成功访问用户数据。如果我们想切换到测试环境进行项目开发，可以激活测试环境配置 spring.profiles.active=test，然后重新启动 my-spring-boot 项目。

17.3 部署

17.3.1 Spring Boot 内置 Tomcat

Tomcat 是一个免费的、开放源代码的 Web 应用服务器，属于轻量级应用服务器。在中小型系统和并发访问用户不是很多的场合下被普遍使用。Spring Boot 默认使用 Tomcat 作为内嵌 Servlet 容器，查看 spring-boot-starter-web 依赖可知，如图 17-7 所示。本书使用 Tomcat 8.0 进行讲解，可到官方网站 https://tomcat.apache.org/download-80.cgi 进行下载，下载完成之后解压到 D 盘，并将解压后的文件夹命名为 tomcat8，如图 17-8 所示。

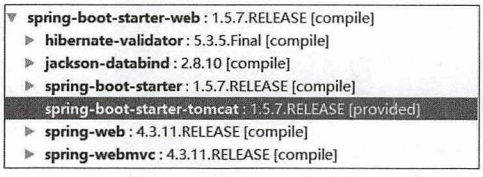


图 17-7　查看依赖

图 17-8　Tomcat 解压目录

如果想使用其他 Servlet 容器，比如 Jetty 作为 Spring Boot 默认内置容器，只需要修改 spring-boot-starter-web 依赖即可。使用 Jetty 容器作为 Spring Boot 默认内置容器，具体修改代码如下：

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

```
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

17.3.2 IntelliJ IDEA 配置 Tomcat

在 IntelliJ IDEA 配置 Tomcat，具体步骤如下：

步骤 01 在 IDEA 开发菜单栏中选择【run】→【Edit Configurations】，在弹出的窗口中选择【Defaults】→【Tomcat Server】→【Local】，在【Application server】中选择 Tomcat 的安装路径，在【JRE】中选择 JDK 的安装路径，如图 17-8 所示。

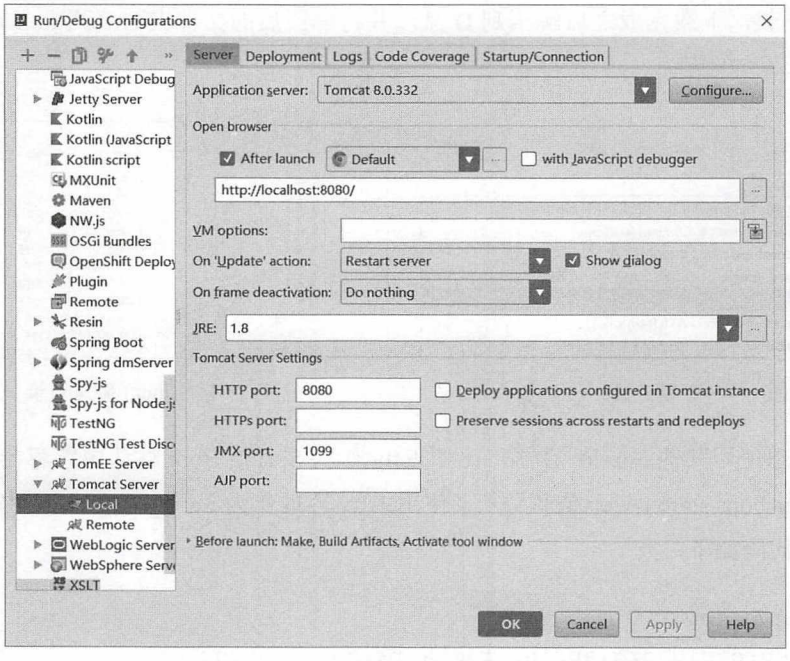


图 17-8 Tomcat 配置

步骤 02 在【Deployment】页签中选择【Artifact】→【my-spring-boot:war exploded】，如图 17-9 所示。

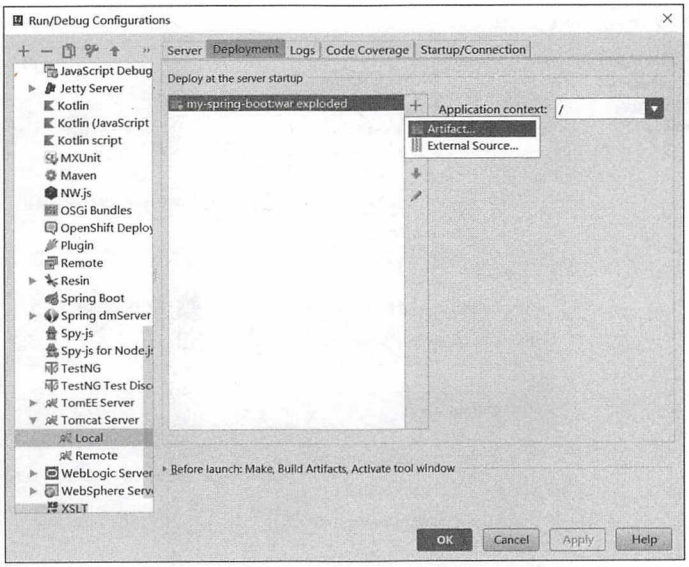


图 17-9　配置 war 包

步骤 03　步骤 1 和步骤 2 只是配置一个 Defaults 默认的 Tomcat 模板，现在我们单击【+】按钮→【Tomcat Server】→【Local】，在弹出的界面中输入 Name（tomcat8），其他信息会从默认模板中获取，如图 17-10 和图 17-11 所示。

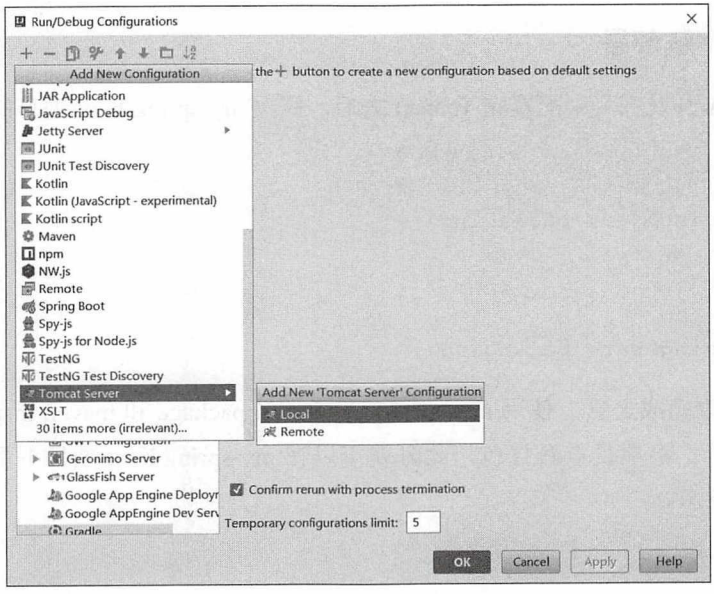


图 17-10　创建 Tomcat 配置

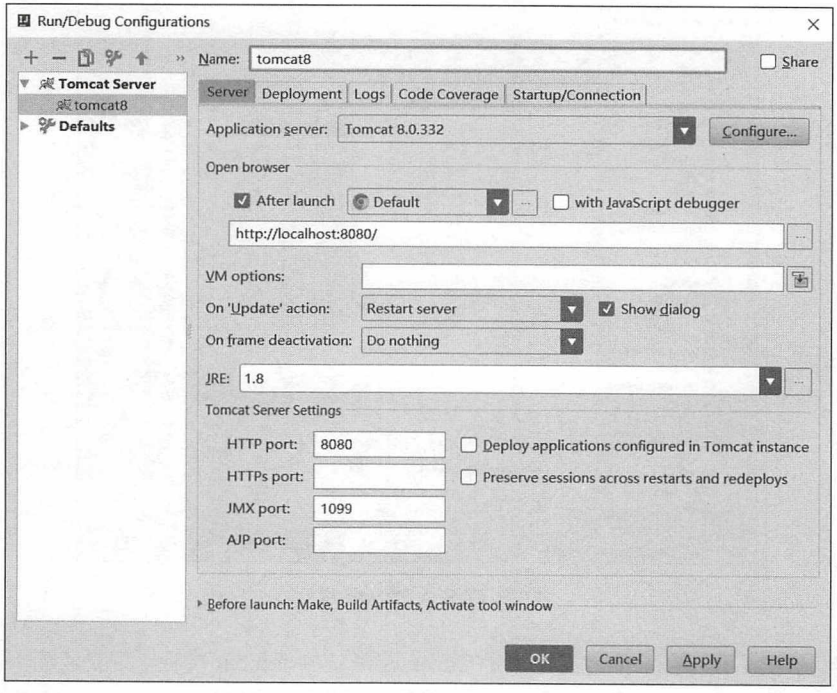


图 17-11 修改 Tomcat 名称

17.3.3 war 包部署

在 IDEA 开发工具中配置完 Tomcat 之后，修改 my-spring-boot 项目的 pom.xml 文件，将配置：

```
<packaging>jar</packaging>
```

修改为：

```
<packaging>war</packaging>
```

配置修改完成之后，使用 maven clean、maven package 和 maven install 命令，如图 17-12 所示。将项目重新打包，此时就可以在 my-spring-boot 项目下看到 war 包，如图 17-13 所示。

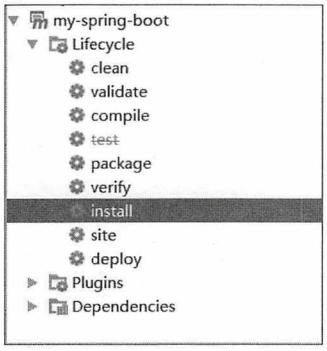


图 17-12　maven 命令

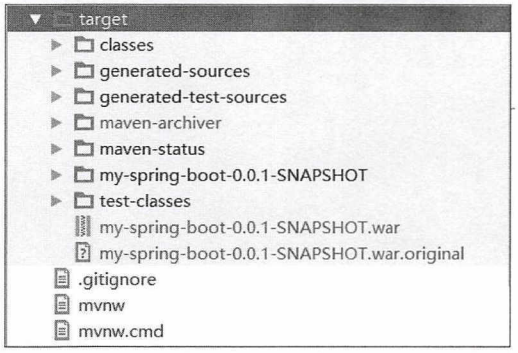


图 17-13　target 下的 war 包

17.3.4　测试

my-spring-boot-0.0.1-SNAPSHOT.war 包生成之后，就可以启动 my-spring-boot 项目了。现在不是使用项目入口类 MySpringBootApplication 启动项目，而是将 my-spring-boot 以 war 包的方式部署到外置的 Tomcat 服务器上。Tomcat 启动之后，我们可以在浏览器上访问 my-spring-boot 项目。

第18章

Spring Boot 原理解析

本章主要回顾 MySpringApplication 入口类上注解和 run 方法的原理、梳理 Spring Boot 启动执行的流程，并简单分析 spring-boot-starter 起步依赖原理等。

18.1 回顾入口类

18.1.1 MySpringBootApplication 入口类

首先，我们先来回顾一下项目 my-spring-boot 的入口类 MySpringBootApplication，具体代码如下：

```
@SpringBootApplication
@ServletComponentScan
```

```
@ImportResource(locations={"classpath:spring-mvc.xml"})
@EnableAsync
@EnableRetry
public class MySpringBootApplication {

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

在入口类 MySpringBootApplication 中，@SpringBootApplication 和 main 方法是 Spring Boot 自动生成的，其他注解都是我们在学习 Spring Boot 时整合其他技术添加上去的。接下来就和大家一起看看 @SpringBootApplication 和 SpringApplication.run 方法到底为我们做了些什么。

18.1.2 @SpringBootApplication 的原理

@SpringBootApplication 开启了 Spring 的组件扫描和 Spring Boot 自动配置功能。实际上它是一个复合注解，包含 3 个重要的注解：@SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan，其源代码如下：

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan
public @interface SpringBootApplication {

    //省略代码
}
```

- @SpringBootConfiguration 注解：标明该类使用 Spring 基于 Java 的注解，Spring Boot 推荐我们使用基于 Java 而不是 XML 的配置，所以本书的实战例子都是基于 Java 而不是 XML 的配置。本书使用的 Spring Boot 版

本为 2.0.0.RC1。查看 @SpringBootConfiguration 源代码，可以看到它对 @Configuration 进行了简单的“包装”，然后取名为 SpringBootConfiguration。@SpringBootConfiguration 源代码如下：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {

}
```

我们对 @Configuration 注解并不陌生，它就是 JavaConfig 形式的 Spring IoC 容器的配置类使用的 @Configuration。

- @EnableAutoConfiguration 注解：该注解可以开启自动配置的功能。@EnableAutoConfiguration 具体源代码如下：

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import({EnableAutoConfigurationImportSelector.class})
public @interface EnableAutoConfiguration {
    // 省略代码
}
```

从 @EnableAutoConfiguration 的源代码可以看出，其包含 @Import 注解。而我们知道，@Import 注解的主要作用就是借助 EnableAutoConfigurationImportSelector 将 Spring Boot 应用所有符合条件的 @Configuration 配置都加载到当前 Spring Boot 创建并使用的 Ioc 容器中，Ioc 容器就是我们所说的 Spring 应用程序上下文 ApplicationContext。学习过 Spring 框架就知道，Spring 框架提供了很多 @Enable 开头的注解定义，比如 @EnableScheduling、@EnableCaching 等。而这些 @Enable 开头的注解都有一个共同的功能，就是借助 @Import 的支持收集和注册特定场景相关的 bean 定义。

- @ComponentScan 注解：启动组件扫描，开发的组件或 bean 定义能被自动发现并注入 Spring 应用程序上下文。比如我们在控制层添加 @Controller 注解、在服务层添加 @Service 注解和 @Component 注解等，这些注解都可以被 @ComponentScan 注解扫描到。

在 Spring Boot 早期的版本中，需要在入口类同时添加这三个注解，但从 Spring Boot 1.2.0 开始，只要在入口类添加 @SpringBootApplication 注解即可。

18.1.3 SpringApplication 的 run 方法

除了 @SpringBootApplication 注解外，我们发现了入口类的一个显眼的地方，那就是 SpringApplication.run 方法。在 run 方法中，首先创建一个 SpringApplication 对象实例，然后调用 SpringApplication 的 run 方法。SpringApplication.run 方法的源代码如下：

```
public class SpringApplication{
    // 省略代码
public ConfigurableApplicationContext run(String... args) {
    StopWatch stopWatch = new StopWatch();
    stopWatch.start();
    ConfigurableApplicationContext context = null;
    FailureAnalyzers analyzers = null;
    configureHeadlessProperty();
        // 开启监听器
    SpringApplicationRunListeners listeners = getRunListeners(args);
    listeners.starting();
    try {
        ApplicationArguments applicationArguments =
                    new DefaultApplicationArguments(args);
        ConfigurableEnvironment environment =
                    prepareEnvironment(listeners,
            applicationArguments);
        Banner printedBanner = printBanner(environment);
        // 创建应用上下文
        context = createApplicationContext();
        analyzers = new FailureAnalyzers(context);
        // 准备上下文
```

```
            prepareContext(context, environment, listeners,
                    applicationArguments, printedBanner);
            //刷新应用上下文
            refreshContext(context);
            //刷新后操作
            afterRefresh(context, applicationArguments);
            listeners.finished(context, null);
            stopWatch.stop();
            if (this.logStartupInfo) {
                new StartupInfoLogger(this.mainApplicationClass)
                        .logStarted(getApplicationLog(), stopWatch);
            }
            return context;
        }
        catch (Throwable ex) {
            handleRunFailure(context, listeners, analyzers, ex);
            throw new IllegalStateException(ex);
        }
    }
```

从源代码可以看出，Spring Boot 首先开启了一个 SpringApplicationRunListeners 监听器，然后通过 createApplicationContext、prepareContext 和 refreshContext 方法创建、准备、刷新应用上下文 ConfigurableApplicationContext，通过上下文加载应用所需的类和各种环境配置等，最后启动一个应用实例。

18.1.4 SpringApplicationRunListeners 监听器

SpringApplicationRunListener 接口规定了 SpringBoot 的生命周期，在各个生命周期广播相应的事件（ApplicationEvent），实际调用的是 ApplicationListener 类。SpringApplicationRunListener 的源代码如下：

```
public interface SpringApplicationRunListener {
    //刚执行 run 方法时触发
    void starting();
    //环境建立好时触发
    void environmentPrepared(ConfigurableEnvironment environment);
```

```
    //上下文建立好的时触发
    void contextPrepared(ConfigurableApplicationContext context);
    //上下文载入配置时触发
    void contextLoaded(ConfigurableApplicationContext context);
    //上下文刷新完成后，run 方法执行完之前触发
    void finished(ConfigurableApplicationContext context,
    Throwable exception);

}
```

ApplicationListener 是 Spring 框架对 Java 中实现的监听器模式的一种框架实现。具体源代码如下：

```
public interface ApplicationListener<E extends ApplicationEvent>
                                              extends EventListener {
    void onApplicationEvent(E var1);
}
```

ApplicationListener 接口只有一个方法 onApplicationEvent，所以自己的类在实现该接口的时候要实现该方法。如果在上下文 ApplicationContext 中部署一个实现了 ApplicationListener 接口的监听器，每当 ApplicationEvent 事件发布到 ApplicationContext 时，该监听器就会得到通知。如果我们要为 Spring Boot 应用添加自定义的 ApplicationListener，那么可以通过 SpringApplication.addListeners() 或者 SpringApplication.setListeners() 方法添加一个或者多个自定义的 ApplicationListener。

18.1.5 ApplicationContextInitializer 接口

在 Spring Boot 准备上下文 prepareContext 时，会对 ConfigurableApplicationContext 实例做进一步的设置或者处理。prepareContext 的源代码如下：

```
private void prepareContext(ConfigurableApplicationContext context,
        ConfigurableEnvironment environment,
        SpringApplicationRunListeners listeners,
    ApplicationArguments applicationArguments, Banner printedBanner) {
    context.setEnvironment(environment);
    postProcessApplicationContext(context);
    //对上下文进行设置和处理
```

```
        applyInitializers(context);
        listeners.contextPrepared(context);
        if (this.logStartupInfo) {
            logStartupInfo(context.getParent() == null);
            logStartupProfileInfo(context);
        }

        // Add boot specific singleton beans
        context.getBeanFactory().registerSingleton
        ("springApplicationArguments",applicationArguments);
        if (printedBanner != null) {
            context.getBeanFactory().
                registerSingleton("springBootBanner", printedBanner);
        }

        // Load the sources
        Set<Object> sources = getSources();
        Assert.notEmpty(sources, "Sources must not be empty");
        load(context, sources.toArray(new Object[sources.size()]));
        listeners.contextLoaded(context);
    }
```

在准备上下文 prepareContext 方法中，通过 applyInitializers 方法对 context 上下文进行设置和处理。applyInitializers 具体源代码如下：

```
    protected void applyInitializers
    (ConfigurableApplicationContext context) {
        for (ApplicationContextInitializer initializer :
        getInitializers()) {
            Class<?> requiredType = GenericTypeResolver.
            resolveTypeArgument(
                initializer.getClass(),
                ApplicationContextInitializer.class);
            Assert.isInstanceOf(requiredType, context,
            "Unable to call initializer.");
            initializer.initialize(context);
        }
    }
```

在 applyInitializers 方法中，主要调用 ApplicationContextInitializer 类的 initialize 方法对应用上下文进行设置和处理。ApplicationContextInitializer 本质上是一个回调接口，用于在 ConfigurableApplicationContext 执行 refresh 操作之前对它进行一些初始化操作。一般情况下，我们基本不需要自定义一个 ApplicationContextInitializer，如果真需要自定义一个 ApplicationContextInitializer，那么可以通过 SpringApplication.addInitializers() 设置。

18.1.6 ApplicationRunner 与 CommandLineRunner

ApplicationRunner 与 CommandLineRunner 接口执行点是在容器启动成功后的最后一步回调，我们可以在回调方法 run 中执行相关逻辑。ApplicationRunner 具体源代码如下：

```
public interface ApplicationRunner {
    void run(ApplicationArguments args) throws Exception;
}
```

CommandLineRunner 具体源代码如下：

```
public interface CommandLineRunner {
    void run(String... args) throws Exception;
}
```

在 ApplicationRunner 或 CommandLineRunner 类中只有一个 run 方法，但是它们的入参不一样，分别是 ApplicationArguments 和可变 String 数组。

如果有多个 ApplicationRunner 或 CommandLineRunner 实现类，而我们需要按一定顺序执行它们，那么可以在实现类上加上 @Order（value= 整数值）注解，SpringBoot 会按照 @Order 中的 value 值从小到大依次执行。

18.2 SpringApplication 执行流程

在 18.1 节，我们对 SpringApplication 的 run 方法进行了简单的学习，这里再简单总结一下 Spring Boot 启动的完整流程，具体流程如图 18-1 所示。

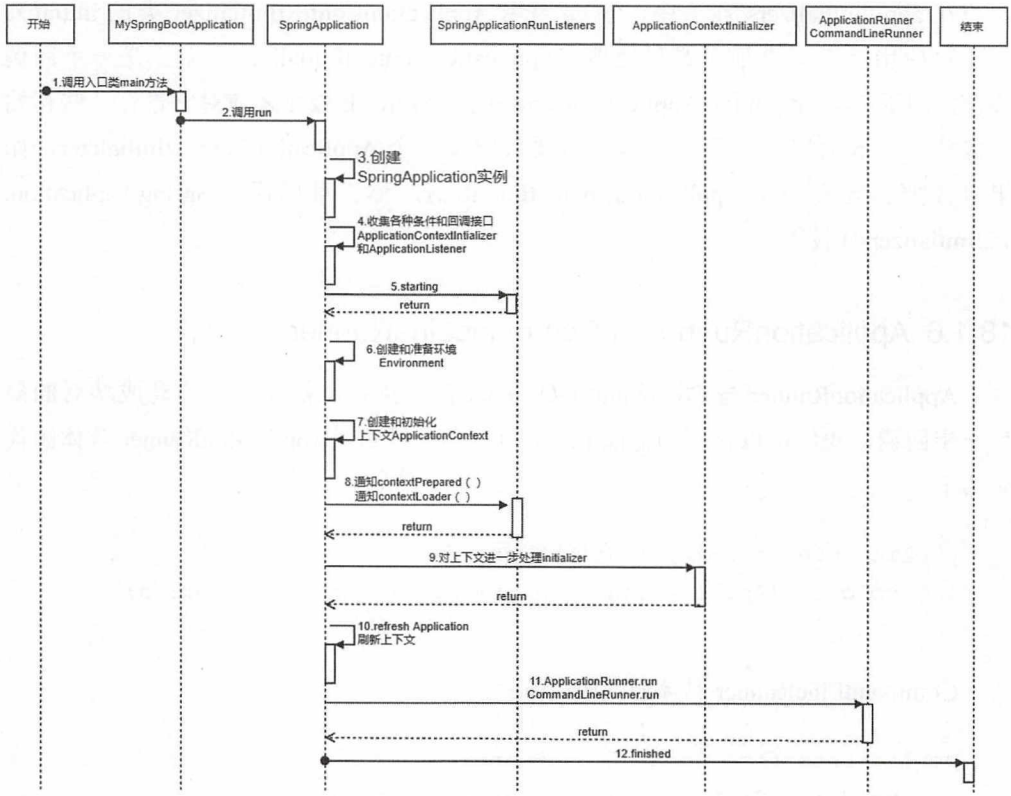


图 18-1 Spring Boot 启动流程

（1）项目启动时，调用入口类 MySpringBootApplication 的 main 方法。

（2）入口类 MySpringBootApplication 的 main 方法会调用 SpringApplication 的静态方法 run。

（3）在 run 方法中首先创建一个 SpringApplication 对象实例，然后调用 SpringApplication 对象实例的 run 方法。

（4）查询和加载所有的 SpringApplicationListener 监听器。

（5）SpringApplicationListener 监听器调用其 starting 方法，Spring Boot 通知这些 SpringApplicationListener 监听器，马上要开始执行了。

（6）创建和准备 Spring Boot 应用将要使用的 Environment 环境，包括配置要使

用的 PropertySource 以及 Profile。

（7）创建和初始化应用上下文 ApplicationContext。这一步只是准备工作，并未开始正式创建。

（8）这一步是最重要的，Spring Boot 会通过 @EnableAutoConfiguration 获取所有配置以及其他形式的 Ioc 容器配置，并加载到已经准备完毕的 ApplicationContext。

（9）主要是调用 ApplicationContextInitializer 类的 initialize 方法对应用上下文进行设置和处理。

（10）调用 ApplicationContext 上下文的 refresh 方法，使 Ioc 容器达到可用状态。

（11）查找当前 ApplicationContext 上下文是否注册 ApplicationRunner 与 CommandLineRunner，如果有，循环遍历执行 ApplicationRunner 和 CommandLineRunner 的 run 方法。

（12）执行 SpringApplicationListener 的 finished 方法，Spring Boot 应用启动完毕。

18.3 spring-boot-starter 原理

在之前的章节中，我们在项目的 pom 文件中引入了很多 spring-boot-starter 依赖，比如 spring-boot-starter-jdbc、spring-boot-starter-jdbc-logging、spring-boot-starter-web 等。这些带有 spring-boot-starter 前缀的依赖都叫作 Spring Boot 起步依赖，它们有助于 Spring Boot 应用程序的构建。

如果没有起步依赖，要使用 Spring MVC 的话，我们根本记不住 Spring MVC 到底要引入哪些依赖包、到底要使用哪个版本的 Spring MVC、Spring MVC 的 Group 和 Artifact ID 又是多少。

Spring Boot 通过提供众多起步依赖降低项目依赖的复杂度。起步依赖本质上是一个 Maven 项目对象模型，定义了对其他库的传递依赖，这些依赖的合集可以对外提供某项功能。起步依赖的命名表明它们提供某种或某类功能。例如，spring-boot-starter-jdbc 表示提供 jdbc 相关的功能，spring-boot-starter-jpa 表示提供 JPA 相关的功能，等等。下面简单列举工作中经常使用的起步依赖，如表 18-1 所示。

表18-1 常用的spring-boot-starter起步依赖

名称	描述
spring-boot-starter-logging	提供logging相关的日志功能
spring-boot-starter-thymeleaf	使用Thymeleaf视图构建MVC Web应用程序的启动器
spring-boot-starter-parent	常被作为父依赖，提供智能资源过滤、智能的插件设置、编译级别和通用的测试框架等
spring-boot-starter-web	使用Spring MVC构建Web，包括RESTful应用程序。使用Tomcat作为默认的嵌入式容器的启动器
spring-boot-starter-test	支持常规的测试依赖，包括JUnit、Hamcrest、Mockito以及spring-test模块
spring-boot-starter-jdbc	使用JDBC与Tomcat JDBC连接池的启动器
spring-boot-starter-data-jpa	使用Spring数据JPA与Hibernate的启动器
spring-boot-starter-data-redis	Redis key-value数据存储与Spring Data Redis和Jedis客户端启动器
spring-boot-starter-log4j2	提供log4j2相关的日志功能
spring-boot-starter-mail	提供邮件相关的功能
spring-boot-starter-activemq	使用Apache ActiveMQ的JMS启动器
spring-boot-starter-data-mongodb	使用MongoDB面向文档的数据库和Spring Data MongoDB的启动器
spring-boot-starter-actuator	提供应用监控与健康相关的功能
spring-boot-starter-security	使用Spring security的启动器
spring-boot-starter-dubbo	提供dubbo框架相关的功能

事实上，起步依赖和项目里的其他依赖没什么区别，引入起步依赖的同时会引入相关的传递依赖。例如，spring-boot-starter-web 起步依赖会引入 spring-webmvc、jackson-databind、spring-boot-starter-tomcat 等传递依赖。如果我们不想用 spring-boot-starter-web 引入的 spring-webmvc 传递依赖，那么可以使用 <exclusions> 标签来排除传递依赖。具体代码如下：

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

```
            <!-- 排查 spring-webmvc -->
            <exclusion>
                <groupId>org.springframework</groupId>
                <artifactId>spring-webmvc</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

假如 spring-boot-starter-web 引入的传递依赖版本过低，我们可以在 pom 文件中直接引入所需的版本，告诉 Maven 现在需要这个版本的依赖。

参考文献

[1] 汪云飞 . Java EE 开发的颠覆者 Spring Boot 实战 [M]. 北京：电子工业出版社，2016.

[2] 郝佳 . Spring 源码深度解析 [M]. 北京：人民邮电出版社，2013.

[3] 王富强 . Spring Boot 揭秘：快速构建微服务体系 [M]. 北京：机械工业出版社，2016.

[4] https://spring.io/guides.

[5] https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/.

本书附赠视频教学（需下载）：

（共12堂课，播放时长120分钟）

1.课程介绍

2.快速搭建Spring Boot项目

3.Spring Boot集成MySQL数据库

4.Spring Boot集成MyBatis

5.Spring Boot整合Quartz定时器

6.Spring Boot整合Redis缓存

7.Spring Boot整合ActiveMQ消息

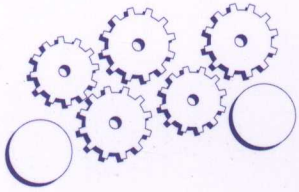
8.Spring Boot整合Swagger

9.Spring Boot整合过滤器filter

10.Spring Boot整合监听器Listener

11.Spring Boot全局异常处理

12.Spring Boot常用标签的使用



Spring Boot作为目前流行的微服务框架，其设计目的是简化Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。Spring Boot致力于在蓬勃发展的快速应用开发领域成为领导者，因此掌握并学会使用Spring Boot是成为Java Web开发人员的必备技能之一。

本书主要内容包括Spring Boot环境搭建、Spring Boot常用标签、Spring Boot集成Redis、数据库MySQL、Spring Data、日志Log4J、Thymeleaf模板引擎、ActiveMQ消息、MyBatis等流行技术，以及利用Spring Boot实现邮件发送、Quartz定时器、过滤器Filter和监听器Listener等。

本书是作者在Spring Boot实际项目开发中的心得与经验结晶，从Spring Boot基础到项目开发，涵盖了目前大部分的热门技术，书中采用了具体案例来讲解实际项目开发，并给出了完整的项目代码。通过学习本书，读者既能够掌握Spring Boot的相关技术和应用，又能够举一反三，在自己的项目开发中活学活用。

本书使用Spring Boot 2.0.0RC1以及Intellij IDEA新版本进行概念讲解与代码开发。

清华社官方微信号

扫 我 有 惊 喜

ISBN 978-7-302-50329-3

9 787302 503293

定价：59.00元