第 3 章

Spring Boot

目前主流的互联网公司在开发 Java 项目时基本会使用 Spring Boot 快速构建应用，相比以前的项目，使用 Spring 开发需要一系列的配置，Spring Boot 提供了 Spring 运行的默认配置，相当于将汽车手动挡改为自动挡，这种方式旨在让开发者能够更专注于业务实现。本章从 Spring Boot 的实战集成入手，介绍集成 Spring Boot 的 3 种方式及 Spring Boot 底层运行的工作原理。希望读者能够对 Spring Boot 框架有一个清楚的认识，深入理解它的底层运行机制。

3.1 自动配置/依赖管理

在古代的某个“Spring 村(春风村)”，居民们过着平静的生活。随着时代的进步，人们对方便的生活方式的要求越来越高，为此，春风村推出了一种名为“春风吹又生”的新科技，让人们可以更轻松地完成每件事。Spring Boot 是一个开放源码的架构，旨在为 Spring 程序的创建、配置和运行提供更简单、更高效的方法。它可以让 Spring 程序更容易创建、配置和运行，同时也让 Spring 程序更加容易管理。有了 Spring Boot 的“冲锋枪”，开发人员可以更快、更容易地完成事情。汤姆就是这样一个例子，他决定为自己的咖啡馆开发一款网络软件。他开始建立一个简单的 Maven 工程，很快就建立起了这个软件的基础结构，然后汤姆使用 Spring Boot 提供的自动设置功能设置应用中缺省的数据库连接和网络 MVC，实现了一个完整的网络应用。Spring Boot 也被称为 Starters，它可以使从属关系管理变得更简单。有了 Starters，Tom 可以简单地增加一个依赖性获取自己想要的类库和能力。Spring Boot 让汤姆在他的“咖啡屋”网络应用软件中变得更加容易工作，而不用去考虑那些烦琐的组态和从属关系管理。另外，“汤姆咖啡”网站的成功例子也使 Spring Boot 在春风村中变得非常有影响力。现在，Spring Boot 已成为开发人员最喜欢的框架之一，它可以让开发人员在不同的应用中更容易地进行开发和部署。

3.2 实战集成

快速构建一个 Spring Boot 项目，通常只需往 pom 文件中添加依赖。本节将介绍如何使用 spring-boot-starter-parent、spring-boot-dependencies、io.spring.pl at form 这 3 种方式

快速集成 Spring Boot 项目。

3.2.1 使用 spring-boot-starter-parent

Spring Boot 官方提供的示例项目依赖于 spring-boot-starter-parent 进行依赖管理,但在企业级微服务架构中,每个模块只能有一个 parent,多个微服务间的继承关系可能导致扩展性问题,开发者需要采取其他手段(如修改父类依赖项或通过 import 导入)实现。在 Spring Boot 的各个发布版本中包含了许多默认版本的依赖项,对开发者而言是一个便利,只需专注于核心功能的开发。

下面去证实这一说法:如何在 Spring Boot 项目中添加依赖。具体方法是新建一个继承 Spring Boot 的项目,并在 pom.xml 文件中添加依赖,代码如下:

```
//第 3 章/3.2.1 继承 spring-boot-starter-parent 代码
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.11</version>
    <relativePath/><!--lookup parent from repository -->
</parent>
```

在 IDEA 开发环境中,通过按住 Ctrl 键并同时单击 spring-boot-starter-parent,可以进入其对应的 pom.xml 文件进行查看。通过查看可知,spring-boot-starter-parent 作为一个父项目,它继承了 spring-boot-dependencies 作为其父依赖管理,代码如下:

```
//第 3 章/3.2.1 继承 spring-boot-dependencies 代码
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-dependencies</artifactId>
    <version>2.6.11</version>
</parent>
```

在 IDEA 开发环境中,按住 Ctrl 键并单击 spring-boot-dependencies,即可进入其对应的 pom.xml 文件进行查看。在该文件的 properties 部分,开发者可以发现许多与依赖管理相关的版本号配置,spring-boot-dependencies 管理版本号如图 3-1 所示。

这也是所有 Spring Boot 的 Starter 无须指定版本号的原因,但如果开发者不想使用默认版本号,则可以在项目中使用 property 的方式覆盖原有依赖项。

本节提到,企业级开发很少使用 spring-boot-starter-parent 作为依赖管理,因为企业通常定义自己的 parent,因此,在这种情况下,继承 spring-boot-starter-parent 并不适用。为了解决这个问题,开发者可以在 dependencyManagement 部分使用<scope>import</scope>的方式进行依赖管理,代码如下:

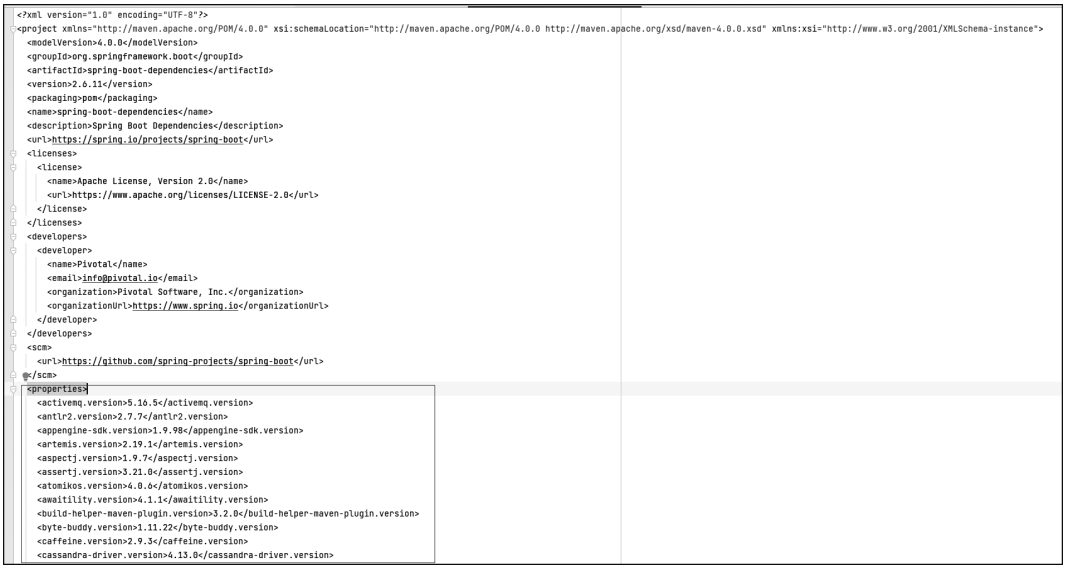

图 3-1 spring-boot-dependencies 管理版本号

```
//第 3 章/3.2.1 在 dependencyManagement 里面导入 import 代码
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-parent</artifactId>
            <version>2.6.11</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

然而,这种方法也存在一定的限制,因为它无法直接覆盖原始的依赖项配置。为了解决这个问题,开发者可以采用一种相对复杂的策略:将之前引入的依赖项移至自己的 dependencyManagement 部分,并使用 spring-boot-dependencies 进行替换,然而,这种方法在企业级开发中并不常见。

除了继承 spring-boot-dependencies 之外,spring-boot-starter-parent 还添加了一些默认配置,例如设置 JDK 版本、使用占位符@、指定编译和打包时使用的 JDK 版本及将编码设置为 UTF-8。具体示例,代码如下:

```
//第 3 章/3.2.1 spring-boot-starter-parent 默认配置代码
<properties>
    <java.version>1.8</java.version>
    <resource.delimiter>@</resource.delimiter>
    <maven.compiler.source>${java.version}</maven.compiler.source>
```

```
    <maven.compiler.target>${java.version}</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      < project. reporting. outputEncoding > UTF - 8 </project. reporting.
outputEncoding>
</properties>
```

此外,还设置了默认读取的配置文件目录和文件,以减少每个微服务都需要开发者自行设置配置文件目录和文件的工作量,代码如下:

```
//第 3 章/3.2.1 读取配置文件目录和文件代码
<resources>
    <resource>
        <directory>${basedir}/src/main/resources</directory>
        <filtering>true</filtering>
        <includes>
        <include>**/application*.yml</include>
        <include>**/application*.yaml</include>
        <include>**/application*.properties</include>
        </includes>
    </resource>
    <resource>
        <directory>${basedir}/src/main/resources</directory>
        <Excludes>
        <Exclude>**/application*.yml</Exclude>
        <Exclude>**/application*.yaml</Exclude>
        <Exclude>**/application*.properties</Exclude>
        </Excludes>
    </resource>
</resources>
```

spring-boot-starter-parent 还覆盖了 spring-boot-dependencies 中的某些插件。

3.2.2 使用 spring-boot-dependencies

spring-boot-dependencies 同样是通过继承 parent 和导入(import)实现的。

第 1 种方式是通过继承 Parent 实现,代码如下:

```
//第 3 章/3.2.2 继承 parent 代码
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-dependencies</artifactId>
    <version>2.6.11</version>
    <relativePath/><!--lookup parent from repository -->
</parent>
```

第 2 种方式是通过导入(import)的方式实现,代码如下:

```
//第 3 章/3.2.2 导入 import 代码
<dependencyManagement>
    <dependencies>
```

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>2.6.11</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

在引用 spring-boot-starter-web 时，可以省略版本号，具体示例，代码如下：

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

以上所述的依赖项涉及 Web 模块，该模块包含大量相关依赖，例如 Spring 相关库和内置的 Tomcat 服务器等。有了这些依赖，开发人员就能够使用 Spring Boot 进行 Web 开发。

Spring Boot 的 spring-boot-dependencies 引入了许多插件。在此，将简要介绍 3 个关键插件：maven-help-plugin 插件用于获取帮助信息；maven-resources-plugin 插件用于处理资源文件；maven-compiler-plugin 插件用于编译 Java 代码，具体示例，代码如下：

```
//第 3 章/3.2.2 maven-help-plugin 插件代码
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-help-plugin</artifactId>
    <version>${maven-help-plugin.version}</version>
</plugin>
```

xml-maven-plugin 插件是用于处理 XML 的 Maven 插件，代码如下：

```
//第 3 章/3.2.2 xml-maven-plugin 插件代码
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>xml-maven-plugin</artifactId>
    <version>${xml-maven-plugin.version}</version>
</plugin>
```

build-helper-maven-plugin 插件可以用来设置主源代码、测试源代码、主资源文件、测试资源文件等目录，代码如下：

```
//第 3 章/3.2.2 build-helper-maven-plugin 插件代码
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <version>${build-helper-maven-plugin.version}</version>
</plugin>
```

综上可得出结论，spring-boot-dependencies 插件的主要作用是管理依赖项的版本号、

管理插件的版本号及引入辅助插件。

3.2.3 使用 io.spring.platform

io.spring.platform 作为 Spring Boot 的基础平台,承担着继承 spring-boot-starter-parent 的角色。同时,spring-boot-starter-parent 继承了 spring-boot-dependencies,共同构成了 Spring Boot 项目的根依赖管理。在日常开发中,开发者经常需要处理多个依赖项的集成,很可能会遇到版本冲突或不兼容的问题。

为满足这种需求,一个重要目标是将已经过集成测试的依赖项整合在一起。由于这些依赖项都经过了全面的集成测试,因此在使用过程中出现问题的概率相对较低。这也是 io.spring.platform 诞生的背景。

实现这一目标的方式是通过继承 parent 和使用 import,以确保项目的依赖管理得到统一和优化。这种方法有助于简化开发者的工作流程,减少潜在的问题,并提高应用程序的稳定性和性能。

第 1 种方式是继承 parent,代码如下:

```
//第 3 章/3.2.3 继承 parent 代码
<parent>
    <groupId>io.spring.platform</groupId>
    <artifactId>platform-bom</artifactId>
    <version>Brussels-SR7</version>
</parent>
```

这种方式的缺点在于,需要明确地添加插件,因为它需要继承一些插件管理。以 Spring Boot 为例,需要显式地添加插件,代码如下:

```
//第 3 章/3.2.3 显式地添加 plugin 代码
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

第 2 种方式是通过导入(import)实现,代码如下:

```
//第 3 章/3.2.3 导入 import 代码
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.spring.platform</groupId>
            <artifactId>platform-bom</artifactId>
            <version>Brussels-SR6</version>
            <type>pom</type>
```

```
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Spring Boot 已集成了许多开源框架，旨在帮助开发者简化第三方依赖管理，然而，在实际开发过程中，仍有很多依赖未包含在内。在大型互联网项目中，各个模块之间的关系往往错综复杂，维护工作可能变得枯燥乏味且具有较高的工作量。

为解决这一问题，io.spring.platform 应运而生，它有助于连接各个依赖。例如，假设开发者需要升级某个依赖，只需更新相应的版本，无须担心版本兼容性问题。如今，一些大型互联网项目会维护自己的基础项目 platform。

3.3 手写一个简易版的 Spring Boot

许多开发者渴望了解 Spring Boot 框架的内部运行机制，但由于阅读源码能力有限，难以深入理解其底层工作原理。为帮助读者更好地理解 Spring Boot 框架，本节将通过手写一个简易版本的 Spring Boot 来阐述其底层运行原理。在学习本节内容之前，建议读者先熟悉 Spring MVC 的工作流程及 Spring IOC 的控制反转概念。

3.3.1 Java 代码直接启动 Tomcat

Spring Boot 框架以内置的 Tomcat 作为其 Web 容器，为 Web 应用提供服务，这是 Spring Boot 的一个显著特点。本节将通过一个简化的示例工程展示如何启动 Tomcat。

1. 工程介绍

先创建一个名为 simple-springboot 的父工程，然后构建两个模块：springboot-module 和 user-module。springboot-module 模块定义了一个自定义注解 ExampleSpringBootApplication，并创建了启动类 ExampleSpringApplication。user-module 模块在 UserApplication 中引入了 springboot-module 模块的注解和启动类。最后，通过 main 方法运行 user-module 模块。项目结构图如图 3-2 所示。

simple-springboot 的父工程需要将 springboot-module 模块和 user-module 模块添加到其依赖中。父工程的 pom.xml 文件示例，代码如下：

```
//第 3 章/3.3.1 在 simple-springboot 父工程将 springboot-module 模块和 user-module
//模块的依赖添加进来
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.example</groupId>
```

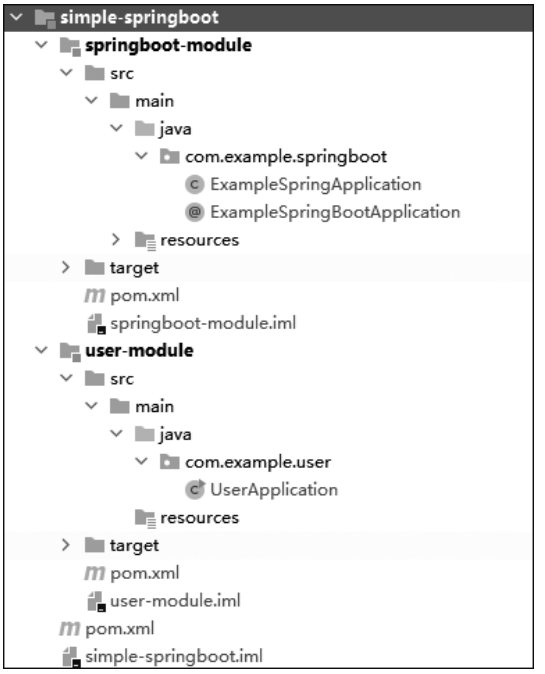

图 3-2　项目结构图

```
    <artifactId>simple-springboot</artifactId>
    <packaging>pom</packaging>
    <version>1.0-SNAPSHOT</version>
    <modules>
        <module>springboot-module</module>
        <module>user-module</module>
    </modules>
</project>
```

user-module 模块的 pom.xml 文件需引入 springboot-module 模块的依赖,代码如下:

```
//第 3 章/3.3.1 在 user-module 模块的 pom.xml 文件中引入 springboot-module 模块的依赖
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>simple-springboot</artifactId>
        <groupId>org.example</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>user-module</artifactId>
```

```
    <dependencies>
        <dependency>
            <groupId>org.example</groupId>
            <artifactId>springboot-module</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
</project>
```

springboot-module 模块引入的是 Spring、Servlet 及与 Tomcat 相关的依赖。其 pom.xml 文件示例，代码如下：

```
//第 3 章/3.3.1 springboot-module 模块引入了 Spring、Servlet 及与 Tomcat 相关的依赖
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>simple-springboot</artifactId>
        <groupId>org.example</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>springboot-module</artifactId>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.3.18</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.3.18</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>5.3.18</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>5.3.18</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
```

```
            <version>5.3.18</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>4.0.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-core</artifactId>
            <version>9.0.60</version>
        </dependency>
    </dependencies>
</project>
```

2. 自定义注解

参考 Spring Boot 框架,可以自定义一个注解,该注解作用于启动类上。以下是该注解的示例,代码如下:

```
//第 3 章/3.3.1 向启动类添加自定义注解
package com.example.springboot;
import java.lang.annotation.*;
/**
* 自定义注解:启动类注解
*/
@Target(ElementType.TYPE)//作用于类上面
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited//一个类用上了@Inherited 修饰的注解子类继承这个注解
public @interface ExampleSpringBootApplication {
}
```

3. 自定义启动类

当浏览器发送请求时,需要启动 Tomcat 才能接受请求。以下示例用于展示如何启动 Tomcat,代码如下:

```
//第 3 章/3.3.1 自定义启动类
package com.example.springboot;
import org.apache.catalina.*;
import org.apache.catalina.connector.Connector;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.core.StandardEngine;
import org.apache.catalina.core.StandardHost;
import org.apache.catalina.startup.Tomcat;
import org.springframework.context.annotation.Bean;
import org.springframework.web.context.WebApplicationContext;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;
```

```
import java.util.Map;
/**
* 自定义启动类
*/
public class ExampleSpringApplication {
    public static void run(Class clazz){
        startTomcat();
    }
    private static void startTomcat(){
         Tomcat tomcat =new Tomcat();
         Server server =tomcat.getServer();
         Service service =server.findService("Tomcat");
         Connector connector =new Connector();
         connector.setPort(8080);
         Engine engine =new StandardEngine();
         engine.setDefaultHost("localhost");
         Host host =new StandardHost();
         host.setName("localhost");
         String contextPath ="";
         Context context =new StandardContext();
         context.setPath(contextPath);
         context.addLifecycleListener(new Tomcat.FixContextListener());
         host.addChild(context);
         engine.addChild(host);
         service.setContainer(engine);
         service.addConnector(connector);
         try {
             tomcat.start();
         } catch (LifecycleException e) {
             e.printStackTrace();
         }
    }
}
```

4. 自定义启动类

user-module 模块的启动类应该使用自定义的注解和自定义的启动类，在 main 方法中运行 run 方法。以下是该启动类的示例，代码如下：

```
//第 3 章/3.3.1 user-module 模块的启动类
package com.example.user;
import com.example.springboot.ExampleSpringApplication;
import com.example.springboot.ExampleSpringBootApplication;
@ExampleSpringBootApplication
public class UserApplication {
     public static void main(String[] args) {
        ExampleSpringApplication.run(UserApplication.class);
     }
}
```

5. 运行项目

启动项目后,可以在控制台查看日志,如图 3-3 所示。

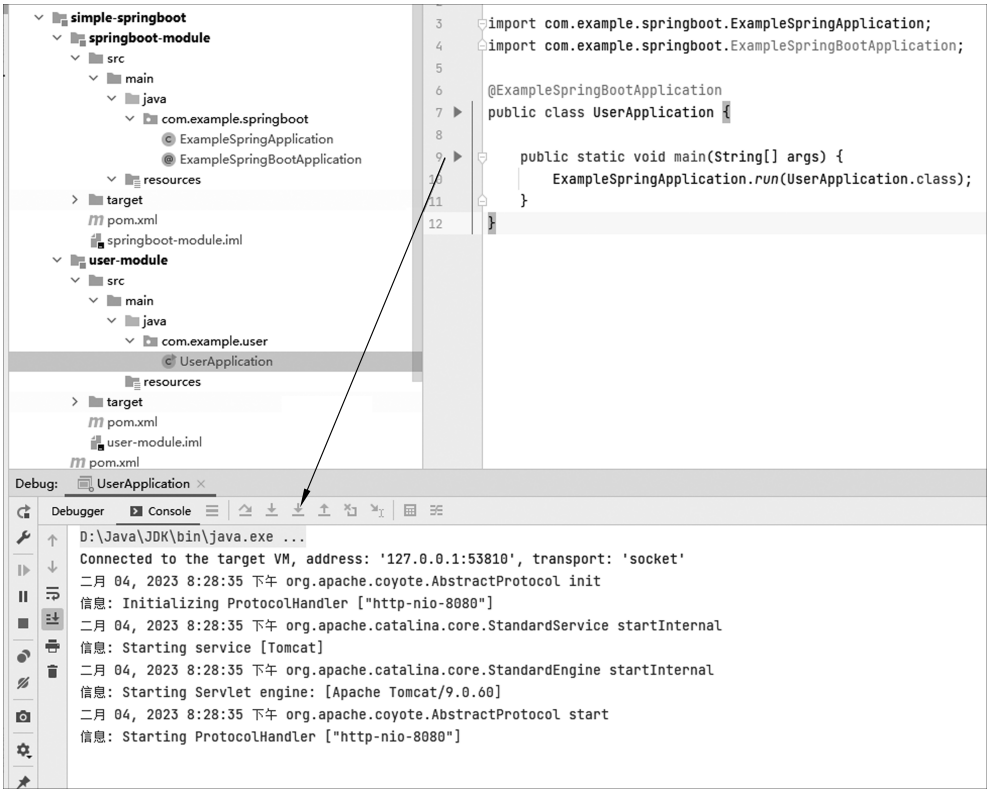

图 3-3 控制台日志

6. 请求处理流程

既然 Tomcat 已经成功启动,那么接下来就需要处理请求了。以下是处理请求的示例,代码如下:

```
tomcat.addServlet(contextPath,"dispatcher",new
DispatcherServlet(webApplicationContext));
context.addServletMappingDecoded("/*","dispatcher");//拦截所有请求给
//DispatcherServlet 处理
```

对于熟悉 Spring MVC 工作流程的开发人员可能会立即联想到前端控制器 DispatcherServlet。在该框架中,所有的请求都会先经过 DispatcherServlet 进行处理。

注意:为了防止一些读者不熟悉 Spring MVC 的工作流程,本节通过一个小故事进行阐述。在丰富多彩的互联网世界中,有一个名叫 Spring MVC 的小镇,其居民热情好客。这个小镇充满生机和活力,居民和谐相处,共同分享美好生活。某一天,一位名叫“请求”的游客慕名而来,想要参加一场特别的盛会。在这场盛会中,Spring MVC 小镇的居民需要设计出最美

的舞蹈和最华丽的舞池，以期望能够迎接这场难得的盛会。

作为小镇的前端控制，DispatcherServlet 以谨慎的态度接待请求。当收到请求时，DispatcherServlet 认识到其重要性，首先对请求进行详尽解析，将 URL 转换为 URI，并调用 HandlerMapping 将请求与相关的控制器和拦截器联系在一起。DispatcherServlet 将联系在一起的对象封装为 HandlerExecutionChain 对象，并选择一个适宜的 HandlerAdapter 处理请求。这位 HandlerAdapter 犹如专业的舞蹈教练，娴熟地将请求转换为优美的舞蹈。

在处理请求的过程中，HandlerAdapter 承担着关键的辅助工作，如数据转换、数据验证和消息转换，确保游客的请求以最佳形式呈现。经过一番努力，Handler 完成任务，向 DispatcherServlet 返回一个包含视图名的 ModelAndView 对象，表明舞池和舞蹈已经准备就绪。DispatcherServlet 根据 ModelAndView 对象中的视图名，选择合适的 ViewResolver 解析视图。这位 ViewResolver 犹如熟练掌握各种舞蹈的舞蹈指导，能够将游客的请求转换为各种美丽的舞蹈画面。

在解析视图名的过程中，ViewResolver 找到了一个最匹配的视图——一个名为 View 的小镇居民。View 利用模型数据（如舞池和舞蹈），渲染出美丽的舞蹈画面。在渲染过程中，View 与模型数据完美结合，精心布置了舞池，编排了一支优雅的舞蹈。游客们被这美丽的舞蹈所吸引，纷纷拿出相机记录下这难忘的时刻。渲染结束后，View 将渲染后的舞蹈画面传递给 DispatcherServlet。DispatcherServlet 将这些画面呈现在游客们的眼前，引发他们的欢呼雀跃，感叹小镇居民们的才华与热情。舞会结束后，游客们与 Spring MVC 小镇的居民们建立了深厚的友谊，这段美好的回忆成为永恒的佳话。从此，Spring MVC 小镇名声远扬，吸引了越来越多的游客，Spring MVC 的故事，也成了互联网世界中一段不朽的传奇。

在本流程中，DispatcherServlet 会对请求的 URL 进行解析，并查找对应的 Controller 方法。事实上，每个 Controller 都是 Spring 容器中的一个 Bean，因此，为了处理请求，需要将一个 Spring 容器传递给 DispatcherServlet。那么，这个容器从何而来呢？

通过查阅 Spring MVC 源代码，找到 DispatcherServlet 的有参数构造方法，代码如下：

```
public DispatcherServlet(WebApplicationContext webApplicationContext) {
    super(webApplicationContext);
    this.setDispatchOptionsRequest(true);
}
```

在这种方法中，需要使用 WebApplicationContext 容器。了解应该使用哪种容器后，便可以直接创建一个 Spring 容器，并将 UserApplication 启动类注册进来，代码如下：

```
//第 3 章/3.3.1  自定义启动类
package com.example.springboot;
import org.apache.catalina.*;
import org.apache.catalina.connector.Connector;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.core.StandardEngine;
import org.apache.catalina.core.StandardHost;
```

```
import org.apache.catalina.startup.Tomcat;
import org.springframework.context.annotation.Bean;
import org.springframework.web.context.WebApplicationContext;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;
import java.util.Map;
/**
* 自定义启动类
*/
public class ExampleSpringApplication {
    public static void run(Class clazz){
        //创建一个 Spring 容器
           AnnotationConfigWebApplicationContext  webApplicationContext  = new
AnnotationConfigWebApplicationContext();
        //注册启动类
        webApplicationContext.register(clazz);
        webApplicationContext.refresh();
        //启动 Tomcat
        startTomcat(webApplicationContext);
    }
    private static void startTomcat(WebApplicationContext webApplicationContext){
        Tomcat tomcat =new Tomcat();
        Server server =tomcat.getServer();
        Service service =server.findService("Tomcat");
        Connector connector =new Connector();
        connector.setPort(8080);
        Engine engine =new StandardEngine();
        engine.setDefaultHost("localhost");
        Host host =new StandardHost();
        host.setName("localhost");
        String contextPath ="";
        Context context =new StandardContext();
        context.setPath(contextPath);
        context.addLifecycleListener(new Tomcat.FixContextListener());
        host.addChild(context);
        engine.addChild(host);
        service.setContainer(engine);
        service.addConnector(connector);
        tomcat.addServlet(contextPath,"dispatcher",new DispatcherServlet
(webApplicationContext));
        //拦截所有请求,交给 DispatcherServlet 处理
        context.addServletMappingDecoded("/*","dispatcher");
        try {
            tomcat.start();
        } catch (LifecycleException e) {
            e.printStackTrace();
        }
    }
}
```

这样，在启动 Tomcat 时，容器会解析 UserApplication 类，并解析其上的自定义注解 ExampleSpringBootApplication，然后开发者将 @ ComponentScan 注解添加到 ExampleSpringBootApplication 中，代码如下：

```
//第 3 章/3.3.1  自定义注解
package com.example.springboot;
import org.springframework.context.annotation.ComponentScan;
import java.lang.annotation.*;
/**
* 自定义注解:启动类注解
*/
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
//@Inherited 是一个标识,用来修饰注解
@Inherited
//扫描 UserApplication 类所在的包路径
@ComponentScan
public @interface ExampleSpringBootApplication {
}
```

由于没有指定具体的扫描路径，所以容器会扫描 ExampleSpringBootApplication 注解作用的类 UserApplication，并解析其包路径 com.example.user。进一步将扫描范围扩大到该包下的所有 Controller。

为了测试上述处理过程，在 user-module 模块下创建一个 TestController，并编写一个简单的接口，代码如下：

```
//第 3 章/3.3.1  TestController 类
package com.example.user.controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class TestController {
    @GetMapping("/test")
    public String test(){
        return "test";
    }
}
```

运行项目，通过浏览器访问接口校验代码的正确性，如图 3-4 所示。

为了更好地理解整个流程，下面通过脑图进行总结，如图 3-5 所示。

3.3.2 多态实现 WebServer

在 3.3.1 节中，Spring Boot 使用的是一种固定的方式来启动 Tomcat，无法切换到其他 Web 容器，例如 Jetty。假设项目需要实现切换到 Jetty 容器的功能，应该如何实现呢？

首先，提供 Tomcat 和 Jetty 的依赖，然后根据依赖情况来确定项目中使用的是

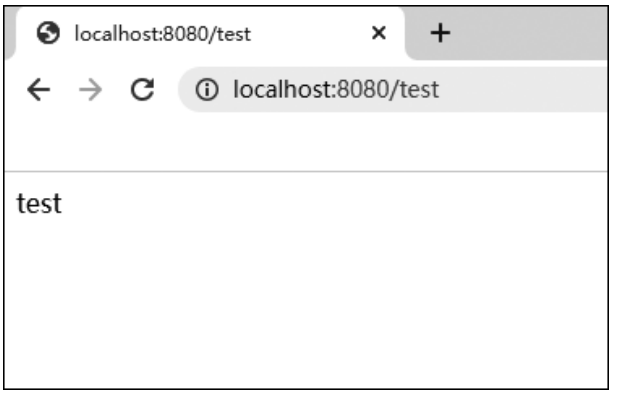

图 3-4　浏览器请求接口

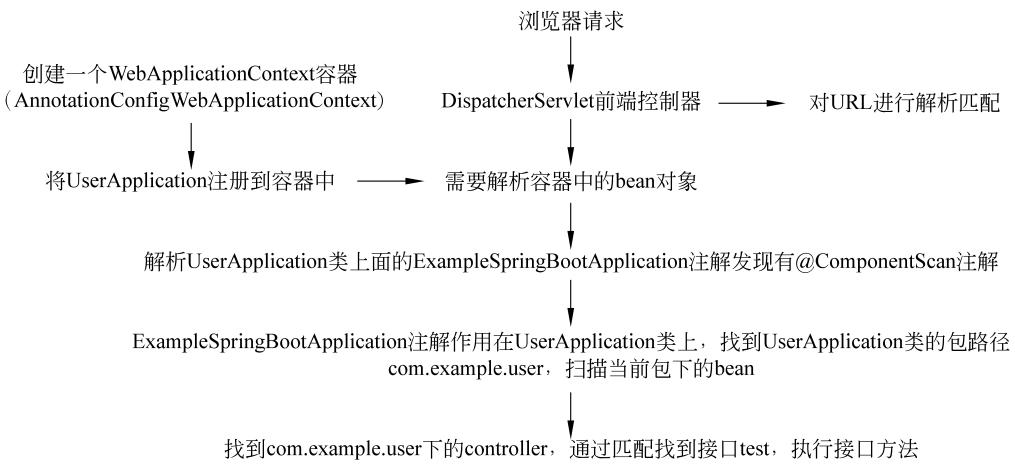

图 3-5　请求处理流程脑图

TomcatWebServer 还是 JettyWebServer 的 Bean，进而决定使用哪种 Web 容器进行执行。在确保项目可以正常切换 Web 容器之后，再进行代码优化。这就是实现这一功能的基本思路。下面对 3.3.1 节的代码进行修改。

1. 引入 Jetty 依赖

在引入 Tomcat 依赖的 springboot-module 模块中，若要引入 Jetty 依赖，则需要在依赖中添加配置<optional>true</optional>，表示该依赖不会被传递给调用服务，即 user-module 服务。由于 springboot-module 模块需要支持多种 Web 容器(Tomcat/Jetty)，所以调用端只能使用其中一种，否则会出现错误，代码如下：

```
//第 3 章/3.3.2 引入 Tomcat 和 Jetty 依赖
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-core</artifactId>
    <version>9.0.60</version>
```

```
</dependency>
<dependency>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-server</artifactId>
    <version>9.4.48.v20220622</version>
    <optional>true</optional>
</dependency>
```

Jetty 依赖应添加至 springboot-module 模块中，而 user-module 模块仅依赖于 springboot-module 模块，并默认使用 Tomcat 依赖，因此，Jetty 依赖无法传递至 user-module 模块。如果 user-module 模块需要使用 Jetty 依赖，就需要在 springboot-module 模块中排除 Tomcat 依赖，并添加 Jetty 依赖，代码如下：

```
//第 3 章/3.3.2 排除 Tomcat 依赖
<dependency>
    <groupId>org.example</groupId>
    <artifactId>springboot-module</artifactId>
    <version>1.0-SNAPSHOT</version>
    <!--排除 Tomcat 依赖-->
    <Excelusions>
        <Excelusion>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-core</artifactId>
        </Excelusion>
    </Excelusions>
</dependency>
<!--引入 Jetty 依赖-->
<dependency>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-server</artifactId>
    <version>9.4.48.v20220622</version>
</dependency>
```

2. 创建 WebServer 接口

已知该项目需要同时使用 Tomcat 和 Jetty 容器，为了避免在后续引入其他 Web 容器时产生混淆，开发者定义了一个 WebServer 接口，用于抽象出 Web 容器的启动功能，代码如下：

```
//第 3 章/3.3.2 创建 WebServer 接口
package com.example.springboot;
import org.springframework.web.context.WebApplicationContext;
/**
* Web 服务接口
*/
public interface WebServer {
public void start(WebApplicationContext applicationContext);
}
```

3. 创建 TomcatWebServer 实现类

将 startTomcat 方法的实现移至实现类的 start 方法中,代码如下:

```
//第 3 章/3.3.2 创建 TomcatWebServer 实现类
package com.example.springboot;
import org.apache.catalina.*;
import org.apache.catalina.connector.Connector;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.core.StandardEngine;
import org.apache.catalina.core.StandardHost;
import org.apache.catalina.startup.Tomcat;
import org.springframework.web.context.WebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;
/**
* Web 服务:启动 Tomcat 相关代码
*/
public class TomcatWebServer implements WebServer{
    @Override
    public void start(WebApplicationContext applicationContext) {
        System.out.println("============启动 Tomcat============");
        Tomcat tomcat =new Tomcat();
        Server server =tomcat.getServer();
        Service service =server.findService("Tomcat");
        Connector connector =new Connector();
        connector.setPort(9081);
        Engine engine =new StandardEngine();
        engine.setDefaultHost("localhost");
        Host host =new StandardHost();
        host.setName("localhost");
        String contextPath ="";
        Context context =new StandardContext();
        context.setPath(contextPath);
        context.addLifecycleListener(new Tomcat.FixContextListener());
        host.addChild(context);
        engine.addChild(host);
        service.setContainer(engine);
        service.addConnector(connector);
        tomcat.addServlet(contextPath, "dispatcher", new
DispatcherServlet(applicationContext));
        context.addServletMappingDecoded("/*", "dispatcher");
        try {
            tomcat.start();
        } catch (LifecycleException e) {
            e.printStackTrace();
        }
    }
}
```

4. 创建 JettyWebServer 实现类

Jetty 启动过程,代码如下:

```
//第 3 章/3.3.2 创建 JettyWebServer 实现类
package com.example.springboot;
import org.springframework.web.context.WebApplicationContext;
/**
* Web 服务:启动 Jetty 相关代码
*/
public class JettyWebServer implements WebServer{
    @Override
    public void start(WebApplicationContext applicationContext) {
        System.out.println("启动 Jetty");
        //省略 Jetty 启动过程的代码,不重点讲解
    }
}
```

5. 改造 ExampleSpringApplication 类

获取 Tomcat 或者 Jetter 容器,代码如下:

```
//第 3 章/3.3.2 改造 ExampleSpringApplication 类
package com.example.springboot;
import org.springframework.web.context.WebApplicationContext;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import java.util.Map;
/**
* 自定义启动类
*/
public class ExampleSpringApplication {
    public static void run(Class clazz){
    //创建一个 Spring 容器 AnnotationConfigWebApplicationContext 支持 SpringMVC
        AnnotationConfigWebApplicationContext applicationContext =new
AnnotationConfigWebApplicationContext();
        applicationContext.register(clazz);//注册一个类进来
        applicationContext.refresh();
        //启动 Web 服务器(Tomcat、Jetty)
        WebServer webServer =getWebServer(applicationContext);
        webServer.start(applicationContext);
    }
    /**
    * 获取 Web 服务:获取 Tomcat 或者 Jetty 容器
    * @param applicationContext
    * @return
    */
    private static WebServer getWebServer(WebApplicationContext applicationContext) {
        Map<String, WebServer>beansOfType =applicationContext.getBeansOfType
(WebServer.class);
        //两个都没有定义: UserApplication 类中没有定义 TomcatWebServer 或者
        //JettyWebServer
```

```
            if (beansOfType.size() ==0) {
                throw new NullPointerException();
            }
            //定义了两个: UserApplication 类中有定义 TomcatWebServer 和 JettyWebServer
            //会报错
            if (beansOfType.size() >1) {
                throw new IllegalStateException();
            }
            //定义第 1 个: UserApplication 类中有定义 TomcatWebServer 或者 JettyWebServer
            //其中一个
            return beansOfType.values().stream().findFirst().get();
        }
    }
```

6. 改造 UserApplication 类

在 UserApplication 类中只能定义 TomcatWebServer 或 JettyWebServer 其中之一,代码如下:

```
//第 3 章/3.3.2 改造 UserApplication 类
package com.example.user;
import com.example.springboot.ExampleSpringApplication;
import com.example.springboot.ExampleSpringBootApplication;
import com.example.springboot.JettyWebServer;
import com.example.springboot.TomcatWebServer;
import org.springframework.context.annotation.Bean;
/**
* TomcatWebServer 和 JettyWebServer 只能定义其中一个。弊端:比较麻烦,需要有一个自动
的配置类识别我要用什么类型的 Web 服务容器。解决方案:WebServerAutoConfiguration
*/
@ExampleSpringBootApplication
public class UserApplication {
    //@Bean
    //public TomcatWebServer tomcatWebServer(){
    //return new TomcatWebServer();
    //}
    @Bean
    public JettyWebServer jettyWebServer(){
        return new JettyWebServer();
    }
    public static void main(String[] args) {
        ExampleSpringApplication.run(UserApplication.class);
    }
}
```

7. 运行项目

启动项目并查看日志,如图 3-6 所示。

读者可以注释 JettyWebServer 或 TomcatWebServer,或同时使用它们,以检查功能是否可实现。