

# SpringBoot 框架

---

SSM框架内容分为如下几个章节，每个章节对应一个文件：《Maven》、《Spring》、《MyBatis》、《SpringMVC》、《SSM整合》、《SpringBoot》、《MyBatis-Plus》。

## 第六章：SpringBoot 框架

---

### 一、SpringBoot3 概述

#### (1) SpringBoot 介绍

[Spring Boot :: Spring Boot](#)

SpringBoot是一种快速启动框架，可以帮我们简单、快速地创建一个独立的、生产级别的 Spring 应用（SpringBoot 底层是 Spring），大多数 SpringBoot 应用只需要编写少量配置即可快速整合 Spring 平台以及第三方技术。

SpringBoot 的主要目标是：

- 为所有 Spring 开发提供更快速、可广泛访问的入门体验；
- 开箱即用，设置合理的默认值，但是也可以根据需求进行适当的调整；
- 提供一系列大型项目通用的非功能性程序（如嵌入式服务器、安全性、指标、运行检查等）；
- 约定大于配置，基本不需要主动编写配置类、也不需要 XML 配置文件。

总结：简化开发，简化配置，简化整合，简化部署，简化监控，简化运维。

#### (2) 快速入门 SpringBoot

##### 1. 简单操作

需求：向浏览器发送 `/hello` 请求，返回"Hello, Spring Boot 3!"。

创建空项目：springboot-part;

创建子工程：springboot-base-quick-1；

在子工程导入配置与依赖：

SpringBoot可以帮我们方便的管理项目依赖，在Spring Boot提供了一个名为 `spring-boot-starter-parent` 的工程，里面已经对各种常用依赖的版本进行了管理，我们的项目需要以这个项目为父工程，这样我们就不用操心依赖的版本问题了，需要什么依赖，直接引入坐标(不需要添加版本)即可。

为了让Spring Boot帮我们完成各种自动配置，我们必须引入Spring Boot提供的自动配置依赖，我们称为启动器。因为我们是web项目，这里我们引入web启动器，在 `pom.xml` 文件中加入如下依赖：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-
4 instance"
5         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
6 http://maven.apache.org/xsd/maven-4.0.0.xsd">
7     <modelVersion>4.0.0</modelVersion>
8
9     <parent>
10         <!-- spring boot -->
11         <groupId>org.springframework.boot</groupId>
12         <artifactId>spring-boot-starter-
13 parent</artifactId>
14         <version>3.0.5</version>
15     </parent>
16
17     <groupId>com.ssh</groupId>
18     <artifactId>springboot-base-quick-1</artifactId>
19     <version>1.0-SNAPSHOT</version>
20
21     <!-- 导入对应的启动器 -->
22     <dependencies>
23         <dependency>
24
25             <groupId>org.springframework.boot</groupId>
```

```
22         <artifactId>spring-boot-starter-  
web</artifactId>  
23         </dependency>  
24     </dependencies>  
25  
26 </project>
```

创建启动类:

```
1  package com.ssh;  
2  
3  import org.springframework.boot.SpringApplication;  
4  import  
    org.springframework.boot.autoconfigure.SpringBootApplication;  
5  
6  /**  
7   * @author 申书航  
8   * @version 1.0  
9   */  
10 @SpringBootApplication // 代表该类是一个Spring Boot的启动  
    类  
11 public class Main {  
12  
13     public static void main(String[] args) {  
14         //创建IoC容器, 加载配置, 启动内置的WEB服务器  
15         SpringApplication.run(Main.class, args); //  
            自动创建IoC容器, 启动Tomcat服务器  
16     }  
17 }
```

Controller 层:

```
1  package com.ssh.controller;  
2  
3  import  
    org.springframework.web.bind.annotation.GetMapping;  
4  import  
    org.springframework.web.bind.annotation.RequestMapping  
    ;
```

```

5  import
   org.springframework.web.bind.annotation.RestController
   ;
6
7  /**
8   * @author 申书航
9   * @version 1.0
10  */
11  @RestController
12  @RequestMapping("/hello")
13  public class HelloController {
14
15      @GetMapping("boot")
16      public String hello() {
17          return "Hello, Spring Boot 3!";
18      }
19  }

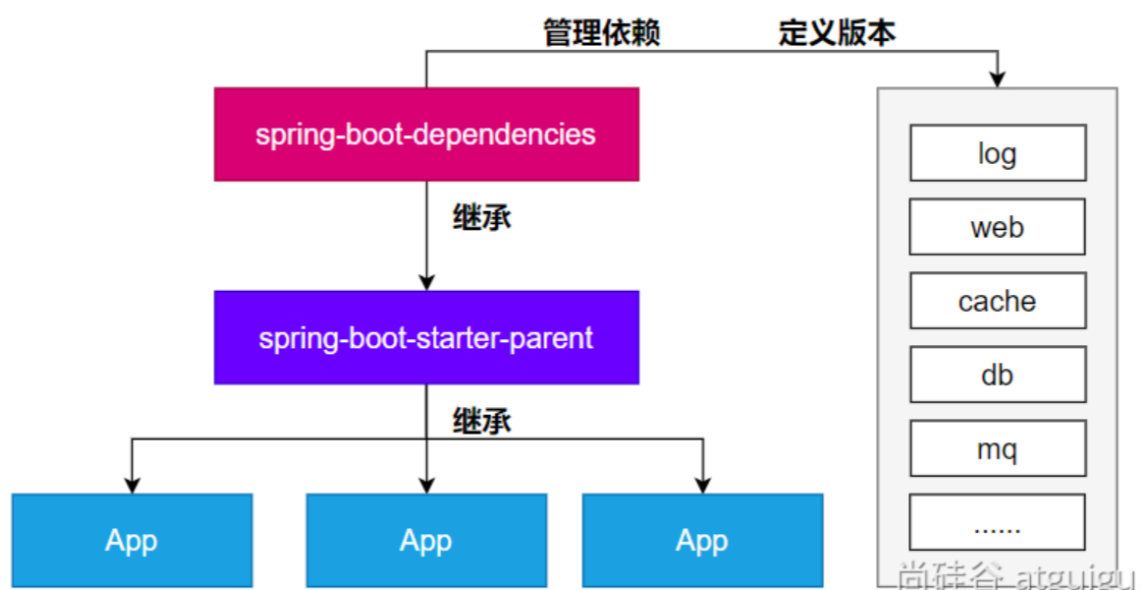
```

访问地址测试即可。

## 2. 入门总结

依赖不需要写版本：

- 每个boot项目都有一个父项目 `spring-boot-starter-parent`；
- parent的父项目是 `spring-boot-dependencies`；
- 父项目版本仲裁中心，把所有常见的jar的依赖版本都声明好了。

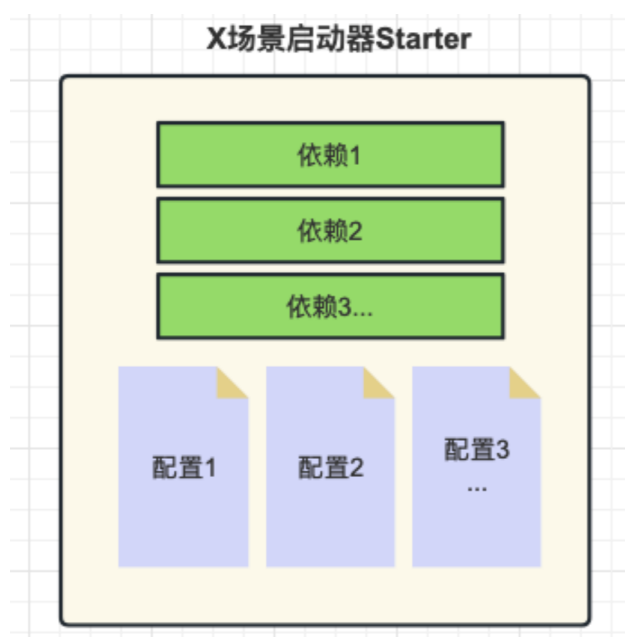


启动器 (Starter) 的概念：

Spring Boot提供了一种叫做Starter的概念，它是一组预定义的依赖项集合，旨在简化Spring应用程序的配置和构建过程。Starter包含了一组相关的依赖项，以便在启动应用程序时自动引入所需的库、配置和功能。

主要作用如下：

1. 简化依赖管理：Spring Boot Starter通过捆绑和管理一组相关的依赖项，减少了手动解析和配置依赖项的工作。只需引入一个相关的Starter依赖，即可获取应用程序所需的全部依赖；
2. 自动配置：Spring Boot Starter在应用程序启动时自动配置所需的组件和功能。通过根据类路径和其他设置的自动检测，Starter可以自动配置Spring Bean、数据源、消息传递等常见组件，从而使应用程序的配置变得简单和维护成本降低；
3. 提供约定优于配置：Spring Boot Starter遵循“约定优于配置”的原则，通过提供一组默认设置和约定，减少了手动配置的需要。它定义了标准的配置文件命名约定、默认属性值、日志配置等，使得开发者可以更专注于业务逻辑而不是繁琐的配置细节；
4. 快速启动和开发应用程序：Spring Boot Starter使得从零开始构建一个完整的Spring Boot应用程序变得容易。它提供了主要领域（如Web开发、数据访问、安全性、消息传递等）的Starter，帮助开发者快速搭建一个具备特定功能的应用程序原型；
5. 模块化和可扩展性：Spring Boot Starter的组织结构使得应用程序的不同模块可以进行分离和解耦。每个模块可以有自己的Starter和依赖项，使得应用程序的不同部分可以按需进行开发和扩展。



Spring Boot提供了许多预定义的Starter，例如spring-boot-starter-web用于构建Web应用程序，spring-boot-starter-data-jpa用于使用JPA进行数据库访问，spring-boot-starter-security用于安全认证和授权等等。

使用Starter非常简单，只需要在项目的构建文件（例如Maven的pom.xml）中添加所需的Starter依赖，Spring Boot会自动处理依赖管理和配置。

通过使用Starter，开发人员可以方便地引入和配置应用程序所需的功能，避免了手动添加大量的依赖项和编写冗长的配置文件的繁琐过程。同时，Starter也提供了一致的依赖项版本管理，确保依赖项之间的兼容性和稳定性。

spring boot提供的全部启动器地址：

<https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.build-systems.starters>

命名规范：

- 官方提供的场景：命名为：spring-boot-starter-\*
- 第三方提供场景：命名为：\*-spring-boot-starter

`@SpringBootApplication` 注解：

`@SpringBootApplication` 添加到启动类上，是一个组合注解，他的功效有具体的子注解实现。

```
1  @SpringBootConfiguration    //配置类
2  @EnableAutoConfiguration    //自动加载其他配置类
3  @ComponentScan              //扫描包，默认扫描当前类所在的包
    及其子包
4  public @interface SpringBootApplication {}
```

`@SpringBootApplication` 注解是Spring Boot框架中的核心注解，它的主要作用是简化和加速Spring Boot应用程序的配置和启动过程。

具体而言，`@SpringBootApplication` 注解起到以下几个主要作用：

1. 自动配置：`@SpringBootApplication` 注解包含了`@EnableAutoConfiguration` 注解，用于启用Spring Boot的自动配

置机制。自动配置会根据应用程序的依赖项和类路径，自动配置各种常见的Spring配置和功能，减少开发者的手动配置工作。它通过智能地分析类路径、加载配置和条件判断，为应用程序提供适当的默认配置；

2. 组件扫描：@SpringBootApplication 注解包含了 @ComponentScan 注解，用于自动扫描并加载应用程序中的组件，例如控制器（Controllers）、服务（Services）、存储库（Repositories）等。它默认会扫描 @SpringBootApplication 注解所在类的包及其子包中的组件，并将它们纳入Spring Boot应用程序的上下文中，使它们可被自动注入和使用；
3. 声明配置类：@SpringBootApplication 注解本身就是一个组合注解，它包含了 @Configuration 注解，将被标注的类声明为配置类。配置类可以包含Spring框架相关的配置、Bean定义，以及其他的自定义配置。通过 @SpringBootApplication 注解，开发者可以将配置类与启动类合并在一起，使得配置和启动可以同时发生。

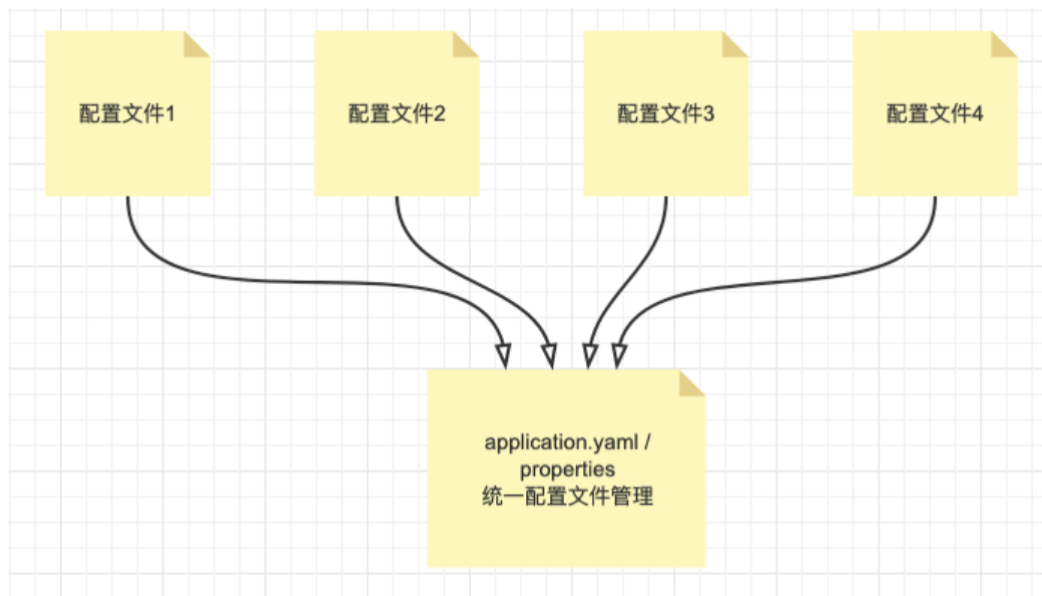
总的来说，@SpringBootApplication 注解的主要作用是简化 Spring Boot 应用程序的配置和启动过程。它自动配置应用程序、扫描并加载组件，并将配置和启动类合二为一，简化了开发者的工作量，提高了开发效率。

## 二、SpringBoot3 配置文件

### (1) 统一配置管理概述

SpringBoot工程下，进行统一的配置管理，你想设置的任何参数（端口号、项目根路径、数据库连接信息等等）都集中到一个固定位置和命名的配置文件（application.properties 或 application.yml）中。

配置文件应该放置在Spring Boot工程的 src/main/resources 目录下。这是因为src/main/resources 目录是Spring Boot默认的路径（classpath），配置文件会被自动加载并可被应用程序访问。



功能配置参数说明：

<https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties>

细节总结：

- 集中式管理配置。统一在一个文件完成程序功能参数设置和自定义参数声明；
- 位置：resources文件夹下，必须命名 application，后缀 .properties 或 .yaml 或 .yml ；
- 如果同时存在 `application.properties` 或 `application.yml(.yaml)` ， `properties` 的优先级更高；
- 配置基本都有默认值。

## (2) 属性配置文件使用

创建配置文件： `application.properties`



```
1 # application.properties 为统一配置文件
2 # 内部包含： 固定功能的key,自定义的key
3 # 此处的配置信息,我们都可以在程序中@value等注解读取
4
5 #使用springboot提供的配置，修改程序的参数，key是固定的
6 # 设置端口号
7 server.port=80
8 # 设置上下文路径
9 server.servlet.context-path=/testPath
10
11 #自定义配置，key自定义，value自定义
12 ssh.name=申书航
13 ssh.age=20
```

Controller 层更改：

```
1 package com.ssh.controller;
2
3 import
4     org.springframework.beans.factory.annotation.Value;
5 import
6     org.springframework.web.bind.annotation.GetMapping;
7 import
8     org.springframework.web.bind.annotation.RequestMapping;
9 ;
10 import
11     org.springframework.web.bind.annotation.RestController;
12 ;
13
14 /**
15  * @author 申书航
16  * @version 1.0
17  */
18 @RestController
19 @RequestMapping("/hello")
20 public class HelloController {
21
22     @Value("${ssh.name}")
23     private String name;
24
25     @GetMapping("boot")
```

```
20     public String hello() {  
21         System.out.println(name);  
22         return "Hello, Spring Boot 3!";  
23     }  
24 }
```

访问 <http://localhost:80/testPath/hello/boot> 测试。

## (3) YAML 配置文件

### 1. YAML 配置文件介绍

YAML (YAML Ain't Markup Language) 是一种基于层次结构的数据序列化格式，旨在提供一种易读、人类友好的数据表示方式。

与 `.properties` 文件相比，YAML格式有以下优势：

1. 层次结构：YAML文件使用缩进和冒号来表示层次结构，使得数据之间的关系更加清晰和直观。这样可以更容易理解和维护复杂的配置，特别适用于深层次嵌套的配置情况；
2. 自我描述性：YAML文件具有自我描述性，字段和值之间使用冒号分隔，并使用缩进表示层级关系。这使得配置文件更易于阅读和理解，并且可以减少冗余的标点符号和引号；
3. 注释支持：YAML格式支持注释，可以在配置文件中添加说明性的注释，使配置更具可读性和可维护性。相比之下，`.properties` 文件不支持注释，无法提供类似的解释和说明；
4. 多行文本：YAML格式支持多行文本的表示，可以更方便地表示长文本或数据块。相比之下，`.properties` 文件需要使用转义符或将长文本拆分为多行；
5. 类型支持：YAML格式天然支持复杂的数据类型，如列表、映射等。这使得在配置文件中表示嵌套结构或数据集合更加容易，而不需要进行额外的解析或转换；
6. 更好的可读性：由于YAML格式的特点，它更容易被人类读懂和解释。它减少了配置文件中需要的特殊字符和语法，让配置更加清晰明了，从而减少了错误和歧义。

综上所述，YAML格式相对于 `.properties` 文件具有更好的层次结构表示、自我描述性、注释支持、多行文本表示、复杂数据类型支持和更好的可读性。这些特点使YAML成为一种有力的配置文件格式，尤其适用于复杂的配置需求和人类可读的场景。然而，选择使用YAML还是 `.properties`

取决于实际需求和团队的偏好，简单的配置可以使用 `.properties`，而复杂的配置可以选择YAML以获得更多的灵活性和可读性。

## 2. YAML 配置文件语法

1. 数据结构用树形结构呈现，通过缩进来表示层级；
2. 连续的项目（集合）通过减号“-”来表示；
3. 键值结构里面的key/value对用冒号“:”来分隔；
4. YAML配置文件的扩展名是 `yaml` 或 `yml`。

配置文件： `application.yaml`

```
1  # yaml 文件有层次，可以继承配置文件的格式
2
3  server:
4      port: 80
5      servlet:
6          context-path: /boot
7
8  # 自定义配置
9  ssh:
10     info:
11         age: 20
12         name: "申书航"
13     user:
14         username: "root"
15         password: "123456"
16     # 集合类型配置
17     gfs:
18         - a
19         - b
20         - c
```

### (3) 批量配置文件注入

`@ConfigurationProperties` 是SpringBoot提供的重要注解，他可以将一些配置属性批量注入到Bean对象。

在类上通过 `@ConfigurationProperties` 注解声明该类要读取属性配置。

`prefix="spring.jdbc.datasource"` 读取属性文件中前缀为 `spring.jdbc.datasource` 的值。前缀和属性名称和配置文件中的 key 必须保持一致才可以注入成功。

配置文件: `application.yaml`

```
1  # yaml 文件有层次，可以继承配置文件的格式
2
3  server:
4      port: 808
5      servlet:
6          context-path: /boot
7
8  # 自定义配置
9  ssh:
10     info:
11         age: 20
12         name: "申书航"
13     user:
14         username: "root"
15         password: "root"
16     # 集合类型配置
17     gfs:
18         - a
19         - b
20         - c
```

启动类:

```
1  package com.ssh;
2
3  import org.springframework.boot.SpringApplication;
4  import
    org.springframework.boot.autoconfigure.SpringBootApplication;
5
6  /**
7   * @author 申书航
8   * @version 1.0
9   * 启动类
10  */
```

```

11 @SpringBootApplication
12 public class Main {
13
14     public static void main(String[] args) {
15         SpringApplication.run(Main.class, args);
16     }
17 }

```

创建实体类并批量注入配置文件:

```

1  package com.ssh.pojo;
2
3  import lombok.Data;
4  import
5  org.springframework.beans.factory.annotation.Value;
6  import
7  org.springframework.boot.context.properties.Configuration
8  Properties;
9  import org.springframework.stereotype.Component;
10
11 import java.util.List;
12
13 /**
14  * @author 申书航
15  * @version 1.0
16  * 用户实体类
17  *
18  * 读取配置文件
19  * 1. @Value: 直接在属性上添加
20  *      @Value的key值必须写全
21  *      @Value只能读取单个值
22  * 2. 批量配置读取: @ConfigurationProperties(prefix = "前
23 缀")
24  *      读取配置文件中以ssh.user开头的配置，并自动绑定到User实
25 体类中
26  *      可以给集合属性赋值
27  */
28 @Data
29 @Component
30 @ConfigurationProperties(prefix = "ssh.user")
31 public class User {

```

```

27
28 //    @Value("${ssh.user.username}")
29     private String username;
30
31 //    @Value("${ssh.user.password}")
32     private String password;
33
34 //    @Value("${ssh.user.gfs}")
35     private List<String> gfs;
36 }

```

Controller 层:

```

1  package com.ssh.controller;
2
3  import com.ssh.pojo.User;
4  import
5  org.springframework.beans.factory.annotation.Autowired
6  ;
7  import
8  org.springframework.web.bind.annotation.GetMapping;
9  import
10 org.springframework.web.bind.annotation.RequestMapping
11 ;
12 import
13 org.springframework.web.bind.annotation.RestController
14 ;
15
16 /**
17  * @author 申书航
18  * @version 1.0
19  */
20 @RestController
21 @RequestMapping("/user")
22 public class UserController {
23
24     @Autowired
25     private User user;
26
27
28     @GetMapping("/show")
29     public User show() {

```

```
22         return user;
23     }
24 }
```

测试配置文件中的访问地址即可。

## (4) 多环境配置与使用

### 1. 多环境配置介绍

在Spring Boot中，可以使用多环境配置来根据不同的运行环境（如开发、测试、生产）加载不同的配置。SpringBoot支持多环境配置让应用程序在不同的环境中使用不同的配置参数，例如数据库连接信息、日志级别、缓存配置等。

以下是实现Spring Boot多环境配置的常见方法：

1. 属性文件分离：将应用程序的配置参数分离到不同的属性文件中，每个环境对应一个属性文件。例如，可以创建 `application-dev.properties`、`application-prod.properties` 和 `application-test.properties` 等文件。在这些文件中，可以定义各自环境的配置参数，如数据库连接信息、端口号等。然后，在 `application.properties` 中通过 `spring.profiles.active` 属性指定当前使用的环境。Spring Boot会根据该属性来加载对应环境的属性文件，覆盖默认的配置。
2. YAML配置文件：与属性文件类似，可以将配置参数分离到不同的YAML文件中，每个环境对应一个文件。例如，可以创建 `application-dev.yml`、`application-prod.yml` 和 `application-test.yml` 等文件。在这些文件中，可以使用YAML语法定义各自环境的配置参数。同样，通过 `spring.profiles.active` 属性指定当前的环境，Spring Boot会加载相应的YAML文件。
3. 命令行参数(动态)：可以通过命令行参数来指定当前的环境。例如，可以使用 `--spring.profiles.active=dev` 来指定使用开发环境的配置。

通过上述方法，Spring Boot会根据当前指定的环境来加载相应的配置文件或参数，从而实现多环境配置。这样可以简化在不同环境之间的配置切换，并且确保应用程序在不同环境中具有正确的配置。

### 2. 多环境配置使用

基于上面的方法2来实现：

如果设置了 `spring.profiles.active`，并且和 `application.yaml` 文件有重叠属性，以 `active` 设置优先。

`application.yaml`：

```
1  # yaml 文件有层次，可以继承配置文件的格式
2
3  server:
4    port: 808
5    servlet:
6      context-path: /boot
7
8  # 自定义配置
9  ssh:
10   info:
11     age: 20
12     name: "申书航"
13   user:
14     # 集合类型配置
15     gfs:
16       - a
17       - b
18       - c
19     username: nameApplication
20
21  # 激活外部配置文件，如果外部配置文件的key和本配置文件的key相同，
    则会覆盖本配置文件的key值
22  spring:
23    profiles:
24      active: test, dev # 激活了 application-test 和
    application-dev 两个配置文件
```

`application-dev.yaml`：

```
1  ssh:
2    user:
3      username: rootdev
```

`application-test.yaml`：



```
1 ssh:
2   user:
3     password: 123456test
```

启动类，实体类与 Controller 层均与上面相同，此处省略，测试即可。

## 三、SpringBoot 整合

### (1) SpringMVC 整合

#### 1. 准备过程

准备项目：springboot-base-springmvc-3

导入依赖：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-
4         instance"
5         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
6         http://maven.apache.org/xsd/maven-4.0.0.xsd">
7     <modelVersion>4.0.0</modelVersion>
8     <parent>
9         <groupId>org.springframework.boot</groupId>
10        <artifactId>spring-boot-starter-
11        parent</artifactId>
12        <version>3.0.5</version>
13    </parent>
14    <groupId>com.ssh</groupId>
15    <artifactId>springboot-base-springmvc-
16    3</artifactId>
17    <version>1.0-SNAPSHOT</version>
18    <dependencies>
19        <dependency>
20            <groupId>org.springframework.boot</groupId>
```

```
21         <artifactId>spring-boot-starter-  
web</artifactId>  
22         </dependency>  
23     </dependencies>  
24  
25 </project>
```

启动类:

```
1 package com.ssh;  
2  
3 import org.springframework.boot.SpringApplication;  
4 import  
org.springframework.boot.autoconfigure.SpringBootApplication;  
5  
6 /**  
7  * @author 申书航  
8  * @version 1.0  
9  * 启动类  
10 */  
11 @SpringBootApplication  
12 public class Main {  
13  
14     public static void main(String[] args) {  
15         SpringApplication.run(Main.class, args);  
16     }  
17 }
```

Controller 层:

```
1 package com.ssh.controller;  
2  
3 import  
org.springframework.web.bind.annotation.GetMapping;  
4 import  
org.springframework.web.bind.annotation.RequestMapping  
;  
5 import  
org.springframework.web.bind.annotation.RestController  
;
```

```

6
7  /**
8   * @author 申书航
9   * @version 1.0
10  */
11  @RestController
12  public class HelloController {
13
14      @GetMapping("hello")
15      public String ret() {
16          return "Hello World";
17      }
18  }

```

## 2. WEB 相关配置

application.yaml:

```

1  # springmvc 相关的WEB配置
2
3  server:
4      port: 80  # 端口号默认是8080，这里修改为80
5      servlet:
6          context-path: /boot  # 设置项目根路径

```

当涉及Spring Boot的Web应用程序配置时，以下是五个重要的配置参数：

1. `server.port`: 指定应用程序的HTTP服务器端口号。默认情况下，Spring Boot使用8080作为默认端口。您可以通过在配置文件中设置 `server.port` 来更改端口号。
2. `server.servlet.context-path`: 设置应用程序的上下文路径。这是应用程序在URL中的基本路径。默认情况下，上下文路径为空。您可以通过在配置文件中设置 `server.servlet.context-path` 属性来指定自定义的上下文路径。
3. `spring.mvc.view.prefix` 和 `spring.mvc.view.suffix`: 这两个属性用于配置视图解析器的前缀和后缀。视图解析器用于解析控制器返回的视图名称，并将其映射到实际的视图页面。  
`spring.mvc.view.prefix` 定义视图的前缀，  
`spring.mvc.view.suffix` 定义视图的后缀。

4. `spring.resources.static-locations`: 配置静态资源的位置。静态资源可以是CSS、JavaScript、图像等。默认情况下, Spring Boot会将静态资源放在 `classpath:/static` 目录下。您可以通过在配置文件中设置 `spring.resources.static-locations` 属性来自定义静态资源的位置。
5. `spring.http.encoding.charset` 和 `spring.http.encoding.enabled`: 这两个属性用于配置HTTP请求和响应的字符编码。 `spring.http.encoding.charset` 定义字符编码的名称 (例如UTF-8) , `spring.http.encoding.enabled` 用于启用或禁用字符编码的自动配置。

这些是在Spring Boot的配置文件中与Web应用程序相关的一些重要配置参数。根据您的需求, 您可以在配置文件中设置这些参数来定制和配置您的Web应用程序。

### 3. 静态资源处理

在WEB开发中我们需要引入一些静态资源, 例如: HTML, CSS, JS, 图片等, 如果是普通的项目静态资源可以放在项目的webapp目录下。现在使用Spring Boot做开发, 项目中没有 `webapp` 目录, 我们的项目是一个jar工程, 那么就没有 `webapp`, 我们的静态资源该放哪里呢?

**默认的静态资源路径为:**

- `classpath:/META-INF/resources/`;
- `classpath:/resources/`;
- `classpath:/static/`;
- `classpath:/public/`。

我们只要静态资源放在这些目录中任何一个, SpringMVC 都会帮我们处理。我们习惯会把静态资源放在 `classpath:/static/` 目录下。在 `resources` 目录下创建 `index.html` 文件。

测试 <http://localhost:80/boot/index.html>;

覆盖路径:

```

1  # springmvc 相关的WEB配置
2
3  server:
4      port: 80  # 端口号默认是8080，这里修改为80
5      servlet:
6          context-path: /boot  # 设置项目根路径
7
8  spring:
9      web:
10         resources:
11             static-locations: classpath:/webapp # 设置静态资源
12                                     文件夹，配置以后默认的静态资源路径就无法访问了
13                                     # 外部在访问静态资源时，不需要写静态资源文件夹

```

在 `webapp` 目录下创建 `register.html` 文件，测试

<http://localhost:80/boot/register.html>

#### 4. 拦截器配置

正常使用配置类，只要保证配置类要在启动类的同包或者子包方可生效。

拦截器定义：

```

1  package com.ssh.interceptor;
2
3  import jakarta.servlet.http.HttpServletRequest;
4  import jakarta.servlet.http.HttpServletResponse;
5  import
6      org.springframework.web.servlet.HandlerInterceptor;
7
8  /**
9   * @author 申书航
10  * @version 1.0
11  * 拦截器
12  */
13  public class MyInterceptor implements
14      HandlerInterceptor {
15
16      @Override
17      public boolean preHandle(HttpServletRequest
18          request, HttpServletResponse response, Object handler)
19          throws Exception {

```

```

16         System.out.println("request = " + request + ",
    handler = " + handler + ", response = " + response);
17         return true;
18     }
19 }

```

拦截器配置:

```

1  package com.ssh.config;
2
3  import com.ssh.interceptor.MyInterceptor;
4  import
    org.springframework.context.annotation.Configuration;
5  import
    org.springframework.web.servlet.config.annotation.Inte
    rceptorRegistry;
6  import
    org.springframework.web.servlet.config.annotation.WebM
    vcConfigurer;
7
8  /**
9   * @author 申书航
10  * @version 1.0
11  * 配置类
12  */
13  @Configuration
14  public class WebMvcConfig implements WebMvcConfigurer
    {
15
16      @Override
17      public void addInterceptors(InterceptorRegistry
    registry) {
18          registry.addInterceptor(new MyInterceptor());
19      }
20  }

```

## (2) Druid 数据源整合

项目准备: `springboot-base-druid-4`

依赖导入: Druid连接池要用1.2.20以上版本才能与 SpringBoot3兼容。

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
4
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <parent>
8         <groupId>org.springframework.boot</groupId>
9         <artifactId>spring-boot-starter-
parent</artifactId>
10        <version>3.0.5</version>
11    </parent>
12
13    <groupId>com.ssh</groupId>
14    <artifactId>springboot-base-druid-4</artifactId>
15    <version>1.0-SNAPSHOT</version>
16
17    <dependencies>
18        <!-- web开发的场景启动器 -->
19        <dependency>
20
21            <groupId>org.springframework.boot</groupId>
22            <artifactId>spring-boot-starter-
web</artifactId>
23            </dependency>
24
25            <!-- 数据库相关配置启动器 jdbc-template 事务相关-->
26            <dependency>
27
28                <groupId>org.springframework.boot</groupId>
29                <artifactId>spring-boot-starter-
jdbc</artifactId>
30                </dependency>
31
32                <!-- druid启动器的依赖 -->
33                <dependency>
34
35                    <groupId>com.alibaba</groupId>
```

```

33         <artifactId>druid-spring-boot-3-
starter</artifactId>
34         <version>1.2.20</version>
35     </dependency>
36
37     <!-- 驱动类-->
38     <dependency>
39         <groupId>mysql</groupId>
40         <artifactId>mysql-connector-
java</artifactId>
41         <version>8.0.28</version>
42     </dependency>
43
44     <dependency>
45         <groupId>org.projectlombok</groupId>
46         <artifactId>lombok</artifactId>
47         <version>1.18.28</version>
48     </dependency>
49 </dependencies>
50
51 </project>

```

启动类:

```

1  package com.ssh;
2
3  import org.springframework.boot.SpringApplication;
4  import
    org.springframework.boot.autoconfigure.SpringBootApplication;
5
6  /**
7   * @author 申书航
8   * @version 1.0
9   * 启动类
10  */
11  @SpringBootApplication
12  public class Main {
13
14      public static void main(String[] args) {
15          SpringApplication.run(Main.class, args);

```



```
16     }
17 }
```

实体类:

```
1 package com.ssh.pojo;
2
3 import lombok.Data;
4
5 /**
6  * @author 申书航
7  * @version 1.0
8  * 员工实体类
9  */
10 @Data
11 public class Employee {
12
13     private Integer empId;
14
15     private String empName;
16
17     private Double empSalary;
18 }
```

配置文件:

```
1 spring:
2     datasource:
3         type: com.alibaba.druid.pool.DruidDataSource # 使
4         # springboot2支持直接配置用户名和密码
5         # username: root
6         # password: root
7     druid:
8         url: jdbc:mysql://localhost:3306/lesson
9         username: root
10        password: root
11        driver-class-name: com.mysql.cj.jdbc.Driver
12        # 初始化时建立物理连接的个数
13        initial-size: 5
14        # 连接池的最小空闲数量
```

```

15     min-idle: 5
16     # 连接池最大连接数量
17     max-active: 20
18     # 获取连接时最大等待时间, 单位毫秒
19     max-wait: 60000
20     # 申请连接的时候检测, 如果空闲时间大于
    timeBetweenEvictionRunsMillis, 执行validationQuery检测连
    接是否有效。
21     test-while-idle: true
22     # 既作为检测的间隔时间又作为testWhileIdle执行的依据
23     time-between-eviction-runs-millis: 60000
24     # 销毁线程时检测当前连接的最后活动时间和当前时间差大于该值
    时, 关闭当前连接(配置连接在池中的最小生存时间)
25     min-evictable-idle-time-millis: 30000
26     # 用来检测数据库连接是否有效的sql 必须是一个查询语句
    (oracle中为 select 1 from dual)
27     validation-query: select 1
28     # 申请连接时会执行validationQuery检测连接是否有效, 开启
    会降低性能, 默认为true
29     test-on-borrow: false
30     # 归还连接时会执行validationQuery检测连接是否有效, 开启
    会降低性能, 默认为true
31     test-on-return: false
32     # 是否缓存preparedStatement, 也就是PSCache, PSCache
    对支持游标的数据库性能提升巨大, 比如说oracle, 在mysql下建议关
    闭。
33     pool-prepared-statements: false
34     # 要启用PSCache, 必须配置大于0, 当大于0时,
    poolPreparedStatements自动触发修改为true。在Druid中, 不会存
    在Oracle下PSCache占用内存过多的问题, 可以把这个数值配置大一些,
    比如说100
35     max-pool-prepared-statement-per-connection-size:
    -1
36     # 合并多个DruidDataSource的监控数据
37     use-global-data-source-stat: true

```

Controller 层:

```

1 package com.ssh.controller;
2
3 import com.ssh.pojo.Employee;

```

```
4  import
    org.springframework.beans.factory.annotation.Autowired
    ;
5  import
    org.springframework.jdbc.core.BeanPropertyRowMapper;
6  import org.springframework.jdbc.core.JdbcTemplate;
7  import
    org.springframework.web.bind.annotation.GetMapping;
8  import
    org.springframework.web.bind.annotation.RequestMapping
    ;
9  import
    org.springframework.web.bind.annotation.RestController
    ;
10
11  import java.util.List;
12
13  /**
14   * @author 申书航
15   * @version 1.0
16   */
17  @RestController
18  @RequestMapping("/employee")
19  public class EmployeeController {
20
21      @Autowired
22      private JdbcTemplate jdbcTemplate;
23
24      @GetMapping("/list")
25      public List<Employee> list() {
26          //数据库
27          String sql = "select * from t_emp";
28
29          List<Employee> employeeList =
            jdbcTemplate.query(sql, new BeanPropertyRowMapper<
              (Employee.class));
30          return employeeList;
31
32      }
33  }
```

测试启动即可。

如果 `druid-spring-boot-3-starter` 版本是1.2.18或以下，虽然适配了SpringBoot3，但缺少自动装配的配置文件，需要手动在 `resources` 目录下创建文件夹 `META-INF/spring`，在该文件夹下创建文件

`org.springframework.boot.autoconfigure.AutoConfiguration.imports`，文件内容如下：

```
1 | com.alibaba.druid.spring.boot3.autoconfigure.DruidDataSourceAutoConfigure
```

## (3) MyBatis 与 Spring 整合

### 1. MyBatis 整合

整合MyBatis 需要在启动类添加 `@MapperScan("包名")` 注解，来指定Mapper 接口的扫描路径。

项目创建：`springboot-base-mybatis-5`；

依赖导入：

```
1 | <?xml version="1.0" encoding="UTF-8"?>
2 | <project xmlns="http://maven.apache.org/POM/4.0.0"
3 |           xmlns:xsi="http://www.w3.org/2001/XMLSchema-
   | instance"
4 |
   | xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   | http://maven.apache.org/xsd/maven-4.0.0.xsd">
5 |     <modelVersion>4.0.0</modelVersion>
6 |
7 |     <parent>
8 |         <groupId>org.springframework.boot</groupId>
9 |         <artifactId>spring-boot-starter-
   | parent</artifactId>
10 |         <version>3.0.5</version>
11 |     </parent>
12 |
13 |     <groupId>com.ssh</groupId>
14 |     <artifactId>springboot-base-mybatis-5</artifactId>
15 |     <version>1.0-SNAPSHOT</version>
16 |
```

```
17     <dependencies>
18         <dependency>
19
20             <groupId>org.springframework.boot</groupId>
21             <artifactId>spring-boot-starter-
web</artifactId>
22             </dependency>
23
24             <dependency>
25                 <groupId>org.mybatis.spring.boot</groupId>
26                 <artifactId>mybatis-spring-boot-
starter</artifactId>
27                 <version>3.0.1</version>
28             </dependency>
29
30             <!-- 数据库相关配置启动器 -->
31             <dependency>
32
33                 <groupId>org.springframework.boot</groupId>
34                 <artifactId>spring-boot-starter-
jdbc</artifactId>
35             </dependency>
36
37             <!-- druid启动器的依赖 -->
38             <dependency>
39                 <groupId>com.alibaba</groupId>
40                 <artifactId>druid-spring-boot-3-
starter</artifactId>
41                 <version>1.2.20</version>
42             </dependency>
43
44             <!-- 驱动类-->
45             <dependency>
46                 <groupId>mysql</groupId>
47                 <artifactId>mysql-connector-
java</artifactId>
48                 <version>8.0.28</version>
49             </dependency>
50
51             <dependency>
52                 <groupId>org.projectlombok</groupId>
```

```

51         <artifactId>lombok</artifactId>
52         <version>1.18.28</version>
53     </dependency>
54
55 </dependencies>
56 </project>

```

启动类:

```

1  package com.ssh;
2
3  import org.mybatis.spring.annotation.MapperScan;
4  import org.springframework.boot.SpringApplication;
5  import
    org.springframework.boot.autoconfigure.SpringBootApplication;
6
7  /**
8   * @author 申书航
9   * @version 1.0
10  * 启动类
11  */
12 @SpringBootApplication
13 @MapperScan("com.ssh.mapper") //指定Mapper接口的扫描路
    径
14 public class Main {
15
16     public static void main(String[] args) {
17         SpringApplication.run(Main.class, args);
18     }
19 }

```

配置文件:

```

1  # Druid连接池
2
3  spring:
4      datasource:
5          type: com.alibaba.druid.pool.DruidDataSource
6          druid:
7              url: jdbc:mysql://localhost:3306/lesson

```

```
8      username: root
9      password: root
10     driver-class-name: com.mysql.cj.jdbc.Driver
11     # 初始化时建立物理连接的个数
12     initial-size: 5
13     # 连接池的最小空闲数量
14     min-idle: 5
15     # 连接池最大连接数量
16     max-active: 20
17     # 获取连接时最大等待时间，单位毫秒
18     max-wait: 60000
19     # 申请连接的时候检测，如果空闲时间大于
timeBetweenEvictionRunsMillis，执行validationQuery检测连
接是否有效。
20     test-while-idle: true
21     # 既作为检测的间隔时间又作为testWhileIdle执行的依据
22     time-between-eviction-runs-millis: 60000
23     # 销毁线程时检测当前连接的最后活动时间和当前时间差大于该值
时，关闭当前连接(配置连接在池中的最小生存时间)
24     min-evictable-idle-time-millis: 30000
25     # 用来检测数据库连接是否有效的sql 必须是一个查询语句
(oracle中为 select 1 from dual)
26     validation-query: select 1
27     # 申请连接时会执行validationQuery检测连接是否有效,开启
会降低性能,默认为true
28     test-on-borrow: false
29     # 归还连接时会执行validationQuery检测连接是否有效,开启
会降低性能,默认为true
30     test-on-return: false
31     # 是否缓存preparedStatement，也就是PSCache,PSCache
对支持游标的数据库性能提升巨大，比如说oracle,在mysql下建议关
闭。
32     pool-prepared-statements: false
33     # 要启用PSCache，必须配置大于0，当大于0时，
poolPreparedStatements自动触发修改为true。在Druid中，不会存
在Oracle下PSCache占用内存过多的问题，可以把这个数值配置大一些，
比如说100
34     max-pool-prepared-statement-per-connection-size:
-1
35     # 合并多个DruidDataSource的监控数据
36     use-global-data-source-stat: true
```

```

37
38 #MyBatis配置 彻底抛弃了mybatis-config.xml配置文件，全部配置
   在application.yaml中
39 # mapper.xml, settings, 别名等
40
41 mybatis:
42     mapper-locations: classpath:/mappers/* # mapper.xml
   文件路径
43     type-aliases-package: com.ssh.pojo # 别名包路径
44     configuration:
45         map-underscore-to-camel-case: true # 驼峰命名规则
46         auto-mapping-behavior: full # 自动映射
47         log-impl:
   org.apache.ibatis.logging.slf4j.Slf4jImpl # 日志实现

```

Mapper 接口:

```

1  package com.ssh.mapper;
2
3  import com.ssh.pojo.Employee;
4
5  import java.util.List;
6
7  /**
8   * @author 申书航
9   * @version 1.0
10  */
11  public interface EmployeeMapper {
12
13      List<Employee> queryAll();
14  }

```

EmployeeMapper.xml:



```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "https://mybatis.org/dtd/mybatis-3-
mapper.dtd">
5
6 <mapper namespace="com.ssh.mapper.EmployeeMapper">
7
8     <select id="queryAll" resultType="employee">
9         select * from t_emp;
10    </select>
11
12 </mapper>

```

Controller 层:

```

1 package com.ssh.controller;
2
3 import com.ssh.mapper.EmployeeMapper;
4 import com.ssh.pojo.Employee;
5 import
org.springframework.beans.factory.annotation.Autowired
;
6 import
org.springframework.web.bind.annotation.GetMapping;
7 import
org.springframework.web.bind.annotation.RequestMapping
;
8 import
org.springframework.web.bind.annotation.RestController
;
9
10 import java.util.List;
11
12 /**
13  * @author 申书航
14  * @version 1.0
15  */
16 @RestController
17 @RequestMapping("/employee")
18 public class EmployeeController {

```

```

19
20     @Autowired
21     private EmployeeMapper mapper;
22
23     @GetMapping
24     public List<Employee> query() {
25         return mapper.queryAll();
26     }
27 }

```

测试启动即可。

## 2. 声明式事务整合

声明式事务需要导入的依赖是JDBC的依赖，在前面用到过：

```

1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-jdbc</artifactId>
4 </dependency>

```

直接在业务方法上添加 `@Transactional` 注解即可自动添加声明式事务。

Mapper 接口：

```

1 package com.ssh.mapper;
2
3 import com.ssh.pojo.Employee;
4
5 import java.util.List;
6
7 /**
8  * @author 申书航
9  * @version 1.0
10 */
11 public interface EmployeeMapper {
12
13     List<Employee> queryAll();
14
15     int delete(int empId);
16 }

```

EmployeeMapper.xml:

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "https://mybatis.org/dtd/mybatis-3-
5     mapper.dtd">
6 <mapper namespace="com.ssh.mapper.EmployeeMapper">
7
8     <select id="queryAll" resultType="employee">
9         select * from t_emp;
10    </select>
11
12    <delete id="delete">
13        delete from t_emp where emp_id = #{empId};
14    </delete>
15
16 </mapper>
```

业务 Service 层:

```
1 package com.ssh.service;
2
3 import com.ssh.mapper.EmployeeMapper;
4 import
5     org.springframework.beans.factory.annotation.Autowired
6     ;
7 import org.springframework.stereotype.Service;
8 import
9     org.springframework.transaction.annotation.Transactional;
10
11 /**
12  * @author 申书航
13  * @version 1.0
14  */
15 @Service
16 public class EmployeeService {
17
18     @Autowired
```

```

16     private EmployeeMapper employeeMapper;
17
18     @Transactional
19     public void delete() {
20         int rows = employeeMapper.delete(4);
21         System.out.println(rows);
22         int i = 1 / 0;    //模拟错误，事务回滚
23     }
24 }

```

Controller 层:

```

1  package com.ssh.controller;
2
3  import com.ssh.mapper.EmployeeMapper;
4  import com.ssh.pojo.Employee;
5  import com.ssh.service.EmployeeService;
6  import
    org.springframework.beans.factory.annotation.Autowired
    ;
7  import
    org.springframework.web.bind.annotation.GetMapping;
8  import
    org.springframework.web.bind.annotation.RequestMapping
    ;
9  import
    org.springframework.web.bind.annotation.RestController
    ;
10
11  import java.util.List;
12
13  /**
14   * @author 申书航
15   * @version 1.0
16   */
17  @RestController
18  @RequestMapping("/employee")
19  public class EmployeeController {
20
21      @Autowired
22      private EmployeeMapper mapper;

```

```

23
24     @Autowired
25     private EmployeeService employeeService;
26
27     @GetMapping
28     public List<Employee> query() {
29         employeeService.delete();
30         return mapper.queryAll();
31     }
32 }

```

### 3. Spring AOP 整合

导入依赖：

```

1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-aop</artifactId>
4 </dependency>

```

日志AOP：

```

1 package com.ssh.advice;
2
3 import org.aspectj.lang.JoinPoint;
4 import org.aspectj.lang.annotation.Aspect;
5 import org.aspectj.lang.annotation.Before;
6 import org.springframework.core.annotation.Order;
7 import org.springframework.stereotype.Component;
8
9 /**
10  * @author 申书航
11  * @version 1.0
12  */
13 @Component
14 @Aspect
15 @Order(5)
16 public class LogAdvice {
17
18     @Before("execution(* com.ssh.service.*.*(..))")
19     public void before(JoinPoint point) {

```

```

20         String simpleName =
point.getTarget().getClass().getSimpleName();
21         String className =
point.getTarget().getClass().getName();
22         String methodName =
point.getSignature().getName();
23         System.out.println("before " + simpleName +
"." + methodName + "() in " + className);
24     }
25 }

```

Service 层:

```

1  package com.ssh.service;
2
3  import com.ssh.mapper.EmployeeMapper;
4  import
org.springframework.beans.factory.annotation.Autowired
;
5  import org.springframework.stereotype.Service;
6  import
org.springframework.transaction.annotation.Transactional;
7
8  /**
9   * @author 申书航
10  * @version 1.0
11  */
12  @Service
13  public class EmployeeService {
14
15      @Autowired
16      private EmployeeMapper employeeMapper;
17
18      @Transactional
19      public void delete() {
20          int rows = employeeMapper.delete(4);
21          System.out.println(rows);
22          //      int i = 1 / 0;
23      }
24  }

```

## Controller 层:

```
1  package com.ssh.controller;
2
3  import com.ssh.mapper.EmployeeMapper;
4  import com.ssh.pojo.Employee;
5  import com.ssh.service.EmployeeService;
6  import
    org.springframework.beans.factory.annotation.Autowired
    ;
7  import
    org.springframework.web.bind.annotation.GetMapping;
8  import
    org.springframework.web.bind.annotation.RequestMapping
    ;
9  import
    org.springframework.web.bind.annotation.RestController
    ;
10
11 import java.util.List;
12
13 /**
14  * @author 申书航
15  * @version 1.0
16  */
17 @RestController
18 @RequestMapping("/employee")
19 public class EmployeeController {
20
21     @Autowired
22     private EmployeeMapper mapper;
23
24     @Autowired
25     private EmployeeService employeeService;
26
27     @GetMapping
28     public List<Employee> query() {
29         employeeService.delete();
30         //         return mapper.queryAll();
31         return null;
32     }
```

启动测试即可。

## (4) SpringBoot 项目打包运行

添加打包插件：

在Spring Boot项目中添加 `spring-boot-maven-plugin` 插件是为了支持将项目打包成可执行的可运行jar包。如果不添加 `spring-boot-maven-plugin` 插件配置，使用常规的 `java -jar` 命令来运行打包后的Spring Boot项目是无法找到应用程序的入口点，因此导致无法运行。

```
1 <!--      SpringBoot应用打包插件-->
2 <build>
3     <plugins>
4         <plugin>
5             <groupId>org.springframework.boot</groupId>
6             <artifactId>spring-boot-maven-
plugin</artifactId>
7         </plugin>
8     </plugins>
9 </build>
```

执行打包：在 Maven 中选择本项目，然后选择生命周期，clear 以后再点击 package 进行打包。

命令启动：

`java -jar` 命令用于在Java环境中执行可执行的JAR文件。下面是关于 `java -jar` 命令的说明：

```
1 命令格式：java -jar [选项] [参数] <jar文件名>
```

1. `-D<name>=<value>`：设置系统属性，可以通过 `System.getProperty()` 方法在应用程序中获取该属性值。例如：  
`java -jar -Dserver.port=8080 myapp.jar`。
2. `-x`：设置JVM参数，例如内存大小、垃圾回收策略等。常用的选项包括：



- `-Xmx<size>`: 设置JVM的最大堆内存大小, 例如 `-Xmx512m` 表示设置最大堆内存为512MB。
  - `-Xms<size>`: 设置JVM的初始堆内存大小, 例如 `-Xms256m` 表示设置初始堆内存为256MB。
3. `-Dspring.profiles.active=<profile>`: 指定Spring Boot的激活配置文件, 可以通过 `application-<profile>.properties` 或 `application-<profile>.yml` 文件来加载相应的配置。例如: `java -jar -Dspring.profiles.active=dev myapp.jar`。

**注意:** `-D` 参数必须要在jar之前, 否则不生效。

**Spring 框架后续内容见: 《MyBatis-Plus》 .....**