

收藏：后期更新，方便查找  
转发：这么好的技术，朋友不会是不是要推荐一下  
评论：问问题，后期更新  
弹幕：互动问题

点赞  
评论  
收藏  
转发  
弹幕



黑马程序员  
[www.itheima.com](http://www.itheima.com)

传智教育旗下  
高端IT教育品牌

## 课程适用人群

- 小白：完全没有用过SpringBoot技术
- 初学者：能使用SpringBoot技术完成基础的SSM整合
- 开发者：能使用SpringBoot技术实现常见的技术整合工作

## 课程适用人群与收获

- 小白
  - ◆ 初步掌握SpringBoot程序的开发流程，能够基于SpringBoot实现基础SSM框架整合
- 初学者
  - ◆ 掌握各式各样的第三方技术与SpringBoot整合的方案
  - ◆ 积累基于SpringBoot的实战开发经验
- 开发者
  - ◆ 提升对Spring及SpringBoot原理的理解层次
  - ◆ 基于原理理解基础上，实现自主研发基于SpringBoot整合任意技术的开发方式

停

## SpringBoot课程模块

- 基础篇
- 实用篇
- 原理篇

## SpringBoot课程模块

- 基础篇
- 实用篇
  - ◆ 运维实用篇
  - ◆ 开发实用篇
- 原理篇

## SpringBoot课程学习目标

- 基础篇
  - ◆ 能够创建SpringBoot工程
  - ◆ 基于SpringBoot实现ssm整合
- 实用篇
  - ◆ 运维实用篇
  - ◆ 开发实用篇
- 原理篇

## SpringBoot课程学习目标

- 基础篇
- 实用篇
  - ◆ 运维实用篇
    - 能够掌握SpringBoot程序多环境开发
    - 能够基于Linux系统发布SpringBoot工程
    - 能够解决线上灵活配置SpringBoot工程的需求
  - ◆ 开发实用篇
    - 能够基于SpringBoot整合任意第三方技术
- 原理篇



## SpringBoot课程学习目标

- 基础篇
- 实用篇
  - ◆ 运维实用篇
  - ◆ 开发实用篇
- 原理篇
  - ◆ 掌握SpringBoot内部工作流程
  - ◆ 理解SpringBoot整合第三方技术的原理
  - ◆ 实现自定义开发整合第三方技术的组件

## SpringBoot课程学习前置知识

- 基础篇

- ◆ Java基础语法

- ◆ Spring与SpringMVC

- 知道Spring是用来管理bean，能够基于Restful实现页面请求交互功能

- ◆ Mybatis与Mybatis-Plus

- 基于Mybatis和MybatisPlus能够开发出包含基础CRUD功能的标准Dao模块

- ◆ 数据库MySQL

- 能够读懂基础CRUD功能的SQL语句

- ◆ 服务器

- 知道服务器与web工程的关系，熟悉web服务器的基础配置

- ◆ maven

- 知道maven的依赖关系，知道什么是依赖范围，依赖传递，排除依赖，可选依赖，继承

- ◆ web技术 (含vue, ElementUI)

- 知道vue如何发送ajax请求，如何获取响应数据，如何进行数据模型双向绑定

## SpringBoot课程学习前置知识

- 实用篇
  - ◆ Linux (CenterOS7)
    - 熟悉常用的Linux基础指令, 熟悉Linux系统目录结构
  - ◆ 实用开发技术
    - 缓存: Redis、MongoDB、.....
    - 消息中间件: RocketMq、RabbitMq、.....
    - .....

## SpringBoot课程学习前置知识

- 原理篇
  - ◆ Spring
    - 了解Spring加载bean的各种方式
    - 知道Spring容器底层工作原理，能够阅读简单的Spring底层源码

停



# Spring Boot

## 基础篇



黑马程序员  
[www.itheima.com](http://www.itheima.com)

传智教育旗下  
高端IT教育品牌



# 目录

Contents

- ◆ 快速上手SpringBoot
- ◆ SpringBoot基础配置
- ◆ 基于SpringBoot实现SSM整合



# 快速上手SpringBoot

- SpringBoot入门程序开发
- 浅谈入门程序工作原理



## SpringBoot入门程序开发

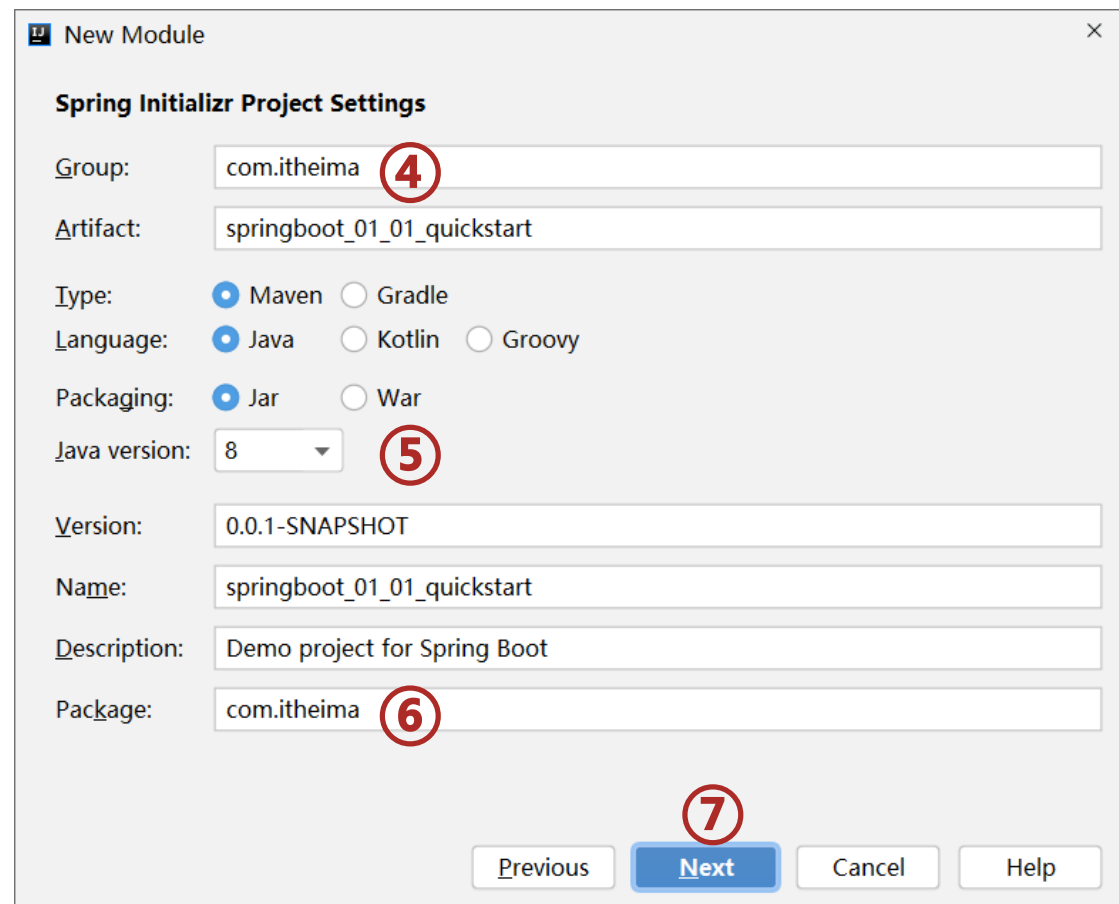
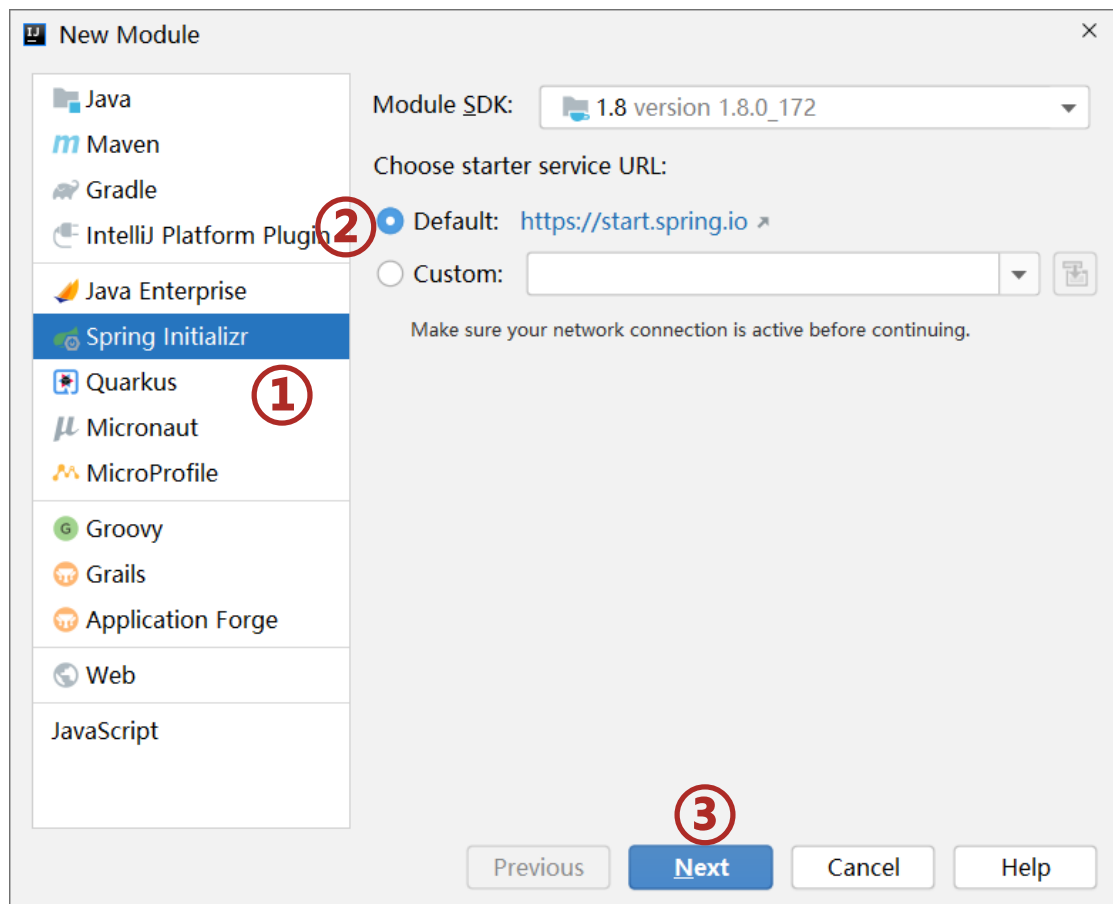
- SpringBoot是由Pivotal团队提供的全新框架，其设计目的是用来简化Spring应用的初始搭建以及开发过程

## SpringBoot入门程序开发

- SpringBoot是由Pivotal团队提供的全新框架，其设计目的是用来简化Spring应用的初始搭建以及开发过程

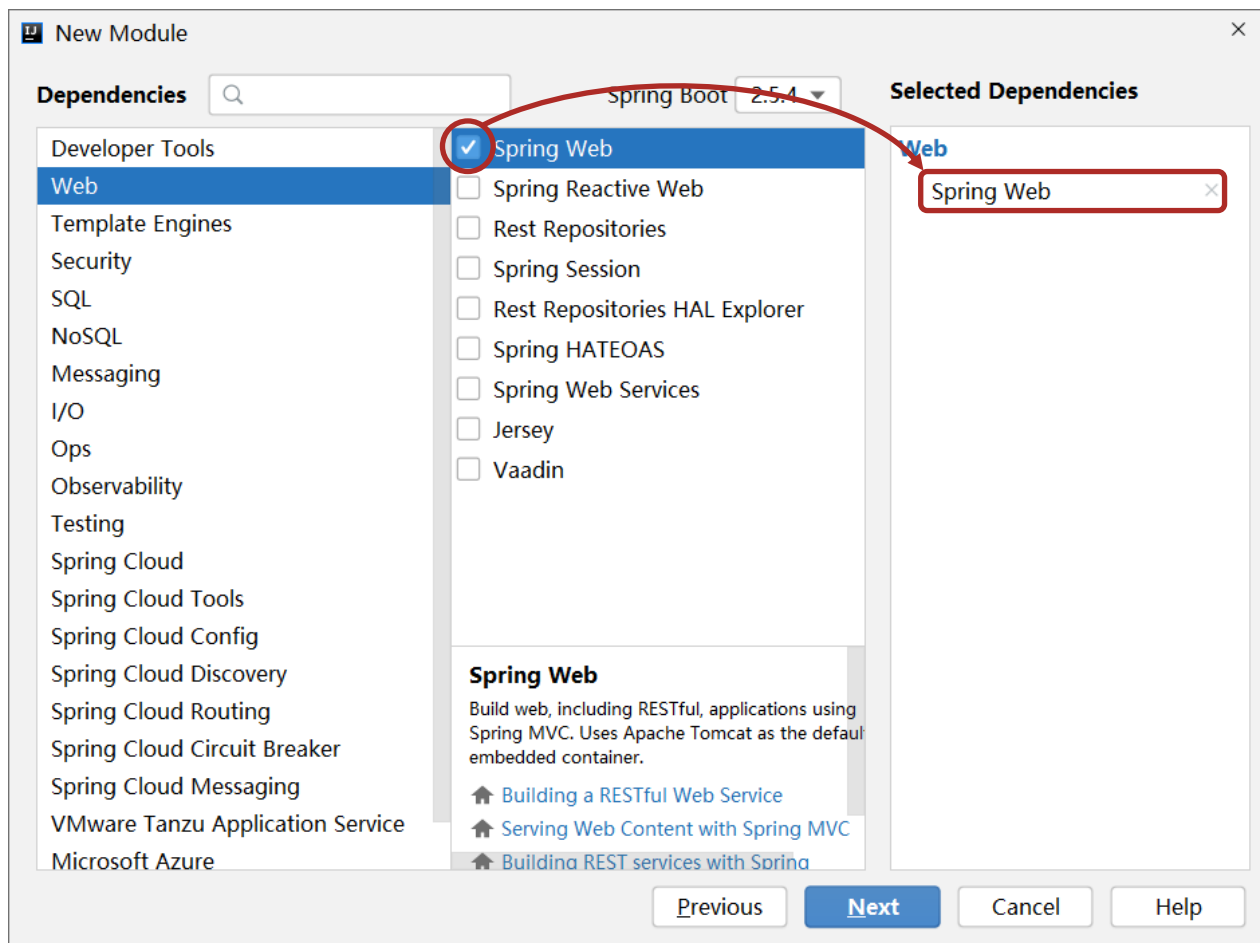
## 步骤 SpringBoot入门程序

①：创建新模块，选择Spring Initializr，并配置模块相关基础信息



## 步骤 SpringBoot入门程序

②：选择当前模块需要使用的技术集





## SpringBoot入门程序

### ③：开发控制器类

```
//Rest 模式
@RestController
@RequestMapping("/books")
public class BookController {
    @GetMapping
    public String getById(){
        System.out.println("springboot is running...");
        return "springboot is running...";
    }
}
```

步骤

#### ④：运行自动生成的Application类

```

\\ / _-_-_-_( )_ -_-_- \\ \\ \\ \\
( ( ) \_ | ' | ' | ' | ' V _ | \\ \\ \\ \\
\\ W _ ) | | ) | | | | | | ( | ) ) ) )
' | _ | . _ | | | | | \_ , / / / /
=====|_|=====|_/=/// //
:: Spring Boot ::

15:21:30.373 INFO 10848 --- [main] c.i.Springboot01QuickstartApplication : Starting Springboot01QuickstartApplication using Java 1.8.0_
15:21:30.381 INFO 10848 --- [main] c.i.Springboot01QuickstartApplication : No active profile set, falling back to default profiles: def
15:21:31.019 INFO 10848 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
15:21:31.024 INFO 10848 --- [main] o.apache.catalina.core.StandardService : Starting service Tomcat
15:21:31.025 INFO 10848 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.46]
15:21:31.066 INFO 10848 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
15:21:31.066 INFO 10848 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 654
15:21:31.277 INFO 10848 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
15:21:31.284 INFO 10848 --- [main] c.i.Springboot01QuickstartApplication : Started Springboot01QuickstartApplication in 1.168 seconds (
15:21:31.285 INFO 10848 --- [main] o.s.b.a.ApplicationAvailabilityBean : Application availability state LivenessState changed to CORR
15:21:31.286 INFO 10848 --- [main] o.s.b.a.ApplicationAvailabilityBean : Application availability state ReadinessState changed to ACC
15:21:36.407 INFO 10848 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
15:21:36.407 INFO 10848 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
15:21:36.408 INFO 10848 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms

```

## 入门案例

- 最简SpringBoot程序所包含的基础文件

- ◆ pom.xml文件

```
<?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="http://www.w3.org/2001/XMLSchema-instance"
    @SpringBootApplication
    public class Application {
        public static void main(String[] args) {
            SpringApplication.run(Application.class, args);
        }
    }
<version>0.0.1-SNAPSHOT</version>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
</project>
```

## 入门案例

- Spring程序与SpringBoot对比

| 类/配置文件             | Spring | Spring Boot |
|--------------------|--------|-------------|
| pom文件中的坐标          | 手工添加   | 勾选添加        |
| web3.0配置类          | 手工制作   | 无           |
| Spring/SpringMVC配置 | 手工制作   | 无           |
| 控制器                | 手工制作   | 手工制作        |

### 注意事项

基于idea开发SpringBoot程序，需要联网且能够加载到程序框架结构





## 小结

1. 开发SpringBoot程序可以根据向导进行联网快速制作
2. SpringBoot程序需要基于JDK8进行制作
3. SpringBoot程序中需要使用何种功能通过勾选选择技术
4. 运行SpringBoot程序通过运行Application程序入口进行

## 入门案例

- Spring程序与SpringBoot程序对比

| 类/配置文件              | Spring | SpringBoot |
|---------------------|--------|------------|
| pom文件中的坐标           | 手工添加   | 勾选添加       |
| web3.0配置类           | 手工制作   | 无          |
| Spring/SpringMVC配置类 | 手工制作   | 无          |
| 控制器                 | 手工制作   | 手工制作       |

痛点


注意事项

基于idea开发SpringBoot程序需要确保联网且能够加载到程序框架结构

停

## 入门案例

- 基于SpringBoot官网创建项目，地址：<https://start.spring.io/>

 **spring initializr**

**Project**  
☒ Maven Project ☐ Gradle Project

**Language**  
☒ Java ☐ Kotlin ☐ Groovy

**Spring Boot**  
☐ 2.6.0 (SNAPSHOT) ☐ 2.6.0 (M2) ☐ 2.5.5 (SNAPSHOT) ☒ 2.5.4  
☐ 2.4.11 (SNAPSHOT) ☐ 2.4.10

**Project Metadata**  
Group   
Artifact   
Name   
Description   
Package name   
Packaging ☒ Jar ☐ War  
Java ☐ 16 ☐ 11 ☒ 8

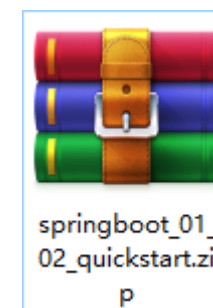
**Dependencies** ADD DEPENDENCIES... CTRL + B

**Spring Web** WEB  
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

GENERATE CTRL + G

EXPLORE CTRL + SPACE

SHARE...





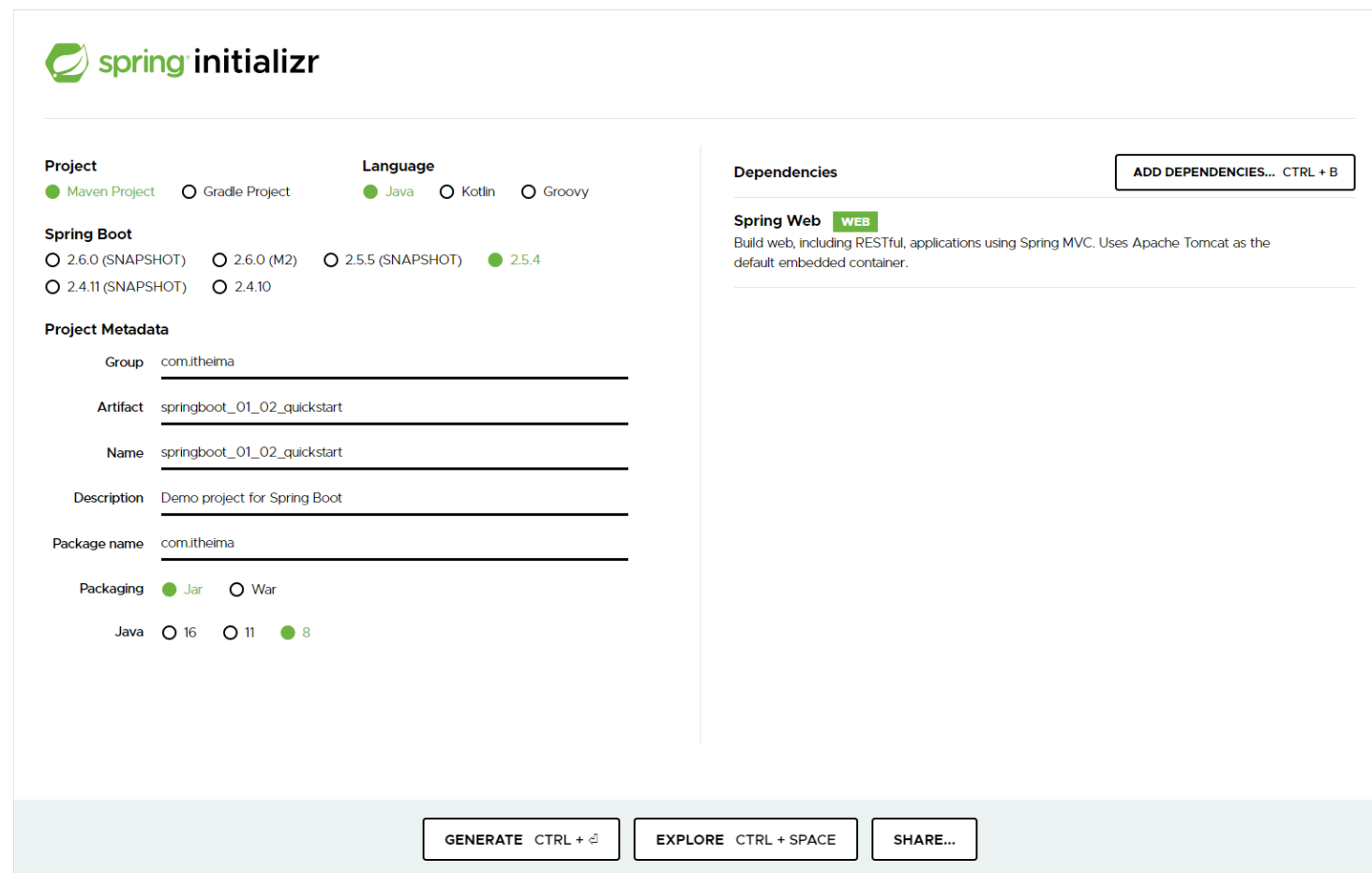
## 小结

1. 打开SpringBoot官网，选择Quickstart Your Project
2. 创建工程，并保存项目
3. 解压项目，通过IDE导入项目

## 入门案例

- 基于SpringBoot官网创建项目，地址：<https://start.spring.io/>

痛点

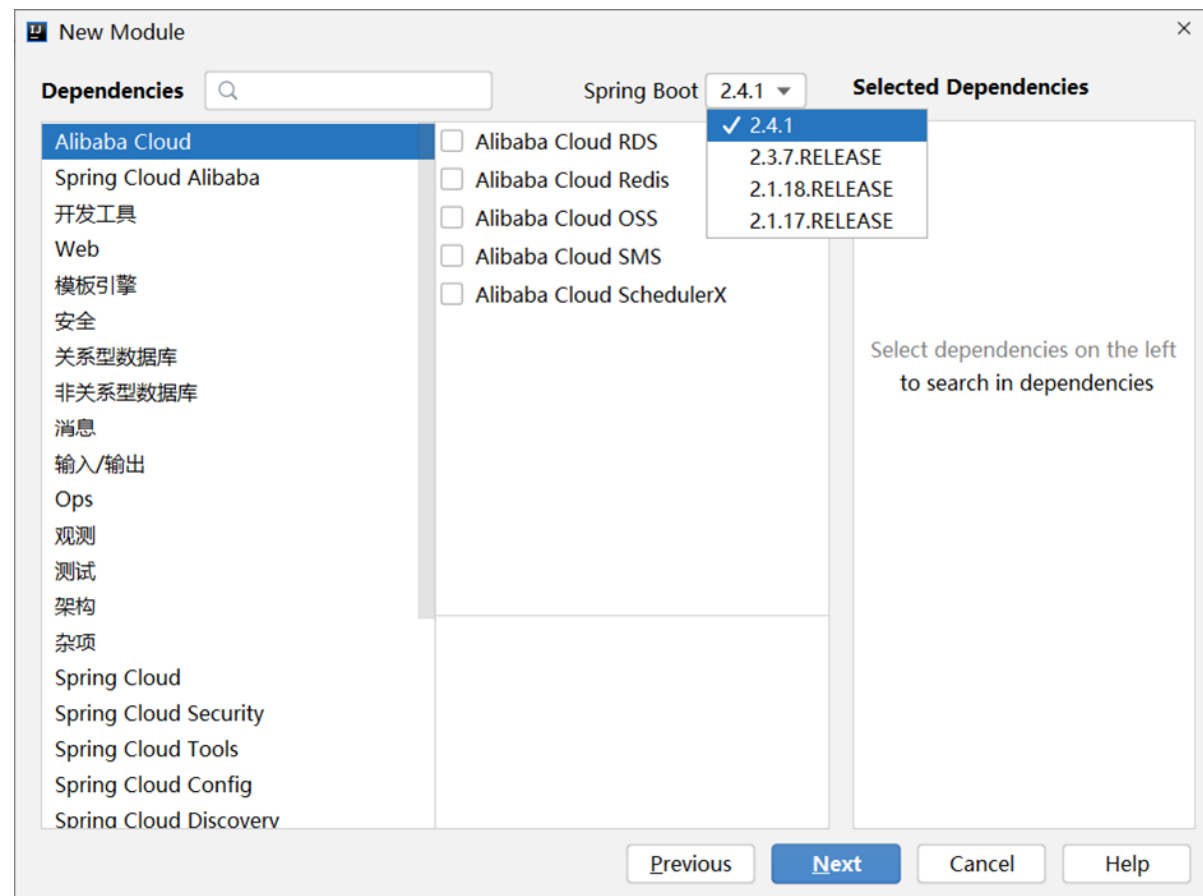
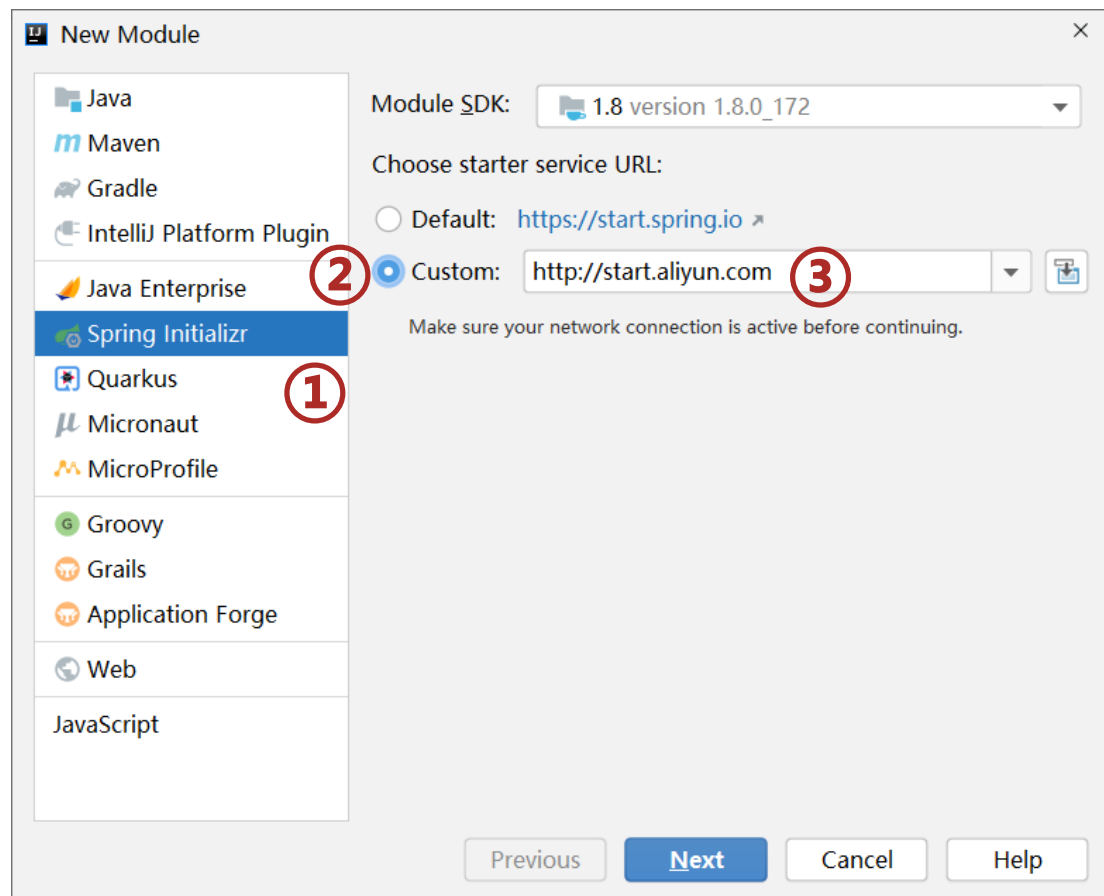


The screenshot displays the Spring Initializr web interface. It features a header with the 'spring initializr' logo. Below the header, there are three main sections: 'Project', 'Language', and 'Dependencies'. The 'Project' section includes radio buttons for 'Maven Project' (selected) and 'Gradle Project'. The 'Language' section includes radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'. The 'Spring Boot' section includes radio buttons for '2.6.0 (SNAPSHOT)', '2.6.0 (M2)', '2.5.5 (SNAPSHOT)', '2.5.4' (selected), and '2.4.11 (SNAPSHOT)'. The 'Project Metadata' section includes input fields for 'Group' (com.itheima), 'Artifact' (springboot\_01\_02\_quickstart), 'Name' (springboot\_01\_02\_quickstart), 'Description' (Demo project for Spring Boot), and 'Package name' (com.itheima). The 'Packaging' section includes radio buttons for 'Jar' (selected) and 'War'. The 'Dependencies' section includes a button 'ADD DEPENDENCIES... CTRL + B' and a 'Spring Web' dependency with a 'WEB' tag. At the bottom, there are three buttons: 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'.

停

## 入门案例

- 基于阿里云创建项目，地址：<https://start.aliyun.com>





## 入门案例

- 基于阿里云创建项目，地址：<https://start.aliyun.com>

### 注意事项

阿里云提供的坐标版本较低，如果需要使用高版本，进入工程后手工切换SpringBoot版本

阿里云提供的工程模板与Spring官网提供的工程模板略有不同



## 小结

1. 选择start来源为自定义URL
2. 输入阿里云start地址
3. 创建项目

## 入门案例

- 基于阿里云创建项目，地址：<https://start.aliyun.com>
- 基于官网创建项目，地址：<https://start.spring.io/>



痛点

停

## 入门案例

- 手工创建项目（手工导入坐标）

```
<?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>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.5.4</version>
  </parent>
  <groupId>com.itheima</groupId>
  <artifactId>springboot_01_03_quickstart</artifactId>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
  </dependencies>
</project>
```

## 入门案例

- 手工创建项目（手工制作引导类）

```
package com.itheima;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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



## 小结

1. 创建普通Maven工程
2. 继承spring-boot-starter-parent
3. 添加依赖spring-boot-starter-web
4. 制作引导类Application



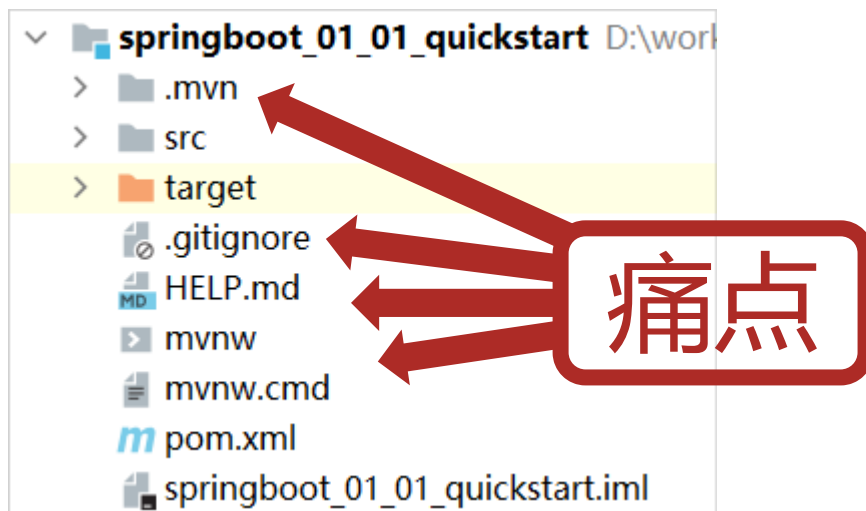
## 总结

### 1. 创建SpringBoot工程的四种方式

- 基于Idea创建SpringBoot工程
- 基于官网创建SpringBoot工程
- 基于阿里云创建SpringBoot工程
- 手工创建Maven工程修改为SpringBoot工程

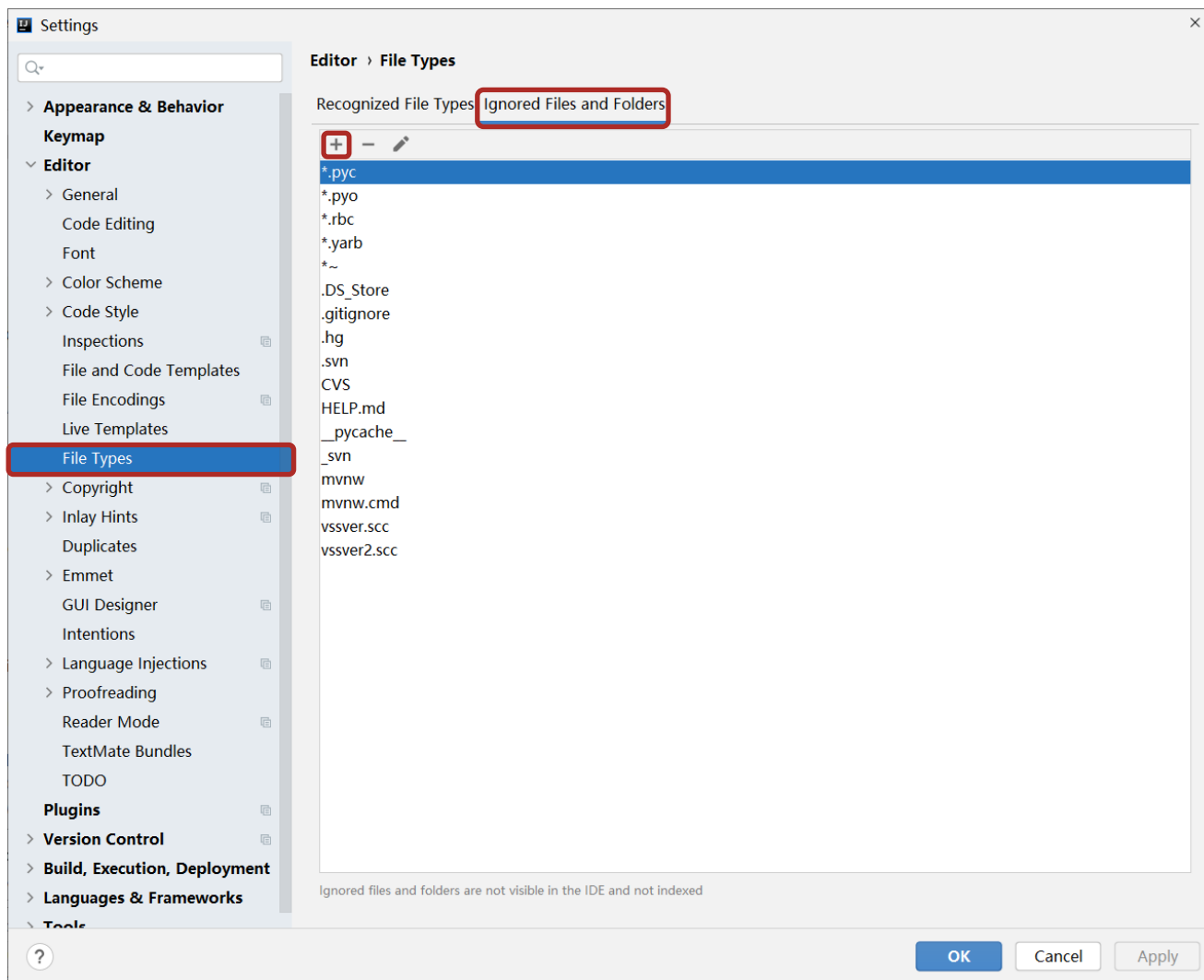


## 入门案例

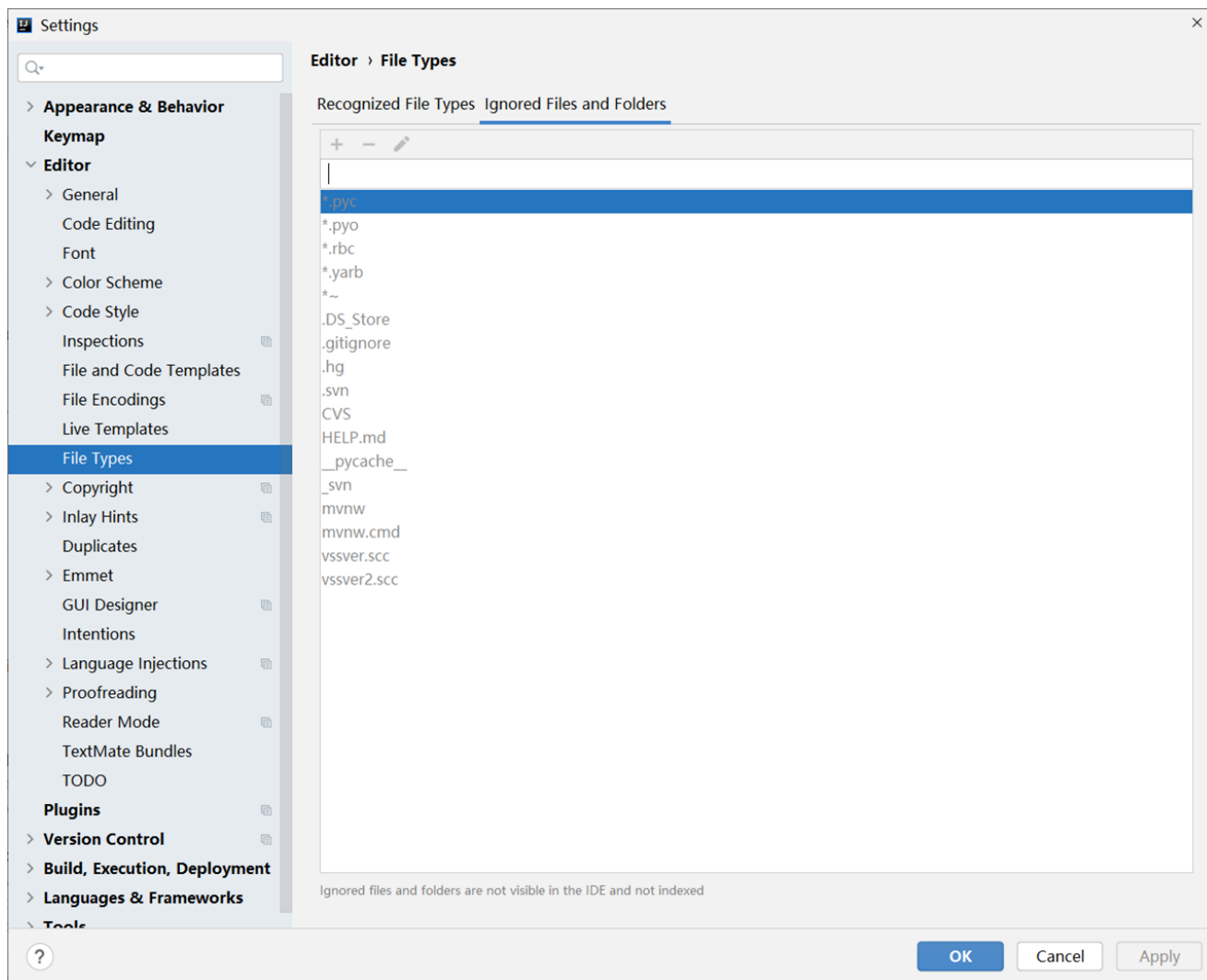


停

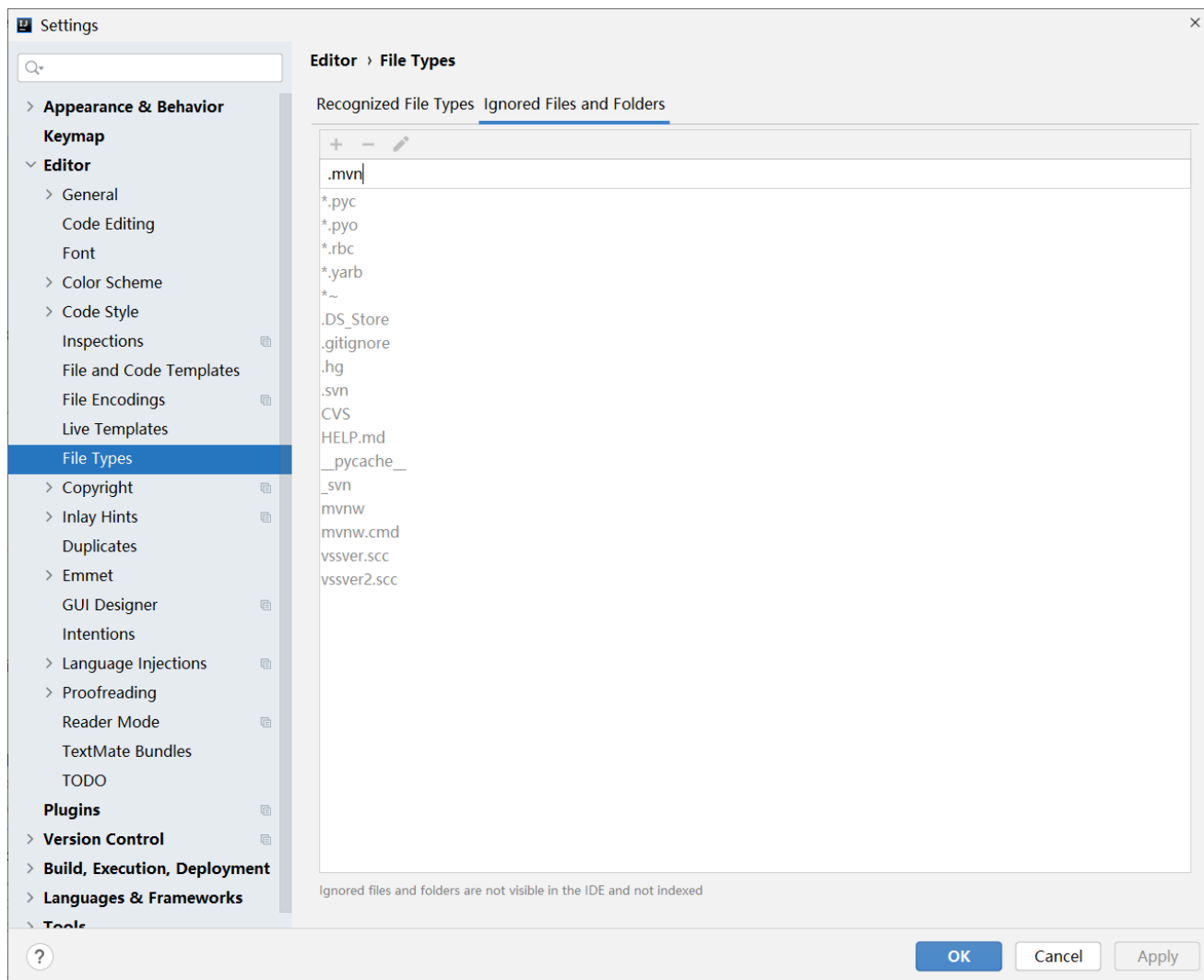
## 教你一招：隐藏指定文件/文件夹



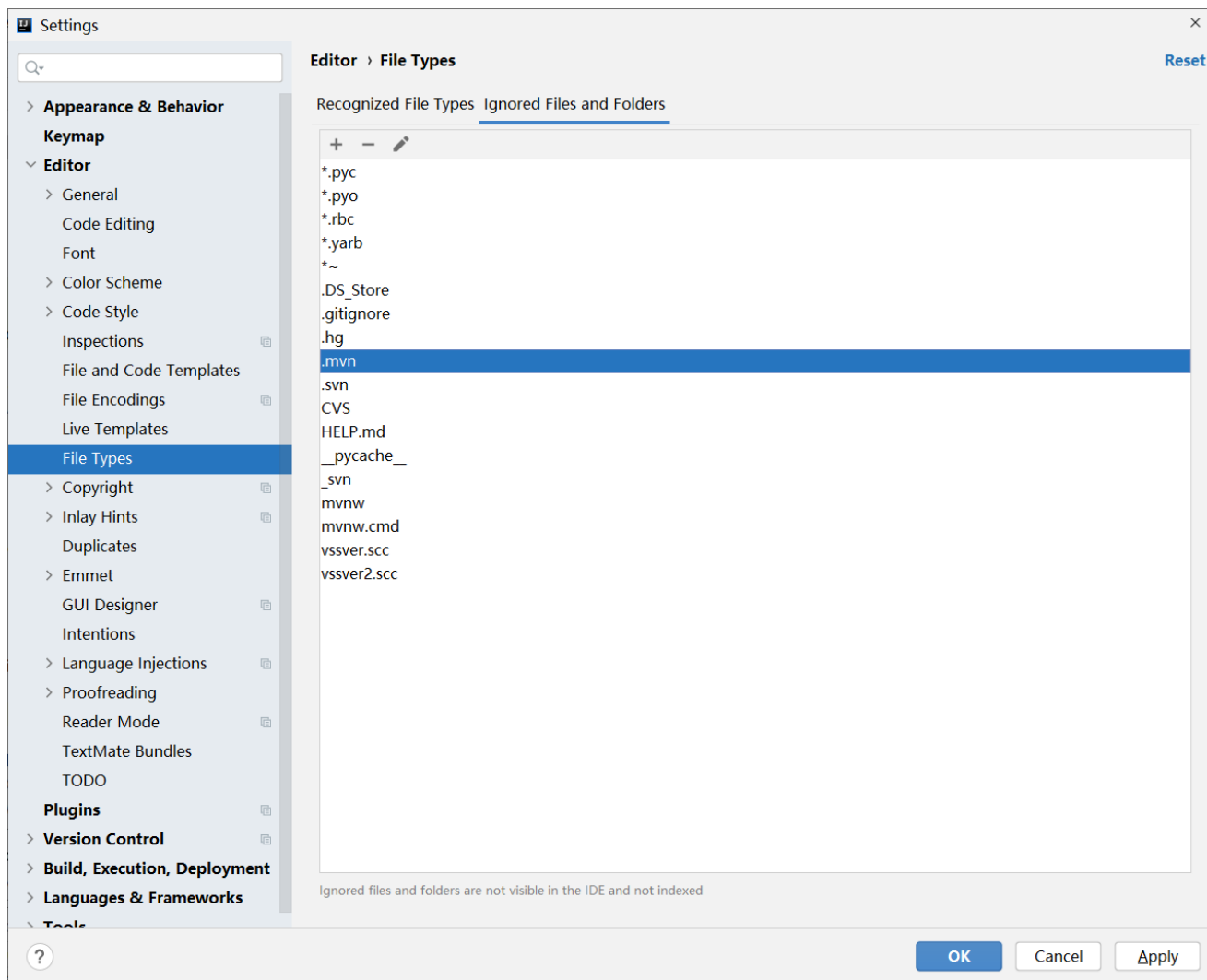
## 教你一招：隐藏指定文件/文件夹



## 教你一招：隐藏指定文件/文件夹



## 教你一招：隐藏指定文件/文件夹





## 小结

### 1. Idea中隐藏指定文件或指定类型文件

- Setting → File Types → Ignored Files and Folders
- 输入要隐藏的文件名，支持\*号通配符
- 回车确认添加

停



## SpringBoot简介

- SpringBoot是由Pivotal团队提供的全新框架，其设计目的是用来简化Spring应用的初始搭建以及开发过程
  - ◆ Spring程序缺点
    - 依赖设置繁琐
    - 配置繁琐
  - ◆ SpringBoot程序优点
    - 起步依赖（简化依赖配置）
    - 自动配置（简化常用工程相关配置）
    - 辅助功能（内置服务器，.....）

## 入门案例解析

- parent
- starter
- 引导类
- 内嵌tomcat

## 入门案例解析

- parent
- starter
- 引导类
- 内嵌tomcat

## 入门案例解析

- parent

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.16</version>
</dependency>

<dependency>
```

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.16</version>
</dependency>

<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.5.6</version>
</dependency>
```

```
<dependency>
  <groupId>itheima</groupId>
  <artifactId>project-dependencies</artifactId>
  <version>1.1.10</version>
</dependency>
```

project-a: pom.xml

project-dependencies: pom.xml

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.16</version>
</dependency>

<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.5.6</version>
</dependency>
```

```
<dependency>
  <groupId>itheima</groupId>
  <artifactId>project-dependencies</artifactId>
  <version>1.1.10</version>
</dependency>
```

project-b: pom.xml

## 入门案例解析

- parent

```
<properties>
  <druid.version>1.1.16</druid.version>
  <mybatis.version>3.5.6</mybatis.version>
  <mysql.version>5.1.47</mysql.version>
  <db2.version>1.2.3</db2.version>
  <oracle.version>2.3.4</oracle.version>
  <sybase.version>3.4.5</sybase.version>
  <dbase.version>4.5.6</dbase.version>
  <foxpro.version>5.6.7</foxpro.version>
  <.....version>6.7.8</.....version>
</properties>
```

①定义一系列的常用坐标版本

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.16</version>
</dependency>
```

```
<groupId>com.alibaba</groupId>
<artifactId>druid</artifactId>
<version>1.1.16</version>
```

②定义一系列的常用坐标组合

```
<groupId>com.alibaba</groupId>
<artifactId>connector-java</artifactId>
<version>1.1.16</version>
```

project-dependencies:pom.xml

```
<parent>
  <groupId>itheima</groupId>
  <artifactId>project-parent</artifactId>
  <version>1.1.10</version>
</parent>
```

③直接使用组合

project-a:pom.xml

```
<parent>
  <groupId>itheima</groupId>
  <artifactId>project-parent</artifactId>
  <version>1.1.10</version>
</parent>
```

project-b:pom.xml

## 入门案例解析

- parent

```
<?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>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.6</version>
    </parent>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>${junit.version}</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>${servlet-api.version}</version>
        </dependency>
    </dependencies>
    <properties>
        <servlet-api.version>4.0.1</servlet-api.version>
        ...
    </properties>
</project>
```



## 小结

1. 开发SpringBoot程序要继承spring-boot-starter-parent
2. spring-boot-starter-parent中定义了若干个依赖管理
3. 继承parent模块可以**避免**多个依赖使用相同技术时出现**依赖版本冲突**
4. 继承parent的形式也可以采用引入依赖的形式实现效果

## 入门案例解析

- parent
- starter
- 引导类
- 内嵌tomcat

仅定义未使用



停

## 入门案例解析

- parent
- starter
- 引导类
- 内嵌tomcat

## 入门案例解析

- starter

- ◆ spring-boot-starter-web.pom

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <version>2.5.4</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>5.3.9</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.3.9</version>
  </dependency>
</dependencies>
```

```
<dependencies>
  <dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-core</artifactId>
    <version>9.0.52</version>
  </dependency>
  <dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-el</artifactId>
    <version>9.0.52</version>
  </dependency>
  <dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-websocket</artifactId>
    <version>9.0.52</version>
    <scope>compile</scope>
  </dependency>
</dependencies>
```

## 入门案例解析


- starter
  - ◆ SpringBoot中常见项目名称，定义了当前项目使用的所有依赖坐标，以达到减少依赖配置的目的
- parent
  - ◆ 所有SpringBoot项目要继承的项目，定义了若干个坐标版本号（依赖管理，而非依赖），以达到减少依赖冲突的目的
  - ◆ spring-boot-starter-parent各版本间存在着诸多坐标版本不同
- 实际开发
  - ◆ 使用任意坐标时，仅书写GAV中的G和A，V由SpringBoot提供，除非SpringBoot未提供对应版本V
  - ◆ 如发生坐标错误，再指定Version（要小心版本冲突）



## 小结

1. 开发SpringBoot程序需要导入坐标时通常导入对应的starter
2. 每个不同的starter根据功能不同，通常包含多个依赖坐标
3. 使用starter可以实现快速配置的效果，达到简化配置的目的

## 入门案例解析

- parent
  - starter
  - 引导类
  - 内嵌tomcat
- 

停

## 入门案例解析

- parent
- starter
- 引导类
- 内嵌tomcat



## 入门案例解析

- 启动方式

```
@SpringBootApplication
public class Springboot01QuickstartApplication {
    public static void main(String[] args) {
        SpringApplication.run(Springboot01QuickstartApplication.class, args);
    }
}
```

- SpringBoot的引导类是Boot工程的执行入口，运行main方法就可以启动项目
- SpringBoot工程运行后初始化Spring容器，扫描引导类所在包加载bean



## 小结

1. SpringBoot工程提供引导类用来启动程序
2. SpringBoot工程启动后创建并初始化Spring容器

## 入门案例解析

- 启动方式

```
@SpringBootApplication
public class Springboot01QuickstartApplication {
    public static void main(String[] args) {
        SpringApplication.run(Springboot01QuickstartApplication.class, args);
    }
}
```

- SpringBoot的引导类是Boot工程的执行入口，运行main方法就可以启动项目
- SpringBoot工程运行后初始化Spring容器，扫描引导类所在包加载bean

未启动Web服务器

停

## 入门案例解析

- parent
- starter
- 引导类
- 内嵌tomcat

## 入门案例解析

- 辅助功能

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

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-tomcat</artifactId>
  <version>2.5.4</version>
</dependency>
```

```
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-core</artifactId>
  <version>9.0.52</version>
</dependency>
```

## 入门案例解析

- 使用maven依赖管理变更起步依赖项

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <!--web起步依赖环境中，排除Tomcat起步依赖-->
    <exclusions>
      <exclusion>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
  <!--添加Jetty起步依赖，版本由SpringBoot的starter控制-->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
  </dependency>
</dependencies>
```

- Jetty比Tomcat更轻量级，可扩展性更强（相较于Tomcat），谷歌应用引擎（GAE）已经全面切换为Jetty

## 内置服务器

- tomcat(默认)      apache出品, 粉丝多, 应用面广, 负载了若干较重的组件
- jetty              更轻量级, 负载性能远不及tomcat
- undertow          undertow, 负载性能勉强跑赢tomcat





## 小结

1. 内嵌Tomcat服务器是SpringBoot辅助功能之一
2. 内嵌Tomcat工作原理是将Tomcat服务器作为对象运行，并将该对象交给Spring容器管理
3. 变更内嵌服务器思想是去除现有服务器，添加全新的服务器



## 总结

1. 入门案例（4种方式）
2. SpringBoot概述
  - parent
  - starter
  - 引导类
  - 辅助功能（内嵌tomcat）

停





## 基础配置

- 属性配置
- 配置文件分类
- yaml文件
- yaml数据读取

## 教你一招：复制工程

- 原则
  - ◆ 保留工程基础结构
  - ◆ 抹掉原始工程痕迹



## 小结

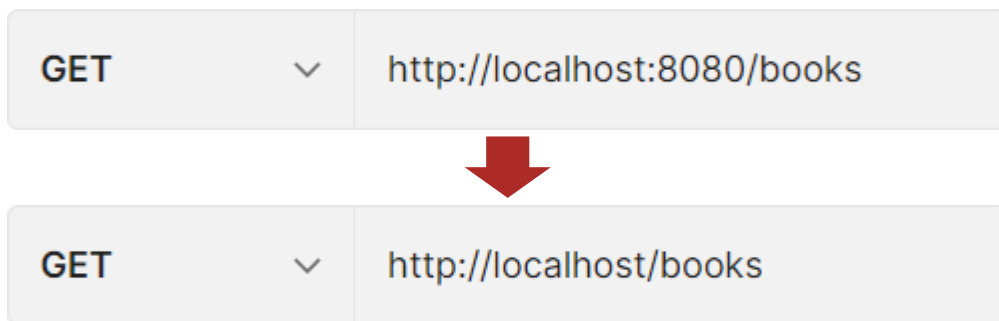
1. 在工作空间中复制对应工程，并修改工程名称
2. 删除与Idea相关配置文件，仅保留src目录与pom.xml文件
3. 修改pom.xml文件中的artifactId与新工程/模块名相同
4. 删除name标签（可选）
5. 保留备份工程供后期使用

停



## 属性配置

- 修改服务器端口



- SpringBoot默认配置文件application.properties，通过键值对配置对应属性

## 属性配置

- 修改配置
  - ◆ 修改服务器端口

```
server.port=80
```



## 小结

1. SpringBoot默认配置文件application.properties

## 属性配置

- 修改配置
  - ◆ 修改服务器端口

`server.port=80`

特定环境

停

## 属性配置

- 修改配置

- ◆ 修改服务器端口

```
server.port=80
```

- ◆ 关闭运行日志图标 (banner)

```
spring.main.banner-mode=off
```

- ◆ 设置日志相关

```
logging.level.root=debug
```

- SpringBoot内置属性查询

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

- ◆ 官方文档中参考文档第一项: Application Properties



## 小结

1. SpringBoot中导入对应starter后，提供对应配置属性
2. 书写SpringBoot配置采用关键字+提示形式书写

## 属性配置

- 配置文件格式

```
server.port=80
server.servlet.context-path=/
logging.level.root=debug
logging.level.com.itheima=info
logging.level.com.itheima.controller=info
logging.level.com.itheima.service=info
logging.level.com.itheima.dao=info
logging.level.com.itheima.domain=info
spring.banner.image.location=logo.png
spring.banner.image.width=120
```

 **痛点**



停

## 属性配置

- SpringBoot提供了多种属性配置方式
  - ◆ application.**properties**

```
server.port=80
```

## 属性配置

- SpringBoot提供了多种属性配置方式

- ◆ application.**properties**

```
server.port=80
```

- ◆ application.**yml**

```
server:  
  port: 81
```

- ◆ application.**yaml**

```
server:  
  port: 82
```



## 小结

### 1. SpringBoot提供了3种配置文件的格式

- properties (传统格式/默认格式)
- **ym1** (主流格式)
- yaml

## 属性配置

- SpringBoot提供了多种属性配置方式

- ◆ application.**properties**

```
server.port=80
```

- ◆ application.**yaml**

```
server:  
  port: 81
```

- ◆ application.**yaml**

```
server:  
  port: 82
```

共存



停

## 属性配置

- SpringBoot配置文件加载顺序
  - ◆ application.**properties** > application.**yml** > application.**yml**

## 属性配置

- SpringBoot配置文件加载顺序
  - ◆ application.**properties** > application.**yaml** > application.**yaml**
- 常用配置文件种类
  - ◆ application.**yaml**





## 小结

1. 配置文件间的加载优先级
  - properties (最高)
  - yml
  - yaml (最低)
2. 不同配置文件中相同配置按照加载优先级相互覆盖，不同配置文件中不同配置全部保留

## 属性配置

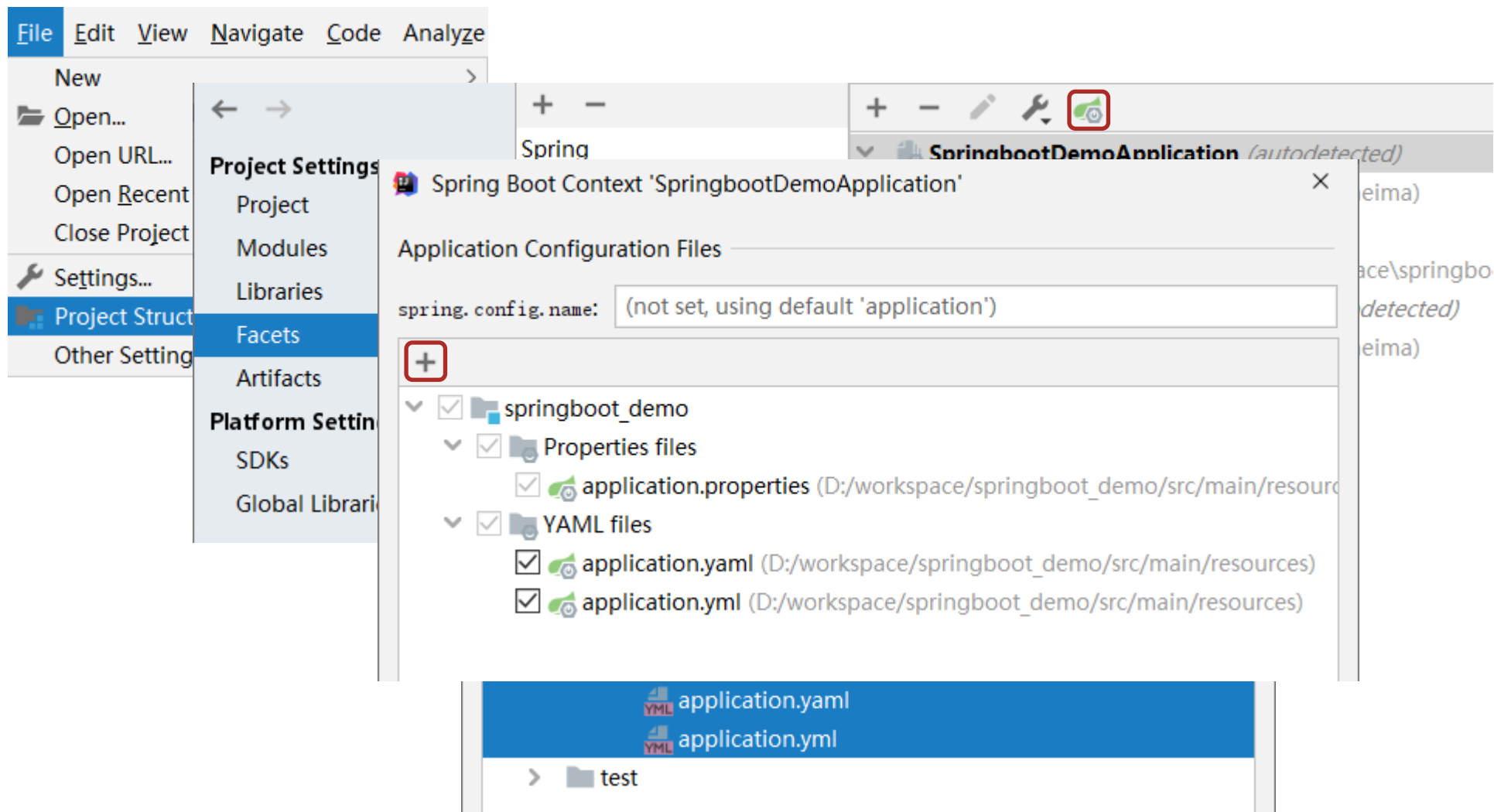
|  |         |
|--|---------|
| port   |         |
| server.port=8080 (Server HTTP port)  | Integer |
| spring.data.cassandra.port=9042 (Port to u...  | Integer |
| spring.data.mongodb.port (Mongo server por...  | Integer |
| spring.integration.rsocket.client.port (TC...  | Integer |
| spring.ldap.embedded.port=0 (Embedded LDAP...  | Integer |
| spring.mail.port (SMTP server port)  | Integer |
| spring.rabbitmq.port (RabbitMQ port)   | Integer |
| spring.redis.port=6379 (Redis server port)   | Integer |
| spring.rsocket.server.port (Server port)   | Integer |
| spring.sendgrid.proxy.port (SendGrid proxy...  | Integer |
| server.tomcat.remoteip.port-header=X-Forwar...   | String  |
| spring.config.import (Import additional List<String>                                       |         |
| Ctrl+向下箭头 and Ctrl+向上箭头 will move caret down and up in the editor <a href="#">Next Tip</a> |         |

|                  |      |
|------------------|------|
| application.yaml |      |
| 1                | port |

痛点

停

## 教你一招：自动提示功能消失解决方案





## 小结

### 1. 指定SpringBoot配置文件

- Setting → Project Structure → Facets
- 选中对应项目/工程
- Customize Spring Boot
- 选择配置文件

## 属性配置

- SpringBoot提供了多种属性配置方式

- ◆ application.**properties**

```
server.port=80
```

- ◆ application.**yaml**

```
server:  
  port: 81
```

← 书写格式

- ◆ application.**yaml**

```
server:  
  port: 82
```

停

## yaml

- YAML (YAML Ain't Markup Language) , 一种数据序列化格式
- 优点:
  - ◆ 容易阅读
  - ◆ 容易与脚本语言交互
  - ◆ 以数据为核心, 重数据轻格式
- YAML文件扩展名
  - ◆ **.yaml (主流)**
  - ◆ .yml

```
<enterprise>
  <name>itcast</name>
  <age>16</age>
  <tel>4006184000</tel>
</enterprise>
```

**XML**

```
enterprise.name=itcast
enterprise.age=16
enterprise.tel=4006184000
```

**Properties**

```
enterprise:
  name: itcast
  age: 16
  tel: 4006184000
```

**yaml**



## yaml语法规则

- 大小写敏感
- 属性层级关系使用多行描述，每行结尾使用冒号结束
- 使用缩进表示层级关系，同层级左侧对齐，只允许使用空格（不允许使用Tab键）
- 属性值前面添加空格（属性名与属性值之间使用冒号+空格作为分隔）
- # 表示注释
- 核心规则：**数据前面要加空格与冒号隔开**

```
enterprise:  
  name: itcast  
  age: 16  
  tel: 4006184000
```

yaml

## yaml语法规则

- 字面值表示方式

|  |  |
|--|--|
| <code>boolean: TRUE</code>                       | <code>#TRUE, true, True, FALSE, false, False均可</code>    |
| <code>float: 3.14</code>                         | <code>#6.8523015e+5 #支持科学计数法</code>                      |
| <code>int: 123</code>                            | <code>#0b1010_0111_0100_1010_1110 #支持二进制、八进制、十六进制</code> |
| <code>null: ~</code>                             | <code>#使用~表示null</code>                                  |
| <code>string: HelloWorld</code>                  | <code>#字符串可以直接书写</code>                                  |
| <code>string2: "Hello World"</code>              | <code>#可以使用双引号包裹特殊字符</code>                              |
| <code>date: 2018-02-17</code>                    | <code>#日期必须使用yyyy-MM-dd格式</code>                         |
| <code>datetime: 2018-02-17T15:02:31+08:00</code> | <code>#时间和日期之间使用T连接, 最后使用+代表时区</code>                    |

## yaml语法规则

- 数组表示方式：在属性名书写位置的下方使用减号作为数据开始符号，每行书写一个数据，减号与数据间空格分隔

```
subject:
- Java
- 前端
- 大数据
enterprise:
  name: itcast
  age: 16
  subject:
    - Java
    - 前端
    - 大数据
```

```
likes: [王者荣耀,刺激战场] #数组书写缩略格式
```

```
users: #对象数组格式
```

```
- name: Tom
  age: 4
- name: Jerry
  age: 5
```

```
users: #对象数组格式二
```

```
-
  name: Tom
  age: 4
-
  name: Jerry
  age: 5
```

#对象数组缩略格式

```
users2: [ { name:Tom , age:4 } , { name:Jerry , age:5 } ]
```



## 小结

### 1. yaml语法规则

- 大小写敏感
- 属性层级关系使用多行描述，每行结尾使用冒号结束
- 使用缩进表示层级关系，同层级左侧对齐，只允许使用空格（不允许使用Tab键）
- 属性值前面添加空格（属性名与属性值之间使用冒号+空格作为分隔）
- # 表示注释

### 2. 注意属性名冒号后面与数据之间有一个**空格**

### 3. 字面值、对象数据格式、数组数据格式（略）

## yaml语法规则

- 数组表示方式：在属性名书写位置的下方使用减号作为数据开始符号，每行书写一个数据，减号与数据间空格分隔

```
subject:
```

- Java
- 前端
- 大数据

```
enterprise:
```

```
  name: itcast
```

```
  age: 16
```

```
  subject:
```

- Java
- 前端
- 大数据

```
likes: [王者荣耀,刺激战场]    #数组书写缩略格式
```

读取数据

停

## yaml数据读取

- 使用@Value读取单个数据，属性名引用方式：**`${一级属性名.二级属性名.....}`**

```
lesson: SpringBoot
```

```
server:
```

```
  port: 82
```

```
enterprise:
```

```
  name: itcast
```

```
  age: 16
```

```
  tel: 4006184000
```

```
  subject:
```

```
    - Java
```

```
    - 前端
```

```
    - 大数据
```

```
@RestController
```

```
@RequestMapping("/books")
```

```
public class BookController {
```

```
    @Value("${lesson}")
```

```
    private String lessonName;
```

```
    @Value("${server.port}")
```

```
    private int port;
```

```
    @Value("${enterprise.subject[1]}")
```

```
    private String[] subject_01;
```

```
}
```



## 小结

1. 使用@Value配合SpEL读取单个数据
2. 如果数据存在多层级，依次书写层级名称即可



## yaml数据读取

- 使用@Value读取单个数据，属性名引用方式：\${一级属性名.二级属性名.....}

```
center:  
  dataDir: /usr/local/fire/data  
  tmpDir: /usr/local/fire/tmp  
  logDir: /usr/local/fire/log  
  msgDir: /usr/local/fire/msgDir
```

痛点

```
center:  
  dataDir: D:/usr/local/fire/data  
  tmpDir: D:/usr/local/fire/tmp  
  logDir: D:/usr/local/fire/log  
  msgDir: D:/usr/local/fire/msgDir
```

痛点

停

## yaml数据读取

- 在配置文件中可以使用属性名引用方式引用属性

```
baseDir: /usr/local/fire

center:
  dataDir: ${baseDir}/data
  tmpDir: ${baseDir}/tmp
  logDir: ${baseDir}/log
  msgDir: ${baseDir}/msgDir
```

- 属性值中如果出现转移字符，需要使用双引号包裹

```
lesson: "Spring\tboot\nlesson"
```



## 小结

1. 在配置文件中可以使用`${属性名}`方式引用属性值
2. 如果属性中出现特殊字符，可以使用双引号包裹起来作为字符解析

## yaml数据读取

- 使用@Value读取单个数据，属性名引用方式：**`${一级属性名.二级属性名.....}`**

```
lesson: SpringBoot
```

```
server:
```

```
  port: 82
```

```
enterprise:
```

```
  name: itcast
```

```
  age: 16
```

```
  tel: 4006184000
```

```
  subject:
```

- Java
- 前端
- 大数据

痛点

```
@RestController
```

```
@RequestMapping("/books")
```

```
public class BookController {
```

```
    @Value("${lesson}")
```

```
    private String lessonName;
```

```
    @Value("${server.port}")
```

```
    private int port;
```

```
    @Value("${enterprise.subject[1]}")
```

```
    private String[] subject_01;
```

```
}
```

停

## yaml数据读取

- 封装全部数据到Environment对象

lesson: SpringBoot

server:

port: 82

enterprise:

name: itcast

age: 16

tel: 4006184000

subject:

- Java

- 前端

- 大数据

```
@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private Environment env;
    @GetMapping("/{id}")
    public String getById(@PathVariable Integer id){
        System.out.println(env.getProperty("lesson"));
        System.out.println(env.getProperty("enterprise.name"));
        System.out.println(env.getProperty("enterprise.subject[0]"));
        return "hello , spring boot!";
    }
}
```



## 小结

1. 使用Environment对象封装全部配置信息
2. 使用@Autowired自动装配数据到Environment对象中



## yaml数据读取

- 封装全部数据到Environment对象

```
lesson: SpringBoot
```

```
server:
```

```
  port: 82
```

```
enterprise:
```

```
  name: itcast
```

```
  age: 16
```

```
  tel: 4006184000
```

```
  subject:
```

- Java
- 前端
- 大数据

 **痛点**

```
@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private Environment env;
    @GetMapping("/{id}")
    public String getById(@PathVariable Integer id){
        System.out.println(env.getProperty("lesson"));
        System.out.println(env.getProperty("enterprise.name"));
        System.out.println(env.getProperty("enterprise.subject[0]"));
        return "hello , spring boot!";
    }
}
```

停

## yaml数据读取

- 自定义对象封装指定数据

```
lesson: SpringBoot
```

```
server:  
  port: 82
```

```
enterprise:  
  name: itcast  
  age: 16  
  tel: 4006184000  
  subject:  
    - Java  
    - 前端  
    - 大数据
```

```
@Component
```

```
@ConfigurationProperties(prefix = "enterprise")
```

```
public class Enterprise {  
    private String name;  
    private Integer age;  
    private String[] subject;  
}
```

```
@RestController
```

```
@RequestMapping("/books")
```

```
public class BookController {  
    @Autowired  
    private Enterprise enterprise;  
}
```

## yaml数据读取

- 自定义对象封装指定数据的作用

```
datasource:  
  driver-class-name: com.mysql.cj.jdbc.Driver  
  url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC  
  username: root  
  password: root
```

```
@Component  
@ConfigurationProperties(prefix = "datasource")  
public class DataSource {  
    private String driverClassName;  
    private String url;  
    private String userName;  
    private String password;  
}
```



## 小结

1. 使用@ConfigurationProperties注解绑定配置信息到封装类中
2. 封装类需要定义为Spring管理的bean, 否则无法进行属性注入





## 整合第三方技术

- 整合JUnit
- 整合MyBatis
- 整合MyBatis-Plus
- 整合Druid

## 整合JUnit

- SpringBoot整合JUnit

```
@SpringBootTest
class Springboot07JUnitApplicationTests {

    @Autowired
    private BookService bookService;

    @Test
    public void testSave(){
        bookService.save();
    }
}
```



## 整合JUnit

- 名称: @SpringBootTest
- 类型: **测试类注解**
- 位置: 测试类定义上方
- 作用: 设置JUnit加载的SpringBoot启动类
- 范例:

```
@SpringBootTest  
class Springboot05JUnitApplicationTests {}
```



## 小结

1. 导入测试对应的starter
2. 测试类使用@SpringBootTest修饰
3. 使用自动装配的形式添加要测试的对象

停

## 整合JUnit

- 名称: @SpringBootTest
- 类型: 测试类注解
- 位置: 测试类定义上方
- 作用: 设置JUnit加载的SpringBoot启动类
- 范例:

```
@SpringBootTest(classes = Springboot05JUnitApplication.class)
class Springboot07JUnitApplicationTests {}
```

- 相关属性
  - ◆ classes: 设置SpringBoot启动类

### 注意事项

如果测试类在SpringBoot启动类的包或子包中, 可以省略启动类的设置, 也就是省略classes的设定



## 小结

1. 测试类如果存在于引导类所在包或子包中无需指定引导类
2. 测试类如果不存在于引导类所在的包或子包中需要通过classes属性指定引导类

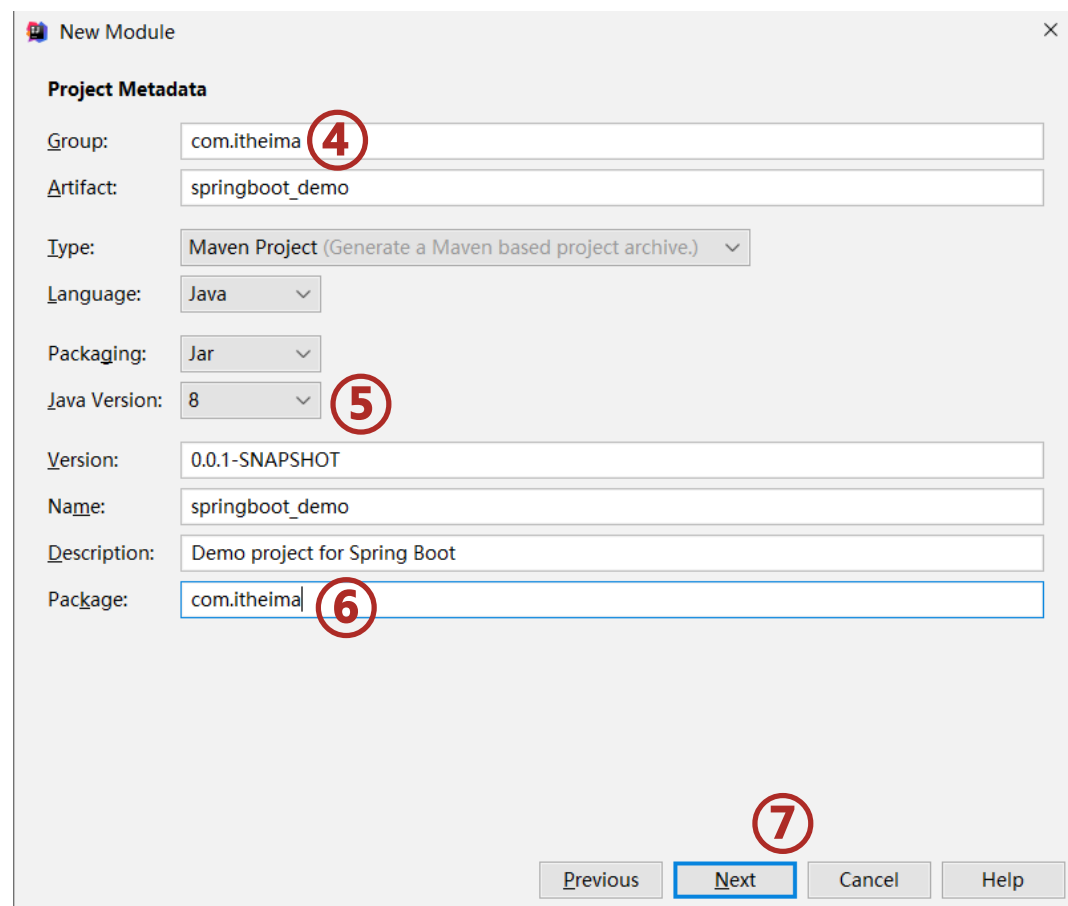
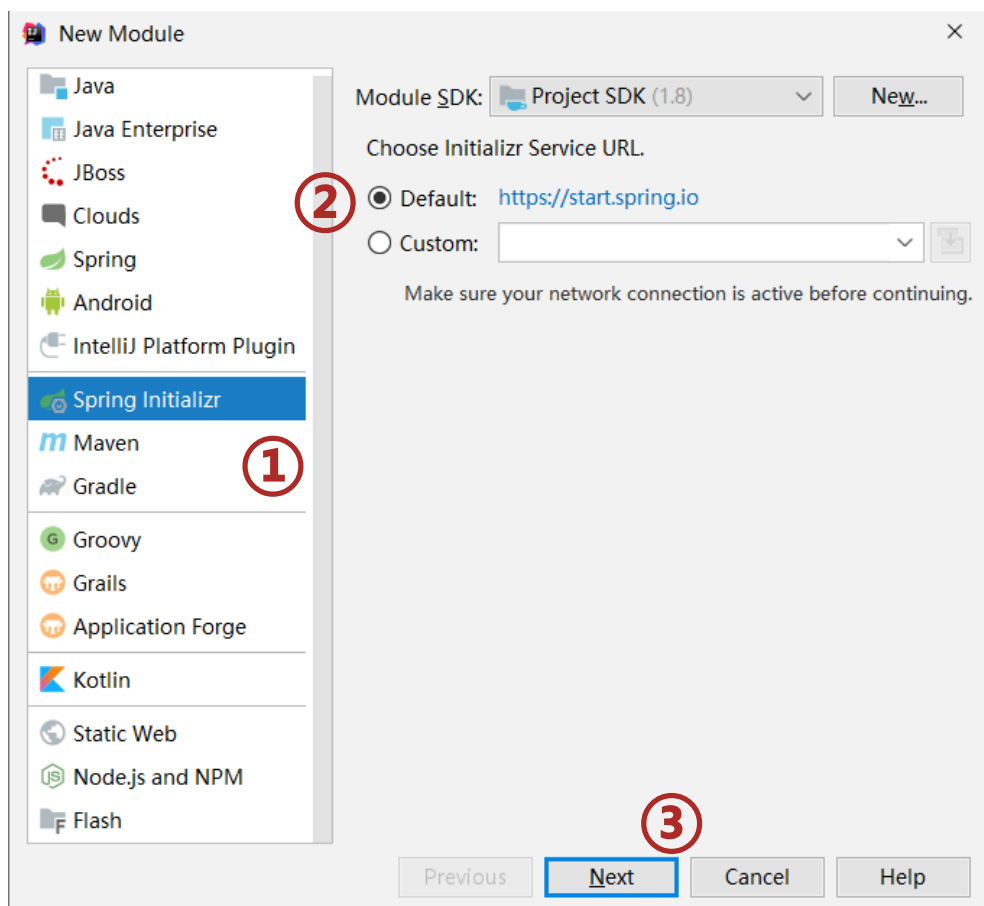
停

## 整合MyBatis

- 核心配置：数据库连接相关信息（连什么？连谁？什么权限）
- 映射配置：SQL映射（XML/注解）

## 步骤 SpringBoot整合MyBatis

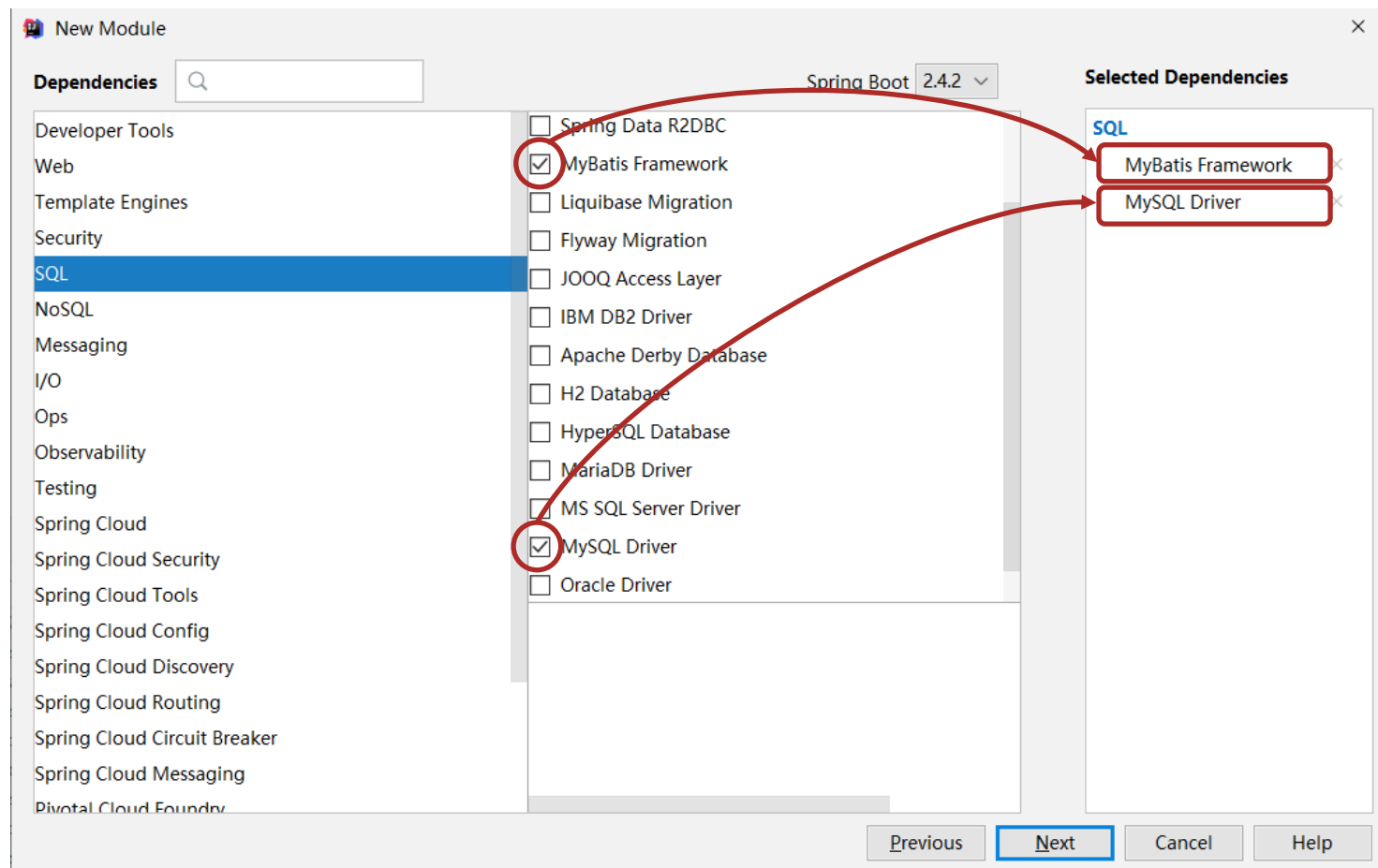
①：创建新模块，选择Spring初始化，并配置模块相关基础信息





## 步骤 SpringBoot整合MyBatis

②：选择当前模块需要使用的技术集（MyBatis、MySQL）



## 步骤 SpringBoot整合MyBatis

### ③：设置数据源参数

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/ssm_db
    username: root
    password: root
```

#### 注意事项

SpringBoot版本低于2.4.3(不含), Mysql驱动版本大于8.0时, 需要在url连接串中配置时区

```
jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC
```

或在MySQL数据库端配置时区解决此问题



## SpringBoot整合MyBatis

### ④：定义数据层接口与映射配置

```
@Mapper
public interface UserDao {
    @Select("select * from user")
    public List<User> getAll();
}
```

## 步骤 SpringBoot整合MyBatis

⑤：测试类中注入dao接口，测试功能

```
@SpringBootTest
class Springboot08MybatisApplicationTests {

    @Autowired
    private BookDao bookDao;

    @Test
    public void testGetById() {
        Book book = bookDao.getById(1);
        System.out.println(book);
    }
}
```



## 小结

1. 勾选MyBatis技术，也就是导入MyBatis对应的starter
2. 数据库连接相关信息转换成配置
3. 数据库SQL映射需要添加@Mapper被容器识别到

## 步骤 SpringBoot整合MyBatis

### ③：设置数据源参数

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/ssm_db
    username: root
    password: root
```

#### 注意事项

SpringBoot版本低于2.4.3(不含), Mysql驱动版本大于8.0时, 需要在url连接串中配置时区

```
jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC
```

或在MySQL数据库端配置时区解决此问题



## 小结

1. MySQL 8.X驱动强制要求设置时区
  - 修改url, 添加serverTimezone设定
  - 修改MySQL数据库配置 (略)
2. 驱动类过时, 提醒更换为com.mysql.cj.jdbc.Driver

停



## 整合MyBatis-Plus

- MyBatis-Plus与MyBatis区别
  - ◆ 导入坐标不同
  - ◆ 数据层实现简化

## 步骤 SpringBoot整合MyBatis-Plus

①：手动添加SpringBoot整合MyBatis-Plus的坐标，可以通过mvnrepository获取

```
<dependency>
  <groupId>com.baomidou</groupId>
  <artifactId>mybatis-plus-boot-starter</artifactId>
  <version>3.4.3</version>
</dependency>
```

### 注意事项

由于SpringBoot中未收录MyBatis-Plus的坐标版本，需要指定对应的Version

## 步骤 SpringBoot整合MyBatis-Plus

②：定义数据层接口与映射配置，继承**BaseMapper**

```
@Mapper
public interface UserDao extends BaseMapper<User> {
}
```



步骤

## SpringBoot整合MyBatis-Plus

③：其他同SpringBoot整合MyBatis

(略)



## 小结

1. 手工添加MyBatis-Plus对应的starter
2. 数据层接口使用BaseMapper简化开发
3. 需要使用的第三方技术无法通过勾选确定时，需要手工添加坐标

停

## 整合Druid

## 整合Druid

- 指定数据源类型

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC
    username: root
    password: root
    type: com.alibaba.druid.pool.DruidDataSource
```



## 整合Druid

- 导入Druid对应的starter

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid-spring-boot-starter</artifactId>
  <version>1.2.6</version>
</dependency>
```

- 变更Druid的配置方式

```
spring:
  datasource:
    druid:
      driver-class-name: com.mysql.cj.jdbc.Driver
      url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC
      username: root
      password: root
```

## 整合任意第三方技术

- 导入对应的starter
- 配置对应的设置或采用默认配置



## 小结

1. 整合Druid需要导入Druid对应的starter
2. 根据Druid提供的配置方式进行配置
3. 整合第三方技术通用方式
  - 导入对应的starter
  - 根据提供的配置格式，配置非默认值对应的配置项

停



## SSMP整合案例

- 案例效果演示
- 案例实施方案分析
  - ◆ 实体类开发——使用Lombok快速制作实体类
  - ◆ Dao开发——整合MyBatisPlus，制作数据层测试类
  - ◆ Service开发——基于MyBatisPlus进行增量开发，制作业务层测试类
  - ◆ Controller开发——基于Restful开发，使用PostMan测试接口功能
  - ◆ Controller开发——前后端开发协议制作
  - ◆ 页面开发——基于VUE+ElementUI制作，前后端联调，页面数据处理，页面消息处理
    - 列表、新增、修改、删除、分页、查询
  - ◆ 项目异常处理
  - ◆ 按条件查询——页面功能调整、Controller修正功能、Service修正功能



## 小结

### 1. SSMP案例效果演示

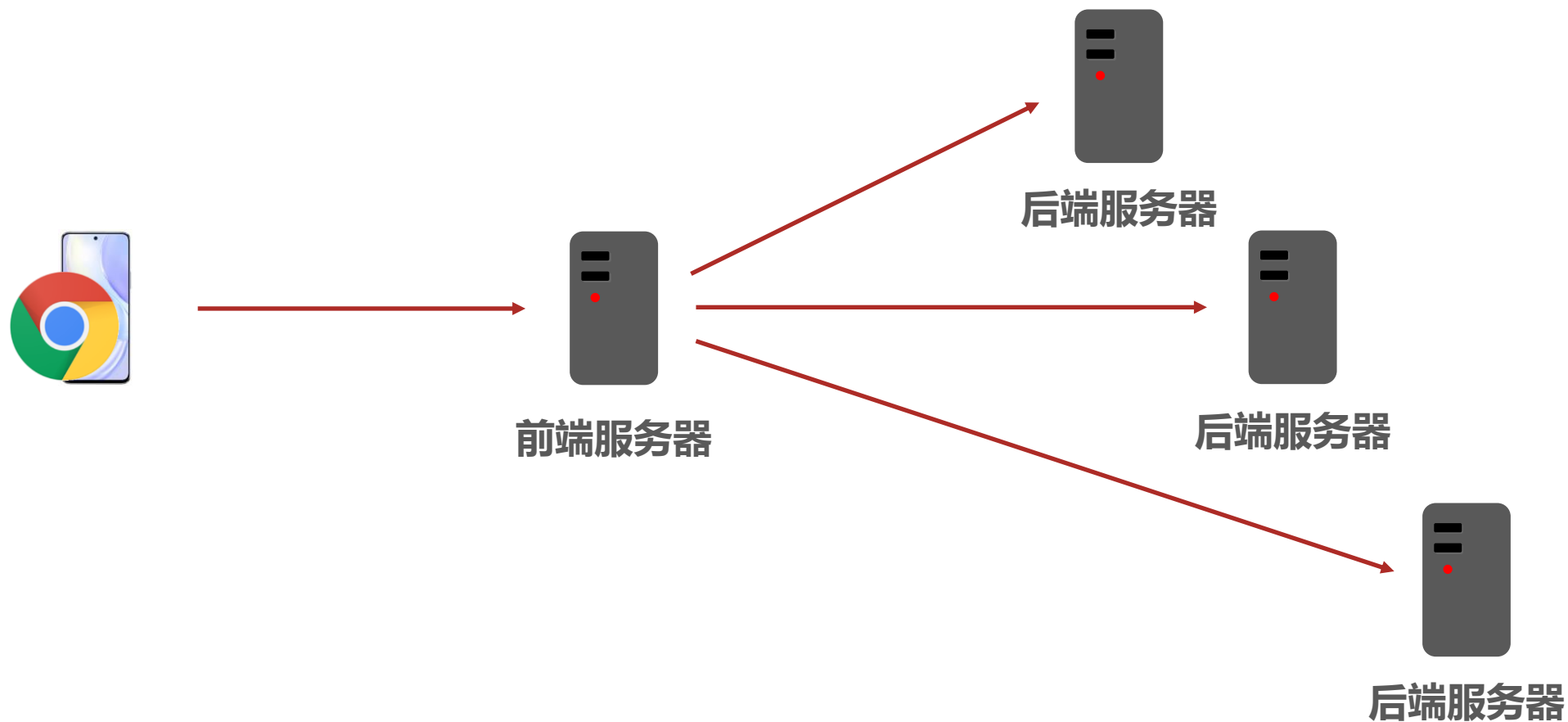
### 2. SSMP案例制作流程解析

- 先开发基础CRUD功能，做一层测一层
- 调通页面，确认异步提交成功后，制作所有功能
- 添加分页功能与查询功能

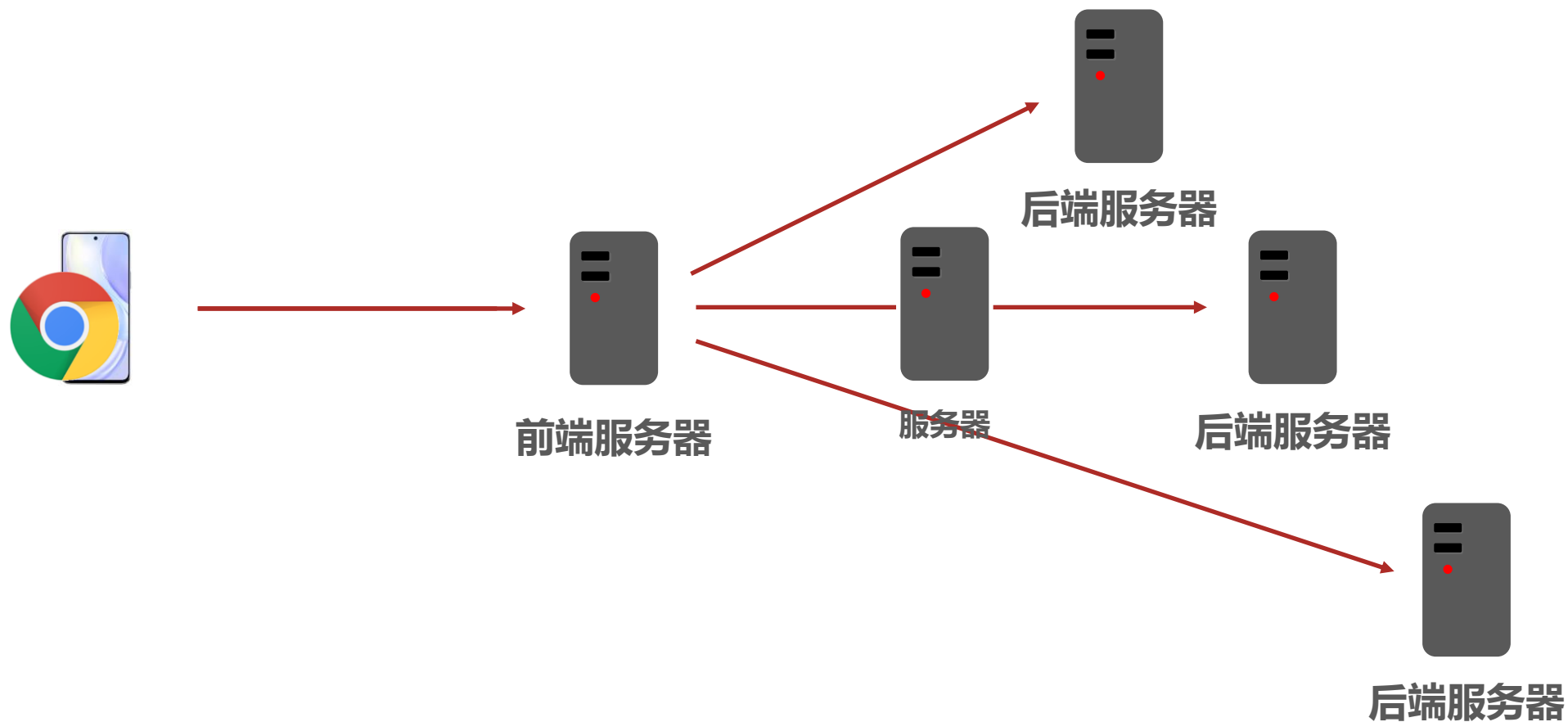
停



## 模块创建



## 模块创建





## 小结

1. 勾选SpringMVC与MySQL坐标
2. 修改配置文件为ym1格式
3. 设置端口为80方便访问

停

## 实体类开发

- Lombok, 一个Java类库, 提供了一组注解, 简化POJO实体类开发

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

- lombok版本由SpringBoot提供, 无需指定版本

## 实体类开发

- 常用注解: @Data

```
@Data
public class Book {
    private Integer id;
    private String type;
    private String name;
    private String description;
}
```

- 为当前实体类在编译期设置对应的get/set方法, toString方法, hashCode方法, equals方法等



## 小结

1. 实体类制作
2. 使用lombok简化开发
  - 导入lombok无需指定版本，由SpringBoot提供版本
  - @Data注解

停



## 数据层开发

- 技术实现方案
  - ◆ MyBatisPlus
  - ◆ Druid

## 数据层开发

- 导入MyBatisPlus与Druid对应的starter

```
<dependency>
  <groupId>com.baomidou</groupId>
  <artifactId>mybatis-plus-boot-starter</artifactId>
  <version>3.4.3</version>
</dependency>
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid-spring-boot-starter</artifactId>
  <version>1.2.6</version>
</dependency>
```

## 数据层开发

- 配置数据源与MyBatisPlus对应的基础配置 (id生成策略使用数据库自增策略)

```
spring:
  datasource:
    druid:
      driver-class-name: com.mysql.cj.jdbc.Driver
      url: jdbc:mysql://localhost:3306/ssm_db?servierTimezone=UTC
      username: root
      password: root

mybatis-plus:
  global-config:
    db-config:
      table-prefix: tbl_
      id-type: auto
```

## 数据层开发

- 继承BaseMapper并指定泛型

```
@Mapper
public interface BookDao extends BaseMapper<Book> {
}
```

## 数据层开发

- 制作测试类测试结果

```
@SpringBootTest
public class BookDaoTest {
    @Autowired
    private BookDao bookDao;
    @Test
    void testSave() {
        Book book = new Book();
        book.setName("测试数据");
        book.setType("测试类型");
        book.setDescription("测试描述数据");
        bookDao.insert(book);
    }
    @Test
    void testGetById() {
        System.out.println(bookDao.selectById(13));
    }
    ...
}
```



## 小结

1. 手工导入starter坐标 (2个)
2. 配置数据源与MyBatisPlus对应的配置
3. 开发Dao接口 (继承BaseMapper)
4. 制作测试类测试Dao功能是否有效

## 数据层开发

- 为方便调试可以开启MyBatisPlus的日志

```
mybatis-plus:  
  configuration:  
    log-impl: org.apache.ibatis.logging.stdout.StdoutImpl
```



## 小结

1. 使用配置方式开启日志，设置日志输出方式为标准输出



停

## 数据层开发——分页功能

- 分页操作需要设定分页对象IPage

```
@Test
void testGetPage(){
    IPage page = new Page(1,5);
    bookDao.selectPage(page,null);
}
```

- IPage对象中封装了分页操作中的所有数据
  - ◆ 数据
  - ◆ 当前页码值
  - ◆ 每页数据总量
  - ◆ 最大页码值
  - ◆ 数据总量

## 数据层开发——分页功能

- 分页操作是在MyBatisPlus的常规操作基础上增强得到，内部是动态的拼写SQL语句，因此需要增强对应的功能，使用MyBatisPlus拦截器实现

```
@Configuration
public class MpConfig {
    @Bean
    public MybatisPlusInterceptor mpInterceptor() {
        //1. 定义Mp拦截器
        MybatisPlusInterceptor mpInterceptor = new MybatisPlusInterceptor();
        //2. 添加具体的拦截器
        mpInterceptor.addInnerInterceptor(new PaginationInnerInterceptor());
        return mpInterceptor;
    }
}
```



## 小结

1. 使用IPage封装分页数据
2. 分页操作依赖MyBatisPlus分页拦截器实现功能
3. 借助MyBatisPlus日志查阅执行SQL语句

停

## 数据层开发——条件查询功能

- 使用QueryWrapper对象封装查询条件，推荐使用LambdaQueryWrapper对象，所有查询操作封装成方法调用

```
@Test
void testGetByCondition(){
    IPage page = new Page(1,10);
    LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();
    lqw.like(Book::getName,"Spring");
    bookDao.selectPage(page,lqw);
}
```

```
@Test
void testGetByCondition(){
    QueryWrapper<Book> qw = new QueryWrapper<Book>();
    qw.like("name","Spring");
    bookDao.selectList(qw);
}
```

## 数据层开发——条件查询功能

- 支持动态拼写查询条件

```
@Test
void testGetByCondition(){
    String name = "Spring";
    IPage page = new Page(1,10);
    LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();
    lqw.like(Strings.isNotEmpty(name),Book::getName,"Spring");
    bookDao.selectPage(page,lqw);
}
```



## 小结

1. 使用QueryWrapper对象封装查询条件
2. 推荐使用LambdaQueryWrapper对象
3. 所有查询操作封装成方法调用
4. 查询条件支持动态条件拼装



停

## 业务层开发

- Service层接口定义与数据层接口定义具有较大区别，不要混用
  - ◆ `selectByUsernameAndPassword(String username,String password);`
  - ◆ `login(String username,String password);`

## 业务层开发

- 接口定义

```
public interface BookService {  
    boolean save(Book book);  
    boolean delete(Integer id);  
    boolean update(Book book);  
    Book getById(Integer id);  
    List<Book> getAll();  
    IPage<Book> getByPage(int currentPage,int pageSize);  
}
```

## 业务层开发

- 实现类定义

```
@Service
public class BookServiceImpl implements BookService {
    @Autowired
    private BookDao bookDao;

    public Boolean save(Book book) {
        return bookDao.insert(book) > 0;
    }

    public Boolean delete(Integer id) {
        return bookDao.deleteById(id) > 0;
    }

    public Boolean update(Book book) {
        return bookDao.updateById(book) > 0;
    }
}
```

## 业务层开发

- 实现类定义

```
@Service
public class BookServiceImpl implements BookService {
    @Autowired
    private BookDao bookDao;

    public Book getById(Integer id) {
        return bookDao.selectById(id);
    }

    public List<Book> getAll() {
        return bookDao.selectList(null);
    }

    public IPage<Book> getByPage(int currentPage, int pageSize) {
        IPage page = new Page<Book>(currentPage, pageSize);
        return bookDao.selectPage(page, null);
    }
}
```

## 业务层开发

- 测试类定义

```
@SpringBootTest
public class BookServiceTest {
    @Autowired
    private BookService bookService;
    @Test
    void testGetById(){
        bookService.getById(9);
    }
    @Test
    void testGetAll(){
        bookService.getAll();
    }
    @Test
    void testGetByPage(){
        bookService.getByPage(1,5);
    }
    ... ..
}
```



## 小结

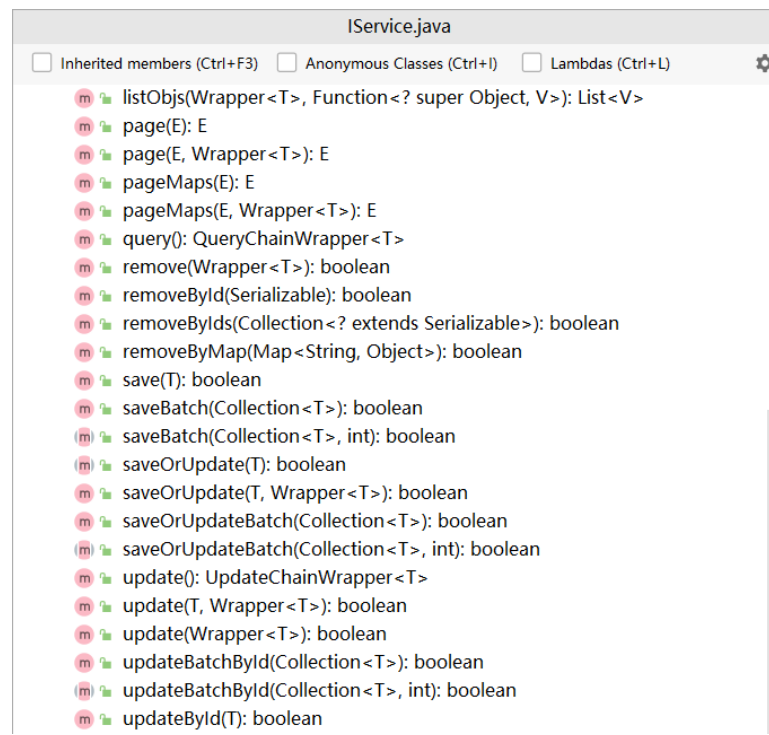
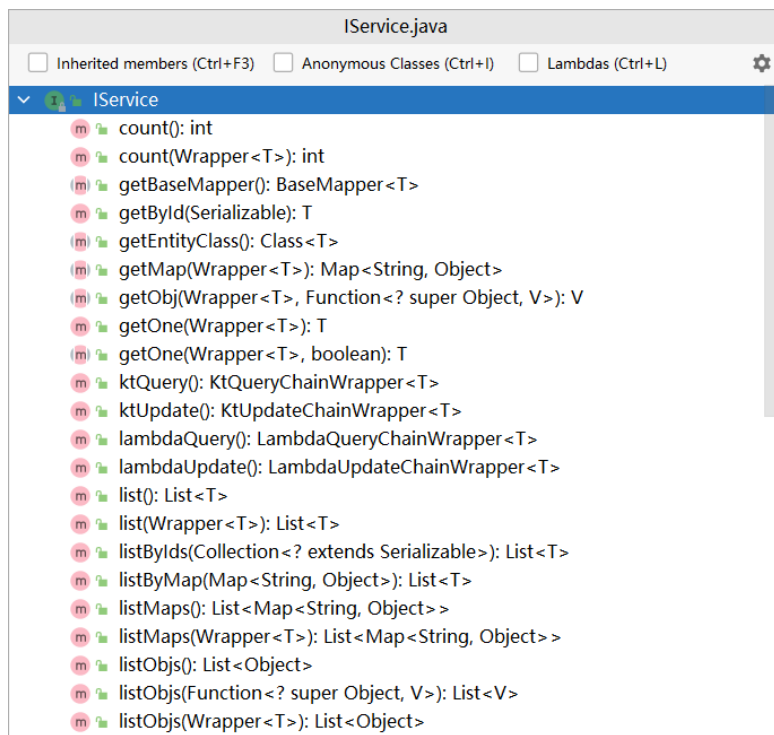
1. Service接口名称定义成业务名称，并与Dao接口名称进行区分
2. 制作测试类测试Service功能是否有效

停



## 业务层开发——快速开发

- 快速开发方案
  - 使用MyBatisPlus提供有业务层通用接口（IServivce<T>）与业务层通用实现类（ServiceImpl<M,T>）
  - 在通用类基础上做功能重载或功能追加
  - 注意重载时不要覆盖原始操作，避免原始提供的功能丢失



## 业务层开发——快速开发

- 接口定义

```
public interface IBookService extends IService<Book> {  
}
```

```
public interface IBookService extends IService<Book> {  
    //追加的操作与原始操作通过名称区分, 功能类似  
    Boolean delete(Integer id);  
    Boolean insert(Book book);  
    Boolean modify(Book book);  
    Book get(Integer id);  
}
```

- remove(Wrapper<T>): boolean
- removeById(Serializable): boolean
- removeByIds(Collection<? extends Serializable>): boolean
- removeByMap(Map<String, Object>): boolean
- save(T): boolean
- saveBatch(Collection<T>): boolean
- saveBatch(Collection<T>, int): boolean
- saveOrUpdate(T): boolean
- saveOrUpdate(T, Wrapper<T>): boolean
- saveOrUpdateBatch(Collection<T>): boolean
- saveOrUpdateBatch(Collection<T>, int): boolean
- update(): UpdateChainWrapper<T>
- update(T, Wrapper<T>): boolean
- update(Wrapper<T>): boolean
- updateBatchById(Collection<T>): boolean
- updateBatchById(Collection<T>, int): boolean
- updateById(T): boolean

## 业务层开发——快速开发

- 实现类定义

```
@Service
public class BookServiceImpl2 extends ServiceImpl<BookDao,Book> implements IBookService {
}
```

## 业务层开发——快速开发

- 实现类追加功能

```
@Service
public class BookServiceImpl2 extends ServiceImpl<BookDao,Book> implements IBookService {
    @Autowired
    private BookDao bookDao;
    public Boolean insert(Book book) {
        return bookDao.insert(book) > 0;
    }
    public Boolean modify(Book book) {
        return bookDao.updateById(book) > 0;
    }
    public Boolean delete(Integer id) {
        return bookDao.deleteById(id) > 0;
    }
    public Book get(Integer id) {
        return bookDao.selectById(id);
    }
}
```

## 业务层开发——快速开发

- 测试类定义（略）



## 小结

1. 使用通用接口 (`IService<T>`) 快速开发Service
2. 使用通用实现类 (`ServiceImpl<M,T>`) 快速开发ServiceImpl
3. 可以在通用接口基础上做功能重载或功能追加
4. 注意重载时不要覆盖原始操作, 避免原始提供的功能丢失

停

## 表现层开发

- 基于Restful进行表现层接口开发
- 使用Postman测试表现层接口功能



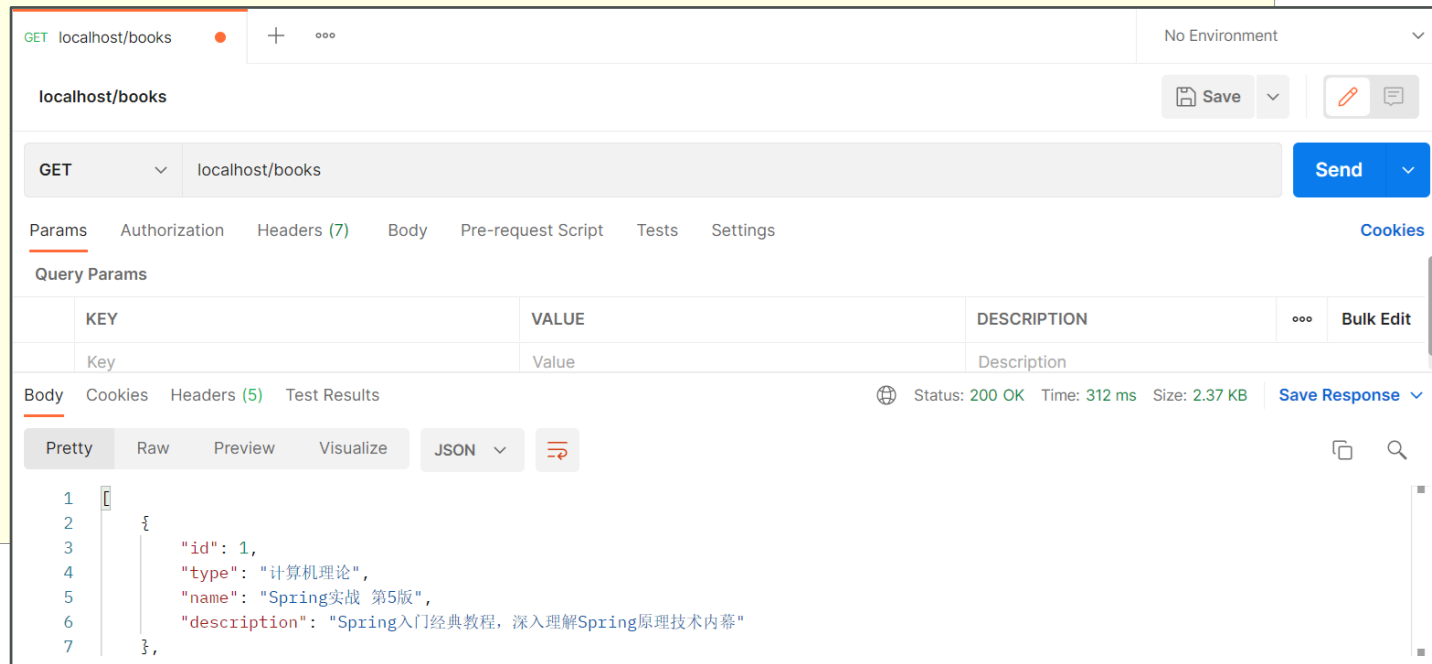
## 表现层开发

- 功能测试

```
@RestController
@RequestMapping("/books")
public class BookController {

    @Autowired
    private IBookService bookService;

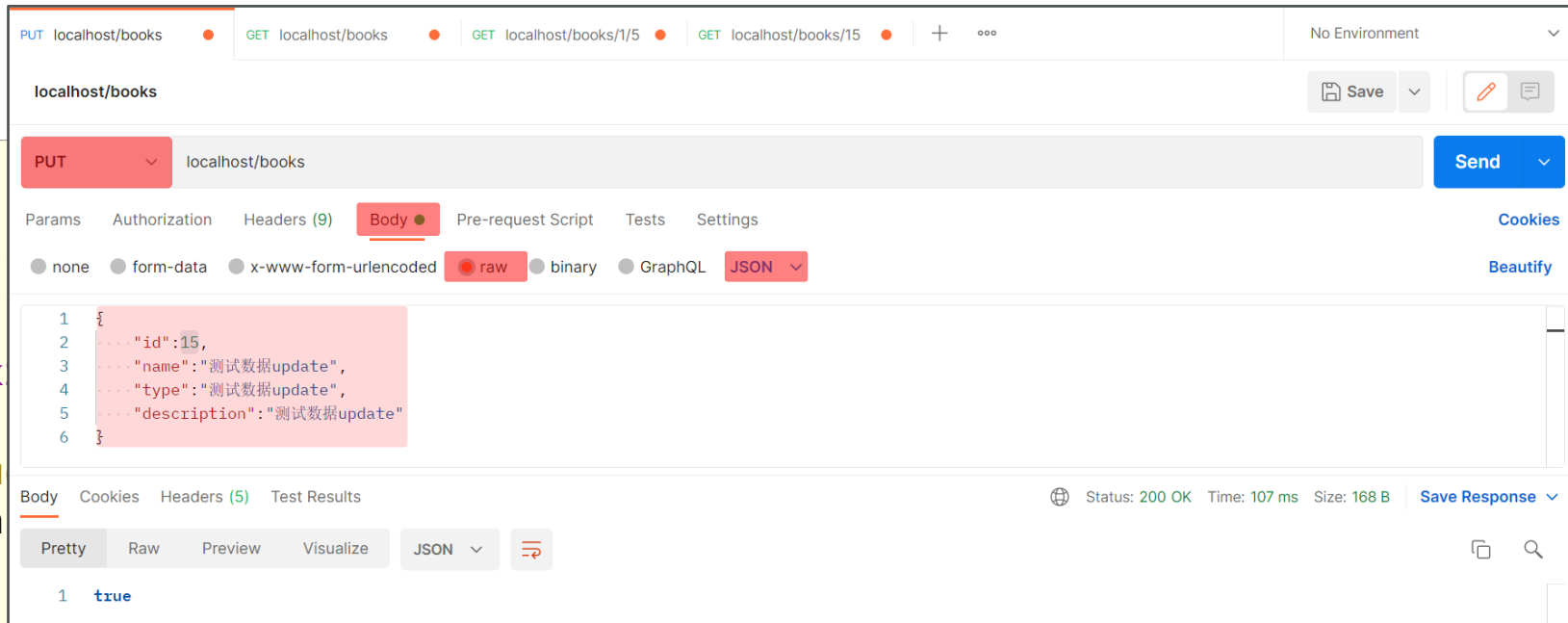
    @GetMapping
    public List<Book> getAll(){
        return bookService.list();
    }
}
```



## 表现层开发

- 表现层接口开发

```
@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private IBookService book
    @PostMapping
    public Boolean save(@RequestBody Book book) {
        return bookService.save(book);
    }
    @PutMapping
    public Boolean update(@RequestBody Book book) {
        return bookService.modify(book);
    }
    @DeleteMapping("/{id}")
    public Boolean delete(@PathVariable Integer id) {
        return bookService.delete(id);
    }
}
```



## 表现层开发

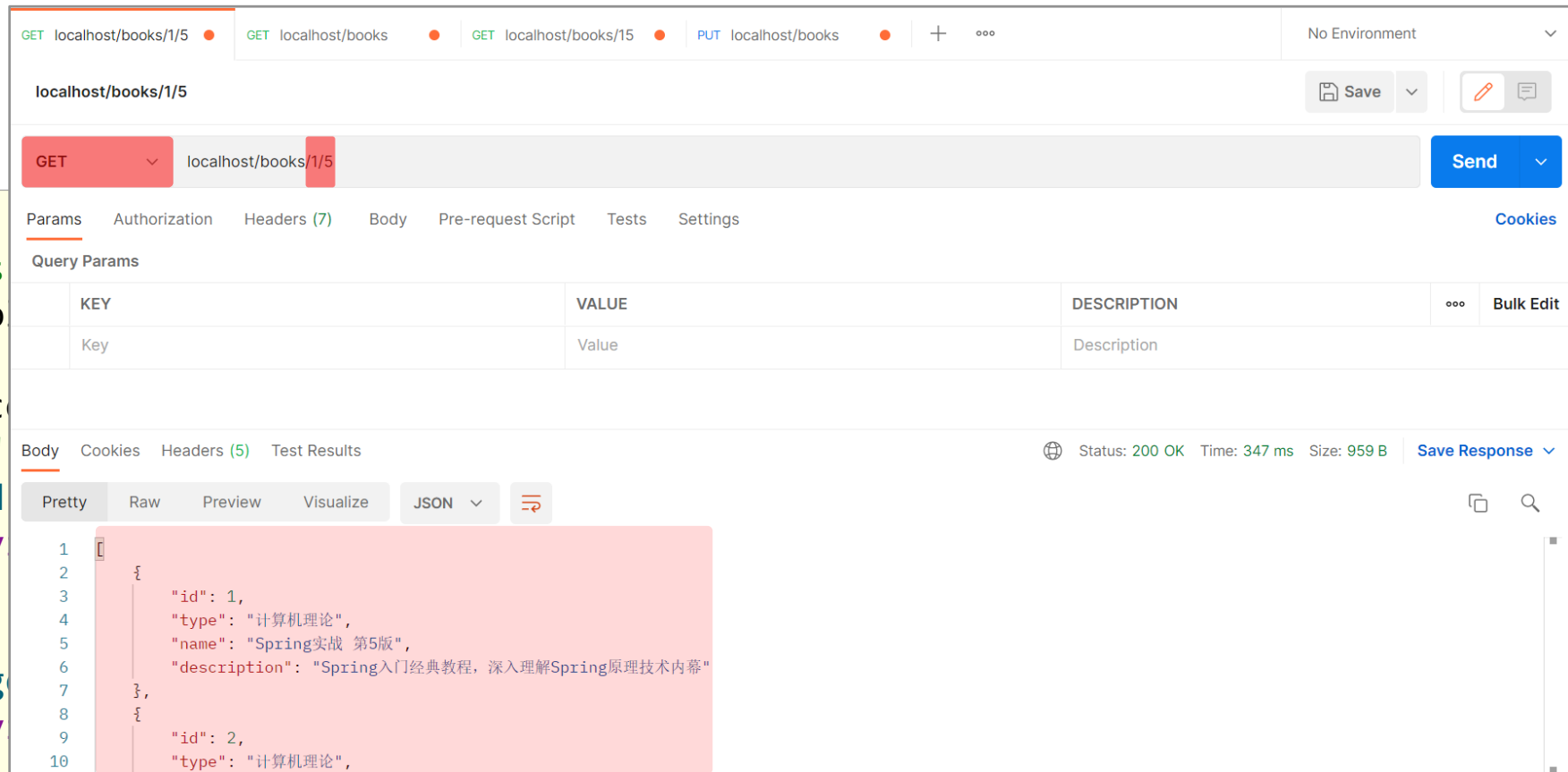
- 表现层接口开发

```
@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private IBookService bookService;

    @GetMapping("/{id}")
    public Book getById(Integer id) {
        return bookService.getById(id);
    }

    @GetMapping
    public List<Book> getAll() {
        return bookService.getAll();
    }

    @GetMapping("/{currentPage}/{pageSize}")
    public List<Book> getAll(@PathVariable Integer currentPage, @PathVariable Integer pageSize) {
        return bookService.getPage(currentPage, pageSize).getRecords();
    }
}
```





## 小结

### 1. 基于Restful制作表现层接口

- 新增: POST
- 删除: DELETE
- 修改: PUT
- 查询: GET

### 2. 接收参数

- 实体数据: @RequestBody
- 路径变量: @PathVariable

停

## 表现层消息一致性处理

- 增删改

true

格式A

- 查单条

格式B

```
{
  "id": 1,
  "type": "计算机理论",
  "name": "Spring实战 第5版",
  "description": "Spring入门经典教程"
}
```

- 查全部

格式C

```
[
  {
    "id": 1,
    "type": "计算机理论",
    "name": "Spring实战 第5版",
    "description": "Spring入门经典教程"
  },
  {
    "id": 2,
    "type": "计算机理论",
    "name": "Spring 5核心原理与30个类手写实战",
    "description": "十年沉淀之作"
  }
]
```

格式D

格式E

**统一格式**

格式G

格式F

## 表现层消息一致性处理

- 增删改

```
{  
  "data":true  
}
```

- 查单条

```
{  
  "data":{  
    "id": 1,  
    "type": "计算机理论",  
    "name": "Spring实战 第5版",  
    "description": "Spring入门经典教程"  
  }  
}
```

```
{  
  "data": null  
}
```

**数据?**

- 查全部

```
{  
  "data":[  
    {  
      "id": 1,  
      "type": "计算机理论",  
      "name": "Spring实战 第5版",  
      "description": "Spring入门经典教程"  
    },  
    {  
      "id": 2,  
      "type": "计算机理论",  
      "name": "Spring 5核心原理与30个类手写实战",  
      "description": "十年沉淀之作"  
    }  
  ]  
}
```

- 查询id不存在的数据, 返回null
- 查询过程中抛出异常, catch中返回null

## 表现层消息一致性处理

### ● 增删改

```
{
  "flag": true,
  "data": null
}
```

**成功**

```
{
  "flag": false,
  "data": null
}
```

**失败**

### ● 查单条

```
{
  "flag": true,
  "data": {
    "id": 1,
    "type": "计算机理论",
    "name": "Spring实战 第5版",
    "description": "Spring入门经典教程"
  }
}
```

```
{
  "flag": true,
  "data": null
}
```

**查询数据不存在**

```
{
  "flag": false,
  "data": null
}
```

**查询过程抛异常**

### ● 查全部

```
{
  "flag": true,
  "data": [
    {
      "id": 1,
      "type": "计算机理论",
      "name": "Spring实战 第5版",
      "description": "Spring入门经典教程"
    },
    {
      "id": 2,
      "type": "计算机理论",
      "name": "Spring 5核心原理与30个类手写实战",
      "description": "十年沉淀之作"
    }
  ]
}
```



## 表现层消息一致性处理

- 设计表现层返回结果的模型类，用于后端与前端进行数据格式统一，也称为**前后端数据协议**

```
@Data
public class R{
    private Boolean flag;
    private Object data;

    public R(){
    }

    public R(Boolean flag){
        this.flag = flag;
    }

    public R(Boolean flag,Object data){
        this.flag = flag;
        this.data = data;
    }
}
```

## 表现层消息一致性处理

- 表现层接口统一返回值类型结果

```
@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private IBookService bookService;
    @PostMapping
    public R save(@RequestBody Book book){
        Boolean flag = bookService.insert(book);
        return new R(flag);
    }
    @PutMapping
    public R update(@RequestBody Book book){
        Boolean flag = bookService.modify(book);
        return new R(flag);
    }
}
```

## 表现层消息一致性处理

- 表现层接口统一返回值类型结果

```
@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private IBookService bookService;
    @DeleteMapping("/{id}")
    public R delete(@PathVariable Integer id){
        Boolean flag = bookService.delete(id);
        return new R(flag);
    }
    @GetMapping("/{id}")
    public R getById(@PathVariable Integer id){
        Book book = bookService.getById(id);
        return new R(true,book);
    }
}
```

## 表现层消息一致性处理

- 表现层接口统一返回值类型结果

```
@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private IBookService bookService;
    @GetMapping
    public R getAll(){
        List<Book> bookList = bookService.list();
        return new R(true,bookList);
    }
    @GetMapping("/{currentPage}/{pageSize}")
    public R getAll(@PathVariable Integer currentPage,@PathVariable Integer pageSize){
        IPage<Book> page = bookService.getPage(currentPage, pageSize);
        return new R(true,page);
    }
}
```



## 小结

1. 设计统一的返回值结果类型便于前端开发读取数据
2. 返回值结果类型可以根据需求自行设定，没有固定格式
3. 返回值结果模型类用于后端与前端进行数据格式统一，也称为前后端数据协议

停

## 前后端协议联调

- 前后端分离结构设计中页面归属前端服务器
- 单体工程中页面放置在resources目录下的static目录中（建议执行clean）

## 前后端协议联调

- 前端发送异步请求，调用后端接口

```
// 列表  
getAll() {  
    axios.get("/books").then((res)=>{  
        console.log(res.data);  
    });  
},
```





## 小结

1. 单体项目中页面放置在resources/static目录下
2. created钩子函数用于初始化页面时发起调用
3. 页面使用axios发送异步请求获取数据后确认前后端是否联  
通

停

## 前后端协议联调

- 列表页

```
// 列表  
getAll() {  
    axios.get("/books").then((res)=>{  
        this.dataList = res.data.data;  
    });  
},
```



## 小结

1. 将查询数据返回到页面，利用前端数据双向绑定进行数据展示

停

## 前后端协议联调

- 弹出添加窗口

```
//弹出添加窗口  
handleCreate() {  
    this.dialogFormVisible = true;  
},
```

## 前后端协议联调

- 清除数据

```
//重置表单  
resetForm() {  
    this.formData = {};  
},
```

```
//弹出添加窗口  
handleCreate() {  
    this.dialogFormVisible = true;  
    this.resetForm();  
},
```

## 前后端协议联调

- 添加

```
//添加
handleAdd () {
    //发送异步请求
    axios.post("/books",this.formData).then((res)=>{
        //如果操作成功, 关闭弹层, 显示数据
        if(res.data.flag){
            this.dialogFormVisible = false;
            this.$message.success("添加成功");
        }else {
            this.$message.error("添加失败");
        }
    }).finally(()=>{
        this.getAll();
    });
},
```



## 前后端协议联调

- 取消添加

```
//取消  
cancel(){  
    this.dialogFormVisible = false;  
    this.$message.info("操作取消");  
},
```



## 小结

1. 请求方式使用POST调用后台对应操作
2. 添加操作结束后动态刷新页面加载数据
3. 根据操作结果不同，显示对应的提示信息
4. 弹出添加Div时清除表单数据

停

## 前后端协议联调

- 删除

```
// 删除
handleDelete(row) {
    axios.delete("/books/"+row.id).then((res)=>{
        if(res.data.flag){
            this.$message.success("删除成功");
        }else{
            this.$message.error("删除失败");
        }
    }).finally(()=>{
        this.getAll();
    });
}
```

## 前后端协议联调

- 删除

```
// 删除
handleDelete(row) {
    //1.弹出提示框
    this.$confirm("此操作永久删除当前数据, 是否继续? ", "提示", {
        type: 'info'
    }).then(()=>{
        //2.做删除业务
        axios.delete("/books/"+row.id).then((res)=>{
            .....
        }).finally(()=>{
            this.getAll();
        });
    }).catch(()=>{
        //3.取消删除
        this.$message.info("取消删除操作");
    });
}
```



## 小结

1. 请求方式使用Delete调用后台对应操作
2. 删除操作需要传递当前行数据对应的id值到后台
3. 删除操作结束后动态刷新页面加载数据
4. 根据操作结果不同，显示对应的提示信息
5. 删除操作前弹出提示框避免误操作

停

## 前后端协议联调

- 弹出修改窗口

```
//弹出编辑窗口
handleUpdate(row) {
    axios.get("/books/"+row.id).then((res)=>{
        if(res.data.flag){
            //展示弹层, 加载数据
            this.formData = res.data.data;
            this.dialogFormVisible4Edit = true;
        }else{
            this.$message.error("数据同步失败, 自动刷新");
        }
    });
},
```



## 前后端协议联调

- 删除消息维护

```
// 删除
handleDelete(row) {
    axios.delete("/books/"+row.id).then((res)=>{
        if(res.data.flag){
            this.$message.success("删除成功");
        }else{
            this.$message.error("数据同步失败, 自动刷新");
        }
    }).finally(()=>{
        this.getAll();
    });
}
```



## 小结

1. 加载要修改数据通过传递当前行数据对应的id值到后台查询数据
2. 利用前端数据双向绑定将查询到的数据进行回显

停

## 前后端协议联调

- 修改

```
//修改
handleEdit() {
  axios.put("/books",this.formData).then((res)=>{
    //如果操作成功, 关闭弹层并刷新页面
    if(res.data.flag){
      this.dialogFormVisible4Edit = false;
      this.$message.success("修改成功");
    }else {
      this.$message.error("修改失败, 请重试");
    }
  }).finally(()=>{
    this.getAll();
  });
},
```

## 前后端协议联调

- 取消添加和修改

```
cancel(){  
    this.dialogFormVisible = false;  
    this.dialogFormVisible4Edit = false;  
    this.$message.info("操作取消");  
},
```



## 小结

1. 请求方式使用PUT调用后台对应操作
2. 修改操作结束后动态刷新页面加载数据（同新增）
3. 根据操作结果不同，显示对应的提示信息（同新增）

停

## 业务消息一致性处理

- 业务操作成功或失败返回数据格式

```
{  
  "flag": true,  
  "data": null  
}
```

```
{  
  "flag": false,  
  "data": null  
}
```

- 后台代码BUG导致数据格式不统一性

```
{  
  "timestamp": "2021-09-15T03:27:31.038+00:00",  
  "status": 500,  
  "error": "Internal Server Error",  
  "path": "/books"  
}
```



## 业务消息一致性处理

- 对异常进行统一处理，出现异常后，返回指定信息

```
@RestControllerAdvice
public class ProjectExceptionHandler {
    @ExceptionHandler(Exception.class)
    public R doOtherException(Exception ex){
        //记录日志
        //发送消息给运维
        //发送邮件给开发人员, ex对象发送给开发人员
        ex.printStackTrace();
        return new R(false, null, "系统错误, 请稍后再试! ");
    }
}
```

## 业务消息一致性处理

- 修改表现层返回结果的模型类，封装出现异常后对应的信息
  - ◆ flag: false
  - ◆ Data: null
  - ◆ 消息(msg): 要显示信息

```
@Data
public class R{
    private Boolean flag;
    private Object data;
    private String msg;

    public R(Boolean flag,Object data,String msg){
        this.flag = flag;
        this.data = data;
        this.msg = msg;
    }
}
```

## 业务消息一致性处理

- 页面消息处理，没有传递消息加载默认消息，传递消息后加载指定消息

```
//添加
handleAdd () {
    //发送ajax请求
    axios.post("/books",this.formData).then((res)=>{
        //如果操作成功, 关闭弹层, 显示数据
        if(res.data.flag){
            this.dialogFormVisible = false;
            this.$message.success("添加成功");
        }else {
            this.$message.error(res.data.msg);
        }
    }).finally(()=>{
        this.getAll();
    });
},
```

## 业务消息一致性处理

- 可以在表现层Controller中进行消息统一处理

```
@PostMapping
public R save(@RequestBody Book book) throws IOException {
    Boolean flag = bookService.insert(book);
    return new R(flag , flag ? "添加成功^^" : "添加失败-_-!");
}
```

- 目的：国际化

## 业务消息一致性处理

- 页面消息处理

```
//添加
handleAdd () {
    //发送ajax请求
    axios.post("/books",this.formData).then((res)=>{
        if(res.data.flag){
            this.dialogFormVisible = false;
            this.$message.success(res.data.msg);
        }else {
            this.$message.error(res.data.msg);
        }
    }).finally(()=>{
        this.getAll();
    });
},
```



## 小结

1. 使用注解@RestControllerAdvice定义SpringMVC异常处理器用来处理异常的
2. 异常处理器必须被扫描加载，否则无法生效
3. 表现层返回结果的模型类中添加消息属性用来传递消息到页面

停

## 分页功能

- 页面使用el分页组件添加分页功能

```
<!-- 分页组件 -->
<div class="pagination-container">
  <el-pagination
    class="pagiantion"
    @current-change="handleCurrentChange"
    :current-page="pagination.currentPage"
    :page-size="pagination.pageSize"
    layout="total, prev, pager, next, jumper"
    :total="pagination.total">
  </el-pagination>
</div>
```



## 分页功能

- 定义分页组件需要使用的数据并将数据绑定到分页组件

```
data:{  
    pagination: {           //分页相关模型数据  
        currentPage: 1,    //当前页码  
        pageSize:10,       //每页显示的记录数  
        total:0,           //总记录数  
    }  
},
```

## 分页功能

- 替换查询全部功能为分页功能

```
getAll() {  
    axios.get("/books/"+this.pagination.currentPage+"/"+this.pagination.pageSize).then((res) => {  
    });  
},
```

## 分页功能

- 分页查询

```
@GetMapping("/{currentPage}/{pageSize}")  
public R getAll(@PathVariable Integer currentPage,@PathVariable Integer pageSize){  
    IPage<Book> pageBook = bookService.getPage(currentPage, pageSize);  
    return new R(null != pageBook ,pageBook);  
}
```

- 使用路径参数传递分页数据或封装对象传递数据

## 分页功能

- 加载分页数据

```
getAll() {  
    axios.get("/books/"+this.pagination.currentPage+"/"+this.pagination.pageSize).then((res) => {  
        this.pagination.total = res.data.data.total;  
        this.pagination.currentPage = res.data.data.current;  
        this.pagination.pagesize = res.data.data.size;  
        this.dataList = res.data.data.records;  
    });  
},
```

## 分页功能

- 分页页码值切换

```
// 切换页码  
handleCurrentChange(currentPage) {  
    this.pagination.currentPage = currentPage;  
    this.getAll();  
},
```



## 小结

1. 使用e1分页组件
2. 定义分页组件绑定的数据模型
3. 异步调用获取分页数据
4. 分页数据页面回显

停

## 删除功能维护

- 对查询结果进行校验，如果当前页码值大于最大页码值，使用最大页码值作为当前页码值重新查询

```
@GetMapping("/{currentPage}/{pageSize}")
public R getPage(@PathVariable int currentPage,@PathVariable int pageSize){
    IPage<Book> page = bookService.getPage(currentPage, pageSize);
    //如果当前页码值大于了总页码值, 那么重新执行查询操作, 使用最大页码值作为当前页码值
    if( currentPage > page.getPages()){
        page = bookService.getPage((int)page.getPages(), pageSize);
    }
    return new R(true, page);
}
```





## 小结

### 1. 基于业务需求维护删除功能

停

## 条件查询功能

- 查询条件数据封装
  - ◆ 单独封装
  - ◆ 与分页操作混合封装

```
pagination: {                //分页相关模型数据
    currentPage: 1,          //当前页码
    pageSize:10,              //每页显示的记录数
    total:0,                  //总记录数
    name: "",
    type: "",
    description: ""
}
```

## 条件查询功能

- 页面数据模型绑定

```
<div class="filter-container">
  <el-input placeholder="图书类别" v-model="pagination.type" class="filter-item"/>
  <el-input placeholder="图书名称" v-model="pagination.name" class="filter-item"/>
  <el-input placeholder="图书描述" v-model="pagination.description" class="filter-item"/>
  <el-button @click="getAll()" class="dalfBut">查询</el-button>
  <el-button type="primary" class="butT" @click="handleCreate()">新建</el-button>
</div>
```

## 条件查询功能

- 组织数据成为get请求发送的数据

```
getAll() {  
    //1. 获取查询条件, 拼接查询条件  
    param = "?name="+this.pagination.name;  
    param += "&type="+this.pagination.type;  
    param += "&description="+this.pagination.description;  
    console.log("-----"+ param);  
    axios.get("/books/"+this.pagination.currentPage+"/"+this.pagination.pageSize+param)  
        .then((res) => {  
            this.dataList = res.data.data.records;  
        });  
},
```

- 条件参数组织可以通过条件判定书写的更简洁

## 条件查询功能

- Controller接收参数

```
@GetMapping("/{currentPage}/{pageSize}")
public R getAll(@PathVariable int currentPage,@PathVariable int pageSize,Book book) {
    System.out.println("参数====>"+book);
    IPage<Book> pageBook = bookService.getPage(currentPage,pageSize);
    return new R(null != pageBook ,pageBook);
}
```

## 条件查询功能

- 业务层接口功能开发

```
public interface IBookService extends IService<Book> {  
    IPage<Book> getPage(Integer currentPage,Integer pageSize,Book queryBook);  
}
```

```
@Service  
public class BookServiceImpl2 extends ServiceImpl<BookDao,Book> implements IBookService {  
    public IPage<Book> getPage(Integer currentPage,Integer pageSize,Book queryBook){  
        IPage page = new Page(currentPage,pageSize);  
        LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();  
        lqw.like(Strings.isNotEmpty(queryBook.getName()),Book::getName,queryBook.getName());  
        lqw.like(Strings.isNotEmpty(queryBook.getType()),Book::getType,queryBook.getType());  
        lqw.like(Strings.isNotEmpty(queryBook.getDescription()),  
                Book::getDescription,queryBook.getDescription());  
        return bookDao.selectPage(page,lqw);  
    }  
}
```

## 条件查询功能

- Controller调用业务层分页条件查询接口

```
@GetMapping("/{currentPage}/{pageSize}")
public R getAll(@PathVariable int currentPage,@PathVariable int pageSize,Book book) {
    IPage<Book> pageBook = bookService.getPage(currentPage,pageSize,book);
    return new R(null != pageBook ,pageBook);
}
```



## 条件查询功能

- 页面回显数据

```
getAll() {  
    //1. 获取查询条件, 拼接查询条件  
    param = "?name="+this.pagination.name;  
    param += "&type="+this.pagination.type;  
    param += "&description="+this.pagination.description;  
    console.log("-----"+ param);  
    axios.get("/books/"+this.pagination.currentPage+"/"+this.pagination.pageSize+param)  
        .then((res) => {  
            this.pagination.total = res.data.data.total;  
            this.pagination.currentPage = res.data.data.current;  
            this.pagination.pagesize = res.data.data.size;  
            this.dataList = res.data.data.records;  
        });  
},
```



## 小结

1. 定义查询条件数据模型（当前封装到分页数据模型中）
2. 异步调用分页功能并通过请求参数传递数据到后台

停

## 基于SpringBoot的SSMP整合案例

### 1. pom.xml

配置起步依赖

### 2. application.yml

设置数据源、端口、框架技术相关配置等

### 3. dao

继承BaseMapper、设置@Mapper

### 4. dao测试类

### 5. service

调用数据层接口或MyBatis-Plus提供的接口快速开发

### 6. service测试类

### 7. controller

基于Restful开发，使用Postman测试跑通功能

### 8. 页面

放置在resources目录下的static目录中



## 小结

### 1. 基于SpringBoot的SSM整合案例



## 总结

1. 整合JUnit
2. 整合MyBatis
3. 整合MyBatis-Plus
4. 整合Druid
5. 基于SpringBoot的SSMP整合案例

## 后续学习

- 基础篇
  - ◆ 能够创建SpringBoot工程
  - ◆ 基于SpringBoot实现ssm/ssmp整合
- 实用篇
  - ◆ 运维实用篇
  - ◆ 开发实用篇
- 原理篇

## 后续学习

- 基础篇
  - ◆ 能够创建SpringBoot工程
  - ◆ 基于SpringBoot实现ssm/ssmp整合
- 实用篇
  - ◆ 运维实用篇
    - 能够掌握SpringBoot程序多环境开发
    - 能够基于Linux系统发布SpringBoot工程
    - 能够解决线上灵活配置SpringBoot工程的需求
  - ◆ 开发实用篇
    - 能够基于SpringBoot整合任意第三方技术
- 原理篇





# Spring Boot

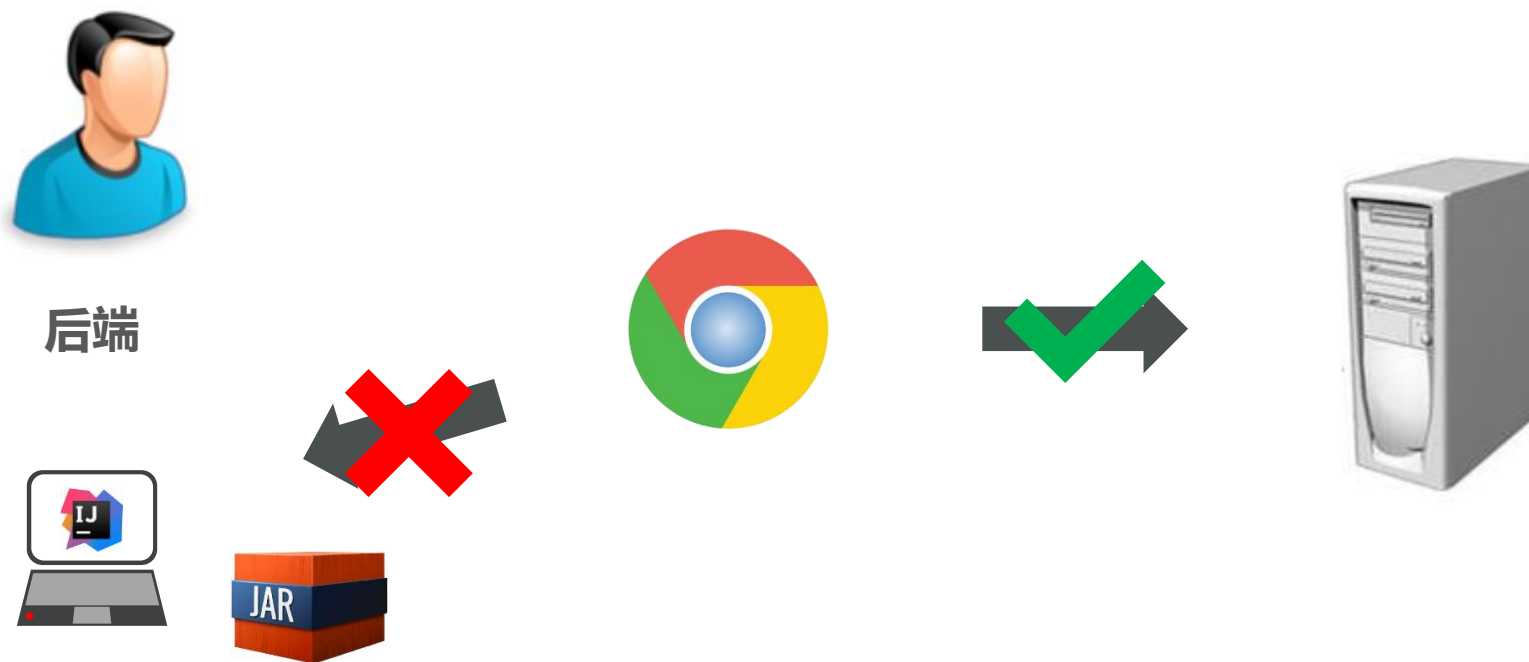
## 实用运维篇



黑马程序员  
[www.itheima.com](http://www.itheima.com)

传智教育旗下  
高端IT教育品牌

## SpringBoot项目快速启动





## SpringBoot项目快速启动

①：对SpringBoot项目打包（执行Maven构建指令package）

## 步骤 SpringBoot项目快速启动

### ②：执行启动指令

```
java -jar springboot.jar
```

#### 注意事项

jar支持命令行启动需要依赖maven插件支持，请确认打包时是否具有SpringBoot对应的maven插件

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```



## 小结

1. SpringBoot工程可以基于java环境下独立运行jar文件启动服务
2. SpringBoot工程执行mvn命令package进行打包
3. 执行jar文件命令: `java -jar 工程名.jar`

## SpringBoot项目快速启动



后端



8800



痛点

停



## 临时参数设置

- 带参数启动SpringBoot

```
java -jar springboot.jar --server.port=80
```

- 携带多个参数启动SpringBoot, 参数间使用空格分隔

```
java -jar springboot.jar --server.port=80 --logging.level.root=debug
```

## 各种参数加载优先顺序

- 参看<https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-external-config>

1. Default properties (specified by setting `SpringApplication.setDefaultProperties` ).
2. `@PropertySource` annotations on your `@Configuration` classes. Please note that such property sources are not added to the `Environment` until the application context is being refreshed. This is too late to configure certain properties such as `logging.*` and `spring.main.*` which are read before refresh begins.
3. Config data (such as `application.properties` files)
4. A `RandomValuePropertySource` that has properties only in `random.*` .
5. OS environment variables.
6. Java System properties ( `System.getProperties()` ).
7. JNDI attributes from `java:comp/env` .
8. `ServletContext` init parameters.
9. `ServletConfig` init parameters.
10. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
11. Command line arguments.
12. `properties` attribute on your tests. Available on `@SpringBootTest` and the [test annotations for testing a particular slice of your application](#).
13. `@TestPropertySource` annotations on your tests.
14. [Devtools global settings properties](#) in the `$HOME/.config/spring-boot` directory when devtools is active.

低



高



## 小结

1. 使用jar命令启动SpringBoot工程时可以使用临时参数设定替换配置文件中的参数
2. 临时参数添加方式: `java -jar 工程名.jar --属性名=值`
3. 多个临时参数之间使用空格分隔
4. 临时参数必须是当前boot工程支持的属性, 否则设置无效

## 临时参数设置



后端

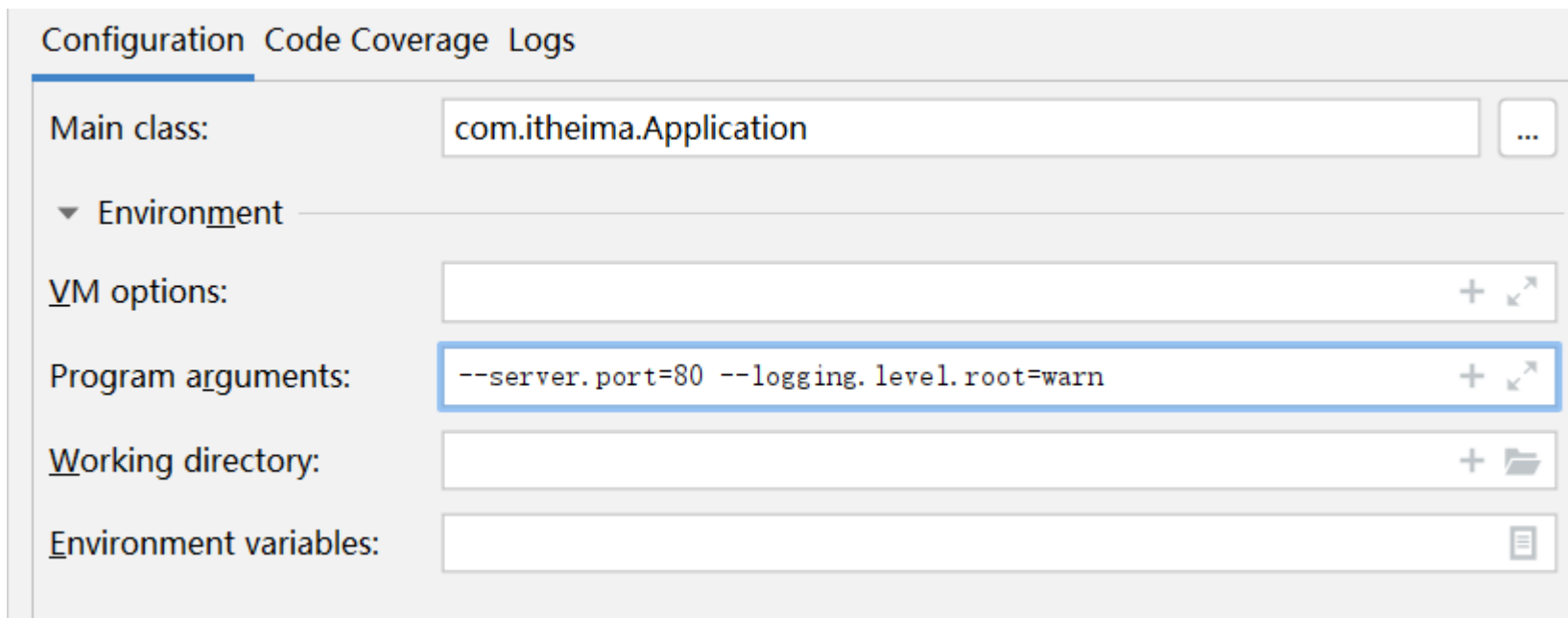


痛点

停

## 临时参数设置（开发环境）

- 带参数启动SpringBoot程序，为程序添加运行参数



Configuration Code Coverage Logs

Main class:  ...

▼ Environment

VM options:  + ↗

Program arguments:  + ↗

Working directory:  + 📁

Environment variables:  📋



## 小结

1. Idea环境下通过运行程序启动参数设置临时参数
2. 位置: Configuration → Program arguments
3. 临时参数添加格式: `--属性名=值 --属性名=值 --属性名=值`
4. 多个临时参数之间使用空格分隔

使用命令行方式展  
示，数量很多，提  
出矛盾



停



## 小结

1. xxxxxxxxxxxxxxxxx

2.临时修改属性

--属性名=值

3.idea环境下如何配置临时属性  
界面

4.临时属性过多，需要修改的内容过多，如何解决  
文件封装，开发级（config/文件）

## 配置文件分类



测试



后端

```
java -jar springboot.jar --spring.profiles.active=test --server.port=85 --server.servlet.context-path=/heima --server.tomcat.connection-timeout=-1 ..... .....
```

## 配置文件分类

- SpringBoot中4级配置文件

1级: file : config/application.yml

2级: file : application.yml

3级: classpath: config/application.yml

4级: classpath: application.yml **【最低】**

**【最高】**

- 作用:

- ◆ 1级与2级留做系统打包后设置通用属性
- ◆ 3级与4级用于系统开发阶段设置通用属性

一分二



## 小结

### 1. 配置文件分类 (4种)

停



## 小结

1. xxxxxxxxxxxxxxxxx

3.idea环境下如何配置临时属性  
界面

4.临时属性过多，需要修改的内容过多，如何解决  
文件封装，开发级（config/文件）

5.上线添加临时属性，或者机密属性，已经打包了，如何解决

文件封装，运维级（文件 config/文件）

停





## 小结

1. xxxxxxxxxxxxxxxx

- 4.临时属性过多，需要修改的内容过多，如何解决  
文件封装，开发级（config/文件）
- 5.上线添加临时属性，或者机密属性，已经打包了，如何解决  
文件封装，运维级（文件 config/文件）
- 6.多种配置文件间的优先顺序，是否干扰此项，还是可以作为全新的补充方案
  - 配置文件间属性互补（需要验证不同类型的配置文件间是否属性互补，或者没有必要）
  - properties与yml共存情况下优先级问题（需要验证）

停

## 小结

1. xxxxxxxxxxxxxxxx

5. 上线添加临时属性，或者机密属性，已经打包了，如何解决

文件封装，运维级（文件 config/文件）

6. 多种配置文件间的优先顺序，是否干扰此项，还是可以作为全新的补充方案

- 配置文件间属性互补（需要验证不同类型的配置文件间是否属性互补，或者没有必要）

- properties与yml共存情况下优先级问题（需要验证）

7. 调整信息备份/配置信息多种

- 自定义配置文件（此处要研发）

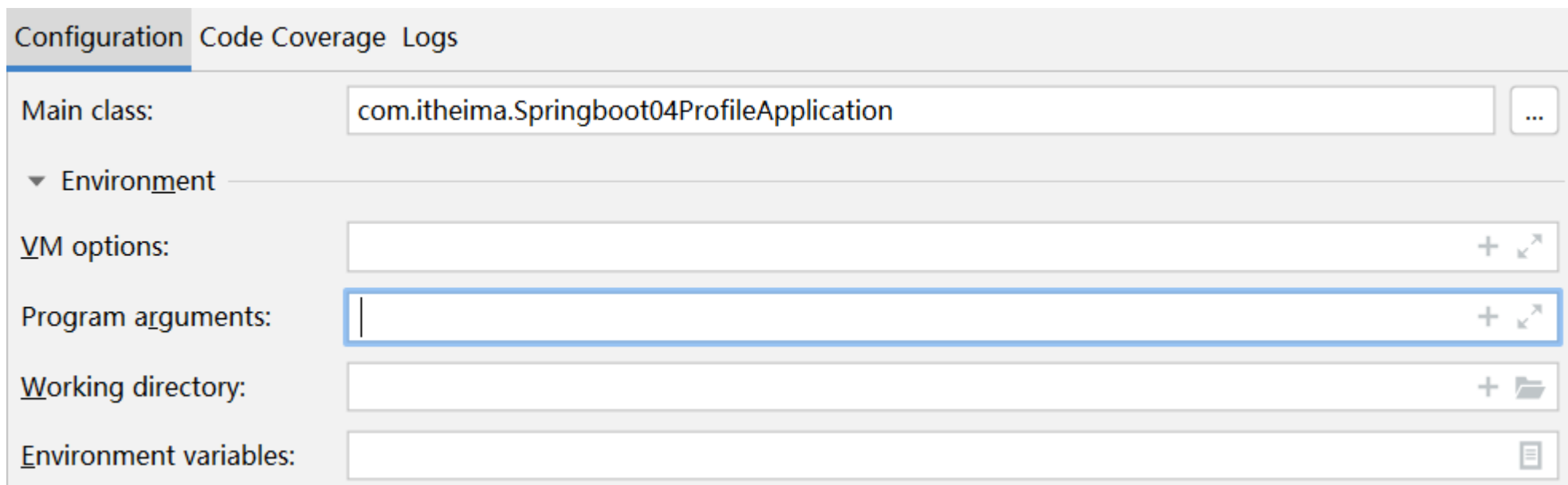
- ##### 参看：<https://www.bilibili.com/video/BV17z4y1X7is?p=14>

- 修改文件名：默认名称application，修改配置文件名：启动命令中设置Program arguments: --spring.config.name=

文件名（不带.properties）

## 配置文件自定义名称（了解）

- SpringBoot支持自定义配置文件名，可以通过参数设置配置文件名，启动命令中设置Program arguments



Configuration Code Coverage Logs

Main class:

▼ Environment

VM options:

Program arguments:

Working directory:

Environment variables:

- ◆ 指定文件名（扩展名默认properties/yml/yaml）：--spring.config.name=自定义文件名
  - ◆ 指定文件绝对路径（系统路径下查找）：--spring.config.location=D:/自定义文件名.yml
  - ◆ 指定文件相对路径（类路径下查找）：--spring.config.location=classpath:/a.yml,b.yml
- 作用：动态设置启动配置文件

停

## 小结

1. xxxxxxxxxxxxxxxx

6. 多种配置文件间的优先顺序，是否干扰此项，还是可以作为全新的补充方案

- 配置文件间属性互补（需要验证不同类型的配置文件间是否属性互补，或者没有必要）
- properties与yml共存情况下优先级问题（需要验证）

7. 调整信息备份/配置信息多种

- 自定义配置文件（此处要研发）
- ##### 参看：<https://www.bilibili.com/video/BV17z4y1X7is?p=14>

- 修改文件名：默认名称application，修改配置文件名：启动命令中设置Program arguments: --spring.config.name=

自定义文件名（不带.properties）

修改配置文件路径，启动命令中设置Program arguments: --spring.config.location=D:/自定义文件名.properties（带

停

## 1.boot工程快速启动

打包后, java -jar

## 2.临时修改属性

--属性名=值

## 3.idea环境下如何配置临时属性

界面

## 4.临时属性过多, 需要修改的内容过多, 如何解决

文件封装, 开发级 (config/文件)

## 5.上线添加临时属性, 或者机密属性, 已经打包了, 如何解决

文件封装, 运维级 (文件 config/文件)

## 6.多种配置文件间的优先顺序, 是否干扰此项, 还是可以作为全新的补充方案

- 配置文件间属性互补 (需要验证不同类型的配置文件间是否属性互补, 或者没有必要)

- properties与yml共存情况下优先级问题 (需要验证)

## 7.调整信息备份/配置信息多种

- 自定义配置文件 (此处要研发)

- ##### 参看: <https://www.bilibili.com/video/BV17z4y1X7is?p=14>

- 修改文件名: 默认名称application, 修改配置文件名: 启动命令中设置Program arguments: --spring.config.name=

自定义文件名 (不带.properties)

- 修改配置文件路径: 启动命令中设置Program arguments: --spring.config.location=D:/自定义文件名.properties

(带后缀)

=classpath:/a.properties,b.properties

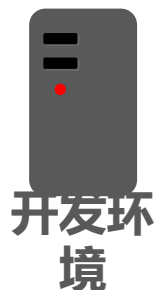
## 8.后期使用springcloud后, 你进行服务器开发会开启非常多的服务器, 这个时候不管你的配置文件放到哪里, 总之不会出现在你的classpath下, 因此后期会放通过springcloud的配置中心进行设定, 动态的加载配置信息, 使用路径来加载配置文件



## 多环境启动



```
jdbc:  
url: jdbc:mysql://21.49.35.241:3306/ccb  
user: ccb_admin  
password: 8Fm#_!@aSdm93]4k
```



```
jdbc:  
url: jdbc:mysql://127.0.0.1:4092/ccb_svms  
user: root  
password: root
```



```
jdbc:  
url: jdbc:mysql://21.49.27.66:4092/ccb_svms  
user: ccb_admin_test  
password: noBUGnoBugnoBUG
```

## 多环境启动

```
spring:  
  profiles:  
    active: pro
```

启动指定环境

```
spring:  
  profiles: pro
```

设置生产环境

```
server:  
  port: 80
```

生产环境具体参数设定

```
spring:  
  profiles: dev
```

设置开发环境

```
server:  
  port: 81
```

开发环境具体参数设定

```
spring:  
  profiles: test
```

设置测试环境

```
server:  
  port: 82
```

测试环境具体参数设定

## 多环境启动

```
spring:
  profiles:
    active: pro
```

---

```
server:
  port: 80
```

```
spring:
  profiles: pro
```

过时格式

```
spring:
  profiles:
    active: pro
```

---

```
server:
  port: 80
```

```
spring:
  config:
    activate:
      on-profile: pro
```

推荐格式

## 多环境独立文件配置

- 主启动配置文件application.yml

```
spring:
  profiles:
    active: dev
```

- 环境分类配置文件application-**pro**.yml

```
server:
  port: 80
```

- 环境分类配置文件application-**dev**.yml

```
server:
  port: 81
```

- 环境分类配置文件application-**test**.yml

```
server:
  port: 82
```

## properties文件多环境启动

- 主启动配置文件application.properties

```
spring.profiles.active=pro
```

- 环境分类配置文件application-**pro**.properties

```
server.port=80
```

- 环境分类配置文件application-**dev**.properties

```
server.port=81
```

- 环境分类配置文件application-**test**.properties

```
server.port=82
```

## 基础配置

- 多环境配置
  - 多环境启动配置（properties版）
  - 多环境启动配置（yaml版）
  - 多环境启动配置（yaml版）单一文件版
  - 多环境启动命令
  - 注意：多环境配置不会在单个程序中进行配置，在微服务架构中通过配置中心进行多环境开发配置管理
- Maven与SpringBoot多环境配置问题

## 小结

1. 多环境启动配置（yaml版）
2. 多环境启动配置（properties版）



## 小结

### 1. 多环境启动命令格式

## 多环境开发控制



profile



profile





## Maven与SpringBoot多环境兼容

### ①: Maven中设置多环境属性

```
<profiles>
  <profile>
    <id>dev_env</id>
    <properties>
      <profile.active>dev</profile.active>
    </properties>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
  </profile>
  <profile>
    <id>pro_env</id>
    <properties>
      <profile.active>pro</profile.active>
    </properties>
  </profile>
  <profile>
    <id>test_env</id>
    <properties>
      <profile.active>test</profile.active>
    </properties>
  </profile>
</profiles>
```

## 步骤 Maven与SpringBoot多环境兼容

### ②: SpringBoot中引用Maven属性

```
spring:
  profiles:
    active: ${profile.active}
```

---

```
spring:
  profiles: pro
server:
  port: 80
```

---

```
spring:
  profiles: dev
server:
  port: 81
```

---

```
spring:
  profiles: test
server:
  port: 82
```

```
<profile>
  <id>dev_env</id>
  <properties>
    <profile.active>dev</profile.active>
  </properties>
  <activation>
    <activeByDefault>true</activeByDefault>
  </activation>
</profile>
```



## Maven与SpringBoot多环境兼容

③：执行Maven打包指令

## 多环境开发控制

- Maven指令执行完毕后，生成了对应的包，其中类参与编译，但是配置文件并没有编译，而是复制到包中

```
spring:
  profiles:
    active: ${profile.active}
```

复制前 → 复制后

```
spring:
  profiles:
    active: ${profile.active}
```

- 解决思路：对于源码中非java类的操作要求加载Maven对应的属性，解析\${}占位符



## Maven与SpringBoot多环境兼容

④：对资源文件开启对默认占位符的解析

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-resources-plugin</artifactId>
      <configuration>
        <encoding>utf-8</encoding>
        <useDefaultDelimiters>true</useDefaultDelimiters>
      </configuration>
    </plugin>
  </plugins>
</build>
```

## 多环境开发控制

- Maven打包加载到属性，打包顺利通过

```
spring:
  profiles:
    active: ${profile.active}
```

复制前 → 复制后

```
spring:
  profiles:
    active: dev
```



## 小结

1. Maven与SpringBoot关联操作（多环境配置）



## 总结

1. 配置文件格式
2. yaml数据格式与数据读取方式
3. 多环境启动
4. 配置文件分类



## 自定义对象封装数据警告解决方案

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-configuration-processor</artifactId>  
  <optional>true</optional>  
</dependency>
```

## 整合MyBatis

```
<configuration>
  <properties resource="jdbc.properties"></properties>
  <typeAliases>
    <package name="com.itheima.domain"/>
  </typeAliases>
  <environments default="mysql">
    <environment id="mysql">
      <transactionManager type="JDBC"></transactionManager>
      <dataSource type="POOLED">
        <property name="driver" value="${jdbc.driver}"></property>
        <property name="url" value="${jdbc.url}"></property>
        <property name="username" value="${jdbc.username}"></property>
        <property name="password" value="${jdbc.password}"></property>
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <package name="com.itheima.dao"></package>
  </mappers>
</configuration>
```

丢失的配置

丢失的配置

## 整合MyBatis

```
<configuration>
  <properties resource="jdbc.properties"></properties>
  <typeAliases>
    <package name="com.itheima.domain"/>
  </typeAliases>
  <environments default="mysql">
    <environment id="mysql">
      <transactionManager type="JDBC"></transactionManager>
      <dataSource type="POOLED">
        <property name="driver" value="${jdbc.driver}"></property>
        <property name="url" value="${jdbc.url}"></property>
        <property name="username" value="${jdbc.username}"></property>
        <property name="password" value="${jdbc.password}"></property>
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <package name="com.itheima.dao">
  </mappers>
</configuration>
```

spring:

datasource:

driver-class-name: com.mysql.cj.jdbc.Driver

url: jdbc:mysql://localhost:3306/ssm\_db?serverTimezone=UTC

username: root

password: root

application.yml

## 整合MyBatis

```
<configuration>
  <properties resource="jdbc.properties"></properties>
  <typeAliases>
    <package name="com.itheima.domain"/>
  </typeAliases>
  <environments default="mysql">
    <environment id="mysql">
      <transactionManager type="JDBC"></transactionManager>
      <dataSource type="POOLED">
        <property name="driver" value="${jdbc.driver}"></property>
        <property name="url" value="${jdbc.url}"></property>
        <property name="username" value="${jdbc.username}"></property>
        <property name="password" value="${jdbc.password}"></property>
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <package name="com.itheima.dao"></package>
  </mappers>
</configuration>
```

@Mapper

```
public interface UserDao {
    @Select("select * from user")
    public List<User> getAll();
}
```



## 小结

1. SpringBoot整合MyBatis时将原始配置中的数据库连接配置转换成了SpringBoot中的数据源配置
2. SpringBoot整合MyBatis时将原始配置中的映射扫描配置转换成了由SpringBoot读取注解@Mapper加载对应信息
3. SpringBoot整合MyBatis时将原始配置中的其他配置转换成了默认配置信息

停