

# SpringBoot基础篇

---

在基础篇中，我给学习者的定位是先上手，能够使用SpringBoot搭建基于SpringBoot的web项目开发，所以内容设置较少，主要包含如下内容：

- SpringBoot快速入门
- SpringBoot基础配置
- 基于SpringBoot整合SSMP

## JC-1.快速上手SpringBoot

---

学习任意一项技术，首先要知道这个技术的作用是什么，不然学完以后，你都不知道什么时候使用这个技术，也就是技术对应的应用场景。SpringBoot技术由Pivotal团队研发制作，功能的话简单概括就是加速Spring程序的开发，这个加速要从如下两个方面来说

- Spring程序初始搭建过程
- Spring程序的开发过程

通过上面两个方面的定位，我们可以产生两个模糊的概念：

1. SpringBoot开发团队认为原始的Spring程序初始搭建的时候可能有些繁琐，这个过程是可以简化的，那原始的Spring程序初始搭建过程都包含哪些东西了呢？为什么觉得繁琐呢？最基本的Spring程序至少有一个配置文件或配置类，用来描述Spring的配置信息，莫非这个文件都可以不写？此外现在企业级开发使用Spring大部分情况下是做web开发，如果做web开发的话，还要在加载web环境时加载时加载指定的spring配置，这都是最基本的需求了，不写的话怎么知道加载哪个配置文件/配置类呢？那换了SpringBoot技术以后呢，这些还要写吗？谜底稍后揭晓，先卖个关子
2. SpringBoot开发团队认为原始的Spring程序开发的过程也有些繁琐，这个过程仍然可以简化。开发过程无外乎使用什么技术，导入对应的jar包（或坐标）然后将这个技术的核心对象交给Spring容器管理，也就是配置成Spring容器管控的bean就可以了。这都是基本操作啊，难道这些东西SpringBoot也能帮我们简化？

带着上面这些疑问我们就着手第一个SpringBoot程序的开发了，看看到底使用SpringBoot技术能简化开发到什么程度。

### 温馨提示

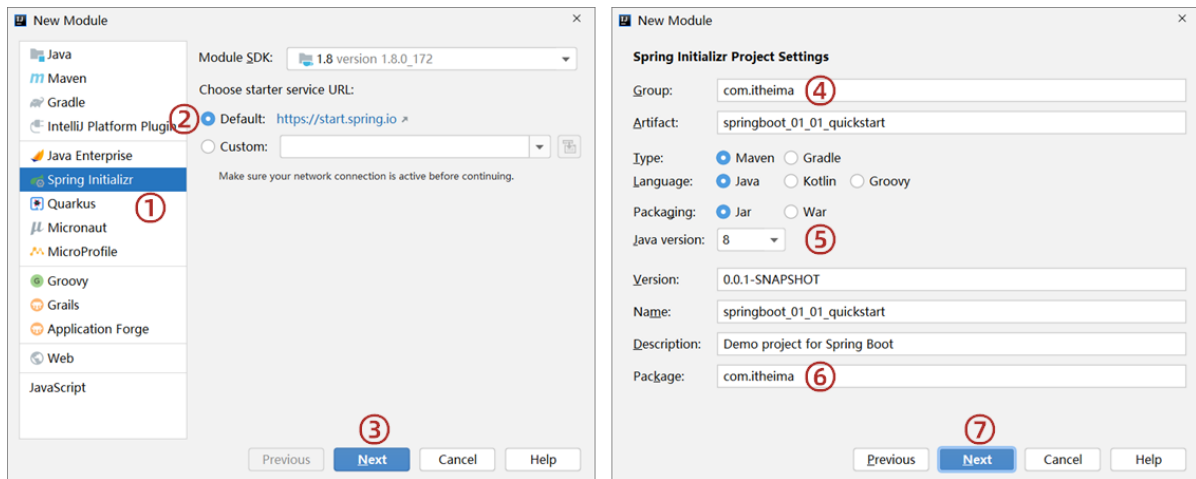
如果对Spring程序的基础开发不太懂的小伙伴，看到这里可以弃坑了，下面的内容学习需要具备Spring技术的知识，硬着头皮学不下去的。

## JC-1-1.SpringBoot入门程序制作（一）

下面让我们开始做第一个SpringBoot程序吧，本课程基于Idea2020.3版本制作，使用的Maven版本为3.6.1，JDK版本为1.8。如果你的环境和上述环境不同，可能在操作界面和操作过程中略有不同，只要软件匹配兼容即可（说到这个Idea和Maven，它们两个还真不是什么版本都能搭到一起的，说多了都是泪啊）。

下面使用SpringBoot技术快速构建一个SpringMVC的程序，通过这个过程体会**简化**二字的含义

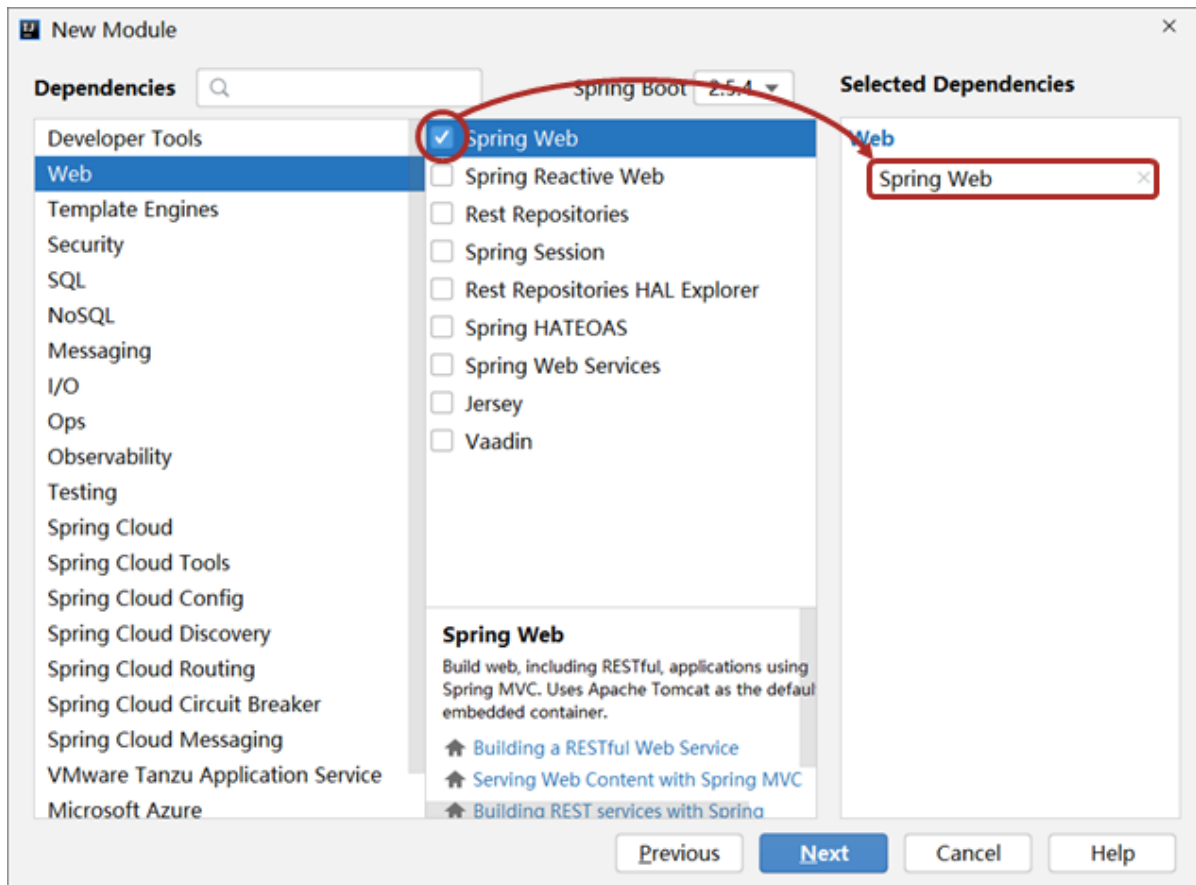
**步骤①：**创建新模块，选择Spring Initializr，并配置模块相关基础信息



**特别关注：**第3步点击Next时，Idea需要联网状态才可以进入到后面那一页，如果不能正常联网，就无法正确到达右面那个设置页了，会一直联网转转转

**特别关注：**第5步选择java版本和你计算机上安装的JDK版本匹配即可，但是最低要求为JDK8或以上版本，推荐使用8或11

**步骤②：**选择当前模块需要使用的技术集



按照要求，左侧选择web，然后在中间选择Spring Web即可，选完右侧就出现了新的内容项，这就表示勾选成功了

**关注：**此处选择的SpringBoot的版本使用默认的就可以了，需要说一点，SpringBoot的版本升级速度很快，可能昨天创建工程的时候默认版本是2.5.4，今天再创建工程默认版本就变成2.5.5了，差别不大，无需过于纠结，回头可以到配置文件中修改对应的版本

**步骤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...";
    }
}
```

入门案例制作的SpringMVC的控制器基于Rest风格开发，当然此处使用原始格式制作SpringMVC的程序也是没有问题的，上例中的@RestController与@GetMapping注解是基于Restful开发的典型注解

**关注：**做到这里SpringBoot程序的最基础的开发已经做完了，现在就可以正常的运行Spring程序了。可能有些小伙伴会有疑惑，Tomcat服务器没有配置，=Spring也没有配置，什么都没有配置这就能用吗？这就是SpringBoot技术的强大之处。关于内部工作流程后面再说，先专心学习开发过程

#### 步骤④：运行自动生成的Application类



```

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

```

使用带main方法的java程序的运行形式来运行程序，运行完毕后，控制台输出上述信息。

不难看出，运行的信息中包含了8080的端口，Tomcat这种熟悉的字样，难道这里启动了Tomcat服务器？是的，这里已经启动了。那服务器没有配置，哪里来的呢？后面再说。现在你可以通过浏览器访问请求的路径，测试功能是否工作正常了

访问路径：<http://localhost:8080/books>

是不是感觉很神奇？目前的效果其实依赖的底层逻辑还是很复杂的，但是从开发者角度来看，目前只有两个文件展现到了开发者面前

- pom.xml

这是maven的配置文件，描述了当前工程构建时相应的配置信息

```
<?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
  https://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.example</groupId>
<artifactId>springboot_01_01_quickstart</artifactId>
<version>0.0.1-SNAPSHOT</version>

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

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

配置中有两个信息需要关注，一个是parent，也就是当前工程继承了另外一个工程，干什么用的后面再说，还有依赖坐标，干什么用的后面再说

- Application类

```

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

```

这个类功能很简单，就一句代码，前面运行程序就是运行的这个类

到这里我们可以大胆推测一下，如果上面这两个文件没有的话，SpringBoot肯定没法玩，看来核心就是这两个文件了。由于是制作第一个SpringBoot程序，先不要关注这两个文件的功能，后面详细讲解内部工作流程。

通过上面的制作，我们不难发现，SpringBoot程序简直太好写了，几乎什么都没写，功能就有了，这也是SpringBoot技术为什么现在这么火的原因，和Spring程序相比，SpringBoot程序在开发的过程中各个层面均具有优势

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

一句话总结一下就是**能少写就少写，能不写就不写**，这就是SpringBoot技术给我们带来的好处，行了，现在你就可以动手做一做SpringBoot程序了，看看效果如何，是否真的帮助你简化开发了

## 总结

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以上版本进行制作
3. SpringBoot程序中需要使用何种功能通过勾选选择技术，也可以手工添加对应的要使用的技术（后期讲解）
4. 运行SpringBoot程序通过运行Application程序入口进行

## 思考

前面制作的时候说过，这个过程必须联网才可以进行，但是有些时候你会遇到一些莫名其妙的问题，比如基于Idea开发时，你会发现你配置了一些坐标，然后Maven下载对应东西的时候死慢死慢的，甚至还会失败。其实这和Idea这款IDE工具有关，万一Idea不能正常访问网络的话，我们是不是就无法制作SpringBoot程序了呢？咱们下一节再说

## JC-1-2.SpringBoot入门程序制作（二）

如果Idea不能正常联网，这个SpringBoot程序就无法制作了吗？开什么玩笑，世上IDE工具千千万，难道SpringBoot技术还必须基于Idea来做了？这是不可能的。开发SpringBoot程序，可以不基于任意的IDE工具进行，其实在SpringBoot的官网里面就可以直接创建SpringBoot程序

SpringBoot官网和Spring的官网是在一起的，都是 [spring.io](http://spring.io)。你可以通过项目一级一级的找到SpringBoot技术的介绍页，然后在页面中间部位找到如下内容



### Quickstart Your Project

Bootstrap your application with [Spring Initializr](#).

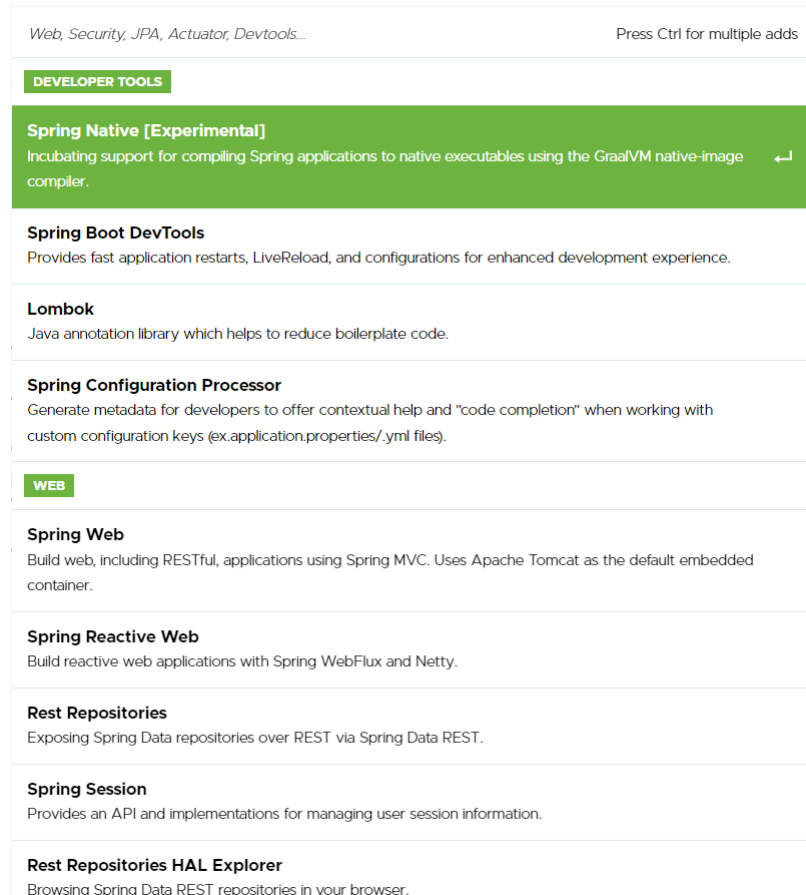
**步骤①：**点击Spring Initializr后进入到创建SpringBoot程序的界面上，下面是输入信息的过程，和前面的一样，只是界面变了而已，根据自己的要求，在左侧选择对应信息和输入对应的信息即可

The image shows the Spring Initializr web form. It is divided into several sections for configuring a new project:

- Project:** Includes radio buttons for **Maven Project** (selected) and **Gradle Project**.
- Language:** Includes radio buttons for **Java** (selected), **Kotlin**, and **Groovy**.
- Spring Boot:** Includes radio buttons for versions: 2.6.0 (SNAPSHOT), 2.6.0 (M2), 2.5.5 (SNAPSHOT), **2.5.4** (selected), 2.4.11 (SNAPSHOT), and 2.4.10.
- Project Metadata:** Includes text 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).
- Packaging:** Includes radio buttons for **Jar** (selected) and **War**.
- Java:** Includes radio buttons for versions: 16, 11, and **8** (selected).
- Dependencies:** Includes a section for **Spring Web** (WEB) with a description: "Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container." There is an **ADD DEPENDENCIES...** button with a keyboard shortcut **CTRL + B**.

At the bottom of the form, there are three buttons: **GENERATE CTRL + G**, **EXPLORE CTRL + SPACE**, and **SHARE...**

**步骤②：**右侧的ADD DEPENDENCIES用于选择使用何种技术，和之前勾选的Spring WEB是在做同一件事，仅仅是界面不同而已，点击后打开网页版的技术选择界面



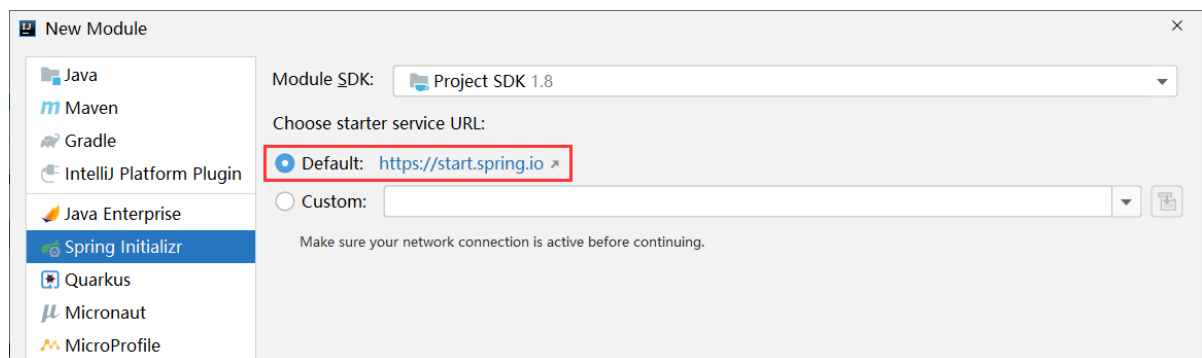
**步骤③：**所有信息设置完毕后，点击下面左侧按钮，生成一个文件包

**步骤④：**保存后得到一个压缩文件，这个文件打开后就是创建的SpringBoot工程文件夹了

**步骤⑤：**解压缩此文件后，得到工程目录，在Idea中导入即可使用，和之前创建的东西完全一样。下面就可以自己创建一个Controller测试一下是否能用了。

### 温馨提示

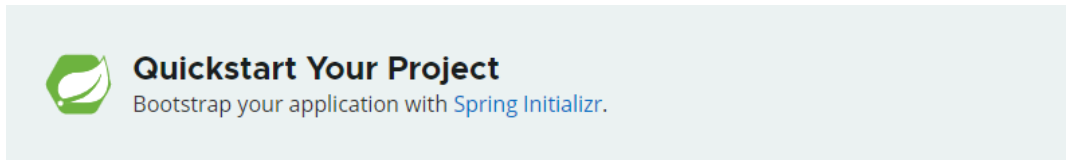
做到这里其实可以透漏一个小秘密，Idea工具中创建SpringBoot工程其实连接的就是SpringBoot的官网，走的就是这个过程，只不过Idea把界面给整合了一下，读取到了Spring官网给的信息，然后展示到了Idea的界面中而已，不信你可以看看下面这个步骤



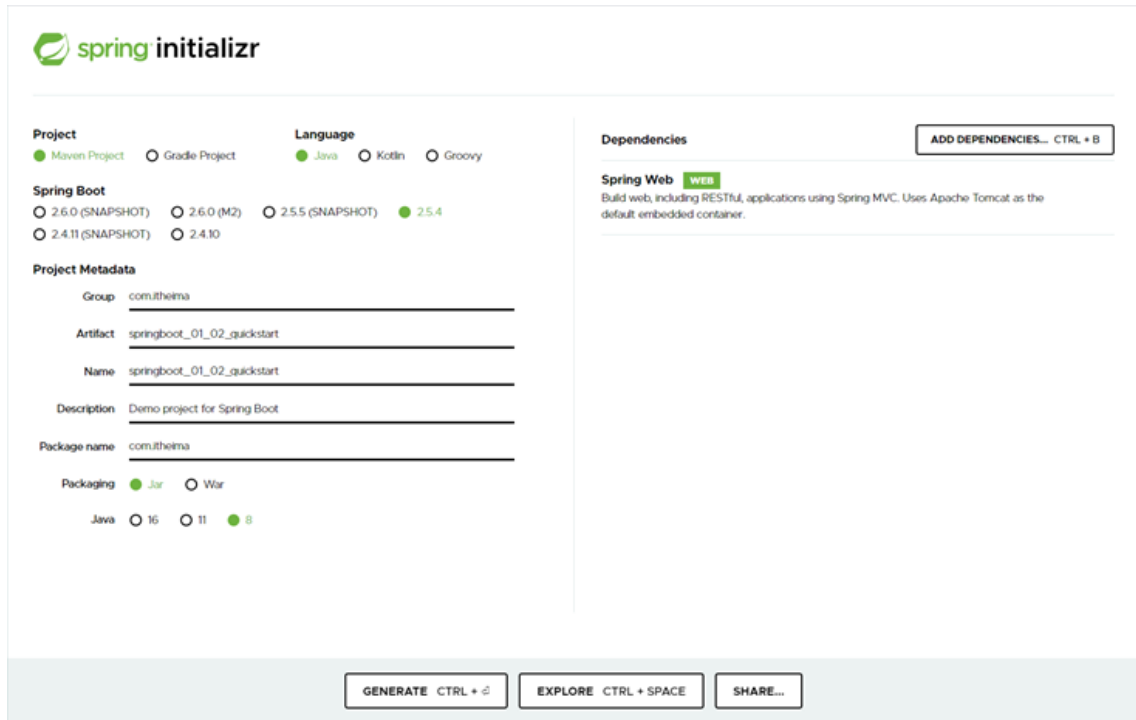
上面描述了连接的网址，再看看SpringBoot官网创建工程的URL地址，是不是一样？

### 总结

## 1. 打开SpringBoot官网，选择Quickstart Your Project



## 2. 创建工程

A screenshot of the Spring Initializr web form. The form is titled "spring initializr" and is divided into several sections. The "Project" section has radio buttons for "Maven Project" (selected) and "Gradle Project". The "Language" section has radio buttons for "Java" (selected), "Kotlin", and "Groovy". The "Spring Boot" section has radio buttons for "2.6.0 (SNAPSHOT)", "2.6.0 (M2)", "2.5.5 (SNAPSHOT)", and "2.5.4" (selected). The "Project Metadata" section has text 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 has radio buttons for "Jar" (selected) and "War". The "Java" section has radio buttons for "16", "11", and "8" (selected). The "Dependencies" section has a button "ADD DEPENDENCIES... CTRL + B" and a "Spring Web" dependency selected. At the bottom, there are three buttons: "GENERATE CTRL + G", "EXPLORE CTRL + SPACE", and "SHARE...".

并保存项目

## 3. 解压项目，通过IDE导入项目

### 思考

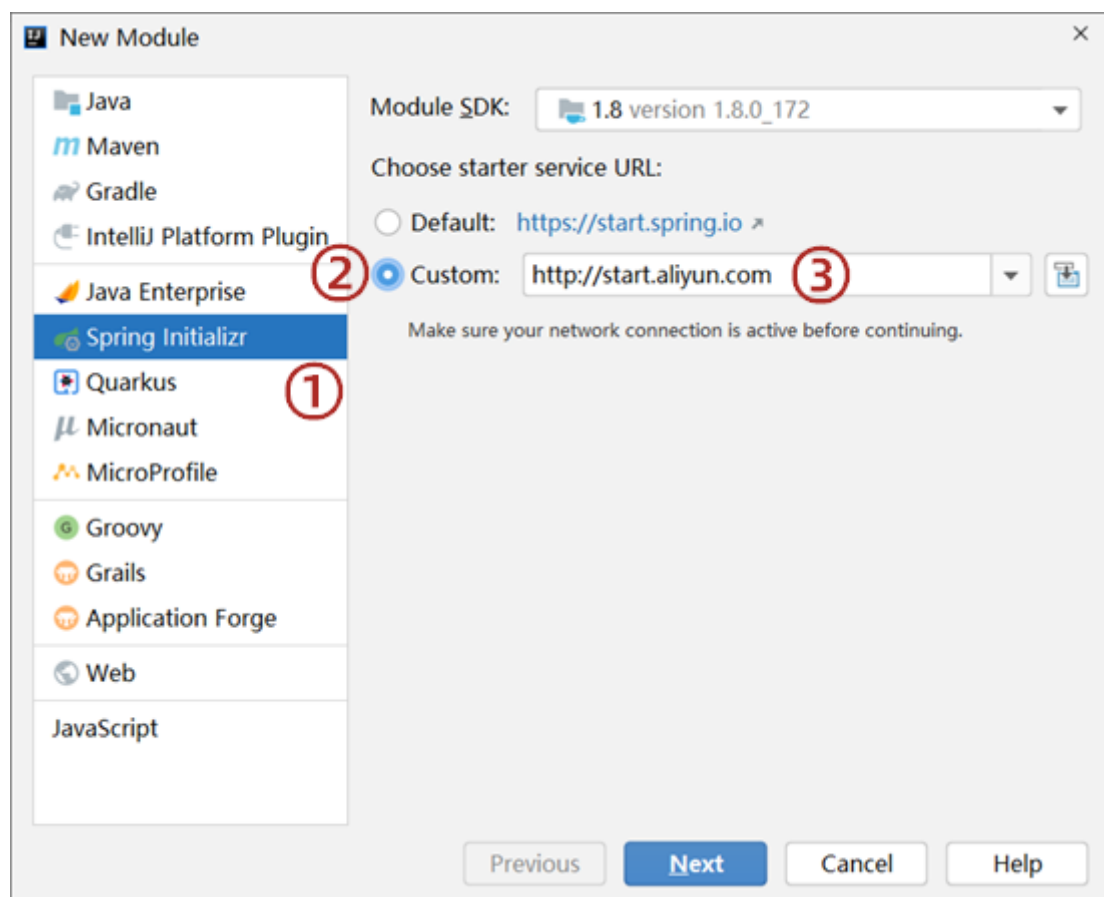
现在创建工程靠的是访问国外的Spring主站，但是互联网访问是可以控制的，如果一天这个网站你在国内都无法访问了，那前面这两种方式都无法创建SpringBoot工程了，这时候又该怎么解决这个问题呢？咱们下一节再说

## JC-1-3.SpringBoot入门程序制作（三）

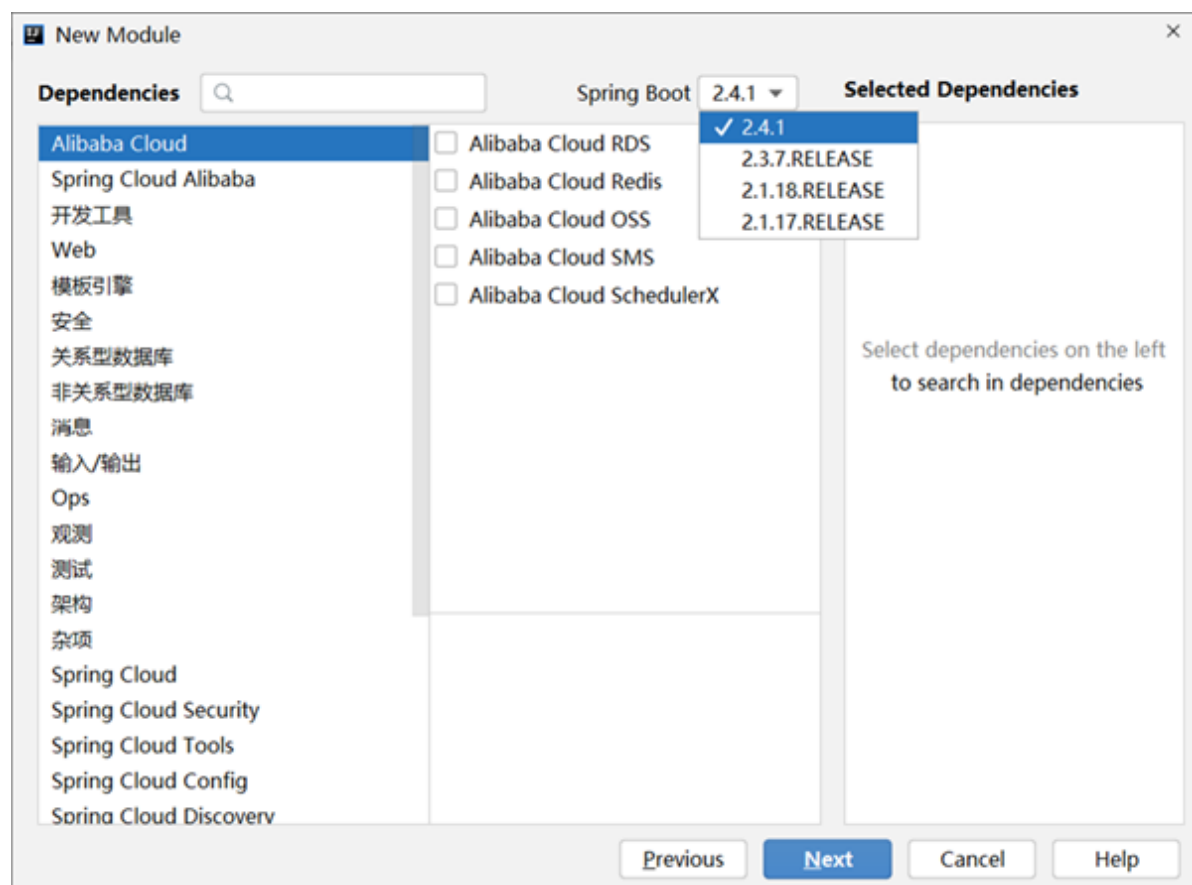
前面提到网站如果被限制访问了，该怎么办？开动脑筋想一想，不管是方式一还是方式二其实都是走的同一个地方，也就是SpringBoot的官网创建的SpringBoot工程，那如果我们国内有这么一个网站能提供这样的功能，是不是就解决了呢？必然的嘛，新的问题又来了，这个国内的网站有吗？还真有，阿里提供了一个，下面问题就简单了，网址告诉我们就OK了，没错，就是这样

创建工程时，切换选择starter服务路径，然后手工输入阿里云提供给我们的使用地址即可。地址：<http://start.aliyun.com>或<https://start.aliyun.com>





阿里为了便于自己开发使用，因此在依赖坐标中添加了一些阿里相关的技术，也是为了推广自己的技术吧，所以在依赖选择列表中，你有了更多的选择。不过有一点需要说清楚，阿里云地址默认创建的SpringBoot工程版本是**2.4.1**，所以如果你想更换其他的版本，创建项目后手工修改即可，别忘了刷新一下，加载新版本信息



阿里云提供的地址更符合国内开发者的使用习惯，里面有一些SpringBoot官网上没有给出的坐标，大家可以好好看一看。



**注意：**阿里云提供的工程创建地址初始化完毕后和实用SpringBoot官网创建出来的工程略有区别。主要是在配置文件的形式上有区别。这个信息在后面讲解Boot程序的执行流程时给大家揭晓

## 总结

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

## 思考

做到这里我们已经有了三种方式创建SpringBoot工程，但是每种方式都要求你必须能上网才能创建工程。假如有一天，你加入了一个保密级别比较高的项目组，整个项目组没有外网，整个事情是不是就不能做了呢？咱们下一节再说

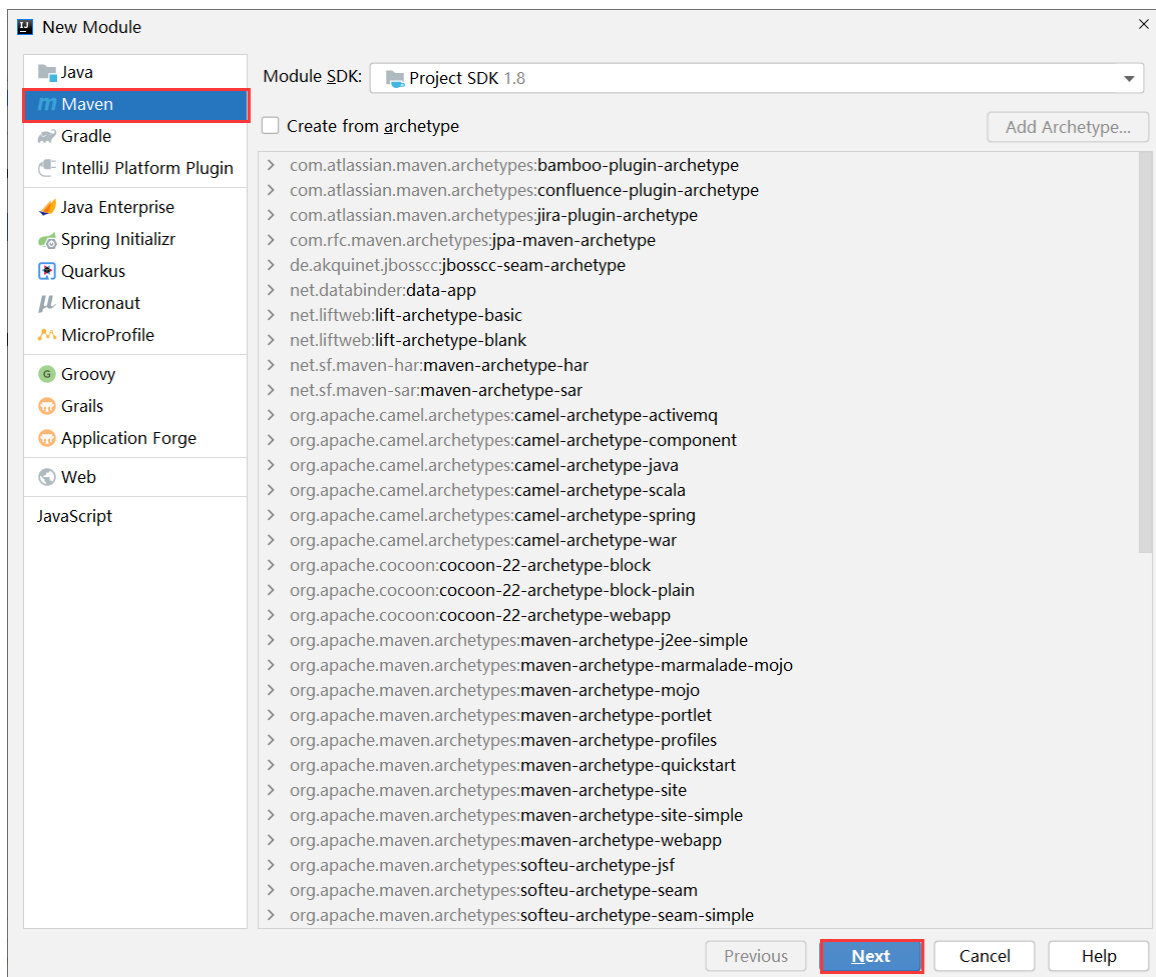
## JC-1-4.SpringBoot入门程序制作（四）

不能上网，还想创建SpringBoot工程，能不能做呢？能做，但是你要先问问自己联网和不联网到底差别是什么？这个信息找到以后，你就发现，你把联网要干的事情都提前准备好，就无需联网了。

联网做什么呢？首先SpringBoot工程也是基于Maven构建的，而Maven工程当使用了一些自己需要使用又不存在的东西时，就要去下载。其实SpringBoot工程创建的时候就是去下载一些必要的组件的。你把这些东西给提前准备好就可以了吗？是的，就是这样。

下面咱们就一起手工创建一个SpringBoot工程

**步骤①：**创建工程时，选择手工创建Maven工程



**步骤②：**参照标准SpringBoot工程的pom文件，书写自己的pom文件即可

```
<?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.example</groupId>
  <artifactId>springboot_01_04_quickstart</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>8</maven.compiler.target>
  </properties>

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

</project>
```

用什么写什么，不用的都可以不写。当然，现在小伙伴们可能还不知道用什么和不用什么，最简单的就是复制粘贴了，随着后面的学习，你就知道哪些可以省略了。此处我删减了一些目前不是必须的东西，一样能用

**步骤③：**之前运行SpringBoot工程需要一个类，这个缺不了，自己手写一个就行了，建议按照之前的目录结构来创建，先别玩花样，先学走后学跑。类名可以自定义，关联的名称一切修改即可

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

**关注：**类上面的注解@SpringBootApplication千万别丢了，这个是核心，后面再介绍

**关注：**类名可以自定义，只要保障下面代码中使用的类名和你自己定义的名称一样即可，也就是run方法中的那个class对应的名称

**步骤④：**下面就可以自己创建一个Controller测试一下是否能用了，和之前没有差别了

看到这里其实应该能够想明白了，通过向导或者网站创建的SpringBoot工程其实就是帮你写了一些代码，而现在是自己手写，写的内容都一样，仅此而已。

### 温馨提示

如果你的计算机上从来没有创建成功过SpringBoot工程，自然也就没有下载过SpringBoot对应的坐标，那用手写创建的方式在不联网的情况下肯定该是不能用的。所谓手写，其实就是自己写别人帮你生成的东西，但是引用的坐标对应的资源必须保障maven仓库里面有才行，如果没有，还是要去下载的

### 总结

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

到这里其实学习了4种创建SpringBoot工程的方式，其实本质是一样的，就是根据SpringBoot工程的文件格式要求，通过不同方式生成或者手写得到对应的文件，效果完全一样。

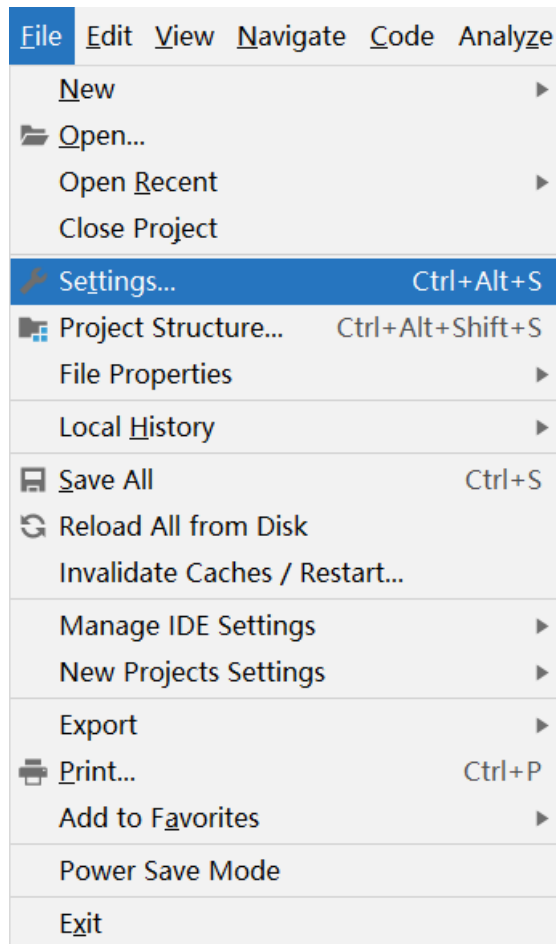
## 教你一招：在Idea中隐藏指定文件/文件夹

创建SpringBoot工程时，使用SpringBoot向导也好，阿里云也罢，其实都是为了一个目的，得到一个标准的SpringBoot工程文件结构。这个时候就有新的问题出现了，标准的工程结构中包含了一些未知的文件夹，在开发的时候看起来特别别扭，这一节就来说说这些文件怎么处理。

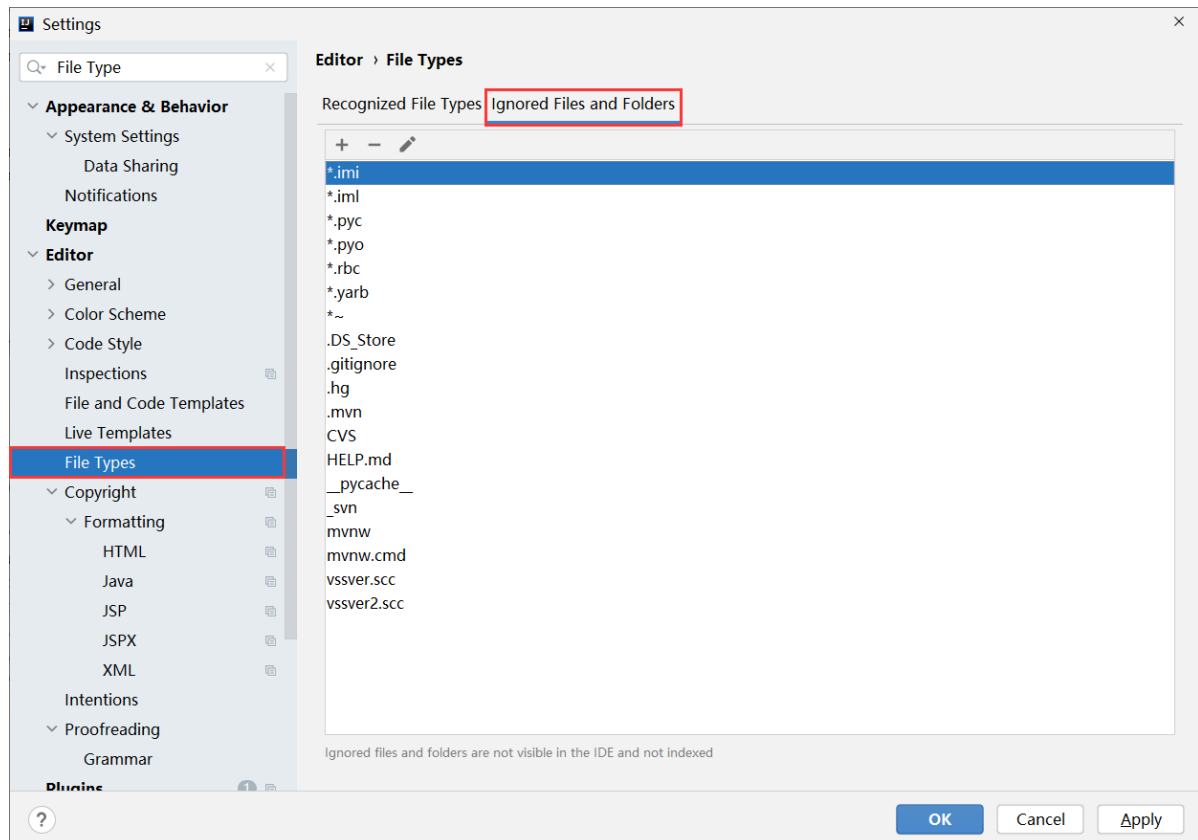
处理方案无外乎两种，如果你对每一个文件/目录足够了解，没有用的完全可以删除掉，或者不删除，但是看着别扭，就设置文件为看不到就行了。删除不说了，直接Delete掉就好了，这一节说说如何隐藏指定的文件或文件夹信息。

既然是在Idea下做隐藏功能，肯定隶属于Idea的设置，设置方式如下。

**步骤①：**打开设置，【Files】→【Settings】



步骤②：打开文件类型设置界面，【Editor】→【File Types】→【Ignored Files and Folders】，忽略文件或文件夹显示



步骤③：添加你要隐藏的文件名称或文件夹名称，可以使用\*号通配符，表示任意，设置完毕即可

到这里就做完了，其实就是Idea的一个小功能

总结

## 1. Idea中隐藏指定文件或指定类型文件

1. 【Files】→【Settings】
2. 【Editor】→【File Types】→【Ignored Files and Folders】
3. 输入要隐藏的名称，支持\*号通配符
4. 回车确认添加

## JC-1-5.SpringBoot简介

入门案例做完了，这个时候回忆一下咱们之前说的SpringBoot的功能是什么还记得吗？加速Spring程序的开发，现在是否深有体会？再来看SpringBoot技术的设计初衷就很容易理解了。

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**。

都简化了了哪些东西呢？其实就是针对原始的Spring程序制作的两个方面进行了简化：

- Spring程序缺点
  - 依赖设置繁琐
    - 以前写Spring程序，使用的技术都要自己一个一个的写，现在不需要了，如果做过原始SpringMVC程序的小伙伴应该知道，写SpringMVC程序，最基础的spring-web和spring-webmvc这两个坐标时必须的，就这还不包含你用json啊等等这些坐标，现在呢？一个坐标搞定面
  - 配置繁琐
    - 以前写配置类或者配置文件，然后用什么东西就要自己写加载bean这些东西，现在呢？什么都没写，照样能用

### 回顾

通过上面两个方面的定位，我们可以产生两个模糊的概念：

1. SpringBoot开发团队认为原始的Spring程序初始搭建的时候可能有些繁琐，这个过程是可以简化的，那原始的Spring程序初始搭建过程都包含哪些东西了呢？为什么觉得繁琐呢？最基本的Spring程序至少有一个配置文件或配置类，用来描述Spring的配置信息，莫非这个文件都可以不写？此外现在企业级开发使用Spring大部分情况下是做web开发，如果做web开发的话，还要在加载web环境时加载时加载指定的spring配置，这都是最基本的需求了，不写的话怎么知道加载哪个配置文件/配置类呢？那换了SpringBoot技术以后呢，这些还要写吗？谜底稍后揭晓，先卖个关子
2. SpringBoot开发团队认为原始的Spring程序开发的过程也有些繁琐，这个过程仍然可以简化。开发过程无外乎使用什么技术，导入对应的jar包（或坐标）然后将这个技术的核心对象交给Spring容器管理，也就是配置成Spring容器管控的bean就可以了。这都是基本操作啊，难道这些东西SpringBoot也能帮我们简化？

再来看看前面提出的两个问题，已经有答案了，都简化了，都不用写了，这就是SpringBoot给我们带来的好处。这些简化操作在SpringBoot中有专业的用语，也是SpringBoot程序的核心功能及优点：

- 起步依赖（简化依赖配置）
  - 依赖配置的书写简化就是靠这个起步依赖达成的
- 自动配置（简化常用工程相关配置）
  - 配置过于繁琐，使用自动配置就可以做响应的简化，但是内部还是很复杂的，后面具体展开说

- 辅助功能（内置服务器，.....）
  - 除了上面的功能，其实SpringBoot程序还有其他的一些优势，比如我们没有配置Tomcat服务器，但是能正常运行，这是SpringBoot程序的一个可以感知到的功能，也是SpringBoot的辅助功能之一。一个辅助功能都能做的这么6，太牛了

下面结合入门程序来说说这些简化操作都在哪些方面进行体现的，一共分为4个方面

- parent
- starter
- 引导类
- 内嵌tomcat

## parent

SpringBoot关注到开发者在进行开发时，往往对依赖版本的选择具有固定的搭配格式，并且这些依赖版本的选择还不能乱搭配。比如A技术的2.0版与B技术的3.5版可以合作在一起，但是和B技术的3.7版合并使用时就有冲突。其实很多开发者都一直想做一件事情，就是将各种各样的技术配合使用的常见依赖版本进行收集整理，制作出了最合理的依赖版本配置方案，这样使用起来就方便多了。

SpringBoot一看这种情况so easy啊，于是将所有的技术版本的常见使用方案都给开发者整理了出来，以后开发者使用时直接用它提供的版本方案，就不用担心冲突问题了，相当于SpringBoot做了无数个技术版本搭配列表，这个技术搭配列表的名字叫做**parent**。

**parent**自身具有很多个版本，每个**parent**版本中包含有几百个其他技术的版本号，不同的parent间使用的各种技术的版本号有可能会发生变化。当开发者使用某些技术时，直接使用SpringBoot提供的**parent**就行了，由**parent**帮助开发者统一的进行各种技术的版本管理

比如你现在要使用Spring配合MyBatis开发，没有parent之前怎么做呢？选个Spring的版本，再选个MyBatis的版本，再把这些技术使用时关联的其他技术的版本逐一确定下来。当你Spring的版本发生变化需要切换时，你的MyBatis版本有可能也要跟着切换，关联技术呢？可能都要切换，而且切换后还可能出现问题。现在这一切工作都可以交给parent来做了。你无需关注这些技术间的版本冲突问题，你只需要关注你用什么技术就行了，冲突问题由**parent**负责处理。

有人可能会提出来，万一**parent**给我导入了一些我不想使用的依赖怎么办？记清楚，这一点很关键，**parent**仅仅帮我们进行版本管理，它不负责帮你导入坐标，说白了用什么还是你自己定，只不过版本不需要你管理了。整体上来说，**使用parent可以帮助开发者进行版本的统一管理**

**关注：**parent定义出来以后，并不是直接使用的，仅仅给了开发者一个说明书，但是并没有实际使用，这个一定要确认清楚

那SpringBoot又是如何做到这一点的呢？可以查阅SpringBoot的配置源码，看到这些定义

- 项目中的pom.xml中继承了一个坐标

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

- 打开后可以查阅到其中又继承了一个坐标

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-dependencies</artifactId>
  <version>2.5.4</version>
</parent>
```

- 这个坐标中定义了两组信息，第一组是各式各样的依赖版本号属性，下面列出依赖版本属性的局部，可以看的出来，定义了若干个技术的依赖版本号

```
<properties>
  <activemq.version>5.16.3</activemq.version>
  <aspectj.version>1.9.7</aspectj.version>
  <assertj.version>3.19.0</assertj.version>
  <commons-codec.version>1.15</commons-codec.version>
  <commons-dbc2.version>2.8.0</commons-dbc2.version>
  <commons-lang3.version>3.12.0</commons-lang3.version>
  <commons-pool.version>1.6</commons-pool.version>
  <commons-pool2.version>2.9.0</commons-pool2.version>
  <h2.version>1.4.200</h2.version>
  <hibernate.version>5.4.32.Final</hibernate.version>
  <hibernate-validator.version>6.2.0.Final</hibernate-validator.version>
  <httpclient.version>4.5.13</httpclient.version>
  <jackson-bom.version>2.12.4</jackson-bom.version>
  <javax-jms.version>2.0.1</javax-jms.version>
  <javax-json.version>1.1.4</javax-json.version>
  <javax-websocket.version>1.1</javax-websocket.version>
  <jetty-e1.version>9.0.48</jetty-e1.version>
  <junit.version>4.13.2</junit.version>
</properties>
```

第二组是各式各样的的依赖坐标信息，可以看出依赖坐标定义中没有具体的依赖版本号，而是引用了第一组信息中定义的依赖版本属性值

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>${hibernate.version}</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

**关注：**上面的依赖坐标定义是出现在标签中的，其实是对引用坐标的依赖管理，并不是实际使用的坐标。因此当你的项目中继承了这组parent信息后，在不使用对应坐标的情况下，前面的这组定义是不会具体导入某个依赖的



**关注：**因为在maven中继承机会只有一次，上述继承的格式还可以切换成导入的形式进行，并且在阿里云的starter创建工程时就使用了此种形式

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

## 总结

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

## 思考

parent中定义了若干个依赖版本管理，但是也没有使用，那这个设定也就不生效啊，究竟谁在使用这些定义呢？

## starter

SpringBoot关注到开发者在实际开发时，对于依赖坐标的使用往往都有一些固定的组合方式，比如使用spring-webmvc就一定要使用spring-web。每次都要固定搭配着写，非常繁琐，而且格式固定，没有任何技术含量。

SpringBoot一看这种情况，看来需要给开发者带来一些帮助了。安排，把所有的技术使用的固定搭配格式都给开发出来，以后你用某个技术，就不用一次写一堆依赖了，还容易写错，我给你做一个东西，代表一堆东西，开发者使用的时候，直接用我做好的这个东西就好了，对于这样的固定技术搭配，SpringBoot给它起了个名字叫做**starter**。

starter定义了使用某种技术时对于依赖的固定搭配格式，也是一种最佳解决方案，**使用starter可以帮助开发者减少依赖配置**

这个东西其实在入门案例里面已经使用过了，入门案例中的web功能就是使用这种方式添加依赖的。可以查阅SpringBoot的配置源码，看到这些定义

- 项目中的pom.xml定义了使用SpringMVC技术，但是并没有写SpringMVC的坐标，而是添加了一个名字中包含starter的依赖

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

- 在spring-boot-starter-web中又定义了若干个具体依赖的坐标

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

之前提到过开发SpringMVC程序需要导入spring-webmvc的坐标和spring整合web开发的坐标，就是上面这组坐标中的最后两个了。

但是我们发现除了这两个还有其他的，比如第二个，叫做spring-boot-starter-json。看名称就知道，这个是与json有关的坐标了，但是看名字发现和最后两个又不太一样，它的名字中也有starter，打开看看里面有什么？

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

```

        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.12.4</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.datatype</groupId>
        <artifactId>jackson-datatype-jdk8</artifactId>
        <version>2.12.4</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.datatype</groupId>
        <artifactId>jackson-datatype-jsr310</artifactId>
        <version>2.12.4</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.module</groupId>
        <artifactId>jackson-module-parameter-names</artifactId>
        <version>2.12.4</version>
        <scope>compile</scope>
    </dependency>
</dependencies>

```

我们可以发现，这个starter中又包含了若干个坐标，其实就是使用SpringMVC开发通常都会使用到Json，使用json又离不开这里面定义的这些坐标，看来还真是方便，SpringBoot把我们开发中使用的东西能用到的都给提前做好了。你仔细看完会发现，里面有一些你没用过的。的确会出现这种过量导入的可能性，没关系，可以通过maven中的排除依赖剔除掉一部分。不过你不管它也没事，大不了就是过量导入呗。

到这里基本上得到了一个信息，使用starter可以帮开发者快速配置依赖关系。以前写依赖3个坐标的，现在写导入一个就搞定了，就是加速依赖配置的。

### starter与parent的区别

朦朦胧胧中感觉starter与parent好像都是帮助我们简化配置的，但是功能又不一样，梳理一下。

**starter**是一个坐标中定了若干个坐标，以前写多个的，现在写一个，**是用来减少依赖配置的书写量的**

**parent**是定义了几百个依赖版本号，以前写依赖需要自己手工控制版本，现在由SpringBoot统一管理，这样就不存在版本冲突了，**是用来减少依赖冲突的**

### 实际开发应用方式

- 实际开发中如果需要用什么技术，先去找有没有这个技术对应的starter
  - 如果有对应的starter，直接写starter，而且无需指定版本，版本由parent提供
  - 如果没有对应的starter，手写坐标即可
- 实际开发中如果发现坐标出现了冲突现象，确认你要使用的可行的版本号，使用手工书写的方式添加对应依赖，覆盖SpringBoot提供给我们的配置管理
  - 方式一：直接写坐标
  - 方式二：覆盖中定义的版本号，就是下面这堆东西了，哪个冲突了覆盖哪个就OK了

```
<properties>
```

```
<activemq.version>5.16.3</activemq.version>
<aspectj.version>1.9.7</aspectj.version>
<assertj.version>3.19.0</assertj.version>
<commons-codec.version>1.15</commons-codec.version>
<commons-dbcp2.version>2.8.0</commons-dbcp2.version>
<commons-lang3.version>3.12.0</commons-lang3.version>
<commons-pool.version>1.6</commons-pool.version>
<commons-pool2.version>2.9.0</commons-pool2.version>
<h2.version>1.4.200</h2.version>
<hibernate.version>5.4.32.Final</hibernate.version>
<hibernate-validator.version>6.2.0.Final</hibernate-validator.version>
<httpClient.version>4.5.13</httpClient.version>
<jackson-bom.version>2.12.4</jackson-bom.version>
<javax-jms.version>2.0.1</javax-jms.version>
<javax-json.version>1.1.4</javax-json.version>
<javax-websocket.version>1.1</javax-websocket.version>
<jetty-el.version>9.0.48</jetty-el.version>
<junit.version>4.13.2</junit.version>
</properties>
```

### 温馨提示

SpringBoot官方给出了好多个starter的定义，方便我们使用，而且名称都是如下格式

命名规则：**spring-boot-starter-技术名称**

所以以后见了spring-boot-starter-aaa这样的名字，这就是SpringBoot官方给出的starter定义。那非官方定义的也有吗？有的，具体命名方式到整合章节再说

### 总结

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

## 引导类

配置说完了，我们发现SpringBoot确实帮助我们减少了很多配置工作，下面说一下程序是如何运行的。目前程序运行的入口就是SpringBoot工程创建时自带的那个类了，带有main方法的那个类，运行这个类就可以启动SpringBoot工程的运行

```
@SpringBootApplication
public class Springboot0101QuickstartApplication {
    public static void main(String[] args) {
        SpringApplication.run(Springboot0101QuickstartApplication.class, args);
    }
}
```

SpringBoot本身是为了加速Spring程序的开发的，而Spring程序运行的基础是需要创建自己的Spring容器对象（IoC容器）并将所有的对象交给Spring的容器管理，也就是一个一个的Bean。那还了SpringBoot加速开发Spring程序，这个容器还在吗？这个疑问不用说，一定在。当前这个类运行后就会产生一个Spring容器对象，并且可以将这个对象保存起来，通过容器对象直接操作Bean。

```

@SpringBootApplication
public class Springboot0101QuickstartApplication {
    public static void main(String[] args) {
        ConfigurableApplicationContext ctx =
SpringApplication.run(Springboot0101QuickstartApplication.class, args);
        BookController bean = ctx.getBean(BookController.class);
        System.out.println("bean=====>" + bean);
    }
}

```

通过上述操作不难看出，其实SpringBoot程序启动还是创建了一个Spring容器对象。这个类在SpringBoot程序中是所有功能的入口，称这个类为**引导类**。

作为一个引导类最典型的特征就是当前类上方声明了一个注解**@SpringBootApplication**

## 总结

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

## 思考

程序现在已经运行了，通过引导类的main方法运行了起来。但是运行java程序不应该是执行完就结束了吗？但是我们现在明显是启动了一个web服务器啊，不然网页怎么能正常访问呢？这个服务器是在哪里写的呢？

## 内嵌tomcat

当前我们做的SpringBoot入门案例勾选了Spring-web的功能，并且导入了对应的starter。

```

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

```

SpringBoot发现，既然你要做web程序，肯定离不开使用web服务器，这样吧，帮人帮到底，送佛送到西。我帮你搞一个web服务器，你要愿意用的，直接使用就好了，干脆我再多给你几种选择，你随便切换。万一你不想用我给你的，也行，你可以自己搞。

由于这个功能不属于程序的主体功能，可用可不用，于是乎SpringBoot将其定位成辅助功能，别小看这么一个辅助功能，它可是帮我们开发者又减少了好多的设置性工作。

下面就围绕着这个内置的web服务器，也可以说是内置的tomcat服务器来研究几个问题

1. 这个服务器在什么位置定义的
2. 这个服务器是怎么运行的
3. 这个服务器如果想换怎么换？虽然这个需求很垃圾，搞得开发者会好多web服务器一样，用别人提供好的不香么？非要自己折腾

## 内嵌Tomcat定义位置

说到定义的位置，我们就想，如果我们不开发web程序，用的着web服务器吗？肯定用不着啊。那如果这个东西被加入到你的程序中，伴随着什么技术进来的呢？肯定是web相关的功能啊，没错，就是前面导入的web相关的starter做的这件事。

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

打开查看web的starter导入了哪些东西

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

第三个依赖就是这个tomcat对应的东西了，居然也是一个starter，再打开看看

```
<dependencies>
  <dependency>
    <groupId>jakarta.annotation</groupId>
    <artifactId>jakarta.annotation-api</artifactId>
    <version>1.3.5</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-core</artifactId>
    <version>9.0.52</version>
    <scope>compile</scope>
  </dependency>
</dependencies>
```

```

        <exclusions>
            <exclusion>
                <artifactId>tomcat-annotations-api</artifactId>
                <groupId>org.apache.tomcat</groupId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-el</artifactId>
        <version>9.0.52</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-websocket</artifactId>
        <version>9.0.52</version>
        <scope>compile</scope>
        <exclusions>
            <exclusion>
                <artifactId>tomcat-annotations-api</artifactId>
                <groupId>org.apache.tomcat</groupId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>

```

这里面有一个核心的坐标，tomcat-embed-core，叫做tomcat内嵌核心。就是这个东西把tomcat功能引入到了我们的程序中。目前解决了第一个问题，找到根儿了，谁把tomcat引入到程序中的？spring-boot-starter-web中的spring-boot-starter-tomcat做的。之所以你感觉很奇妙的原因就是，这个东西是默认加入到程序中了，所以感觉很神奇，居然什么都不做，就有了web服务器对应的功能，再来说第二个问题，这个服务器是怎么运行的

## 内嵌Tomcat运行原理

Tomcat服务器是一款软件，而且是一款使用java语言开发的软件，熟悉的小伙伴可能有印象，tomcat安装目录中保存有jar，好多个jar。

下面的问题来了，既然是使用java语言开发的，运行的时候肯定符合java程序运行的原理，java程序运行靠的是什？对象呀，一切皆对象，万物皆对象。那tomcat运行起来呢？也是对象。

如果是对象，那Spring容器是用来管理对象的，这个对象能不能交给Spring容器管理呢？把吗去掉，是个对象都可以交给Spring容器管理，行了，这下通了。tomcat服务器运行其实是以对象的形式在Spring容器中运行的，怪不得我们没有安装这个tomcat，而且还能用。闹了白天这东西最后是以一个对象的形式存在，保存在Spring容器中悄悄运行的。具体运行的是什么呢？其实就是上前面提到的那个tomcat内嵌核心

```

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

```



那既然是个对象，如果把这个对象从Spring容器中去掉是不是就没有web服务器的功能呢？是这样的，通过依赖排除可以去掉这个web服务器功能

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

上面对web-starter做了一个操作，使用maven的排除依赖去掉了使用tomcat的starter。这下好了，容器中肯定没有这个对象了，重新启动程序可以观察到程序运行了，但是并没有像之前那样运行后会等着用户发请求，而是直接停掉了，就是这个原因了。

### 更换内嵌Tomcat

那根据上面的操作我们思考是否可以换个服务器呢？必须的嘛。根据SpringBoot的工作机制，用什么技术，加入什么依赖就行了。SpringBoot提供了3款内置的服务器

- 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：负载性能勉强跑赢tomcat

想用哪个，加个坐标就OK。前提是把tomcat排除掉，因为tomcat是默认加载的。

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

现在就已经成功替换了web服务器，核心思想就是用什么加入对应坐标就可以了。如果有starter，优先使用starter。

### 总结

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

到这里第一章快速上手SpringBoot就结束了，这一章我们学习了两大块知识

1. 使用了4种方式制作了SpringBoot的入门程序，不管是哪一种，其实内部都是一模一样的
2. 学习了入门程序的工作流程，知道什么是parent，什么是starter，这两个东西是怎么配合工作的，以及我们的程序为什么启动起来是一个tomcat服务器等等

第一章到这里就结束了，再往下学习就要去基于会创建SpringBoot工程的基础上，研究SpringBoot工程的具体细节了。

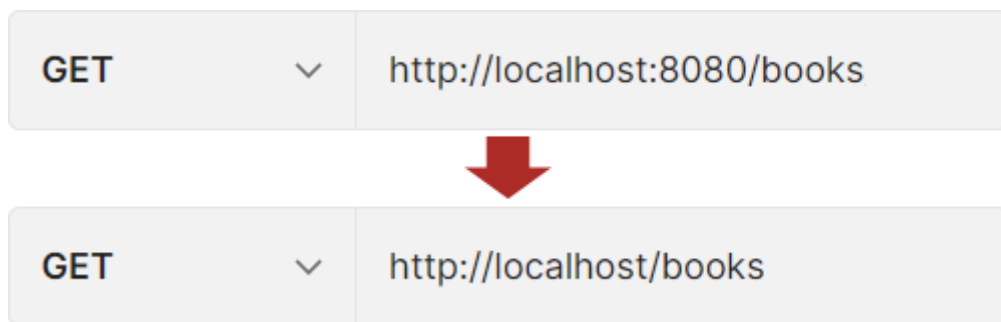
## JC-2.SpringBoot基础配置

入门案例做完了，下面就要研究SpringBoot的用法了。通过入门案例，各位小伙伴能够感知到一个信息，SpringBoot没有具体的功能，它在辅助加快Spring程序的开发效率。我们发现现在几乎不用做任何的配置，功能就有了，确实很好用。但是仔细想想，没有做配置意味着什么？意味着配置已经做好了，不用你自己写了。但是新的问题又来了，如果不想用已经写好的默认配置，该如何干预呢？这就是这一章咱们要研究的问题。

如果我们想修改默认的配置，这个信息应该写在什么位置呢？目前我们接触的入门案例中一共有3个文件，第一是pom.xml文件，设置项目的依赖的，这个没什么好研究的，相关的高级内容咱们到原理篇再说，第二是引导类，这个是执行SpringBoot程序的入口，也不像是做配置的地方，其实还有一个信息，就是在resources目录下面有一个空白的文件，叫做application.properties。一看就是个配置文件，咱们这一章就来说说配置文件怎么写，能写什么，怎么干预SpringBoot的默认配置，修改成自己的配置。

### JC-2-1.属性配置

SpringBoot通过配置文件application.properties就可以修改默认的配置，那咱们就先找个简单的配置下手，当前访问tomcat的默认端口是8080，好熟悉的味道，但是不便于书写，我们先改成80，通过这个操作来熟悉一下SpringBoot的配置格式是什么样的



那该如何写呢？properties格式的文件书写规范是key=value

```
name=example
```

这个格式肯定是不能颠覆的，那就尝试性的写就行了，改端口，写port。当你输入port后，神奇的事情就发生了，这玩意儿带提示，太好了

port	
p server.port=8080 (Server HTTP port)	Integer
p spring.data.cassandra.port=9042 (Port to u...	Integer
p spring.data.mongodb.port (Mongo server por...	Integer
p spring.integration.rsocket.client.port (TC...	Integer
p spring.ldap.embedded.port=0 (Embedded LDAP...	Integer
p spring.mail.port (SMTP server port)	Integer
p spring.rabbitmq.port (RabbitMQ port)	Integer
p spring.redis.port=6379 (Redis server port)	Integer
p spring.rsocket.server.port (Server port)	Integer
p spring.sendgrid.proxy.port (SendGrid proxy...	Integer
p server.tomcat.remoteip.port-header=X-Forwar...	String
p spring.config.import (Import additional List<String>...	

根据提示敲回车，输入80端口，搞定

```
server.port=80
```

下面就可以直接运行程序，测试效果了。

我们惊奇的发现SpringBoot这玩意儿狠啊，以前修改端口在哪里改？tomcat服务器的配置文件中改，现在呢？SpringBoot专用的配置文件中改，是不是意味着以后所有的配置都可以写在这一个文件中呢？是的，简化开发者配置的书写位置，集中管理。妙啊，妈妈再也不用担心我找不到配置文件了。

其实到这里我们应该得到如下三个信息

1. SpringBoot程序可以在application.properties文件中进行属性配置
2. application.properties文件中只要输入要配置的属性关键字就可以根据提示进行设置
3. SpringBoot将配置信息集中在一个文件中写，不管你是服务器的配置，还是数据库的配置，总之都写在一起，逃离一个项目十几种配置文件格式的尴尬局面

## 总结

1. SpringBoot默认配置文件是application.properties

做完了端口的配置，趁热打铁，再做几个配置，目前项目启动时会显示一些日志信息，就来改一改这里的一些设置。

## 关闭运行日志图表 (banner)

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

## 设置运行日志的显示级别

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

你会发现，现在这么搞配置太爽了，以前你做配置怎么做？不同的技术有自己专用的配置文件，文件不同格式也不统一，现在呢？不用东奔西走的找配置文件写配置了，统一格式了，这就是大秦帝国啊，统一六国。SpringBoot比大秦狠，因为未来出现的技术还没出现呢，但是现在已经确认了，配置都写这个文件里面。

我们现在配置了3个信息，但是又有新的问题了。这个配置是随便写的吗？什么都能配？有没有一个东西显示所有能配置的项呢？此外这个配置和什么东西有关呢？会不会因为我写了什么东西以后才可以写什么配置呢？比如我现在没有写数据库相关的东西，能否配置数据呢？一个一个来，先说第一个问题，都能配置什么。

打开SpringBoot的官网，找到SpringBoot官方文档，打开查看附录中的Application Properties就可以获取到对应的配置项了，网址奉上：<https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#application-properties>

能写什么的问题解决了，再来说第二个问题，这个配置项和什么有关。在pom中注释掉导入的spring-boot-starter-web，然后刷新工程，你会发现配置的提示消失了。闹了半天是设定使用了什么技术才能做什么配置。也合理，不然配置的东西都没有使用对应技术，配了也是白配。

### 温馨提示

所有的starter中都会依赖下面这个starter，叫做spring-boot-starter。这个starter是所有的SpringBoot的starter的基础依赖，里面定义了SpringBoot相关的基础配置，关于这个starter我们到开发应用篇和原理篇中再深入讲解。

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

### 总结

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

## JC-2-2.配置文件分类

现在已经能够进行SpringBoot相关的配置了，但是properties格式的配置写起来总是觉得看着不舒服，所以就期望存在一种书写起来更简便的配置格式提供给开发者使用。有吗？还真有，SpringBoot除了支持properties格式的配置文件，还支持另外两种格式的配置文件。分别如下：

- properties格式
- yaml格式
- yml格式

一看到全新的文件格式，各位小伙伴肯定想，这下又要学习新的语法格式了。怎么说呢？从知识角度来说，要学，从开发角度来说，不用学。为什么呢？因为SpringBoot的配置在Idea工具有提示啊，跟着提示走就行了。下面列举三种不同文件格式配置相同的属性范例，先了解一下

- application.properties (properties格式)

```
server.port=80
```

- application.yml (yaml格式)

```
server:
  port: 81
```

- application.yaml (yaml格式)

```
server:
  port: 82
```

仔细看会发现yml格式和yaml格式除了文件名后缀不一样，格式完全一样，是这样的，yml和yaml文件格式就是一模一样的，只是文件后缀不同，所以可以合并成一种格式来看。那对于这三种格式来说，以后用哪一种比较多呢？记清楚，以后基本上都是用yml格式的，本课程后面的所有知识都是基于yml格式来制作的，以后在企业开发过程中用这个格式的机会也最多，一定要重点掌握。

## 总结

1. SpringBoot提供了3种配置文件的格式
  - properties (传统格式/默认格式)
  - **yml** (主流格式)
  - yaml

## 思考

现在我们已经知道使用三种格式都可以做配置了，好奇宝宝们就有新的灵魂拷问了，万一我三个都写了，他们三个谁说了算呢？打一架吗？

## 配置文件优先级

其实三个文件如果共存的话，谁生效说的就是配置文件加载的优先级别。先说一点，虽然以后这种情况很少出现，但是这个知识还是可以学习一下的。我们就让三个配置文件书写同样的信息，比如都配置端口，然后我们让每个文件配置的端口号都不一样，最后启动程序后看启动端口是多少就知道谁的加载优先级比较高了。

- application.properties (properties格式)

```
server.port=80
```

- application.yml (yml格式)

```
server:
  port: 81
```

- application.yaml (yaml格式)

```
server:
  port: 82
```

启动后发现目前的启动端口为80，把80对应的文件删除掉，然后再启动，现在端口又改成了81。现在我们就已经知道了3个文件的加载优先顺序是什么

```
application.properties > application.yml > application.yaml
```

虽然得到了一个知识结论，但是我们实际开发的时候还是要看最终的效果为准。也就是你要的最终效果是什么自己是明确的，上述结论只能帮助你分析结论产生的原因。这个知识了解一下就行了，因为以后同时写多种配置文件格式的情况实在是较少。

最后我们把配置文件内容给修改一下

- application.properties (properties格式)

```
server.port=80
spring.main.banner-mode=off
```

- application.yml (yaml格式)

```
server:
  port: 81
logging:
  level:
    root: debug
```

- application.yaml (yaml格式)

```
server:
  port: 82
```

我们发现不仅端口生效了，最终显示80，同时其他两条配置也生效了，看来每个配置文件中的项都会生效，只不过如果多个配置文件中有相同类型的配置会优先级高的文件覆盖优先级的文件中的配置。如果配置项不同的话，那所有的配置项都会生效。

## 总结

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

## 教你一招：自动提示功能消失解决方案

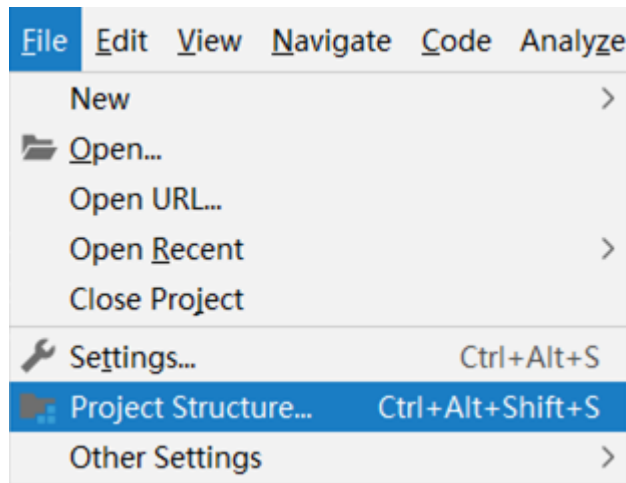
可能有些小伙伴会基于各种各样的原因导致配置文件中没有提示，这个确实很让人头疼，所以下面给大家说一下如果自动提示功能消失了怎么解决。

先要明确一个核心，就是自动提示功能不是SpringBoot技术给我们提供的，是我们在Idea工具下编程，这个编程工具给我们提供的。明白了这一点后，再来说为什么会出现这种现象。其实这个自动提示功能消失的原因还是蛮多的，如果想解决这个问题，就要知道为什么会消失，大体原因有如下3种：

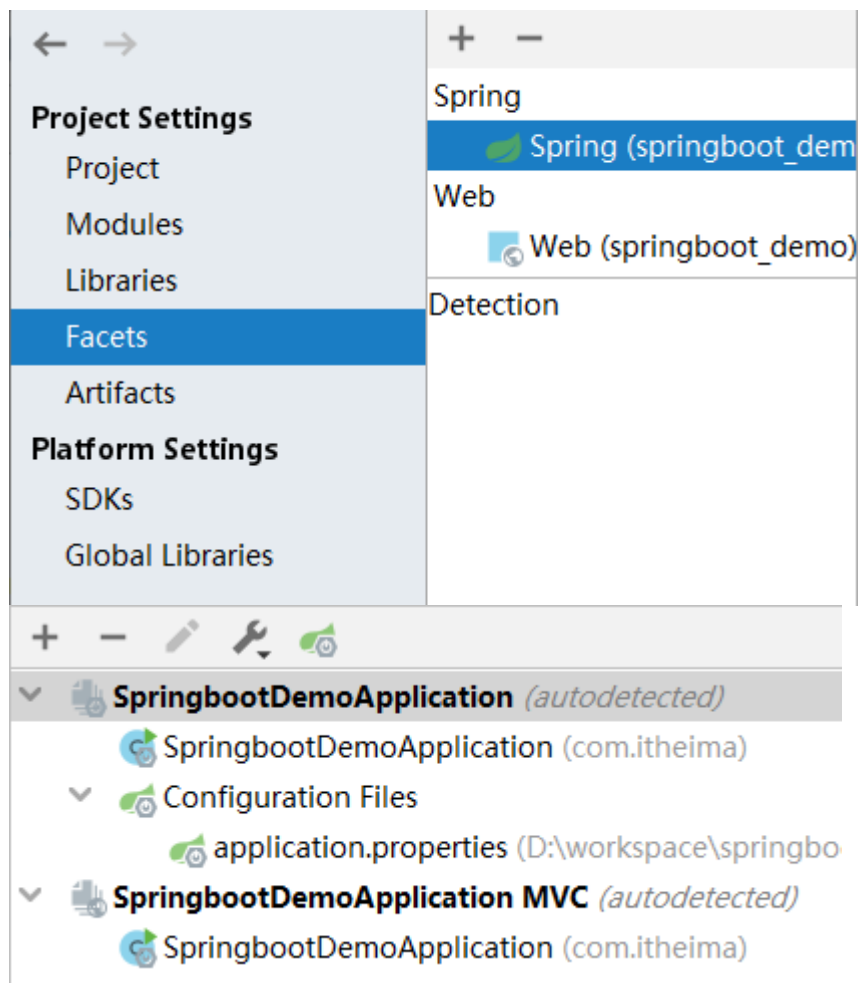
1. Idea认为你现在写配置的文件不是个配置文件，所以拒绝给你提供提示功能
2. Idea认定你是合理的配置文件，但是Idea加载不到对应的提示信息

这里我们主要解决第一个现象，第二种现象到原理篇再讲解。第一种现象的解决方式如下：

**步骤①：** 打开设置，【Files】→【Project Structure...】

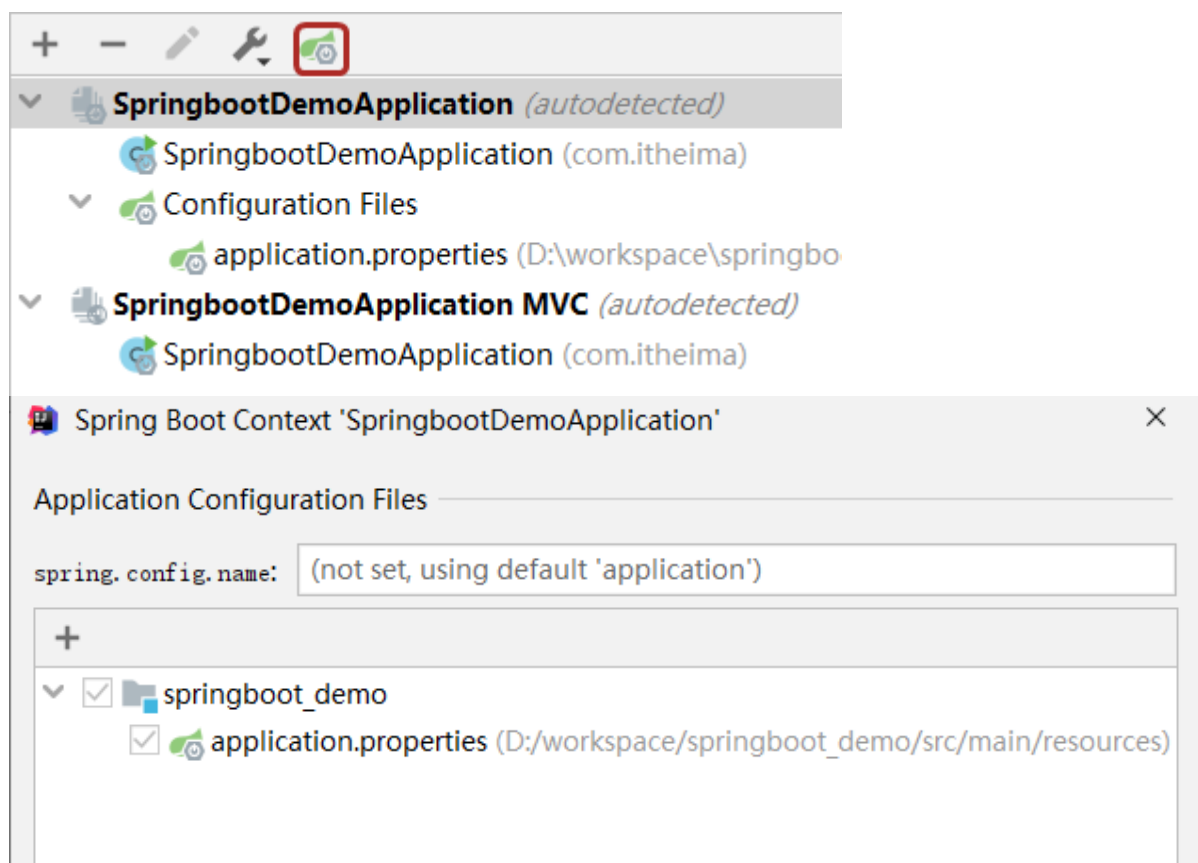


步骤②：在弹出窗口中左侧选择【Facets】，右侧选中Spring路径下对应的模块名称，也就是你自动提示功能消失的那个模块

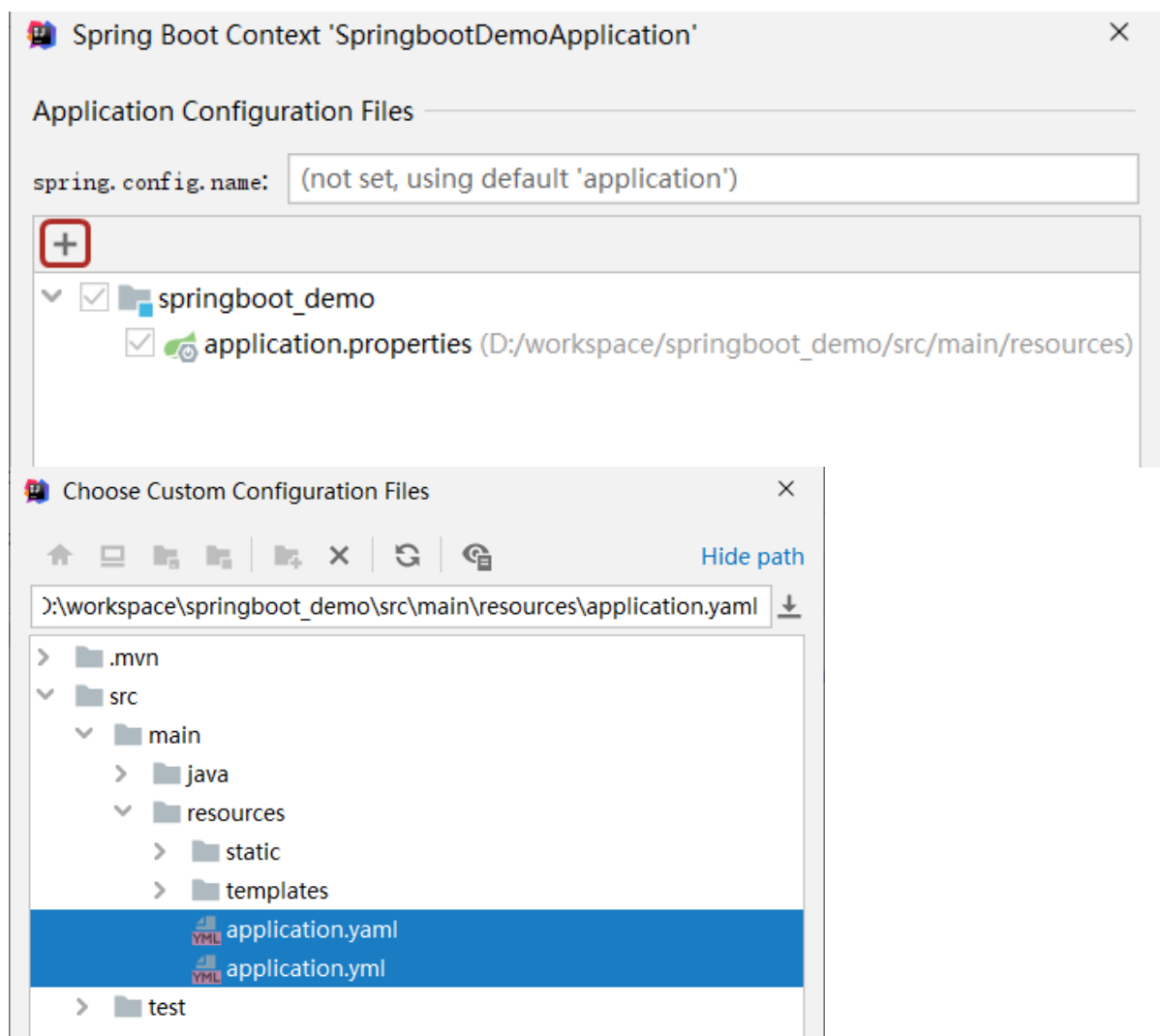


步骤③：点击Customize Spring Boot按钮，此时可以看到当前模块对应的配置文件是哪些了。如果没有你想要称为配置文件的文件格式，就有可能无法弹出提示

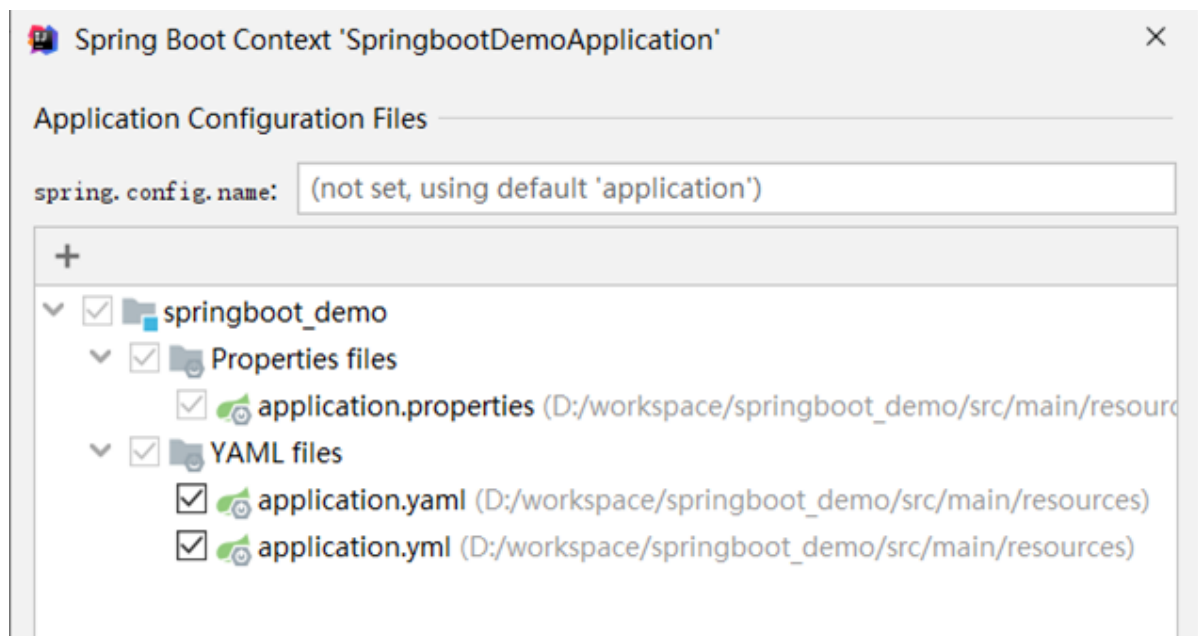




步骤④：选择添加配置文件，然后选中要作为配置文件的具体文件就OK了



到这里就做完了，其实就是Idea的一个小功能



## 总结

### 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

## JC-2-3.yaml文件

SpringBoot的配置以后主要使用yaml结尾的这种文件格式，并且在书写时可以通过提示的形式加载正确的格式。但是这种文件还是有严格的书写格式要求的。下面就来说一下具体的语法规则。

YAML (YAML Ain't Markup Language)，一种数据序列化格式。具有容易阅读、容易与脚本语言交互、以数据为核心，重数据轻格式的特点。常见的文件扩展名有两种：

- .yaml格式（主流）
- .yml格式

对于文件自身在书写时，具有严格的语法规则要求，具体如下：

1. 大小写敏感
2. 属性层级关系使用多行描述，**每行结尾使用冒号结束**
3. 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）
4. 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）
5. #号 表示注释

上述规则不要死记硬背，按照书写习惯慢慢适应，并且在Idea下由于具有提示功能，慢慢适应着写格式就行了。核心的一条规则要记住，**数据前面要加空格与冒号隔开**

下面列出常见的数据书写格式，熟悉一下

```

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

```

此外, yaml格式中也可以表示数组, 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: abc
  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:                          #对象数组格式一
  - name: Tom
    age: 4
  - name: Jerry
    age: 5
users:                          #对象数组格式二
  -
    name: Tom
    age: 4
  -
    name: Jerry
    age: 5
users2: [ { name:Tom , age:4 } , { name:Jerry , age:5 } ]  #对象数组缩略格式

```

## 总结

### 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)
- #号 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

## 思考

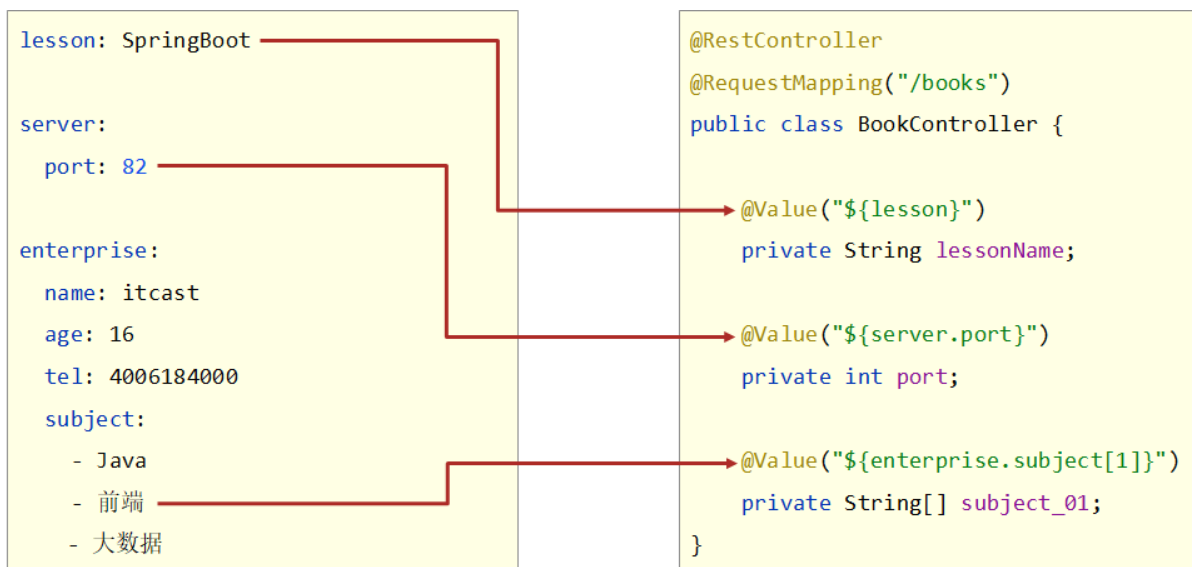
现在我们已经知道了yaml具有严格的数据格式要求，并且已经可以正确的书写yaml文件了，那这些文件书写后其实是在定义一些数据。这些数据时给谁用的呢？大部分是SpringBoot框架内部使用，但是如果 we 想配置一些数据自己使用，能不能用呢？答案是可以的，那如何读取yaml文件中的数据呢？咱们下一节再说。

## JC-2-4.yaml数据读取

对于yaml文件中的数据，其实你就可以想象成这就是一个小型的数据库，里面保存有若干数据，每个数据都有一个独立的名字，如果你想读取里面的数据，肯定是支持的，下面就介绍3种读取数据的方式

### 读取单一数据

yaml中保存的单个数据，可以使用Spring中的注解直接读取，使用@Value可以读取单个数据，属性名引用方式：**`${一级属性名.二级属性名.....}`**



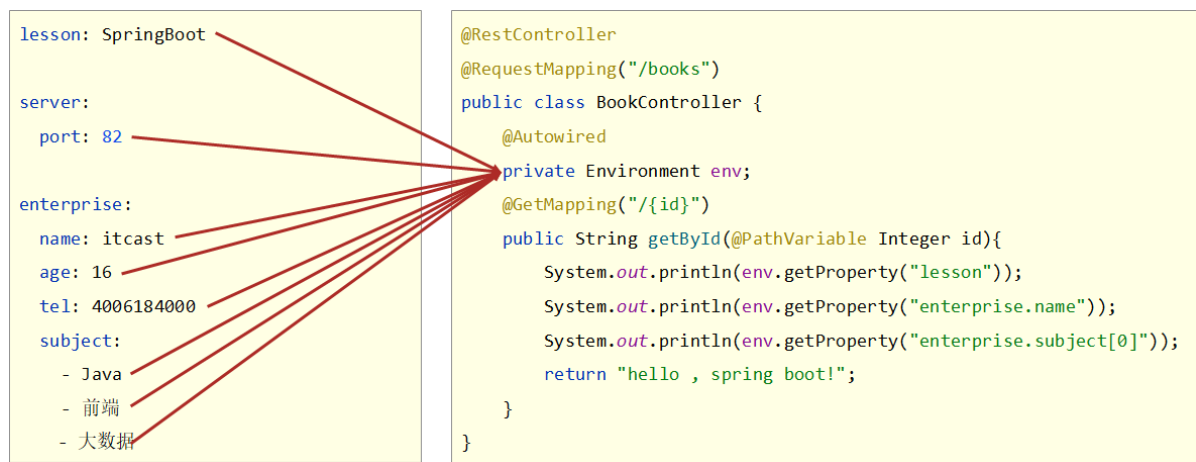
记得使用@Value注解时，要将该注入写在某一个指定的Spring管控的bean的属性名上方。现在就可以读取到对应的单一数据行了

### 总结

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

### 读取全部数据

读取单一数据可以解决读取数据的问题，但是如果定义的数据量过大，这么一个一个书写肯定会累死人的，SpringBoot提供了一个对象，能够把所有的数据都封装到这一个对象中，这个对象叫做Environment，使用自动装配注解可以将所有的yaml数据封装到这个对象中



数据封装到了Environment对象中，获取属性时，通过Environment的接口操作进行，具体方法时getProperties (String) ，参数填写属性名即可

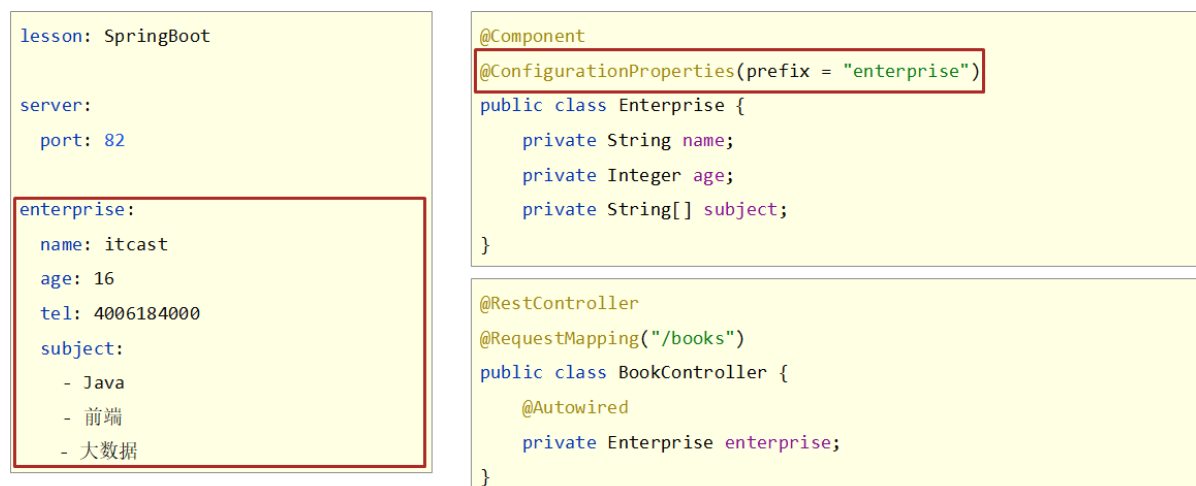
## 总结

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

## 读取对象数据

单一数据读取书写比较繁琐，全数据封装又封装的太厉害了，每次拿数据还要一个一个的getProperties () ,总之用起来都不是很舒服。由于Java是一个面向对象的语言，很多情况下，我们会将一组数据封装成一个对象。SpringBoot也提供了可以将一组yaml对象数据封装一个Java对象的操作

首先定义一个对象，并将该对象纳入Spring管控的范围，也就是定义成一个bean，然后使用注解@ConfigurationProperties指定该对象加载哪一组yaml中配置的信息。



这个@ConfigurationProperties必须告诉他加载的数据前缀是什么，这样当前前缀下的所有属性就封装到这个对象中。记得数据属性名要与对象的变量名——对应啊，不然没法封装。其实以后如果你要定义一组数据自己使用，就可以先写一个对象，然后定义好属性，下面到配置中根据这个格式书写即可。

```
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;
}
```

### 温馨提示

细心的小伙伴会发现一个问题，自定义的这种数据在yaml文件中书写时没有弹出提示，是这样的，咱们到原理篇再揭秘如何弹出提示。

### 总结

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

## yaml文件中的数据引用

如果你在书写yaml数据时，经常出现如下现象，比如很多个文件都具有相同的目录前缀

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

或者

```
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
```

这个时候你可以使用引用格式来定义数据，其实就是搞了个变量名，然后引用变量了，格式如下：

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

还有一个注意事项，在书写字符串时，如果需要使用转义字符，需要将数据字符串使用双引号包裹起来

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

## 总结

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

到这里有关yaml文件的基础使用就先告一段落，在实用篇中再继续研究更深入的内容。

## JC-3.基于SpringBoot实现SSMP整合

重头戏来了，SpringBoot之所以好用，就是它能方便快捷的整合其他技术，这一部分咱们就来聊聊一些技术的整合方式，通过这一章的学习，大家能够感受到SpringBoot到底有多酷炫。这一章咱们学习如下技术的整合方式

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

上面这些技术都整合完毕后，我们做一个小案例，也算是学有所用吧。涉及的技术比较多，综合运用一下。

### JC-3-1.整合JUnit

SpringBoot技术的定位用于简化开发，再具体点是简化Spring程序的开发。所以在整合任意技术的时候，如果你想直观感触到简化的效果，你必须先知道使用非SpringBoot技术时对应的整合是如何做的，然后再看基于SpringBoot的整合是如何做的，才能比对出来简化在了哪里。

我们先来看一下不使用SpringBoot技术时，Spring整合JUnit的制作方式

```
//加载spring整合junit专用的类运行器
@RunWith(SpringJUnit4ClassRunner.class)
//指定对应的配置信息
@ContextConfiguration(classes = SpringConfig.class)
public class AccountServiceTestCase {
    //注入你要测试的对象
    @Autowired
    private AccountService accountService;
    @Test
    public void testGetById(){
        //执行要测试的对象对应的方法
        System.out.println(accountService.findById(2));
    }
}
```



其中核心代码是前两个注解，第一个注解@RunWith是设置Spring专用于测试的类运行器，简单说就是Spring程序执行程序有自己的一套独立的运行程序的方式，不能使用JUnit提供的类运行方式了，必须指定一下，但是格式是固定的，琢磨一下，**每次都指定一样的东西，这个东西写起来没有技术含量啊**，第二个注解@ContextConfiguration是用来设置Spring核心配置文件或配置类的，简单说就是加载Spring的环境你要告诉Spring具体的环境配置是在哪里写的，虽然每次加载的文件都有可能不同，但是仔细想想，如果文件名是固定的，这个貌似也是一个固定格式。似然**有可能是固定格式，那就有可能每次都写一样的东西，也是一个没有技术含量的内容书写**

SpringBoot就抓住上述两条没有技术含量的内容书写进行开发简化，能走默认值的走默认值，能不写的就不写，具体格式如下

```
@SpringBootTest
class Springboot04JUnitApplicationTests {
    //注入你要测试的对象
    @Autowired
    private BookDao bookDao;
    @Test
    void contextLoads() {
        //执行要测试的对象对应的方法
        bookDao.save();
        System.out.println("two...");
    }
}
```

看看这次简化成什么样了，一个注解就搞定了，而且还没有参数，再体会SpringBoot整合其他技术的优势在哪里，就两个字——**简化**。使用一个注解@SpringBootTest替换了前面两个注解。至于内部是怎么回事？和之前一样，只不过都走默认值。

这个时候有人就问了，你加载的配置类或者配置文件是哪一个？就是我们前面启动程序使用的引导类。如果想手工指定引导类有两种方式，第一种方式使用属性的形式进行，在注解@SpringBootTest中添加classes属性指定配置类

```
@SpringBootTest(classes = Springboot04JUnitApplication.class)
class Springboot04JUnitApplicationTests {
    //注入你要测试的对象
    @Autowired
    private BookDao bookDao;
    @Test
    void contextLoads() {
        //执行要测试的对象对应的方法
        bookDao.save();
        System.out.println("two...");
    }
}
```

第二种方式回归原始配置方式，仍然使用@ContextConfiguration注解进行，效果是一样的

```

@SpringBootTest
@ContextConfiguration(classes = Springboot04JunitApplication.class)
class Springboot04JunitApplicationTests {
    //注入你要测试的对象
    @Autowired
    private BookDao bookDao;
    @Test
    void contextLoads() {
        //执行要测试的对象对应的方法
        bookDao.save();
        System.out.println("two...");
    }
}

```

### 温馨提示

使用SpringBoot整合JUnit需要保障导入test对应的starter，由于初始化项目时此项是默认导入的，所以此处没有提及，其实和之前学习的内容一样，用什么技术导入对应的starter即可。

### 总结

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
4. 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
5. 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

## JC-3-2.整合MyBatis

整合完JUnit下面再来说一下整合MyBatis，这个技术是大部分公司都要使用的技术，务必掌握。如果对Spring整合MyBatis不熟悉的小伙伴好好复习一下，下面列举出原始整合的全部内容，以配置类的形式为例进行

- 导入坐标，MyBatis坐标不能少，Spring整合MyBatis还有自己专用的坐标，此外Spring进行数据库操作的jdbc坐标是必须的，剩下还有mysql驱动坐标，本例中使用了Druid数据源，这个倒是可以不要

```

<dependencies>
    <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>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>

```

```

</dependency>
<!--1.导入mybatis与spring整合的jar包-->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-spring</artifactId>
    <version>1.3.0</version>
</dependency>
<!--导入spring操作数据库必选的包-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.2.10.RELEASE</version>
</dependency>
</dependencies>

```

- Spring核心配置

```

@Configuration
@ComponentScan("com.example")
@PropertySource("jdbc.properties")
public class SpringConfig {
}

```

- MyBatis要交给Spring接管的bean

```

//定义mybatis专用的配置类
@Configuration
public class MyBatisConfig {
    //    定义创建SqlSessionFactory对应的bean
    @Bean
    public SqlSessionFactoryBean sqlSessionFactory(DataSource dataSource){
        //SqlSessionFactoryBean是由mybatis-spring包提供的，专用于整合用的对象
        SqlSessionFactoryBean sfb = new SqlSessionFactoryBean();
        //设置数据源替代原始配置中的environments的配置
        sfb.setDataSource(dataSource);
        //设置类型别名替代原始配置中的typeAliases的配置
        sfb.setTypeAliasesPackage("com.example.domain");
        return sfb;
    }
    //    定义加载所有的映射配置
    @Bean
    public MapperScannerConfigurer mapperScannerConfigurer(){
        MapperScannerConfigurer msc = new MapperScannerConfigurer();
        msc.setBasePackage("com.example.dao");
        return msc;
    }
}

```

- 数据源对应的bean，此处使用Druid数据源

```

@Configuration
public class JdbcConfig {
    @Value("${jdbc.driver}")
    private String driver;
}

```

```

@Value("${jdbc.url}")
private String url;
@Value("${jdbc.username}")
private String userName;
@Value("${jdbc.password}")
private String password;

@Bean("dataSource")
public DataSource dataSource(){
    DruidDataSource ds = new DruidDataSource();
    ds.setDriverClassName(driver);
    ds.setUrl(url);
    ds.setUsername(userName);
    ds.setPassword(password);
    return ds;
}
}

```

- 数据库连接信息 (properties格式)

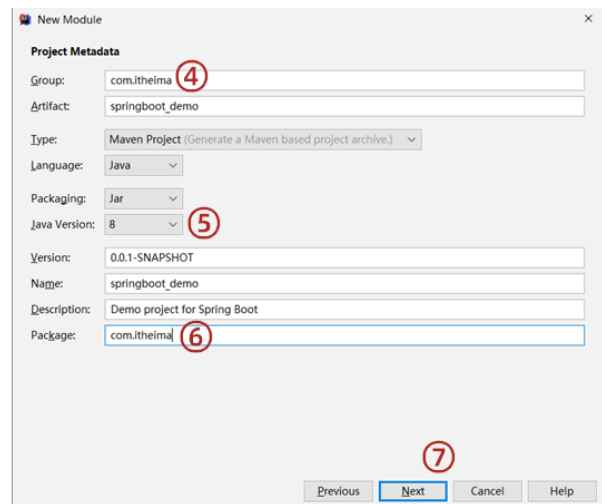
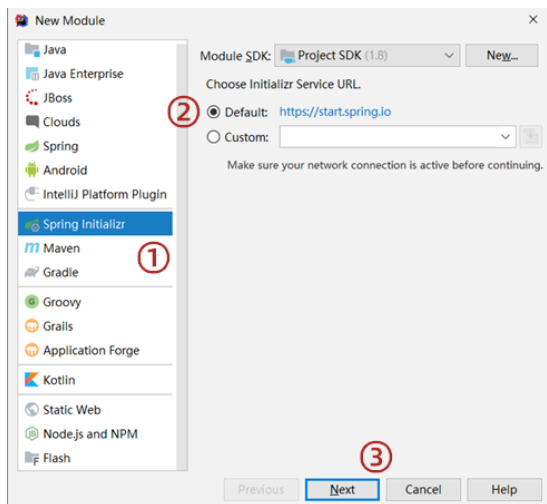
```

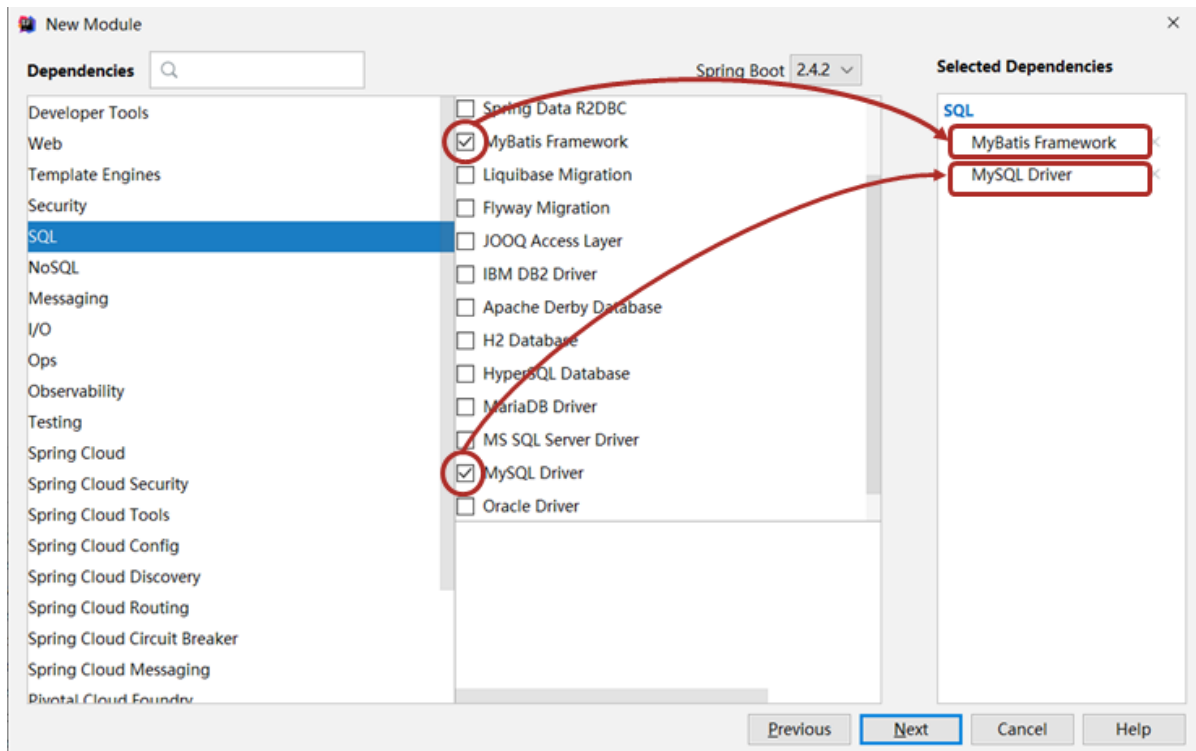
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/spring_db?useSSL=false
jdbc.username=root
jdbc.password=root

```

上述格式基本上是简格式了，要写的东西还真不少。下面看看SpringBoot整合MyBatis格式

**步骤①：**创建模块时勾选要使用的技术，MyBatis，由于要操作数据库，还要勾选对应数据库





或者手工导入对应技术的starter，和对应数据库的坐标

```
<dependencies>
  <!--1.导入对应的starter-->
  <dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.2.0</version>
  </dependency>

  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

**步骤②：**配置数据源相关信息，没有这个信息你连接哪个数据库都不知道

## #2. 配置相关信息

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

完了，就这么多，没了。有人就很纳闷，这就结束了？对，这就结束了，SpringBoot把配置中所有可能出现的通用配置都简化了。下面就可以写一下MyBatis程序运行需要的Dao（或者Mapper）就可以运行了

**实体类**

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

## 映射接口 (Dao)

```
@Mapper
public interface BookDao {
    @Select("select * from tbl_book where id = #{id}")
    public Book getById(Integer id);
}
```

## 测试类

```
@SpringBootTest
class Springboot05MybatisApplicationTests {
    @Autowired
    private BookDao bookDao;
    @Test
    void contextLoads() {
        System.out.println(bookDao.getById(1));
    }
}
```

完美，开发从此变的就这么简单。再体会一下SpringBoot如何进行第三方技术整合的，是不是很优秀？具体内部的原理到原理篇再展开讲解

**注意：**当前使用的SpringBoot版本是2.5.4，对应的坐标设置中Mysql驱动使用的是8x版本。当SpringBoot2.4.3（不含）版本之前会出现一个小BUG，就是MySQL驱动升级到8以后要求强制配置时区，如果不设置会出问题。解决方案很简单，驱动url上面添加上对应设置就行了

### #2. 配置相关信息

spring:

datasource:

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

这里设置的UTC是全球标准时间，你也可以理解为是英国时间，中国处在东八区，需要在这个基础上加上8小时，这样才能和中国地区的时间对应的，也可以修改配置不写UTC，写Asia/Shanghai也可以解决这个问题。

## #2. 配置相关信息

spring:

datasource:

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

如果不想每次都设置这个东西，也可以去修改mysql中的配置文件mysql.ini，在mysqld项中添加default-time-zone=+8:00也可以解决这个问题。其实方式方法很多，这里就说这么多吧。

此外在运行程序时还会给出一个提示，说数据库驱动过时的警告，根据提示修改配置即可，弃用com.mysql.jdbc.Driver，换用com.mysql.cj.jdbc.Driver。前面的例子中已经更换了驱动了，在此说明一下。

```
Loading class 'com.mysql.jdbc.Driver'. This is deprecated. The new driver class
is 'com.mysql.cj.jdbc.Driver'. The driver is automatically registered via the SPI
and manual loading of the driver class is generally unnecessary.
```

## 总结

1. 整合操作需要勾选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
4. 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
5. 驱动类过时，提醒更换为com.mysql.cj.jdbc.Driver

## JC-3-3.整合MyBatis-Plus

做完了两种技术的整合了，各位小伙伴要学会总结，我们做这个整合究竟哪些是核心？总结下来就两句话

- 导入对应技术的starter坐标
- 根据对应技术的要求做配置

虽然看起来有点虚，但是确实是这个理儿，下面趁热打铁，再换一个技术，看看是不是上面这两步。

接下来在MyBatis的基础上再升级一下，整合MyBaitsPlus（简称MP），国人开发的技术，符合中国人开发习惯，谁用谁知道。来吧，一起做整合

**步骤①：**导入对应的starter

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



关于这个坐标，此处要说明一点，之前我们看的starter都是spring-boot-starter-???，也就是说都是下面的格式

```
Spring-boot-start-***
```

而这个坐标的名字书写比较特殊，是第三方技术名称在前，boot和starter在后。此处简单提一下命名规范，后期原理篇会再详细讲解

starter所属	命名规则	示例
官方提供	spring-boot-starter-技术名称	spring-boot-starter-web spring-boot-starter-test
第三方提供	第三方技术名称-spring-boot-starter	druid-spring-boot-starter
第三方提供	第三方技术名称-boot-starter（第三方技术名称过长，简化命名）	mybatis-plus-boot-starter

温馨提示

有些小伙伴在创建项目时想通过勾选的形式找到这个名字，别翻了，没有。截止目前，SpringBoot官网还未收录此坐标，而我们Idea创建模块时读取的是SpringBoot官网的Spring Initializr，所以也没有。如果换用阿里云的url创建项目可以找到对应的坐标

步骤②：配置数据源相关信息

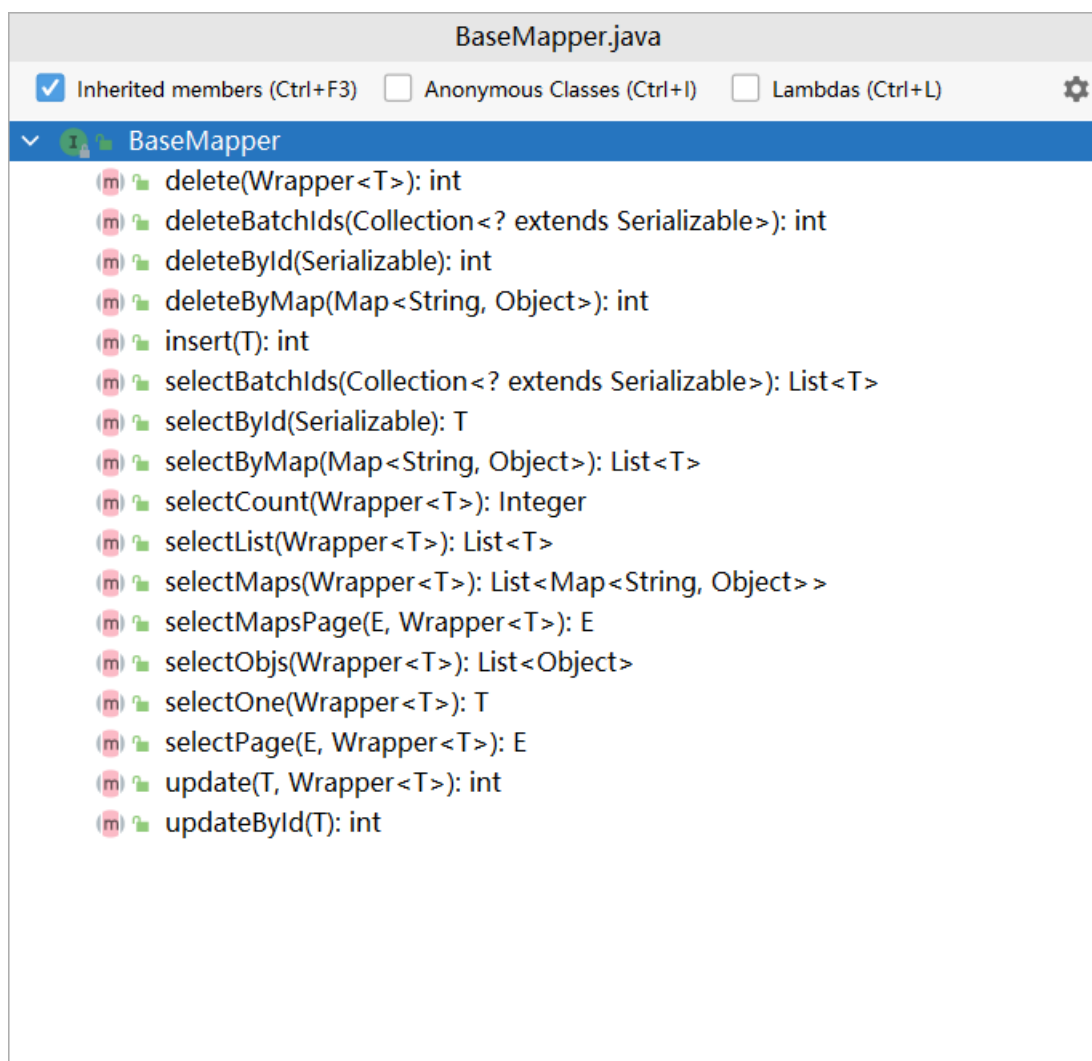
```
#2. 配置相关信息
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/ssm_db
    username: root
    password: root
```

没了，就这么多，剩下的就是写MyBaitsPlus的程序了

映射接口 (Dao)

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

核心在于Dao接口继承了一个BaseMapper的接口，这个接口中帮助开发者预定了若干个常用的API接口，简化了通用API接口的开发工作。



下面就可以写一个测试类进行测试了，此处省略。

### 温馨提示

目前数据库的表名定义规则是tbl\_模块名称，为了能和实体类相对应，需要做一个配置，相关知识各位小伙伴可以到MyBatisPlus课程中去学习，此处仅给出解决方案。配置application.yml文件，添加如下配置即可，设置所有表名的通用前缀名

```
mybatis-plus:
  global-config:
    db-config:
      table-prefix: tbl_      #设置所有表的通用前缀名称为tbl_
```

### 总结

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

## JC-3-4.整合Druid

使用SpringBoot整合了3个技术了，发现套路基本相同，导入对应的starter，然后做配置，各位小伙伴需要一直强化这套思想。下面再整合一个技术，继续深入强化此思想。

前面整合MyBatis和MP的时候，使用的数据源对象都是SpringBoot默认的数据源对象，下面我们手工控制一下，自己指定了一个数据源对象，Druid。

在没有指定数据源时，我们的配置如下：

### #2. 配置相关信息

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

此时虽然没有指定数据源，但是根据SpringBoot的德行，肯定帮我们选了一个它认为最好的数据源对象，这就是HiKari。通过启动日志可以查看到对应的身影。

```
2021-11-29 09:39:15.202 INFO 12260 --- [main]
com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2021-11-29 09:39:15.208 WARN 12260 --- [main]
com.zaxxer.hikari.util.DriverDataSource : Registered driver with
driverClassName=com.mysql.jdbc.Driver was not found, trying direct instantiation.
2021-11-29 09:39:15.551 INFO 12260 --- [main]
com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
```

上述信息中每一行都有HiKari的身影，如果需要更换数据源，其实只需要两步即可。

1. 导入对应的技术坐标
2. 配置使用指定的数据源类型

下面就切换一下数据源对象

**步骤①：**导入对应的坐标（注意，是坐标，此处不是starter）

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

**步骤②：**修改配置，在数据源配置中有一个type属性，专用于指定数据源类型

```

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

```

这里其实要提出一个问题的，目前的数据源配置格式是一个通用格式，不管你换什么数据源都可以用这种形式进行配置。但是新的问题又来了，如果对数据源进行个性化的配置，例如配置数据源对应的连接数量，这个时候就有新的问题了。每个数据源技术对应的配置名称都一样吗？肯定不是啊，各个厂商不可能提前商量好都写一样的名字啊，怎么办？就要使用专用的配置格式了。这个时候上面这种通用格式就不能使用了，怎么办？还能怎么办？按照SpringBoot整合其他技术的通用规则来套啊，导入对应的starter，进行相应的配置即可。

**步骤①：** 导入对应的starter

```

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

```

**步骤②：** 修改配置

```

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

```

注意观察，配置项中，在datasource下面并不是直接配置url这些属性的，而是先配置了一个druid节点，然后再配置的url这些东西。言外之意，url这个属性是druid下面的属性，那你能想到吗？除了这4个常规配置外，还有druid专用的其他配置。通过提示功能可以打开druid相关的配置查阅

```

P spring.datasource.druid.access-to-underlying-conne...
P spring.datasource.druid.active-connection-stack-tr...
P spring.datasource.druid.active-connections          Set<Dru...
P spring.datasource.druid.aop-patterns                String[]
P spring.datasource.druid.async-close-connection-ena...
P spring.datasource.druid.async-init                  Boolean
P spring.datasource.druid.break-after-acquire-failure
P spring.datasource.druid.check-execute-time          Boolean
P spring.datasource.druid.clear-filters-enable        Boole...
P spring.datasource.druid.connect-properties           Map<Str...
P spring.datasource.druid.connection-error-retry-att...
P spring.datasource.druid.connection-init-sql         Collec...
Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards Next Tip

```

与druid相关的配置超过200条以上，这就告诉你，如果想做druid相关的配置，使用这种格式就可以了，这里就不展开描述了，太多了。

这是我们做的第4个技术的整合方案，还是那两句话：**导入对应starter，使用对应配置**。没了，SpringBoot整合其他技术就这么简单粗暴。

## 总结

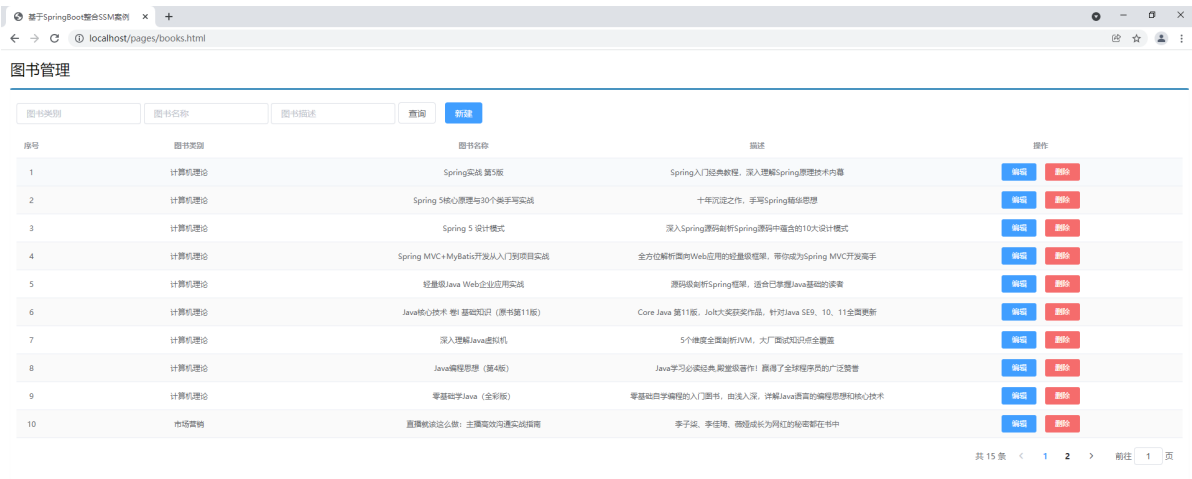
1. 整合Druid需要导入Druid对应的starter
2. 根据Druid提供的配置方式进行配置
3. 整合第三方技术通用方式
  - 导入对应的starter
  - 根据提供的配置格式，配置非默认值对应的配置项

## JC-3-5.SSMP整合综合案例

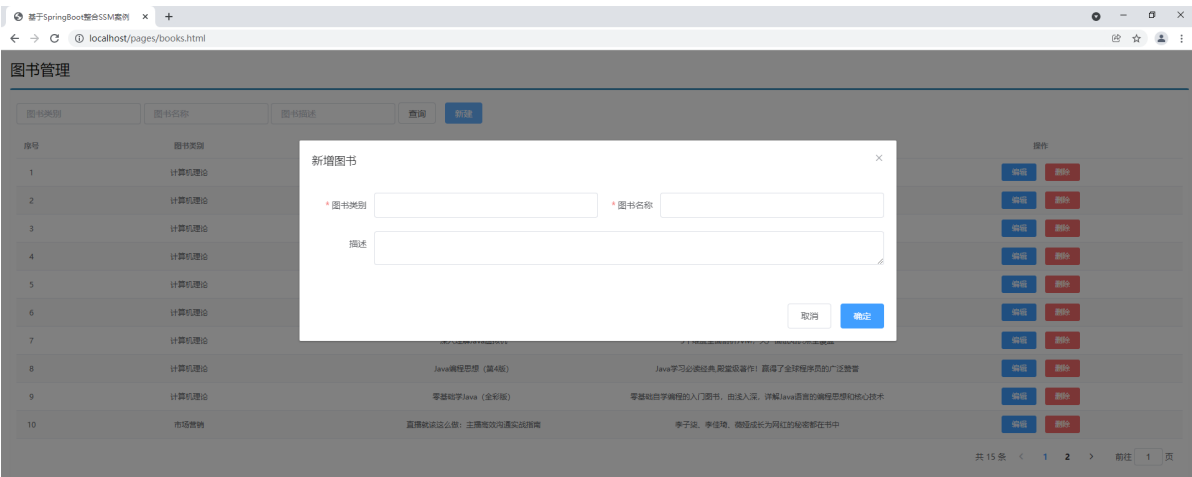
SpringBoot能够整合的技术太多太多了，对于初学者来说慢慢来，一点点掌握。前面咱们做了4个整合了，下面就通过一个稍微综合一点的案例，将所有知识贯穿起来，同时做一个小功能，体会一下。不过有言在先，这个案例制作的时候，你可能会会有这种感觉，说好的SpringBoot整合其他技术的案例，为什么感觉SpringBoot整合其他技术的身影不多呢？因为这东西书写太简单了，简单到瞬间写完，大量的时间做的不是这些整合工作。

先看一下这个案例的最终效果

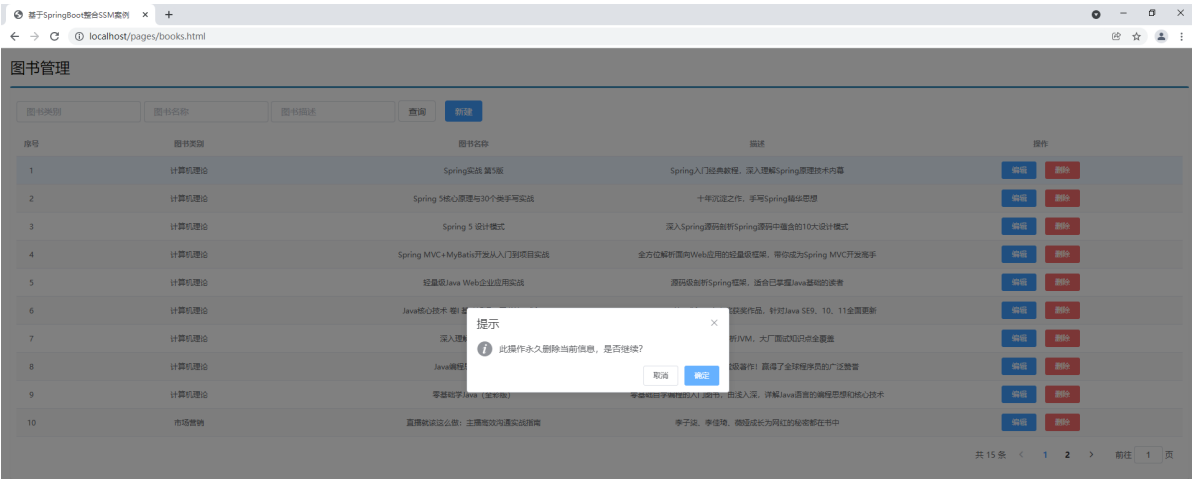
## 主页面



## 添加



删除



修改

分页



条件查询



整体案例中需要采用的技术如下，先了解一下，做到哪一个说哪一个

- 1. 实体类开发————使用Lombok快速制作实体类
- 2. Dao开发————整合MyBatisPlus，制作数据层测试
- 3. Service开发————基于MyBatisPlus进行增量开发，制作业务层测试类
- 4. Controller开发————基于Restful开发，使用PostMan测试接口功能
- 5. Controller开发————前后端开发协议制作
- 6. 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

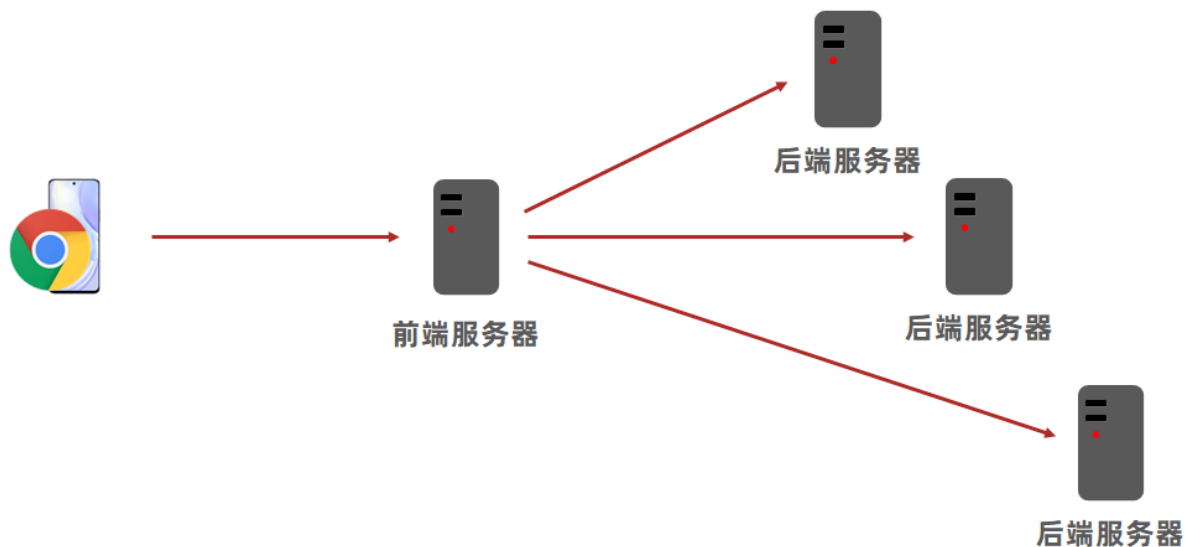
## 7. 项目异常处理

### 8. 按条件查询———页面功能调整、Controller修正功能、Service修正功能

可以看的出来，东西还是很多的，希望通过这个案例，各位小伙伴能够完成基础开发的技能训练。整体开发过程采用做一层测一层的形式进行，过程完整，战线较长，希望各位能跟进进度，完成这个小案例的制作。

## 0.模块创建

对于这个案例如果按照企业开发的形式进行应该制作后台微服务，前后端分离的开发。



我知道这个对初学的小伙伴要求太高了，咱们简化一下。后台做单体服务器，前端不使用前后端分离的制作了。



一个服务器即充当后台服务调用，又负责前端页面展示，降低学习的门槛。

下面我们就可以创建一个新的模块，加载要使用的技术对应的starter，修改配置文件格式为yml格式，并把web访问端口先设置成80。

**pom.xml**



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

## application.yml

```
server:
  port: 80
```

## 1. 实体类开发

本案例对应的模块表结构如下：

```
-- -----
-- Table structure for tbl_book
-- -----
DROP TABLE IF EXISTS `tbl_book`;
CREATE TABLE `tbl_book` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `type` varchar(20) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
  `name` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
  `description` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 51 CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Dynamic;

-- -----
-- Records of tbl_book
-- -----
INSERT INTO `tbl_book` VALUES (1, '计算机理论', 'Spring实战 第5版', 'Spring入门经典教程，深入理解Spring原理技术内幕');
INSERT INTO `tbl_book` VALUES (2, '计算机理论', 'Spring 5核心原理与30个类手写实战', '十年沉淀之作，手写Spring精华思想');
INSERT INTO `tbl_book` VALUES (3, '计算机理论', 'Spring 5 设计模式', '深入Spring源码剖析Spring源码中蕴含的10大设计模式');
INSERT INTO `tbl_book` VALUES (4, '计算机理论', 'Spring MVC+MyBatis开发从入门到项目实战', '全方位解析面向web应用的轻量级框架，带你成为Spring MVC开发高手');
INSERT INTO `tbl_book` VALUES (5, '计算机理论', '轻量级Java web企业应用实战', '源码级剖析Spring框架，适合已掌握Java基础的读者');
INSERT INTO `tbl_book` VALUES (6, '计算机理论', 'Java核心技术 卷I 基础知识（原书第11版）', 'Core Java 第11版，Jolt大奖获奖作品，针对Java SE9、10、11全面更新');
INSERT INTO `tbl_book` VALUES (7, '计算机理论', '深入理解Java虚拟机', '5个维度全面剖析JVM，大厂面试知识点全覆盖');
```

```

INSERT INTO `tbl_book` VALUES (8, '计算机理论', 'Java编程思想（第4版）', 'Java学习必读经典,殿堂级著作！赢得了全球程序员的广泛赞誉');
INSERT INTO `tbl_book` VALUES (9, '计算机理论', '零基础学Java（全彩版）', '零基础自学编程的入门图书，由浅入深，详解Java语言的编程思想和核心技术');
INSERT INTO `tbl_book` VALUES (10, '市场营销', '直播就该这么做：主播高效沟通实战指南', '李子柒、李佳琦、薇娅成长为网红的秘密都在书中');
INSERT INTO `tbl_book` VALUES (11, '市场营销', '直播销讲实战一本通', '和秋叶一起学系列网络营销书籍');
INSERT INTO `tbl_book` VALUES (12, '市场营销', '直播带货：淘宝、天猫直播从新手到高手', '一本教你如何玩转直播的书，10堂课轻松实现带货月入3w+');

```

根据上述表结构，制作对应的实体类

## 实体类

```

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

```

实体类的开发可以自动通过工具手工生成get/set方法，然后覆盖toString()方法，方便调试，等等。不过这一套操作书写很繁琐，有对应的工具可以帮助我们简化开发，介绍一个小工具，lombok。

Lombok，一个Java类库，提供了一组注解，简化POJO实体类开发，SpringBoot目前默认集成了lombok技术，并提供了对应的版本控制，所以只需要提供对应的坐标即可，在pom.xml中添加lombok的坐标。

```

<dependencies>
    <!--lombok-->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
    </dependency>
</dependencies>

```

使用lombok可以通过一个注解@Data完成一个实体类对应的getter，setter，toString，equals，hashCode等操作的快速添加

```

import lombok.Data;

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

```

到这里实体类就做好了，是不是比不使用lombok简化好多，这种工具在Java开发中还有N多，后面课程中遇到了能用的东西时，在不增加各位小伙伴大量的学习时间的情况下，尽量多给大家介绍一些

## 总结

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

## 2.数据层开发——基础CRUD

数据层开发本次使用MyBatisPlus技术，数据源使用前面学习的Druid，学都学了都用上

**步骤①：**导入MyBatisPlus与Druid对应的starter，当然mysql的驱动不能少

```
<dependencies>
  <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>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

**步骤②：**配置数据库连接相关的数据源配置

```
server:
  port: 80

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

**步骤③：**使用MP的标准通用接口BaseMapper加速开发，别忘了@Mapper和泛型的指定

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

**步骤④：**制作测试类测试结果，这个测试类制作是个好习惯，不过在企业开发中往往都为加速开发跳过此步，且行且珍惜吧

```

package com.example.dao;

import com.baomidou.mybatisplus.core.conditions.query.LambdaQueryWrapper;
import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.baomidou.mybatisplus.core.metadata.IPage;
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.example.domain.Book;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
public class BookDaoTestCase {

    @Autowired
    private BookDao bookDao;

    @Test
    void testGetById(){
        System.out.println(bookDao.selectById(1));
    }

    @Test
    void testSave(){
        Book book = new Book();
        book.setType("测试数据123");
        book.setName("测试数据123");
        book.setDescription("测试数据123");
        bookDao.insert(book);
    }

    @Test
    void testUpdate(){
        Book book = new Book();
        book.setId(17);
        book.setType("测试数据abcdefg");
        book.setName("测试数据123");
        book.setDescription("测试数据123");
        bookDao.updateById(book);
    }

    @Test
    void testDelete(){
        bookDao.deleteById(16);
    }

    @Test
    void testGetAll(){
        bookDao.selectList(null);
    }
}

```

温馨提示

MP技术默认的主键生成策略为雪花算法，生成的主键ID长度较大，和目前的数据库设定规则不相符，需要配置一下使MP使用数据库的主键生成策略，方式嘛还是老一套，做配置。在application.yml中添加对应配置即可，具体如下

```
server:
  port: 80

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

mybatis-plus:
  global-config:
    db-config:
      table-prefix: tbl_          #设置表名通用前缀
      id-type: auto               #设置主键id字段的生成策略为参照数据库设定的策略，当前数据
                                #库设置id生成策略为自增
```

## 查看MP运行日志

在进行数据层测试的时候，因为基础的CRUD操作均由MP给我们提供了，所以就出现了一个局面，开发者不需要书写SQL语句了，这样程序运行的时候总有一种感觉，一切的一切都是黑盒的，作为开发者我们啥也不知道就完了。如果程序正常运行还好，如果报错了，这个时候就很崩溃，你甚至都不知道从何下手，因为传递参数、封装SQL语句这些操作完全不是你干预开发出来的，所以查看执行期运行的SQL语句就成为当务之急。

SpringBoot整合MP的时候充分考虑到了这点，通过配置的形式就可以查阅执行期SQL语句，配置如下

```
mybatis-plus:
  global-config:
    db-config:
      table-prefix: tbl_
      id-type: auto
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdoutImpl
```

再来看运行结果，此时就显示了运行期执行SQL的情况。

```
Creating a new SqlSession
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@2c9a6717] was
not registered for synchronization because synchronization is not active
JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@6ca30b8a] will not be managed
by Spring
==> Preparing: SELECT id,type,name,description FROM tbl_book
==> Parameters:
<==    Columns: id, type, name, description
<==      Row: 1, 计算机理论, Spring实战 第5版, Spring入门经典教程, 深入理解Spring原理技
术内幕
<==      Row: 2, 计算机理论, Spring 5核心原理与30个类手写实战, 十年沉淀之作, 手写Spring
精华思想
```

```

<==      Row: 3, 计算机理论, Spring 5 设计模式, 深入Spring源码剖析Spring源码中蕴含的10
大设计模式
<==      Row: 4, 计算机理论, Spring MVC+MyBatis开发从入门到项目实战, 全方位解析面向Web
应用的轻量级框架, 带你成为Spring MVC开发高手
<==      Row: 5, 计算机理论, 轻量级Java web企业应用实战, 源码级剖析Spring框架, 适合已掌
握Java基础的读者
<==      Row: 6, 计算机理论, Java核心技术 卷I 基础知识 (原书第11版), Core Java 第11
版, Jolt大奖获奖作品, 针对Java SE9、10、11全面更新
<==      Row: 7, 计算机理论, 深入理解Java虚拟机, 5个维度全面剖析JVM, 大厂面试知识点全覆盖
<==      Row: 8, 计算机理论, Java编程思想 (第4版), Java学习必读经典,殿堂级著作! 赢得了全
球程序员的广泛赞誉
<==      Row: 9, 计算机理论, 零基础学Java (全彩版), 零基础自学编程的入门图书, 由浅入深,
详解Java语言的编程思想和核心技术
<==      Row: 10, 市场营销, 直播就该这么做: 主播高效沟通实战指南, 李子柒、李佳琦、薇娅成长
为网红的秘密都在书中
<==      Row: 11, 市场营销, 直播销讲实战一本通, 和秋叶一起学系列网络营销书籍
<==      Row: 12, 市场营销, 直播带货: 淘宝、天猫直播从新手到高手, 一本教你如何玩转直播的
书, 10堂课轻松实现带货月入3w+
<==      Row: 13, 测试类型, 测试数据, 测试描述数据
<==      Row: 14, 测试数据update, 测试数据update, 测试数据update
<==      Row: 15, -----, 测试数据123, 测试数据123
<==      Total: 15

```

其中清晰的标注了当前执行的SQL语句是什么, 携带了什么参数, 对应的执行结果是什么, 所有信息应有尽有。

此处设置的是日志的显示形式, 当前配置的是控制台输出, 当然还可以由更多的选择, 根据需求切换即可

```

mybatis-plus:
  global-config:
    db-config:
      table-prefix: tbl_
  configuration:
    log-impl: |
      StdOutImpl (org.apache.ibatis.logging.stdout)
      JakartaCommonsLoggingImpl (org.apache.ibatis.log...
      Jdk14LoggingImpl (org.apache.ibatis.logging.jdk1...
      Log4j2AbstractLoggerImpl (org.apache.ibatis.logg...
      Log4j2Impl (org.apache.ibatis.logging.log4j2)
      Log4j2LoggerImpl (org.apache.ibatis.logging.log4...
      Log4jImpl (org.apache.ibatis.logging.log4j)
      NoLoggingImpl (org.apache.ibatis.logging.nologgi...
      Slf4jImpl (org.apache.ibatis.logging.slf4j)
      Slf4jLocationAwareLoggerImpl (org.apache.ibatis....
      Slf4jLoggerImpl (org.apache.ibatis.logging.slf4j)
      Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards Next Tip

```

## 总结

1. 手工导入starter坐标 (2个), mysql驱动 (1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
5. 使用配置方式开启日志，设置日志输出方式为标准输出即可查阅SQL执行日志

### 3.数据层开发——分页功能制作

前面仅仅是使用了MP提供的基础CRUD功能，实际上MP给我们提供了几乎所有的基础操作，这一节说一下如果实现数据库端的分页操作

MP提供的分页操作API如下

```
@Test
void testGetPage(){
    IPage page = new Page(2,5);
    bookDao.selectPage(page, null);
    System.out.println(page.getCurrent());
    System.out.println(page.getSize());
    System.out.println(page.getTotal());
    System.out.println(page.getPages());
    System.out.println(page.getRecords());
}
```

其中selectPage方法需要传入一个封装分页数据的对象，可以通过new的形式创建这个对象，当然这个对象也是MP提供的，别选错包了。创建此对象时就需要指定分页的两个基本数据

- 当前显示第几页
- 每页显示几条数据

可以通过创建Page对象时利用构造方法初始化这两个数据

```
IPage page = new Page(2,5);
```

将该对象传入到查询方法selectPage后，可以得到查询结果，但是我们会发现当前操作查询结果返回值仍然是一个IPage对象，这又是怎么回事？

```
IPage page = bookDao.selectPage(page, null);
```

原来这个IPage对象中封装了若干个数据，而查询的结果作为IPage对象封装的一个数据存在的，可以理解为查询结果得到后，又塞到了这个IPage对象中，其实还是为了高度的封装，一个IPage描述了分页所有的信息。下面5个操作就是IPage对象中封装的所有信息了



```

@Test
void testGetPage(){
    IPage page = new Page(2,5);
    bookDao.selectPage(page, null);
    System.out.println(page.getCurrent());    //当前页码值
    System.out.println(page.getSize());      //每页显示数
    System.out.println(page.getTotal());     //数据总量
    System.out.println(page.getPages());     //总页数
    System.out.println(page.getRecords());   //详细数据
}

```

到这里就知道这些数据如何获取了，但是当你去执行这个操作时，你会发现并不像我们分析的这样，实际上这个分页当前是无效的。为什么这样呢？这个要源于MP的内部机制。

对于MySQL的分页操作使用limit关键字进行，而并不是所有的数据库都使用limit关键字实现的，这个时候MP为了制作的兼容性强，将分页操作设置为基础查询操作的升级版，你可以理解为iPhone6与iPhone6S-PLUS的关系。

基础操作中有查询全部的功能，而在这个基础上只需要升级一下（PLUS）就可以得到分页操作。所以MP将分页操作做成了一个开关，你用分页功能就把开关开启，不用就不需要开启这个开关。而我们现在没有开启这个开关，所以分页操作是没有的。这个开关是通过MP的拦截器的形式存在的，其中的原理这里不分析了，有兴趣的小伙伴可以学习MyBatisPlus这门课程进行详细解读。具体设置方式如下

### 定义MP拦截器并将其设置为Spring管控的bean

```

@Configuration
public class MPConfig {
    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor(){
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        interceptor.addInnerInterceptor(new PaginationInnerInterceptor());
        return interceptor;
    }
}

```

上述代码第一行是创建MP的拦截器栈，这个时候拦截器栈中没有具体的拦截器，第二行是初始化了分页拦截器，并添加到拦截器栈中。如果后期开发其他功能，需要添加全新的拦截器，按照第二行的格式继续add进去新的拦截器就可以了。

### 总结

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

## 4.数据层开发——条件查询功能制作

除了分页功能，MP还提供有强大的条件查询功能。以往我们写条件查询要自己动态拼写复杂的SQL语句，现在简单了，MP将这些操作都制作成API接口，调用一个又一个的方法就可以实现各种套件的拼装。这里给大家普及一下基本格式，详细的操作还是到MP的课程中查阅吧

下面的操作就是执行一个模糊匹配对应的操作，由like条件书写变为了like方法的调用

```

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

```

其中第一句QueryWrapper对象是一个用于封装查询条件的对象，该对象可以动态使用API调用的方法添加条件，最终转化成对应的SQL语句。第二句就是一个条件了，需要什么条件，使用QueryWrapper对象直接调用对应操作即可。比如做大于小于关系，就可以使用lt或gt方法，等于使用eq方法，等等，此处不做更多的解释了。

这组API使用还是比较简单的，但是关于属性字段名的书写存在着安全隐患，比如查询字段name，当前是以字符串的形态书写的，万一写错，编译器还没有办法发现，只能将问题抛到运行器通过异常堆栈告诉开发者，不太友好。

MP针对字段检查进行了功能升级，全面支持Lambda表达式，就有了下面这组API。由QueryWrapper对象升级为LambdaQueryWrapper对象，这下就变了上述问题的出现

```

@Test
void testGetBy2(){
    String name = "1";
    LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();
    lqw.like(Book::getName,name);
    bookDao.selectList(lqw);
}

```

为了便于开发者动态拼写SQL，防止将null数据作为条件使用，MP还提供了动态拼装SQL的快捷书写方式

```

@Test
void testGetBy2(){
    String name = "1";
    LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();
    //if(name != null) lqw.like(Book::getName,name);           //方式一：JAVA代码控制
    lqw.like(name != null,Book::getName,name);                 //方式二：API接口提供控制开关
    bookDao.selectList(lqw);
}

```

其实就是个格式，没有区别。关于MP的基础操作就说到这里吧，如果这一块知识不太熟悉的小伙伴还是去完整的学习一下MP的知识吧，这里只是蜻蜓点水的用了几个操作而已。

## 总结

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

## 5.业务层开发

数据层开发告一段落，下面进行业务层开发，其实标准业务层开发很多初学者认为就是调用数据层，怎么说呢？这个理解是没有大问题的，更精准的说法应该是**组织业务逻辑功能，并根据业务需求，对数据持久层发起调用**。有什么差别呢？目标是为了组织出符合需求的业务逻辑功能，至于调不调用数据层还真不好说，有需求就调用，没有需求就不调用。

一个常识性的知识普及一下，业务层的方法名定义一定要与业务有关，例如登录操作

```
login(String username,String password);
```

而数据层的方法名定义一定与业务无关，是一定，不是可能，也不是有可能，例如根据用户名密码查询

```
selectByUserNameAndPassword(String username,String password);
```

我们在开发的时候是可以根据完成的工作不同划分成不同职能的开发团队的。比如一个哥们制作数据层，他就可以不知道业务是什么样子，拿到的需求文档要求可能是这样的

接口：传入用户名与密码字段，查询出对应结果，结果是单条数据

接口：传入ID字段，查询出对应结果，结果是单条数据

接口：传入离职字段，查询出对应结果，结果是多条数据

但是进行业务功能开发的哥们，拿到的需求文档要求差别就很大

接口：传入用户名与密码字段，对用户名字段做长度校验，4-15位，对密码字段做长度校验，8到24位，对喵喵字段做特殊字符校验，不允许存在空格，查询结果为对象。如果为null，返回BusinessException，封装消息码INFO\_LOGON\_USERNAME\_PASSWORD\_ERROR

你比较一下，能是一回事吗？差别太大了，所以说业务层方法定义与数据层方法定义差异化很大，只不过有些入门级的开发者手懒或者没有使用过公司相关的ISO标准化文档而已。

多余的话不说了，咱们做案例就简单制作了，业务层接口定义如下：

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

业务层实现类如下，转调数据层即可

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

```

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

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

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

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

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

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

```

别忘了对业务层接口进行测试，测试类如下

```

@SpringBootTest
public class BookServiceTest {
    @Autowired
    private IBookService bookService;

    @Test
    void testGetById(){
        System.out.println(bookService.getById(4));
    }

    @Test
    void testSave(){
        Book book = new Book();
        book.setType("测试数据123");
        book.setName("测试数据123");
        book.setDescription("测试数据123");
        bookService.save(book);
    }

    @Test
    void testUpdate(){
        Book book = new Book();
        book.setId(17);
    }
}

```

```

        book.setType("-----");
        book.setName("测试数据123");
        book.setDescription("测试数据123");
        bookService.updateById(book);
    }

    @Test
    void testDelete(){
        bookService.removeById(18);
    }

    @Test
    void testGetAll(){
        bookService.list();
    }

    @Test
    void testGetPage(){
        IPage<Book> page = new Page<Book>(2,5);
        bookService.page(page);
        System.out.println(page.getCurrent());
        System.out.println(page.getSize());
        System.out.println(page.getTotal());
        System.out.println(page.getPages());
        System.out.println(page.getRecords());
    }
}

```

## 总结

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

## 业务层快速开发

其实MP技术不仅提供了数据层快速开发方案，业务层MP也给了一个通用接口，个人观点不推荐使用，凑合能用吧，其实就是一个封装+继承的思想，代码给出，实际开发慎用

### 业务层接口快速开发

```

public interface IBookService extends IService<Book> {
    //添加非通用操作API接口
}

```

业务层接口实现类快速开发，关注继承的类需要传入两个泛型，一个是数据层接口，另一个是实体类

```

@Service
public class BookServiceImpl extends ServiceImpl<BookDao, Book> implements
IBookService {
    @Autowired
    private BookDao bookDao;
    //添加非通用操作API
}

```

如果感觉MP提供的功能不足以支撑你的使用需要，其实是一定不能支撑的，因为需求不可能是通用的，在原始接口基础上接着定义新的API接口就行了，此处不再说太多了，就是自定义自己的操作了，但是不要和已有的API接口名冲突即可。

## 总结

1. 使用通用接口 (IService) 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

## 6.表现层开发

终于做到表现层了，做了这么多都是基础工作。其实你现在回头看看，哪里还有什么SpringBoot的影子？前面1,2步就搞完了。继续完成表现层制作吧，咱们表现层的开发使用基于Restful的表现层接口开发，功能测试通过Postman工具进行

表现层接口如下：

```
@RestController
@RequestMapping("/books")
public class BookController2 {

    @Autowired
    private IBookService bookService;

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

    @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);
    }

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

    @GetMapping("/{currentPage}/{pageSize}")
```

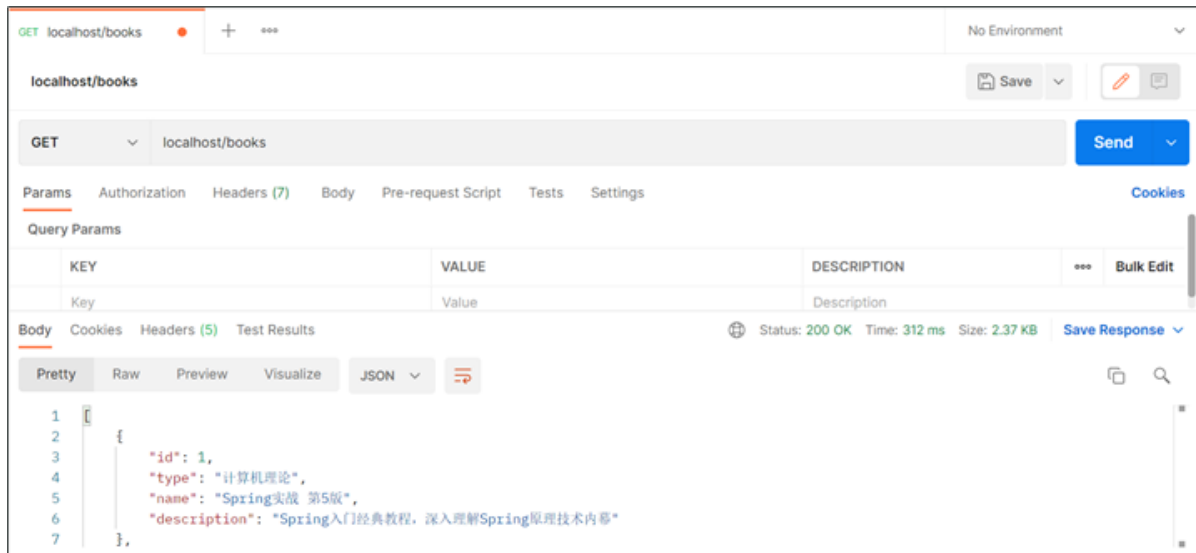
```

public IPage<Book> getPage(@PathVariable int currentPage,@PathVariable int
pageSize){
    return bookService.getPage(currentPage,pagesize, null);
}
}

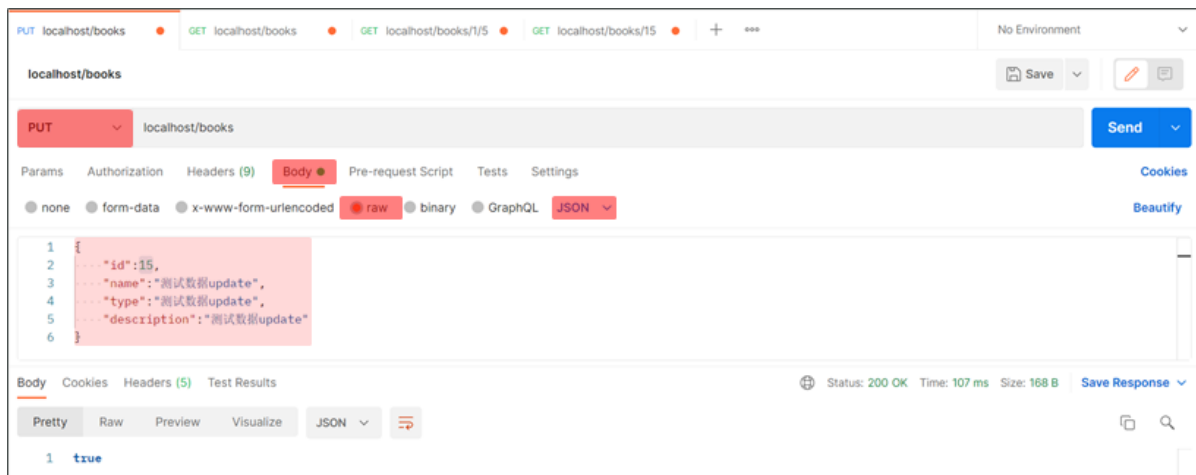
```

在实用Postman测试时关注提交类型，对应上即可，不然就会报405的错误码了

## 普通GET请求



## PUT请求传递json数据，后台实用@RequestBody接收数据



## GET请求传递路径变量，后台实用@PathVariable接收数据

## 总结

### 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



## 7.表现层消息一致性处理

目前我们通过Postman测试后业务层接口功能时通的，但是这样的结果给到前端开发者会出现一个小问题。不同的操作结果所展示的数据格式差异化严重

### 增删改操作结果

```
true
```

### 查询单个数据操作结果

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

### 查询全部数据操作结果

```
[
  {
    "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"
  }
]
```

每种不同操作返回的数据格式都不一样，而且还不知道以后还会有什么格式，这样的结果让前端人员看了是很容易让人崩溃的，必须将所有操作的操作结果数据格式统一起来，需要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;
}
```

其中flag用于标识操作是否成功，data用于封装操作数据，现在的数据格式就变了

```
{
  "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"
  }
}
```

表现层开发格式也需要转换一下

```
@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);
    }
}
```

```
@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);
    }
}
```

```
@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);
    }
}
```

结果这么一折腾，全格式统一，现在后端发送给前端的数据格式就统一了，免去了不少前端解析数据的麻烦。

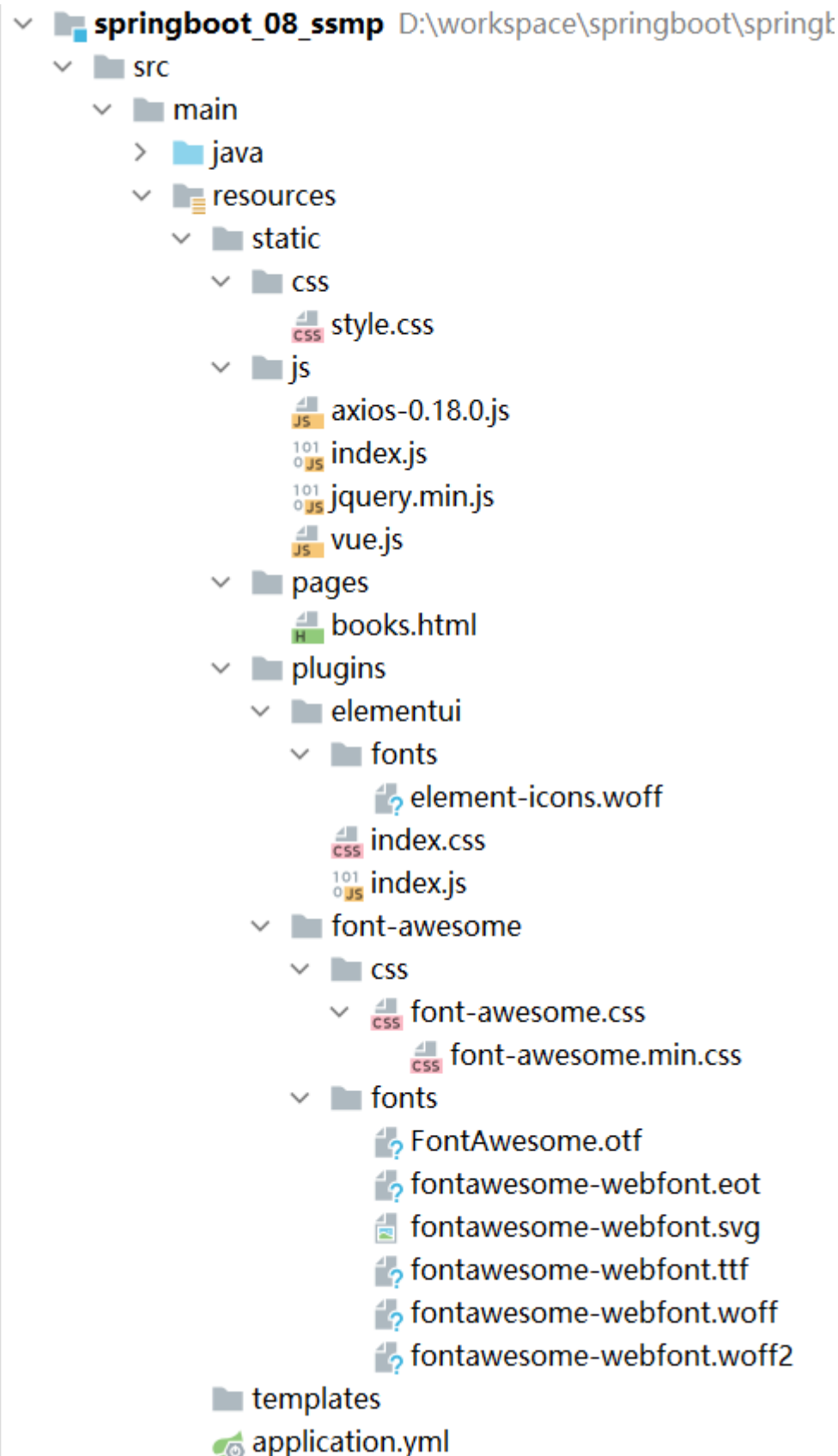
## 总结

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

## 8.前后端联通性测试

后端的表现层接口开发完毕，就可以进行前端的开发了。

将前端人员开发的页面保存到resources目录下的static目录中，建议执行maven的clean生命周期，避免缓存的问题出现。



在进行具体的功能开发之前，先做联通性的测试，通过页面发送异步提交（axios），这一步调试通过后再进行进一步的功能开发

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

只要后台代码能够正常工作，前端能够在日志中接收到数据，就证明前后端是通的，也就可以进行下一步的功能开发了

## 总结

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

## 9.页面基础功能开发

### F-1.列表功能（非分页版）

列表功能主要操作就是加载完数据，将数据展示到页面上，此处要利用VUE的数据模型绑定，发送请求得到数据，然后页面上读取指定数据即可

#### 页面数据模型定义

```
data:{
  dataList: [],//当前页要展示的列表数据
  ...
},
```

#### 异步请求获取数据

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

这样在页面加载时就可以获取到数据，并且由VUE将数据展示到页面上了

总结：

1. 将查询数据返回到页面，利用前端数据绑定进行数据展示

### F-2.添加功能

添加功能用于收集数据的表单是通过一个弹窗展示的，因此在添加操作前首先要进行弹窗的展示，添加后隐藏弹窗即可。因为这个弹窗一直存在，因此当页面加载时首先设置这个弹窗为不可显示状态，需要展示，切换状态即可

#### 默认状态

```
data:{
  dialogFormVisible: false,//添加表单是否可见
  ...
},
```

#### 切换为显示状态

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

由于每次添加数据都是使用同一个弹窗录入数据，所以每次操作的痕迹将在下一次操作时展示出来，需要在每次操作之前清理掉上次操作的痕迹

### 定义清理数据操作

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

### 切换弹窗状态时清理数据

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

至此准备工作完成，下面就要调用后台完成添加操作了

### 添加操作

```
//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();
    });
},
```

1. 将要保存的数据传递到后台，通过post请求的第二个参数传递json数据到后台
2. 根据返回的操作结果决定下一步操作
  - 如果是true就关闭添加窗口，显示添加成功的消息
  - 如果是false保留添加窗口，显示添加失败的消息
3. 无论添加是否成功，页面均进行刷新，动态加载数据（对getAll操作发起调用）

### 取消添加操作

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

## 总结

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

## F-3.删除功能

模仿添加操作制作删除功能，差别之处在于删除操作仅传递一个待删除的数据id到后台即可

### 删除操作

```
// 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();
  });
},
```

### 删除操作提示信息

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

```
        this.$message.info("取消删除操作");
    });
},
```

## 总结

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

## F-4.修改功能

修改功能可以说是列表功能、删除功能与添加功能的合体。几个相似点如下：

1. 页面也需要有一个弹窗用来加载修改的数据，这一点与添加相同，都是要弹窗
2. 弹出窗口中要加载待修改的数据，而数据需要通过查询得到，这一点与查询全部相同，都是要查数据
3. 查询操作需要将要修改的数据id发送到后台，这一点与删除相同，都是传递id到后台
4. 查询得到数据后需要展示到弹窗中，这一点与查询全部相同，都是要通过数据模型绑定展示数据
5. 修改数据时需要将被修改的数据传递到后台，这一点与添加相同，都是要传递数据

所以整体上来看，修改功能就是前面几个功能的大合体

### 查询并展示数据

```
//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");
        }
    });
},
```

### 修改操作

```
//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();
    });
},

```

## 总结

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（同删除与查询全部）
2. 利用前端双向数据绑定将查询到的数据进行回显（同查询全部）
3. 请求方式使用PUT调用后台对应操作（同新增传递数据）
4. 修改操作结束后动态刷新页面加载数据（同新增）
5. 根据操作结果不同，显示对应的提示信息（同新增）

## 10.业务消息一致性处理

目前的功能制作基本上达成了正常使用情况，什么叫正常使用呢？也就是这个程序不出BUG，如果我们搞一个BUG出来，你会发现程序马上崩溃掉。比如后台手工抛出一个异常，看看前端接收到的数据什么样子

```

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

```

面对这种情况，前端的同学又不会了，这又是什么格式？怎么和之前的格式不一样？

```

{
  "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"
  }
}

```

看来不仅要正确的操作数据格式做处理，还要对错误的操作数据格式做同样的格式处理

首先在当前的数据结果中添加消息字段，用来兼容后台出现的操作消息

```

@Data
public class R {
    private Boolean flag;
    private Object data;
    private String msg;    //用于封装消息
}

```

后台代码也要根据情况做处理，当前是模拟的错误

```

@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-_-!");
}

```

然后在表现层做统一的异常处理，使用SpringMVC提供的异常处理器做统一的异常处理

```

@RestControllerAdvice
public class ProjectExceptionAdvice {
    @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！");
    }
}

```

页面上得到数据后，先判定是否有后台传递过来的消息，标志就是当前操作是否成功，如果返回操作结果false，就读取后台传递的消息

```

//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); //消息来自于后台传递过来，而非
            固定内容
        }
    }).finally(()=>{
        this.getAll();
    });
},

```

## 总结

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

## 11. 页面功能开发

### F-5. 分页功能

分页功能的制作用于替换前面的查询全部，其中要使用到elementUI提供的分页组件

```
<!--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>
```

为了配合分页组件，封装分页对应的数据模型

```
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
  }
},
```

修改查询全部功能为分页查询，通过路径变量传递页码信息参数

```
getAll() {

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

后台提供对应的分页功能

```
@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);
}
```

页面根据分页操作结果读取对应数据，并进行数据模型绑定

```

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;
    });
},

```

对切换页码操作设置调用当前分页操作

```

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

```

## 总结

1. 使用el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

## F-6.删除功能维护

由于使用了分页功能，当最后一页只有一条数据时，删除操作就会出现BUG，最后一页无数据但是独立展示，对分页查询功能进行后台功能维护，如果当前页码值大于最大页码值，重新执行查询。其实这个问题解决方案很多，这里给出比较简单的一种处理方案

```

@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);
}

```

## F-7.条件查询功能

最后一个功能来做条件查询，其实条件查询可以理解为分页查询的时候除了携带分页数据再多带几个数据的查询。这些多带的数据就是查询条件。比较一下不带条件的分页查询与带条件的分页查询差别之处，这个功能就好做了

- 页面封装的数据：带不带条件影响的仅仅是一次性传递到后台的数据总量，由传递2个分页相关的数据转换成2个分页数据加若干个条件

- 后台查询功能：查询时由不带条件，转换成带条件，反正不带条件的时候查询条件对象使用的是 null，现在换成具体条件，差别不大
  - 查询结果：不管带不带条件，出来的数据只是有数量上的差别，其他都差别，这个可以忽略
- 经过上述分析，看来需要在页面发送请求的格式方面做一定的修改，后台的调用数据层操作时发送修改，其他没有区别

页面发送请求时，两个分页数据仍然使用路径变量，其他条件采用动态拼装url参数的形式传递

### 页面封装查询条件字段

```
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: ""
},
```

页面添加查询条件字段对应的数据模型绑定名称

```
<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>
```

将查询条件组织成url参数，添加到请求url地址中，这里可以借助其他类库快速开发，当前使用手工形式拼接，降低学习要求

```
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;
  });
},
```

后台代码中定义实体类封装查询条件

```

@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);
}

```

对应业务层接口与实现类进行修正

```

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.get
Name());

        lqw.like(Strings.isNotEmpty(queryBook.getType()),Book::getType,queryBook.get
Type());

        lqw.like(Strings.isNotEmpty(queryBook.getDescription()),Book::getDescription
,queryBook.getDescription());
        return bookDao.selectPage(page,lqw);
    }
}

```

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;
    });
},

```

总结

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