

## SpringBoot

课程主要内容：

1. springBoot基本应用
2. Springboot原理深入及源码剖析
3. Springboot数据访问
4. 针对ssm框架整合进行改造（集成springboot）

# 1. SpringBoot基本应用

## 1.1 约定优于配置

Build Anything with Spring Boot: Spring Boot is the starting point for building all Spring-based applications. Spring Boot is designed to get you up and running as quickly as possible, with minimal upfront configuration of Spring.

上面是引自官网的一段话，大概是说：Spring Boot 是所有基于 Spring 开发的项目的起点。Spring Boot 的设计是为了让你尽可能快的跑起来 Spring 应用程序并且尽可能减少你的配置文件。

约定优于配置（Convention over Configuration），又称按约定编程，是一种软件设计范式。

本质上是说，系统、类库或框架应该假定合理的默认值，而非要求提供不必要的配置。比如说模型中有一个名为User的类，那么数据库中对应的表就会默认命名为user。只有在偏离这一个约定的时候，例如想要将该表命名为person，才需要写有关这个名字的配置。

比如平时架构师搭建项目就是限制软件开发随便写代码，制定出一套规范，让开发人员按统一的要求进行开发编码测试之类的，这样就加强了开发效率与审查代码效率。所以说写代码的时候就需要按要求命名，这样统一规范的代码就有良好的可读性与维护性了

约定优于配置简单来理解，就是遵循约定

## 1.2 SpringBoot概念

### 1.2.1 spring优缺点分析

**优点：**

spring是Java企业版(Java Enterprise Edition, JEE, 也称J2EE)的轻量级代替品。无需开发重量级的Enterprise JavaBean(EJB), Spring为企业级Java开发提供了一种相对简单的方法，通过依赖注入和面向切面编程，用简单的Java对象(Plain Old Java Object, POJO)实现了EJB的功能

**缺点：**

虽然Spring的组件代码是轻量级的，但它的配置却是重量级的。一开始，Spring用XML配置，而且是很多XML配置。Spring 2.5引入了基于注解的组件扫描，这消除了大量针对应用程序自身组件的显式XML配置。Spring 3.0引入了基于Java的配置，这是一种类型安全的可重构配置方式，可以代替XML。

所有这些配置都代表了开发时的损耗。因为在思考Spring特性配置和解决业务问题之间需要进行思维切换，所以编写配置挤占了编写应用程序逻辑的时间。和所有框架一样，Spring实用，但与此同时它要求的回报也不少。

除此之外，项目的依赖管理也是一件耗时耗力的事情。在环境搭建时，需要分析要导入哪些库的坐标，而且还需要分析导入与之有依赖关系的其他库的坐标，一旦选错了依赖的版本，随之而来的不兼容问题就会严重阻碍项目的开发进度

### 1.2.2 SpringBoot解决上述spring问题

SpringBoot对上述Spring的缺点进行的改善和优化，基于约定优于配置的思想，可以让开发人员不必在配置与逻辑业务之间进行思维的切换，全身心的投入到逻辑业务的代码编写中，从而大大提高了开发的效率，一定程度上缩短了项目周期

#### 起步依赖

起步依赖本质上是一个Maven项目对象模型(Project Object Model, POM)，定义了对其他库的传递依赖，这些东西加在一起即支持某项功能。

简单的说，起步依赖就是将具备某种功能的坐标打包到一起，并提供一些默认的功能。

#### 自动配置

springboot的自动配置，指的是springboot，会自动将一些配置类的bean注册进ioc容器，我们可以需要的地方使用@Autowired或者@Resource等注解来使用它。

“自动”的表现形式就是我们只需要引我们想用功能的包，相关的配置我们完全不用管，springboot会自动注入这些配置bean，我们直接使用这些bean即可

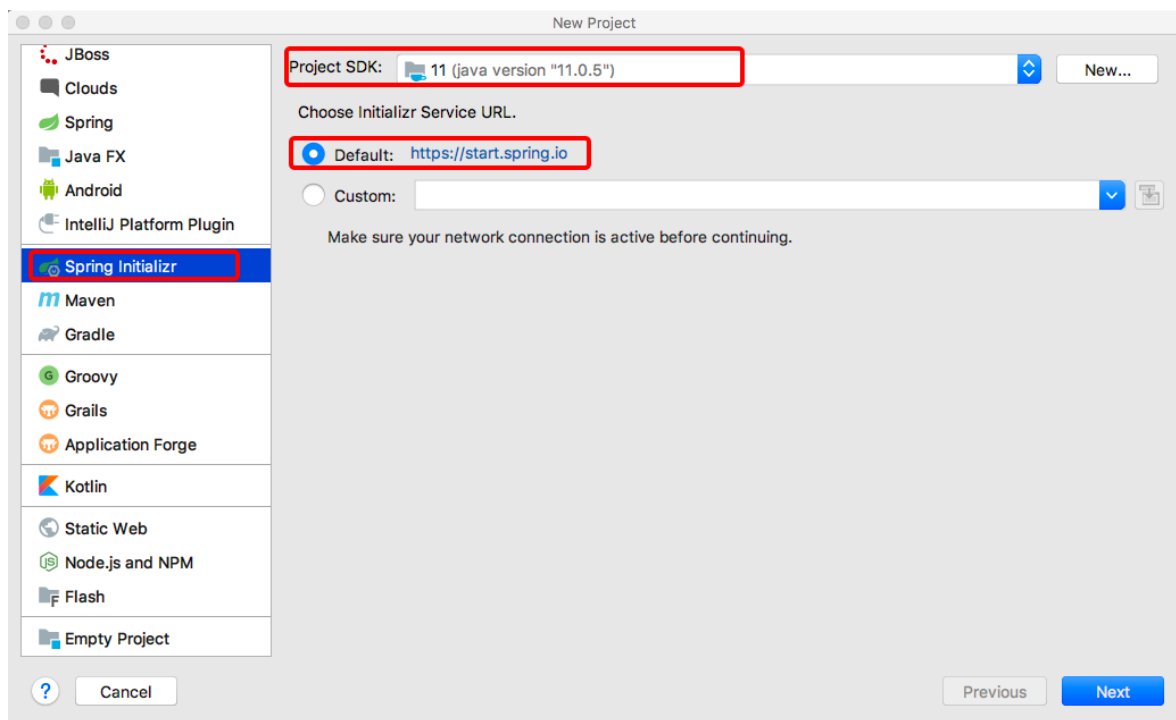
springboot: 简单、快速、方便地搭建项目；对主流开发框架的无配置集成；极大提高了开发、部署效率

## 1.3 SpringBoot 案例实现

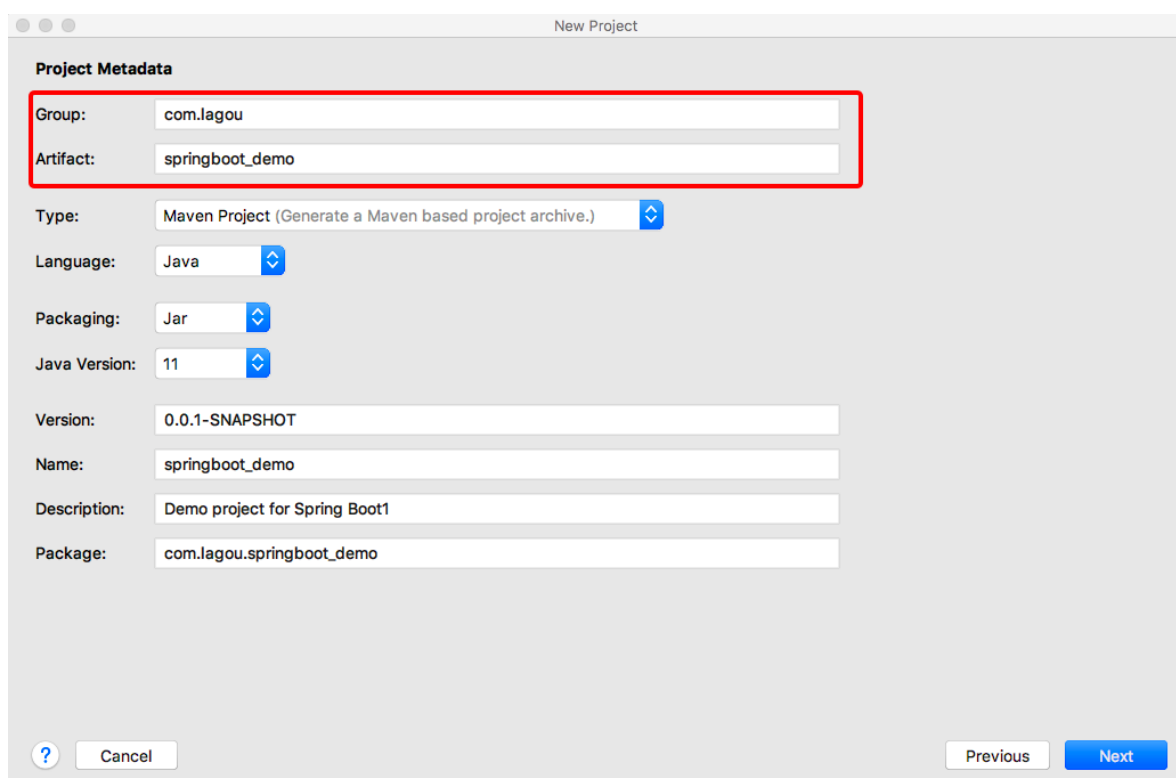
案例需求：请求Controller中的方法，并将返回值响应到页面

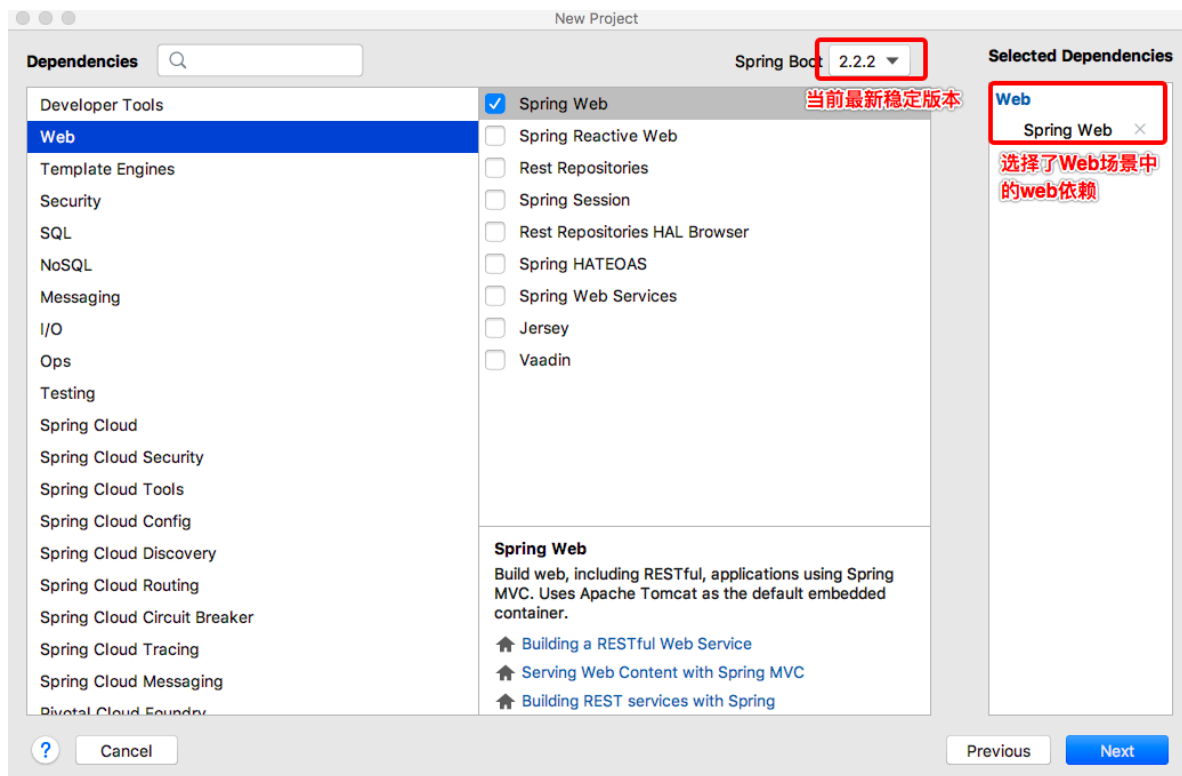
### (1) 使用Spring Initializr方式构建Spring Boot项目

本质上说，Spring Initializr是一个Web应用，它提供了一个基本的项目结构，能够帮助我们快速构建一个基础的Spring Boot项目

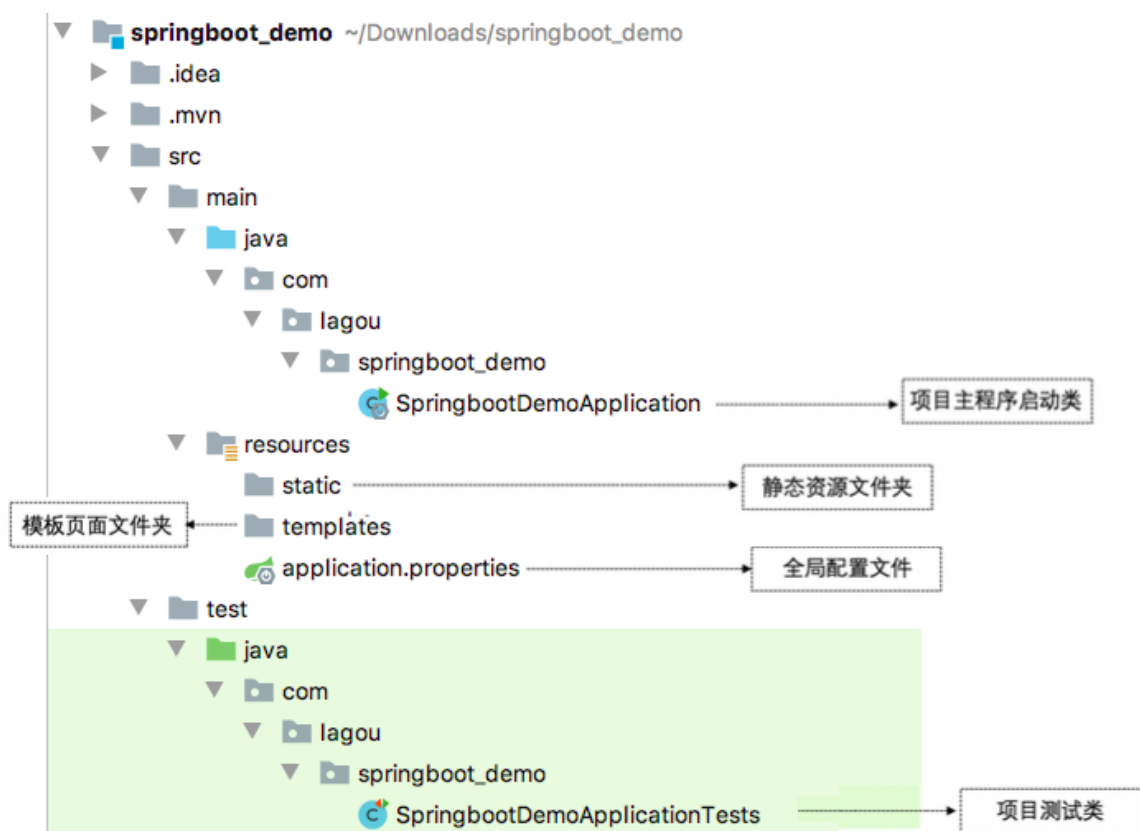


Project SDK”用于设置创建项目使用的JDK版本，这里，使用之前初始化设置好的JDK版本即可；在“Choose Initializr Service URL（选择初始化服务地址）”下使用默认的初始化服务地址“<https://start.spring.io>”进行Spring Boot项目创建（注意使用快速方式创建Spring Boot项目时，所在主机须在联网状态下）





Spring Boot项目就创建好了。创建好的Spring Boot项目结构如图：



使用Spring Initializr方式构建的Spring Boot项目会默认生成项目启动类、存放前端静态资源和页面的文件夹、编写项目配置的配置文件以及进行项目单元测试的测试类

## (2) 创建一个用于Web访问的Controller

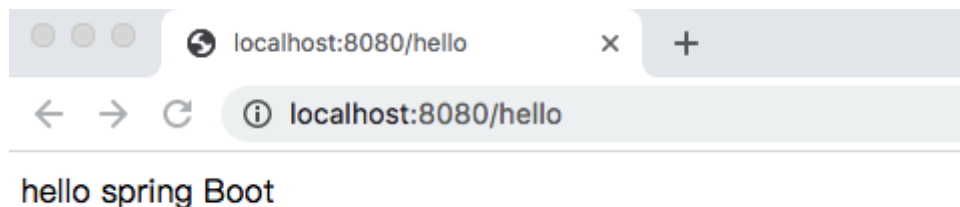
com.lagou包下创建名称为controller的包，在该包下创建一个请求处理控制类HelloController，并编写一个请求处理方法（注意：将项目启动类SpringBootDemoApplication移动到com.lagou包下）

```
@RestController // 该注解为组合注解，等同于Spring中@Controller+@ResponseBody注解
public class DemoController {

    @RequestMapping("/demo")
    public String demo(){
        return "hello spring Boot";
    }
}
```

### (3) 运行项目

运行主程序启动类SpringbootDemoApplication，项目启动成功后，在控制台上会发现Spring Boot项目默认启动的端口号为8080，此时，可以在浏览器上访问“<http://localhost:8080/hello>”



页面输出的内容是“hello Spring Boot”，至此，构建Spring Boot项目就完成了

附：解决中文乱码：

解决方法一：

```
@RequestMapping(produces = "application/json; charset=utf-8")
```

解决方法二：

```
#设置响应为utf-8
spring.http.encoding.force-response=true
```

## 1.4 单元测试与热部署

### (1) 单元测试

开发中，每当完成一个功能接口或业务方法的编写后，通常都会借助单元测试验证该功能是否正确。Spring Boot对项目的单元测试提供了很好的支持，在使用时，需要提前在项目的pom.xml文件中添加spring-boot-starter-test测试依赖启动器，可以通过相关注解实现单元测试

演示：

1. 添加spring-boot-starter-test测试依赖启动器

在项目的pom.xml文件中添加spring-boot-starter-test测试依赖启动器，示例代码如下：

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

注意：使用Spring Initializr方式搭建的Spring Boot项目，会自动加入spring-boot-starter-test测试依赖启动器，无需再手动添加

## 2. 编写单元测试类和测试方法

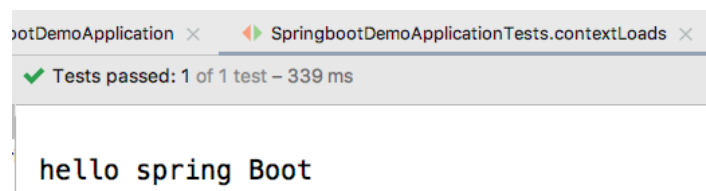
使用Spring Initializr方式搭建的Spring Boot项目，会在src.test.java测试目录下自动创建与项目主程序启动类对应的单元测试类

```
@RunWith(SpringRunner.class) // 测试启动器，并加载Spring Boot测试注解
@SpringBootTest // 标记为Spring Boot单元测试类，并加载项目的ApplicationContext上下文环境
class SpringbootDemoApplicationTests {

    @Autowired
    private DemoController demoController;

    // 自动创建的单元测试方法实例
    @Test
    void contextLoads() {
        String demo = demoController.demo();
        System.out.println(demo);
    }
}
```

上述代码中，先使用@Autowired注解注入了DemoController实例对象，然后在contextLoads()方法中调用了DemoController类中对应的请求控制方法contextLoads()，并输出打印结果



## (2) 热部署

在开发过程中，通常会对一段业务代码不断地修改测试，在修改之后往往需要重启服务，有些服务需要加载很久才能启动成功，这种不必要的重复操作极大的降低了程序开发效率。为此，Spring Boot框架专门提供了进行热部署的依赖启动器，用于进行项目热部署，而无需手动重启项目

演示：

### 1. 添加spring-boot-devtools热部署依赖启动器

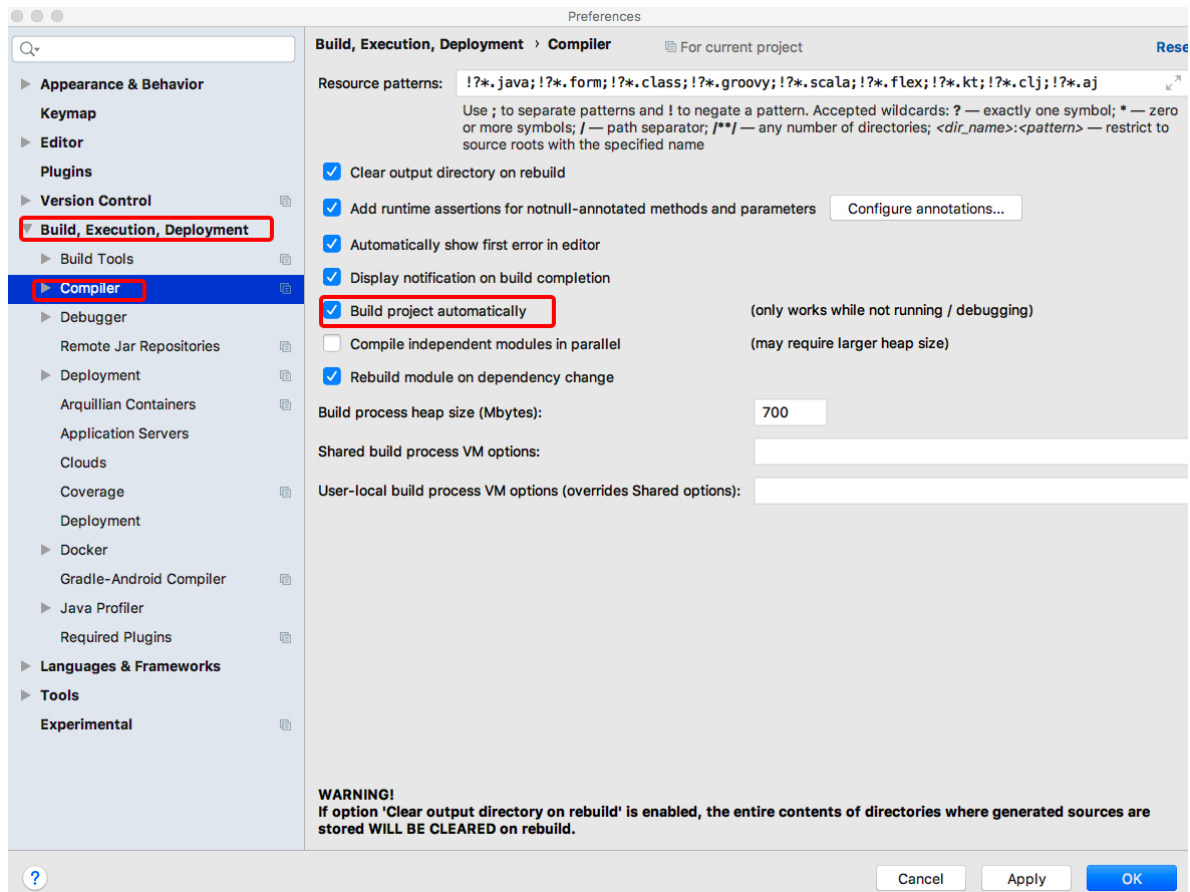
在Spring Boot项目进行热部署测试之前，需要先在项目的pom.xml文件中添加spring-boot-devtools热部署依赖启动器：

```
<!-- 引入热部署依赖 -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

由于使用的是IDEA开发工具，添加热部署依赖后可能没有任何效果，接下来还需要针对IDEA开发工具进行热部署相关的功能设置

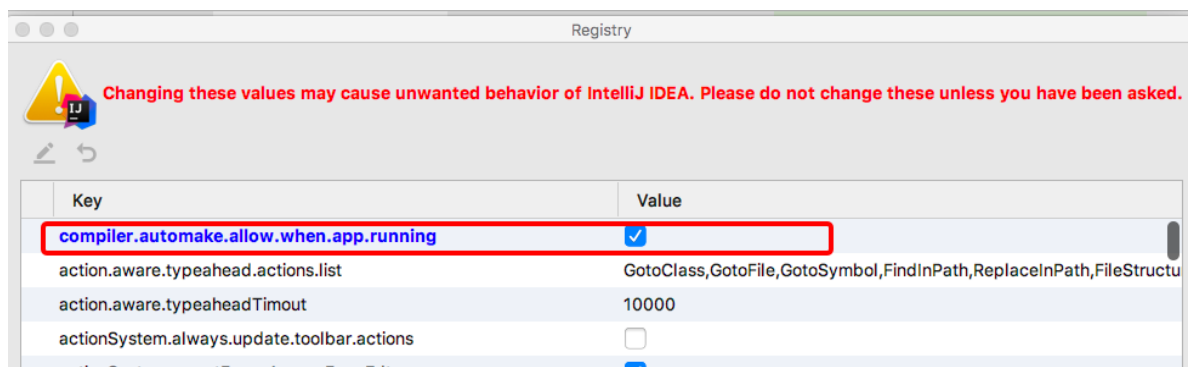
## 2. IDEA工具热部署设置

选择IDEA工具界面的【File】->【Settings】选项，打开Compiler面板设置页面



选择Build下的Compiler选项，在右侧勾选“Build project automatically”选项将项目设置为自动编译，单击【Apply】→【OK】按钮保存设置

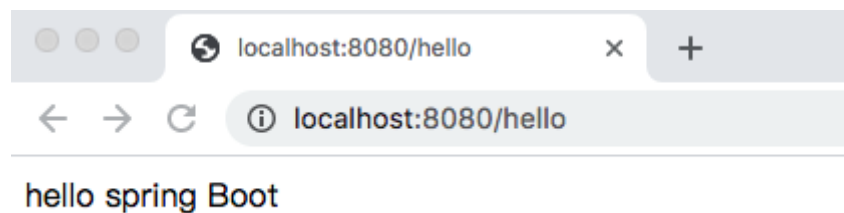
在项目任意页面中使用组合快捷键“Ctrl+Shift+Alt+/'”打开Maintenance选项框，选中并打开Registry页面，具体如图1-17所示



列表中找到“compiler.automake.allow.when.app.running”，将该选项后的Value值勾选，用于指定IDEA工具在程序运行过程中自动编译，最后单击【Close】按钮完成设置

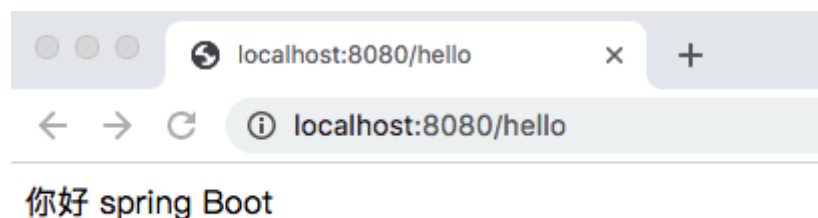
### 3. 热部署效果测试

启动chapter01<http://localhost:8080/hello>



页面原始输出的内容是“hello Spring Boot”。

为了测试配置的热部署是否有效，接下来，在不关闭当前项目的情况下，将DemoController 类中的请求处理方法hello()的返回值修改为“你好，Spring Boot”并保存，查看控制台信息会发现项目能够自动构建和编译，说明项目热部署生效



可以看出，浏览器输出了“你好，Spring Boot”，说明项目热部署配置成功

## 1.5 全局配置文件

全局配置文件能够对一些

默认配置值进行修改。Spring Boot使用一个application.properties或者application.yaml的文件作为全局配置文件，该文件存放在src/main/resource目录或者类路径的/config，一般会选择resource目录。接下来，将针对这两种全局配置文件进行讲解：

### 1.5.1 application.properties配置文件

使用Spring Initializr方式构建Spring Boot项目时，会在resource目录下自动生成一个空的application.properties文件，Spring Boot项目启动时会自动加载application.properties文件。

我们可以在application.properties文件中定义Spring Boot项目的相关属性，当然，这些相关属性可以是系统属性、环境变量、命令参数等信息，也可以是自定义配置文件名称和位置

```
server.port=8081
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.config.additional-location=
spring.config.location=
spring.config.name=application
```

接下来，通过一个案例对Spring Boot项目中application.properties配置文件的具体使用进行讲解

演示：



预先准备了两个实体类文件，后续会演示将application.properties配置文件中的自定义配置属性注入到Person实体类的对应属性中

(1) 先在项目的com.lagou包下创建一个pojo包，并在该包下创建两个实体类Pet和Person

```
public class Pet {  
  
    private String type;  
    private String name;  
    // 省略属性getXX()和setXX()方法  
    // 省略toString()方法  
  
}
```

```
@Component    //用于将Person类作为Bean注入到Spring容器中  
@ConfigurationProperties(prefix = "person") //将配置文件中以person开头的属性注入到该类  
中  
public class Person {  
  
    private int id;           //id  
    private String name;      //名称  
    private List hobby;       //爱好  
    private String[] family;  //家庭成员  
    private Map map;  
    private Pet pet;          //宠物  
    // 省略属性getXX()和setXX()方法  
    // 省略toString()方法  
  
}
```

@ConfigurationProperties(prefix = "person")注解的作用是将配置文件中以person开头的属性值通过setXX()方法注入到实体类对应属性中

@Component注解的作用是将当前注入属性值的Person类对象作为Bean组件放到Spring容器中，只有这样才能被@ConfigurationProperties注解进行赋值

(2) 打开项目的resources目录下的application.properties配置文件，在该配置文件中编写需要对Person类设置的配置属性



```
application.properties x pom.xml x DemoController.java x  
person.id=1  
person.name=tom  
person.hobby=吃饭,睡觉,打豆豆  
person.family=father,mother  
person.map.k1=v1  
person.map.k2=v2  
person.pet.type=dag  
person.pet.name=旺财
```

编写application.properties配置文件时，由于要配置的Person对象属性是我们自定义的，Spring Boot无法自动识别，所以不会有任何书写提示。在实际开发中，为了出现代码提示的效果来方便配置，在使用@ConfigurationProperties注解进行配置文件属性值注入时，可以在pom.xml文件中添加一个Spring Boot提供的配置处理器依赖：

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

在pom.xml中添加上述配置依赖后，还需要重新运行项目启动类或者使用“Ctrl+F9”快捷键（即Build Project）重构当前Spring Boot项目方可生效

(3) 查看application.properties配置文件是否正确，同时查看属性配置效果，打开通过IDEA工具创建的项目测试类，在该测试类中引入Person实体类Bean，并进行输出测试

```
@RunWith(SpringRunner.class) // 测试启动器，并加载Spring Boot测试注解
@SpringBootTest // 标记为Spring Boot单元测试类，并加载项目的ApplicationContext上下文环境
class SpringbootDemoApplicationTests {

    // 配置测试
    @Autowired
    private Person person;

    @Test
    void configurationTest() {
        System.out.println(person);
    }
}
```

打印结果：

```
Person{id=1, name='tom', hobby=[吃饭, 睡觉, 打豆豆], family=[father, mother], map={k1=v1, k2=v2}, pet=Pet{type='dog', name='旺财'}}
```

可以看出，测试方法configurationTest()运行成功，同时正确打印出了Person实体类对象。至此，说明application.properties配置文件属性配置正确，并通过相关注解自动完成了属性注入

## 1.5.2 application.yml配置文件

YAML文件格式是Spring Boot支持的一种JSON超集文件格式，相较于传统的Properties配置文件，YAML文件以数据为核心，是一种更为直观且容易被电脑识别的数据序列化格式。application.yml配置文件的工作原理和application.properties是一样的，只不过yaml格式配置文件看起来更简洁一些。

- YAML文件的扩展名可以使用.yml或者.yaml。
- application.yml文件使用“key: (空格) value”格式配置属性，使用缩进控制层级关系。

这里，针对不同数据类型的属性值，介绍一下YAML

(1) value值为普通数据类型（例如数字、字符串、布尔等）

当YAML配置文件中配置的属性值为普通数据类型时，可以直接配置对应的属性值，同时对于字符串类型的属性值，不需要额外添加引号，示例代码如下

```
server:
  port: 8080
  servlet:
    context-path: /hello
```

## (2) value值为数组和单列集合

当YAML配置文件中配置的属性值为数组或单列集合类型时，主要有两种书写方式：缩进式写法和行内式写法。

其中，缩进式写法还有两种表示形式，示例代码如下

```
person:
  hobby:
    - play
    - read
    - sleep
```

或者使用如下示例形式

```
person:
  hobby:
    play,
    read,
    sleep
```

上述代码中，在YAML配置文件中通过两种缩进式写法对person对象的单列集合（或数组）类型的爱好hobby赋值为play、read和sleep。其中一种形式为“-（空格）属性值”，另一种形式为多个属性值之前加英文逗号分隔（注意，最后一个属性值后不要加逗号）。

```
person:
  hobby: [play,read,sleep]
```

通过上述示例对比发现，YAML配置文件的行内式写法更加简明、方便。另外，包含属性值的中括号“[]”还可以进一步省略，在进行属性赋值时，程序会自动匹配和校对

## (3) value值为Map集合和对象

当YAML配置文件中配置的属性值为Map集合或对象类型时，YAML配置文件格式同样可以分为两种书写方式：缩进式写法和行内式写法。

其中，缩进式写法的示例代码如下

```
person:
  map:
    k1: v1
    k2: v2
```

对应的行内式写法示例代码如下

```
person:
  map: {k1: v1,k2: v2}
```

在YAML配置文件中，配置的属性值为Map集合或对象类型时，缩进式写法的形式按照YAML文件格式编写即可，而行内式写法的属性值要用大括号“{}”包含。

接下来，在Properties配置文件演示案例基础上，通过配置application.yaml配置文件对Person对象进行赋值，具体使用如下

(1) 在项目的resources目录下，新建一个application.yaml配置文件，在该配置文件中编写为Person类设置的配置属性

```
#对实体类对象Person进行属性配置
person:
  id: 1
  name: Lucy
  hobby: [吃饭, 睡觉, 打豆豆]
  family: [father, mother]
  map: {k1: v1, k2: v2}
  pet: {type: dog, name: 旺财}
```

(2) 再次执行测试

```
Person{id=1, name='lucy', hobby=[吃饭, 数据, 打豆豆], family=[father, mother], map={k1=v1, k2=v2}, pet=Pet{type='dog', name='旺财'}}
```

可以看出，测试方法configurationTest()同样运行成功，并正确打印出了Person实体类对象。

需要说明的是，本次使用application.yaml配置文件进行测试时需要提前将application.properties配置文件中编写的配置注释，这是因为application.properties配置文件会覆盖application.yaml配置文件

## 1.6 配置文件属性值的注入

使用Spring Boot全局配置文件设置属性时：

如果配置属性是Spring Boot已有属性，例如服务端口server.port，那么Spring Boot内部会自动扫描并读取这些配置文件中的属性值并覆盖默认属性。

如果配置的属性是用户自定义属性，例如刚刚自定义的Person实体类属性，还必须在程序中注入这些配置属性方可生效。

Spring Boot支持多种注入配置文件属性的方式，下面来介绍如何使用注解@ConfigurationProperties和@Value注入属性

### 1.6.1 使用@ConfigurationProperties注入属性

Spring Boot提供的@ConfigurationProperties注解用来快速、方便地将配置文件中的自定义属性值批量注入到某个Bean对象的多个对应属性中。假设现在有一个配置文件，如果使用@ConfigurationProperties注入配置文件的属性，示例代码如下：

```
@Component
@ConfigurationProperties(prefix = "person")
public class Person {
    private int id;
    // 属性的setxx()方法
    public void setId(int id) {
        this.id = id;
    }
}
```

上述代码使用@Component和@ConfigurationProperties(prefix = "person")将配置文件中的每个属性映射到person类组件中。

### 1.6.2 使用@Value注入属性

@Value注解是Spring框架提供的，用来读取配置文件中的属性值并逐个注入到Bean对象的对应属性中，Spring Boot框架从Spring框架中对@Value注解进行了默认继承，所以在Spring Boot框架中还可以使用该注解读取和注入配置文件属性值。使用@Value注入属性的示例代码如下

```
@Component
public class Person {
    @Value("${person.id}")
    private int id;
}
```

上述代码中，使用@Component和@Value注入Person实体类的id属性。其中，@Value不仅可以配置文件的属性注入Person的id属性，还可以直接给id属性赋值，这点是@ConfigurationProperties不支持的

演示@Value注解读取并注入配置文件属性的使用:

- (1) 在com.lagou.pojo包下新建一个实体类Student，并使用@Value注解注入属性

```
@Component
public class Student {

    @Value("${person.id}")
    private int id;
    @Value("${person.name}")
    private String name; //名称

    //省略toString
}
```

Student类使用@Value注解将配置文件的属性值读取和注入。

从上述示例代码可以看出，使用@Value注解方式需要对每一个属性注入设置，同时又免去了属性的setXX()方法

- (2) 再次打开测试类进行测试

```
@Autowired
private Student student;

@Test
public void studentTest() {
    System.out.println(student);
}
```

打印结果:

```
Student{id=1, name='lucy'}
```

可以看出，测试方法studentTest()运行成功，同时正确打印出了Student实体类对象。需要说明的是，本示例中只是使用@Value注解对实例中Student对象的普通类型属性进行了赋值演示，而@Value注解对于包含Map集合、对象以及YAML文件格式的行内式写法的配置文件的属性注入都不支持，如果赋值会出现错误

## 1.7 自定义配置

spring Boot免除了项目中大部分的手动配置，对于一些特定情况，我们可以通过修改全局配置文件以适应具体生产环境，可以说，几乎所有的配置都可以写在application.properties文件中，Spring Boot会自动加载全局配置文件从而免除我们手动加载的烦恼。但是，如果我们自定义配置文件，Spring Boot是无法识别这些配置文件的，此时就需要我们手动加载。接下来，将针对Spring Boot的自定义配置文件及其加载方式进行讲解

### 1.7.1 使用@PropertySource加载配置文件

对于这种加载自定义配置文件的需求，可以使用@PropertySource注解来实现。  
@PropertySource注解用于指定自定义配置文件的具体位置和名称

当然，如果需要将自定义配置文件中的属性值注入到对应类的属性中，可以使用@ConfigurationProperties或者@Value注解进行属性值注入

演示：

(1) 打开Spring Boot项目的resources目录，在项目的类路径下新建一个test.properties自定义配置文件，在该配置文件中编写需要设置的配置属性

```
#对实体类对象MyProperties进行属性配置
test.id=110
test.name=test
```

(2) 在com.lagou.pojo包下新创建一个配置类MyProperties，提供test.properties自定义配置文件中对应的属性，并根据@PropertySource注解的使用进行相关配置

```
@Component    // 自定义配置类
@PropertySource("classpath:test.properties")    // 指定自定义配置文件位置和名称
@ConfigurationProperties(prefix = "test")    // 指定配置文件注入属性前缀
public class MyProperties {
    private int id;
    private String name;
    // 省略属性getXX()和setXX()方法
    // 省略toString()方法
}
```

主要是一个自定义配置类，通过相关注解引入了自定义的配置文件，并完成了自定义属性值的注入。针对示例中的几个注解，具体说明如下

- @PropertySource("classpath:test.properties")注解指定了自定义配置文件的位置和名称，此示例表示自定义配置文件为classpath类路径下的test.properties文件；
- @ConfigurationProperties(prefix = "test")注解将上述自定义配置文件test.properties中以test开头的属性值注入到该配置类属性中。

(3) 进行测试

```
@Autowired
private MyProperties myProperties;

@Test
public void myPropertiesTest() {
    System.out.println(myProperties);
}
```

打印结果：

MyProperties{id=110, name='test'}

### 1.7.2 使用@Configuration编写自定义配置类

在Spring Boot框架中，推荐使用配置类的方式向容器中添加和配置组件

在Spring Boot框架中，通常使用@Configuration注解定义一个配置类，Spring Boot会自动扫描和识别配置类，从而替换传统Spring框架中的XML配置文件。

当定义一个配置类后，还需要在类中的方法上使用@Bean注解进行组件配置，将方法的返回对象注入到Spring容器中，并且组件名称默认使用的是方法名，当然也可以使用@Bean注解的name或value属性自定义组件的名称

演示：

(1) 在项目下新建一个com.lagou.config包，并在该包下新创建一个类MyService，该类中不需要编写任何代码

```
public class MyService {  
}
```

创建了一个空的MyService类，而该类目前没有添加任何配置和注解，因此还无法正常被Spring Boot扫描和识别

(2) 在项目的com.lagou.config包下，新建一个类MyConfig，并使用@Configuration注解将该类声明一个配置类，内容如下：

```
@Configuration // 定义该类是一个配置类````  
public class MyConfig {  
    @Bean // 将返回值对象作为组件添加到Spring容器中，该组件id默认为方法名  
    public MyService myService(){  
        return new MyService();  
    }  
}
```

MyConfig是@Configuration注解声明的配置类（类似于声明了一个XML配置文件），该配置类会被Spring Boot自动扫描识别；使用@Bean注解的myService()方法，其返回值对象会作为组件添加到了Spring容器中（类似于XML配置文件中的标签配置），并且该组件的id默认是方法名myService

(3) 测试类

```
@Autowired  
private ApplicationContext applicationContext;  
@Test  
public void iocTest() {  
    System.out.println(applicationContext.containsBean("myService"));  
}
```

上述代码中，先通过@Autowired注解引入了Spring容器实例ApplicationContext，然后在测试方法iocTest()中测试查看该容器中是否包括id为myService的组件。



执行测试方法iocTest()，查看控制台输出效果，结果如下：

**true**

从测试结果可以看出，测试方法iocTest()运行成功，显示运行结果为true，表示Spring的IOC容器中也已经包含了id为myService的实例对象组件，说明使用自定义配置类的形式完成了向Spring容器进行组件的添加和配置

## 2. SpringBoot原理深入及源码剖析

传统的Spring框架实现一个Web服务，需要导入各种依赖JAR包，然后编写对应的XML配置文件等，相较而言，Spring Boot显得更加方便、快捷和高效。那么，Spring Boot究竟如何做到这些的呢？

接下来分别针对Spring Boot框架的依赖管理、自动配置通过源码进行深入分析

注意：需要先完成源码环境搭建（springboot源码环境搭建.pdf）

### 2.1 依赖管理

问题：（1）为什么导入dependency时不需要指定版本？

在Spring Boot入门程序中，项目pom.xml文件有两个核心依赖，分别是spring-boot-starter-parent和spring-boot-starter-web，关于这两个依赖的相关介绍具体如下：

#### 1. spring-boot-starter-parent依赖

在chapter01项目中的pom.xml文件中找到spring-boot-starter-parent依赖，示例代码如下：

```
<!-- Spring Boot父项目依赖管理 -->
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent<11./artifactId>
  <version>2.2.2.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

上述代码中，将spring-boot-starter-parent依赖作为Spring Boot项目的统一父项目依赖管理，并将项目版本号统一为2.2.2.RELEASE，该版本号根据实际开发需求是可以修改的

使用“Ctrl+鼠标左键”进入并查看spring-boot-starter-parent底层源文件，发现spring-boot-starter-parent的底层有一个父依赖spring-boot-dependencies，核心代码具体如下

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-dependencies</artifactId>
  <version>2.2.2.RELEASE</version>
  <relativePath>../..spring-boot-dependencies</relativePath>
</parent>
```

继续查看spring-boot-dependencies底层源文件，核心代码具体如下：

```
<properties>
  <activemq.version>5.15.11</activemq.version>
```



```

...
<solr.version>8.2.0</solr.version>
<mysql.version>8.0.18</mysql.version>
<kafka.version>2.3.1</kafka.version>
<spring-amqp.version>2.2.2.RELEASE</spring-amqp.version>
<spring-restdocs.version>2.0.4.RELEASE</spring-restdocs.version>
<spring-retry.version>1.2.4.RELEASE</spring-retry.version>
<spring-security.version>5.2.1.RELEASE</spring-security.version>
<spring-session-bom.version>Corn-RELEASE</spring-session-bom.version>
<spring-ws.version>3.0.8.RELEASE</spring-ws.version>
<sqlite-jdbc.version>3.28.0</sqlite-jdbc.version>
<sun-mail.version>${jakarta-mail.version}</sun-mail.version>
<tomcat.version>9.0.29</tomcat.version>
<thymeleaf.version>3.0.11.RELEASE</thymeleaf.version>
<thymeleaf-extras-data-attribute.version>2.0.1</thymeleaf-extras-data-
attribute.version>
...
</properties>

```

从spring-boot-dependencies底层源文件可以看出，该文件通过标签对一些常用技术框架的依赖文件进行了统一版本号管理，例如activemq、spring、tomcat等，都有与Spring Boot 2.2.2版本相匹配的版本，这也是pom.xml引入依赖文件不需要标注依赖文件版本号的原因。

需要说明的是，如果pom.xml引入的依赖文件不是 spring-boot-starter-parent管理的，那么在pom.xml引入依赖文件时，需要使用标签指定依赖文件的版本号。

(2) 问题2： spring-boot-starter-parent父依赖启动器的主要作用是进行版本统一管理，那么项目运行依赖的JAR包是从何而来的？

## 2. spring-boot-starter-web依赖

查看spring-boot-starter-web依赖文件源码，核心代码如下

```

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>2.2.2.RELEASE</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-json</artifactId>
    <version>2.2.2.RELEASE</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <version>2.2.2.RELEASE</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
    <version>2.2.2.RELEASE</version>

```

```

<scope>compile</scope>
<exclusions>
  <exclusion>
    <artifactId>tomcat-embed-el</artifactId>
    <groupId>org.apache.tomcat.embed</groupId>
  </exclusion>
</exclusions>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
  <version>5.2.2.RELEASE</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.2.2.RELEASE</version>
  <scope>compile</scope>
</dependency>
</dependencies>

```

从上述代码可以发现，spring-boot-starter-web依赖启动器的主要作用是提供Web开发场景所需的底层所有依赖

正是如此，在pom.xml中引入spring-boot-starter-web依赖启动器时，就可以实现Web场景开发，而不需要额外导入Tomcat服务器以及其他Web依赖文件等。当然，这些引入的依赖文件的版本号还是由spring-boot-starter-parent父依赖进行的统一管理。

Spring Boot除了提供有上述介绍的Web依赖启动器外，还提供了其他许多开发场景的相关依赖，我们可以打开Spring Boot官方文档，搜索“Starters”关键字查询场景依赖启动器

Table 13.1. Spring Boot application starters

Name	Description	Pom
spring-boot-starter	Core starter, including auto-configuration support, logging and YAML	<a href="#">Pom</a>
spring-boot-starter-activemq	Starter for JMS messaging using Apache ActiveMQ	<a href="#">Pom</a>
spring-boot-starter-amqp	Starter for using Spring AMQP and Rabbit MQ	<a href="#">Pom</a>
spring-boot-starter-aop	Starter for aspect-oriented programming with Spring AOP and AspectJ	<a href="#">Pom</a>
spring-boot-starter-artemis	Starter for JMS messaging using Apache Artemis	<a href="#">Pom</a>
spring-boot-starter-batch	Starter for using Spring Batch	<a href="#">Pom</a>
spring-boot-starter-cache	Starter for using Spring Framework's caching support	<a href="#">Pom</a>
spring-boot-starter-cloud-connectors	Starter for using Spring Cloud Connectors which simplifies connecting to services in cloud platforms like Cloud Foundry and Heroku	<a href="#">Pom</a>
spring-boot-starter-data-cassandra	Starter for using Cassandra distributed database and Spring Data Cassandra	<a href="#">Pom</a>
spring-boot-starter-data-cassandra-reactive	Starter for using Cassandra distributed database and Spring Data Cassandra Reactive	<a href="#">Pom</a>
spring-boot-starter-data-couchbase	Starter for using Couchbase document-oriented database and Spring Data Couchbase	<a href="#">Pom</a>

列出了Spring Boot官方提供的部分场景依赖启动器，这些依赖启动器适用于不同的场景开发，使用时只需要在pom.xml文件中导入对应的依赖启动器即可。

需要说明的是，Spring Boot官方并不是针对所有场景开发的技术框架都提供了场景启动器，例如数据库操作框架MyBatis、阿里巴巴的Druid数据源等，Spring Boot官方就没有提供对应的依赖启动器。为了充分利用Spring Boot框架的优势，在Spring Boot官方没有整合这些技术框架的情况下，MyBatis、Druid等技术框架所在的开发团队主动与Spring Boot框架进行了整合，实现了各自的依赖启动器，例如

mybatis-spring-boot-starter、druid-spring-boot-starter等。我们在pom.xml文件中引入这些第三方的依赖启动器时，切记要配置对应的版本号

## 2.2 自动配置

概念：能够在我们添加jar包依赖的时候，自动为我们配置一些组件的相关配置，我们无需配置或者只需要少量配置就能运行编写的项目

问题：Spring Boot到底是如何进行自动配置的，都把哪些组件进行了自动配置？

Spring Boot应用的启动入口是@SpringBootApplication注解标注类中的main()方法，@SpringBootApplication能够扫描Spring组件并自动配置Spring Boot

下面，查看@SpringBootApplication内部源码进行分析，核心代码如下

```
@SpringBootApplication
public class SpringbootDemoApplication {

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

```
@Target({ElementType.TYPE}) //注解的适用范围,Type表示注解可以描述在类、接口、注解或枚举中
@Retention(RetentionPolicy.RUNTIME) //表示注解的生命周期,Runtime运行时
@Documented //表示注解可以记录在javadoc中
@Inherited //表示可以被子类继承该注解
@SpringBootConfiguration // 标明该类为配置类
@EnableAutoConfiguration // 启动自动配置功能
@ComponentScan( // 包扫描器
    excludeFilters = {@Filter(
        type = FilterType.CUSTOM,
        classes = {TypeExcludeFilter.class}
    )}, @Filter(
        type = FilterType.CUSTOM,
        classes = {AutoConfigurationExcludeFilter.class}
    )}
)
public @interface SpringBootApplication {
    ...
}
```

从上述源码可以看出，@SpringBootApplication注解是一个组合注解，前面4个是注解的元数据信息，我们主要看后面3个注解：@SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan三个核心注解，关于这三个核心注解的相关说明具体如下：

### 1. @SpringBootConfiguration注解

@SpringBootConfiguration注解表示Spring Boot配置类。查看@SpringBootConfiguration注解源码，核心代码如下。

```

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration //配置IOC容器
public @interface SpringBootConfiguration {
}

```

从上述源码可以看出，@SpringBootConfiguration注解内部有一个核心注解@Configuration，该注解是Spring框架提供的，表示当前类为一个配置类（XML配置文件的注解表现形式），并可以被组件扫描器扫描。由此可见，@SpringBootConfiguration注解的作用与@Configuration注解相同，都是标识一个可以被组件扫描器扫描的配置类，只不过@SpringBootConfiguration是被Spring Boot进行了重新封装命名而已

## 2. @EnableAutoConfiguration注解

@EnableAutoConfiguration注解表示开启自动配置功能，该注解是Spring Boot框架最重要的注解，也是实现自动化配置的注解。同样，查看该注解内部查看源码信息，核心代码如下

```

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage // 自动配置包
@Import({AutoConfigurationImportSelector.class}) // 自动配置类扫描导入
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";
    Class<?>[] exclude() default {};
    String[] excludeName() default {};
}

```

可以发现它是一个组合注解，Spring 中有很多以Enable开头的注解，其作用就是借助@Import来收集并注册特定场景相关的bean，并加载到IoC容器。@EnableAutoConfiguration就是借助@Import来收集所有符合自动配置条件的bean定义，并加载到IoC容器。

下面，对这两个核心注解分别讲解：

### (1) @AutoConfigurationPackage注解

查看@AutoConfigurationPackage注解内部源码信息，核心代码如下：

```

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import({Registrar.class}) // 导入Registrar中注册的组件
public @interface AutoConfigurationPackage {
}

```

从上述源码可以看出，@AutoConfigurationPackage注解的功能是由@Import注解实现的，它是spring框架的底层注解，它的作用就是给容器中导入某个组件类，例如@Import(AutoConfigurationPackage.Registrar.class)，它就是将Registrar这个组件类导入到容器中，可查看Registrar类中registerBeanDefinitions方法，这个方法就是导入组件类的具体实现：

```

static class Registrar implements ImportBeanDefinitionRegistrar, DeterminableImports {
    Registrar() {
    }

    public void registerBeanDefinitions(AnnotationMetadata metadata, BeanDefinitionRegistry registry) {
        AutoConfigurationPackages.register(registry, (
            new AutoConfigurationPackages.PackageImport(metadata)).getPackageName());
    }
}

```

从上述源码可以看出，在Registrar类中有一个registerBeanDefinitions()方法，使用Debug模式启动项目，可以看到选中的部分就是com.lagou。也就是说，@AutoConfigurationPackage注解的主要作用就是将主程序类所在包及所有子包下的组件到扫描到spring容器中。

因此 在定义项目包结构时，要求定义的包结构非常规范，项目主程序启动类要定义在最外层的根目录位置，然后在根目录位置内部建立子包和类进行业务开发，这样才能够保证定义的类能够被组件扫描器扫描

(2) @Import({AutoConfigurationImportSelector.class}): 将AutoConfigurationImportSelector这个类导入到spring容器中，AutoConfigurationImportSelector可以帮助springboot应用将所有符合条件的@Configuration配置都加载到当前SpringBoot创建并使用的IoC容器(Application Context)中

继续研究AutoConfigurationImportSelector这个类，通过源码分析这个类中是通过selectImports这个方法告诉springboot都需要导入那些组件：

```

@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    //获得自动配置元信息，需要传入beanClassLoader这个类加载器
    AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader
        .loadMetadata(this.beanClassLoader);

    AutoConfigurationEntry autoConfigurationEntry = getAutoConfigurationEntry(
        autoConfigurationMetadata, annotationMetadata);
    return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
}

```

深入研究loadMetadata方法

```

protected static final String PATH = "META-INF/"
    + "spring-autoconfigure-metadata.properties"; //文件中为需要加载的配置类的类路径

public static AutoConfigurationMetadata loadMetadata(ClassLoader classLoader) {
    return loadMetadata(classLoader, PATH);
}

static AutoConfigurationMetadata loadMetadata(ClassLoader classLoader, String path) {
    try {

        //读取spring-boot-autoconfigure-2.1.5.RELEASE.jar包中spring-autoconfigure-metadata.properties的信息生成url
        Enumeration<URL> urls = (classLoader != null) ? classLoader.getResources(path)
            : ClassLoader.getSystemResources(path);
        Properties properties = new Properties();

        //解析urls枚举对象中的信息封装成properties对象并加载
        while (urls.hasMoreElements()) {
            properties.putAll(PropertiesLoaderUtils
                .loadProperties(new UrlResource(urls.nextElement())));
        }

        //根据封装好的properties对象生成AutoConfigurationMetadata对象返回
        return loadMetadata(properties);
    }
    catch (IOException ex) {
        throw new IllegalArgumentException(
            "Unable to load @ConditionalOnClass location [" + path + "]", ex);
    }
}

```

## 深入getCandidateConfigurations方法

这个方法中有一个重要方法loadFactoryNames，这个方法是让SpringFactoryLoader去加载一些组件的名字。

```

protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
    AnnotationAttributes attributes) {

    /**
     * 这个方法需要传入两个参数getSpringFactoriesLoaderFactoryClass()和getBeanClassLoader()
     * getSpringFactoriesLoaderFactoryClass()这个方法返回的是EnableAutoConfiguration.class
     * getBeanClassLoader()这个方法返回的是beanClassLoader（类加载器）
     */
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
        getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
    Assert.notEmpty(configurations,
        "No auto configuration classes found in META-INF/spring.factories. If you "
            + "are using a custom packaging, make sure that file is correct.");

    return configurations;
}

/**
 * Return the class used by {@link SpringFactoriesLoader} to load configuration
 * candidates.
 * @return the factory class
 */
protected Class<?> getSpringFactoriesLoaderFactoryClass() {
    return EnableAutoConfiguration.class;
}

protected ClassLoader getBeanClassLoader() {
    return this.beanClassLoader;
}

```

继续点开loadFactory方法

```

public static List<String> loadFactoryNames(Class<?> factoryClass, @Nullable
ClassLoader classLoader) {

    //获取出入的键
    String factoryClassName = factoryClass.getName();
    return
        (List)loadSpringFactories(classLoader).getOrDefault(factoryClassName,
            Collections.emptyList());
}

```



```

    }

    private static Map<String, List<String>> loadSpringFactories(@Nullable
ClassLoader classLoader) {
        Multimap<String, String> result =
(Multimap)cache.get(classLoader);
        if (result != null) {
            return result;
        } else {
            try {

                //如果类加载器不为null, 则加载类路径下spring.factories文件, 将其中设置的
                配置类的全路径信息封装 为Enumeration类对象
                Enumeration<URL> urls = classLoader != null ?
classLoader.getResources("META-INF/spring.factories") :
classLoader.getSystemResources("META-INF/spring.factories");
                LinkedMultimap result = new LinkedMultimap();

                //循环Enumeration类对象, 根据相应的节点信息生成Properties对象, 通过传入
                的键获取值, 在将值切割为一个小小的字符串转化为Array, 方法result集合中
                while(urls.hasMoreElements()) {
                    URL url = (URL)urls.nextElement();
                    UrlResource resource = new UrlResource(url);
                    Properties properties =
PropertiesLoaderUtils.loadProperties(resource);
                    Iterator var6 = properties.entrySet().iterator();

                    while(var6.hasNext()) {
                        Entry<?, ?> entry = (Entry)var6.next();
                        String factoryClassName =
((String)entry.getKey()).trim();
                        String[] var9 =
StringUtils.commaDelimitedListToStringArray((String)entry.getValue());
                        int var10 = var9.length;

                        for(int var11 = 0; var11 < var10; ++var11) {
                            String factoryName = var9[var11];
                            result.add(factoryClassName, factoryName.trim());
                        }
                    }
                }

                cache.put(classLoader, result);
                return result;
            }
        }
    }
}

```

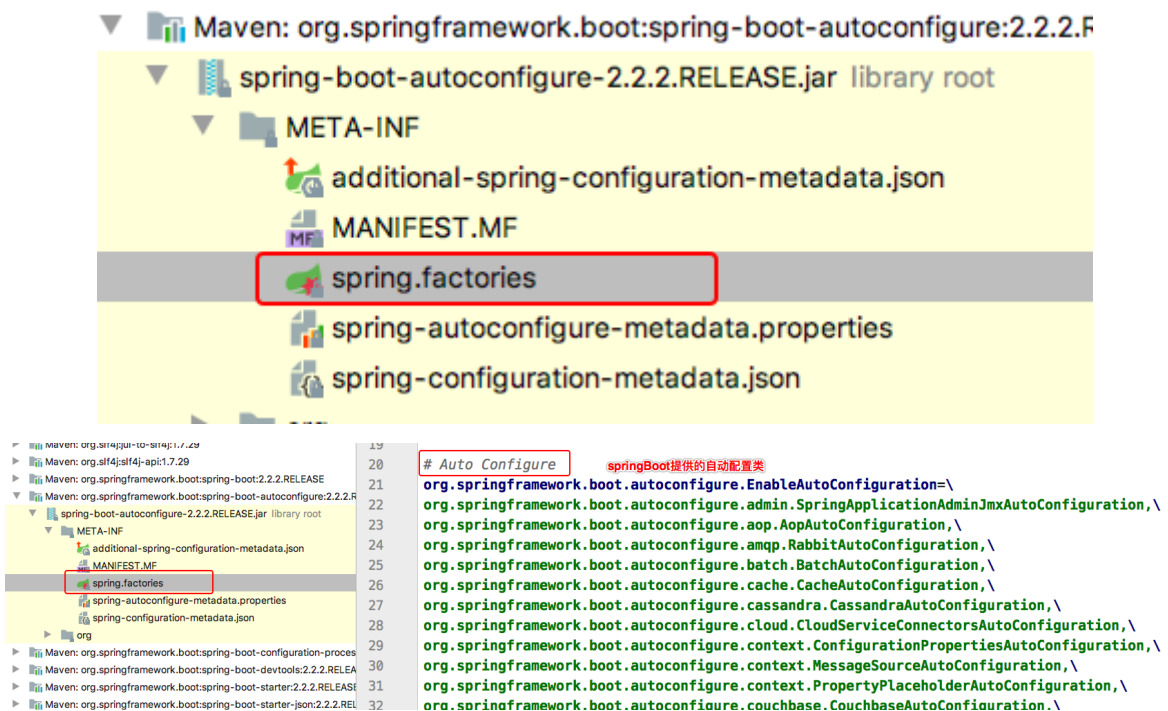
会去读取一个 spring.factories 的文件, 读取不到会报这个错误, 我们继续根据会看到, 最终路径的长这样, 而这个是spring提供的一个工具类

```

public final class SpringFactoriesLoader {
    public static final String FACTORIES_RESOURCE_LOCATION = "META-
INF/spring.factories";
}

```

它其实是去加载一个外部的文件, 而这文件是在



@EnableAutoConfiguration就是从classpath中搜寻META-INF/spring.factories配置文件，并将其中org.springframework.boot.autoconfigure.EnableAutoConfiguration对应的配置项通过反射（Java Reflection）实例化为对应的标注了@Configuration的JavaConfig形式的配置类，并加载到IOC容器中

以刚刚的项目为例，在项目中加入了Web环境依赖启动器，对应的WebMvcAutoConfiguration自动配置类就会生效，打开该自动配置类会发现，在该配置类中通过全注解配置类的方式对Spring MVC运行所需环境进行了默认配置，包括默认前缀、默认后缀、视图解析器、MVC校验器等。而这些自动配置类的本质是传统Spring MVC框架中对应的XML配置文件，只不过在Spring Boot中以自动配置类的形式进行了预先配置。因此，在Spring Boot项目中加入相关依赖启动器后，基本上不需要任何配置就可以运行程序，当然，我们也可以对这些自动配置类中默认的配置进行更改

## 总结

因此springboot底层实现自动配置的步骤是：

1. springboot应用启动；
2. @SpringBootApplication起作用；
3. @EnableAutoConfiguration；
4. @AutoConfigurationPackage：这个组合注解主要是@Import(AutoConfigurationPackages.Registrar.class)，它通过将Registrar类导入到容器中，而Registrar类作用是扫描主配置类同级目录以及子包，并将相应的组件导入到springboot创建管理的容器中；
5. @Import(AutoConfigurationImportSelector.class)：它通过将AutoConfigurationImportSelector类导入到容器中，AutoConfigurationImportSelector类作用是通过selectImports方法执行的过程中，会使用内部工具类SpringFactoriesLoader，查找classpath上所有jar包中的META-INF/spring.factories进行加载，实现将配置类信息交给SpringFactory加载器进行一系列的容器创建过程

## 3. @ComponentScan注解

@ComponentScan注解具体扫描的包的根路径由Spring Boot项目主程序启动类所在包位置决定，在扫描过程中由前面介绍的@AutoConfigurationPackage注解进行解析，从而得到Spring Boot项目主程序启动类所在包的具体位置



总结：

@SpringBootApplication 的注解的功能就分析差不多了，简单来说就是 3 个注解的组合注解：

```
| - @SpringBootConfiguration
|   | - @Configuration //通过javaConfig的方式来添加组件到IOC容器中
| - @EnableAutoConfiguration
|   | - @AutoConfigurationPackage //自动配置包，与@ComponentScan扫描到的添加到IOC
|   | - @Import(AutoConfigurationImportSelector.class) //到META-INF/spring.factories中定义的bean添加到IOC容器中
| - @ComponentScan //包扫描
```

## 3. SpringBoot数据访问

### 3.1 Spring Boot整合MyBatis

MyBatis 是一款优秀的持久层框架，Spring Boot官方虽然没有对MyBatis进行整合，但是MyBatis团队自行适配了对应的启动器，进一步简化了使用MyBatis进行数据的操作

因为Spring Boot框架开发的便利性，所以实现Spring Boot与数据访问层框架（例如MyBatis）的整合非常简单，主要是引入对应的依赖启动器，并进行数据库相关参数设置即可

**基础环境搭建：**

#### 1.数据准备

在MySQL中，先创建了一个数据库springbootdata，然后创建了两个表t\_article和t\_comment并向表中插入数据。其中评论表t\_comment的a\_id与文章表t\_article的主键id相关联

```
# 创建数据库
CREATE DATABASE springbootdata;
# 选择使用数据库
USE springbootdata;
# 创建表t_article并插入相关数据
DROP TABLE IF EXISTS t_article;
CREATE TABLE t_article (
  id int(20) NOT NULL AUTO_INCREMENT COMMENT '文章id',
  title varchar(200) DEFAULT NULL COMMENT '文章标题',
  content longtext COMMENT '文章内容',
  PRIMARY KEY (id)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
INSERT INTO t_article VALUES ('1', 'Spring Boot基础入门', '从入门到精通讲解...');
INSERT INTO t_article VALUES ('2', 'Spring Cloud基础入门', '从入门到精通讲解...');

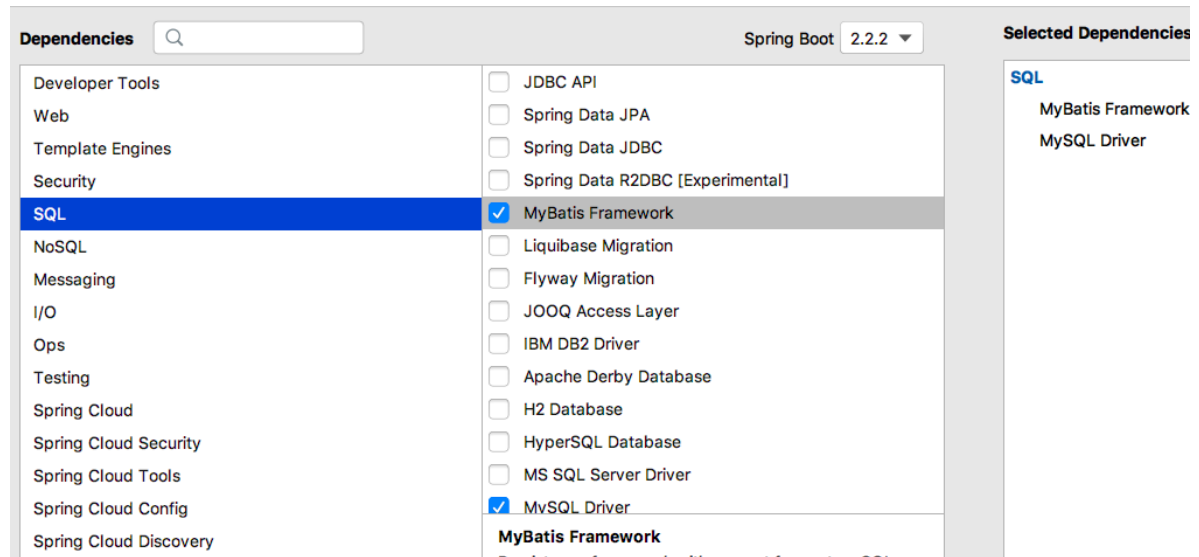
# 创建表t_comment并插入相关数据
DROP TABLE IF EXISTS t_comment;
CREATE TABLE t_comment (
  id int(20) NOT NULL AUTO_INCREMENT COMMENT '评论id',
  content longtext COMMENT '评论内容',
  author varchar(200) DEFAULT NULL COMMENT '评论作者',
  a_id int(20) DEFAULT NULL COMMENT '关联的文章id',
```

```

PRIMARY KEY (id)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;
INSERT INTO t_comment VALUES ('1', '很全、很详细', 'lucy', '1');
INSERT INTO t_comment VALUES ('2', '赞一个', 'tom', '1');
INSERT INTO t_comment VALUES ('3', '很详细', 'eric', '1');
INSERT INTO t_comment VALUES ('4', '很好, 非常详细', '张三', '1');
INSERT INTO t_comment VALUES ('5', '很不错', '李四', '2');

```

## 2. 创建项目，引入相应的启动器



## 3. 编写与数据库表t\_comment和t\_article对应的实体类Comment和Article

```

public class Comment {
    private Integer id;
    private String content;
    private String author;
    private Integer aId;
    // 省略属性getXX()和setXX()方法
    // 省略toString()方法
}

```

```

public class Article {

    private Integer id;
    private String title;
    private String content;
    // 省略属性getXX()和setXX()方法
    // 省略toString()方法
}

```

## 4. 编写配置文件

- (1) 在application.properties配置文件中进行数据库连接配置

```
# MySQL数据库连接配置
spring.datasource.url=jdbc:mysql://localhost:3306/springbootdata?
serverTimezone=UTC&characterEncoding=UTF-8
spring.datasource.username=root
spring.datasource.password=root
```

## 注解方式整合Mybatis

(1) 创建一个对t\_comment表数据操作的接口CommentMapper

```
public interface CommentMapper {
    @Select("SELECT * FROM t_comment WHERE id =#{id}")
    public Comment findById(Integer id);
}
```

(2) 在Spring Boot项目启动类上添加@MapperScan("xxx")注解

```
@SpringBootApplication
@MapperScan("com.lagou.dao")
public class Springboot02MybatisApplication {

    public static void main(String[] args) {
        SpringApplication.run(Springboot02MybatisApplication.class, args);
    }

}
```

(3) 编写测试方法

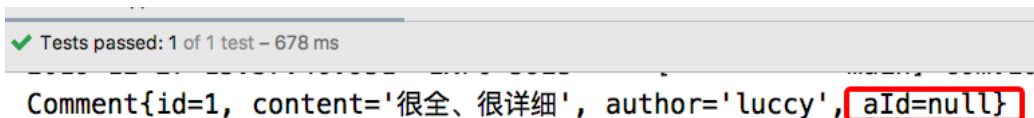
```
@RunWith(SpringRunner.class)
@SpringBootTest
class SpringbootPersistenceApplicationTests {

    @Autowired
    private CommentMapper commentMapper;

    @Test
    void contextLoads() {
        Comment comment = commentMapper.findById(1);
        System.out.println(comment);
    }

}
```

打印结果:



```
✓ Tests passed: 1 of 1 test - 678 ms
Comment{id=1, content='很全、很详细', author='lucy', aId=null}
```

控制台中查询的Comment的aId属性值为null，没有映射成功。这是因为编写的实体类Comment中使用了驼峰命名方式将t\_comment表中的a\_id字段设计成了aId属性，所以无法正确映射查询结果。

为了解决上述由于驼峰命名方式造成的表字段值无法正确映射到类属性的情况，可以在Spring Boot全局配置文件application.properties中添加开启驼峰命名匹配映射配置，示例代码如下

```
#开启驼峰命名匹配映射
mybatis.configuration.map-underscore-to-camel-case=true
```

打印结果：

✓ Tests passed: 1 of 1 test – 738 ms

```
Comment{id=1, content='很全、很详细', author='lucy', aId=1}
```

## 使用配置文件的方式整合MyBatis

(1) 创建一个用于对数据库表t\_article数据操作的接口ArticleMapper

```
@Mapper
public interface ArticleMapper {
    public Article selectArticle(Integer id);
}
```

(2) 创建XML映射文件

resources目录下创建一个统一管理映射文件的包mapper，并在该包下编写与ArticleMapper接口方应的映射文件ArticleMapper.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.lagou.mapper.ArticleMapper">
    <select id="selectArticle" resultType="Article">
        select * from Article
    </select>
</mapper>
```

(3) 配置XML映射文件路径。在项目中编写的XML映射文件，Spring Boot并无从知晓，所以无法扫描到该自定义编写的XML配置文件，还必须在全局配置文件application.properties中添加MyBatis映射文件路径的配置，同时需要添加实体类别名映射路径，示例代码如下

```
#配置MyBatis的xml配置文件路径
mybatis.mapper-locations=classpath:mapper/*.xml
#配置XML映射文件中指定的实体类别名路径
mybatis.type-aliases-package=com.lagou.pojo
```

(4) 编写单元测试进行接口方法测试

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.1.3</version>
  </dependency>
</dependencies>
```

```

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

<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.3</version>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
  <exclusions>
    <exclusion>
      <groupId>org.junit.vintage</groupId>
      <artifactId>junit-vintage-engine</artifactId>
    </exclusion>
  </exclusions>
</dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>

```

### 3. User实体类编写

```

public class User {

  private Integer id;
  private String username;
  private String password;
  private String birthday;

  public Integer getId() {
    return id;
  }

```

```

    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getBirthday() {
        return birthday;
    }

    public void setBirthday(String birthday) {
        this.birthday = birthday;
    }

    @Override
    public String toString() {
        return "User{" +
            "id=" + id +
            ", username='" + username + '\'' +
            ", password='" + password + '\'' +
            ", birthday='" + birthday + '\'' +
            '}';
    }
}

```

#### 4. UserDao编写

```

public interface UserDao {

    /**
     * 查询所有用户
     */
    public List<User> findAllUser();
}

```

#### 5.UserService接口及实现类编写

```
public interface UserService {

    /**
     * 查询所有用户
     */
    public List<User> findAllUser();

}
```

## 6.UserController编写

```
@Controller
public class UserController {

    @Autowired
    private UserService userService;

    @RequestMapping("/findAllUser")
    public String findAllUser(){
        List<User> allUser = userService.findAllUser();
        System.out.println(allUser);
        return "success";
    }

}
```

## 7.success.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    success...
</body>
</html>
```

## 8. application.yml

```
##服务器配置
server:
  port: 8090
  servlet:
    context-path: /

##
spring:
```



```

datasource:
  name: druid
  type: com.alibaba.druid.pool.DruidDataSource
  url: jdbc:mysql://localhost:3306/spring_db?characterEncoding=utf-
8&serverTimezone=UTC
  username: root
  password: root
  driver-class-name: com.mysql.jdbc.Driver
mvc:
  view:
    prefix: /
    suffix: .html

#整合mybatis
mybatis:
  mapper-locations: classpath:mapper/*Mapper.xml #声明Mybatis映射文件所在的位置
  #config-location: classpath:mybatis.xml #声明Mybatis配置文件所在位置

```

## 9.启动项目，进行访问测试

```

SsmSpringbootApplication
Console Endpoints
2020-09-01 14:02:24.230 INFO 46572 --- [main] org.springframework.boot.web.embedded.tomcat.TomcatWebServer : Starting Servlet engine: [Apache Tomcat/9.0.37]
2020-09-01 14:02:24.230 INFO 46572 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2020-09-01 14:02:24.230 INFO 46572 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1
2020-09-01 14:02:24.591 INFO 46572 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-09-01 14:02:24.767 INFO 46572 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8090 (http) with context path
2020-09-01 14:02:24.778 INFO 46572 --- [main] com.lagou.SsmSpringbootApplication : Started SsmSpringbootApplication in 2.26 seconds (JVM run
2020-09-01 14:02:57.668 INFO 46572 --- [nio-8090-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2020-09-01 14:02:57.669 INFO 46572 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2020-09-01 14:02:57.676 INFO 46572 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 7 ms
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
2020-09-01 14:02:57.735 INFO 46572 --- [nio-8090-exec-1] com.alibaba.druid.pool.DruidDataSource : {dataSource-1} inited
[User{id=1, username='子慕', password='123', birthday='2020-12-12'}, User{id=2, username='应顺', password='123', birthday='2020-12-12'}]

```

← → ↻ ① localhost:8090/findAllUser

success...