

Spring Boot



课程目标

- 了解 Java 配置的优势。
- 掌握 Java 配置的使用。
- 理解 Spring Boot 的作用。
- 掌握 Spring Boot 的使用。
- 掌握 Spring Boot 属性绑定。
- 了解 Spring Boot 自动配置原理。
- 掌握 Spring Boot 搭建 SSM 开发环境。
- 了解日志的作用。

一、JavaConfig（掌握）

我们通常使用 Spring 都会使用 XML 配置，随着功能以及业务逻辑的日益复杂，应用伴随着大量的 XML 配置文件以及复杂的 bean 依赖关系，使用起来很不方便。

在 Spring 3.0 开始，Spring 官方就已经开始推荐使用 Java 配置来代替传统的 XML 配置了，它允许开发者将 bean 的定义和 Spring 的配置编写到 Java 类中，不过似乎在国内并未推广盛行。当 Spring Boot 来临，人们才慢慢认识到 Java 配置的优雅，但是，也仍然允许使用经典的 XML 方式来定义 bean 和 配置 Spring。其有以下优势：

- 面向对象的配置。由于配置被定义为 JavaConfig 中的类，因此用户可以充分使用 Java 中的面向对象功能。一个配置类可以继承另一个，重写它的 @Bean 方法等。
- 减少或者消除 XML 配置。提供了一种纯 Java 的方式来配置与 XML 配置概念相似的 Spring 容器。
- 类型安全和重构友好。提供了一种类型安全的方法来配置 Spring 容器，由于 Java 5 对泛型的支持，现在可以按类型而不是名称检索 bean，不需要任何的强制转换或者基于字符串的查找。

进行下面学习先搭建一个基于 Maven 构建的项目 java-config-demo，添加如下依赖：

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>5.0.8.RELEASE</version>
</dependency>
```

1、XML 方式配置 IoC

1.1、定义两个 Bean

```
public class SomeBean {
    private OtherBean otherBean;

    public void setOtherBean(OtherBean otherBean) {
        this.otherBean = otherBean;
    }

    public SomeBean() {
        System.out.println("SomeBean 被创建");
    }

    public void init() {
        System.out.println("SomeBean 被初始化");
    }

    public void destroy() {
        System.out.println("SomeBean 被销毁");
    }
    // 省略 toString 方法
}

public class OtherBean {
    public OtherBean() {
        System.out.println("OtherBean 被创建");
    }
}
```

1.2、编写配置

在 XML 配置文件中配置这些 Bean 交给 Spring 管理。

```
<!-- applicationContext.xml -->
<bean id="someBean" class="cn.wolfcode.bean.SomeBean"/>
```

1.3、启动 Spring

启动 Spring 读取该 XML 文件创建容器对象，从容器中获取 SomeBean 对象。

```

public class IoCTest {
    @Test
    public void testXmlConfig() {
        ApplicationContext ctx = new
ClassPathXmlApplicationContext("classpath:applicationContext.xml");
        SomeBean someBean = ctx.getBean(SomeBean.class);
        System.out.println(someBean);
    }
}

```

2、JavaConfig 方式配置 IoC

JavaConfig 方式中使用注解彻底的替代 XML 文件，那么到底要怎么告诉 Spring 容器，bean 没有定义在 XML 文件中，而是定义在一个 Java 配置类中。

- `@Configuration`：在类上贴该注解表示该类是 Spring 的配置类，具有 applicationContext.xml 文件的作用。
- `@Bean`：在 Spring 的配置类的方法上贴该注解后，该方法返回的对象会交给 Spring 容器管理，替代 applicationContext.xml 中的 bean 标签。
- `@ComponentScan`：在 Spring 配置类上贴该注解表示开启组件扫描器，默认扫描当前配置类所在的包，也可以自己指定，替代 XML 配置中的 `<context:component-scan />` 标签。
- `AnnotationConfigApplicationContext`：该类是 ApplicationContext 接口的实现类，该对象是基于 JavaConfig 的方式来运作的 Spring 容器。

2.1、定义一个配置类

替代之前的 XML 文件，类中定义方法，返回 bean 对象交给 Spring 管理。

```

/**
 * @Configuration
 * 贴有该注解的类表示 Spring 的配置类
 * 用于替代传统的 applicationContext.xml
 */
@Configuration
public class JavaConfig {

    /**
     * @Bean
     * 该注解贴在配置类的方法上，该方法会被 Spring 容器自动调用
     * 并且返回的对象交给 Spring 管理
     * 相当于 <bean id="someBean" class="cn.wolfcode.bean.SomeBean"/>
     */
    @Bean
    public SomeBean someBean() {
        return new SomeBean();
    }
}

```

2.2、启动 Spring

加载配置类，启动 AnnotationConfigApplicationContext 容器对象，测试效果。

```

public class IoCTest {
    @Test
    public void testJavaConfig() {
        // 加载配置类, 创建 Spring 容器
        ApplicationContext ctx = new
        AnnotationConfigApplicationContext(JavaConfig.class);
        // 从容器中取出 SomeBean 对象
        SomeBean someBean = ctx.getBean(SomeBean.class);
        System.out.println(someBean);
    }
}

```

2.3、@Bean 注解中的属性

在 XML 配置 bean 的方式中, 我们可以在 bean 标签中的 id, name, init-method, destroy-method, scope 等属性来完成对应的配置, 在使用 JavaConfig 方式中我们也一样能通过相应的配置来完成同样的效果, 这些效果大多封装到 @Bean 注解的属性中。@Bean 注解中的属性有以下:

- name: 对应 bean 标签中的 name 属性, 用于给 bean 取别名;
- initMethod: 对应 bean 标签中的 init-method 属性, 配置 bean 的初始化方法;
- destroyMethod: 对应 bean 标签中的 destroy-method 属性, 配置 bean 的销毁方法。

注意: 在配置类的方式中有许多的默认规定, 比如:

- bean 的 id 就是当前方法名;
- 配置多例则是在方法上添加 @Scope("prototype") 注解来实现, 一般不用配, 默认单例即可。

3、XML 方式配置 DI

```

<bean id="someBean" class="cn.wolfcode.bean.someBean">
    <property name="otherBean" ref="otherBean"/>
</bean>

<bean id="otherBean" class="cn.wolfcode.bean.OtherBean"/>

```

4、JavaConfig 方式配置 DI

在配置类方式中我们有两种方式可以完成依赖注入, 无论是哪种方式, 前提都是要先把 bean 交给 Spring 管理, 然后在把 bean 注入过去后再使用 setter 方法设置关系。通用步骤: 先把两个 bean 交给 Spring 管理。

```

@Bean
public SomeBean someBean() {
    SomeBean someBean = new SomeBean();
    return someBean;
}

```

```

@Bean
public OtherBean otherBean() {
    return new OtherBean();
}

```

4.1、通过方法形参注入

把需要注入的 bean 对象作为参数传入到另一个 bean 的方法声明中，形参名称最好跟 bean 的 id 一致。在容器里面有的 bean，都可以用这种方式注入。

```
// 在声明 SomeBean 的方法形参中直接注入 OtherBean 对象
@Bean
public SomeBean someBean(OtherBean otherBean) {
    SomeBean someBean = new SomeBean();
    someBean.setOtherBean(otherBean);
    return someBean;
}
```

4.2、调用方法注入

```
// 调用上面已经声明的 otherBean 方法
@Bean
public SomeBean someBean() {
    SomeBean someBean = new SomeBean();
    someBean.setOtherBean(otherBean());
    return someBean;
}
```

原理：Spring 容器在调用实例方法时，根据方法返回对象类型，判断容器中是否已经存在该类型的实例对象，如果不存在则执行实例方法，将返回对象实例交给容器管理，如果该实例已经存在了，直接从容器中拿已经存在实例对象方法，不执行实例方法。

5、使用 IoC DI 注解简化配置

以上案例中，在配置类内部去定义方法返回 bean 对象交给 Spring 管理的方式存在一个问题，就是如果需要创建的 bean 很多的话，那么就需要定义很多的方法，会导致配置类比较累赘，使用起来不方便。以前可以通过注解简化 XML 配置，现在同样也可以通过注解简化 JavaConfig，这里需要使用到 @ComponentScan 注解，等价于之前 XML 配置的 `<context:component-scan base-package="贴了 IoC DI 注解的类所在的包"/>`。

```
@ToString
@Component
public class SomeBean {
    private OtherBean otherBean;

    @Autowired
    public void setOtherBean(OtherBean otherBean) {
        this.otherBean = otherBean;
    }

    public SomeBean() {
        System.out.println("SomeBean 被创建");
    }

    public void init() {
        System.out.println("SomeBean 被初始化");
    }

    public void destroy() {
        System.out.println("SomeBean 被销毁");
    }
}
```

```

}

@Component
public class OtherBean {
    public OtherBean() {
        System.out.println("OtherBean 被创建");
    }
}

```

```

<!-- applicationContext.xml -->
<context:component-scan base-package="cn.wolfcode.bean"/>

```

```

@Configuration // 表示该类是 Spring 的配置类
@ComponentScan // 开启组件扫描器，默认扫描当前类所在的包，及其子包
public class JavaConfig { }

```

若需要扫描的包不是配置类所在的包时，我们可以通过注解中的 value 属性来修改扫描的包。

注意：组件扫描的方式只能扫描我们自己写的组件，若某个 bean 不是我们写的，则还是要通过在配置类中定义方法来处理，两者是可以同时存在的。

6、Spring Test 方式加载配置类

首先在 pom.xml 添加如下依赖：

```

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.0.8.RELEASE</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.6.3</version>
    <scope>test</scope>
</dependency>

```

6.1、JUnit4 的方式

6.1.1、基于 XML

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:XML文件路径")
public class IoCTest {
    @Autowired
    private SomeBean someBean;

    @Test
    public void test() {
        System.out.println(someBean);
    }
}

```

6.1.2、基于配置类

`@ContextConfiguration` 注解不仅支持 XML 方式启动 Spring 测试，也支持配置类的方式，配置 `classes` 属性来指定哪些类是配置类即可。

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes={配置类1.class, 配置类2.class, ...})
public class IoCTest {
    @Autowired
    private SomeBean someBean;

    @Test
    public void test() {
        System.out.println(someBean);
    }
}

```

6.2、JUnit5 的方式

注意测试类和测试方法都不用 `public` 修饰，测试类只需要贴 `@SpringJUnitConfig` 指定加载的配置即可。

```

@SpringJUnitConfig(配置类.class)
class IoCTest {
    @Autowired
    private SomeBean someBean;

    @Test
    void test() {
        System.out.println(someBean);
    }
}

```

7、配置类的导入

在 Spring 项目中一般都会有多个 Spring 的配置文件，分别配置不同的组件，最后关联到主配置文件中，该功能也是同样可以在配置类的方式中使用的。

7.1、XML 方式

```
<!-- 例如 mvc.xml 中导入 applicationContext.xml -->
<import resource="classpath: applicationContext.xml"/>
```

7.2、配置类方式

需要使用 `@Import` 来完成，指定导入的配置类。

```
// 主配置类
@Configuration
@Import(OtherJavaConfig.class) // 在主配置类中关联次配置类
public class JavaConfig { ... }

// 次配置类
@Configuration
public class OtherJavaConfig { ... }

// 测试
@SpringJUnitConfig(classes = JavaConfig.class) // 加载主配置类
public class IoCTest { ... }
```

7.3、配置类导入 XML 配置

需要使用 `@ImportResource` 来完成，指定导入 XML 配置文件的路径。

```
// 主配置类
@Configuration
@ImportResource("classpath:XML文件路径") // 在主配置类中关联 XML 配置
public class JavaConfig { ... }

// 测试
@SpringJUnitConfig(classes = JavaConfig.class) // 加载主配置类
public class IoCTest { ... }
```

二、Spring Boot 介绍（理解）

[Spring Boot](#) 是由 Pivotal 团队提供的全新框架，其设计目的是用来简化新 Spring 应用的初始搭建以及开发过程。

人们把 Spring Boot 称为搭建程序的 **脚手架**。其最主要作用就是帮我们快速的构建庞大的 Spring 项目，并且尽可能的减少一切 XML 配置，做到开箱即用，迅速上手，让我们关注与业务而非配置。

该框架非常火，目前新开项目几乎都是基于 Spring Boot 搭建，非常符合微服务架构要求，企业招聘大多都要求有 Spring Boot 开发经验，属于面试必问的点。

1、优点

- 创建独立运行的 Spring 应用程序；
- 可嵌入 Tomcat，无需部署 war 文件；
- 简化 Maven 配置；
- 自动配置 Spring；
- 提供生产就绪型功能，如：日志，健康检查和外部配置等；
- 不要求配置 XML；
- 非常容易和第三方框架集成起来。

2、缺点

- 版本更新较快，可能出现较大变化；
- 因为约定大于配置，所以经常会出现一些很难解决的问题。

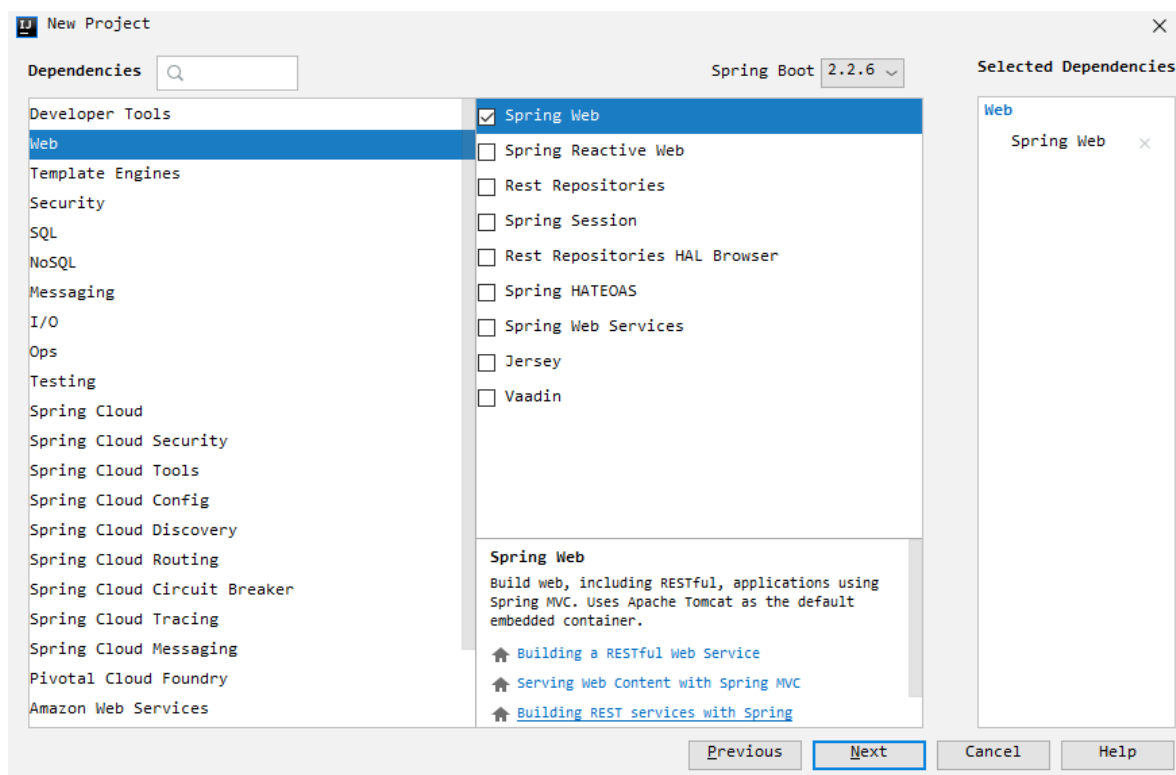
三、Spring Boot 快速入门（掌握）

1、使用 IDEA 创建 Spring Boot 工程

Spring Boot 建议使用官方提供的[工具](#)来快速构建项目。IDEA 自带该功能，但需要联网使用。

注意：官方提供的构建工具默认只能选择固定的版本，有些版本之间的差异非常大，所以如果需要选择某个版本可以自行在 pom.xml 文件中修改版本。

1.1、勾选依赖



1.2、编写 Controller 代码

```
@Controller
public class HelloController {

    @RequestMapping("/hello")
    @ResponseBody
    public String hello() {
        return "Hello Spring Boot";
    }
}
```

然后通过 main 方法启动程序，观察控制台输出内容，最后浏览器中输入 <http://localhost:8080/hello> 验证效果。

1.3、启动类和测试类

使用 IDEA 创建的项目会自动生成一个启动类，其实本质也是一个配置类，如下：

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

使用 IDEA 创建的项目会自动生成一个测试类，测试类贴有 `@SpringBootTest` 注解，可以通过通过注解属性指定加载的配置类，若没有指定，默认加载的是贴 `@SpringBootApplication` 注解的配置类，如下：

```
@SpringBootTest
class XxxApplicationTest {
    // ...
}
```

2、创建普通 Maven 工程

2.1、添加依赖

```
<!-- 打包方式 jar -->
<packaging>jar</packaging>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.3.RELEASE</version>
</parent>

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

2.2、编写 Controller 代码

```
@Controller
public class HelloController {

    @RequestMapping("/hello")
    @ResponseBody
    public String hello() {
        return "Hello Spring Boot";
    }
}
```

2.3、编写启动程序

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

然后通过 main 方法启动程序，观察控制台输出内容，最后浏览器中输入 <http://localhost:8080/hello> 验证效果。

3、疑问

- 当前项目继承的 spring-boot-starter-parent 项目有什么用？
- 导入的依赖 spring-boot-starter-web 有什么用？
- 占用 8080 端口的 Tomcat9 服务器哪来的？
- 之前的 Web 应用打包是 war，为什么现在的打包方式是 jar？
- @SpringBootApplication 注解有什么用？
- main 方法中执行的代码 SpringApplication.run(..) 有什么用？

四、入门案例分析（理解）

1、spring-boot-starter-parent

Spring Boot 提供了一个名为 spring-boot-starter-parent 的工程，里面已经对各种常用依赖（并非全部）的版本进行了管理，我们的项目需要以这个项目为父工程，这样我们就不用操心依赖的版本问题了，需要什么依赖，直接引入坐标即可！

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.3.3.RELEASE</version>
</parent>
```

继承是 Maven 中很强大的一种功能，继承可以使得子 pom 可以获得 parent 中的部分配置（groupId，version，dependencies，build，dependencyManagement 等），可以对子 pom 进行统一的配置和依赖管理。

- parent 项目中的 dependencyManagement 里的声明的依赖，只具有声明的作用，并不实现引入，因此子项目需要显式的声明需要用的依赖。如果不在子项目中声明依赖，是不会从父项目中继承下来的；只有在子项目中写了该依赖项，并且没有指定具体版本，才会从父项目中继承该项，并且 version 和 scope 都读取自父 pom；另外若子项目中指定了版本号，那么会使用子项目中指定的 jar 版本。
- parent 项目中的 dependencies 里声明的依赖会被所有的子项目继承。

2、Spring Boot Starter

Spring Boot 非常优秀的地方在于提供了非常多以 spring-boot-starter-* 开头的开箱即用的 starter 启动器（依赖包），使得我们在开发业务代码时能够非常方便的、不需要过多关注框架的配置，而只需要关注业务即可。

Spring Boot 在配置上相比 Spring 要简单许多，其核心在于 spring-boot-starter，在使用 Spring Boot 来搭建一个项目时，只需要引入官方提供的 starter，就可以直接使用，免去了各种配置。

官方目前已提供的常见的 Starter 如下：

spring-boot-starter：核心启动器，提供了自动配置，日志和 YAML 配置支持。

spring-boot-starter-aop：支持使用 Spring AOP 和 AspectJ 进行切面编程。

spring-boot-starter-freemarker：支持使用 FreeMarker 视图构建 Web 应用。

spring-boot-starter-test：支持使用 JUnit，测试 Spring Boot 应用。

spring-boot-starter-web：支持使用 Spring MVC 构建 Web 应用，包括 RESTful 应用，使用 Tomcat 作为默认的嵌入式容器。

spring-boot-starter-actuator：支持使用 Spring Boot Actuator 提供生产级别的应用程序监控和管理功能。

spring-boot-starter-logging：提供了对日志的支持，默认使用 Logback。

有关 Spring Boot Starter 命名规范，所有官方发布的 Starter 都遵循以下命名模式：spring-boot-starter-*, 其中 * 指特定的应用程序代号或名称。任何第三方提供的 Starter 都不能以 spring-boot 作为前缀，应该将应用程序代号或名称作为前缀，譬如 mybatis-spring-boot-starter。

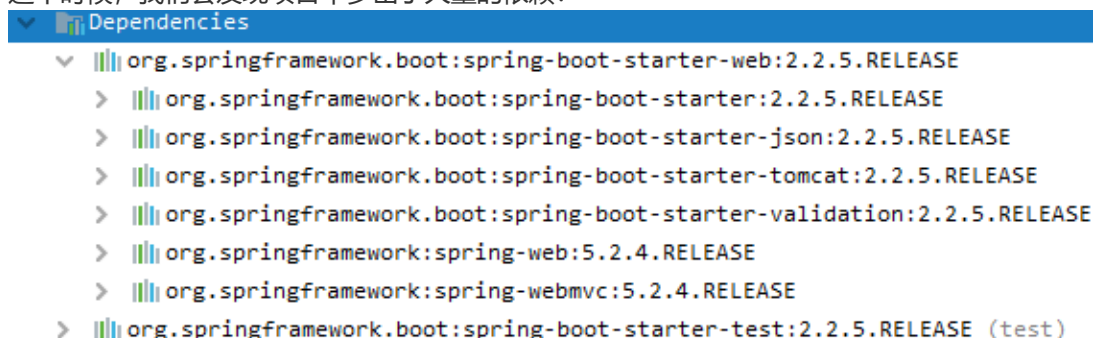
3、Web 启动器

这是 Spring Boot 提供的 Web 启动器，是一个快速集成 Web 模块的工具包，包含 Spring MVC，Jackson 相关的依赖，以及嵌入了 Tomcat9 服务器，默认端口 8080。

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

需要注意的是，我们并没有在这里指定版本信息。因为 Spring Boot 的父工程已经对版本进行了管理了。

这个时候，我们会发现项目中多出了大量的依赖：



```
Dependencies
└─ org.springframework.boot:spring-boot-starter-web:2.2.5.RELEASE
   └─ org.springframework.boot:spring-boot-starter:2.2.5.RELEASE
      └─ org.springframework.boot:spring-boot-starter-json:2.2.5.RELEASE
         └─ org.springframework.boot:spring-boot-starter-tomcat:2.2.5.RELEASE
            └─ org.springframework.boot:spring-boot-starter-validation:2.2.5.RELEASE
               └─ org.springframework:spring-web:5.2.4.RELEASE
                  └─ org.springframework:spring-webmvc:5.2.4.RELEASE
                     └─ org.springframework.boot:spring-boot-starter-test:2.2.5.RELEASE (test)
```

这些都是 Spring Boot 根据 spring-boot-starter-web 这个依赖自动引入的，而且所有的版本都已经管理好，不会出现冲突。

4、打包独立运行

对于 Spring Boot 项目来说无论是普通应用还是 Web 应用，其打包方式都是 jar 即可，当然 Web 应用也能打 war 包，但是需要额外添加许多插件来运行，比较麻烦。

默认的 Maven 打包方式是不能正常的打包 Spring Boot 项目的，需要额外的引入打包插件，才能正常的对 Spring Boot 项目打包，以后只要拿到该 jar 包就能脱离 IDE 工具独立运行了。

```
<!-- pom.xml 中添加插件 -->
<build>
  <plugins>
    <!-- Spring Boot 打包插件 -->
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

- 使用 maven 的 package 命令进行打包；
- 使用命令 `java -jar xxx.jar` 运行 jar 包 (`--server.port=80`) 。

五、Spring Boot 参数配置（掌握）

1、参数来源

- 命令行启动项目时传入的参数，如：`java -jar xxx.jar --server.port=80`；
- `application.properties` 或者 `application.yml` 文件。

一般用的比较多的就是直接在 `application.properties` 或者 `application.yml` 配置，其次是命令行启动方式。

1.1、application.properties 语法

```
server.port=80
server.session-timeout=30
server.tomcat.uri-encoding=UTF-8

spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/rbac
spring.datasource.username=root
spring.datasource.password=admin
```

1.2、application.yml 语法

```
server:
  port: 80
  session-timeout: 30
  tomcat.uri-encoding: UTF-8

spring:
  datasource:
    url: jdbc:mysql://localhost:3306/crm
    username: root
    password: admin
    driverClassName: com.mysql.jdbc.Driver
```

2、配置优先级（了解）

一个项目中可以有多个配置文件存放在不同目录中，此时他们会遵循固定的优先级来处理有冲突的属性配置，优先级由高到底，高优先级的配置会覆盖低优先级的配置。用 application.properties 文件举例子，下面文件优先级由高到低排序：

- 项目/config/application.properties
- 项目/application.properties
- classpath:config/application.properties
- classpath:application.properties

一般都在 **classpath:application.properties** 做配置，其他方式不使用。

3、参数属性绑定

通过配置参数，来自定义程序的运行。一般配置参数编写 application.properties 或者我们自定义的 properties 文件中。

3.1、参数配置在自定义的 properties

回顾之前使用 XML 配置时，想让 Spring 知道我们指定自定义的 properties 文件，就需要如下配置：

```
# db.properties
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/rbac
jdbc.username=root
jdbc.password=admin
```

```
<context:property-placeholder location="classpath:db.properties" system-
properties-mode="NEVER"/>
```

而现在使用 JavaConfig 配置，就得使用 `@PropertySource` + `@Value` 两个注解配合完成。`@PropertySource` 的作用就等价于上面那段 XML 配置。

```
/**
 * @PropertySource: 把属性配置加载到 Spring 的环境对象中
 * @Value: 从 Spring 环境对象中根据 key 读取 value
 */
@Configuration
@PropertySource("classpath:db.properties")
public class JavaConfig {
    @Value("${jdbc.driverClassName}")
    private String driverClassName;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String username;
    @Value("${jdbc.password}")
    private String password;

    @Bean
    public MyDataSource dataSource() {
        MyDataSource dataSource = new MyDataSource();
        dataSource.setDriverClassName(driverClassName);
    }
}
```

```

        dataSource.setUrl(url);
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        return dataSource;
    }
}

```

3.2、参数配置在 application.properties

准备好 application.properties 和一个类 MyDataSource，配置如下：

```

# application.properties
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/crm
jdbc.username=root
jdbc.password=admin

```

```

public class MyDataSource {
    private String driverClassName;
    private String url;
    private String username;
    private String password;

    // 省略 toString 方法
}

```

3.2.1、@Value 绑定单个属性

在自定义的类上绑定属性如下：

```

@Component
public class MyDataSource {
    @Value("${jdbc.driverClassName}")
    private String driverClassName;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String username;
    @Value("${jdbc.password}")
    private String password;

    // 省略 toString 方法
}

@Configuration
@ComponentScan("上面类所在的包路径")
public class JavaConfig { }

```

在配置类上绑定属性如下：

```

@Configuration
public class JavaConfig {
    // @Value: 从 Spring 环境对象中根据 key 读取 value
    @Value("${jdbc.driverClassName}")
    private String driverClassName;
}

```

```

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

@Bean
public MyDataSource dataSource() {
    MyDataSource dataSource = new MyDataSource();
    dataSource.setDriverClassName(driverClassName);
    dataSource.setUrl(url);
    dataSource.setUsername(username);
    dataSource.setPassword(password);
    return dataSource;
}
}

```

3.2.2、@ConfigurationProperties 绑定对象属性

若觉得上面的方式比较笨重，可以把前缀编写到 `@ConfigurationProperties` 属性上，并且设置类属性与需要绑定的参数名相同，可实现自动绑定，但是**注意**，若是使用测试类加载贴有 `@Configuration` 的配置类，则需要在配置类中添加 `@EnableConfigurationProperties` 注解；若是使用测试类加载贴有 `@SpringBootApplication` 的配置类，则不需要。

```

@Component
@ToString
@Setter
@ConfigurationProperties(prefix="jdbc")
public class MyDataSource {
    private String driverClassName;
    private String url;
    private String username;
    private String password;
}

```

或者像下面这样配置：

```

@Bean
@ConfigurationProperties("jdbc")
public MyDataSource dataSource() {
    return new MyDataSource();
}

```

`@EnableConfigurationProperties` 文档中解释：当 `@EnableConfigurationProperties` 注解应用到你的 `@Configuration` 时，任何贴 `@ConfigurationProperties` 注解的 beans 将自动被 Environment 进行属性绑定。

3.3、使用 Spring 的 Environment 对象绑定属性

当要绑定的参数过多时，直接在配置类中注入 Spring 的 Environment 对象，这样就不需要贴在字段或者形参上太多的 `@Value` 注解，相对比较简洁。

从 Environment 对象中可以获取到 `application.properties` 里面的参数，也可以获取到 `@PropertySource` 中的参数（即对配置在什么文件中没有要求）。


```

@Configuration
@PropertySource("classpath:db.properties")
public class JavaConfig {
    /**
     * environment: 表示 Spring 的环境对象，该对象包含了加载的属性数据
     * 可以获取到 application.properties 里面的参数，也可以获取到 @PropertySource 中的
    参数
     * 但 application.properties 的优先级比 @PropertySource 高
     */
    @Autowired
    private Environment environment;

    @Bean
    public MyDataSource dataSource() {
        MyDataSource dataSource = new MyDataSource();

        dataSource.setDriverClassName(environment.getProperty("jdbc.driverClassName"));
        dataSource.setUrl(environment.getProperty("jdbc.url"));
        dataSource.setUsername(environment.getProperty("jdbc.username"));
        dataSource.setPassword(environment.getProperty("jdbc.password"));
        return dataSource;
    }
}

```

六、Spring Boot 自动配置原理（了解）

使用 Spring Boot 之后，做一个整合了 Spring MVC 的 Web 工程开发，变的无比简单，那些繁杂的配置都消失不见了，这是如何做到的？一切魔力的开始，都是从我们的 main 函数来的，所以我们再次来看下启动类：

```

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

```

从上面代码我们发现特别的地方有两个：

- 注解：@SpringBootApplication；
- run方法：SpringApplication.run()。

1、了解 @SpringBootApplication

点击进入，查看源码：

```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
    @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {

```

这里重点的注解有3个：

- @SpringBootConfiguration
- @EnableAutoConfiguration
- @ComponentScan

1.1、@SpringBootConfiguration

```
/**
 * Indicates that a class provides Spring Boot application
 * {@link Configuration @Configuration}. Can be used as an alternative to the Spring's
 * standard {@code @Configuration} annotation so that configuration can be found
 * automatically (for example in tests).
 * <p>
 * Application should only ever include <em>one</em> {@code @SpringBootConfiguration} and
 * most idiomatic Spring Boot applications will inherit it from
 * {@code @SpringBootApplication}.
 *
 * @author Phillip Webb
 * @since 1.4.0
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {
}
```

通过这段我们可以看出，在这个注解上面，又有一个 @Configuration 注解。通过上面的注释阅读我们知道：这个注解的作用就是声明当前类是一个配置类，然后 Spring 会自动扫描到添加了 @Configuration 的类，并且读取其中的配置信息。而 @SpringBootConfiguration 是用来声明当前类是 Spring Boot 应用的配置类，项目中只能有一个。所以一般我们无需自己添加。

1.2、@ComponentScan

我们的 @SpringBootApplication 注解声明的类就是 main 函数所在的启动类，因此扫描的包是该类所在包及其子包。因此，一般启动类会放在一个比较前的包目录中。

1.3、@EnableAutoConfiguration

@EnableAutoConfiguration 的作用，告诉 Spring Boot 基于你所添加的依赖，去“猜测”你想要如何配置 Spring。比如我们引入了 spring-boot-starter-web，而这个启动器中帮我们添加了 tomcat、SpringMVC 的依赖。此时自动配置就知道你是要开发一个 Web 应用，所以就帮你完成了 Web 及 Spring MVC 的默认配置了！

Spring Boot 内部对大量的第三方库或 Spring 内部库进行了默认配置，这些配置是否生效，取决于我们是否引入了对应库所需的依赖，如果有，那么默认配置就会生效。那么带来新的问题，如下：

- 这些默认配置是在哪里定义的呢？
- 为何依赖引入就会触发配置呢？

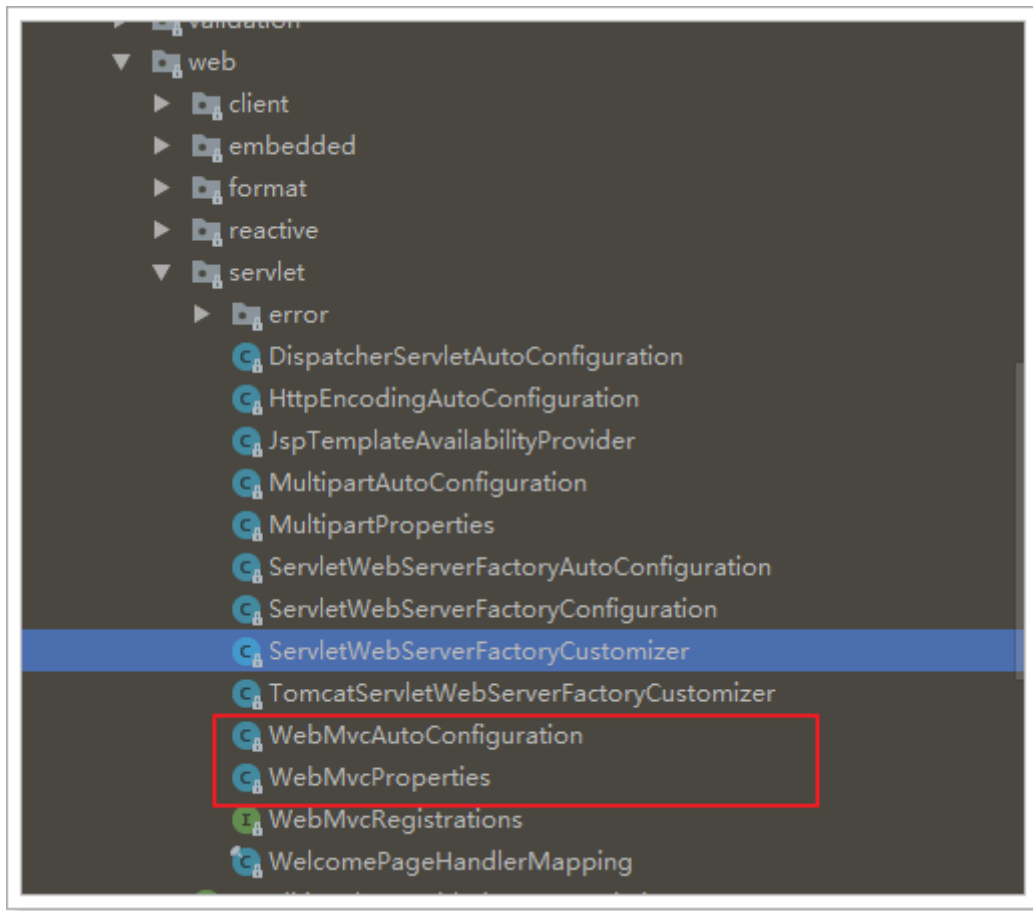
2、如何自动配置

@EnableAutoConfiguration 注解中导入了一个 AutoConfigurationImportSelector 配置类，该类中有个 getCandidateConfigurations 方法，方法的作用是委托 SpringFactoriesLoader 去读取 jar 包中的 META-INF/spring.factories 文件，并加载里面配置的自动配置对象，包括：AOP，PropertyPlaceholder，FreeMarker，HttpMessageConverter，Jackson，DataSourceDataSourceTransactionManager，DispatcherServlet 等等。

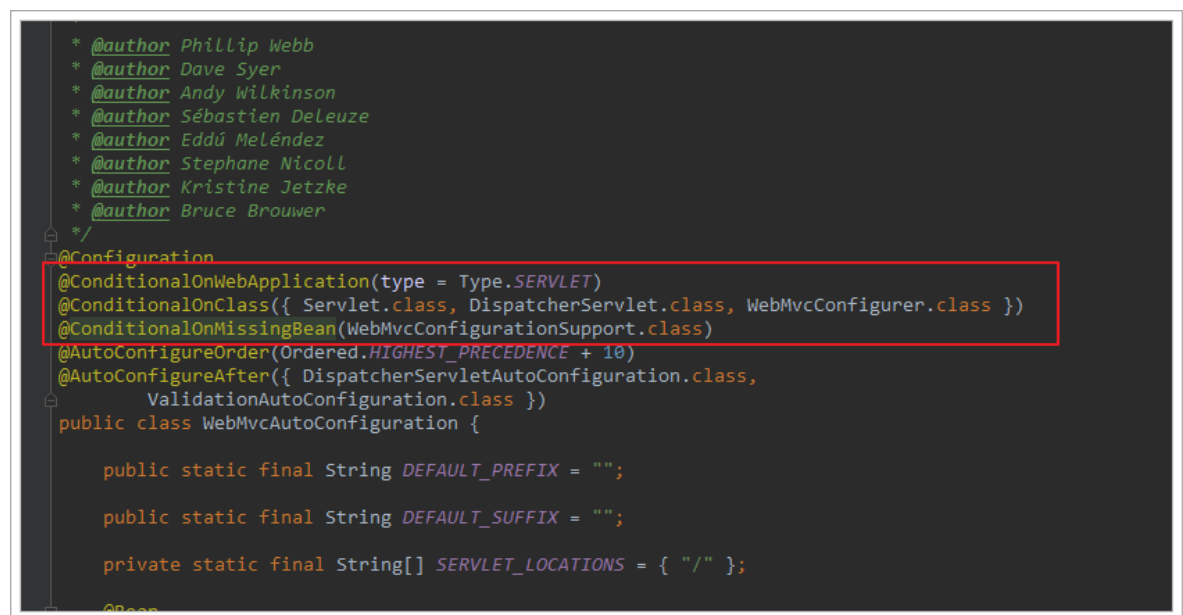
其实在我们的项目中，已经自动引入了一个依赖：spring-boot-autoconfigure，上面说到的这些自动配置类，都来自该包。

虽然 spring.factories 文件中定义了很多配置类，但并不是所有配置类都会生效，只有符合条件的是才会自动装配。

我们来看一个我们熟悉的，例如 Spring MVC，查看其自动配置类：



打开 WebMvcAutoConfiguration：



我们看到这个类上的 4 个注解：

- `@Configuration`：声明这个类是一个配置类
- `@ConditionalOnWebApplication(type = Type.SERVLET)`

ConditionalOn，翻译就是在某个条件下，此处就是满足项目的类是是 Type.SERVLET 类型，我们现在的项目就满足了，就是一个 Web 工程。

- `@ConditionalOnClass({Servlet.class, DispatcherServlet.class, WebMvcConfigurer.class })`

这里的条件是 OnClass，也就是满足以下类存在：Servlet、DispatcherServlet、WebMvcConfigurer。这里就是判断你是否引入了 Spring MVC 相关依赖，引入依赖后该条件成立，当前类的配置才会生效！

- `@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)`

这个条件与上面不同，OnMissingBean，是说环境中没有指定的 bean 这个才生效。其实这就是自定义配置的入口，也就是说，如果我们自己配置了一个 WebMvcConfigurationSupport 的 bean，代表容器里已经存在该 bean 了，那么这个默认配置就会失效！

接着，我们查看 WebMvcAutoConfiguration 该类中定义了什么：

视图解析器：

```
@Bean
@ConditionalOnMissingBean
public InternalResourceViewResolver defaultViewResolver() {
    InternalResourceViewResolver resolver = new InternalResourceViewResolver();
    resolver.setPrefix(this.mvcProperties.getView().getPrefix());
    resolver.setSuffix(this.mvcProperties.getView().getSuffix());
    return resolver;
}

@Bean
@ConditionalOnBean(View.class)
@ConditionalOnMissingBean
public BeanNameViewResolver beanNameViewResolver() {
    BeanNameViewResolver resolver = new BeanNameViewResolver();
    resolver.setOrder(Ordered.LOWEST_PRECEDENCE - 10);
    return resolver;
}
```

WebMvcAutoConfiguration 中使用了 `@AutoConfigureAfter` 注解，意为指定的类加载完了后，再加载本类。

```
@AutoConfigureAfter({ DispatcherServletAutoConfiguration.class,
    TaskExecutionAutoConfiguration.class, ValidationAutoConfiguration.class })
```

而 DispatcherServletAutoConfiguration 中又做了很多事情，比如配置了前端控制器：

```
@Bean(name = DEFAULT_DISPATCHER_SERVLET_BEAN_NAME)
public DispatcherServlet dispatcherServlet(webMvcProperties webMvcProperties) {
    DispatcherServlet dispatcherServlet = new DispatcherServlet();

    dispatcherServlet.setDispatchOptionsRequest(webMvcProperties.isDispatchOptionsRequest());

    dispatcherServlet.setDispatchTraceRequest(webMvcProperties.isDispatchTraceRequest());

    dispatcherServlet.setPublishEvents(webMvcProperties.isPublishRequestHandledEvents());

    dispatcherServlet.setEnableLoggingRequestDetails(webMvcProperties.isLogRequestDetails());
    return dispatcherServlet;
}
```

3、总结

- `@SpringBootApplication` 注解内部是3大注解功能的集成
 - `@ComponentScan`：开启组件扫描
 - `@SpringBootConfiguration`：作用等同于 `@Configuration` 注解，也是用于标记配置类
 - `@EnableAutoConfiguration`：内部导入 `AutoConfigurationImportSelector`，该类中有个 `getCandidateConfigurations` 方法，读取 jar 包中 `META-INF/spring.factories` 文件中配置类，再根据条件进行加载和配置，比如：AOP，PropertyPlaceholder，FreeMarker，HttpMessageConverter，Jackson，DataSourceDataSourceTransactionManager，DispatcherServlet，WebMvc 等等
- `SpringApplication.run(..)`的作用
 - 启动 Spring Boot 应用
 - 加载自定义的配置类，完成自动配置功能
 - 把当前项目配置到嵌入的 Tomcat 服务器
 - 启动嵌入的 Tomcat 服务器

七、Spring Boot 实战（掌握）

现在我们来把之前上课的项目改造为 Spring Boot 项目。

1、迁移 ssm 项目

1.1、快速创建 Spring Boot 项目

Project Metadata

Group:

Artifact:

Type:

Language:

Packaging:

Java Version:

Version:

Name:

Description:

Package:

1.2、添加依赖

如果是普通 Maven 项目，需要手动添加。

```
<!-- 打包方式 jar 包 -->
<packaging>jar</packaging>

<!-- 指定父工程 -->
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.3.3.RELEASE</version>
</parent>

<dependencies>
  <!-- spring boot web 包 -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <!-- spring boot Test 包 -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
  <!-- Lombok -->
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

1.3、添加启动类

如果是普通 Maven 项目，需要手动添加。

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

1.4、拷贝项目代码

- 把 src\main\java 中的 java 代码全部拷贝到新项目中的 src\main\java 里面。
- 把 src\main\resources 中存放 mapper.xml 的目录也拷贝到新项目中的 src\main\resources 里面。

注意：按照下面步骤用什么拷贝什么。

2、配置数据库连接池

2.1、添加依赖

```
<!-- MySQL 驱动 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

2.2、自动配置方式

application.properties

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql:///rbac?
useUnicode=true&characterEncoding=utf8&serverTimezone=GMT%2B8
spring.datasource.username=root
spring.datasource.password=admin
```

此时运行测试获取的数据库连接池对象可以执行成功，并且我们看到了使用的连接池是 **Hikari**，全称是 Hikaricp。

```
2020-04-01 15:10:21.014 INFO 6824 --- [ Thread-2] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down ExecutorService 'applicat:
2020-04-01 15:10:21.016 INFO 6824 --- [ Thread-2] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Shutdown initiated...
2020-04-01 15:10:21.019 INFO 6824 --- [ Thread-2] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Shutdown completed.
```

其实，在 Spring Boot 2.0 之后，采用的默认连接池就是 **Hikari**，号称“史上最快的连接池”，所以我们没有添加依赖也能直接用，Spring Boot 的自动配置中含有 DataSourceAutoConfiguration 配置类，会先检查容器中是否已经有连接池对象，没有则会使用默认的连接池，并根据特定的属性来自动配置连接池对象，用到的属性值来源于 DataSourceProperties 对象。

2.3、配置 Druid 连接池

当然如果我们在项目中还是想要使用 **Druid** 作为连接池也是可以的。只需要添加依赖即可，此时加的是 **Druid** 的 druid-spring-boot-starter 自动配置包，里面包含了 DruidDataSourceAutoConfigure 自动配置类，会自动创建 Druid 的连接池对象，所以 Spring Boot 发现已经有连接池对象了，则不会再使用 **Hikari**。

```

<!-- druid -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.21</version>
</dependency>

```

```

2020-04-01 15:36:52.404 INFO 7296 --- [ Thread-2] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down ExecutorService 'applicationTaskExecutor'
2020-04-01 15:36:52.406 INFO 7296 --- [ Thread-2] com.alibaba.druid.pool.DruidDataSource : {dataSource-1} closing ...
2020-04-01 15:36:52.408 INFO 7296 --- [ Thread-2] com.alibaba.druid.pool.DruidDataSource : {dataSource-1} closed

```

注意：如果添加的依赖是以前那种普通包，只有 Druid 自身的依赖，并不是自动配置包，则需要进行以下配置：

```

<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.19</version>
</dependency>

```

```
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
```

所以一般如果已经提供了 Spring Boot 相关的自动配置包，直接使用自动配置的会更方便些。

对于 Hikari 以及 Druid 两款都是开源产品，阿里的 Druid 有中文的开源社区，交流起来更加方便，并且经过阿里多个系统的实验，想必也是非常的稳定，而 Hikari 是 Spring Boot 2.0 默认的连接池，全世界使用范围也非常广，对于大部分业务来说，使用哪一款都是差不多的，毕竟性能瓶颈一般都不在连接池。

3、集成 MyBatis

3.1、添加依赖

```

<!-- Mybatis 集成到 SpringBoot 中的依赖 -->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.1</version>
</dependency>

```

3.2、配置 Mapper 对象

扫描 Mapper 接口只要在配置类上贴个注解 `@MapperScan` 并指定扫描的包路径即可。

```

@SpringBootApplication
@MapperScan("cn.wolfcode.mapper")
public class SsmApplication {
    public static void main(String[] args) {
        SpringApplication.run(SsmApplication.class, args);
    }
}

```

3.3、配置属性

在 application.properties 配置以前在 XML 配置了的那些 MyBatis 的属性，属性前缀 mybatis。

```
# 配置别名
mybatis.type-aliases-package=cn.wolfcode.domain

# 打印 SQL 日志
logging.level.cn.wolfcode.mapper=trace
```

4、事务管理

4.1、添加依赖

```
<!-- Spring JDBC 和 TX -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

4.2、XML方式（了解）

采取配置类和 XML 混用的策略，在配置类上使用 @ImportResource("classpath:spring-tx.xml")。

4.3、注解方式

直接在业务层实现类上或者其方法上直接贴 @Transactional 注解即可。

Spring Boot 自动配置中提供了 TransactionAutoConfiguration 事务注解自动配置类，引入了事务的依赖后，可直接使用 @Transactional 注解。

4.4、配置代理实现

Spring Boot 默认优先选择 CGLIB 代理，如果需要改为优先使用 JDK 代理，需要做以下配置：

```
spring.aop.proxy-target-class=false
```

4.5、测试验证

在测试类上添加方法，打印业务对象看下其真实类型。

```
@Test
public void testSave() {
    System.out.println(departmentService.getClass());
}
```

5、集成 Web

5.1、添加依赖

```
<!-- spring boot web 启动器（之前已添加了） -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

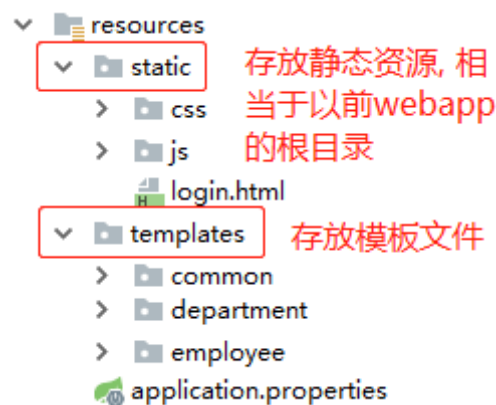
5.1、修改端口

在 application.properties 配置，如下：

```
server.port=80
```

5.2、目录结构

拷贝静态资源及模板文件



5.3、静态资源处理

- 默认情况下，Spring Boot 会从 classpath 下的 /static, /public, /resources, /META-INF/resources 下载静态资源。
- 可以在 application.properties 中配置 spring.resources.staticLocations 属性来修改静态资源加载地址。
- 因为应用是打成 jar 包，所以之前的 src/main/webapp 就作废了，如果有文件上传，那么就必须去配置图片所在的路径。

```
# 告诉 Spring Boot 什么访问的路径是找静态资源
spring.mvc.static-path-pattern=/static/**
```

5.3、前端控制器映射路径配置（了解）

在 Spring Boot 自动配置中，WebMvcAutoConfiguration 自动配置类导入了 DispatcherServletAutoConfiguration 配置对象，会自动创建 DispatcherServlet 前端控制器，默认的映射路径是 /，Spring Boot 多数用于前后端分离和微服务开发，默认支持 RESTful 风格，所以一般都是使用默认的即可，不做改动。

```
# 在匹配模式时是否使用后缀模式匹配
spring.mvc.pathmatch.use-suffix-pattern=true
```

6、集成 Thymeleaf

6.1、添加依赖

```
<!-- 引入 Thymeleaf 依赖 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

6.2、相关配置

```
# Thymelea 模板配置
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
spring.thymeleaf.mode=HTML
spring.thymeleaf.encoding=UTF-8

# 热部署文件，页面不产生缓存，及时更新
spring.thymeleaf.cache=false
```

6.3、修改模板文件

使用 Thymeleaf 的语法替换之前 JSP 中的 EL 表达式和 JSTL。

7、统一异常处理

7.1、框架自带方式

Spring Boot 默认情况下，会把所有错误都交给 `BasicErrorController` 类完成处理，错误的视图导向到 `classpath:/static/error/` 和 `classpath:/templates/error/` 路径上，HTTP 状态码就是默认视图的名称。如：

- 出现 404 错误 -> `classpath:/static/error/404.html`
- 出现 5xx 错误 -> `classpath:/static/error/5xx.html`

7.2、控制器增强器方式

自己定义一个控制器增强器，专门用于统一异常处理，该方式一般用于 5xx 类错误。

```
@ControllerAdvice // 控制器增强器
public class ExceptionControllerAdvice {
    @ExceptionHandler(RuntimeException.class) // 处理什么类型的异常
    public String handleException(RuntimeException e, Model model) {
        e.printStackTrace(); // 记得这行代码保留，不然项目后台出异常，开发工具控制台看不多
        错误信息
        return "errorview"; // 指定错误页面视图名称
    }
}
```

8、添加拦截器

在传统的 XML 方式中，我们需要在 `<mvc: interceptors>` 标签中去注册我们自定义的拦截器。在 Spring Boot 中，提供了 `WebMvcConfigurer` 配置接口，是使用 `JavaConfig` 配置 Spring MVC 的标准，如果我们需要对 Spring MVC 做配置，则需要让我们自定义配置类实现该接口，若是需要注册拦截器，则实现接口中的 `addInterceptors` 方法即可。

8.1、编写拦截器

让 Spring 管理拦截器 bean 对象，在拦截器类上贴 `@Component` 注解。

```
@Component
public class LoginInterceptor implements HandlerInterceptor {
    // ...
}
```

```
@Component
public class PermissionInterceptor implements HandlerInterceptor {
    // ...
}
```

8.2、配置拦截器

定义一个配置类，实现 `WebMvcConfigurer` 接口，在 `addInterceptors` 方法注册拦截器。

```
@Configuration
public class MvcJavaConfig implements WebMvcConfigurer {

    @Autowired
    private LoginInterceptor loginInterceptor;

    @Autowired
    private PermissionInterceptor permissionInterceptor;

    public void addInterceptors(InterceptorRegistry registry) {
        // 注册登录拦截器
        registry.addInterceptor(loginInterceptor)
            // 对哪些资源起过滤作用
            .addPathPatterns("/**")
            // 对哪些资源起排除作用
            .excludePathPatterns("/login", "/static/**");
        // 注册登录拦截器
        registry.addInterceptor(permissionInterceptor)
            // 对哪些资源起过滤作用
            .addPathPatterns("/**")
            // 对哪些资源起排除作用
            .excludePathPatterns("/login", "/static/**");
    }
}
```

八、系统日志（了解）

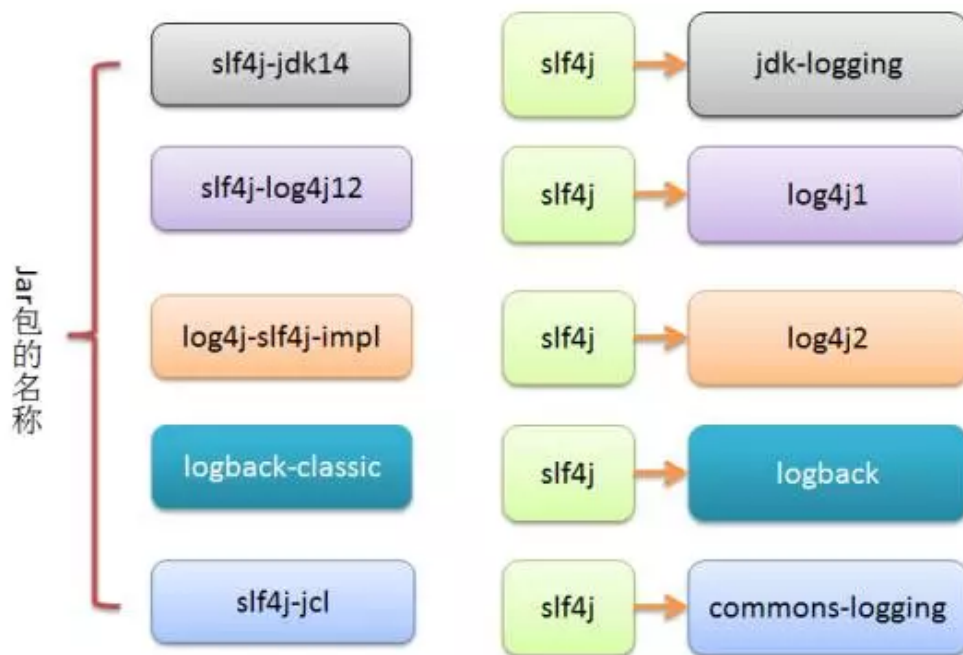
1、为什么要用日志

- 比起 `System.out.println`，日志框架更为灵活，可以把日志的输出和代码分离。

- 日志框架可以方便的定义日志的输出环境，控制台，文件，数据库。
- 日志框架可以方便的定义日志的输出格式和输出级别。

2、Spring Boot 中的日志介绍

- Spring Boot 默认已经开启日志，其默认的日志格式为：时间 日志级别 线程ID 线程名称 日志类 日志说明。
- Spring Boot 的日志分为：系统日志和应用日志。
- 日志级别，级别越高，输出的内容越少，如果设置的级别为 info，则 debug 以及 trace 级别的都无法显示，日志级别由低到高 trace < debug < info < warn < error。
- Spring Boot 默认选择 Logback 作为日志框架，也能选择其他日志框架，但是没有必要。



3、输出日志

在我们自定义的类中可以使用日志框架来输出。

3.1、方式一

在类中定义一个静态 Logger 对象，传入当前类的作用是方便输出日志时可以清晰地看到该日志信息是属于哪个类的。

```
private static final Logger log = LoggerFactory.getLogger(当前类.class);
```

3.2、方式二

使用 Lombok 提供的 @Slf4j 注解来简化代码，其实和方式一的作用是一样的。

```
@Slf4j
@Service
public class PermissionServiceImpl implements IPermissionService {}
```

在需要输出日志的地方使用日志的输出方法

```
log.info(...);
log.error(...);
...
// 输出日志中有变量可以使用 {} 作为占位符
log.info("删除id为{}的数据", id);
```

4、日志级别

我们来尝试一下日志级别：

```
log.debug("权限插入成功: {}", expression);
log.info("权限插入成功: {}", expression);
log.warn("权限插入成功: {}", expression);
```

执行权限加载功能后，发现控制台出现 info 与 warn 的信息，debug 的没有显示，原因是因为 Spring Boot 默认的日志级别是 info，所以 debug 低于 info 级别，就不会显示出来了。

```
2020-04-01 21:53:54.413 DEBUG 5492 --- [nio-8082-exec-9] c.w.crm.mapper.PermissionMapper.insert : ==> Preparing: insert into permission (name, expression) values (?, ?)
2020-04-01 21:53:54.414 DEBUG 5492 --- [nio-8082-exec-9] c.w.crm.mapper.PermissionMapper.insert : ==> Parameters: department:list(String), department:list(String)
2020-04-01 21:53:54.424 DEBUG 5492 --- [nio-8082-exec-9] c.w.crm.mapper.PermissionMapper.insert : <== Updates: 1
2020-04-01 21:53:54.426 INFO 5492 --- [nio-8082-exec-9] c.w.e.s.impl.PermissionServiceImpl : 权限插入成功: department:list
2020-04-01 21:53:54.428 WARN 5492 --- [nio-8082-exec-9] c.w.e.s.impl.PermissionServiceImpl : 权限插入成功: department:list
2020-04-01 21:53:55.630 DEBUG 5492 --- [nio-8082-exec-1] c.w.e.m.P.selectForList_COUNT : ==> Preparing: SELECT count(0) FROM permission
2020-04-01 21:53:55.631 DEBUG 5492 --- [nio-8082-exec-1] c.w.e.m.P.selectForList_COUNT : ==> Parameters:
2020-04-01 21:53:55.632 TRACE 5492 --- [nio-8082-exec-1] c.w.e.m.P.selectForList_COUNT : <== Columns: count(0)
```

若要修改日志级别，最快速的方式是在 application.properties 配置，配置如下

```
# 把日志级别修改为 debug，不过我们一般不会更改，除非要调试找 bug，不然控制台显示的内容太多也容易乱
logging.level.root=debug
```

5、Logback 配置文件的使用

Logback 框架默认会自动加载 classpath:logback.xml，作为框架的配置文件，在 Spring Boot 中使用时，还会额外的支持自动加载classpath:logback-spring.xml，在 Spring Boot 中推荐使用 logback-spring.xml，功能更强些。

样板文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    scan: 开启日志框架的热部署，默认值 true 表示开启
    scanPeriod: 热部署的频率，默认值 60 second
    debug: 设置输出框架内部的日志，默认值 false
-->
<configuration scan="true" scanPeriod="60 second" debug="false">
    <property name="appName" value="springboot demo" />
    <contextName>${appName}</contextName>

    <!-- appender: 日志输出对象，配置不同的类拥有不同的功能
         ch.qos.logback.core.ConsoleAppender: 日志输出到控制台
    -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd-HH: mm: ss} %level [%thread] -%logger{35} >>
            %msg %n</pattern>
        </encoder>
```

```

</appender>

<!-- ch.qos.logback.core.FileAppender: 日志输出到文件中
<appender name="fileAppender" class="ch.qos.logback.core.FileAppender">
    <encoder>
        <pattern>%-4relative [%thread] %level %logger{35} - %msg
%n</pattern>
    </encoder>
    <append>true</append>
    <file>mylog.log</file>
</appender>
-->

<!--
    root 是项目通用的 logger，一般情况下都是使用 root 配置的日志输出
    level: 按照级别输出日志，日志级别，级别越高，输出的内容越少
-->
<root level="info">
    <appender-ref ref="STDOUT" />
</root>

<!-- 自定义的 logger，用于专门输出特定包中打印的日志
<logger name="cn.wolfcode.crm.mapper" level="trace">
</logger>
-->
</configuration>

```

参考日志格式：

- %d{yyyy-MM-dd-HH: mm: ss} %level [%thread]-%class: %line >> %msg %n

格式中的标识符组成：

- %logger{n}: 输出 Logger 对象类名，n 代表长度
- %class{n}: 输出所在类名
- %d{pattern} 或者 date{pattern}: 输出日志日期，格式同 Java
- %L/line: 日志所在行号
- %m/msg: 日志内容
- %method: 所在方法名称
- %p/level: 日志级别
- %thread: 所在线程名称

八、其他功能（了解）

1、修改 banner

Spring Boot 提供了一些扩展点，比如修改 banner：在 resources 根目录中放入 banner.txt 文件，替换默认的 banner。

```

# application.properties
# 关闭 banner
spring.main.banner-mode=off

```

2、热部署插件

除了使用 JRebel 来实现热部署，还可以使用 Spring Boot 提供的 spring-boot-devtools 包来完成 Springboot 应用热部署。

```
<!-- Spring Boot 热部署插件 -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

Spring Boot 重启是 reload 重启，通过监控 classpath 的变化，如果 classpath 中的文件发生变化，即触发重启。Spring Boot 通过两个 classpath 来完成 reload，一个 basic classloader 中加载不变的类（jar 包中的类），一个 restart classloader 中加载 classpath 中的类（自己写的类），重启的时候，restart classloader 中的类丢弃并重新加载。

```
# 默认排除的资源
spring.devtools.restart.exclude=static/**, templates/**, public/**

# 增加额外的排除资源
# 处理默认配置排除之外的
spring.devtools.restart.additional-exclude=public/**

# 禁用自动重启
spring.devtools.restart.enabled=false
```

3、切换运行环境

在实际开发中，一个系统是有多套运行环境的，如开发时有开发的环境，测试时有测试的环境，不同的环境中，系统的参数设置是不同的，如：连接开发数据和测试数据库的 URL 绝对是不同的，那么怎么快速的切换系统运行的环境呢？我们需要为不同的环境创建不同的配置文件，如下：

```
# application-dev.properties
server.port=8081
```

```
# application-test.properties
server.port=8082
```

```
# 在 application.properties 中指定需要使用的环境即可
spring.profiles.active=dev
```

练习

- 基于 JavaConfig 使用 Spring 的 IoC DI 功能，之后再使用 IoC DI 注解简化配置。
- 当参数配置在 application.properties 中，如何在程序中获取配置的值。
- 在之前的 SSM 项目基础上，改用 Spring Boot 搭建，视图技术选用 Thymeleaf。