课前案例：https://gitee.com/huadahua/spring-lifecycle-boot-2206.git

- 目录：ppt

有内容的git：https://gitee.com/huadahua/spring-life-boot-hascontent-2206.git

内容大纲：

- Bean的生命周期 -- 理论+代码
- SpringBoot -- 理论

Bean生命周期相关的2个注解

- 三种依赖注入的顺序

```
构造注入-->字段注入-->setter注入
```

- @PostConstruct

```
该注解作用于方法，要求方法无参且返回值为void，该方法会在依赖注入之后被自动调用，作用为自定义初始化操作
```

- @PreDestroy

```
该注解作用于方法，要求方法无参且返回值为void，该方法在Bean销毁之前被自动调用
注意点：
    1. 若bean的scope不是singleton，则添加了该注解的方法在Bean销毁前不会被调用
        -- 若bean的作用域是prototype，每获取一次Bean，均重新创建新对象返回
    2. 若进程意外终止或进程被强制kill，则添加该注解的方法不会被调用
    3. JVM关闭，会触发hook（钩子），让context关闭，context关闭之前自动调用该注解的方法
```

- 通过@Bean的属性在Bean初始化和销毁前执行自定义操作

```
显示配置Bean时，可以通过添加@Bean的属性initMethod和destoryMethod来指定对应的方法。效果和@PostConstruct，@PreDestory是一样的
@Bean(initMethod="populateCache", destroyMethod="flushCache")

注意：两种方式选择问题
    1. 若是自定义的类，则可以通过隐式配置Bean，此时可以使用@PostConstruct和@PreDestory
    2. 若配置的不是自己写的类，则可以通过显示配置Bean，此时可以使用属性initMethod和destoryMethod
```

Bean的生命周期

初始化

- 加载并后处理Bean的定义

```
1. 加载Bean定义，将显示配置，隐式配置的Bean定义加载到BeanFactory（ApplicationContext）
    -- 每个bean都在id和类型下被索引
```

2. 后处理Bean的定义
    -- 修改BeanFactory中的Bean定义的属性
    -- 常见操作：1. 修改作用域 2. 读取属性 --@Value("${属性名}")
    --涉及到的接口： BeanFactoryPostProcessor ，简称BFPP

    --模拟如何在BFPP的实现类中修改属性 -- 作用域

```
public class MyBFPP implements BeanFactoryPostProcessor {
    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory
factory) throws BeansException {
        //修改UserService的作用域
        //从BeanFactory中获取对应的Bean定义
        BeanDefinition definition =
factory.getBeanDefinition("userService");
        definition.setScope("prototype");
    }
}
```

注意：后处理Bean定义结束，则Bean属性修改完成，加载Bean以及后处理Bean完全结束，接下来即将开始实例化Bean

实例化Bean以及初始化

1. 查找Bean的依赖
2. 实例化Bean
3. 依赖注入
4. 调用初始化器进行Bean的后处理
    可以在初始化之前和之后添加操作，通过接口BeanPostProcessor（BPP）完成--该阶段叫做Bean的后处理过程
    注意：BPP中重写两个方法后，两个方法会在每个Bean的初始化器之前和之后都调用

    BPP的应用：若想在所有的Bean初始化之前和之后统一的执行一些操作，此时可以通过接口BPP完成。

BPP代码测试

```
@Component
public class MyBeanPostProcessor implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String
beanName) throws BeansException {
        System.out.println(beanName+"的BPP的beforeInit方法执行。。。。。。");
        return BeanPostProcessor.super.postProcessBeforeInitialization(bean,
beanName);
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String
beanName) throws BeansException {
        System.out.println(beanName+"的BPP的afterInit方法执行。。。。。。");
        return BeanPostProcessor.super.postProcessAfterInitialization(bean,
beanName);
    }
}
```

SpringBoot

- 为什么用SpringBoot框架

搭建一个Spring框架项目，创建maven项目，然后在项目中倒入spring框架用到的依赖
    spring-core-xx   spring-context-xx spring-aop-xx
    ....,
在Spring框架中导入依赖时是需要指定版本号的，此时就可能会产生版本不兼容问题。
eg:导入SpringMVC的依赖，导入mybatis的依赖，

框架之间的整合问题：SSM SpringMVC Spring Mybatis，必须导入整合的依赖

在SPring框架中，若想使用框架提供的组件，必须在配置文件中进行显示配置，配置的Bean会非常多

```
@Bean
DataSource

@Bean
SqlSessionFactory
```

搭建Spring框架是非常繁琐的，且可能会产生各种问题（版本不兼容），而以上Spring所有的繁琐操作，SpringBoot框架都帮我们完成了，所以更倾向于使用SpringBoot框架

- SpringBoot框架描述

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。

在Spring框架这个大家族中，产生了很多衍生框架，比如 Spring、SpringMvc框架等，Spring的核心内容在于控制反转(IOC)和依赖注入(DI),所谓控制反转并非是一种技术，而是一种思想，在操作方面是指在spring配置文件中创建<bean>，依赖注入即为由spring容器为应用程序的某个对象提供资源。

SpringBoot是一个框架，一种全新的编程规范，他的产生简化了框架的使用，所谓简化是指简化了Spring众多框架中所需的大量且繁琐的配置文件，所以 SpringBoot是一个服务于框架的框架，服务范围是简化配置。

SpringBoot最明显的特点是，让文件配置变的相当简单、让应用部署变的简单（SpringBoot内置服务器，并装备启动类代码），可以快速开启一个Web容器进行开发。
  jar-java压缩包   war - web压缩包
  war部署到服务器，在服务器运行项目，运行war包项目需要各种环境，eg:tomcat,jdk,mysql--linux

      tomcat-->webapps tomcat启动，这个项目就启动了

Spring Boot的核心功能
1、 可独立运行的Spring项目：Spring Boot可以以jar包的形式独立运行。---略过
2、 内嵌的Servlet容器：Spring Boot可以选择内嵌Tomcat、Jetty或者Undertow，无须以war包形式部署项目。
3、 简化的Maven配置：Spring提供推荐的基础 POM 文件来简化Maven 配置。
4、 自动配置Spring：Spring Boot会根据项目依赖来自动配置Spring 框架，极大地减少项目要使用的配置。
5、 无代码生成和xml配置：Spring Boot不生成代码。完全不需要任何xml配置即可实现Spring的所有配置。

SpringBoot的核心

- 依赖管理

```
如何减少需要导入的依赖项以及如何解决版本兼容问题？
    父级pom -- 解决版本兼容问题
    starter依赖
        spring-boot-starter    解决16个jar
        spring-boot-starter-test    解决测试相关的jar包
```

- 自动配置

```
SpringBoot是如何自动配置Bean的？
    @EnableAutoConfiguration 会读取工厂文件，这个工厂文件中罗列自动配置类，SpringBoot会去执行这些自动配置类（自动配置类的目的是配置Bean），Bean是否配置需要考虑以下因素：
        1. classpath下的内容 -- @ConditionalOnClass(XX.class) @ConditionalOnMissingClass(XX.class)
        2. 设置的属性 -- 数据源Bean的配置    @ConditionalOnProperty
        3. 已定义/未定义的Bean    配置JdbcTemplate,必须存在数据源 @ConditionalOnBean
```

  - 自定义Bean和自动配置Bean的顺序

```
自定义Bean的配置一定优先于自动配置
```

  - 如何控制自动配置

```
1. 自己显示配置Bean，则SPringBoot不会再自动配置该类型的Bean
2. 设置属性
    属性定义在属性文件中    xx.properties
        一个项目中若既有application.properties/yml ,又有其他属性文件，则读取顺序是：
        先读取application.properties,然后读取其他属性文件，这里可能会存在属性覆盖问题。
    SpringBoot会自动配置日志相关的Bean，默认日志级别是Error级别，若想修改该Bean的日志级别，则可以在属性文件中进行修改，注意日志级别属性只能定义在application.properties中

3. 显示的禁用某些自动配置类
    可以通过注解@EnableAutoConfiguration(execlude=...class)--@SpringBootAppliation注解同样有该属性
    或者在属性文件中添加配置：
   spring.autoconfigure.exclude=\

 org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
4. 更换依赖项
```

- 打包

maven插件会将项目打包成2个jar包
一个是fatjar，包会大，其中包括代码，依赖库，内嵌tomcat
一个是传统的jar包

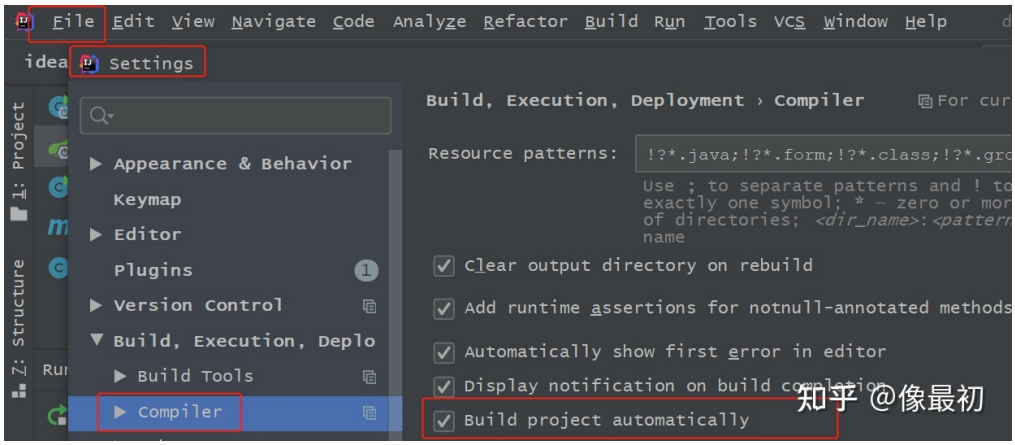
打包操作：maven-->lifecycle-->package-->运行
在target下可以找到打包好的2个jar

- 热部署

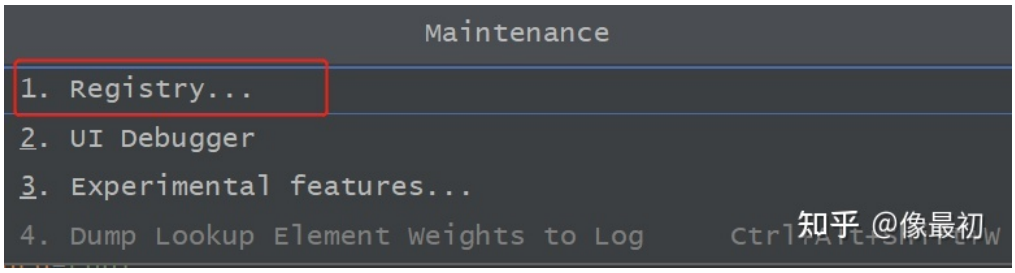
写好一个功能，要启动项目从浏览器进行测试，若在项目运行期间修改了代码，不重新启动项目，修改不生效
若想在项目启动期间修改代码，且想不重新项目，让修改生效，则可以使用热部署实现

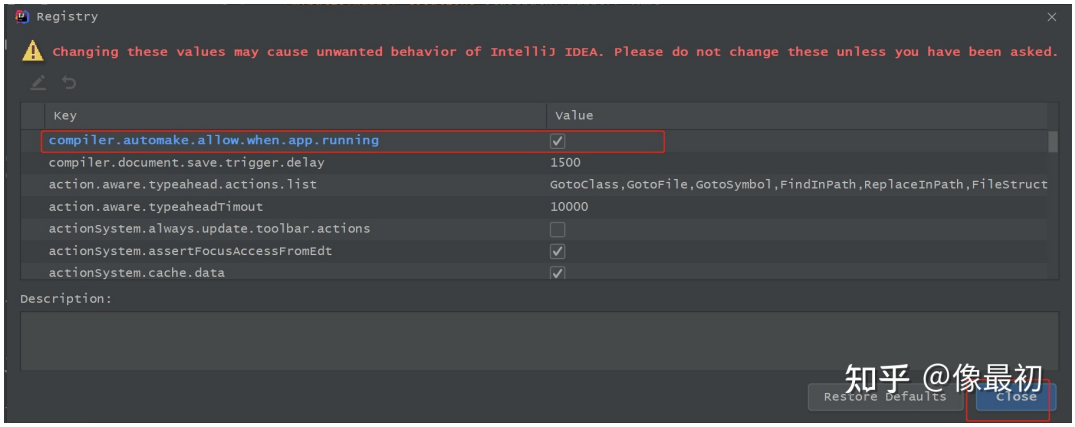
SpringBoot的热部署功能开启：

1. 设置IDEA自动编译



2. 同时按住 Ctrl + Shift + Alt + / 然后进入Registry， 勾选自动编译 -> compiler.automake.allow.when.app.running





3. 在项目pom.xml添加热部署插件jar包

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional>
</dependency>
```

4. Idea热部署开启后，修改后通常会等待一会才会生效，而我们通常都想及时生效，在idea中有个小锤子的图标，点击这里，重新build项目，立即生效
5. 关闭浏览器缓存