**1什么是 Spring Boot？**

SpringBoot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器，开发者能快速上手。

**2为什么要用 Spring Boot？**

Spring Boot 优点非常多，如：

一、独立运行

Spring Boot而且内嵌了各种servlet容器，Tomcat、Jetty等，现在不再需要打成war包部署到容器中，Spring Boot只要打成一个可执行的jar包就能独立运行，所有的依赖包都在一个jar包内。

二、内嵌容器

内嵌有Tomcat 或 Jetty 等servlet容器

三、简化配置

spring-boot-starter-web启动器自动依赖其他组件，简少了maven的配置。

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

过去springmvc的时候或者ssm整合时，使用了大量包

四、自动配置

Spring Boot能根据当前类路径下的类、jar包来自动配置bean，如添加一个spring-boot-starter-web启动器就能拥有web的功能，无需其他配置。

五、无代码生成和XML配置

Spring Boot配置过程中无代码生成，也无需XML配置文件就能完成所有配置工作，这一切都是借助于条件注解完成的，这也是Spring4.x的核心功能之一。(@Conditional根据满足某一个特定条件创建一个特定的Bean。比如说，当某一个jar包在某一个类路径下的时候，自动配置一个或者多个Bean；或者只有某个Bean被创建才会创建另外一个Bean。总的来说，就是根据特定条件来控制Bean的创建行为，这样我们可以利用这个特性进行一些自动的配置。)

用法参考项目：conditional\_demo

策略：

SpringBoot框架中还有两个非常重要的策略：开箱即用和约定优于配置。

（1）开箱即用就是指在maven项目pom中添加需要的依赖jar，然后使用注解来代替繁琐的xml配置，这种策略使开发人员能够摆脱繁琐的配置和管理依赖，更加注重业务逻辑

（2）约定优于配置就是指springboot自己配置目标结构，开发者在结构中添加信息的设计范式，缺点是：降低灵活性，增加bug定位难度；优点是：减少xml配置，将代码编译，测试，打包自动化

**3 Spring Boot 的核心配置文件有哪几个？它们的区别是什么？**

spring cloud官网：

<https://cloud.spring.io/spring-cloud-static/Greenwich.SR1/single/spring-cloud.html#_the_bootstrap_application_context>

Spring Boot 中有以下两种配置文件：

bootstrap (.yml 或者 .properties)

application (.yml 或者 .properties)

bootstrap/ application 的区别：

Spring Cloud 构建于 Spring Boot 之上，在 Spring Boot 中有两种上下文，一种是 bootstrap,，另外一种是 application,，bootstrap 是应用程序的父上下文，也就是说 bootstrap 加载优先于 applicaton。bootstrap 主要用于从额外的资源来加载配置信息，还可以在本地外部配置文件中解密属性。这两个上下文共用一个环境，它是任何Spring应用程序的外部属性的来源。bootstrap 里面的属性会优先加载，它们默认也不能被本地相同配置覆盖。

因此，对比 application 配置文件，bootstrap 配置文件具有以下几个特性：

boostrap 由父 ApplicationContext 加载，比 applicaton 优先加载

boostrap 里面的属性不能被覆盖

bootstrap/ application 的应用场景：

application 配置文件这个容易理解，主要用于 Spring Boot 项目的自动化配置。

bootstrap 配置文件有以下几个应用场景：

使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中添加连接到配置中心的配置属性来加载外部配置中心的配置信息；

一些固定的不能被覆盖的属性

一些加密/解密的场景。



**4Spring Boot 的配置文件有哪几种格式？它们有什么区别？**

.properties 和 .yml，它们的区别主要是书写格式不同。

1).properties

server.port=8081

2).yml

server:

port: 8082

另外，.yml 格式不支持 @PropertySource 注解导入配置。

yml的优先级高于.properties 可以两个配置文件中都配置访问端口号，看哪个配置生效

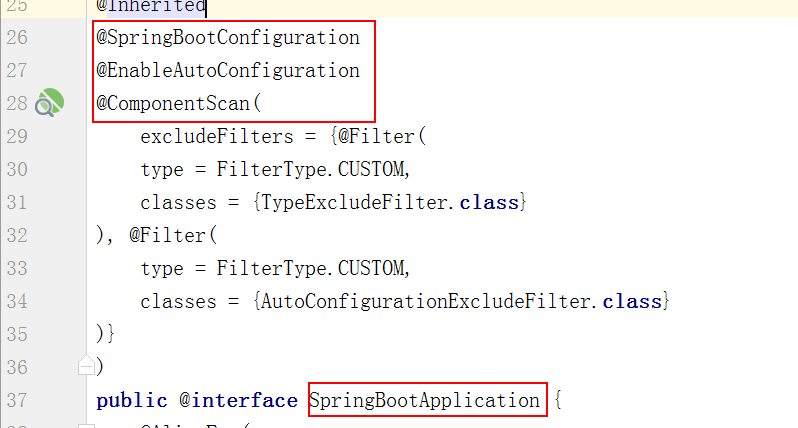
**5 Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的？**

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下 3 个注解：

@SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。

@EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。

@ComponentScan：Spring组件扫描。



**6 开启 Spring Boot 特性有哪几种方式？**

有以下两种方式：

1. 继承spring-boot-starter-parent项目（普通spring-boot项目）

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.2.4.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

2. 导入spring-boot-dependencies项目依赖（参考时springcloud项目搭建的方法）

<dependencyManagement>

<!--spring boot的所有包-->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-dependencies</artifactId>

<version>${spring-boot.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencyManagement>

dependencyManagement 和 dependencies区别

dependencyManagement 当使用父子项目创建工程架构时，父项目可以使用该标签

来规定子项目可以使用哪些包，但是并不是直接引入，而是子项目按需引入，并且引入

时不需要加版本号

dependencies 使用的话，直接父项目把所有包都引入了，不管子项目是否用得到

<scope>import</scope> maven默认也是单继承 加上import 相当于父项目中使用了

多继承，既有springcloud的所有包，又有springboot的所有包，子项目，可以直接使用

Spring Boot依赖注意点：

1. 属性覆盖只对继承有效

Spring Boot依赖包里面的组件的版本都是和当前Spring Boot绑定的，如果要修改里面组件的版本，只需要添加如下属性覆盖即可，但这种方式只对继承有效，导入的方式无效。

<properties>

<slf4j.version>1.7.25<slf4j.version>

</properties>



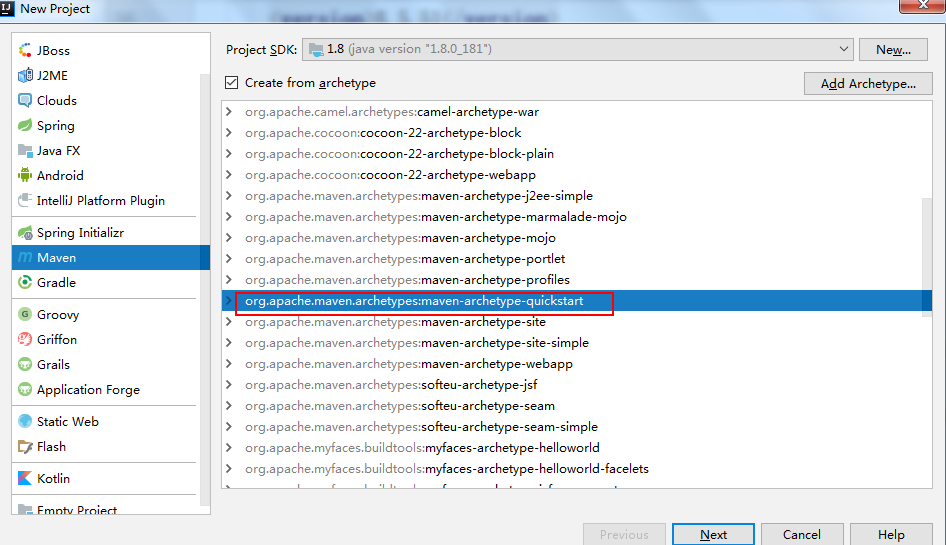
如果导入的方式要实现版本的升级，达到上面的效果，这样也可以做到，把要升级的组件依赖放到Spring Boot之前。需要注意，要修改Spring Boot的依赖组件版本可能会造成不兼容的问题。

**Spring Boot 需要独立的容器运行吗？**

可以不需要，内置了 Tomcat/Jetty 等容器。

模拟内置tomcat功能，参考项目：embed\_tomcat

1，创建普通的java项目



2，引入jar包

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-core</artifactId>

<version>8.5.51</version>

</dependency>

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-el</artifactId>

<version>8.5.51</version>

</dependency>

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>

<version>8.5.51</version>

</dependency>

3，编写启动类

package com.aaa;

import org.apache.catalina.Context;

import org.apache.catalina.WebResourceRoot;

import org.apache.catalina.startup.Tomcat;

import org.apache.catalina.webresources.DirResourceSet;

import org.apache.catalina.webresources.StandardRoot;

import java.io.File;

/\*\*

\* fileName:SpringApplication

\* description:

\* author:zz

\* createTime:2020/2/26 18:17

\* version:1.0.0

\*/

public class SpringApplication {

/\*\*

\* 模拟内置tomcat

\*/

public static void run(){

//创建tomcat实例

Tomcat tom = new Tomcat();

//设置端口

tom.setPort(8888);

try{

//获取项目编译后的class 路径

String path = SpringApplication.class.getResource("/").getPath();

//获取webapp 文件 D:\idea\_qy107\embed\_tomcat\src\main\webapp

String filePath = new File("src/main/webapp").getAbsolutePath();

//然后将webapp下的项目添加至tomcat的context容器（context对应一个运行的项目）

Context context =tom.addWebapp("/my\_web\_app",filePath);

//参数1：一般是项目名 对应请求url中的项目名

//webResourceRoot 用于加载 项目的class文件

WebResourceRoot webResource = new StandardRoot(context);

webResource.addPreResources(new DirResourceSet(webResource,"/WEB-INF/classes",path,"/"));

tom.start();

}catch (Exception e) {

e.printStackTrace();

}

//阻塞 ，等待前端连接

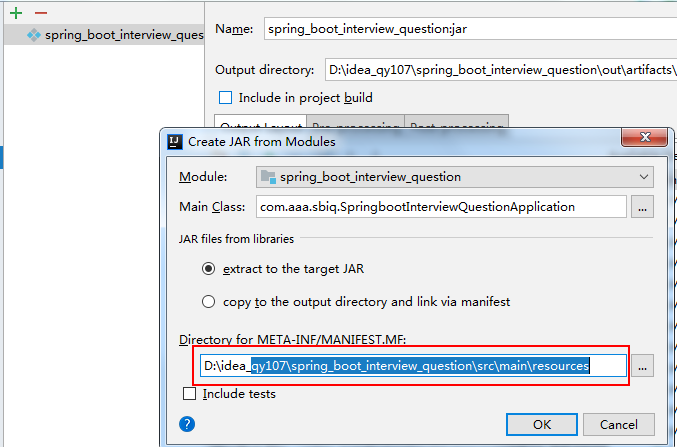
tom.getServer().await();

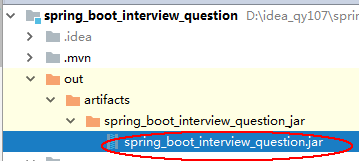
}

}

**7 运行 Spring Boot 有哪几种方式？**

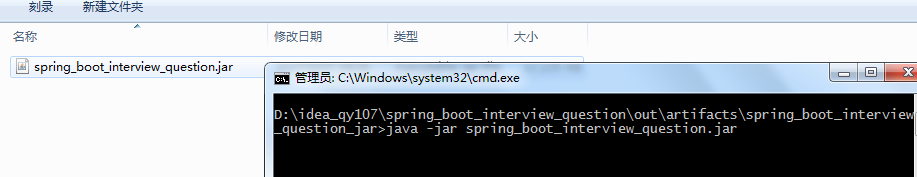
1）打包用命令或者放到容器中运行





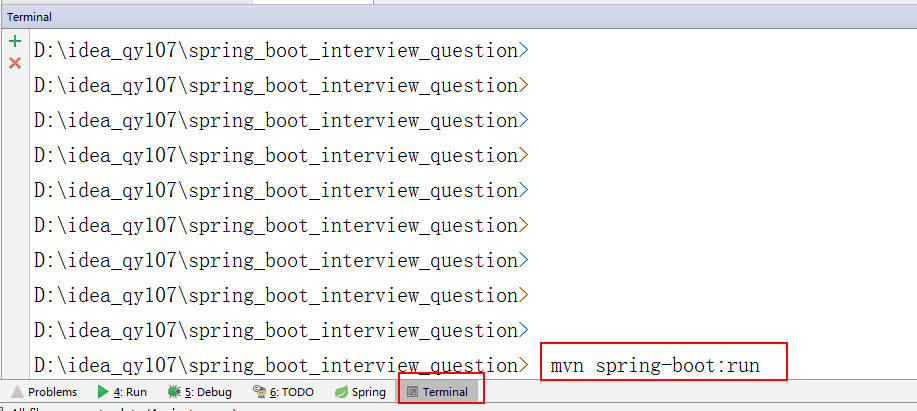
在jar所在位置，shift+鼠标右键 打开命令行，输入命令：

java -jar spring\_boot\_interview\_question.jar [启动类]



2）用 Maven/Gradle 插件运行

进入项目目录，执行下面命令



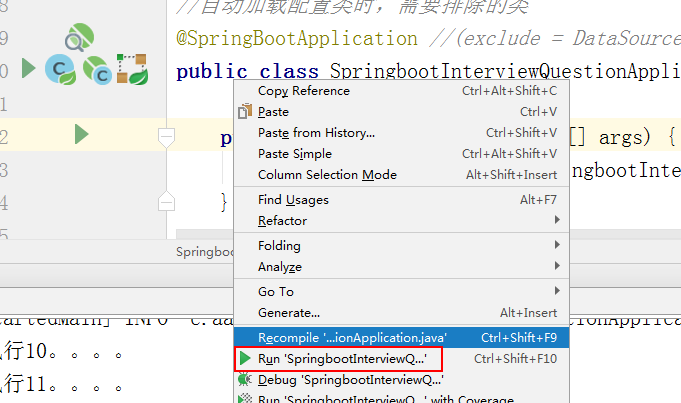
mvn spring-boot:run

启动测试：

<http://localhost:8082/showEntity>

ctrl+c 终止运行，如果是cmd命令窗口，直接关闭就可以了

3）直接执行 main 方法运行



**9 Spring Boot 自动配置原理是什么？**

为自动配置做铺垫：

@import 三种用法：

直接将pojo类注入到Spring IOC容器

1， 创建java类：

public class PojoClass {

//空构造

public PojoClass() {

System.out.println("PojoClass交给了spirngioc管理了。。。 ");

}

}

2，在@Configuration注解的类上加入@Import(value = {PojoClass.class})

3，测试

把它注入到任意一个已经交给spring管理的类中

@Autowired

private PojoClass pojoClass;

public Emp getById(Integer empNo){

System.out.println("springioc容器中是否存在pojoClass对象："+pojoClass);

return empService.getById(empNo);

}

通过ImportBeanDefinitionRegistrar将类注入到Spring IOC容器。

1， 创建java类：

public class BeanDefinitionRegisterClass {

public BeanDefinitionRegisterClass() {

System.out.println("通过ImportBeanDefinitionRegistrar接口把该类交给springioc容器进行管理。。。");

}

}

2，创建ImportBeanDefinitionRegistrar实现类

public class MyImportBeanDefinitionRegister implements ImportBeanDefinitionRegistrar {

@Override

public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {

//交给springioc容器管理的是BeanDefinition bean定义

RootBeanDefinition rootBeanDefinition =new RootBeanDefinition(BeanDefinitionRegisterClass.class);

registry.registerBeanDefinition("beanDefinitionRegisterClass",rootBeanDefinition);

}

}

3，在@Configuration注解的类上加入@Import(value ={MyImportBeanDefinitionRegister.class})

4，测试

@Autowired

private BeanDefinitionRegisterClass beanDefinitionRegisterClass;

public Emp getById(Integer empNo){

System.out.println("springioc容器中是否存在pojoClass对象："+pojoClass);

System.out.println("springioc容器中是否存在beanDefinitionRegisterClass对象："+beanDefinitionRegisterClass);

return empService.getById(empNo);

}

通过ImportSelector方式注入Bean

1，创建多个java类

public class ImportSelectorClassA {

public ImportSelectorClassA() {

System.out.println("通过ImportSelector接口把该类ImportSelectorClassA交给springioc容器进行管理。。。");

}

}

2，创建ImportSelector的实现类

public class MyImportSelector implements ImportSelector {

@Override

public String[] selectImports(AnnotationMetadata annotationMetadata) {

return new String[]{"com.aaa.sbac.ac2.entity.ImportSelectorClassA",

"com.aaa.sbac.ac2.entity.ImportSelectorClassB"};

}

}

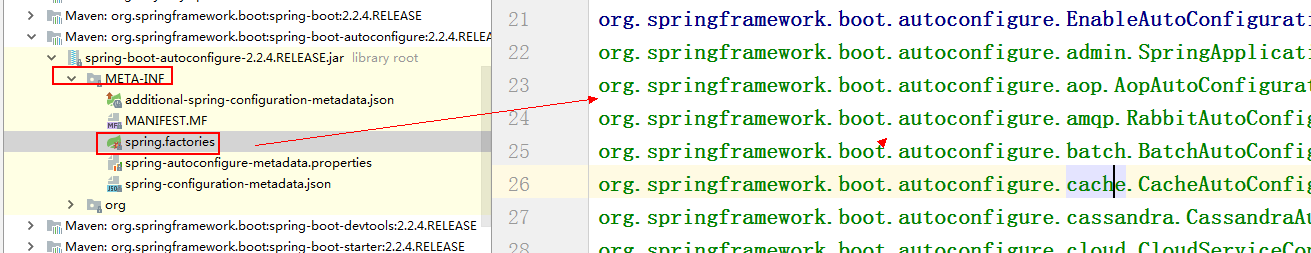
3，在@Configuration注解的类上加入@Import(value ={MyImportSelector.class})

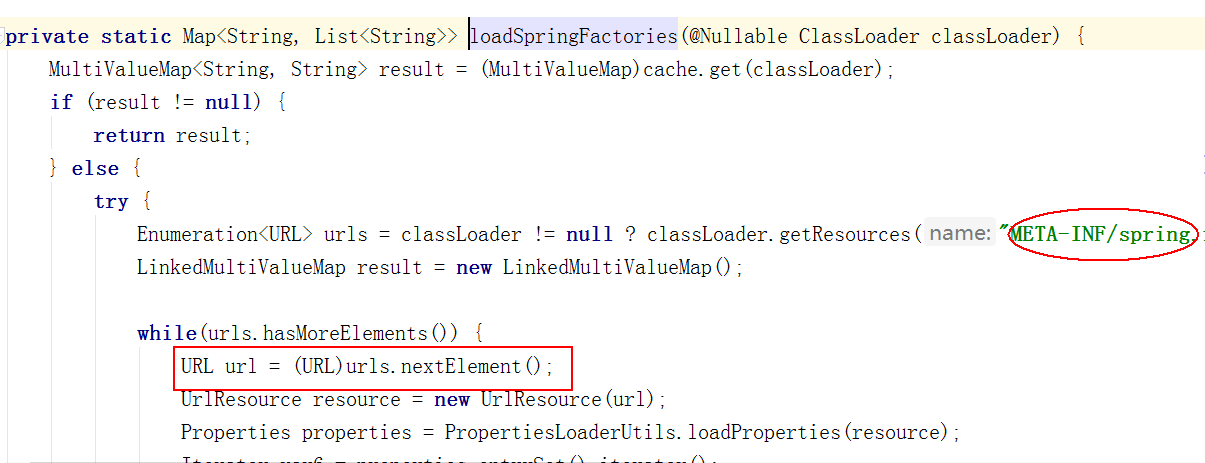
4，测试

测试同上

**参考项目：spring\_boot\_auto\_configuration 第3个例子：com.aaa.sbac.ac3包的例子：**

Spring Boot的自动配置注解是@EnableAutoConfiguration， 从上面的@Import的类可以找到下面自动加载自动配置的映射。





URL url = urls.nextElement();

这个方法会加载类路径及所有jar包下META-INF/spring.factories配置中映射的自动配置的类。

public static final String FACTORIES\_RESOURCE\_LOCATION = "META-INF/spring.factories";

查看Spring Boot自带的自动配置的包：org.springframework.boot:spring-boot-autoconfigure-2.2.4.RELEASE.jar ，打开其中的META-INF/spring.factories文件会找到自动配置的映射。

再来看看数据源自动配置的实现注解

@Configuration,@ConditionalOnClass就是自动配置的核心，首先它得是一个配置文件，其次根据类路径下是否有这个类去自动配置。

**具体答案：**

@SpringBootApplication是一个复合注解或派生注解，在@SpringBootApplication中有一个注解@EnableAutoConfiguration，翻译成人话就是**开启自动配置，**而这个注解也是一个派生注解，其中的关键功能由@Import提供，其导入的AutoConfigurationImportSelector的selectImports()方法通过SpringFactoriesLoader.loadFactoryNames()扫描所有具有META-INF/spring.factories的jar包。spring-boot-autoconfigure-x.x.x.x.jar里就有一个这样的spring.factories文件。这个spring.factories文件也是一组一组的key=value的形式，其中一个key是EnableAutoConfiguration类的全类名，而它的value是一个xxxxAutoConfiguration的类名的列表，这些类名以逗号分隔，这个@EnableAutoConfiguration注解通过@SpringBootApplication被间接的标记在了Spring Boot的启动类上。在SpringApplication.run(...)的内部就会执行selectImports()方法，找到所有JavaConfig自动配置类的全限定名对应的class，然后将所有自动配置类加载到Spring容器中。

**10 Spring Boot 的目录结构是怎样的？**

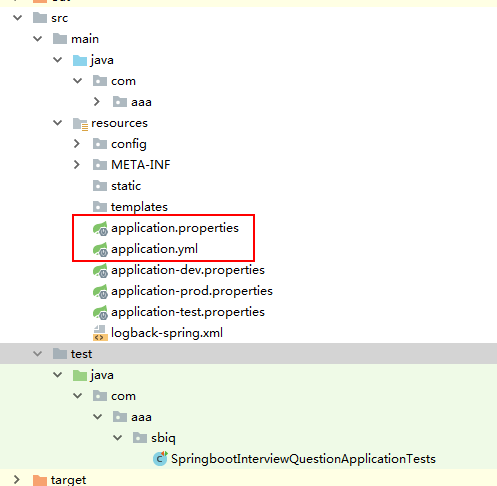
Spring Boot 与传统项目最大的区别是，传统项目都是打成 WAR（idea或者eclipse工具会帮助我们把web项目打成war包） 包部署到服务器（tomcat 下的webapps）上面，需要额外的 Servlet 容器， 而 Spring Boot 则可以直接打成 jar包，并内置集成了(内置tomcat) Servlet 容器，通过命令 java -jar xx.jar 则可以直接运行，不需要独立的 Servlet 容器。

目录：

src/main/java：主程序入口 Application，可以通过直接运行该类来 启动 Spring Boot应用

src/main/resources：配置目录，该目录用来存放应用的一些配置信息，比如应用名、服务端口、数据库配置等。由于我们应用了Web模块，因此产生了 static目录与templates目录，前者用于存放静态资源，如图片、CSS、JavaScript等；后者用于存放Web页面的模板文件。

src/test：单元测试目录，生成的 ApplicationTests 通过 JUnit4实现，可以直接用运行 Spring Boot应用的测试。



application.properties/application.yml 用于存放程序的各种依赖模块的配置信息，比如 服务端口，数据库连

(1)代码层的结构

　　根目录：com.springboot

　　　　1.工程启动类(ApplicationServer.java)置于com.springboot.build包下

　　　　 2.实体类(entity，domain)置于com.springboot.domain

　　　　3.数据访问层(Dao)置于com.springboot.repository

　　　　4.数据服务层(Service)置于com,springboot.service,数据服务的实现接口(serviceImpl) 至于com.springboot.service.impl

　　　　5.前端控制器(Controller)置于com.springboot.controller

　　　　6.工具类(utils)置于com.springboot.utils

　　　　7.常量接口类(constant)置于com.springboot.constant

　　　　8.配置信息类(config)置于com.springboot.config

　　　　9.数据传输类(vo)置于com.springboot.vo

pojo plain ordinary java object 简单 普通java object

在实际项目，根据不同用途，把普通java 类变成各种不一样称呼

po persistent objcet dao层使用 持久化对象 和数据中表的列的对应（hibernate,mybatis,jpa....）

bo /do business object / domain object 业务对象 业务层封装方法时使用的对象，相当于一个多个 po 组成

dto data transfer object 数据传输对象 把业务层对象传输给表现 层或者把表现层vo对象传输业务层 (bo 有30字段 而显示层只要10个字段 dto定义10个字段的对象)

vo view object 显示层对象（web）与页面form表单 一个页面写的数据会向数据多个表存储

简单项目 entity bean

复杂项目向，大项目微服务项目中，不同分层里面有大量不同项目，项目与项目交互时，会使用到上面 po bo/do dto ob

（2）资源文件的结构

　　根目录:src/main/resources

　　　　1.配置文件(.properties/.json等)置于config文件夹下

　　　　2.国际化(i18n))置于i18n文件夹下

　　　　3.spring.xml置于META-INF/spring文件夹下

　　　　4.页面以及js/css/image等置于static文件夹下的各自文件下接配置等。。。

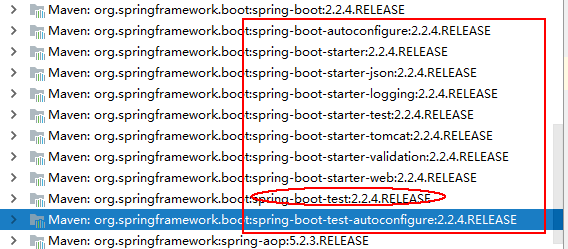
**11 你如何理解 Spring Boot 中的 Starters？**

Starters是什么：

Starters可以理解为启动器，它包含了一系列可以集成到应用里面的依赖包，你可以一站式集成Spring及其他技术，而不需要到处找示例代码和依赖包。如你想使用Spring JPA访问数据库，只要加入spring-boot-starter-data-jpa启动器依赖就能使用了。Starters包含了许多项目中需要用到的依赖，它们能快速持续的运行，都是一系列得到支持的管理传递性依赖。

Starters命名：

Spring Boot官方的启动器都是以spring-boot-starter-命名的，代表了一个特定的应用类型。第三方的启动器不能以spring-boot开头命名，它们都被Spring Boot官方保留。一般一个第三方的应该这样命名，像mybatis的mybatis-spring-boot-starter。



Starters分类：

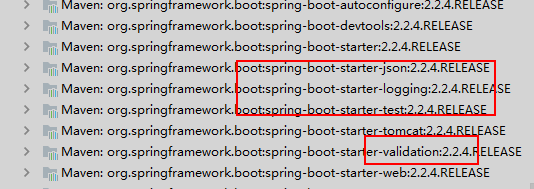
1. Spring Boot应用类启动器



2. Spring Boot生产启动器

clipboard.png

3. Spring Boot技术类启动器



4. 其他第三方启动器

**12 如何在 Spring Boot 启动的时候运行一些特定的代码？**

如果你想在Spring Boot启动的时候运行一些特定的代码，你可以实现接口ApplicationRunner或者CommandLineRunner，这两个接口实现方式一样，它们都只提供了一个run方法。

CommandLineRunner：启动获取命令行参数

package com.aaa.sbiq.startup\_run;

import org.springframework.boot.CommandLineRunner;

import org.springframework.core.annotation.Order;

import org.springframework.stereotype.Component;

/\*\*

\* fileName:MyCommandLineRunner

\* description:

\* author:zz

\* createTime:2020/2/26 17:50

\* version:1.0.0

\*/

@Component

@Order(value = 10)

public class MyCommandLineRunner implements CommandLineRunner {

@Override

public void run(String... args) throws Exception {

System.out.println("CommandLineRunner：启动开始执行10。。。。");

}

}

ApplicationRunner：启动获取应用启动的时候参数

使用方式：

package com.aaa.sbiq.startup\_run;

import org.springframework.boot.ApplicationArguments;

import org.springframework.boot.ApplicationRunner;

import org.springframework.core.annotation.Order;

import org.springframework.stereotype.Component;

/\*\*

\* fileName:MyApplicationRunner

\* description:

\* author:zz

\* createTime:2020/2/26 17:48

\* version:1.0.0

\*/

@Component

@Order(value = 11)

public class MyApplicationRunnerA implements ApplicationRunner {

@Override

public void run(ApplicationArguments args) throws Exception {

System.out.println("ApplicationRunner：启动开始执行11。。。。");

}

}

或者这样

package com.aaa.sbiq.startup\_run;

import org.apache.catalina.servlets.DefaultServlet;

import org.springframework.boot.ApplicationArguments;

import org.springframework.boot.ApplicationRunner;

import org.springframework.core.Ordered;

import org.springframework.stereotype.Component;

/\*\*

\* fileName:MyApplicationRunnerB

\* description:

\* author:zz

\* createTime:2020/2/26 17:55

\* version:1.0.0

\*/

public class MyApplicationRunnerB implements ApplicationRunner,Ordered{

@Override

public void run(ApplicationArguments args) throws Exception {

System.out.println("ApplicationRunnerB：启动开始执行12。。。。");

}

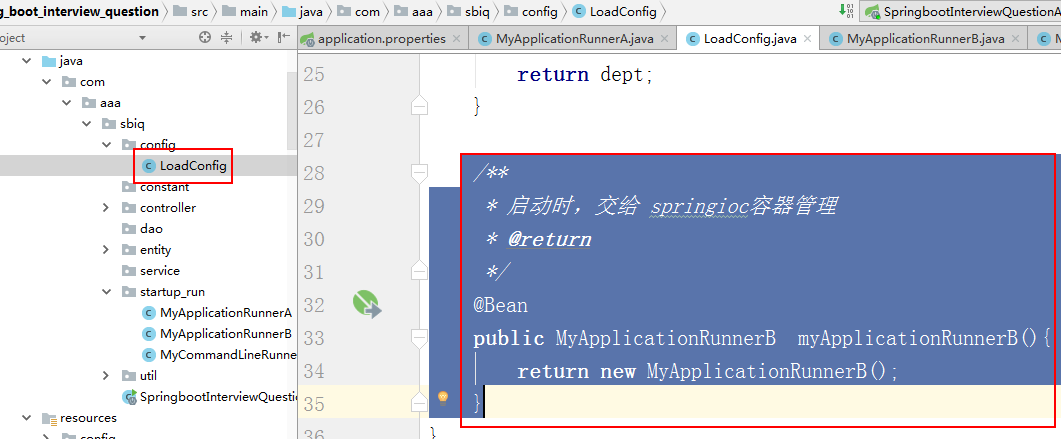
@Override

public int getOrder() {

return 12;

}

}



/\*\*

\* 启动时，交给 springioc容器管理

\* @return

\*/

@Bean

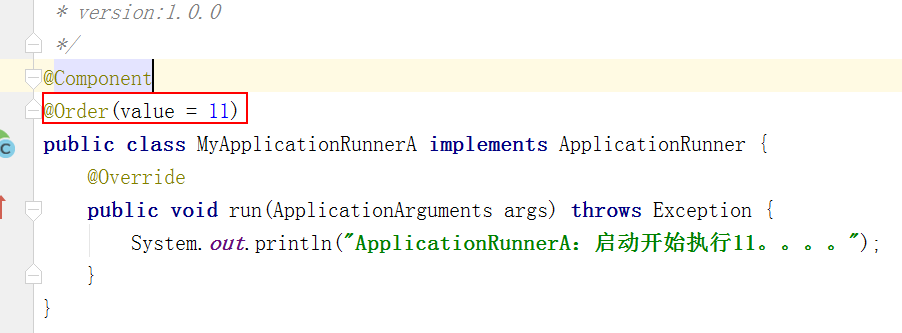
public MyApplicationRunnerB myApplicationRunnerB(){

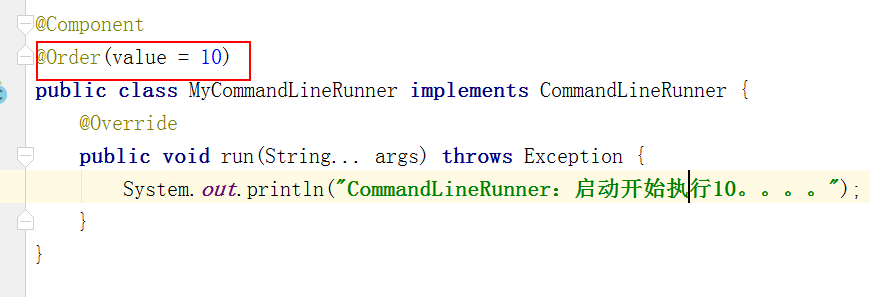
return new MyApplicationRunnerB();

}

启动顺序：

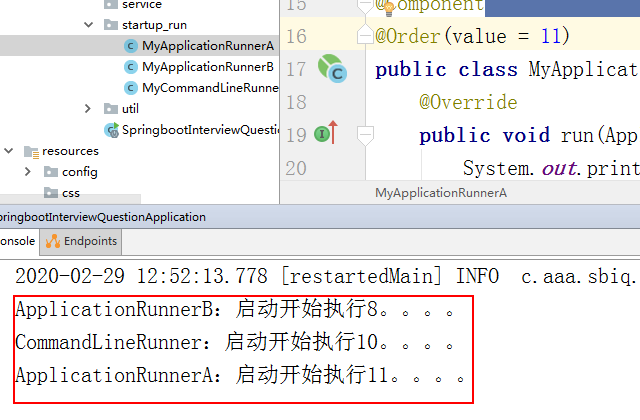
如果启动的时候有多个ApplicationRunner和CommandLineRunner，想控制它们的启动顺序，可以实现 org.springframework.core.Ordered接口或者使用 org.springframework.core.annotation.Order注解。







order值越小，优先级越高



**13 Spring Boot 有哪几种读取配置的方式？**

**读取application文件：**

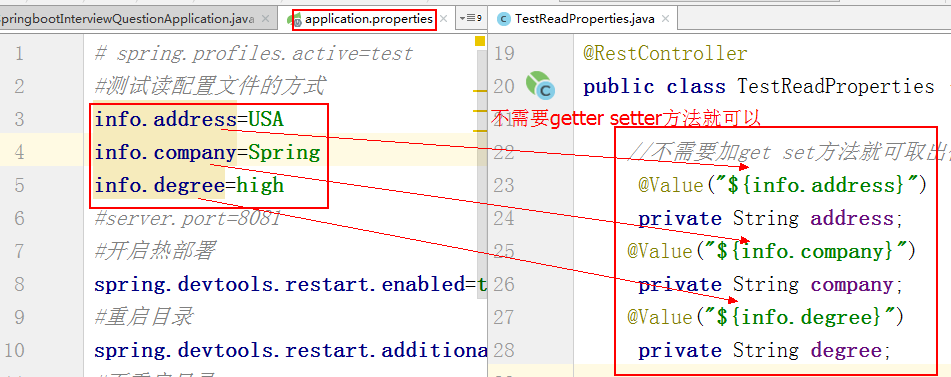
在application.yml或者properties文件中添加：

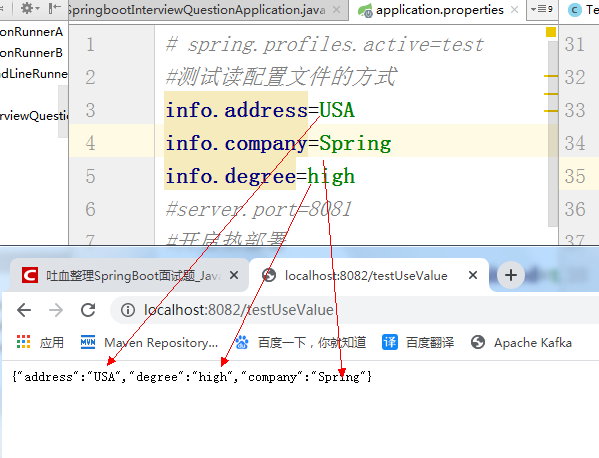
info.address=USA

info.company=Spring

info.degree=high

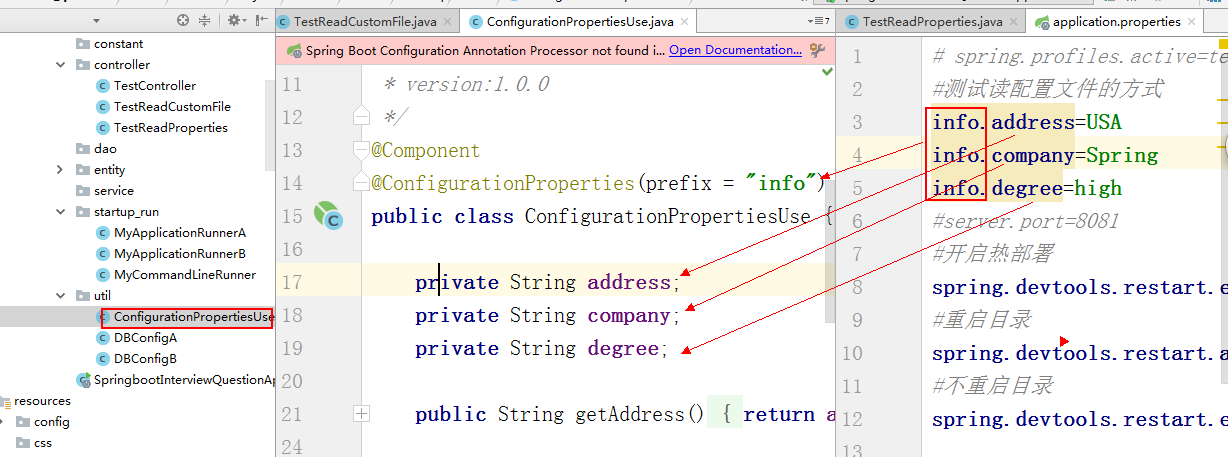
一、@Value注解读取方式：

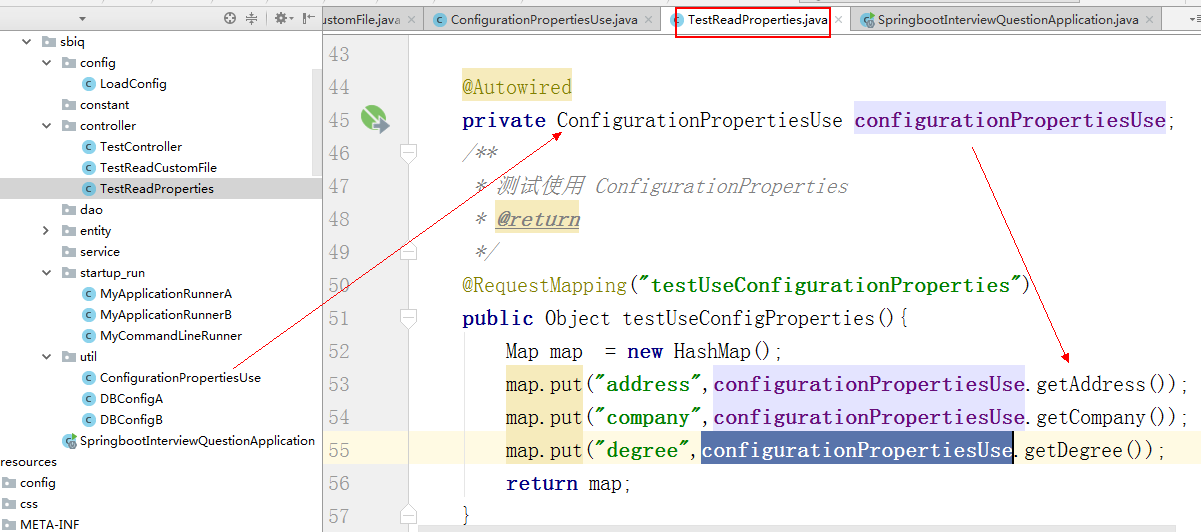




<http://localhost:8082/testUseValue>

二、@ConfigurationProperties注解读取方式：





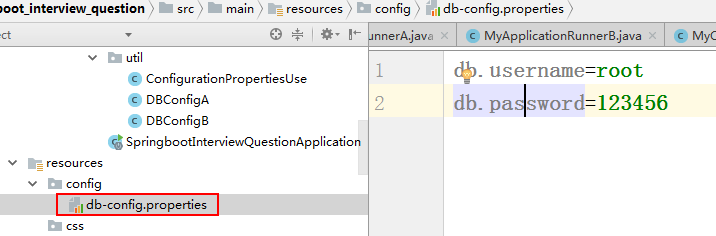
<http://localhost:8082/testUseConfigurationProperties>

**读取指定（自定义的）文件：**

资源目录下建立config/db-config.properties:

db.username=root

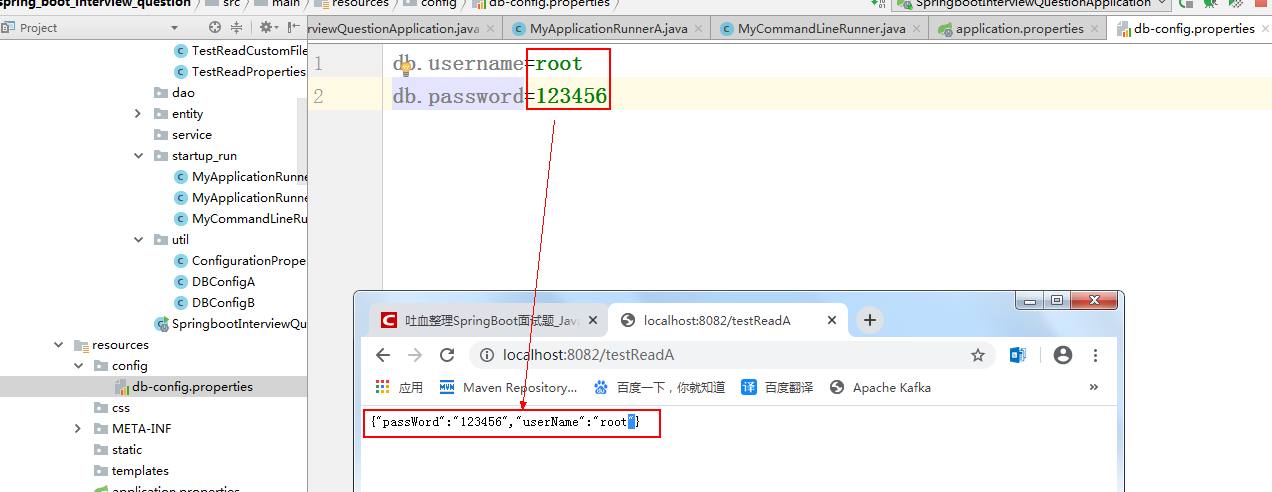
db.password=123456



一、@PropertySource+@Value注解读取方式：

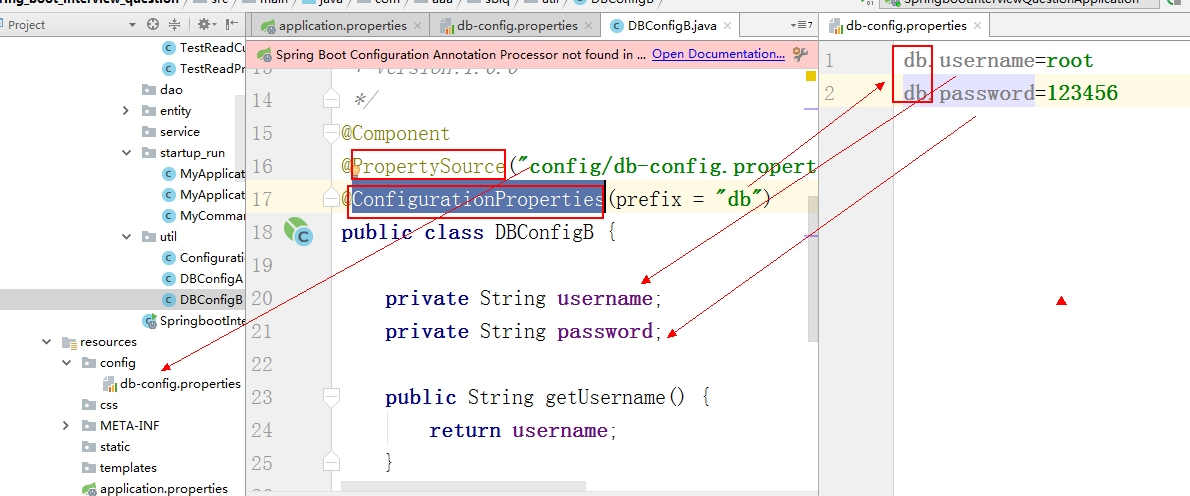
注意：@PropertySource不支持yml文件读取。

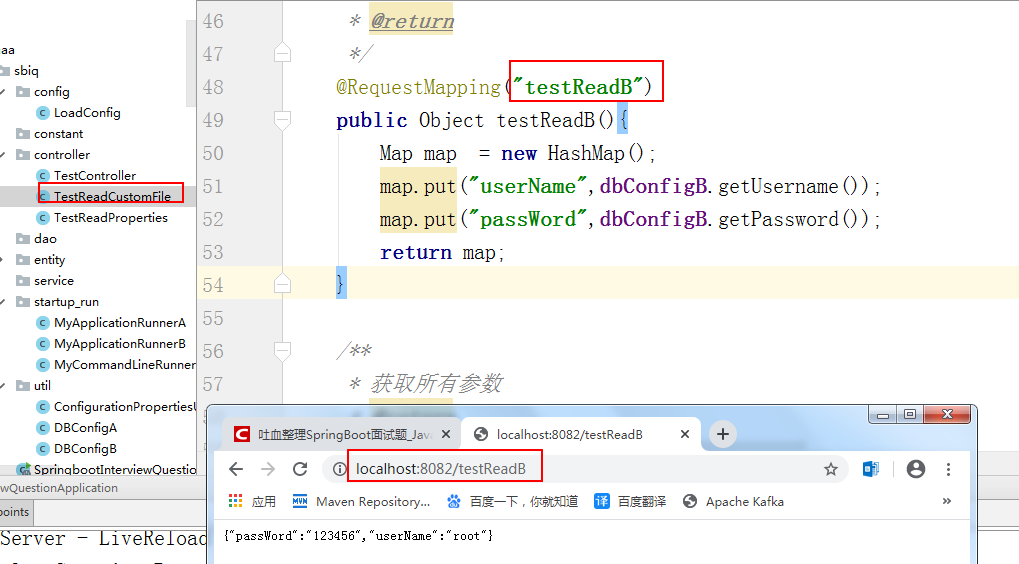




<http://localhost:8082/testReadA>

二、@PropertySource+@ConfigurationProperties注解读取方式：（参考前面讲redis时的代码）



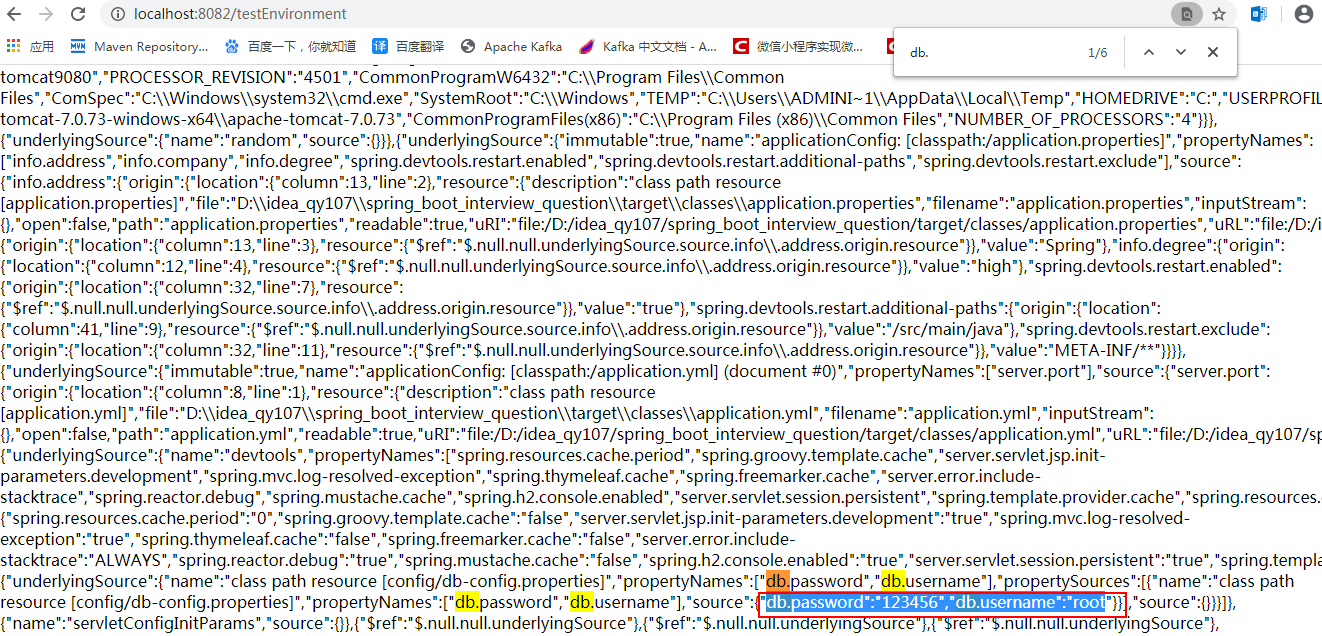


<http://localhost:8082/testReadB>

三、Environment读取方式：

以上所有加载出来的配置都可以通过Environment注入获取到：





<http://localhost:8082/testEnvironment>

总结

从以上示例来看，Spring Boot可以通过@PropertySource,@Value,@Environment,@ConfigurationProperties来绑定变量。

**Spring Boot 支持哪些日志框架？推荐和默认的日志框架是哪个？**

Spring Boot支持Java Util Logging,Log4j2,Lockback作为日志框架，如果你使用starters启动器，Spring Boot将使用Logback作为默认日志框架。无论使用哪种日志框架，Spring Boot都支持配置将日志输出到控制台或者文件中。



spring-boot-starter启动器包含spring-boot-starter-logging启动器并集成了slf4j日志抽象及Logback日志框架。

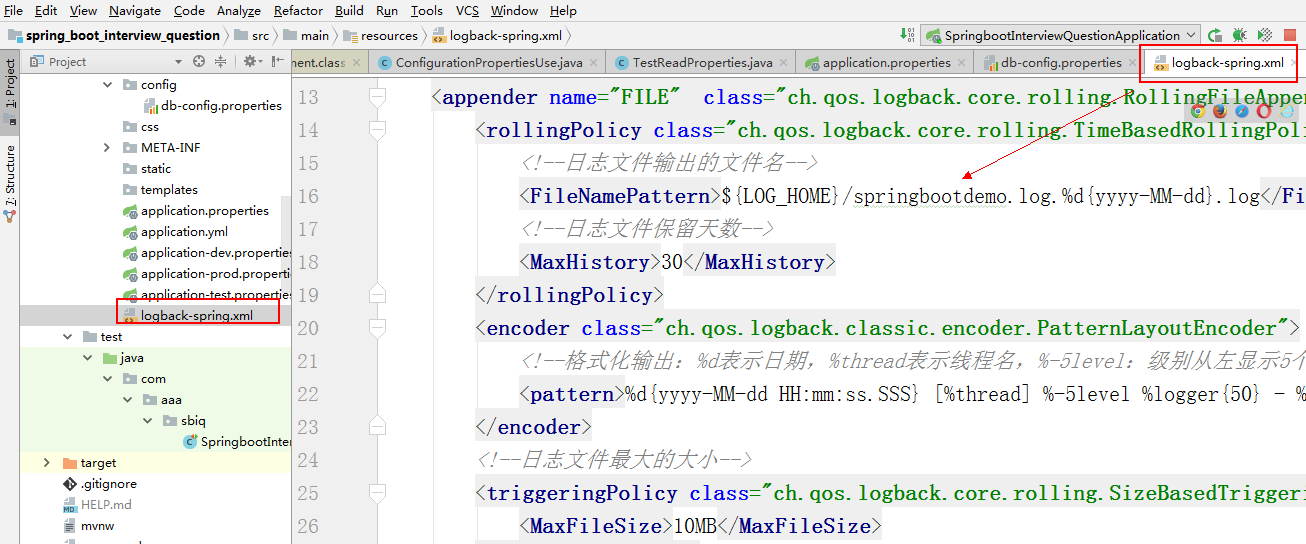
spring-boot-starter-parent ----父包---> spring-boot-dependencies ----父包--->spring-boot-dependencies -----查找----> spring-boot-starter-logging ---点击进去----> 能看到日志包：logback-classic，log4j-to-slf4j.....

属性配置日志：

Spring Boot支持属性配置日志参数，这个不是很灵活，不细讲。

根据不同的日志框架，默认加载的日志配置文件的文件名，放在资源根目录下，其他的目录及文件名不能被加载。既然默认自带了Logback框架，Logback也是最优秀的日志框架，往资源目录下创建一个logback-spring.xml即可。

强烈推荐使用logback-spring.xml作为文件名，因为logback.xml加载太早。



日志初始化在ApplicationContext创建之前，所以@PropertySources加载的配置是读取不到的，系统环境变量、Spring Environment及application，bootstrap配置文件中的信息可以读取到。

读取系统环境属性：

读取当前应用Environment中的属性：

Spring Boot也支持通过springProfile来加载不同profiles下的配置：

**SpringBoot 实现热部署有哪几种方式？**

在Spring Boot实现代码热部署是一件很简单的事情，代码的修改可以自动部署并重新热启动项目。

一、引用devtools依赖：

<!--devtools热部署-->

<dependency>

<groupId>org.springframework.boot</groupId>

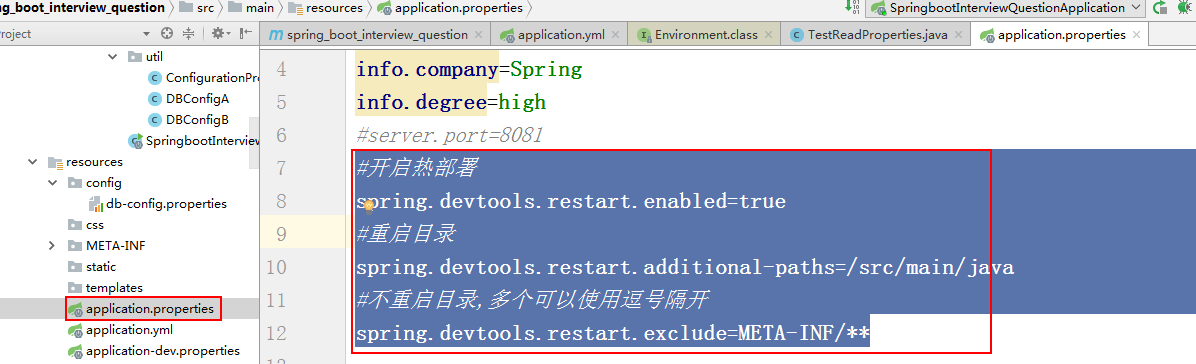
<artifactId>spring-boot-devtools</artifactId>

<optional>true</optional>

</dependency>

这样，当修改一个java类时就会热更新。

二、自定义配置热部署：



#开启热部署

spring.devtools.restart.enabled=true

#重启目录

spring.devtools.restart.additional-paths=/src/main/java

#不重启目录,多个可以使用逗号隔开

spring.devtools.restart.exclude=META-INF/\*\*

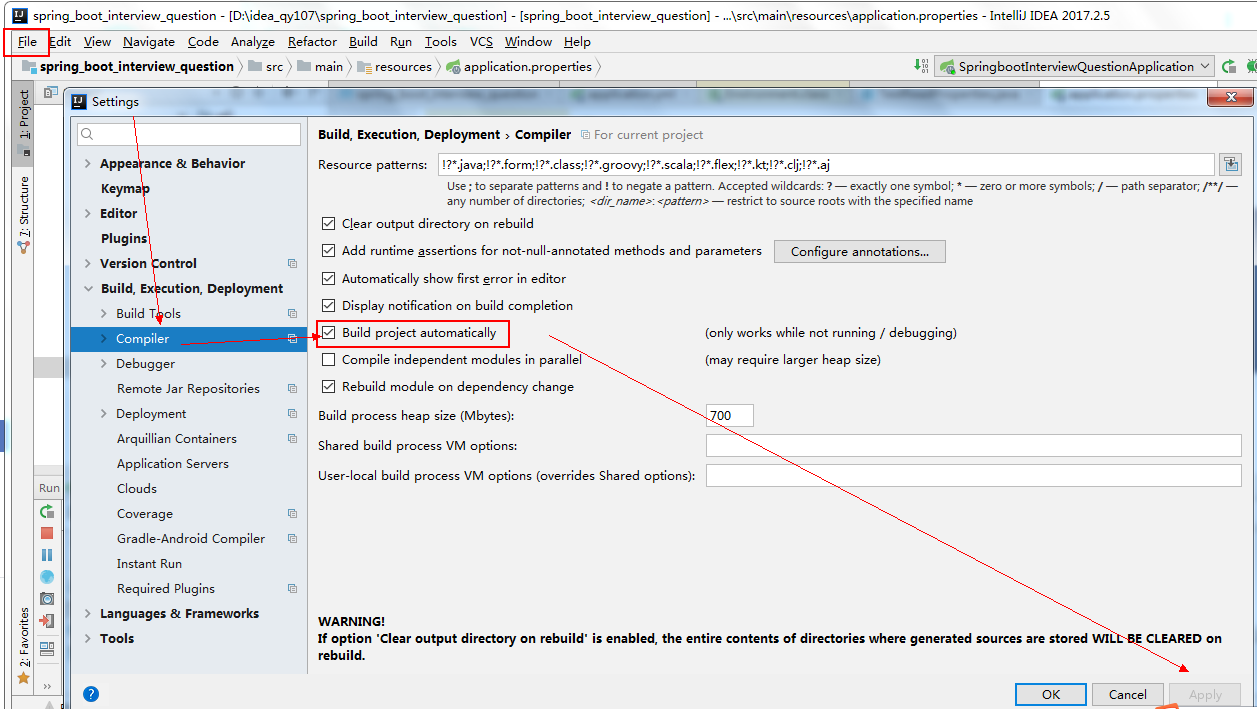
以下配置用于自定义配置热部署，可以不设置。

三、Intellij Idea修改：

如果是idea，需要改以下两个地方

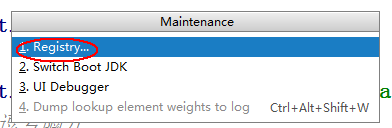
1、勾上自动编译或者手动重新编译

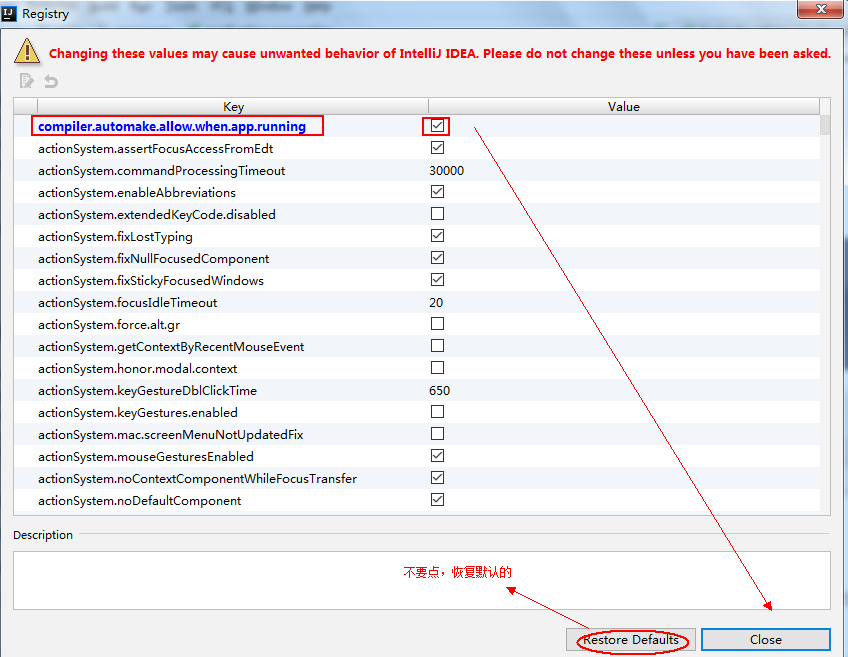
File > Settings > Compiler>Build Project automatically

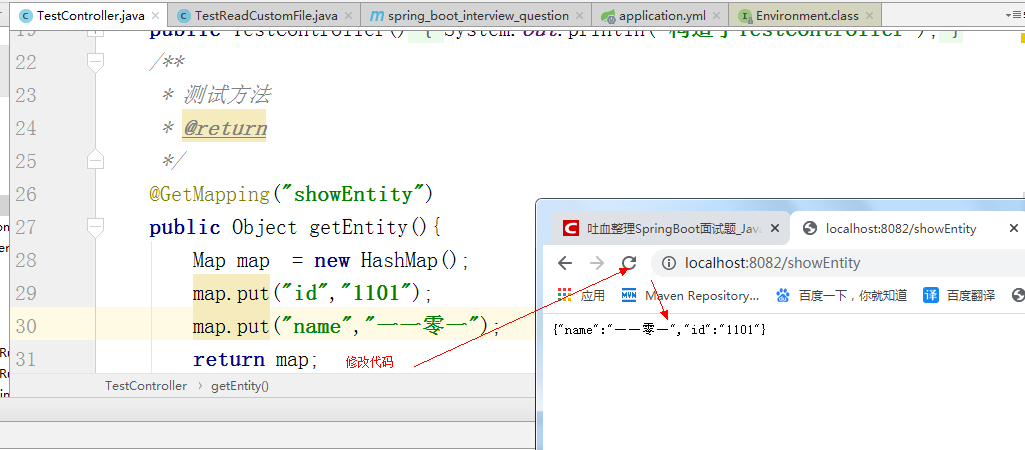


2、注册

ctrl + shift + alt + / > Registry > 勾选Compiler autoMake allow when app running







注意事项：

1、生产环境devtools将被禁用，如java -jar方式或者自定义的类加载器等都会识别为生产环境。

2、打包应用默认不会包含devtools，除非你禁用SpringBoot Maven插件的excludeDevtools属性。

3、Thymeleaf无需配置 spring.thymeleaf.cache:false，devtools默认会自动设置，参考完整属性。

4、devtools会在windows资源管理器占用java进程，在开发工具里面杀不掉，只能手动kill掉，不然重启会选成端口重复绑定报错。

热部署自带缓存，和redis缓存有冲突

**14 Spring Boot 如何定义多套不同环境配置？**

首先我们要了解一个名词：Profile

简单来说，Profile就是Spring Boot可以对不同环境或者指令来读取不同的配置文件。

假如有开发、测试、生产三个不同的环境，需要定义三个不同环境下的配置。

基于properties文件类型

你可以另外建立3个环境下的配置文件：

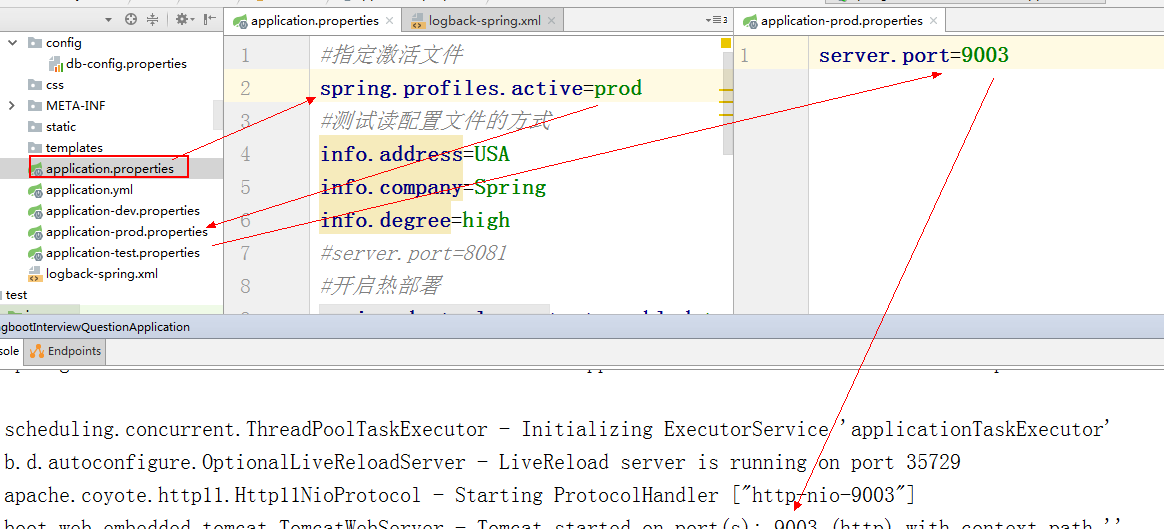
applcation.properties

application-dev.properties

application-test.properties

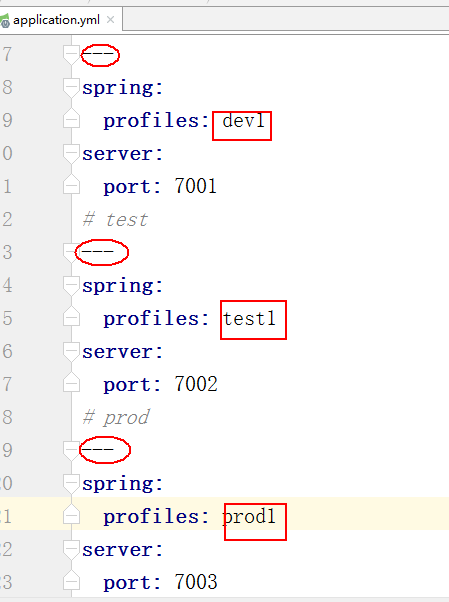
application-prod.properties

然后在applcation.properties文件中指定当前的环境spring.profiles.active=test，这时候读取的就是application-test.properties文件。

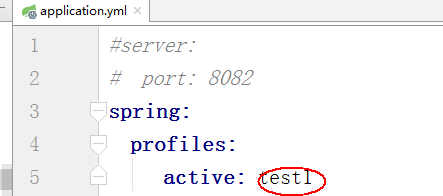


基于yml文件类型

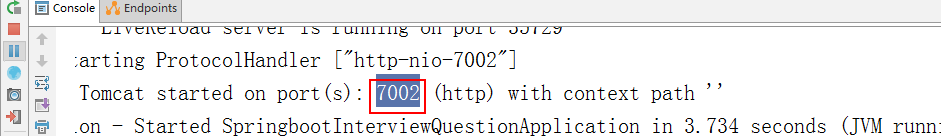
只需要一个applcation.yml文件就能搞定，推荐此方式。



配置指定使用上面截图中哪个配置

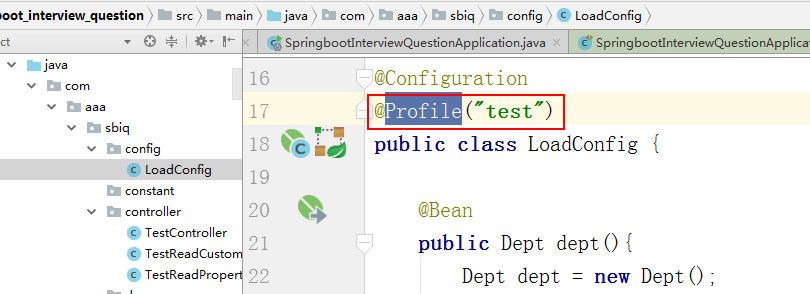


测试，重启项目：



基于Java代码

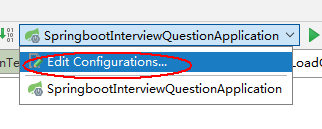
在JAVA配置代码中也可以加不同Profile下定义不同的配置文件，@Profile注解只能组合使用@Configuration和@Component注解。

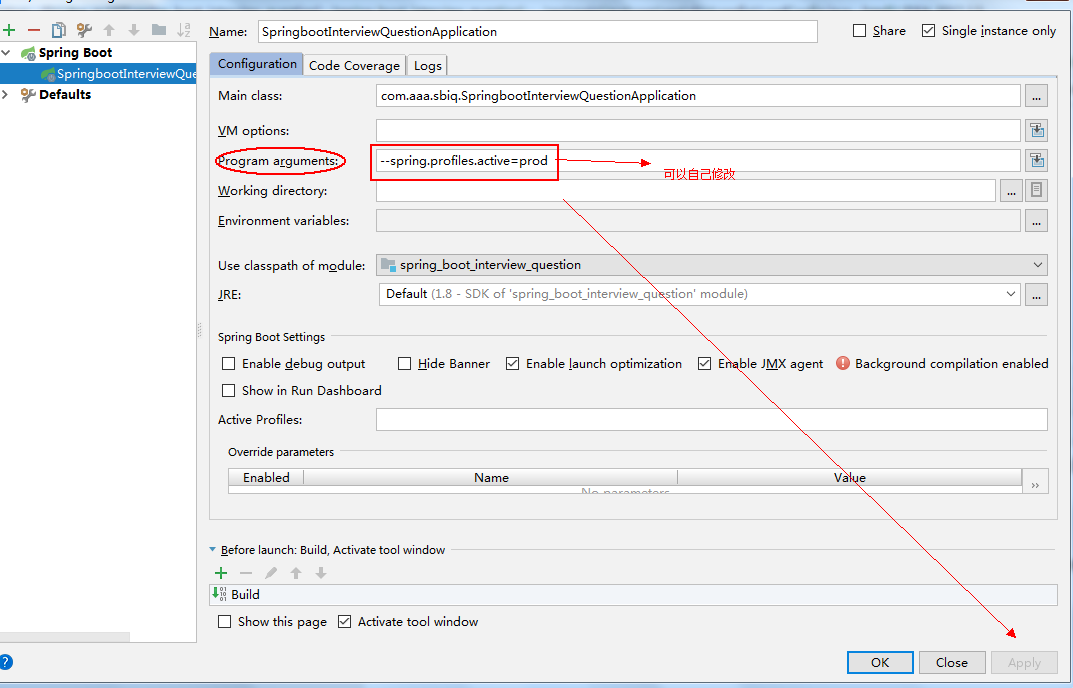


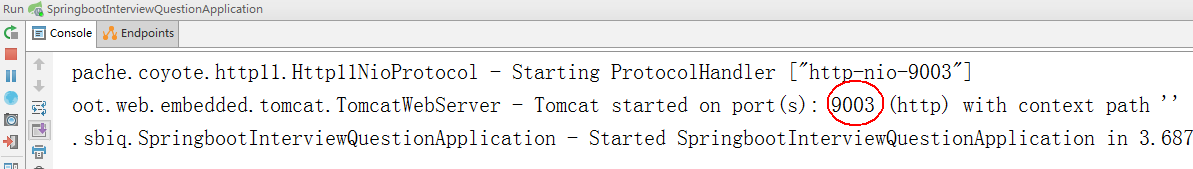
启动类测试失败，原因不详

main方法启动方式：

--spring.profiles.active=test





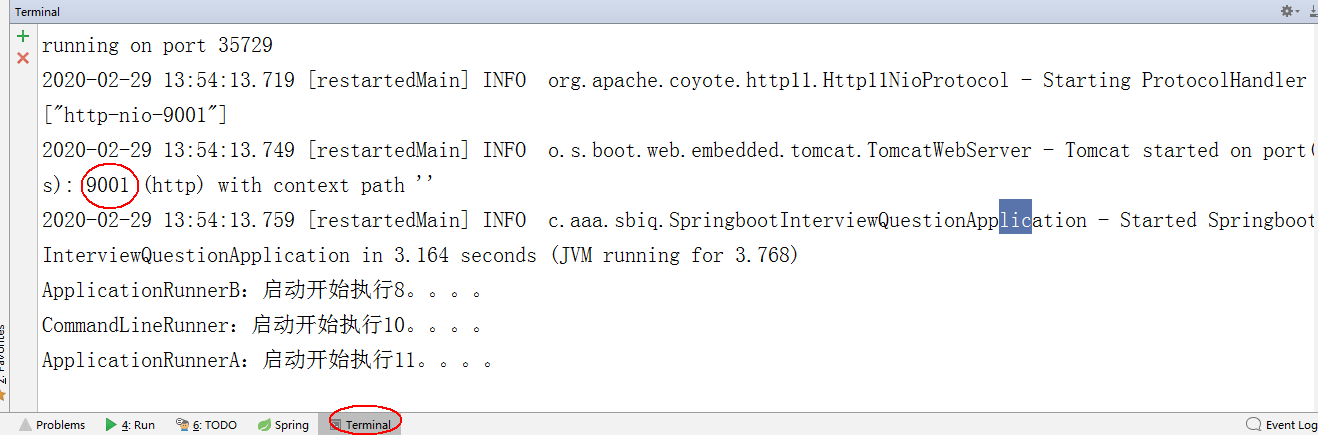


插件启动方式：

mvn spring-boot:run -Dspring-boot.run.profiles=dev



测试结果：



ctrl+c退出

jar运行方式：

打成jar包

运行

java -jar XXX.jar -Dspring-boot.run.profiles=dev

**Spring Boot 可以兼容老 Spring 项目吗，如何做？**

可以兼容，使用 @ImportResource 注解导入老 Spring 项目配置文件。

**15 保护 Spring Boot 应用有哪些方法？**

1.在生产中使用HTTPS

2.使用Snyk检查你的依赖关系(官网:<https://snyk.io/>不断地发现和修复开放源代码库和容器中的漏洞)

3.升级到最新版本（每一次升级，大多数是为了解决项目本身漏洞的）

4.启用CSRF保护(跨站请求伪造（英语：Cross-site request forgery 跨站请求攻击，简单地说，是攻击者通过一些技术手段欺骗用户的浏览器去访问一个自己曾经认证过的网站并运行一些操作（如发邮件，发消息，甚至财产操作如转账和购买商品）。由于浏览器曾经认证过，所以被访问的网站会认为是真正的用户操作而去运行。)

5.使用内容安全策略防止XSS攻击(Cross Site Scripting XSS攻击通常指的是通过利用网页开发时留下的漏洞，通过巧妙的方法注入恶意指令代码到网页，使用户加载并执行攻击者恶意制造的网页程序。这些恶意网页程序通常是JavaScript，但实际上也可以包括Java、 VBScript、ActiveX、 Flash 或者甚至是普通的HTML。攻击成功后，攻击者可能得到包括但不限于更高的权限（如执行一些操作）、私密网页内容、会话和cookie等各种内容。)

6.使用OpenID Connect进行身份验证(OpenID Connect是基于OAuth 2.0规范族的可互操作的身份验证协议。它使用简单的REST / JSON消息流来实现，和之前任何一种身份认证协议相比，开发者可以轻松集成。

OpenID Connect允许开发者验证跨网站和应用的用户，而无需拥有和管理密码文件。OpenID Connect允许所有类型的客户,包括基于浏览器的[JavaScript](https://link.jianshu.com/?t=http://lib.csdn.net/base/javascript)和本机移动应用程序,启动登录流动和接收可验证断言对登录用户的身份。)

7.管理密码？使用密码哈希！

8.安全地存储敏感信息

9.使用OWASP的ZAP测试您的应用程序(OWASP是一个开源的、非盈利的全球性安全组织，致力于应用软件的安全研究。其使命是使应用软件更加安全，使企业和组织能够对应用安全风险作出更清晰的决策。目前OWASP全球拥有220个分部近六万名会员，共同推动了安全标准、安全测试工具、安全指导手册等应用安全技术的发展。OWASP ZAP，全称：OWASP Zed Attack Proxy攻击代理服务器是世界上最受欢迎的免费安全工具之一。ZAP可以帮助我们在开发和测试应用程序过程中，自动发现 Web应用程序中的安全漏洞。另外，它也是一款提供给具备丰富经验的渗透测试人员进行人工安全测试的优秀工具)

10.让你的安全团队进行代码审查

**16 Spring Boot 2.X 有什么新特性？与 1.X 有什么区别？**

配置变更

JDK 版本升级

第三方类库升级

响应式 Spring 编程支持

HTTP/2 支持

配置属性绑定

更多改进与加强...

<https://mp.weixin.qq.com/s/-WWBvWpD0Prib02XoU1sjw>

**17 SpringBoot框架中，JavaBean都是单例的吗？多例怎么设置？**

spring bean作用域有以下5个：

singleton：单例模式，当spring创建applicationContext容器的时候，spring会欲初始化所有的该作用域实例，加上lazy-init就可以避免预处理；

prototype：原型模式，每次通过getBean获取该bean就会新产生一个实例，创建后spring将不再对其管理；

====下面是在web项目下才用到的===

request：搞web的大家都应该明白request的域了吧，就是每次请求都新产生一个实例，和prototype不同就是创建后，接下来的管理，spring依然在监听；

session：每次会话，同上；

global session：全局的web域，类似于servlet中的application。

好了，上面都说了spring的controller默认是单例，那很自然就是singleton了。

为什么spring要默认是单例呢？原因有二：

1、为了性能：这个不用废话了，单例不用每次都new，当然快了。

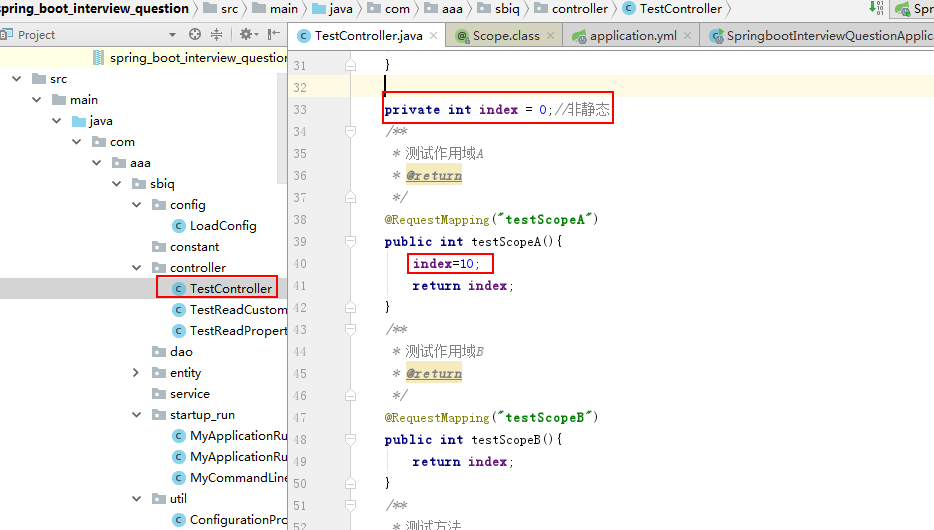
2、不需要多例：不需要多例会让很多人迷惑，因为spring mvc官方也没明确说不可以多例。

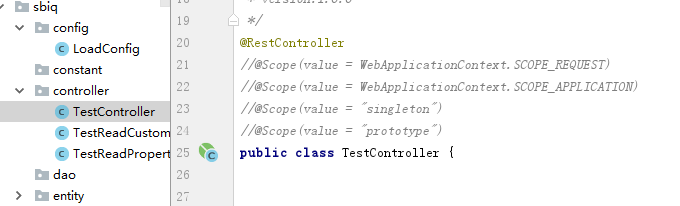
我这里说不需要的原因是看开发者怎么用了，如果你给controller中定义很多的属性，那么单例肯定会出现竞争访问了。

因此，只要controller中不定义属性，那么单例完全是安全的。下面给个例子说明下：

从此可见，单例是不安全的，会导致属性重复使用。

测试：





什么都不写或者

@Scope(value = "singleton")

<http://localhost:8080/testScopeA>

<http://localhost:8080/testScopeB>

方法A设置的值，方法B没有变化，说明是统一个TestController对象

@Scope(value = "prototype") //不是单例模式

<http://localhost:8080/testScopeA>

<http://localhost:8080/testScopeB>

方法A设置的值(index=10) ,方法B访问时发生了变化（index=0）,说明不是同一个TestController对象

@Scope(value = WebApplicationContext.SCOPE\_REQUEST)

<http://localhost:8080/testScopeA>

<http://localhost:8080/testScopeB>

方法A设置的值(index=10) ,方法B访问时发生了变化（index=0）,说明不是同一个TestController对象，每次请求，都产生一个新对象

@Scope(value = WebApplicationContext.SCOPE\_SESSION)

相同浏览器

<http://localhost:8080/testScopeA>

<http://localhost:8080/testScopeB>

另外浏览器

<http://localhost:8080/testScopeB>

相同浏览器，方法A设置的值，方法B都没有变化，只要浏览器不关闭，session不过期，不自行注销Session，每次请求访问的都是同一个TestController对象，另外浏览器，肯定session不同，所以访问的值是index=0

@Scope(value = WebApplicationContext.SCOPE\_APPLICATION)

相同浏览器

<http://localhost:8080/testScopeA>

<http://localhost:8080/testScopeB>

另外浏览器

<http://localhost:8080/testScopeB>

A方法设置的值，方法B ，不管是不是同一个浏览器，不管浏览器是否重新打开，得到的都是index=10,因为是全局的，只要服务不重启，该值全局访问都是相同的

最佳实践：

1、不要在controller中定义成员变量。

2、万一必须要定义一个非静态成员变量时候，则通过注解@Scope("prototype")，将其设置为多例模式。

<https://www.cnblogs.com/lingboweifu/p/11797926.html>

<https://blog.csdn.net/Kevin_Gu6/article/details/88547424>

springboot源码下载地址：

<https://github.com/spring-projects/spring-boot>