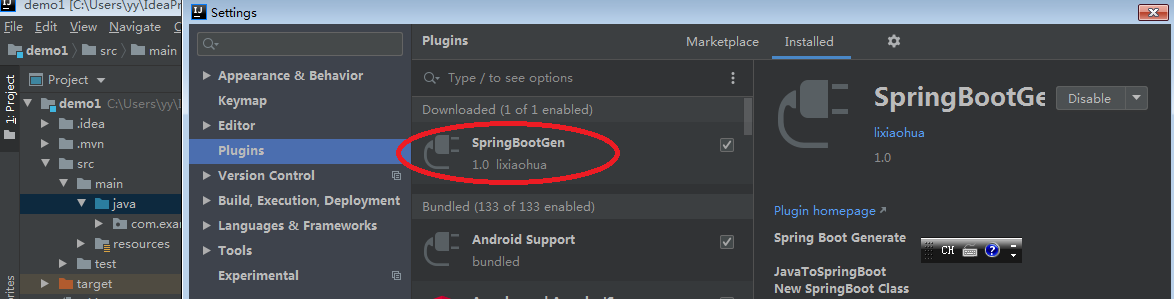
# 学习springboot的最基本的用法

## 搭建无ide的springboot的开发环境

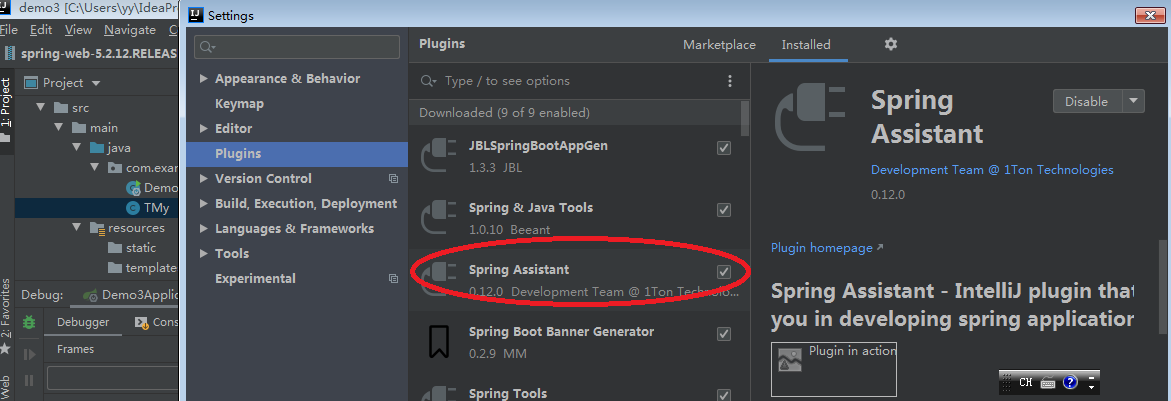
## 搭建IntelliJ IDEA的springboot的开发环境

### 运行IntelliJ IDEA

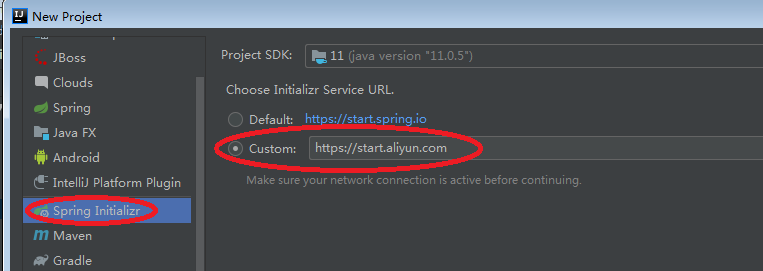
### 在IntelliJ安装SpringbootGen插件



### 安装Spring Assistant插件



### 新建工程， 选择



### 选择dependence

### 

然后，按下一步， 知道完成完成

### 等待intellij下载完成依赖

## 增加Controller类

@RestController  
class Controller{

}

## 增加get方法

### 无请求参数的get

<http://localhost:8080/hello1/>

hello1地址后面有其他内容，否则就会找不到页面，包404错误

@GetMapping("hello1")  
public String hello1(){  
 return "hello1()";  
}

### 参数直接在路径中

<http://localhost:8080/hello2/123>

Hello2地址后面必须要有其他内容，否则就会找不到页面，包404错误

http://localhost:8080/hello2/123/123也会报404错误

@GetMapping("hello2/{name}")  
public String hello2(@PathVariable("name") String name){  
 return "hello2() 获取到的name是：" + name;  
}

### 参数跟在 ? 号后面

<http://localhost:8080/hello3?name=123>

@GetMapping("hello3")  
public String hello3(@RequestParam("name") String name){  
 return "hello3() 获取到的name是：" + name;  
}

### 参数跟在 ? 号后面，且**required = false** 标注参数是非必须的

@GetMapping("hello4")  
public String hello4(@RequestParam(name = "name",required = false) String name){  
 return "hello4() 获取到的name是：" + name;  
}

则有下面几种场景：

#### 正常场景



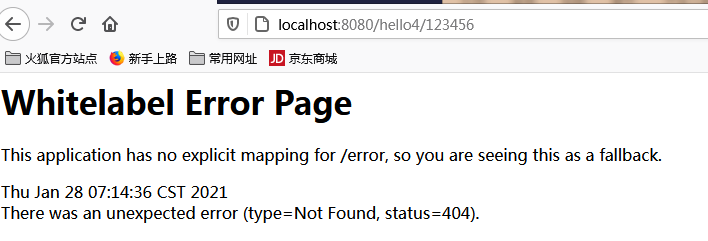
#### 指定名词不指定值



#### 不指定参数



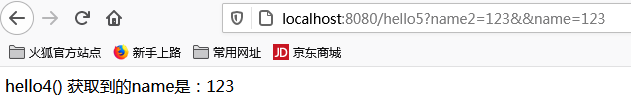
#### 不指定参数，而是跟斜杠



### 参数跟在 ? 号后面，且指定个默认值

@GetMapping("hello5")  
public String hello5(@RequestParam(name = "name",defaultValue = "xxx") String name){  
 return "hello4() 获取到的name是：" + name;  
}

#### 正常场景



#### 指定参数，但是缺少需要的参数



#### 指定参数但不指定值



#### 指定参数但不指定等号



#### 不指定参数



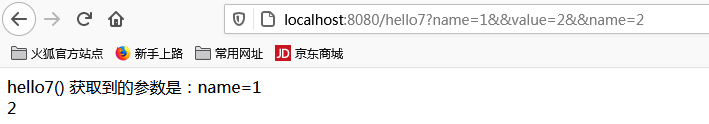
### 使用 map 来接收参数

@GetMapping("hello6")  
public String hello6(@RequestParam Map<String, Object> params){  
 return "hello6() 获取到的参数是：name=" + params.get("name") + ", vamue=" + params.get("value");  
}



### 接收一个数组

@GetMapping("hello7")  
public String hello7(@RequestParam("name") String[] names){  
 String result = "";  
 for(String name:names){  
 result += name + "<br>";  
 }  
 return "hello7() 获取到的参数是：name=" + result;  
}



### 定义类来接收参数

## 增加swagger，生成接口文档

### 在pom文件中：**导入依赖**

<dependency>  
 <groupId>io.springfox</groupId>  
 <artifactId>springfox-swagger2</artifactId>  
 <version>2.9.2</version>  
</dependency>  
<dependency>  
 <groupId>io.springfox</groupId>  
 <artifactId>springfox-swagger-ui</artifactId>  
 <version>2.9.2</version>  
</dependency>

增加后如下图：

### **编写Swagger2的配置类**

**添加如下的类**

@Configuration  
@EnableSwagger2  
public class SwaggerConfig {  
 @Bean  
 public Docket createRestApi() {  
 return new Docket(DocumentationType.*SWAGGER\_2*)  
 .pathMapping("/")  
 .select()  
 .apis(RequestHandlerSelectors.*basePackage*("com.example.demo"))  
 .paths(PathSelectors.*any*())  
 .build().apiInfo(new ApiInfoBuilder()  
 .title("SpringBoot整合Swagger")  
 .description("SpringBoot整合Swagger，详细信息......")  
 .version("9.0")  
 //.contact(new Contact("啊啊啊啊","blog.csdn.net","aaa@gmail.com"))  
 .license("The Apache License")  
 .licenseUrl("http://www.baidu.com")  
 .build());  
 }  
}

### **配置某个Controller**

**在controller类前加注解:**

@Api(tags = "用户接口")  
@RestController  
@EnableSwagger2  
class Controller{

### **配置某个接口方法**

@GetMapping("hello2/{name}")  
@ApiOperation("第二接口")  
@ApiImplicitParams({  
 @ApiImplicitParam(name = "name", value = "用户名", defaultValue = "李四")  
})

### **测试**

在浏览器输入地址:http://localhost:8080/swagger-ui.html#/

### **使用新的主题**

**在pom增加依赖**

<dependency>  
 <groupId>com.github.caspar-chen</groupId>  
 <artifactId>swagger-ui-layer</artifactId>  
 <version>1.1.3</version>  
</dependency>

**在浏览器输入新的地址**

<http://localhost:8080/docs.html>

调试

## 增加post方法

### 接收 **form-data** 格式的 **POST** 数据

@PostMapping("/hello")  
public String hello(@RequestParam("name") String name,  
 @RequestParam("age") Integer age) {  
 return "name：" + name + "\nage：" + age;  
}

### 接收 **form-data** 格式的 **POST** 数据， 并且容许数据不填

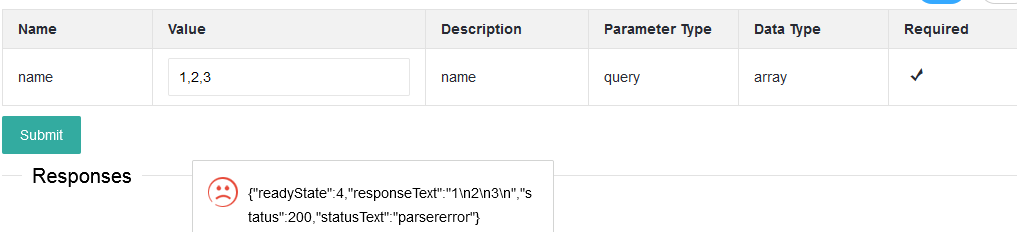
@PostMapping("/hello10")  
public String hello10(@RequestParam(name="name", defaultValue = "xxx") String name,  
 @RequestParam(name="age", required = false) Integer age) {  
 return "name：" + name + "\nage：" + age;  
}

### 使用map来接收参数

@PostMapping("/hello11")  
public String hello11(@RequestParam Map<String,Object> params) {  
 return "name：" + params.get("name") + "\nage：" + params.get("age");  
}

### 使用数组来接收参数

@PostMapping("/hello12")  
public String hello12(@RequestParam("name") String[] names) {  
 String result = "";  
 for(String name:names){  
 result += name + "\n";  
 }  
 return result;  
}



## 增加delete方法

### url传递参数

@DeleteMapping("/delete1/{id}")  
public String delete1(@PathVariable("id")String id){  
  
 return id;  
}

### json

## 增加put方法

1. 通过url传递参数

@PutMapping("/password")  
public String editPassword(String oldPassword,String newPassword){  
 return "";  
}

1. 通过data传递参数

## 增加数据库访问

## 增加数据库连接池的访问方法

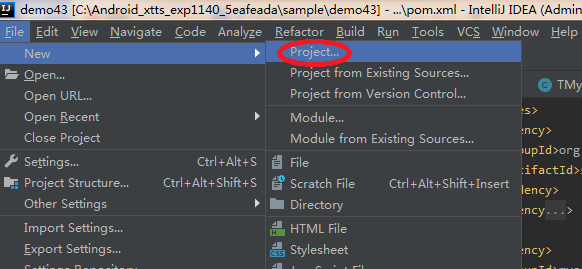
## 代码打包

# Springboot连接mysql数据库

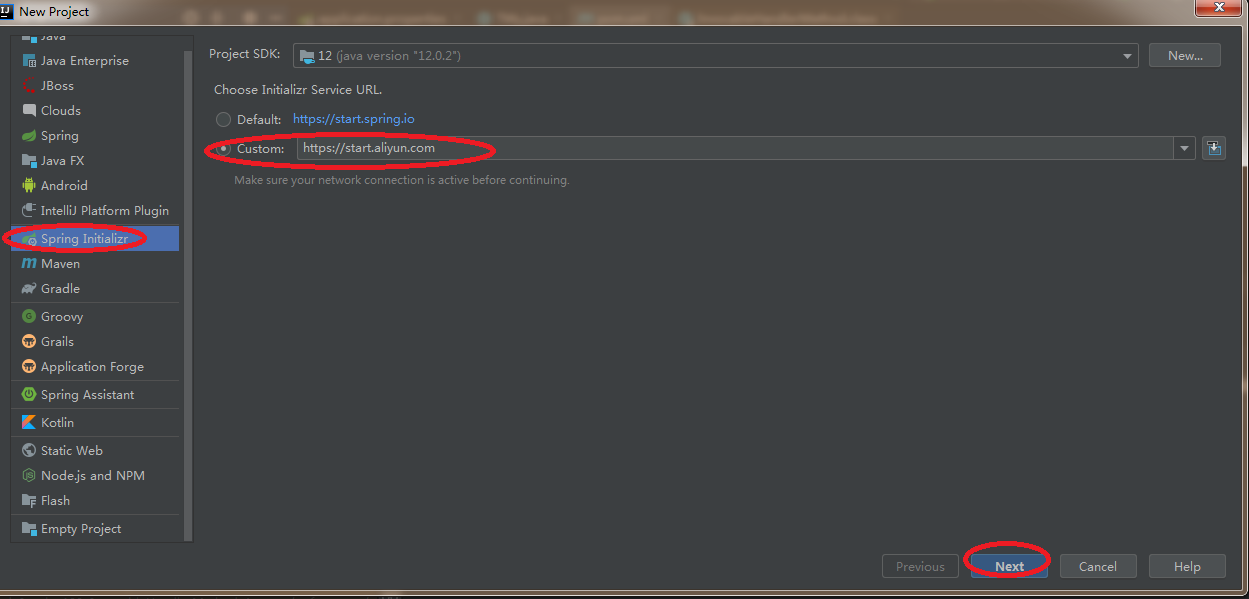
## 新建工程

### 打开intellij

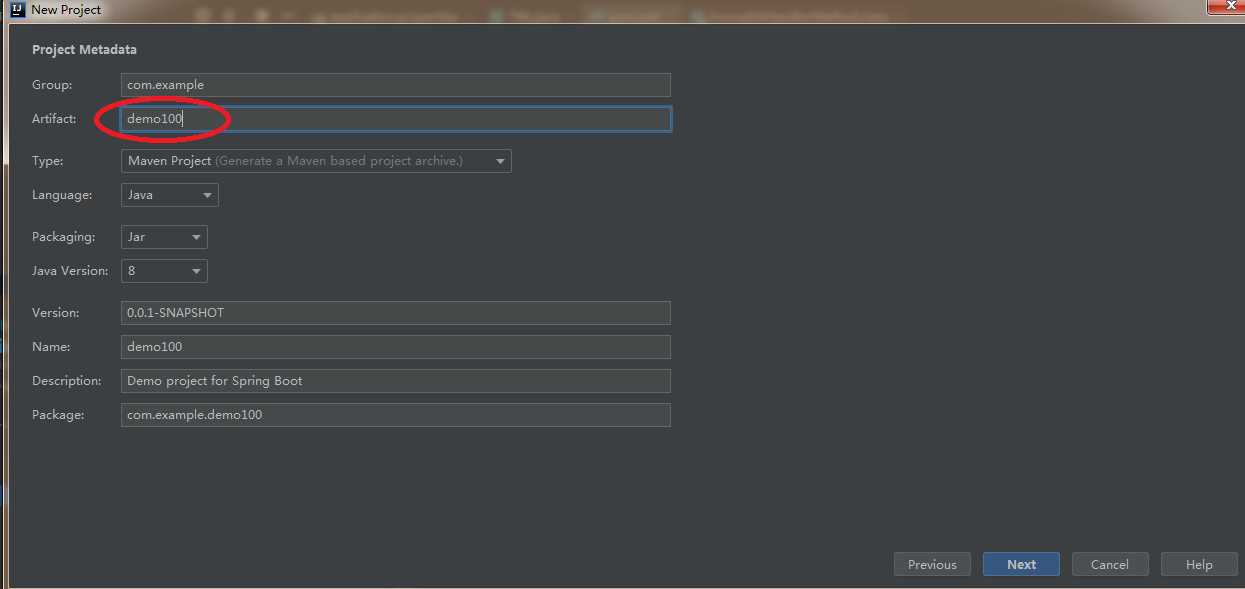
### 点击[File][New][Project]



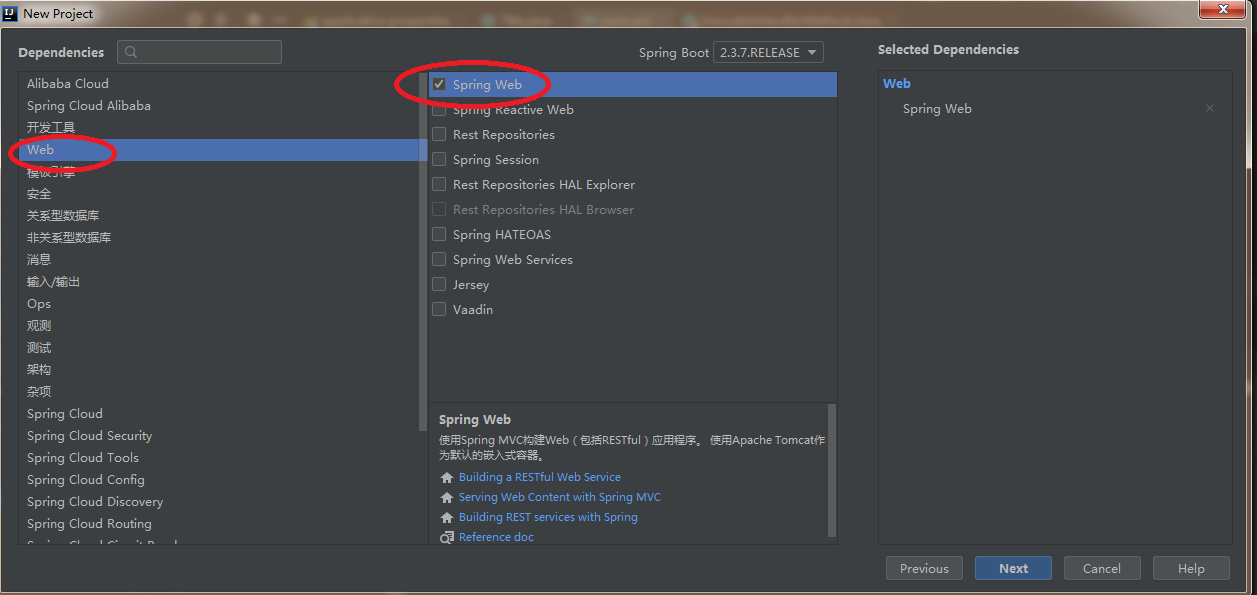
### 选择[Spring Initializr][Next]

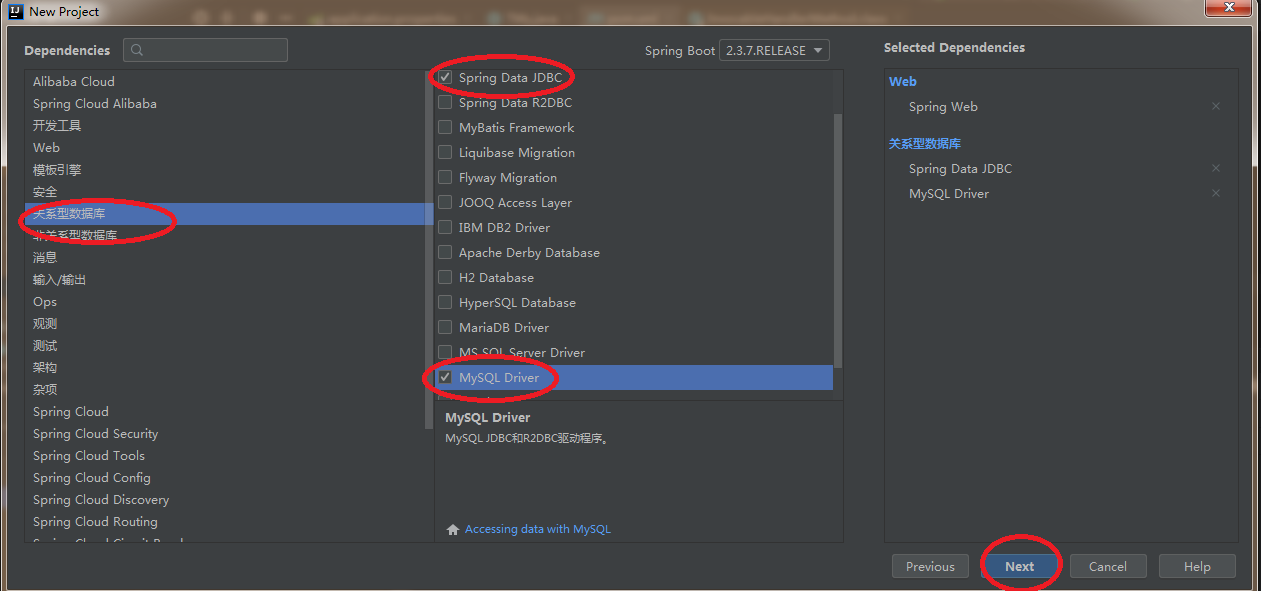


### 修改为自己的工程名字， Next

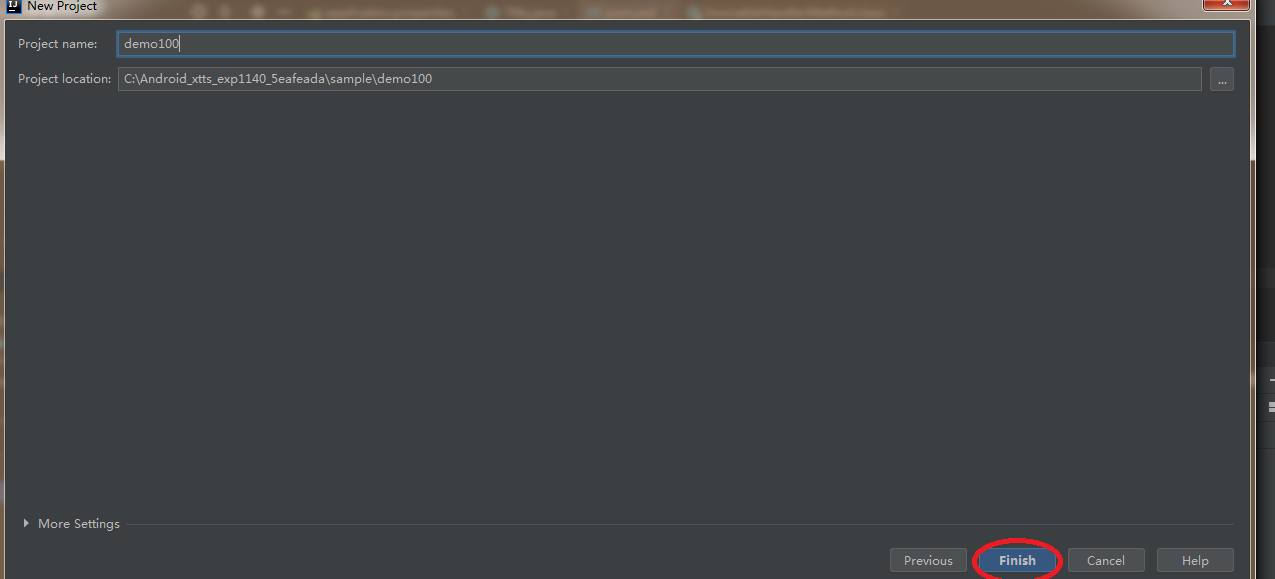


### 选择依赖包[spring web]+[spring Data JDBC]+[MySql Driver]

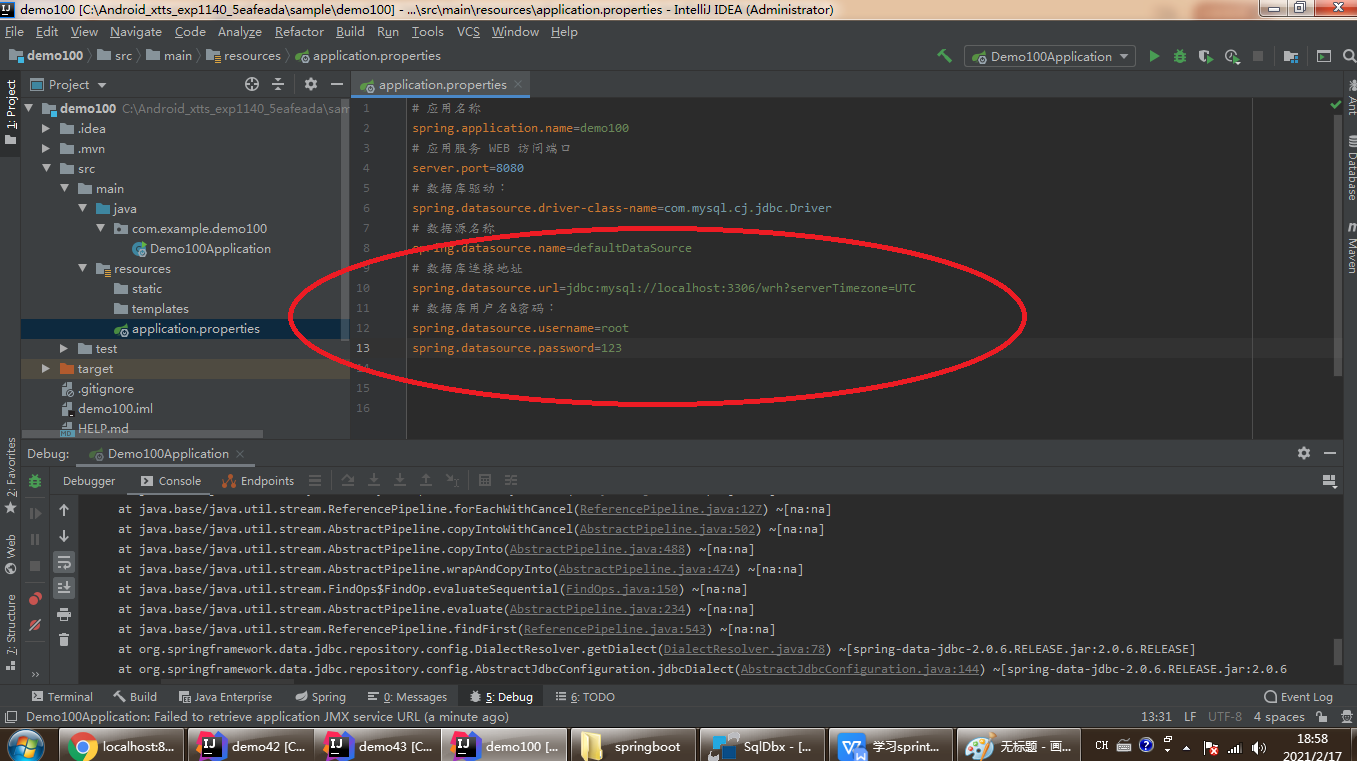




### [Finish]新建工程

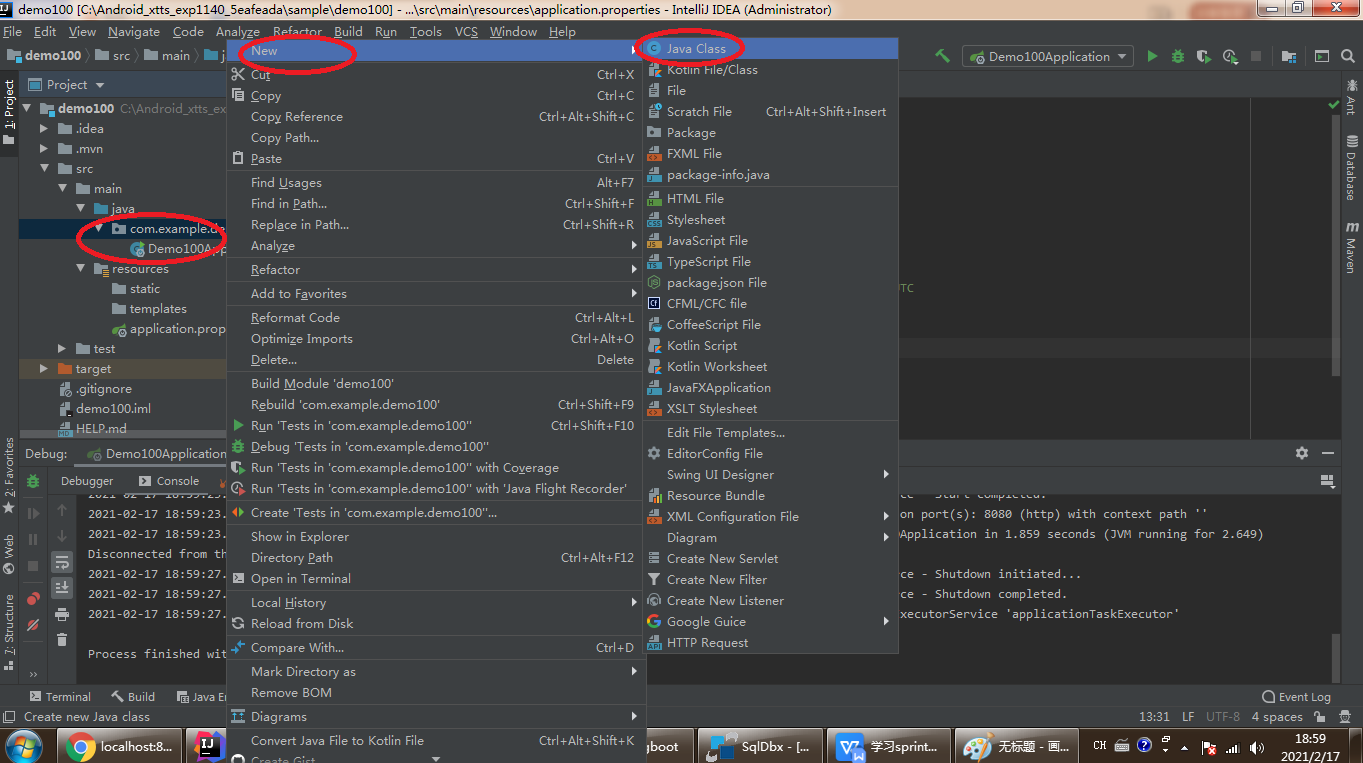


## 配置数据库连接

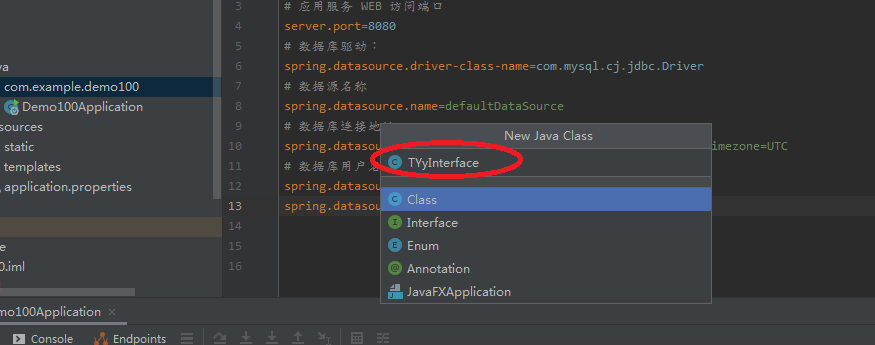


## 添加http的path接口

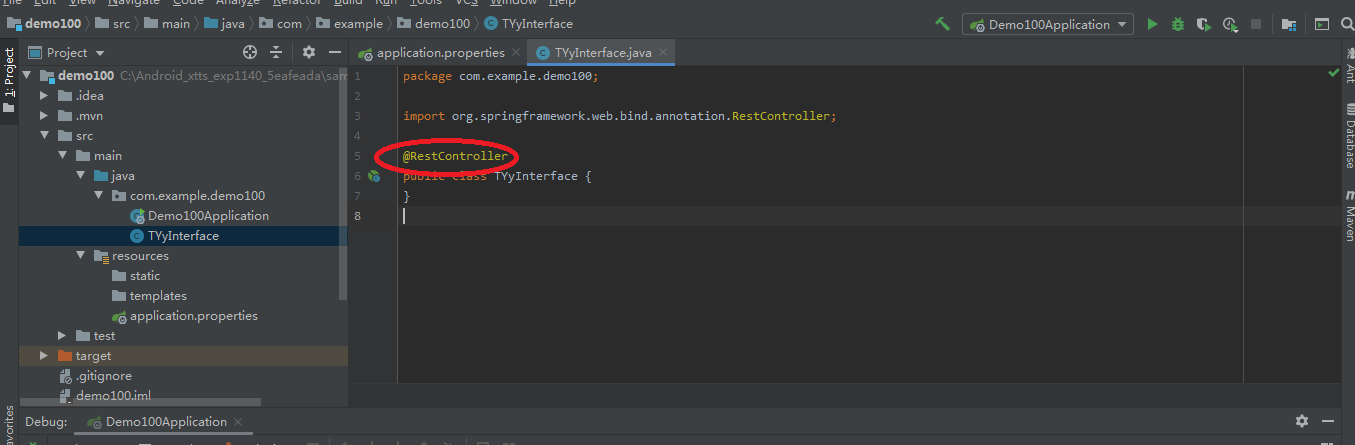
### 正在源代码目录， 右键菜单， 选择[New][Java Class]



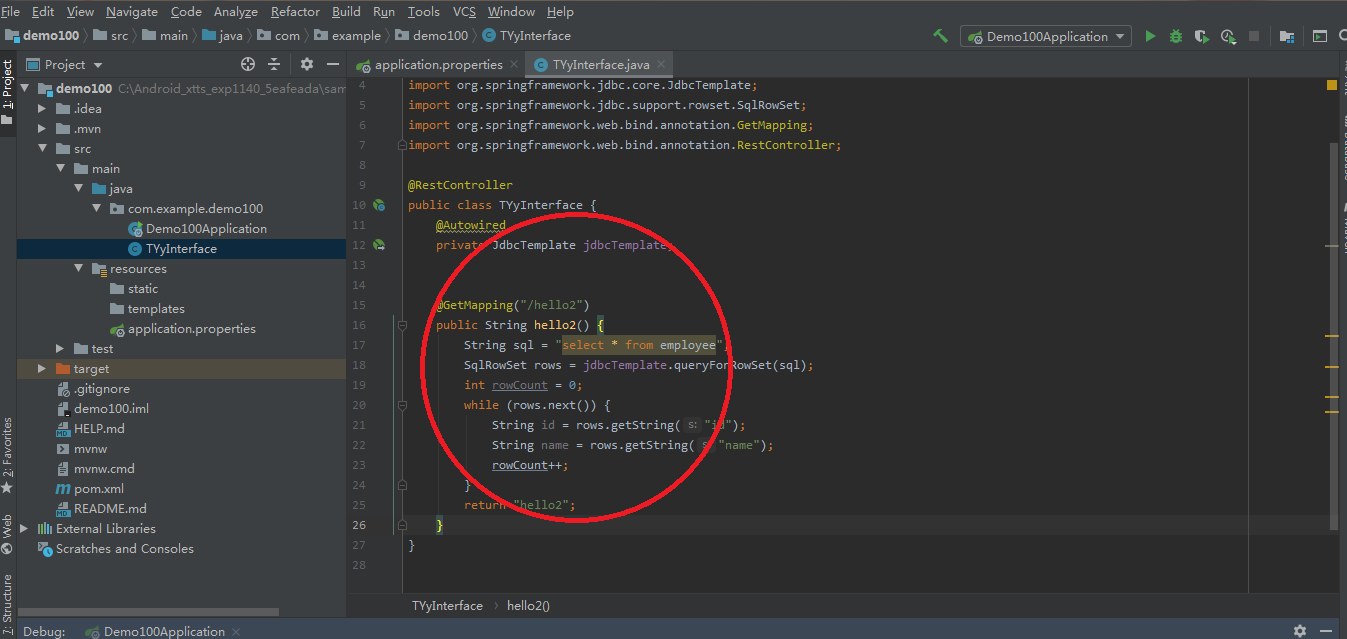
### 输入新类的类名TYyInterface



### 为新类添加一个注释



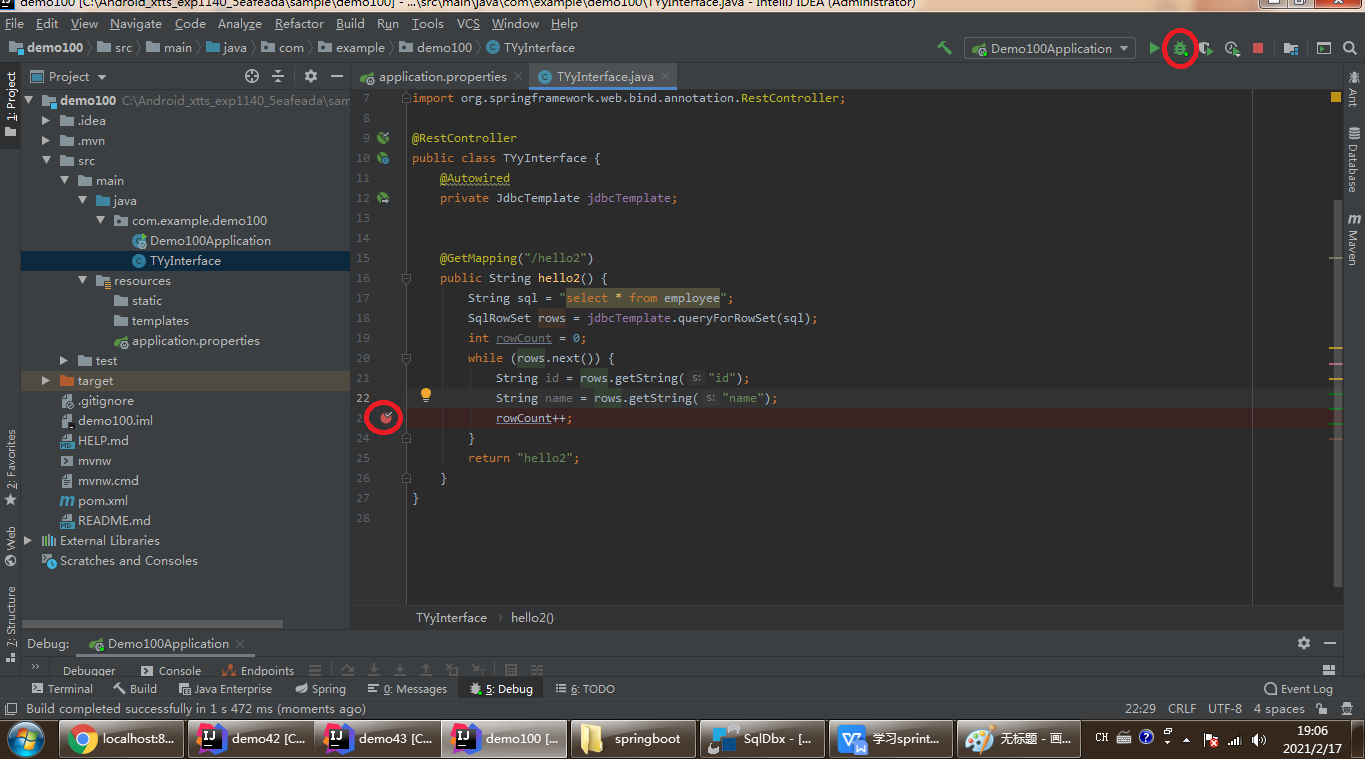
### 为新类增加一个接口



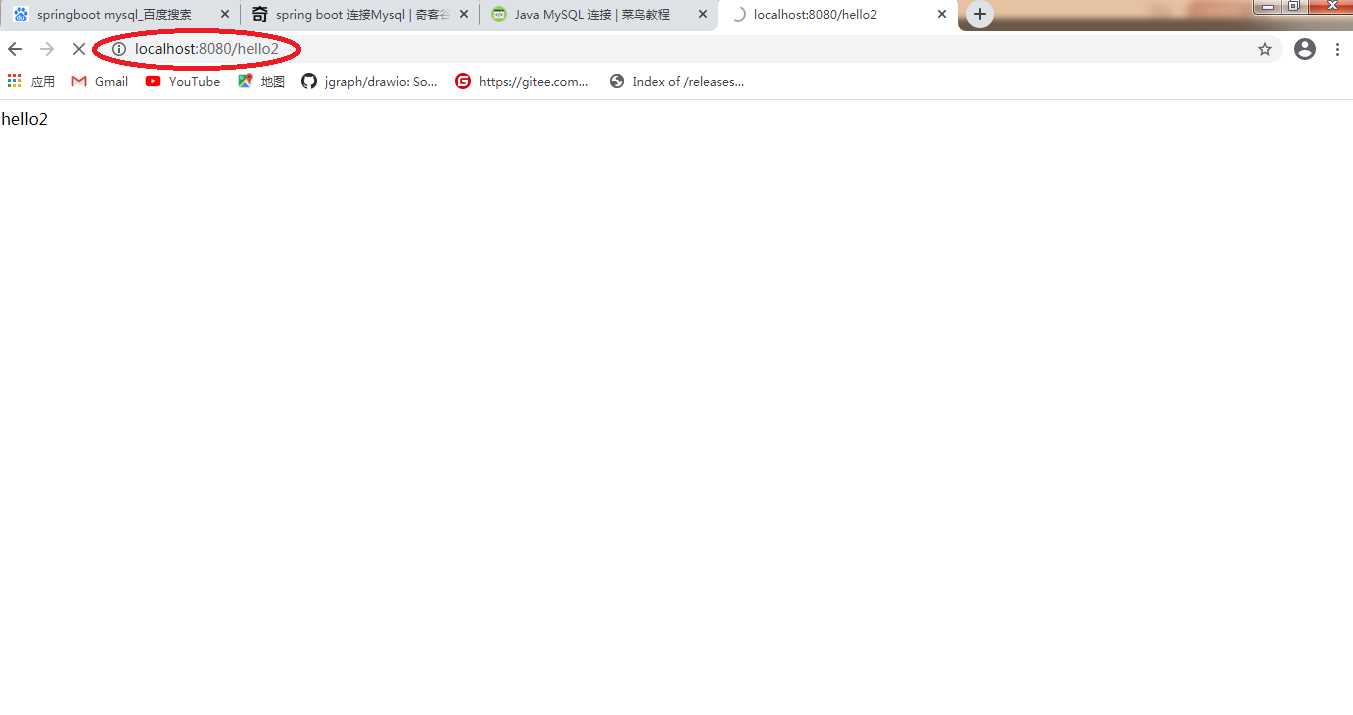
新接口的代码如下

@RestController  
public class TYyInterface {  
 @Autowired  
 private JdbcTemplate jdbcTemplate;  
  
  
 @GetMapping("/hello2")  
 public String hello2() {  
 String sql = "select \* from employee";  
 SqlRowSet rows = jdbcTemplate.queryForRowSet(sql);  
 int rowCount = 0;  
 while (rows.next()) {  
 String id = rows.getString("id");  
 String name = rows.getString("name");  
 rowCount++;  
 }  
 return "hello2";  
 }  
}

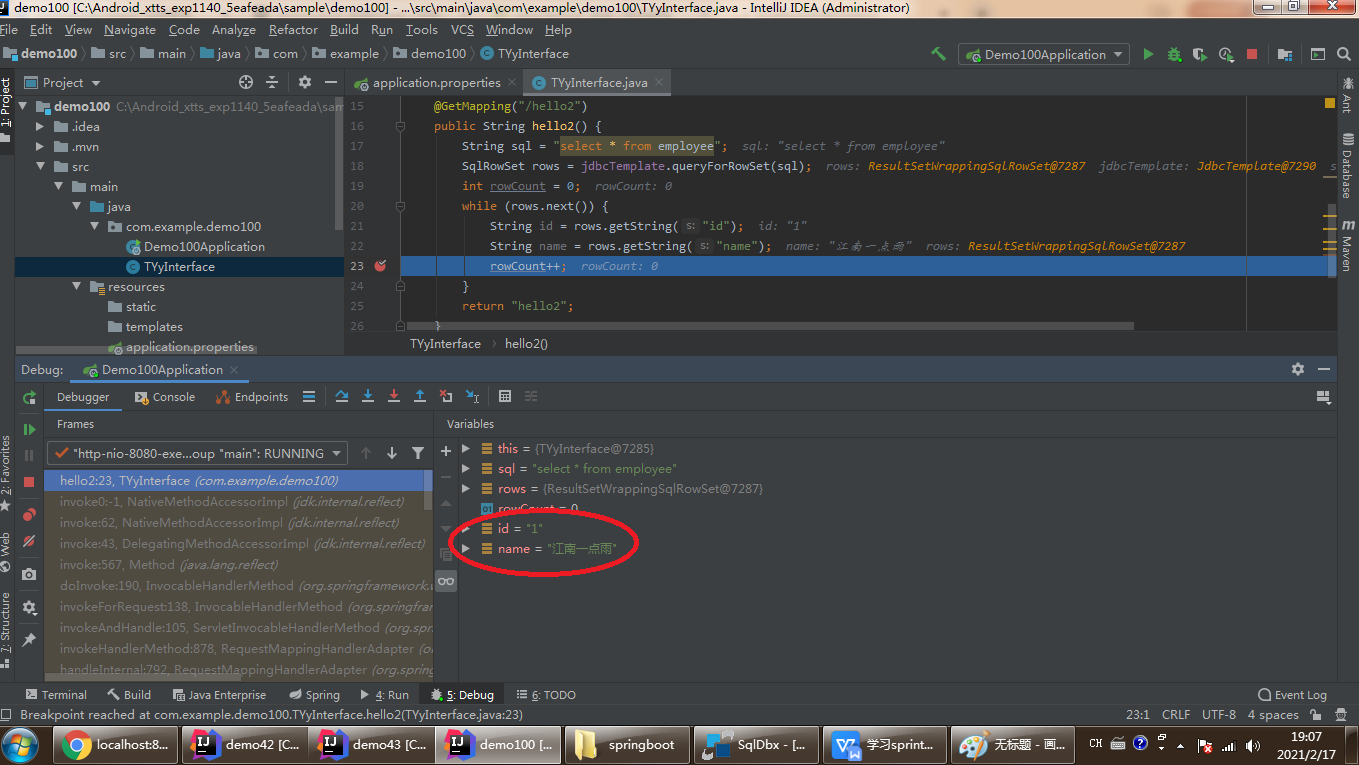
## 调试进程



## 浏览器请求连接



## 服务器调试



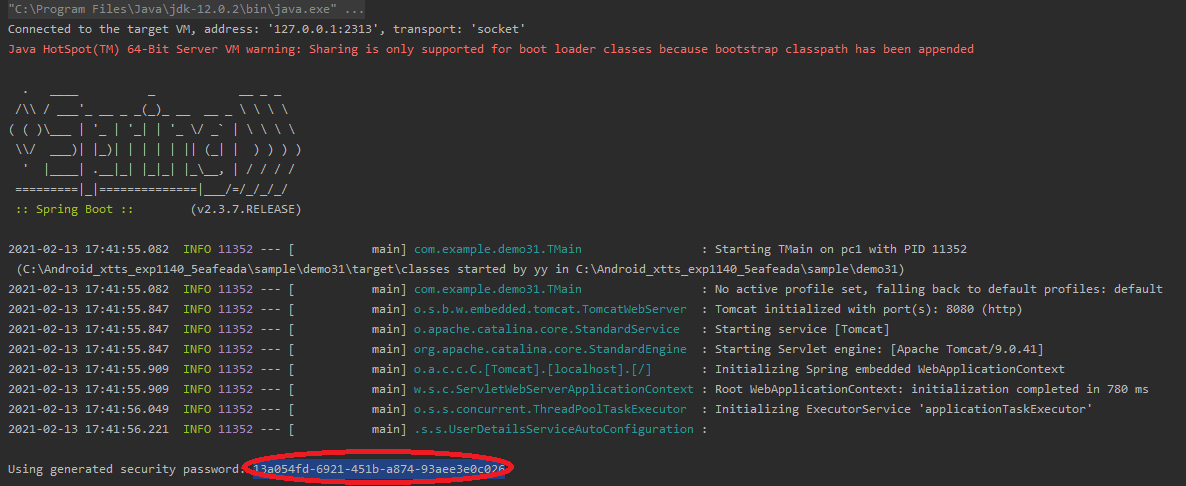
# 学习Spring security

## 配置步骤

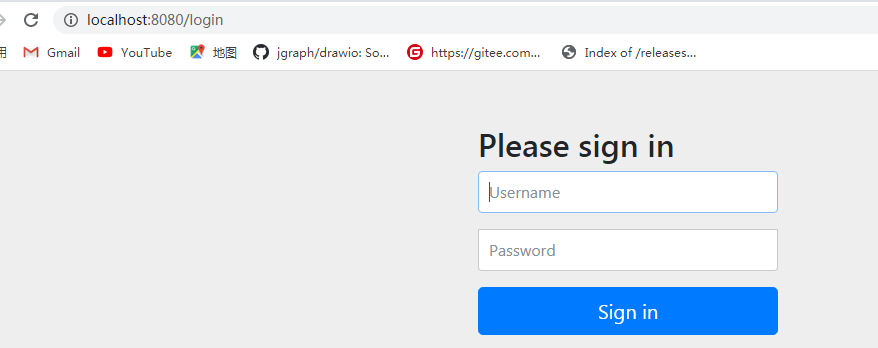
### 在pom文件中增加如下的依赖

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

运行代码后，在spring的控制塔器， 寻找spring生成的缺省密码

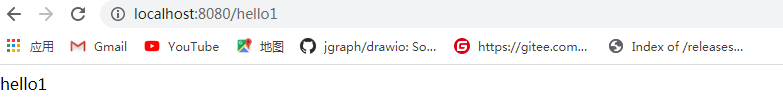


访问接口就需要先输入用户名密码认证

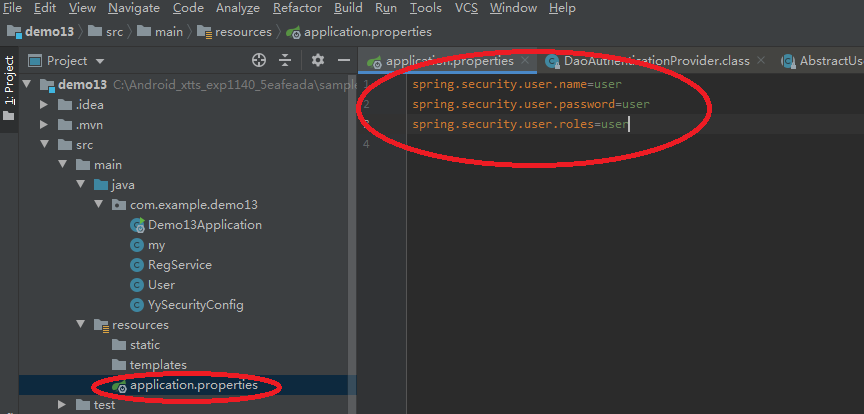


输入用户名: user, 密码见上上图的红色圈中的密码

然后就可以看到登录后成功的页面



### 在配置文件中配置用户名密码



运行后， 输入用户名密码： user/user,登录ok

### 基于内存认证

在工程中增加WebSecurityConfigurerAdapter， 并且增加下面3个函数：加密函数，用户认证授权函数，目录权限函数

@Configuration  
public class YySecurityConfig extends WebSecurityConfigurerAdapter {  
 PasswordEncoder passwordEncoder(){  
 return NoOpPasswordEncoder.*getInstance*();  
 }  
  
  
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.inMemoryAuthentication()  
 .withUser("yy").password("yy").roles("ADMIN")  
 .and().withUser("admin").password("admin1").roles("ADMIN")  
 .and().withUser("user").password("$2a$10$/Dvu/HoBwEgQT4fw/vS0QOVAHzaFNrdKmxl8zngFlY4InoVPnRrcy").roles("USER")  
 ;  
 }  
  
  
 protected void configure(HttpSecurity http) throws Exception {  
 http.authorizeRequests()  
 .antMatchers("/admin/\*\*").hasRole("ADMIN")  
 .antMatchers("/user/\*\*").hasRole("user")  
 //.anyRequest().authenticated()  
 .and().formLogin().loginProcessingUrl("/login").permitAll()  
 .and().logout().logoutUrl("/logout").clearAuthentication(true).invalidateHttpSession(true).addLogoutHandler(new LogoutHandler() {  
 @Override  
 public void logout(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Authentication authentication) {  
  
 }  
 }).logoutSuccessHandler(new LogoutSuccessHandler() {  
 @Override  
 public void onLogoutSuccess(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Authentication authentication) throws IOException, ServletException {  
 httpServletResponse.sendRedirect("/login");  
 }  
 }  
 )  
 .and().csrf().disable()  
 ;  
 }

}

### 编写自己的认证授权类

设置使得框架使用自己新编写的认证类

RegService MyUserDetailsService = new MyUserDetailsService();

protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 /\*auth.inMemoryAuthentication()  
 .withUser("yy").password("yy").roles("ADMIN")  
 .and().withUser("admin").password("admin1").roles("ADMIN")  
 .and().withUser("user").password("$2a$10$/Dvu/HoBwEgQT4fw/vS0QOVAHzaFNrdKmxl8zngFlY4InoVPnRrcy").roles("USER")  
 ;  
 \*/  
 auth.userDetailsService(regService);  
}

添加自己的认证授权类

public class MyUserDetailsService implements UserDetailsService {  
 @Override  
 public UserDetails loadUserByUsername(String s) throws UsernameNotFoundException {  
 MyUserDetails user = new MyUserDetails();  
 user.username = "user";  
 user.password = "user";  
 user.role = "ROLE\_user";  
 return user;  
 }  
}

添加自己的用户类

public class MyUserDetails implements UserDetails {  
  
 public String username;  
 public String password;  
 public String role;  
  
 @Override  
 public Collection<? extends GrantedAuthority> getAuthorities() {  
 List<SimpleGrantedAuthority> authorities = new ArrayList<>();  
 authorities.add(new SimpleGrantedAuthority(role));  
 return authorities;  
 }  
  
 @Override  
 public String getPassword() {  
 return password;  
 }  
  
 @Override  
 public String getUsername() {  
 return username;  
 }  
  
 @Override  
 public boolean isAccountNonExpired() {  
 return true;  
 }  
  
 @Override  
 public boolean isAccountNonLocked() {  
 return true;  
 }  
  
 @Override  
 public boolean isCredentialsNonExpired() {  
 return true;  
 }  
  
 @Override  
 public boolean isEnabled() {  
 return true;  
 }  
}

### 加密方法

PasswordEncoder passwordEncoder(){  
 //return NoOpPasswordEncoder.*getInstance*();  
 return new BCryptPasswordEncoder(10);  
}

然后通过下面函数求到密码加密后对用的密文：

static public int reg(String username, String password){  
 BCryptPasswordEncoder encoder= new BCryptPasswordEncoder(10);  
 String pass=encoder.encode(password);  
 //return saveToDb(username,pass);  
 return 1;  
}

最后，修改配置的密码

.and().withUser("user").password("$2a$10$/Dvu/HoBwEgQT4fw/vS0QOVAHzaFNrdKmxl8zngFlY4InoVPnRrcy").roles("USER")

### 注销

在登录配置后面增加logout的配置

.and().logout().logoutUrl("/logout").clearAuthentication(true).invalidateHttpSession(true).addLogoutHandler(new LogoutHandler() {  
 @Override  
 public void logout(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Authentication authentication) {  
  
 }  
 }).logoutSuccessHandler(new LogoutSuccessHandler() {  
 @Override  
 public void onLogoutSuccess(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Authentication authentication) throws IOException, ServletException {  
 httpServletResponse.sendRedirect("/login");  
 }  
 }  
 )

# Spring boot中调用其他rest接口

@GetMapping("/post")  
public String post() {  
 RestTemplate restTemplate = new RestTemplate();  
 String ret = restTemplate.getForObject("http://localhost:8080/select",String.class);  
 return ret;  
}

# Spring的特性Ioc 控制反转

## 怎么使用ioc

1. 名词解释

下面类容，摘自百度百科

**控制反转**（Inversion of Control，缩写为**IoC**），是[面向对象编程](https://baike.baidu.com/item/%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1%E7%BC%96%E7%A8%8B/254878" \t "https://baike.baidu.com/item/%E6%8E%A7%E5%88%B6%E5%8F%8D%E8%BD%AC/_blank)中的一种设计原则，可以用来减低计算机[代码](https://baike.baidu.com/item/%E4%BB%A3%E7%A0%81/86048" \t "https://baike.baidu.com/item/%E6%8E%A7%E5%88%B6%E5%8F%8D%E8%BD%AC/_blank)之间的[耦合度](https://baike.baidu.com/item/%E8%80%A6%E5%90%88%E5%BA%A6/2603938" \t "https://baike.baidu.com/item/%E6%8E%A7%E5%88%B6%E5%8F%8D%E8%BD%AC/_blank)。其中最常见的方式叫做**[依赖注入](https://baike.baidu.com/item/%E4%BE%9D%E8%B5%96%E6%B3%A8%E5%85%A5/5177233" \t "https://baike.baidu.com/item/%E6%8E%A7%E5%88%B6%E5%8F%8D%E8%BD%AC/_blank)**（Dependency Injection，简称**DI**），还有一种方式叫“依赖查找”（Dependency Lookup）。通过控制反转，对象在被创建的时候，由一个调控系统内所有对象的外界实体将其所依赖的对象的引用传递给它。也可以说，依赖被注入到对象中。

Class A中用到了Class B的对象b，一般情况下，需要在A的代码中显式的new一个B的对象。

采用依赖注入技术之后，A的代码只需要定义一个私有的B对象，不需要直接new来获得这个对象，而是通过相关的容器控制程序来将B对象在外部new出来并注入到A类里的引用中。而具体获取的方法、对象被获取时的状态由配置文件（如XML）来指定。 [1]

1. 注解实现
2. 定义2个类，如下

public class Test {  
 public String ClassName(){  
 return "Test";  
 }  
}

public class Test1 extends Test{  
 public String ClassName(){  
 return "Test1";  
 }  
}

1. 定义引用基类字段的代码

@RestController  
public class MyIt {  
 @Autowired  
 private Test m\_test;  
 @GetMapping("//")  
 public String hello(){  
 return m\_test.ClassName();  
 }  
}

1. 在Test注释为Service，则返回为Test

表示， Test的示例被输入到了m\_test

1. 在Test1注释@Sercice，则返回Test1

表示，Test1的示例被注入到了m\_test

1. 如果Test， Test1都注释为@Service, 则m\_test申明出就会报错：说有2个Bean对象都可以被注入， 不知道使用哪个来注入
2. 还是通过@Bean注解，来生成被被注入的实例

@Configuration  
public class MainConfig {  
 @Bean  
 public Test1 test1(){  
 return new Test2();  
 }  
}

1. 还有其他很多的实现办法

## Ioc原理

首先扫描指定目录下的所有的class文件

加载每个class文件， 遍历class的成员变量， 成员函数；然后遍历类，成员函数， 成员变量的注解

对于使用@Service的类， 直接在内存的map表中，生成一个全局的单例对象”类名”=类的单实例

对于被@Autowired注释的成员变量， 直接根据其类名称，从内存中找到该类名称对应的类的单实例，设置之。

# Spring的特性：aop切面编程

## 怎么使用aop

1. 原来的代码逻辑

@Service  
public class OldClass {  
 public int div(int i,int j){  
 System.*out*.println("MathCalculator...div...");  
 return i/j;  
 }  
}

@RestController  
public class TMyIt {  
 @Autowired  
 public OldClass m\_OldClass;  
  
 @GetMapping("//")  
 public String hello(){  
 m\_OldClass.div(10,2);  
 return "hello-aop";  
 }  
}

1. 定义一个切面（就是一个inject类）

@Service  
@Aspect  
public class InjectClass {  
 @Pointcut("execution(public int com.example.aop.OldClass.\*(..))")  
 public void pointCut(){};  
  
 @Before("pointCut()")  
 public void logStart(JoinPoint joinPoint){  
 Object[] args = joinPoint.getArgs();  
 System.*out*.println(""+joinPoint.getSignature().getName()+"运行。。。@Before:参数列表是：{"+ Arrays.*asList*(args)+"}");  
 }  
}

1. 运行后

div运行。。。@Before:参数列表是：{[10, 2]}

MathCalculator...div...

1. 但是对代码new 的对象没有效果

也就是说， aop仅仅对由注解生成的单例对象有效

## aop原理（动态代理）

public interface Person {  
 void giveTask();  
}

public class Student implements Person{  
  
 private String name;  
 public Student(String name) {  
 this.name = name;  
 }  
  
 public void giveTask() {  
 System.*out*.println(name + "交语文作业");  
 }  
}

public class ProxyHandler<T> implements InvocationHandler {  
  
 T target;  
  
 public ProxyHandler(T target) {  
 this.target = target;  
 }  
  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 String x = method.getName();  
 if(!x.equals("toString")) {  
 System.*out*.println("代理执行[" + x + "]方法");  
 }  
 Object result = method.invoke(target, args);  
 return result;  
 }  
}

测试：

Student s = new Student("yy");  
ProxyHandler<Student> p = new ProxyHandler<Student>(s);  
Person stuProxy = (Person) Proxy.*newProxyInstance*(Person.class.getClassLoader(), new Class<?>[]{Person.class}, p);  
stuProxy.giveTask();

所以， spring的aop就是在生成全局的单例对象时， 对于设置过要切入的类， 生成代理的动态调用， 以便在每次调用函数时被拦截。

注意： 上面的“public Object invoke(Object proxy, Method method, Object[] args)”在设置断点调试的场景下， 每次都会被冲入多次。所以尽量别再这调试

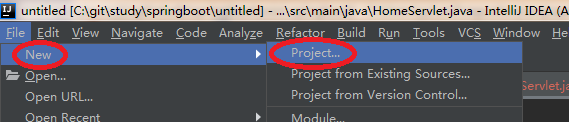
# Swigger codegen

# 自己动手写“最简版的springboot，”最基本的原理）

## 编写embed tamcat的框架

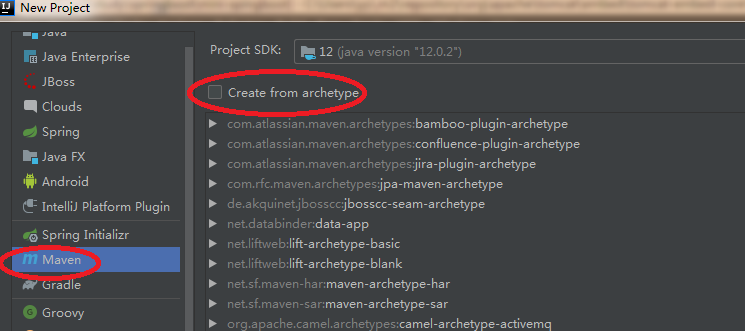
### 打开intellj

### 新建项目

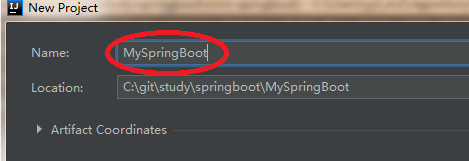


### 选择创建Maven的工程

注意： 不选择“Create from archetype”

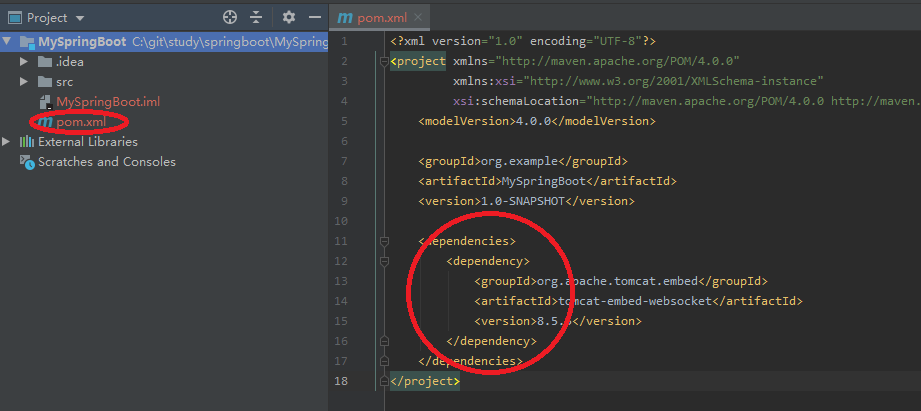


### 设置工程名称为”MySpringBoot”



### 完成工程的创建

### 在工程的pom文件中添加依赖

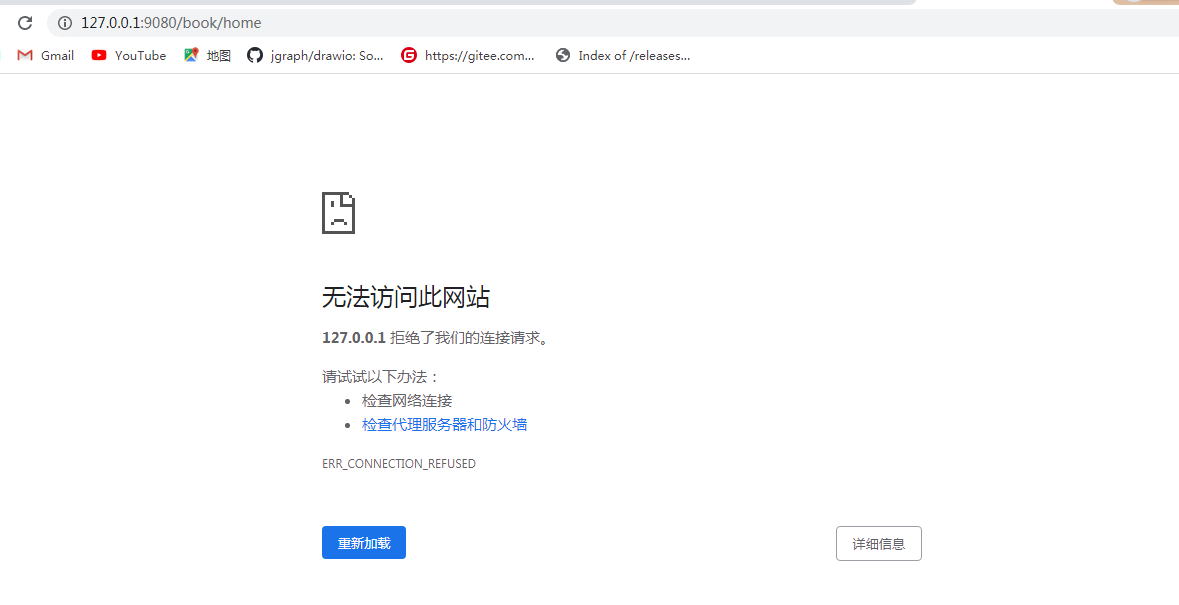


依赖如下

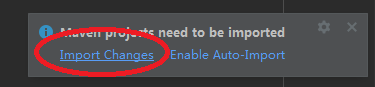
<dependencies>  
 <dependency>  
 <groupId>org.apache.tomcat.embed</groupId>  
 <artifactId>tomcat-embed-websocket</artifactId>  
 <version>8.5.5</version>  
 </dependency>  
</dependencies>

注意： 版本必须填8.5.5， 其他的版本不一定ok（例如：10.0.2就测试不行）

<version>10.0.2</version>

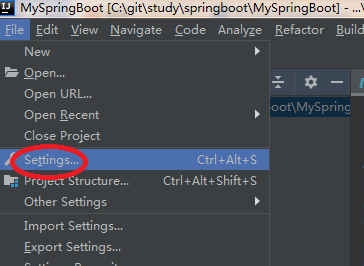


### 添加依赖后，等会儿Intellij将会弹出下面的对话框， 选择“Import Change”

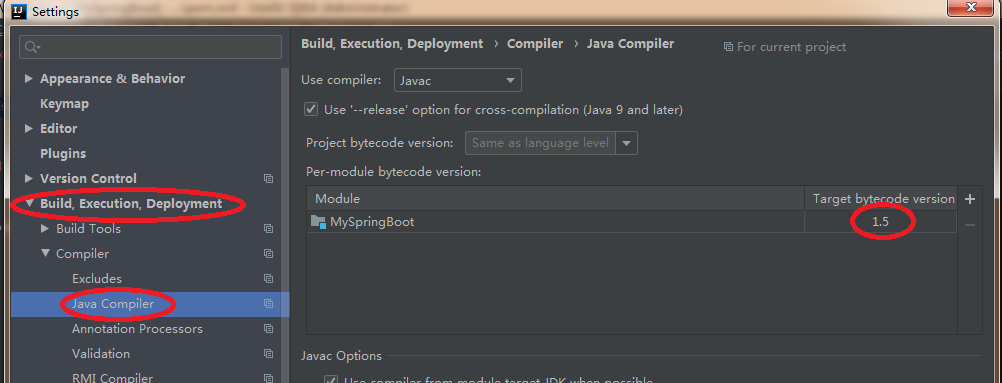


### “Import Change”后，工程的java编译版本会被自动修改成1.5， 这个是不对的， 需要手工改回为”9”

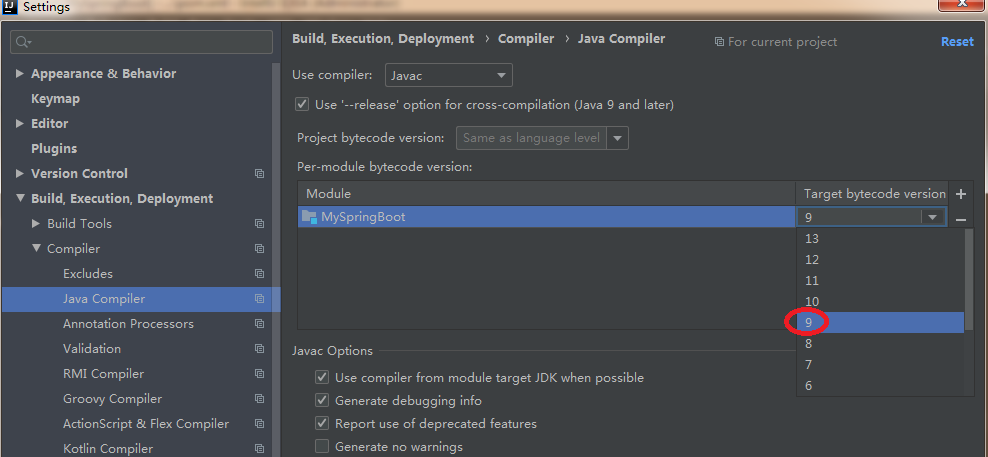
1. 选择菜单[File][Settings]



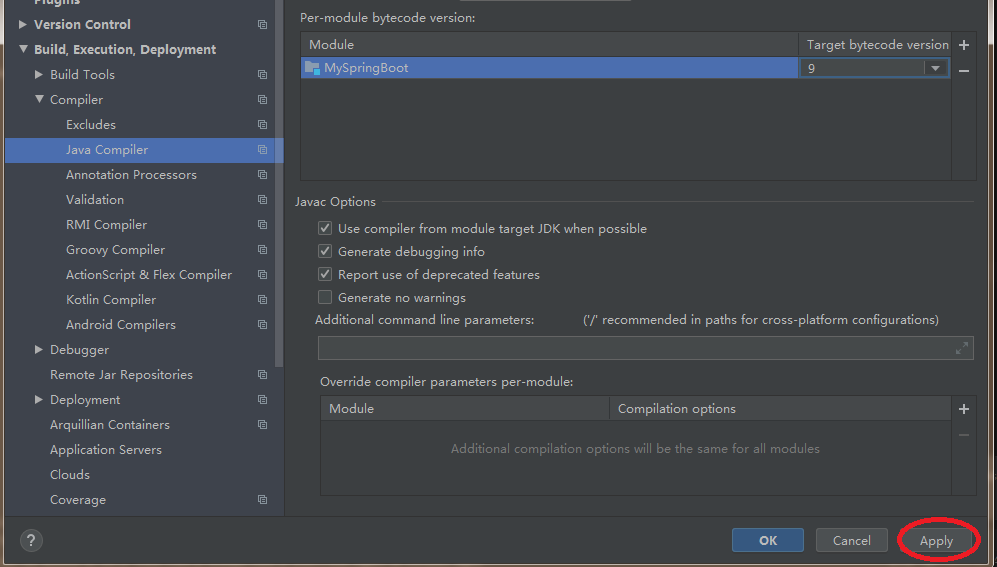
1. 展开[Build, Execution,Deployment][Java Comolier], 可以看到”Target bytecode version”被修改成了1.5



1. 将”Target bytecode version”修改为9



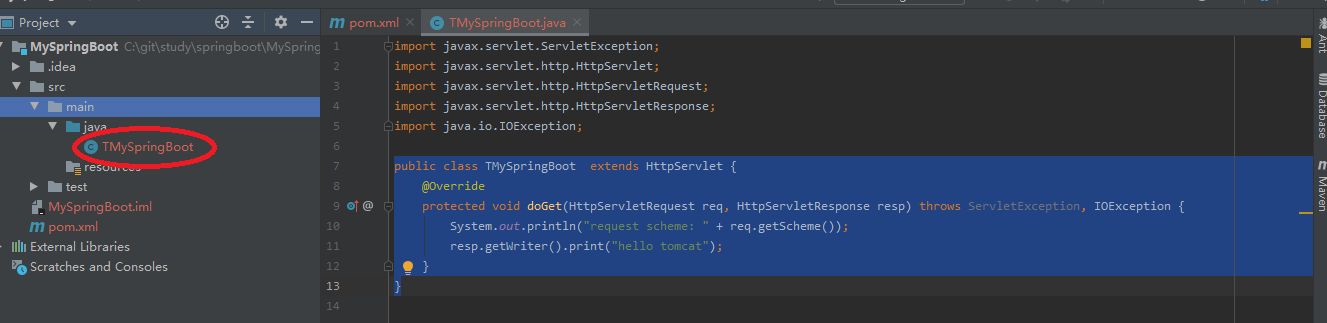
1. 应用修改



### 向工程添加新类TMySpringBoot

public class TMySpringBoot extends HttpServlet {  
 @Override  
 protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  
 System.*out*.println("request scheme: " + req.getScheme());  
 resp.getWriter().print("hello tomcat");  
 }  
}

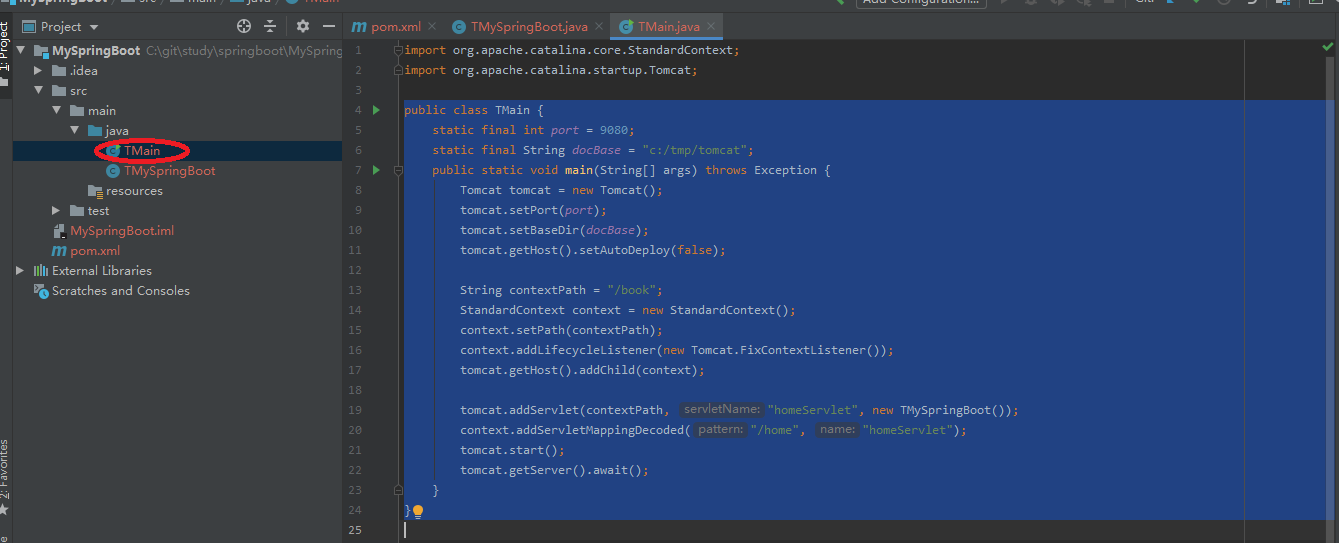
添加后如下



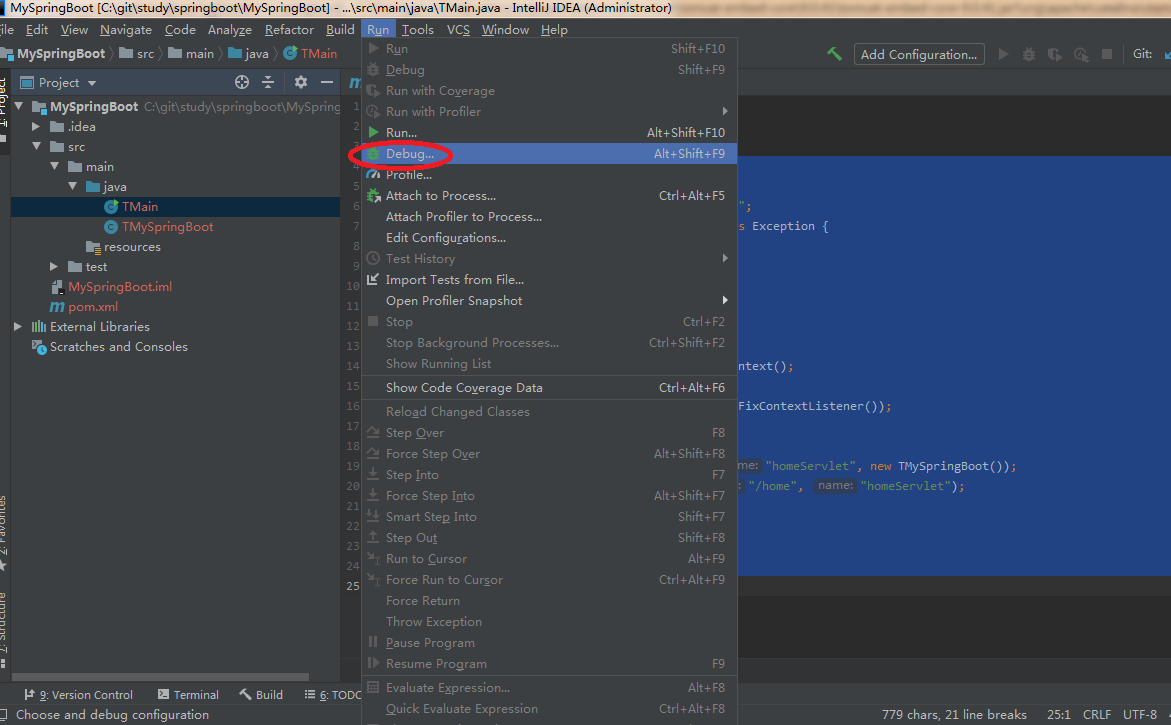
### 向工程添加TMain类

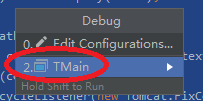
public class TMain {  
 static final int *port* = 9080;  
 static final String *docBase* = "c:/tmp/tomcat";  
 public static void main(String[] args) throws Exception {  
 Tomcat tomcat = new Tomcat();  
 tomcat.setPort(*port*);  
 tomcat.setBaseDir(*docBase*);  
 tomcat.getHost().setAutoDeploy(false);  
  
 String contextPath = "/book";  
 StandardContext context = new StandardContext();  
 context.setPath(contextPath);  
 context.addLifecycleListener(new Tomcat.FixContextListener());  
 tomcat.getHost().addChild(context);  
  
 tomcat.addServlet(contextPath, "homeServlet", new TMySpringBoot());  
 context.addServletMappingDecoded("/home", "homeServlet");  
 tomcat.start();  
 tomcat.getServer().await();  
 }  
}

添加后如下图

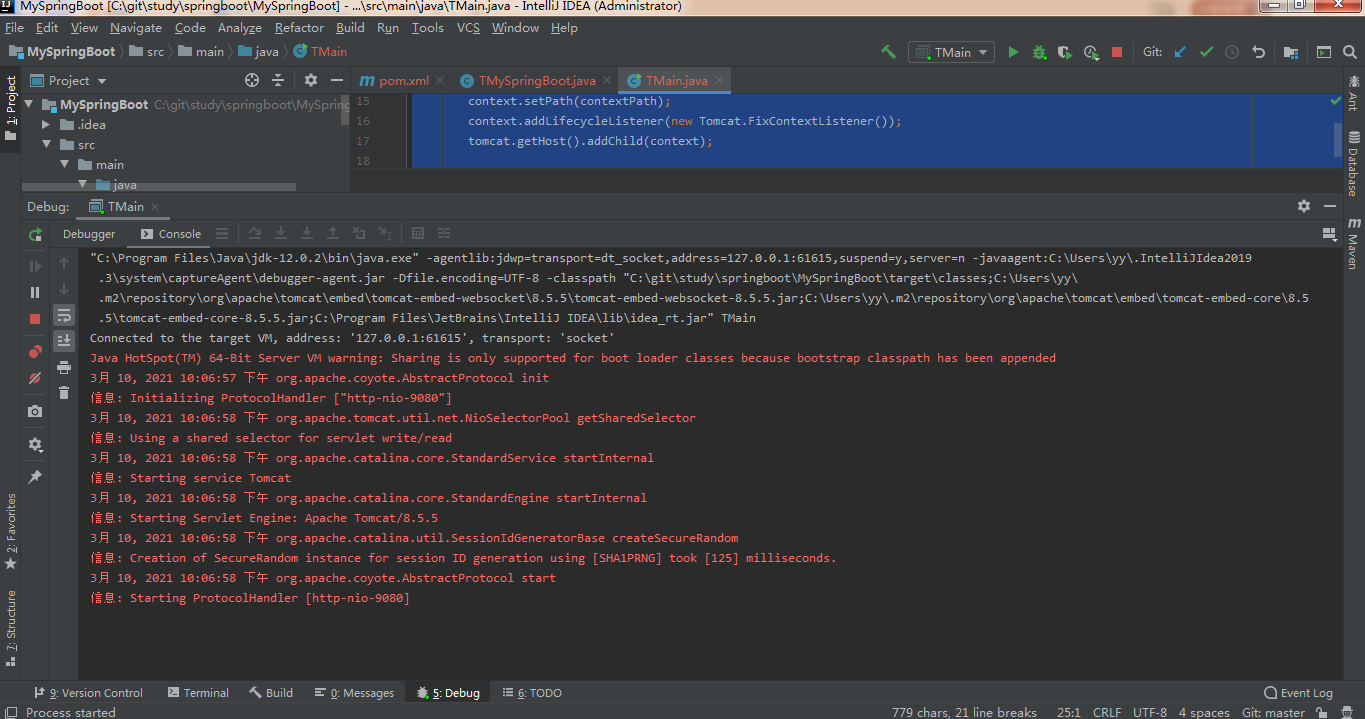


### 运行程序



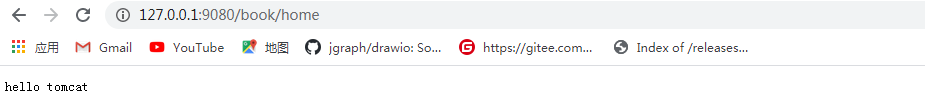


### 看到下面的字样， 表示tomcal已经运行起来了



### 测试

在浏览器地址栏输入：<http://127.0.0.1:9080/book/home>



### 向TMySpringBoot类添加下面的方法

@Override  
public void init(ServletConfig config) throws ServletException {  
 super.init(config);  
}

### 设置断点， 重新调试

可以看到早起启动完成都不会调用init函数， 知道第一个http请求到来后，才会首先执行init函数，再执行doGet函数；后面的更多次的http请求， 就不再执行init函数，而是只执行具体的请求函数（例如：只执行doGet函数）

## 添加自己的注释

## 增加init函数，扫描所有的类，实现ioc,apo,自动映射配置等

# 自己动手写“简化版的spring boot”，了解spring boot的大致结构

# 学习spring boot的源代码

# spring boot微服务部署

# spring boot反向代理