**Spring Security介绍**

**Shiro**

Shrio轻量、简单、易于集成，但是对OAuth2*（即开放授权，是为用户资源的授权定义了一个安全、开放及简单的标准，第三方无需知道用户的账号及密码，就可获取到用户的授权信息）*支持不够，在SpringBoot面前无法充分展示自己的优势。

**核心功能**

* 认证：登录
* 授权：权限鉴定，看看请求是否具备相应的权限

一些比较常见的认证方式：

* HTTP BASIC authentication headers：基于IETF RFC 标准。
* HTTP Digest authentication headers：基于IETF RFC 标准。
* HTTP X.509 client certificate exchange：基于IETF RFC 标准。
* LDAP：跨平台身份验证。
* Form-based authentication：基于表单的身份验证。
* Run-as authentication：用户用户临时以某一个身份登录。
* OpenID authentication：去中心化认证。

授权方式：Spring Security支持基于URL的请求授权，支持方法访问授权以及对象访问授权

**Spring Security初体验**

**1.新建项目**

引入Spring Security和Web依赖

Spring Security核心包如下：

计算机生成了可选文字:
Maven:org.springframework.security:spring-security-config:5.2.4.RELEASE
Maven:org.springframework.security:spring-security-core:5.2.4.RELEASE
Maven:org.springframework.security:spring-security-test:5.2.4.RELEASE
Maven:org.springframework.security:spring-security-web:5.2.4.RELEASE

**2.添加一个测试的HelloController**

计算机生成了可选文字:
@RestContr011er
publicclassHelloContr011er{
@GetMapping("/he110"）
publicStringhe1100{
return"he110"

**3.启动项目**

控制台打印如下一行日志

计算机生成了可选文字:
Usinggeneratedsecuritypassword：69f8215b一a43e一4a7e一b0e9一d64ac5d6cc39

这就是Spring Security为默认用户user生成的临时密码，是一个UUID（*唯一标识码*）字符串

**4.访问localhost:8080/hello**

计算机生成了可选文字:
．儲的ost：808釤℃gin
PleaseSignin
admin
SlgnIn

自动重定向到login登录页面

用户名默认是user，密码即为控制台打印出来的密码，输入就可以访问到/hello接口了。

在Spring Security中，默认的登录页面和登录接口，都是/login，只不过一个是get请求(登录页面)，另一个是post请求(登录接口)。

**源码查看分析：**

1.找到执行该语句的类：UserDetailsServiceAutoConfiguration

计算机生成了可选文字:
2929一95一1111：26：56．914
Usinggeneratedsecurity
password：85f65237一bbf5一4793一ae33一a88849de549b
INFO12696
main]
。S。S。UserDetai1sServiceAutoConfiguration

2.进入UserDetailsServiceAutoConfiguration类，和用户相关的自动化配置信息都在这个类里面，里面有一个getOrDeducePassword方法。

计算机生成了可选文字:
privateStringgetOrDeducePassword(SecurityProperties.Useruser,PasswordEncoderencoder){
Stringpassword
user.getpassword(),
if(user.isPasswordGenerated()){
logger.info(String.format('%n%nlJsinggeneratedsecuritypassword:%s%n"
user.getpassword())),
if(encoder！=null目PASSWORDALGORITHMPATTERN.matcher(password).matches()){
returnpassword

通过该方法可以了解到，如果密码是默认生成的，那么就会执行日志输出，打印用户的密码，接下来进入getPassword方法，查看密码的来源。

计算机生成了可选文字:
publicstaticclassUser{
privateStringname
uSer
privateStringpassword
UUID.randomUUID(）.toString()，
privateList<String>roles
newArrayList<>(),
privatebooleanpasswordGenerated
true;
publicStringgetName(){returnthis.name;}
publicvoidsetName(Stringname){this.name
publicStringgetPassword(){
returnthis.password;
name

**用户配置**

考虑到默认的密码每次重启项目都会变，很不方便，所以先介绍两种非主流的用户名/密码配置方案。

**配置文件**

在application.properties中配置默认的用户名和密码

首先找到SecurityProperties类，默认的用户就定义在它里面，是一个静态内部类User，如果我们要定义自己的用户名密码，就要去覆盖默认配置：



在配置的时候需要以spring.security为前缀

计算机生成了可选文字:
spring·security·user·name=yyj
spring.security·uSer·password=123

以上就是我们新定义的用户名和密码

在properties中定义的用户名和密码最终通过set方式注入到属性中

此时重启项目就可以使用给我们自己定义的用户名和密码进行登录了。

**配置类**

在配置类中配置，就需要指定PasswordEncoder了。

加密方案：

密码加密一般用到散列函数，为了更加安全，一般在密码加密过程中还需要加盐（*所谓的盐可以是一个随机数也可以是用户名，加盐后，即使密码明文相同的用户生成的密码密文也不相同*）

传统的加盐方式需要在数据库中有专门的字段来记录盐值，配置起来比较繁琐，Spring Security提供了多种加密方式，官方推荐使用BCryptPasswordEncoder，不同于Shiro中需要自己处理密码加盐，在Spring Security中，BCryptPasswordEncoder就自带了盐，处理非常方便。

PasswordEncoder：里面就定义了三个方法：

计算机生成了可选文字:
Stringencode(CharSequencerawPassword)，
booleanmatches(CharSequencerawPassword,StringencodedPassword),
defaultbooleanupgradeEncoding(StringencodedPassword){returnfalse;}

* encode：用来对明文密码进行加密，返回加密之后的密文
* matches：密码校对方法，在用户登录的时候，将用户传来的明文密码和数据库中保存的密文密码作为参数，传入到这个方法中，根据返回的boolean来判断用户输入的密码是否正确
* upgradeEncoding：是否还要再进行再次加密，一般不使用

配置：

计算机生成了可选文字:
@Configuration
publicclassSecurityConfigextendsWebSecurityConfigurerAdapter{
@Bean
PasswordEncoderpasswordEncoder(){
returnNe到艹d-E艹e．9s艹e0；
@Override
protectedvoidconfigure(AuthenticationManagerBui1derauth)throwsException{
auth.inMemoryAuthentication()lnMemoryUserDetailsManagerConfigurer<AuthenticationManac
.withUser(username："yyj")UserDetailsManagerConfigurer<AuthenticationManagerB
．passord（"123456"）UserDetailsManagerConfigurer<AuthenticationManagerBuilder,丨n
·roles（"admin"），

首先我们自定义SecurityConfig继承自WebSecurityConfigurerAdapter，重写里面的configure方法

这里我们提供了一个PasswordEncoder的实例，暂时先不给密码进行加密

在configure方法中，通过inMemoryAuthentication来开启在内存中定义用户，withUser中是用户名，password是密码，roles是用户角色，如果需要配置多个用户，用and相连。

配置完成后，再次启动项目，Java配置类中的配置会覆盖掉配置文件中的配置

**自定义表单登录页**

**服务端定义**

完善前面的SecurityConfig类，继续重写它的Configure(WebSecurity web)和configure(HttpSecurity http)方法，如下：



计算机生成了可选文字:
@Override
publicvoidconfigure(WebSecurityweb)throws身圣月9卩{
*房来配忽亏RLi芏更是对旁，忝之
web.ignoring().antMatchers(...antPatterns:"/js/**""/css/**"
"/images/**"

然后从别的地方拷贝一个login.html文件，form表单下的action标签改为/login.html，就可以了

**定制Spring Security的表单登录**

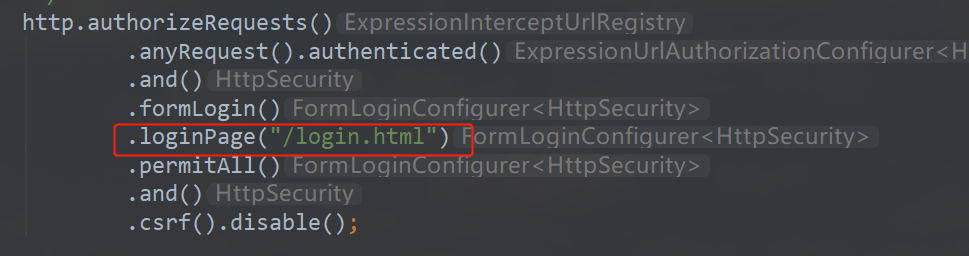
**1.登录接口**

* 登录页面就是你看到的浏览器展示出来的登录页面样子
* 登录接口则是提交登录数据的地方，就是登录页面里面的form表单的action属性对应的值

在Spring Security中，如果我们不做任何配置，默认的登录页面和登录接口的地址都是/login，也就是说，默认会存在如下两个请求：

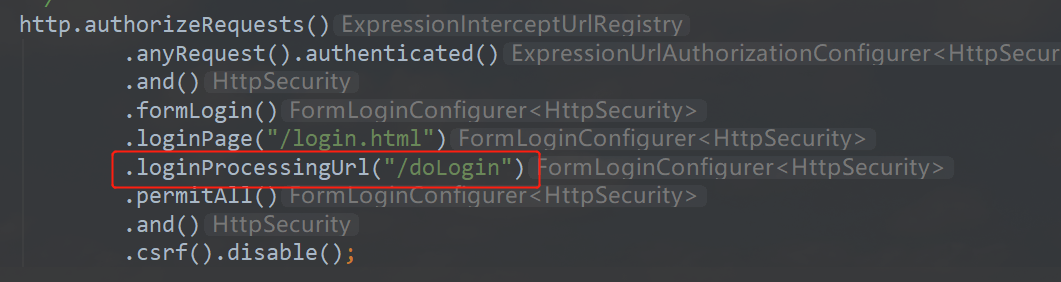
* GET <http://localhost:8080/login.html>
* POST <http://localhost:8080/login.html>

如果是GET请求表示你想访问登录页面，如果是POST请求，表示你想提交登录数据



当我们在配置类中配置了loginPage为/login.html之后，除了设置登录页面的地址为/login.html外，也把登录接口地址也设置为了/login.html。

在SecurityConfig中，通过loginProcessingUrl方法来指定登录接口地址，如下：



这样就实现了登录页面地址和登录接口地址分开，此时还需要再修改登录页面里面的action属性，改为/doLogin



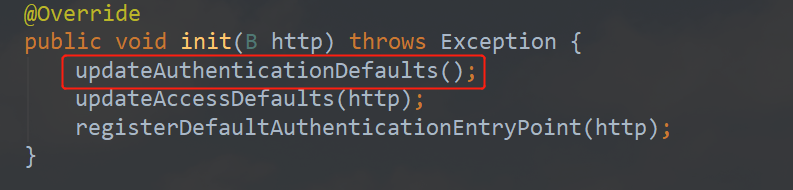
探究：为什么默认情况下两个配置地址是一样的呢？

找到form表单的相关配置类：FormLoginConfigurer，该类继承自AbstractAuthenticationFilterConfigurer，查看该父类的构造方法：

计算机生成了可选文字:
protectedAbstractAuthenticationFi1terConfigurer(){
setLoginPage("/10gn"），

默认配置的loginPage就是/login

另一方面，FormLoginConfigurer的初始化方法init方法中也调用了父类的init方法：



而在父类的 init 方法中，又调用了 updateAuthenticationDefaults，我们来看下这个方法：

计算机生成了可选文字:
protectedfinalvoidupdateAuthenticationDefau1ts(){
if(loginProcessingUrl
null){
loginProcessingUr1(loginPage)，

因此我们可以知道，如果用户没有给loginProcessingUrl设置值的话，默认就会使用loginPage作为loginProcessingUrl

**2.登录参数**

在login.html登录表单中的参数是username和password，默认情况下是不允许变的。

计算机生成了可选文字:
<formaction="/doLogin"method="post">
<divclass="input">
<labelfor="name">用户名</label>
<inputtype="text"ame="username'
<spanclass="spin"></span>
</div>
<divclass='input">
<labelfor="pass">密码</label>
id=
name">
id=
<inputtype=password"name='password"
<spanclass="spin"></span>
</div>

原因：回到FormLoginConfigurer类，在它的构造方法中，我们可以看到有两个配置用户名和密码的方法：

计算机生成了可选文字:
publicFormLoginConfigurer(){
super(newUsernamePasswordAuthenticationFi1ter(),
passwordParameter(""),
defaultLoginProcessingUrl:
null);

首先，调用了父类的构造方法，传入了UsernamePasswordAuthenticationFilter实例，该实例将被赋值给父类的authFilter属性。

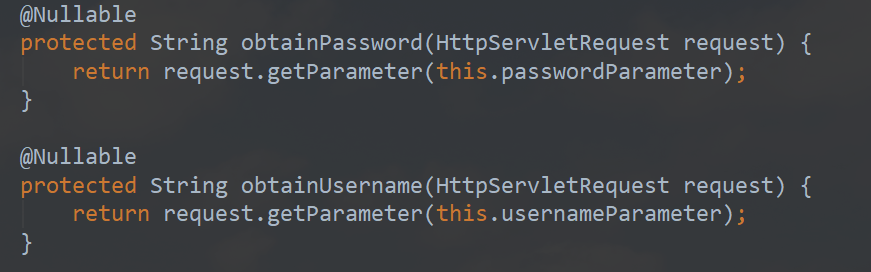
接下来查看usernameParameter方法如下：

计算机生成了可选文字:
publicFormLoginConfigurer<H>usernameParameter(StringusernameParameter){
getAuthenticationFi1ter(）·setUsernameParameter(usernameParameter)，
returnthis;

getAuthenticationFilter 实际上是父类的方法，在这个方法中返回了 authFilter 属性，也就是一开始设置的 UsernamePasswordAuthenticationFilter 实例，然后调用该实例的 setUsernameParameter 方法去设置登录用户名的参数：

计算机生成了可选文字:
publicvoidsetUsernameParameter(StringusernameParameter){
Assert.hasText(usernameparameter,message:"Usernameparametermustnotbeempty0rnu11');
this·

这里的设置有什么用呢？当登录请求从浏览器来到服务器端以后，我们要从请求的HttpServletRequest中取出用户登录的用户名和密码，如何取呢？还是在UsernamePasswordAuthenticationFilter类中，有两个方法：



可以看到，这个时候就用到默认配置的username和password了。

如果说我们想要在SecurityConfig类中配置usernameParameter和passwordParameter的值，那么在前端页面也要修改input的name属性值，保持和服务端一致。

**3.登录回调**

前后端不分离的登录回调

**3.1登录成功回调**

在Spring Security中，和登录成功重定向URL相关的方法有两个：

* defaultSuccessUrl
* successForwardUrl

首先我们在配置的时候，defaultSuccessUrl和successForwardUrl只需要配置一个即可，区别如下：

1.defaultSuccessUrl有一个重载的方法，先说只有一个参数的defaultSuccessUrl方法：如果我们在defaultSuccessUrl中指定登陆成功的跳转页面为/index，此时分两种情况，如果你是直接在浏览器中输入的登录地址，登录成后，就直接跳转到/index，如果你是在浏览器中输入了其他地址，例如：localhost:8080/hello，结果因为没有登录，又重定向到登录页面，此时登录成功后，就不会来到/index，而是来到/hello页面。

2.defaultSuccessUrl还有一个重载的方法，第二个参数如果不设置，默认为false，同上，如果设置为true，则效果和successForwardUrl一致。

3.successForwardUrl表示不管你是从哪里来的，登录后一律跳转到successForwardUrl指定的地址。例如 successForwardUrl 指定的地址为 /index ，你在浏览器地址栏输入http://localhost:8080/hello，结果因为没有登录，重定向到登录页面，当你登录成功之后，就会服务端跳转到 /index 页面；或者你直接就在浏览器输入了登录页面地址，登录成功后也是来到 /index。



3.2登录失败回调

* failureForwardUrl
* failureUrl

这两个方法再设置的时候也是设置一个即可,failureForwardUrl是登录失败之后会发生服务端跳转,failureUrl则是在登陆失败以后,会发生重定向.

**4.注销登录**

注销登录的默认接口是/logout，

1.默认注销的URL是/logout，是一个GET请求，我们可以通过logoutUrl方法来修改默认的注销URL

2.logoutRequestMatcher方法不仅可以修改注销URL，还可以修改请求方式，实际项目中，这个方法和logoutUrl任意设置一个即可。

3.logoutSuccessUrl表示注销成功后要跳转的页面

4.deleteCookies用来清除cookie

5.clearAuthentication和invalidateHttpSession分别表示清除认证信息和使HttpSession失效，默认可以不用配置，默认就会清除。

**Spring Security做前后端分离，JSON交互取代页面跳转**

**1.无状态登录**

**1.1什么是有状态**

有状态服务，即服务端需要记录每次会话的客户端信息，从而识别客户端身份，根据用户身份进行请求的处理，典型的设计如Tomcat的Session，例如登录：用户登录以后，我们把用户的信息保存在服务端session中，并且给用户一个cookie值，记录对应的session，然后下次请求，用户携带cookie值来，我们就能识别到对应session，从而找到用户的信息。

缺陷：

* 服务端保存大量数据，增加服务器压力；
* 服务端保存用户状态，不支持集群化部署

**1.2什么是无状态**

微服务集群中的每个服务，对外提供的都使用RESTful风格的接口，而RESTful风格的一个最重要的规范就是：服务的无状态性，即：

* 服务端不保存任何客户端请求者信息
* 客户端的每次请求必须具备自描述信息，通过这些信息识别客户端身份

无状态性的好处：

* 客户端请求不依赖服务端你的信息，多次请求不需要必须访问到同一台服务器
* 服务端的集群和状态对客户端透明
* 服务端可以任意的迁移和伸缩（方便进行集群化部署）
* 减小服务端存储压力

**1.3如何实现无状态**

* 首先客户端发送用户名和密码到服务端进行认证
* 认证通过后，服务端将用户信息加密并且编码成一个token，返回给客户端
* 以后客户端每次发送请求，都需要携带认证的token
* 服务端对客户端发送过来的token进行解密，判断是否有效，并且获取用户登录信息

**1.4各自的优缺点**

* 使用session最大的优点在于方便，不用做过多的处理，缺点是如果前端是Android，iOS，小程序等，这些App没有天然的cookie，所以需要适配session。
* 而像JWT这种无状态登录就显现出优势了，这些登录方式所依赖的token可以通过普通参数传递，也可以通过请求头传递，具有很强的灵活性。
* 如果前后端分离只是网页+服务端，建议用session

**2.登录交互**

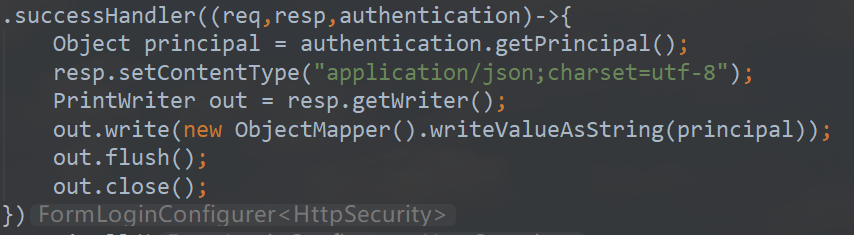
**2.1前后端分离的数据交互**

在前后端分离的开发架构下，前后端的交互都是通过JSON来进行，无论登录成功还是失败，都不会有什么服务端跳转或者客户端跳转之类的事情，服务端总是会返回一段JSON数据给前端，有前端自行决定。

**2.2登录成功**

之前配置登录成功的处理是通过defaultSuccessUrl和successForwardUrl这两个方法来配置的，其原理都是配置跳转地址，适用于前后端不分的开发，如果是前后端分离的开发，就要用到successHandler了。

Successhandlerd功能十分强大，完全包含前面两种方式的功能：



successHandler方法的参数是一个AuthenticationSuccessHandler对象，这个对象中我们要实现的方法是onAuthenticationSuccess。

onAuthenticationSuccess方法有三个参数，分别是：

* HttpServletRequest
* HttpServletResponse
* Authentication

有了前面两个参数，我们就可以在这里随心所欲地返回数据了，利用HttpServletRequest我们可以做服务端跳转，利用HttpServletResponse我们可以做客户端跳转，当然还可以返回JSON数据。

第三个参数Authentication则保存了我们刚刚登录成功的用户信息。

配置完成后，我们再去登录，就可以看到登录成功的用户信息通过JSON返回到前端了，如下；



**2.3登录失败**



失败的回调也是三个参数，第三个是一个Exception，用于保存登录失败的原因，我们可以将之通过JSON返回到前端。



这里我们是直接打印错误信息，其实也可以自己写一个异常类，以返回更加清晰的错误提示，类似这样：

计算机生成了可选文字:
resp。setContentType("application/json；Charset=utf—8")，
p「intW「out=resp.getWrite「（）；
Resp8ean「esp3ean=Resp8ean．error(e.getkssage(））；
if(einstanceofLockedException){
「esean．set凹俨账户裰定，清联系苣理员.");
}已15ef(einstanceofCredentialsExpiredException){
「esean．set凹俨密码逦，清联系苣理员.");
}已15ef(einstanceofAccountExpiredException){
「esean．set凹俨账户逦，清联系苣理员.");
}已15ef(einstanceofDisabledException){
「esean．set凹俨账户裰用，清联系苣理员.");
}已15ef(einstanceofBadCredentiaIsException){
「esean．set凹俨用户名或者密码输入误，清重新输入！");
out.writeValueAsString(resp8ean))，
out到flush()；
out．[1。5e0；

用户名错误也会提示用户名或密码错误是为了增加安全性，其内部实现在AbstractUserDetailsAuthenticationProvider类中。

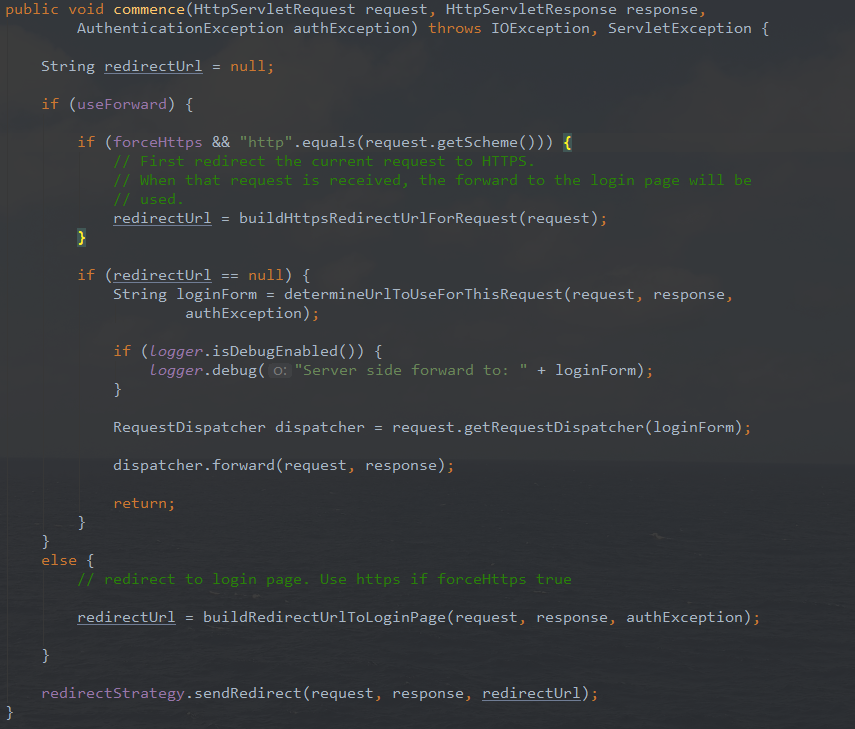
计算机生成了可选文字:
uSer
retrieveUser(username,
(UsernamePasswordAuthenticationToken)authentication),
catch(UsernameNotFoundExceptionnotFound){
logger.debug(O："User
notfound");
+uSername+
if(hideUserNotFoundExceptions){
thrownewBadCredentia1sException(messages.getMessage(
code:"AbstractUserDetai1sAuthenticationProvider.badCredentia1s"
defaultMessage:"Badcredentials")),
else{
thrownotFound,

**3.未认证处理方案**

系统默认是直接重定向到登录页面

但是在前后端分离中，这个逻辑明显是有问题的，如果用户没有登录就访问一个需要认证后才能访问的页面，这个时候我们不应该让用户重定向到登录页面，而是应该给用户一个尚未登陆的提示，前端收到提示之后，再自行决定页面跳转。

要解决这个问题，就涉及到Spring Security中的一个接口，AuthenticationEntryPoint，该接口有一个实现类：LoginUrlAuthenticationEntryPoint，该类中有一个方法commerce，如下：



这个方法是用来决定到底是要重定向还是要forward，通过Debug追踪，默认情况下useForward的值是false，所以请求走进了重定向。

如何解决呢？

直接重写这个方法，在方法中返回JSON即可，具体配置如下：



在Spring Security的配置中加上自定义的AuthenticationEntryPoint处理方法，该方法中直接返回相应的JSON提示即可，这样，如果用户再去直接访问一个需要认证之后才可以访问的请求，就不会发生重定向操作了，服务端会直接给浏览器一个JSON提示，浏览器收到JSON提示之后，该干嘛就干嘛。

**4.注销登录**

在前后端分离的系统中，注销登录成功后也不应该自动跳转到登录页面，而是应该直接返回JSON即可，有客户端自行决定，配置如下：

**Spring Security中的授权操作**

**1.授权**

所谓的授权，就是用户如果要访问某一个资源，我们要去检查用户是否具备这样的权限，如果具备就访问，如果不具备，则不允许访问。

**2.准备测试用户**

测试用户基于内存来配置，有两种方式，第一种是前面提到的配置方式：

计算机生成了可选文字:
auth.inMemoryAuthentication()lnMemoryUserDetailsMa
.withUser(username："yyj"）UserDetailsManag
.password("123456"）UserDetailsManagerConfig
.roles("admin")UserDetailsManagerConfigureB
.and()InMemoryUserDetailsManagerConfigurer<
.withUser(username："田丿]'）UserDetailsMana(
.password("123"）UserDetailsManagerConfigureI
·roles("user"),

还有一种方式，由于Spring Security支持多种数据源，例如：内存、数据库、LDAP等，可以将这些不同来源的数据封装成一个USerDetailService接口，任何实现了该接口的对象都可以作为认证数据源。

因此我们还可以通过重写WebSecurityConfigurerAdapter中的userDetailsService方法来提供一个UserDetailService实例，进而配置多个用户。

（貌似木有起作用）

计算机生成了可选文字:
@Override
@Bean
protectedUserDetai1sServiceuserDetai1sService(){
InMemoryUserDetai1sManagermanager
newInMemoryUserDetai1sManager(），
manager·createUser(User·withUsername("yyj"）·password("123456"）·roles〈"admin'）·build））；
manager·createUser(User·withUsername("田力"）·password("123"）·roles〈"user"）·build(）），
manager·createUser(User·withUsername("杨玉杰"）·password("12）·roles（"admin"）·build(））；
manager;

**3.准备测试接口**

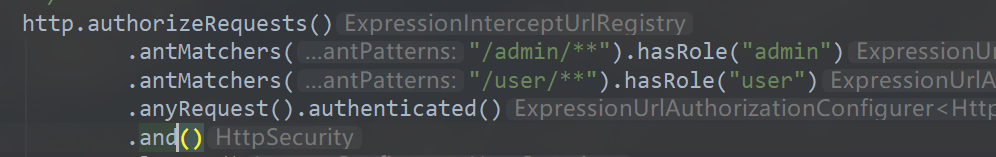
计算机生成了可选文字:
@RestContr011er
publicclassHelloContr011er{
@GetMapping("/he110"）
publicStringhe1100{
return"he110"
@GetMapping("/admin/hello"）
publicStringadmin(){
return
@GetMapping("/user/hellcl"）
publicStringuser(){
return

这三个测试接口分别如下：

* /hello是任何人都可以访问的接口
* /admin/hello是具有admin身份的人才能访问的接口
* /user/hello是具有user身份的人才能访问的接口
* 所有user能够访问的资源，admin都能访问

**4.配置**

接下来我们来配置权限的拦截规则，在Spring Security的configure(HttpSercurity http)方法中，代码如下：



注意顺序别写错了

这里的匹配规则采用了Ant风格的路径匹配符，规则如下：

计算机生成了可选文字:
通配符
含义
匹配多层路径
匹配一层路径
匹配任意望个字符

上面配置的含义是：

1. 如果请求路径满足/admin/\*\*格式，则用户需要具备admin角色
2. 如果请求路径满足/user/\*\*格式，则用户需要具备user角色
3. 剩余的其他格式的请求路径，只需要认证后就可以访问

和Shiro类似，Spring Security在匹配的时候也是按照从上往下的顺序来匹配的，一旦匹配到了就不继续匹配了，所以拦截规则的顺序不能写错。

从语义上理解，anyRequest应该放在最后，表示除了前面拦截规则之外，剩下的请求要如何处理。

**5.启动测试**

首先通过yyj的身份进行登录，登录成功后分别访问：/hello，/admin/hello 以及 /user/hello 三个接口，发现都是可以访问到的，

如果用田力的身份进行登录，则无法访问到/admin/hello下的资源，提示403错误。

**6.角色继承**

为了实现所有user能够访问到的资源，admin都能访问，这涉及到另一个知识点，叫做角色继承。

计算机生成了可选文字:
*实现痦色孬
@Bean
RoleHierarchyroleHierarchy(){
RoleHierarchyImp1hierarchy=newRoleHierarchyImp1(),
hierarchy.setHierarchy("ROLEadm主n〔>]ROLEuser"),
returnhierarchy,

在配置时，需要给角色手动加上 ROLE\_ 前缀。上面的配置表示 ROLE\_admin 自动具备 ROLE\_user 的权限，如果是有多个角色继承关系，目前写法如下：

计算机生成了可选文字:
@Bean
RoleHierarchyroleHierarchy(){
RoleHierarchyImp1roleHierarchy=newRoleHierarchyImp1(）
Stringhierarchy
"ROLEdba>ROLEadmi\n
LEadmin>ROLEuser"
roleHierarchy·setHierarchy(hierarchy),
returnroleHierarchy,

**Spring Security将用户数据存入数据库**

真正的项目中，大部分情况下，都需要我们自己设计权限数据库，不过Spring Security也给我们提供了一个它自己设计好的权限数据库。

**1.UserDetailService**

Spring Security支持多种不同的数据源，这些不同的数据源最终都将被封装成UserDetailsService的实例，我们除了可以自己来创建一个类来实现UserDetailsService接口外，也可以使用系统默认提供的UserDetailsService实例，例如之前介绍的InMemoryUserDetailsManager。

先来看看UserDetailsService都有哪些实现类：

计算机生成了可选文字:
0JdbcDaolmpl
。UserDetailsServiceDelegator
UserDetailsService
UserDetailsManager
A
0eReactiveUserDetailsServiceAdapter
0CachingUserDetailsService
0JdbcUserDetailsManager
0lnMemoryUserDetailsManager

JdbcUserDetailsManager可以让我们通过JDBC的方式将数据库和Spring Security连接起来。

**2.JdbcUserDetailsManager**

JdbcUserDetailsManager自己提供了一个数据库模型，这个数据库模型保存在如下位置：

org/springframework/security/core/userdetails/jdbc/users.ddl

存储的脚本内容如下：

计算机生成了可选文字:
createtableusers(usernamevarchar_ignorecase(50)notnullprimarykey,passwordvarchar_ignorecase(500)notnull，
createtableauthorities(usernamevarcharignorecase(50)notnull,authority、、varchar_ignorecase(50)notnull,const
createuniqueindexixauthuSernameonauthorities(username,authority)]

脚本中有一种数据类型varchar\_ignorecase，把它替换成varchar，然后将sql代码拷贝到数据库中生成表users和authorities。

users表中保存了用户的用户名、密码以及账户是否可用

计算机生成了可选文字:
username
password
enabled

authorities中保存了用户的角色



两个表通过username关联起来

接下来将之前用到的通过InmemoryUserDetailsManager提供的用户数据用JdbcUserDetailsManager代替掉，如下：

计算机生成了可选文字:
@Autowired
DataSourcedataSource;
@*erride
@Bean
protectedUserDetai1sServiceuserDetai1sService(){
工nMemor、ylJser、Detailsrvlanager、manager
new工nrvlemor、ylJser、Detailsrvlanager、0；
JdbcUserDetai1sManagermanager
newJdbcUserDetai1sManager(dataSource),
if(!manager.userExists(username："yyj"){
manager·createUser(User.withUsername((Y)'j'）·password("12345引）·roles("admin'）·build())；
if(!manager.userExists(username："田力'月{
manager..withUsername("田力'）·password("123'）·roles("user'）·build());
if(!manager.userExists(username："杨玉杰'））{
manager..withUsername("杨玉杰'）·password("123'）·roles("admin'）·build());
manager;

此时没有数据源，因此引入数据库依赖以及mysql驱动

计算机生成了可选文字:
<dependency>
<groupld>mysql</groupld>
<artifactld>mysql-connector-java</artifactld>
<version>8.e.19<1version>
</dependency>
<dependency>
<groupld.org·springframework·boot</groupld>
<artifactld>spring-boot-starter-jdbc</artifactld>
</dependency>

接下来在properties配置文件中配置数据库信息

计算机生成了可选文字:
spring·datasource·url=jdbc：mysql：///vhr?useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai
spring·datasource·username=root
spring·datasource·password=root
spring·datasource·driver-class。mysql·cj·jdbc·Driver

配置完成后，启动项目，先查看数据库中是否成功将数据添加到数据库了

计算机生成了可选文字:
username
杨玉杰
田力
ROIFadmin
ROLE—admin
ROLEuser

计算机生成了可选文字:
username
杨玉杰
田力
password
1234％
123
123
enabled
1
1
1

Ok

**4.测试**

首先我们以田力的身份进行登录：

1. /hello 因为登录后就可以访问，这个接口访问成功。
2. /admin/hello 需要 admin 身份，所以访问失败。
3. /user/hello 需要 user 身份，所以访问成功。

然后我们以yyj的身份进行登录：能正常访问到/user/hello