目录

[1.项目结构 2](#_Toc44944932)

[2.Restful 12](#_Toc44944933)

[2.1.区别 12](#_Toc44944934)

[2.2.JsonPath说明 14](#_Toc44944935)

[2.3.JsonView使用 14](#_Toc44944936)

[2.3.1使用接口来声明多个视图 14](#_Toc44944937)

[2.3.2在值对象的get方法上指定视图 14](#_Toc44944938)

[2.3.3在controller方法上指定视图 14](#_Toc44944939)

[2.4.@Valid 参数校验, 15](#_Toc44944940)

[2.4.1校验 15](#_Toc44944941)

[2.4.2.自定义校验 16](#_Toc44944942)

[2.5.Restful Api拦截 18](#_Toc44944943)

[2.5.1过滤器 18](#_Toc44944944)

[2.5.2.使用拦截器 19](#_Toc44944945)

[2.5.3.使用切面 22](#_Toc44944946)

[2.6.上传下载 23](#_Toc44944947)

[2.7.异步处理Rest服务 25](#_Toc44944948)

[2.7.1使用Runnable异步处理 25](#_Toc44944949)

[2.7.2使用DeferredResult异步处理 25](#_Toc44944950)

[2.8使用swagger生产文档 28](#_Toc44944951)

[2.9.wiremock 29](#_Toc44944952)

[3. Spring Security认证 30](#_Toc44944953)

[3.1说明 30](#_Toc44944954)

[3.2.自定义用户逻辑认证 32](#_Toc44944955)

[3.2.1处理用户信息获取逻辑 32](#_Toc44944956)

[3.2.2.个性化用户认证流程 33](#_Toc44944957)

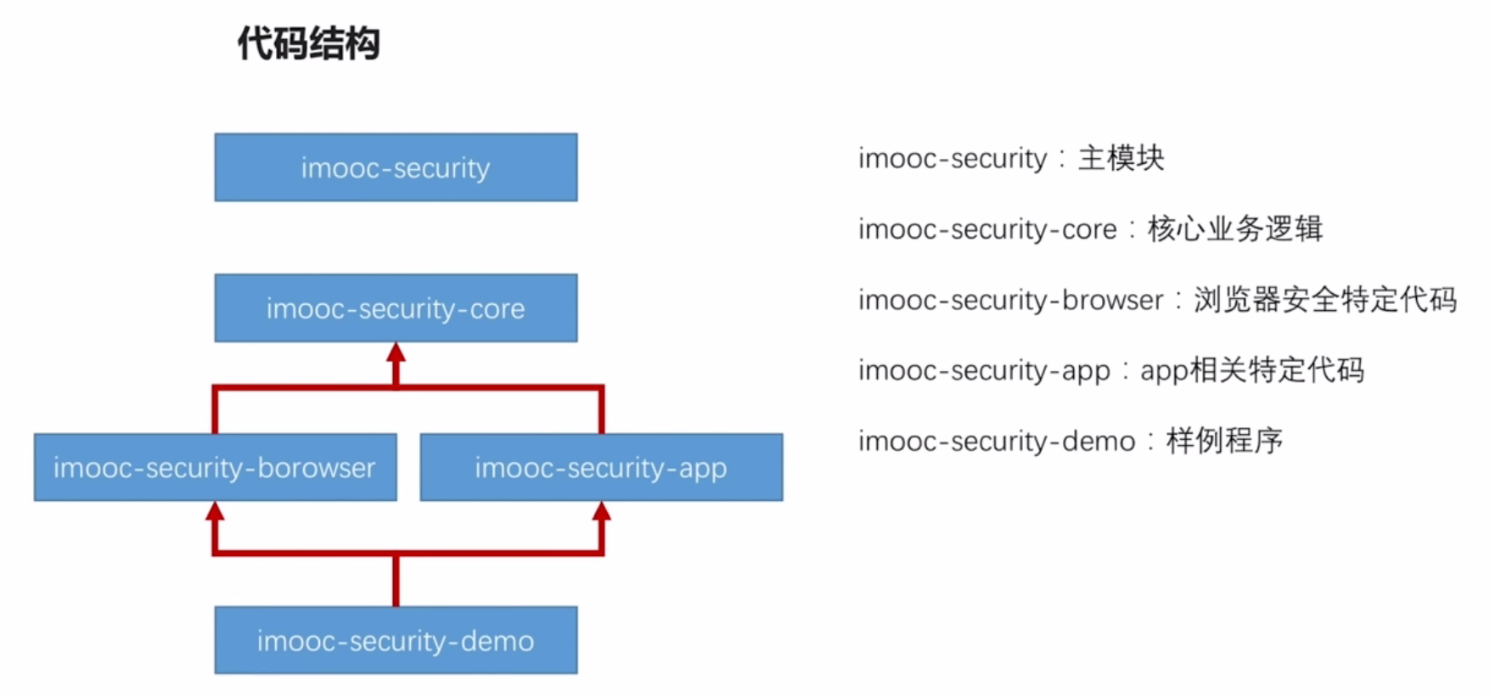
[3.3图片验证码 41](#_Toc44944958)

[3.3.1实现图片验证码 41](#_Toc44944959)

[3.3.2.验证码校验 43](#_Toc44944960)

[3.3.3.图形验证码重构 45](#_Toc44944961)

# 1.项目结构



com.swb.security

imooc-security:没有任何java代码,作用：统一下面的四个模块，对四个子模块统一执行maven命令,打包等

imooc-security-core:登录实现逻辑是在这里实现的

imooc-security-browser, imooc-security-app：封装浏览器和app 安全相关的代码，是安全模块提供给其他模块项目使用。

imooc-security-demo是演示其他项目如何使用演示。

1. 创建一个空项目security

maven分别创建5个模块

imooc-security 导入io.spring.platform，spring-cloud-dependencies

|  |
| --- |
| <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.swb.security</groupId>  <artifactId>imooc-security</artifactId>  <version>1.0.0</version>  <packaging>pom</packaging>  <properties>  <imooc.security.version>1.0.0</imooc.security.version>  </properties>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>io.spring.platform</groupId>  <artifactId>platform-bom</artifactId>  <version>Cairo-SR7</version>  <type>pom</type>  <scope>import</scope>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Greenwich.SR5</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>2.3.2</version>  <configuration>  <source>1.8</source>  <target>1.8</target>  <encoding>UTF-8</encoding>  </configuration>  </plugin>  </plugins>  </build>  <modules>  <module>../imooc-security-app</module>  <module>../imooc-security-browser</module>  <module>../imooc-security-core</module>  <module>../imooc-security-demo</module>  </modules>  </project> |

imooc-security-core

|  |
| --- |
| <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <artifactId>imooc-security-core</artifactId>  <parent>  <groupId>com.swb.security</groupId>  <artifactId>imooc-security</artifactId>  <version>1.0.0</version>  <relativePath>../imooc-security</relativePath>  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-oauth2</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-redis</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jdbc</artifactId>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>8.0.15</version>  </dependency>  <dependency>  <groupId>org.springframework.social</groupId>  <artifactId>spring-social-config</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.social</groupId>  <artifactId>spring-social-core</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.social</groupId>  <artifactId>spring-social-security</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.social</groupId>  <artifactId>spring-social-web</artifactId>  </dependency>  <dependency>  <groupId>commons-lang</groupId>  <artifactId>commons-lang</artifactId>  </dependency>  <dependency>  <groupId>commons-collections</groupId>  <artifactId>commons-collections</artifactId>  </dependency>  <dependency>  <groupId>commons-beanutils</groupId>  <artifactId>commons-beanutils</artifactId>  </dependency>  </dependencies>  </project> |

imooc-security-browser,

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <artifactId>imooc-security-browser</artifactId>  <parent>  <groupId>com.swb.security</groupId>  <artifactId>imooc-security</artifactId>  <version>1.0.0</version>  <relativePath>../imooc-security</relativePath>  </parent>  <dependencies>  <dependency>  <groupId>com.swb.security</groupId>  <artifactId>imooc-security-core</artifactId>  <version>${imooc.security.version}</version>  </dependency>  <dependency>  <groupId>org.springframework.session</groupId>  <artifactId>spring-session</artifactId>  </dependency>  </dependencies>  </project> |

imooc-security-app

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <artifactId>imooc-security-app</artifactId>  <parent>  <groupId>com.swb.security</groupId>  <artifactId>imooc-security</artifactId>  <version>1.0.0</version>  <relativePath>../imooc-security</relativePath>  </parent>  <dependencies>  <dependency>  <groupId>com.swb.security</groupId>  <artifactId>imooc-security-core</artifactId>  <version>${imooc.security.version}</version>  </dependency>  </dependencies>  </project> |

imooc-security-demo

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <artifactId>imooc-security-demo</artifactId>  <parent>  <groupId>com.swb.security</groupId>  <artifactId>imooc-security</artifactId>  <version>1.0.0</version>  <relativePath>../imooc-security</relativePath>  </parent>  <dependencies>  <dependency>  <groupId>com.swb.security</groupId>  <artifactId>imooc-security-browser</artifactId>  <version>${imooc.security.version}</version>  </dependency>  </dependencies>  </project> |

以上为项目的主要结构，maven管理项目，视频是使用STL与IDEA有差异

1. Gradle创建项目

项目实际使用gradle管理

建立一个Spring Boot项目SecurityGradle

本地路径: D:\MY\IDEA\securityGradle

Github地址:

Gradle版本为4.10.2 ，使用当时最新的6.2.2构建失败故用原先电脑的版本

group = 'com.swb.security' name='security-gradle' version='1.0.0'

空项目，什么都没有

创建四个子模块

右键项目,选择新建模块，选择gradle项目,创建security-core,app,browser,demo

修改父模块的build.gradle文件，加入

|  |
| --- |
| group 'com.swb.security'  version '1.0.0'  allprojects {  gradle.projectsEvaluated {  tasks.withType(JavaCompile) {  options.compilerArgs << "-Xlint:unchecked" << "-Xlint:deprecation"  }  }  }  buildscript {  repositories {  maven { url "http://maven.aliyun.com/nexus/content/groups/public" }  mavenCentral()  }  dependencies {  classpath 'org.springframework.boot:spring-boot-gradle-plugin:2.2.5.RELEASE'  }  }  subprojects {  apply plugin: 'java'  sourceCompatibility = 1.8  targetCompatibility = 1.8  [compileJava, compileTestJava]\*.options\*.encoding = 'UTF-8'  repositories {  mavenCentral()  }  dependencies {  }  } |

Core模块

|  |
| --- |
| buildscript {  repositories {  maven { url "http://maven.aliyun.com/nexus/content/groups/public" }  mavenCentral()  }  dependencies {  classpath 'org.springframework.boot:spring-boot-gradle-plugin:2.2.5.RELEASE'  }  }  apply plugin: 'java'  apply plugin: 'org.springframework.boot'  group 'com.swb.security'  version '1.0.0'  sourceCompatibility = 1.8  ext {  springBootVersion = "2.2.5.RELEASE"  cloudVersion = "2.2.1.RELEASE"  springBootMybatisVersion = "2.1.2"  mysqlConnector = "8.0.19"  springSocial = "1.1.6.RELEASE"  junitVersion = '4.13'  pagehelperVersion = '1.2.10'  mapperVersion = '2.1.5'  lombokVersion = '1.18.12'  compileJava.options.encoding = 'UTF-8'  compileTestJava.options.encoding = 'UTF-8'  }  dependencies {  compile "org.springframework.boot:spring-boot-starter:${springBootVersion}"  compile "org.springframework.boot:spring-boot-starter-data-redis:${springBootVersion}"  compile "org.springframework.boot:spring-boot-starter-web:${springBootVersion}"  compile "org.mybatis.spring.boot:mybatis-spring-boot-starter:${springBootMybatisVersion}"  compile "org.springframework.cloud:spring-cloud-starter-oauth2:${cloudVersion}"  compile "org.springframework.boot:spring-boot-starter-test:${springBootVersion}"  compile "mysql:mysql-connector-java:${mysqlConnector}"  compile "org.springframework.cloud:spring-cloud-starter:${cloudVersion}"  compile "org.springframework.social:spring-social-core:${springSocial}"  compile "org.springframework.social:spring-social-config:${springSocial}"  compile "org.springframework.social:spring-social-security:${springSocial}"  compile "org.springframework.social:spring-social-web:${springSocial}"  compile "com.github.pagehelper:pagehelper-spring-boot-starter:${pagehelperVersion}"  compile "tk.mybatis:mapper-spring-boot-starter:${mapperVersion}"  compile "org.projectlombok:lombok:${lombokVersion}"  annotationProcessor "org.projectlombok:lombok:1.18.12"//gradle打包是lombok生效  compile group: 'commons-lang', name: 'commons-lang', version: '2.6'  compile group: 'commons-collections', name: 'commons-collections', version: '3.2.2'  compile group: 'commons-beanutils', name: 'commons-beanutils', version: '1.9.4'  testCompile group: 'junit', name: 'junit', version: '4.12'  } |

app依赖core项目

|  |
| --- |
| buildscript {  repositories {  maven { url "http://maven.aliyun.com/nexus/content/groups/public" }  mavenCentral()  }  dependencies {  classpath 'org.springframework.boot:spring-boot-gradle-plugin:2.2.5.RELEASE'  }  }  apply plugin: 'java'  apply plugin: 'org.springframework.boot'  group 'com.swb.security'  version '1.0.0'  sourceCompatibility = 1.8  repositories {  mavenCentral()  }  dependencies {  implementation project(":security-core")  annotationProcessor "org.projectlombok:lombok:1.18.12"//gradle打包是lombok生效  testCompile group: 'junit', name: 'junit', version: '4.12'  } |

browser依赖core同时加入Spring-Session

|  |
| --- |
| buildscript {  repositories {  maven { url "http://maven.aliyun.com/nexus/content/groups/public" }  mavenCentral()  }  dependencies {  classpath 'org.springframework.boot:spring-boot-gradle-plugin:2.2.5.RELEASE'  }  }  apply plugin: 'java'  apply plugin: 'org.springframework.boot'  group 'com.swb.security'  version '1.0.0'  sourceCompatibility = 1.8  repositories {  mavenCentral()  }  dependencies {  compile project(":security-core")  compile "org.springframework.session:spring-session:1.3.5.RELEASE"  annotationProcessor "org.projectlombok:lombok:1.18.12"//gradle打包是lombok生效  testCompile group: 'junit', name: 'junit', version: '4.12'  } |

demo依赖browser

|  |
| --- |
| group 'com.swb.security'  version '1.0.0'  sourceCompatibility = 1.8  buildscript {  repositories {  maven { url "http://maven.aliyun.com/nexus/content/groups/public" }  mavenCentral()  }  dependencies {  classpath 'org.springframework.boot:spring-boot-gradle-plugin:2.2.5.RELEASE'  }  }  apply plugin: 'java'  apply plugin: 'org.springframework.boot'  dependencies {  compile project(":security-browser")  annotationProcessor "org.projectlombok:lombok:1.18.12"//gradle打包是lombok生效  testCompile group: 'junit', name: 'junit', version: '4.12'  } |

至此项目构建完成

测试Demo

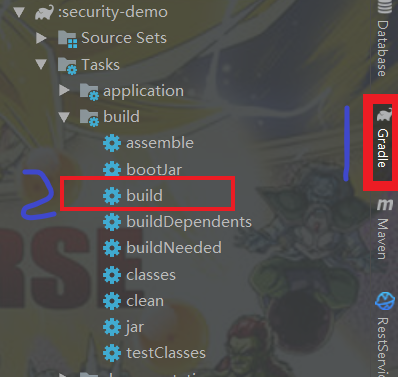
创建security-demo\src\main\java\com\swb\security\demo\DemoApplication.java启动类

|  |
| --- |
| @SpringBootApplication(exclude = {SecurityAutoConfiguration.class,  ManagementWebSecurityAutoConfiguration.class})  //关闭security校验  @RestController  public class DemoApplication {  public static void main(String[] args) {  SpringApplication.run(DemoApplication.class,args);  }  @GetMapping("/hello")  public String getHello(){  return "Hello Spring Security";  }  } |

application.yml

|  |
| --- |
| spring:  datasource:  url: jdbc: mysql://127.0.0.1:3306/SecurityDemo?serverTimezone=Shanghai&useUnicode= yes&characterEncoding=UTF-8&useSSL=false  username:  password:  driver-class-name: com.mysql.cj.jdbc.Driver  session:  store-type: none  server:  port: 8080  mapper:  #公用接口类路径,不能与  mappers: com.swb.security.demo.dao.IBaseDao  identity: MYSQL  pagehelper:  helper-dialect: mysql  reasonable: true  support-methods-arguments: true  params: count=countSql  logging:  level:  com:  weige:  springboot:  mapper: debug |

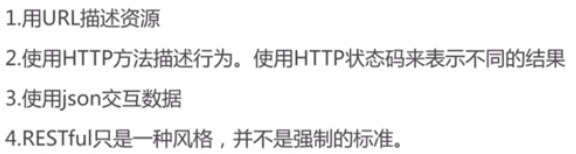
构建jar包



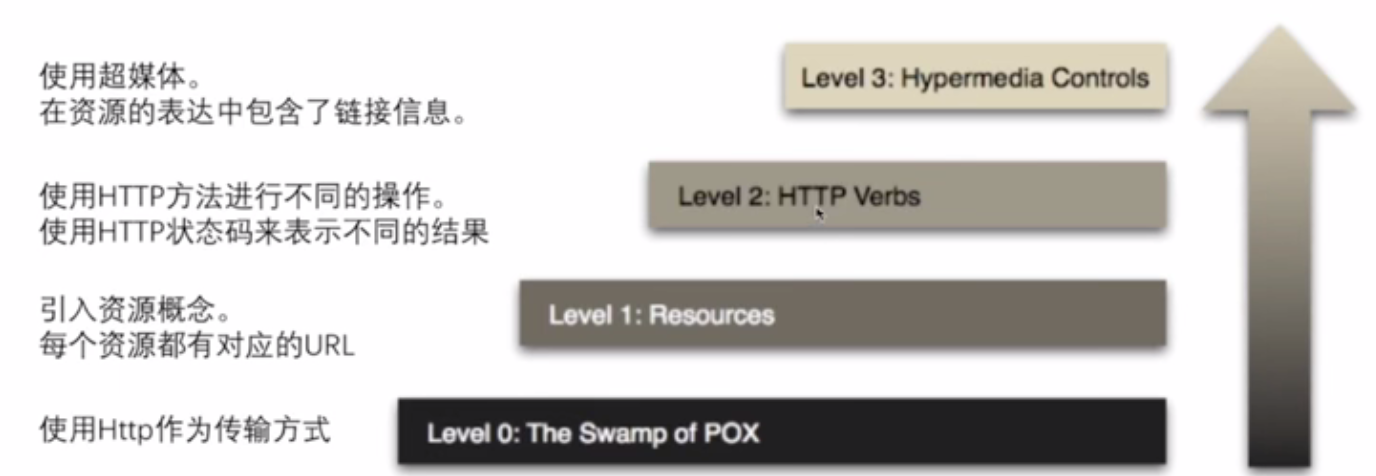
# 2.Restful

## 2.1.区别





成熟度模型



@RequestMapping value(path)参数表示url路径 method表示方法类型

@RequestParam name和value(相同)指定http请求中的参数名 required是否必须

@PageableDefault SpringData Pageable设置默认值

SpringBoot Test UserControllerTest

|  |
| --- |
| @RunWith(SpringRunner.class)//使用SpringRunner执行测试用例  @SpringBootTest  public class UserControllerTest {  @Autowired  private WebApplicationContext wac;  private MockMvc mvc;//伪造的mvc环境  /\*\*  \* Before注解在每次测试前都会执行  \*/  @Before  public void setup() {  mvc = MockMvcBuilders.webAppContextSetup(wac).build();//构建mvc的环境  }  @Test  public void whenQuerySuccess() throws Exception {  mvc.perform(MockMvcRequestBuilders.get("/user")//get的路径  .param("username","冴岛钢牙")//参数  .contentType(MediaType.APPLICATION\_JSON\_UTF8))//类型  //对上面请求结果的期望， 希望返回状态码是200  .andExpect(MockMvcResultMatchers.status().isOk())  //使用JsonPath判断 返回一个集合，长度为3  .andExpect(MockMvcResultMatchers.jsonPath("$.length()").value(3))  //获取返回 获取Response 转变成String 要设置编码格式  .andReturn().getResponse().getContentAsString();  /\*  \* response= xxx.getResponse();  \* response.setCharacterEncoding("UTF-8");  \* String json = response.getContentAsString();  \*  }  } |

## 2.2.JsonPath说明

见github; <https://github.com/json-path/JsonPath>

@PathVariable使用正则表达式匹配

|  |
| --- |
| @GetMapping(value = "/user/{id:\\d+}") //匹配数字 public User getUserInfo(@PathVariable(name ="id") String id){ |

## 2.3.JsonView使用

场景：list<User>返回时没有密码， 单个user返回时需要有密码。

使用方法：

### 2.3.1使用接口来声明多个视图

在User类声明接口

|  |
| --- |
| public interface UserSimpleView{}; public interface UserDetailView extends UserSimpleView{}; |

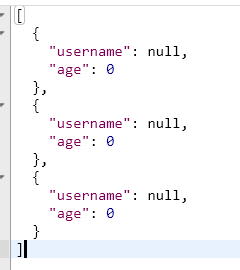
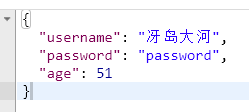
### 2.3.2在值对象的get方法上指定视图

|  |
| --- |
| 由于使用lombok，直接在属性上加  @JsonView(UserSimpleView.class) private String username; @JsonView(UserDetailView.class) private String password; @JsonView(UserSimpleView.class) private int age; |

### 2.3.3在controller方法上指定视图

|  |
| --- |
| @RequestMapping(value = "/user",method = RequestMethod.GET) @JsonView(User.UserSimpleView.class) public List<User> queryUser(){}  @GetMapping(value = "/user/{id:\\d+}")  @JsonView(User.UserDetailView.class)  public User getUserInfo(){} |

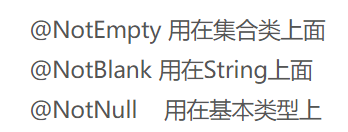
List返回 单个对象User

## 2.4.@Valid 参数校验,

### 2.4.1校验

在controller的方法的参数的对象前加上该注解会校验







当不符合数据规范时将抛出异常：

org.springframework.web.bind.MethodArgumentNotValidException

使用全局切面处理异常@ControllerAdvice

|  |
| --- |
| @ControllerAdvice  public class CheckAdvice {  @ExceptionHandler(MethodArgumentNotValidException.class)  public ResponseEntity<String> handleBindException(  MethodArgumentNotValidException e) {  return new ResponseEntity<String>(toStr(e), HttpStatus.BAD\_REQUEST);  }  private String toStr(MethodArgumentNotValidException ex) {  return ex.getBindingResult().getFieldErrors().stream()  .map(e->e.getDefaultMessage()) .reduce("",(s1,s2)->s1+"\n"+s2); }  } |

### 2.4.2.自定义校验

自定义校验注解

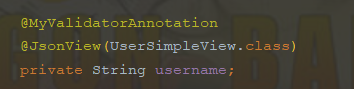
首先创建注解

|  |
| --- |
| @Target({ METHOD, FIELD})  @Retention(RUNTIME)  @Constraint(validatedBy = { MyConstraintValidator.class})//指定执行哪个类  public @interface MyValidatorAnnotation {  String message() default "你是没有继承牙狼称号的人";  Class<?>[] groups() default { };  Class<? extends Payload>[] payload() default { };  String prefix() default "冴岛";  } |

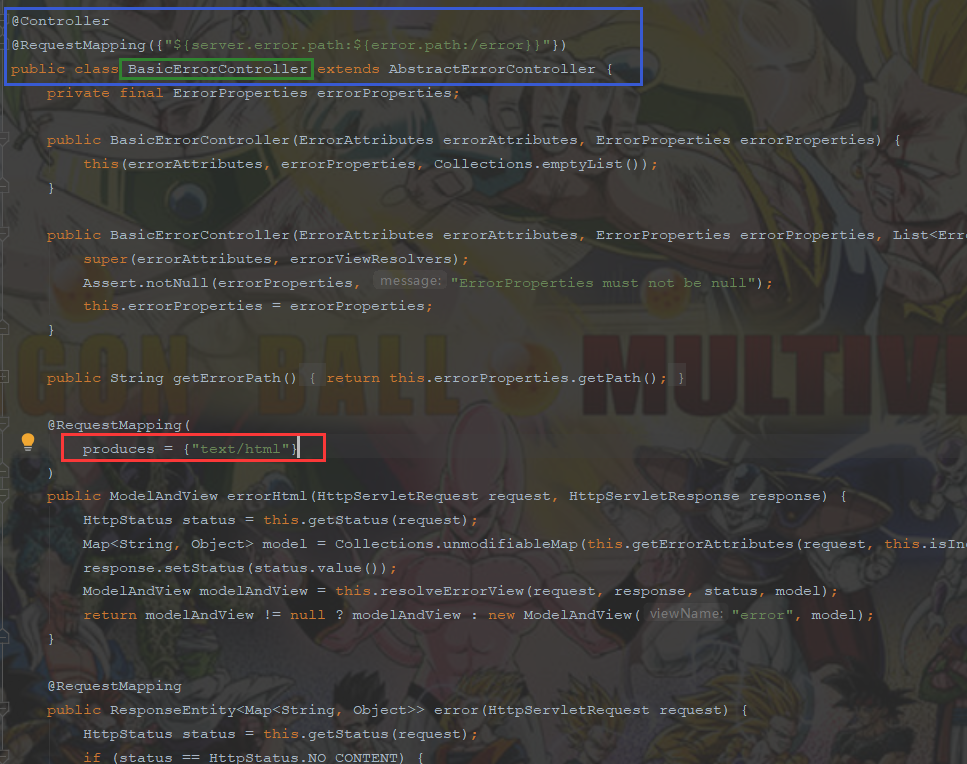
之后创建执行类

|  |
| --- |
| public class MyConstraintValidator implements ConstraintValidator< MyValidatorAnnotation, Object> {  //可以直接使用 @Autowired 注解，而不需要在类上加 @Component  private String prefix;  @Override  public void initialize(MyValidatorAnnotation constraintAnnotation) {  System.out.println("自定义注解校验类开始初始化");  //这里获取注解的信息 例如Range的min， max。在本类中添加相关的字段max,min，然后获取constraintAnnotation信息，赋值。在校验方法里，使用校验  prefix=constraintAnnotation.prefix();  }  @Override  public boolean isValid(Object value, ConstraintValidatorContext context) {  System.out.println("自定义注解开始校验");  System.out.println(value); //这是要校验的值，比如用在username上，那就是username的值  if("".equals(prefix)){  return true;  }  if(value==null ||"".equals(value)){  return false;  }  return value.indexOf(prefix)==0;  }  } |

之后在相关位置使用注解即可。

判断用户名的前缀，默认是冴岛

Spring Boot有默认的处理机制 一般出现异常会根据是否是浏览器请求，返回不同的格式。



自定义404页面,在resources文件下创建 resources\error文件夹 创建404.html 当浏览器访问路径404时，会返回404页面

## 2.5.Restful Api拦截

过滤器(Filter),拦截器(Interceptor),切片(Aspect)

### 2.5.1过滤器

1. 创建一个过滤器继承javax.servlet.Filter

|  |
| --- |
| @Component //将过滤器加入Spring Boot 中  @Slf4j  public class TimeFilter implements Filter {  @Override  public void init(FilterConfig filterConfig) throws ServletException {  log.info("TimeFilter.init");  }  @Override  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {  log.info("TimeFilter.doFilter");  log.info("start");  long startTime=System.currentTimeMillis();  chain.doFilter(request,response); //执行controller中的方法  log.info("filter time:"+(System.currentTimeMillis()-startTime)+"ms");  log.info("end");  }  @Override  public void destroy() {  log.info("TimeFilter.destroy");  }  } |

2.使用第三方过滤器 无法使用@Component注解时，手动配置。

创建一个config类，使用@Configuration注解 在该类中配置一个bean(相当于以前的web.xml配置)不能获取controller信息。

|  |
| --- |
| @Configuration //上面过滤器不能有@Component  public class WebConfig {  @Bean  public FilterRegistrationBean timeFilter(){  FilterRegistrationBean bean=new FilterRegistrationBean();  TimeFilter filter=new TimeFilter();  bean.setFilter(filter);  //过滤路径 多个  List<String> urls=new ArrayList<>();  urls.add("/user/\*");  bean.setUrlPatterns(urls);  return bean;  }} |

### 2.5.2.使用拦截器

可以获取到controller的信息

创建拦截器 实现HandlerInterceptor接口的三个方法，并使用注解@Component

|  |
| --- |
| @Component  public class TimeInterceptor implements HandlerInterceptor {  @Override  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  log.info("TimeInterceptor.preHandle");  log.info(((HandlerMethod)handler).getBean().getClass().getName());  log.info(((HandlerMethod)handler).getMethod().getName());  request.setAttribute("startTime", System.currentTimeMillis());  return false;  }  @Override  public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {  log.info("TimeInterceptor.postHandle");  Long time = (Long) request.getAttribute("startTime");  log.info("耗时："+(System.currentTimeMillis()-time));  }  @Override  public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {  log.info("TimeInterceptor.afterCompletion");  Long time = (Long) request.getAttribute("startTime");  log.info("耗时："+(System.currentTimeMillis()-time));  log.info("ex is ="+ex);  }  } |

之后创建一个config类(使用上面的WebConfig类)实现WebMvcConfigurer接口(原先：继承WebMvcConfigurerAdapter类过期)

|  |
| --- |
| @Configuration  public class WebConfig implements WebMvcConfigurer {  @Autowired  private TimeInterceptor timeInterceptor;  @Autowired  private TimeInterceptor2 timeInterceptor2;  @Override  public void addInterceptors(InterceptorRegistry registry) {  // timeInterceptor比timeInterceptor2先执行  //添加拦截器 registry.addInterceptor*(*timeInterceptor*)* //指定顺序 order越小优先级越高  .order*(*1*)* //指定路径  .addPathPatterns*(*"/user"*)*; registry.addInterceptor*(*timeInterceptor2*)*.order*(*2*)*.addPathPatterns*(*"/user"*)*;  }  } |

拦截器的执行顺序 1

|  |  |  |
| --- | --- | --- |
| 优先级高的preHandle | | |
| return true | | return false |
| 优先级低的preHandle | | 默认状态码200，无内容 |
| return true | return false |  |
| controller | 优先级高的afterCompletion | / |
| 优先级低的postHandle | 默认状态码200，无内容 | / |
| 优先级高的postHandle | / | / |
| View渲染 | / | / |
| 优先级低的afterCompletion | / | / |
| 优先级高的afterCompletion | / | / |

拦截器的执行顺序 2

|  |
| --- |
| 优先级高的preHandle |
| return true | |
| 优先级低的preHandle | |
| return true | |
| Controller方法 | |
| Controller方法发生异常 | |
| 优先级低的afterCompletion | |
| 优先级高的afterCompletion | |

拦截器的执行顺序 3

|  |  |
| --- | --- |
| 优先级高的preHandle | |
| return true | return false |
| 优先级低的preHandle报异常 | 默认状态码200，无内容 |
| 优先级高的afterCompletion | / |
| 返回异常信息 | / |

拦截器的执行顺序 4

|  |
| --- |
| 优先级高的preHandle |
| 优先级低的preHandle |
| controller |
| 优先级低的postHandle抛出异常 |
| View渲染 |
| 优先级低的afterCompletion(可以获取到异常) |
| 优先级高的afterCompletion (可以获取到异常) |
| 返回状态码200 controller返回的内容, 后台打印异常信息 |

拦截器的执行顺序 5

|  |
| --- |
| 优先级高的preHandle |
| 优先级低的preHandle |
| controller |
| 优先级低的postHandle |
| 优先级高的postHandle |
| View渲染 |
| 优先级低的afterCompletion 抛出异常 后台打印异常信息 |
| 优先级高的afterCompletion (获取不到到异常) |
| 返回状态码200 controller返回的内容 |

如果controller中出现异常被@ControllerAdvice注解的类处理后，afterCompletion不会检测到异常类信息, 但是postHandle不会执行。

路径匹配问题

|  |  |
| --- | --- |
| ? | 匹配任何单字符 |
| \* | 匹配0或者任意数量的字符 |
| \*\* | 匹配0或者更多的目录 |

例如 /user\* 能匹配 /user /user1 /userasd 不能匹配 /user/1

/user/\*\* 能匹配 /user /user/dsad /user/1123/45 不能匹配 /user2

/user\*\* 同/user\*

/user\*/\*\* 能匹配 /user /user1 /user/ds /user12/ds /user/sd/fdfd 全能匹配

拦截器的缺点，无法拿到controller方法中的参数对象的值。

原理DispatcherServlet的doService调用了doDispatch()方法

doDispatch方法中有一下方法

|  |
| --- |
| //先调用拦截器preHandle方法，如果返回false直接结束  if (!mappedHandler.applyPreHandle(processedRequest, response)) {   return; } // Actually invoke the handler 数据拼装在这步  mv = ha.handle(processedRequest, response, mappedHandler.getHandler()); |

### 2.5.3.使用切面

添加切面依赖

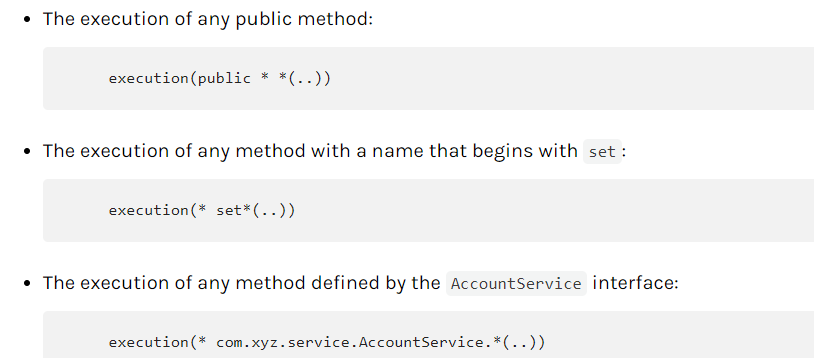
|  |
| --- |
| compile "org.springframework.boot:spring-boot-starter-aop:${springBootVersion}" |

创建切面

|  |
| --- |
| @Aspect  @Component  public class TimeAspect {  @Before //类似拦截器的preHandle  @After //类似拦截器的postHandle  @AfterThrowing //类似拦截器的afterCompletion  //包围，覆盖了前3种方式(RequestMapping包含GetMapping和PostMapping一样)定义了在什么方法上起作用，在什么时候起作用  @Around("execution(\* com.swb.security.demo.controller.UserController.\*(..))")  public Object handleControllerMethod(ProceedingJoinPoint pjp) throws Throwable {  log.info("TimeAspect.handleControllerMethod");  Object[] args = pjp.getArgs(); //获取参数，  for (Object arg:args){  System.out.println(arg);  }  log.info("============================");  //controller方法返回的对象，调用该方法才执行controller的方法  Object o=pjp.proceed();  log.info("============================");  System.out.println(o);  return o;  }  } |

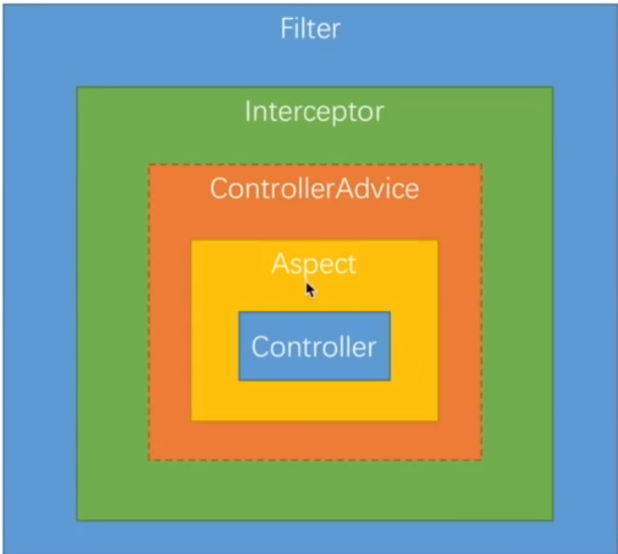
Around 括号内填写相关表达式 详见例子

https://docs.spring.io/spring/docs/5.2.4.RELEASE/spring-framework-reference/core.html#aop-pointcuts-examples



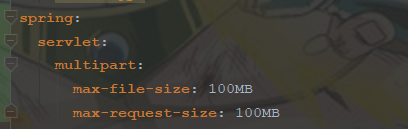
三者比较

|  |  |  |
| --- | --- | --- |
|  | 好处 | 坏处 |
| Filter(过滤器) | 能拿到原始的http请求信息 | 拿不到处理请求的方法信息 |
| Interceptor(拦截器) | 能拿到原始的http请求信息  能拿到处理请求的方法信息 | 拿不到调用时的参数值 |
| Aspect(切面) | 能拿到调用时的参数值 | 拿不到http请求信息  拿不到处理请求的方法信息 |



## 2.6.上传下载

application.yml配置

Controller

|  |
| --- |
| @RestController  @RequestMapping("/files")  public class FileController {  @Value("${filePath}")  private String filePath;  @PostMapping  public FileInfo fileUpload(HttpServletRequest request, @RequestParam(value = "file", required = false) MultipartFile file) throws IOException {  System.out.println(file.getName());  System.out.println(file.getOriginalFilename());  System.out.println(file.getSize());  // file.getInputStream();获取输入流  String fileName = System.currentTimeMillis() + ".txt";  File loadFile = new File(filePath, fileName);  file.transferTo(loadFile);  return new FileInfo("/file/" + fileName);  }  } |

文件下载问题

|  |
| --- |
| //使用静态拦截方法,创建配置类，注解Configuration 实现WebMvcConfigurer  //重写addResourceHandlers方法 txt和图片需要另存为  @Override  public void addResourceHandlers(ResourceHandlerRegistry registry) {  //映射文件地址  registry.addResourceHandler("/file/\*\*").addResourceLocations("file:" + filePath);  } |
| //controller方法 获取文件输入流，输出到response中，设置ContentType为下载  @GetMapping("/{id}")  public void download(@PathVariable String id, HttpServletResponse response) throws Exception {  try (InputStream inputStream = new FileInputStream(new File(filePath, id + ".txt"));  OutputStream outputStream = response.getOutputStream();) {  response.setContentType ("application/x-download");  response.addHeader("Content-Disposition", "attachment;filename=test.txt");  IOUtils.copy(inputStream, outputStream);  outputStream.flush();  }  } |

## 2.7.异步处理Rest服务

使用Runnable异步处理,使用DeferredResult异步处理，异步处理配置

异步处理增加吞吐量

### 2.7.1使用Runnable异步处理

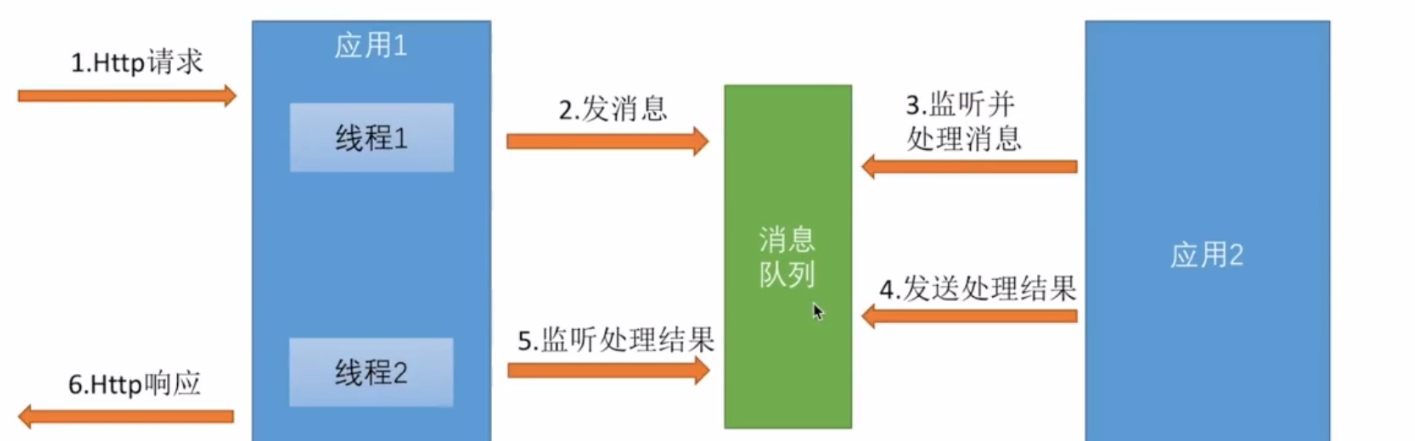
|  |
| --- |
| @RestController @Slf4j public class AsyncController {  @GetMapping("/order")  public Callable<String> order() throws InterruptedException {  log.info("主线程开始");  Callable<String> callable=()->{  log.info("副线程开始");  Thread.sleep(1000L);  log.info("副线程结束");  return "order";  };  log.info("主线程结束");  return callable;  } } |
|  |

客户端拿到的数据的时间还是1000ms之后,tomcat主线程结束很快，可以处理之后的Http请求,副线程处理完后，再返回结果。

### 2.7.2使用DeferredResult异步处理

使用Runnable处理时实在controller方法中创建Callable处理的,有主线程调用副线程

DeferredResult使用场景



创建一个模拟队列,即消息队列和应用2

|  |
| --- |
| @AllArgsConstructor  @NoArgsConstructor  @Slf4j  @Component  public class MockQueue {  private String placeOrder;  private String completeOrder;  public String getPlaceOrder() {  return placeOrder;  }  //重要方法  public void setPlaceOrder(String placeOrder) {  new Thread(()->{  log.info("接到下单请求:" + placeOrder);  try {  Thread.sleep(1000L);  this.completeOrder = placeOrder;  } catch (InterruptedException e) {  e.printStackTrace();  }  log.info("下单请求:"+placeOrder+"处理完毕");  }).start();  }  public String getCompleteOrder() {  return completeOrder;  }  public void setCompleteOrder(String completeOrder) {  this.completeOrder = completeOrder;  }  } |

创建DeferredResultHolder在应用1的2个线程中传递DeferredResult对象

|  |
| --- |
| @Component  @Data  public class DeferredResultHolder {  /\*\*  \* map中的String 是订单号  \* DeferredResult<String>是处理结果  \*/  private Map<String, DeferredResult<String>> map=new HashMap<String, DeferredResult <String>>();  } |

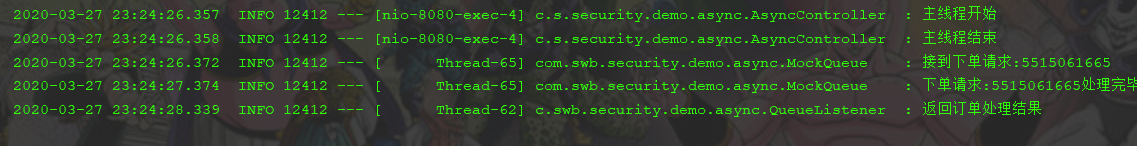
创建应用1的线程1,即相应的Http请求

|  |
| --- |
| @RestController  @Slf4j  @AllArgsConstructor  public class AsyncController {  private MockQueue mockQueue; private DeferredResultHolder deferredResultHolder;  @GetMapping("/order2") public DeferredResult<String> order2() {  log.info("主线程开始");  String orderNumber= RandomStringUtils.randomNumeric(10);  mockQueue.setPlaceOrder(orderNumber);  DeferredResult<String> result=new DeferredResult<>();  deferredResultHolder.getMap().put(orderNumber,result);  log.info("主线程结束");  return result; }  } |

创建队列监听器，即应用1的线程2

|  |
| --- |
| @Component  @Slf4j  @AllArgsConstructor  public class QueueListener implements ApplicationListener<ContextRefreshedEvent> {  private MockQueue mockQueue;  private DeferredResultHolder deferredResultHolder;  @SuppressWarnings("AlibabaAvoidManuallyCreateThread")  @Override  public void onApplicationEvent(ContextRefreshedEvent event) {  //noinspection AlibabaAvoidManuallyCreateThread  new Thread(()->{  while (true){  if (StringUtils.isNotBlank(mockQueue.getCompleteOrder())){  String orderNumber=mockQueue.getCompleteOrder();  log.info("返回订单处理结果");  //当调用setResult方法时，返回处理结果  deferredResultHolder.getMap().get(orderNumber).setResult("order");  mockQueue.setCompleteOrder(null);  }else{  try {  Thread.sleep(1000L);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  }).start();  }  } |

bean注入方式，除@Autowired外，使用构造方法时，不能有无参构造函数，只能一个构造函数，包含所有要注入的bean，可以使用lombok的@AllArgsConstructor注解



Tomcat的一个线程接受请求，之后由另一个线程处理，最后由第三个线程返回。而tomcat的线程早就结束了，可以处理新的请求。

前端拿到数据的时间是最后第三个线程的调用DeferredResult. setResult()方法的时间。

异步处理配置

WebConfig实现的WebMvcConfigurer接口的一个方法configureAsyncSupport其参数AsyncSupportConfigurer 主要配置异步的拦截器，超时时间，指定Runnable使用的线程池。

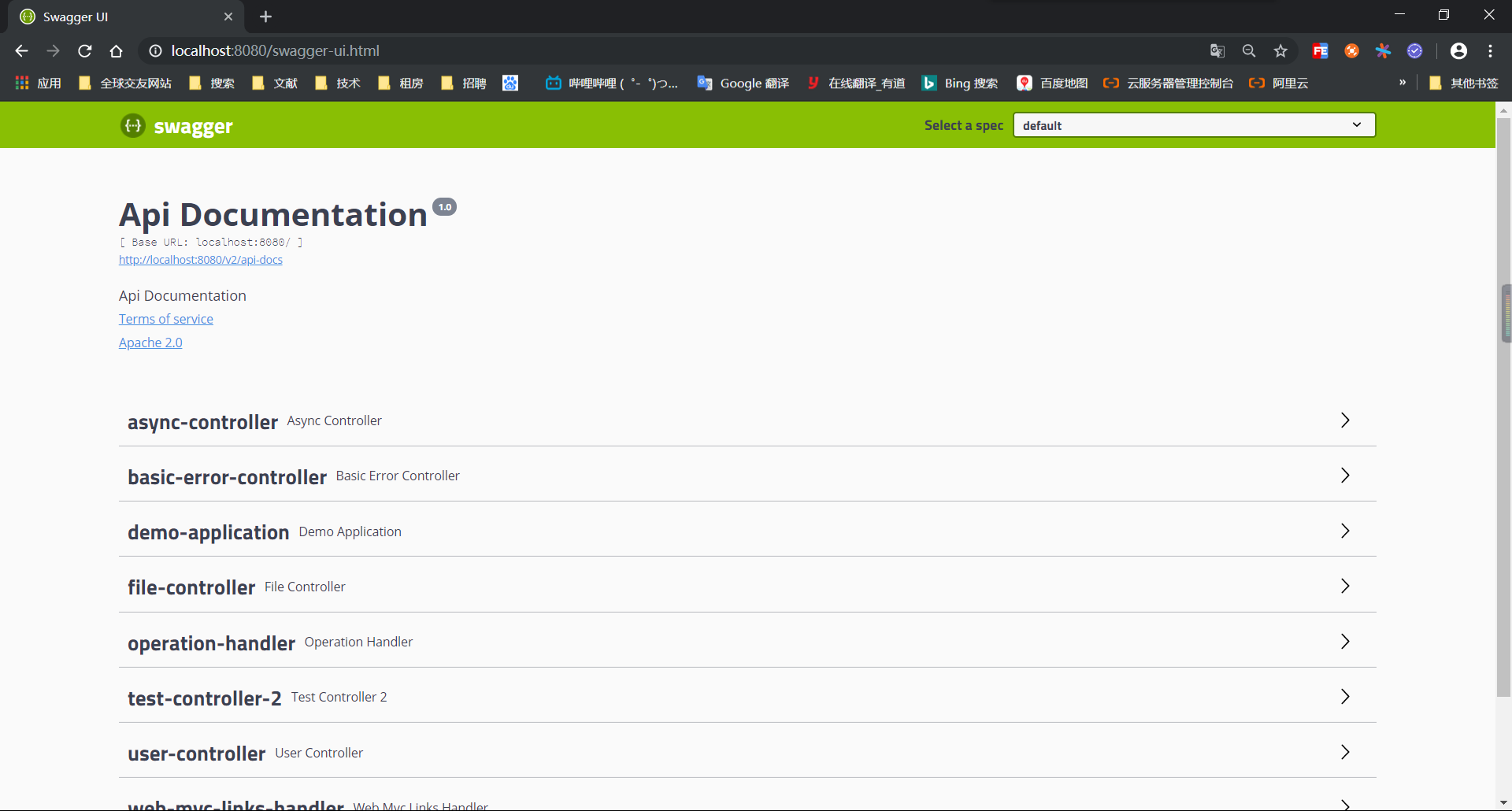
## 2.8使用swagger生产文档

demo添加依赖

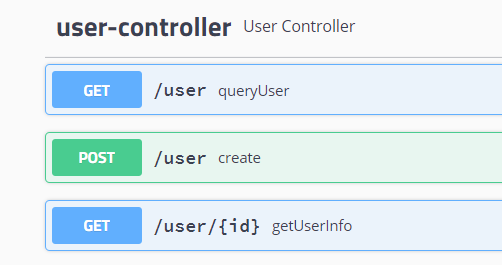
|  |
| --- |
| compile group: 'io.springfox', name: 'springfox-swagger2', version: "${springfoxVersion}" compile group: 'io.springfox', name: 'springfox-swagger-ui', version: "${springfoxVersion}" |

在SpringBoot启动类上添加@EnableSwagger2注解

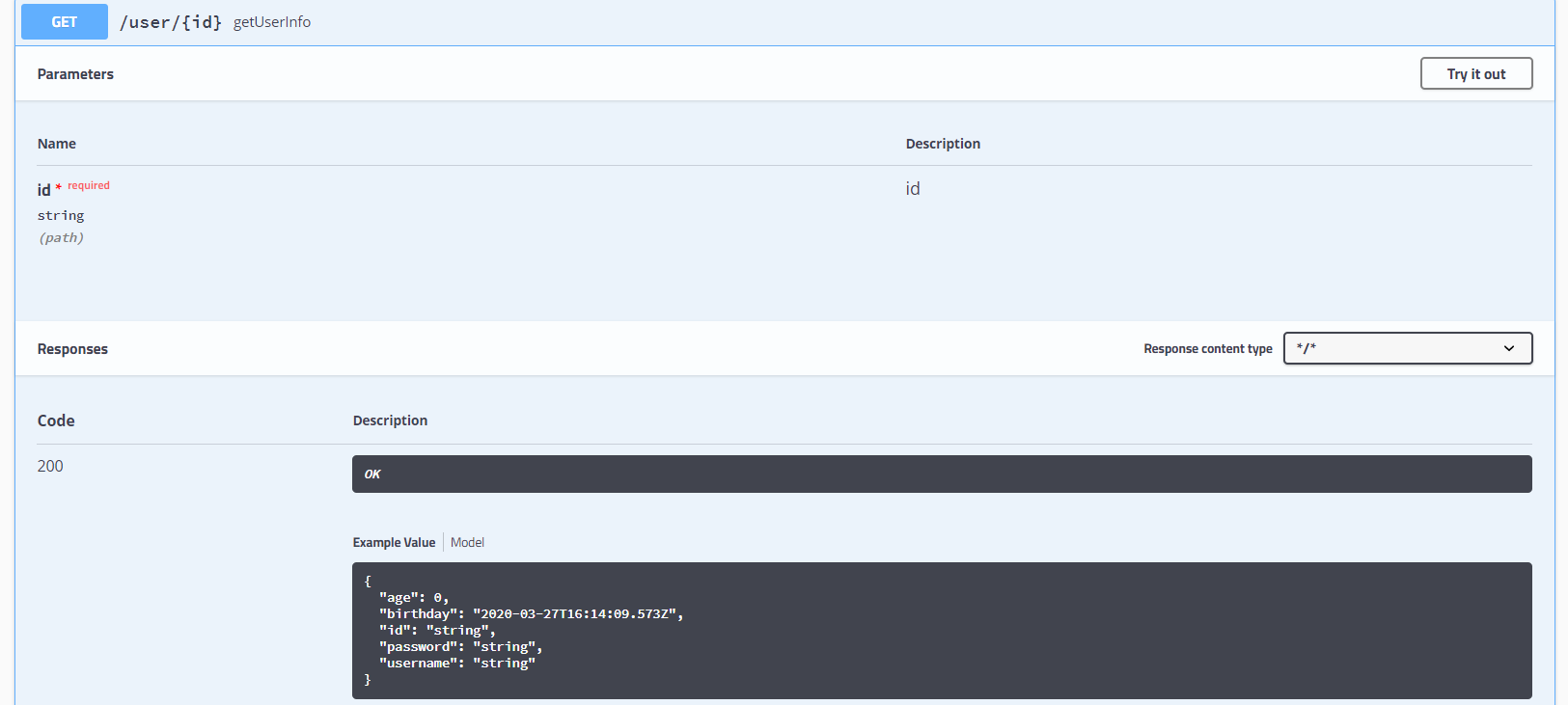
启动项目访问：<http://localhost:8080/swagger-ui.html>



点击user-controller下方展示所有的请求



点击请求展示相关示例



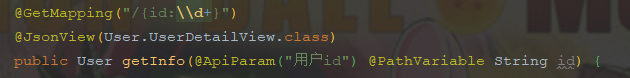
以上都是默认显示，如请求后面显示方法名,对请求进行描述：在方法上添加注解



对参数写注解，如果参数是对象那么在对象上的属性写注解@ApiModelProperty



对于参数为基本类型的注解



Json返回对象时，也在属性上添加注解ApiModelProperty

详见swagger文档。

## 2.9.wiremock

使用wiremock伪造假数据

官网下载wiremock的jar包：<http://wiremock.org/docs/running-standalone/>

demo中添加wiremock依赖

|  |
| --- |
| testCompile group: 'com.github.tomakehurst', name: 'wiremock', version: '2.26.3', ext: 'pom' |

在test包下创建MockServer类

|  |
| --- |
| public class MockServer {  //添加一个url(/order/1 ,返回withBody里的json值，状态码200),运行main方法即可  public static void main(String[] args) {  //端口  configureFor(8062);  removeAllMappings();  stubFor(get(urlPathEqualTo("/order/1"))  .willReturn(aResponse().withBody("{\"id\":1}").withStatus(200)));  }  } |

访问<http://localhost:8060/order/1> 将返回 json 

使用详情见官网

# 3. Spring Security认证

## 3.1说明

核心功能：认证(你是谁-登录)，授权(能做什么)，攻击防护(防止伪造身份)

打开demo的security配置,在启动类上删除SpringBootApplication注解上的exclude内容中的Security相关类。

启动Demo,访问<http://localhost:8080/user>时，出现登录提示，登录后才能访问。

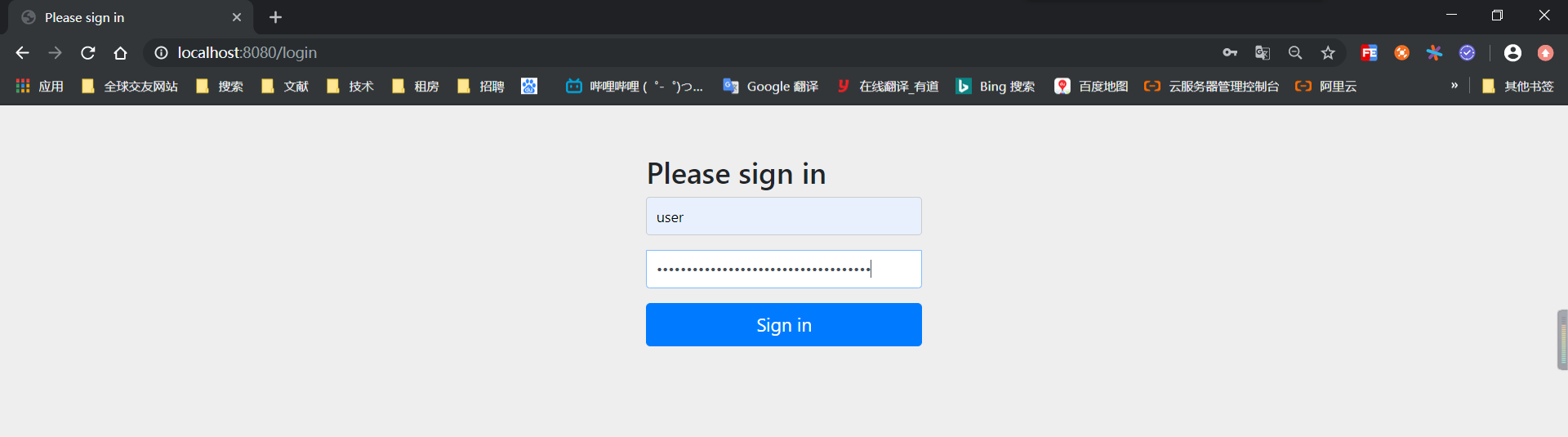
在browser中添加一个Security的配置类，重写Http请求时的配置

|  |
| --- |
| @Configuration  @Slf4j  public class BrowserSecurityConfig extends WebSecurityConfigurerAdapter {  @Override  protected void configure(HttpSecurity http) throws Exception {  log.info("BrowserSecurityConfig.configure");  // 指定身份认证的方式 表单登录  http.formLogin()  // basic方式认证  // http.httpBasic()  .and()  // 开始请求权限配置  .authorizeRequests()  // 下面2行是对http所有的请求必须通过授权认证才可以访问  // 对任何请求  .anyRequest()  // 都需要身份验证  .authenticated();  }  } |

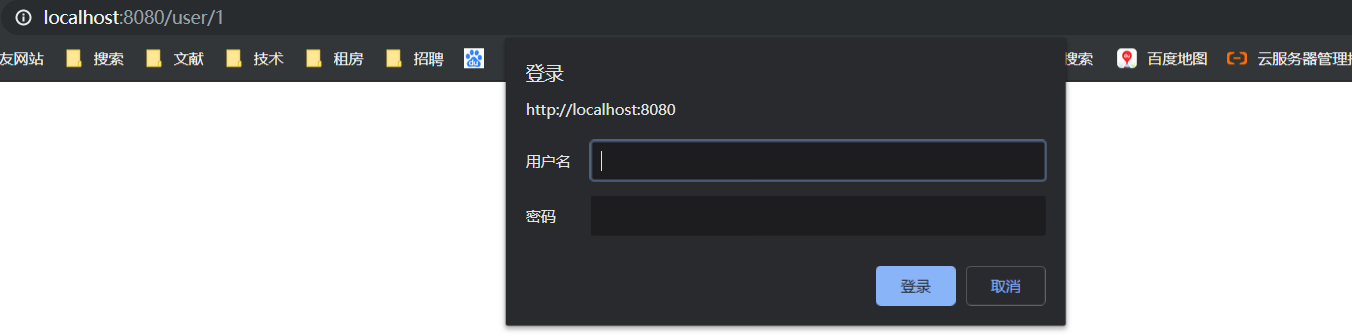
Demo引用了core和Browser，因此在Browser中配置，可以供其他模块使用，并且可以提取出来给后续功能点使用。

Demo启动类位于com.swb.security.demo包下，而briwser都位于com.swb.security.browser包下，因此需要在config类(添加了@Configuration注解的类)上添加注解,指定额外扫描某个包,或者将启动类放到com.swb.security包下。@ComponentScan(basePackages = { "com.swb.security.browser"})，查看是否加载到可以在方法中添加一行日志

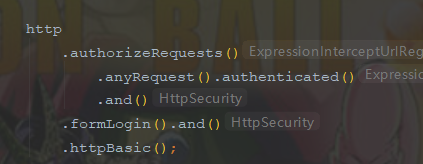
Form登录方式



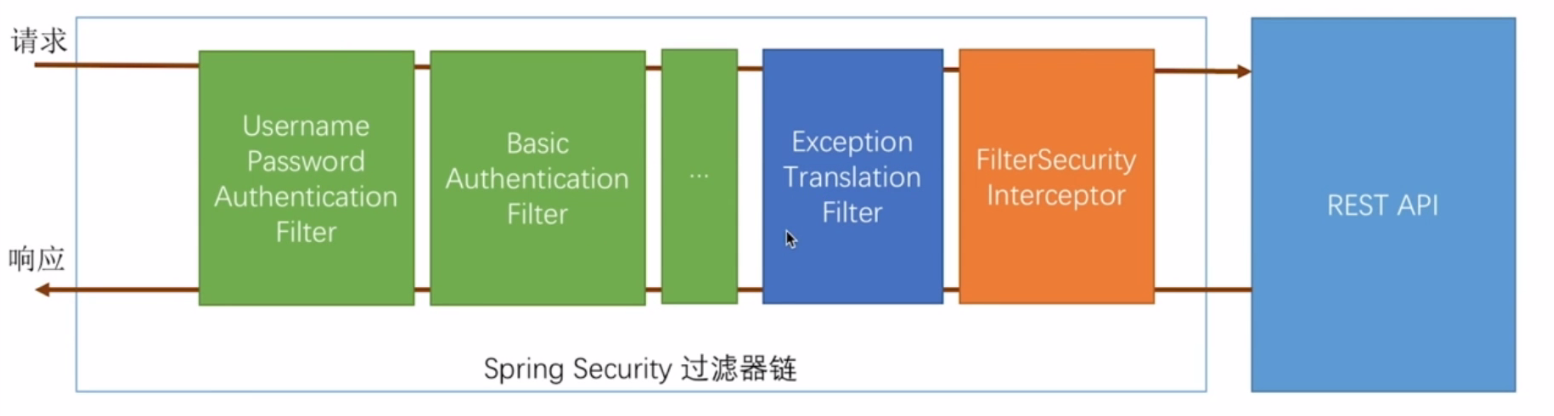
httpBasic方式登录



默认是



Security过程



绿色的为所有认证用户身份使用。每种过滤器处理一种登录方式。只要有一个认证通过，该请求会被标记，表示请求认证成功，如果不成功，继续下一个过滤器。

当认证成功后，有橙色的过滤器判断是否能够访问。如果不能访问，会抛出异常由蓝色过滤器处理，例如没权限，没认证等异常。之后绿色才能修改，比如禁用某个认证。而其他过滤器不能修改。

## 3.2.自定义用户逻辑认证

处理用户信息获取逻辑(从数据库中获取)，处理用户校验逻辑，处理密码加密解密

### 3.2.1处理用户信息获取逻辑

实现UserDetailsService接口 该接口只有一个loadUserByUsername方法。读取用户信息，进行校验。

在browser中实现该接口

|  |
| --- |
| @Component  @Slf4j  public class MyUserDetailsService implements UserDetailsService {  /\*\*  \* 注入持久层信息 例如 mybatis的mapper, SpringDataJPA的Repository  \* 输入用户名，从数据库中获取数据，这里是模拟获取  \* User对象为org.springframework.security.core.userdetails包下的，实现了UserDetails接口  \* 参数，用户名，密码，权限List<GrantedAuthority>  \* @param username  \* @return  \* @throws UsernameNotFoundException  \*/  @Override  public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  log.info("MyUserDetailsService.loadUserByUsername");  log.info("登录名为:"+username);  BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();  String password = encoder.encode("123456");  return new User(username,password, AuthorityUtils.commaSeparatedStringToAuthorityList("admin"));  }  } |

Security5需要添加加密处理

在browser的BrowserSecurityConfig中添加配置

|  |
| --- |
| @Bean public PasswordEncoder passwordEncoder() {  // 使用 BCrypt 加密  return new BCryptPasswordEncoder();  } |

1. 用户校验逻辑

分为密码是否匹配(由security处理)，用户是否冻结，是否过期等

由UserDetails提供的4个方法实现

|  |
| --- |
| //账户有没有过期 true没有过期  boolean isAccountNonExpired();  //账户是否锁定 true没有锁定 可表示用户被冻结  boolean isAccountNonLocked();  //密码有没有过期 true没有过期  boolean isCredentialsNonExpired();  //是否可用 true用户可用 可表示用户被删除  boolean isEnabled(); |

使用org.springframework.security.core.userdetails包下的User对象时，可以在构造函数上添加参数，自己定义的类则是重写上述方法。

1. 加密解密处理

加密已再上方添加过来，可以采用BcryptPasswordEncoder加密，也可以自定义加密。

BcryptPasswordEncoder每次加密时，输出的密码都不相同，判断时需要使用matches方法

### 3.2.2.个性化用户认证流程

自定义登录页面,自定义登录成功处理，自定义登录失败处理

#### 3.2.2.1.自定义登录页面

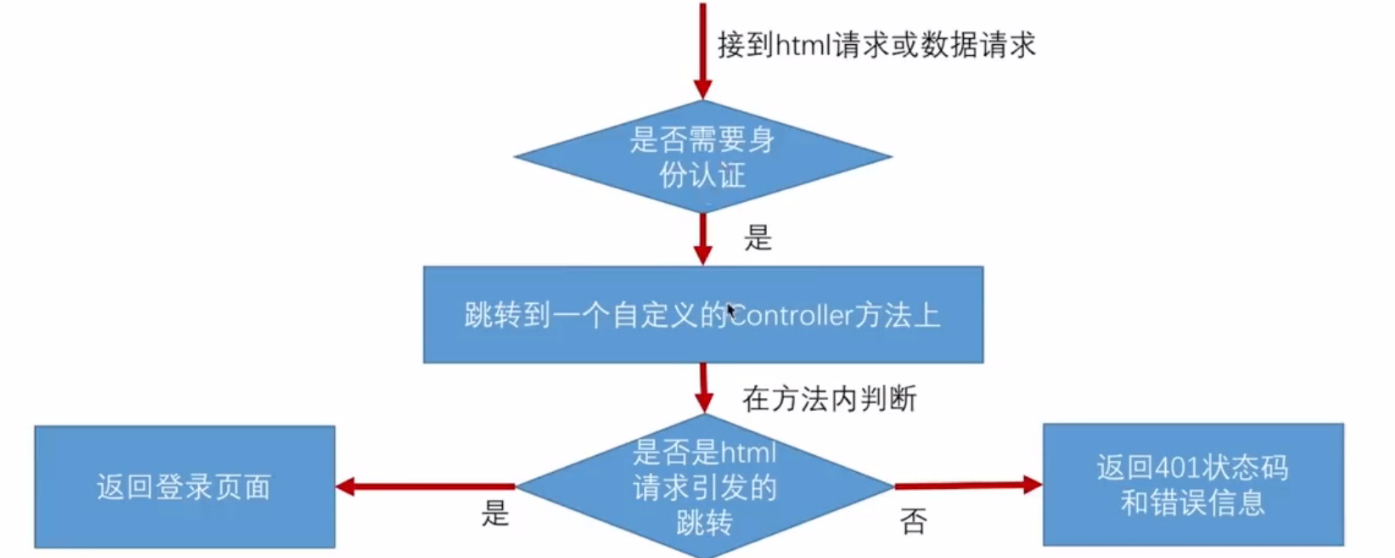
重新修改登录页面，在browser的security中修改相关的配置

|  |
| --- |
| @Override protected void configure(HttpSecurity http) throws Exception {  log.info("BrowserSecurityConfig.configure"); // 指定身份认证的方式 表单登录  http.formLogin()  //指定登录页面  .loginPage("/login.html")  //自定义登录请求路径,默认是post方式的login . loginProcessingUrl("/authentication/form") // basic方式认证 // http.httpBasic()  .and() // 开始请求权限配置  authorizeRequests()  //当前路径不需要认证  .antMatchers("/login.html").permitAll() // 下面2行是对http所有的请求必须通过授权认证才可以访问 // 对任何请求  .anyRequest() // 都需要身份验证  .authenticated().and().csrf().disable(); } |

在browser中添加一个登录页面src\main\resources\resources\login.html

|  |
| --- |
| <form method="post" action="/authentication/form">  <input type="text" id="username" name="username" maxlength="50" placeholder="请输入账号">  <input type="password" id="password" name="password" maxlength="30" placeholder="请输入密码" > |

处理不同类型的请求，html请求或数据请求



在core中添加配置相关的类,应用于以后的多个系统，配置相关信息



在core中添加SecurityProperties,BrowserProperties

|  |
| --- |
| @Component  @ConfigurationProperties(prefix = "swb.security")  @Data  public class SecurityProperties {  private BrowserProperties browser=new BrowserProperties();  }  @Data  public class BrowserProperties {  //默认值  private String loginPage="/browserLogin.html";  } |

修改browser的securityConfig配置

|  |
| --- |
| @Override protected void configure(HttpSecurity http) throws Exception {  log.info("BrowserSecurityConfig.configure"); // 指定身份认证的方式 表单登录  http.formLogin()  //指定未登录时跳转位置  .loginPage("/authentication/require")  //自定义登录请求路径,默认是post方式的login  .loginProcessingUrl("/authentication/form") // basic方式认证 // http.httpBasic()  .and() // 开始请求权限配置  .authorizeRequests()  //当前路径不需要认证, 未登录时跳转位置,登录页面  .antMatchers("/authentication/require",securityProperties.getBrowser().getLoginPage())  .permitAll() // 下面2行是对http所有的请求必须通过授权认证才可以访问 // 对任何请求  .anyRequest() // 都需要身份验证  .authenticated().and().csrf().disable(); } |

在browser中添加一个controller用于处理请求是否需要跳转

|  |
| --- |
| @RestController @Slf4j public class BrowserSecurityController {  private RequestCache cache=new HttpSessionRequestCache();  private RedirectStrategy redirectStrategy=new DefaultRedirectStrategy();  @Autowired  private SecurityProperties securityProperties;  /\*\*  \* 当需要身份认证是跳转到这里  \* @param request  \* @param response  \* @return  \*/  @RequestMapping("/authentication/require")  @ResponseStatus(code= HttpStatus.UNAUTHORIZED)  public SimpleResponse requireAuthentication(HttpServletRequest request,  HttpServletResponse response) throws IOException {  SavedRequest savedRequest=cache.getRequest(request,response);  if(savedRequest!=null){  String target=savedRequest.getRedirectUrl();  log.info("引发跳转的请求是："+target);  if(StringUtils.endsWithIgnoreCase(target,".html")){  //跳转路径 redirectStrategy.sendRedirect(request,response,securityProperties.getBrowser().getLoginPage());  }  }  return new SimpleResponse("访问信息需要身份认证");  } } |

当未登录时，如果访问的是数据(未以.html结尾时)返回401信息，如果是页面，返回登录页面。当application.yml中未配置swb.security.browser.loginPage值时，默认登录页面是browser，当指定某个值时，跳转到该页面。

#### 3.2.2.2.自定义登录成功处理

在browser中实现AuthenticationSuccessHandler接口

|  |
| --- |
| @Component("swbSecurityAuthenticationSuccessHandler")  @Slf4j  public class SecurityAuthenticationSuccessHandler implements AuthenticationSuccessHandler {  //json转化  @Autowired  private ObjectMapper objectMapper;  /\*\*  \* @param request  \* @param response  \* @param authentication security核心接口,封装了一些认证信息(包括请求的ip,session,UserDetails)  \* @throws IOException  \* @throws ServletException  \*/  @Override  public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response, Authentication authentication) throws IOException, ServletException {  log.info("SecurityAuthenticationSuccessHandler.onAuthenticationSuccess:登录成功");  //将authentication转化为json返回  response.setContentType(MediaType.APPLICATION\_JSON\_UTF8\_VALUE);  response.getWriter().write(objectMapper.writeValueAsString(authentication));  }  } |

在browser的security中添加配置

|  |
| --- |
| @Configuration  @Slf4j  public class BrowserSecurityConfig extends WebSecurityConfigurerAdapter {  @Autowired  private SecurityProperties securityProperties;  @Autowired  @Qualifier("swbSecurityAuthenticationSuccessHandler")  private AuthenticationSuccessHandler successHandler;  @Override  protected void configure(HttpSecurity http) throws Exception {  log.info("BrowserSecurityConfig.configure");  // 指定身份认证的方式 表单登录  http.formLogin()  //指定登录页面  .loginPage("/authentication/require")  //自定义登录请求路径,默认是post方式的login  .loginProcessingUrl("/authentication/form")  //登录成功自定义处理注入  .successHandler(successHandler)  // basic方式认证  // http.httpBasic()  .and()  // 开始请求权限配置  .authorizeRequests()  //当前路径不需要认证  .antMatchers("/authentication/require",securityProperties.getBrowser().getLoginPage())  .permitAll()  // 下面2行是对http所有的请求必须通过授权认证才可以访问  // 对任何请求  .anyRequest()  // 都需要身份验证  .authenticated().and().csrf().disable();  }  @Bean  public PasswordEncoder passwordEncoder() {  // 使用 BCrypt 加密  return new BCryptPasswordEncoder();  }  } |

登录之后页面返回一下json格式,会根据登录方式不同，而返回不同的方式



#### 3.2.2.3.登录错误处理(与成功类似)

Browser中实现AuthenticationFailureHandler接口

|  |
| --- |
| @Component("swbSecurityAuthenticationFailureHandler")  @Slf4j  public class SecurityAuthenticationFailureHandler implements AuthenticationFailureHandler {  @Autowired  private ObjectMapper objectMapper;  @Override  public void onAuthenticationFailure(HttpServletRequest request, HttpServletResponse response, AuthenticationException exception) throws IOException, ServletException {  log.info("SecurityAuthenticationFailureHandler.onAuthenticationFailure:登录失败");  //将authentication转化为json返回  response.setStatus(HttpStatus.INTERNAL\_SERVER\_ERROR.value());  response.setContentType(MediaType.APPLICATION\_JSON\_UTF8\_VALUE);  response.getWriter().write(objectMapper.writeValueAsString(exception));  }  } |

SecurityConfig中添加配置

|  |
| --- |
| …………  @Autowired @Qualifier("swbSecurityAuthenticationFailureHandler") private AuthenticationFailureHandler failureHandler;  …………  //登录成功自定义处理注入 .successHandler(successHandler) //登录失败自定义处理注入 .failureHandler(failureHandler)  ………… |

#### 3.2.2.4.多种返回格式

上述修改之后便是全部变成返回json

之后修改变成同时支持2种方式

在core中添加一个枚举，并在browser中添加一个属性

|  |
| --- |
| /\*\*  \* @author swb  \* 时间 2020-03-30 00:23  \* 文件 LoginType  \*/  public enum LoginType {  /\*\*  \* 跳转  \*/  REDIRECT,  /\*\*  \* 返回json  \*/  JSON  }  BrowserProperties{……  private LoginType loginType = LoginType.JSON; ……} |

修改自定义的成功处理器，不实现接口，改为继承已实现默认的处理器SavedRequestAwareAuthenticationSuccessHandler

|  |
| --- |
| @Component("swbSecurityAuthenticationSuccessHandler")  @Slf4j  @AllArgsConstructor  public class SecurityAuthenticationSuccessHandler extends SavedRequestAwareAuthenticationSuccessHandler {  private ObjectMapper objectMapper;  private SecurityProperties securityProperties;  /\*\*  \* @param request  \* @param response  \* @param authentication security核心接口,封装了一些认证信息(包括请求的ip,session,UserDetails)  \* @throws IOException  \* @throws ServletException  \*/  @Override  public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response, Authentication authentication) throws IOException, ServletException {  log.info("SecurityAuthenticationSuccessHandler.onAuthenticationSuccess:登录成功");  if (LoginType.JSON.equals(securityProperties.getBrowser().getLoginType())) {  //将authentication转化为json返回  response.setContentType(MediaType.APPLICATION\_JSON\_UTF8\_VALUE);  response.getWriter().write(objectMapper.writeValueAsString(authentication));  } else {  super.onAuthenticationSuccess(request, response, authentication);  }  }  } |

登录失败继承SimpleUrlAuthenticationFailureHandler类

默认为JSON, application.yml的配置

|  |
| --- |
| swb.security.browser.loginType: REDIRECT  swb.security.browser.loginType: JSON |

#### 3.2.2.5.获取用户信息

获取用户的信息 上面的2种方法包含登录的其他信息，下面的方法只包含user信息(数据库中的信息)

|  |
| --- |
| @GetMapping("/security/me") public Object getCurrentUser(){  return SecurityContextHolder.getContext().getAuthentication(); } |
| @GetMapping("/security/me2") public Object getCurrentUser2(Authentication authentication){  return authentication; } |
| @GetMapping("/security/me3") public Object getCurrentUser2(@AuthenticationPrincipal UserDetails user){  return user; } |

## 3.3图片验证码

### 3.3.1实现图片验证码

开发生产图形验证码接口，在认证流程中加入图形验证码校验，重构代码

生产图形原验证码：随机数生成图片；将随机数存放到Session（redis）中；将生成的图片写道接口的响应中，展示到前端

写在core中，加入hutool(github上的)依赖

创建imageCode，实现AbstractCaptcha接口，加入过期时间属性(时间类)其他同其提供的CircleCaptcha实现类相同，也可以直接使用以提供的实现类

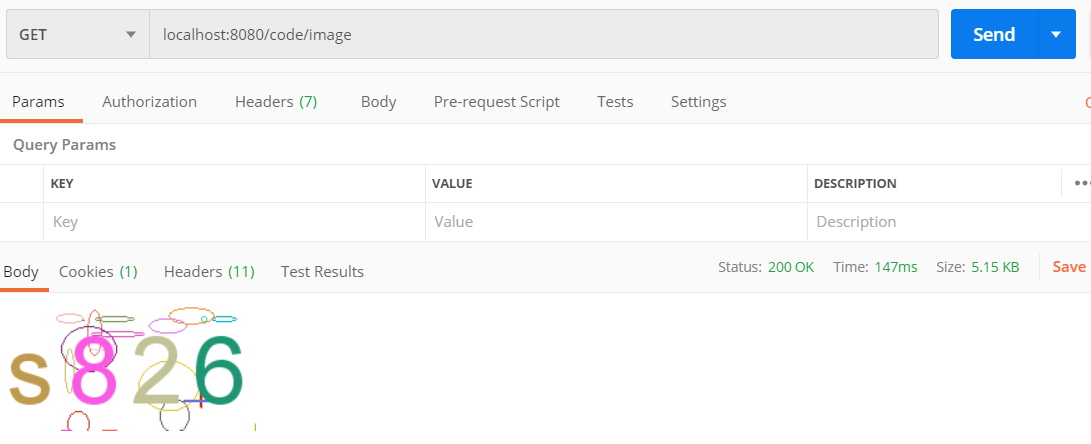
|  |
| --- |
| public class ImageCode extends AbstractCaptcha {  @Setter  @Getter  private LocalDateTime expireTime;  private static final long serialVersionUID = 3180820918087507384L;  /\*\*  \* 构造  \*  \* @param width 图片宽  \* @param height 图片高  \*/  public ImageCode(int width, int height) {  this(width, height, 5);  }  /\*\*  \* 构造  \*  \* @param width 图片宽  \* @param height 图片高  \* @param codeCount 字符个数  \*/  public ImageCode(int width, int height, int codeCount) {  this(width, height, codeCount, 15, 300);  }  /\*\*  \* 构造  \*  \* @param width 图片宽  \* @param height 图片高  \* @param codeCount 字符个数  \* @param interfereCount 验证码干扰元素个数  \* @param expireIn 过期秒数  \*/  public ImageCode(int width, int height, int codeCount, int interfereCount, int expireIn) {  super(width, height, codeCount, interfereCount);  this.expireTime = LocalDateTime.now().plusSeconds(expireIn);  }  @Override  public Image createImage(String code) {  final BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE\_INT\_RGB);  final Graphics2D g = ImgUtil.createGraphics(image, ObjectUtil.defaultIfNull(this.background, Color.WHITE));  // 随机画干扰圈圈  drawInterfere(g);  // 画字符串  drawString(g, code);  return image;  }  // ----------------------------------------------------------------------------------------------------- Private method start  /\*\*  \* 绘制字符串  \*  \* @param g {@link Graphics}画笔  \* @param code 验证码  \*/  private void drawString(Graphics2D g, String code) {  // 指定透明度  if (null != this.textAlpha) {  g.setComposite(this.textAlpha);  }  GraphicsUtil.drawStringColourful(g, code, this.font, this.width, this.height);  }  /\*\*  \* 画随机干扰  \*  \* @param g {@link Graphics2D}  \*/  private void drawInterfere(Graphics2D g) {  final ThreadLocalRandom random = RandomUtil.getRandom();  for (int i = 0; i < this.interfereCount; i++) {  g.setColor(ImgUtil.randomColor(random));  g.drawOval(random.nextInt(width), random.nextInt(height), random.nextInt(height >> 1), random.nextInt(height >> 1));  }  }  public boolean isExpried() {  return LocalDateTime.now().compareTo(expireTime) >= 0;  } } |

core中创建controller

|  |
| --- |
| @RestController  public class ValidateCodeController {  private static final String SESSION\_KEY = "SESSION\_KEY\_IMAGE\_CODE";  private SessionStrategy sessionStrategy = new HttpSessionSessionStrategy();  @GetMapping("/code/image")  public void createCode(HttpServletRequest request, HttpServletResponse response) throws IOException {  ImageCode imageCode = new ImageCode(200, 100, 4, 20,600);  sessionStrategy.setAttribute(new ServletWebRequest(request), SESSION\_KEY, imageCode);  imageCode.write(response.getOutputStream());  }  } |

在Browser中，将conreoller中的方法放行(不需要认证)。

访问/code/image即可看到



### 3.3.2.验证码校验

创建一个过滤器,在security的UsernamePasswordAuthenticationFilter过滤器之前校验验证码.验证码通过调用UsernamePasswordAuthenticationFilter的过滤器,失败的话返回错误信息.

|  |
| --- |
| @Data  public class ValidateCodeFilter extends OncePerRequestFilter {  private AuthenticationFailureHandler authenticationFailureHandler;  private SessionStrategy sessionStrategy = new HttpSessionSessionStrategy();  @Override  protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {  //是登陆的post请求  if (AppConst.LOGIN\_PATH.equals(request.getRequestURI()) && AppConst.METHOD.equalsIgnoreCase(request.getMethod())) {  try {  validate(new ServletWebRequest(request));  } catch (ValidateCodeException e) {  authenticationFailureHandler.onAuthenticationFailure(request, response, e);  return;  }  }  filterChain.doFilter(request, response);  }  private void validate(ServletWebRequest request) throws ServletRequestBindingException {  ImageCode codeSession = (ImageCode) sessionStrategy.getAttribute(request, AppConst.SESSION\_KEY);  String codeRequest= ServletRequestUtils.getStringParameter(request.getRequest(),AppConst.CODE\_NAME,"");  if(codeRequest.isEmpty()){  throw new ValidateCodeException("验证码不能为空");  }  if(codeSession==null){  throw new ValidateCodeException("验证码不存在");  }  if(codeSession.isExpried()){  throw new ValidateCodeException("验证码已过期");  }  if(!codeSession.verify(codeRequest)){  throw new ValidateCodeException("验证码错误");  }  sessionStrategy.removeAttribute(request,AppConst.SESSION\_KEY);  }  } |

自定义异常类ValidateCodeException

|  |
| --- |
| public class ValidateCodeException extends AuthenticationException {  public ValidateCodeException(String msg) {  super(msg);  }  } |

在browser中修改config，将过滤器加入

|  |
| --- |
| …………………………………  ValidateCodeFilter filter=new ValidateCodeFilter(); filter.setAuthenticationFailureHandler(failureHandler); // 指定身份认证的方式 表单登录 http.addFilterBefore(filter, UsernamePasswordAuthenticationFilter.class).formLogin()  //指定登录页面  .loginPage("/authentication/require")  ………………………………… |

### 3.3.3.图形验证码重构

1. 验证码参数配置(图片长宽，验证长度，过滤接口等)



core配置，在properties中添加一个信息的配置信息类ImageCodeProperties(含默认)

|  |
| --- |
| @Data  public class ImageCodeProperties {  private int width=200;  private int height=100;  private int length=4;  private int interfereCount=20;  private int expire=600;  } |

创建ValidateCodeProperties，存放ImageCodeProperties和短信

|  |
| --- |
| @Data  public class ValidateCodeProperties {  private ImageCodeProperties imageCode=new ImageCodeProperties();  } |

在SecurityProperties中加入ValidateCodeProperties

|  |
| --- |
| @Component  @ConfigurationProperties(prefix = "swb.security")  @Data  public class SecurityProperties {  private BrowserProperties browser=new BrowserProperties();  private ValidateCodeProperties validateCode=new ValidateCodeProperties();  } |

以上为应用级配置，在yml中可配置

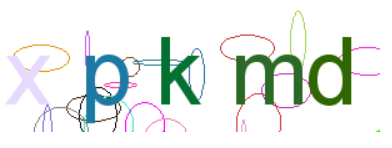
|  |
| --- |
| swb.security.validateCode.imageCode.length: 5 |

设置请求级配置，在生成验证码图片的controller中修改

|  |
| --- |
| @RestController  public class ValidateCodeController {  @Autowired  private SecurityProperties securityProperties;  private SessionStrategy sessionStrategy = new HttpSessionSessionStrategy();  @GetMapping("/code/image")  public void createCode(HttpServletRequest request, HttpServletResponse response) throws IOException {  ImageCodeProperties codeProperties = securityProperties.getValidateCode().getImageCode();  int width= ServletRequestUtils.getIntParameter(request,"width", codeProperties.getWidth());  int height= ServletRequestUtils.getIntParameter(request,"height", codeProperties.getHeight());  ImageCode imageCode = new ImageCode(width, height, codeProperties.getLength(),  codeProperties.getInterfereCount(),  codeProperties.getExpireIn());  sessionStrategy.setAttribute(new ServletWebRequest(request), AppConst.SESSION\_KEY, imageCode);  imageCode.write(response.getOutputStream());  }  } |

在yml中可以指定ImageCode的所有参数，会覆盖默认的参数，在请求中可以包含长宽2个参数，会覆盖yml中配置的参数以及默认的参数。

例如默认验证码数为4，长200，yml中配置验证码数为6，长为300，请求中设置100, 实际结果如下： 请求中不设置参数：

1. 动态校验验证码路径

配置验证验证码的过滤路径，即哪些操作需要验证码(如登录，查询敏感信息(个人信息)等)

在ImageCodeProperties中添加一个属性url。在yml中以’,’隔开处理

|  |
| --- |
| private String url; |

验证码过滤器ValidateCodeFilter实现InitializingBean接口，实现初始化bean后的操作(在属性设置之后做的事情)。

|  |
| --- |
| @Data  public class ValidateCodeFilter extends OncePerRequestFilter implements InitializingBean {  private AuthenticationFailureHandler authenticationFailureHandler;  private SessionStrategy sessionStrategy = new HttpSessionSessionStrategy();  private AntPathMatcher pathMatcher=new AntPathMatcher();  private Set<String> urls = new HashSet<>();  private SecurityProperties securityProperties;  @Override  public void afterPropertiesSet() throws ServletException {  super.afterPropertiesSet();  String url = securityProperties.getValidateCode().getImageCode().getUrl();  if(!StringUtils.isEmpty(url)) {//不为空  // //使用Collections加入配置的url到set  // Collections.addAll(urls, url.split(","));  // 使用Stream流  urls = Stream.of(url.split(",")).collect(Collectors.toSet());  }  //加入登录的请求  urls.add(AppConst.LOGIN\_PATH);  }  @Override  protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {  String uri = request.getRequestURI();  //Stream判断哪些路径是需要过滤的，pathMatcher.match判断\*等通配符  if (urls.stream().anyMatch(s->pathMatcher.match(s,uri))) {  try {  validate(new ServletWebRequest(request));  } catch (ValidateCodeException e) {  authenticationFailureHandler.onAuthenticationFailure(request, response, e);  return;  }  }  filterChain.doFilter(request, response);  }  ………………………………………………. |

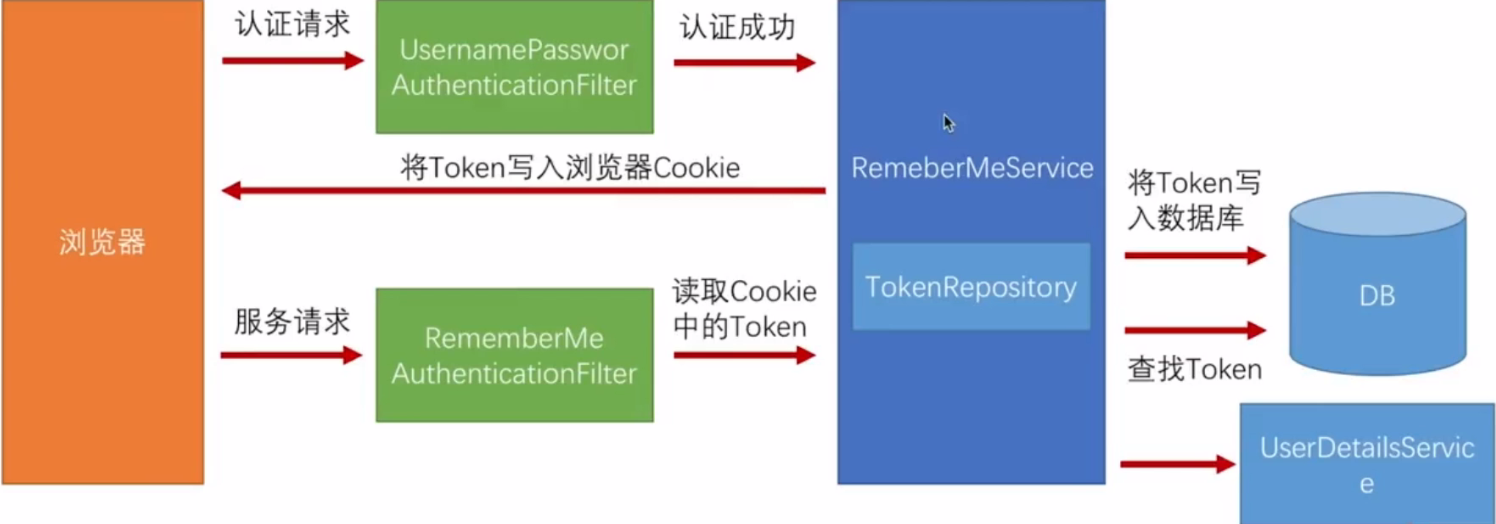
在browser的SecurityConfig中设置filter的属性

|  |
| --- |
| ……………  ValidateCodeFilter filter=new ValidateCodeFilter();  filter.setAuthenticationFailureHandler(failureHandler);  filter.setSecurityProperties(securityProperties);  filter.afterPropertiesSet();  // 指定身份认证的方式 表单登录  http.addFilterBefore(filter, UsernamePasswordAuthenticationFilter.class).formLogin()  ……………………. |

在demo的yml中配置url属性,访问/user没问题,访问/user/12 /user/1提示验证码不能为空

|  |
| --- |
| swb.security.validateCode.imageCode.url: /user/\* |

## 3.4记住我功能原理：



在页面加一个checkbox，name为remember-me

在browser中添加一个PersistentTokenRepository(SecurityConfig类中配置), JdbcTokenRepositoryImpl实在数据库中建立一个一张表，将token存放到数据库中,之后将JdbcTokenRepositoryImpl和userDetailsService以及过期时间(单位s)配置到Secyrity中

|  |
| --- |
| @Autowired private DataSource dataSource; @Autowired @Qualifier("myUserDetailsService") private UserDetailsService userDetailsService;  @Bean public PersistentTokenRepository persistentTokenRepository(){  JdbcTokenRepositoryImpl jdbcTokenRepository=new JdbcTokenRepositoryImpl();  jdbcTokenRepository.setDataSource(dataSource);  return jdbcTokenRepository; }  @Override protected void configure(HttpSecurity http) throws Exception {  http. ……………  //配置记住我 .and().rememberMe() .tokenRepository(persistentTokenRepository()) .tokenValiditySeconds(securityProperties.getBrowser().getRememberMeSeconds()) .userDetailsService(userDetailsService)  ……………………………………  } |

建表语句

|  |
| --- |
| create table persistent\_logins (username varchar(64) not null, series varchar(64) primary key, token varchar(64) not null, last\_used timestamp not null); |

在登录时勾选记住我后，会在本地写入一个Cookie（remember-me）将该cookie放到postman中可以免登录。在数据库中会多一条user的登录记录。

## 3.5短信验证逻辑

1.开发验证码接口 2.校验短信验证码接口并登录 3.重构

### 3.5.1发送短信验证码

core中定义一个发送的接口以及一个默认的实现

|  |
| --- |
| public interface SmsCodeSender {  void send(String mobile,String code);  }  @Slf4j  public class DefaultSmsCodeSender implements SmsCodeSender{  @Override  public void send(String mobile, String code) {  log.info("向手机"+mobile+"发送验证码:"+code);  }  } |

注入bean

|  |
| --- |
| @Configuration  public class ValidateCodeConfig {  /\*\*  \* 配置默认的发送短信实现  \* ConditionalOnMissingBean 当系统中存在已有的name的bean时覆盖当前bean,用于重写或扩展  \* @return  \*/  @Bean  @ConditionalOnMissingBean(name = "smsCodeSender")  public SmsCodeSender smsCodeSender(){  return new DefaultSmsCodeSender();  }  } |

验证码类

|  |
| --- |
| @Data  @NoArgsConstructor  @AllArgsConstructor  public class ValidateCode {  private String code;  private String mobile;  private LocalDateTime expireTime;  public boolean isExpried() {  return LocalDateTime.now().compareTo(expireTime) >= 0;  }  public boolean verify(String codeRequest,String mobileRequest) {  boolean flag=StringUtils.equals(mobile,mobileRequest);  if(flag){  flag=StringUtils.equals(code,codeRequest);  }  return flag;  }  public ValidateCode(String code,String mobile, int expire) {  this.code = code;  this.mobile = mobile;  this.expireTime = LocalDateTime.now().plusSeconds(expire);  }  } |

创建验证码方法

|  |
| --- |
| public ValidateCode generateSmsCode(String mobile){  SmsCodeProperties sms = securityProperties.getValidateCode().getSms();  String code = RandomStringUtils.randomNumeric(sms.getLength());  ValidateCode validateCode=new ValidateCode(code,mobile,sms.getExpireIn());  return validateCode; } |

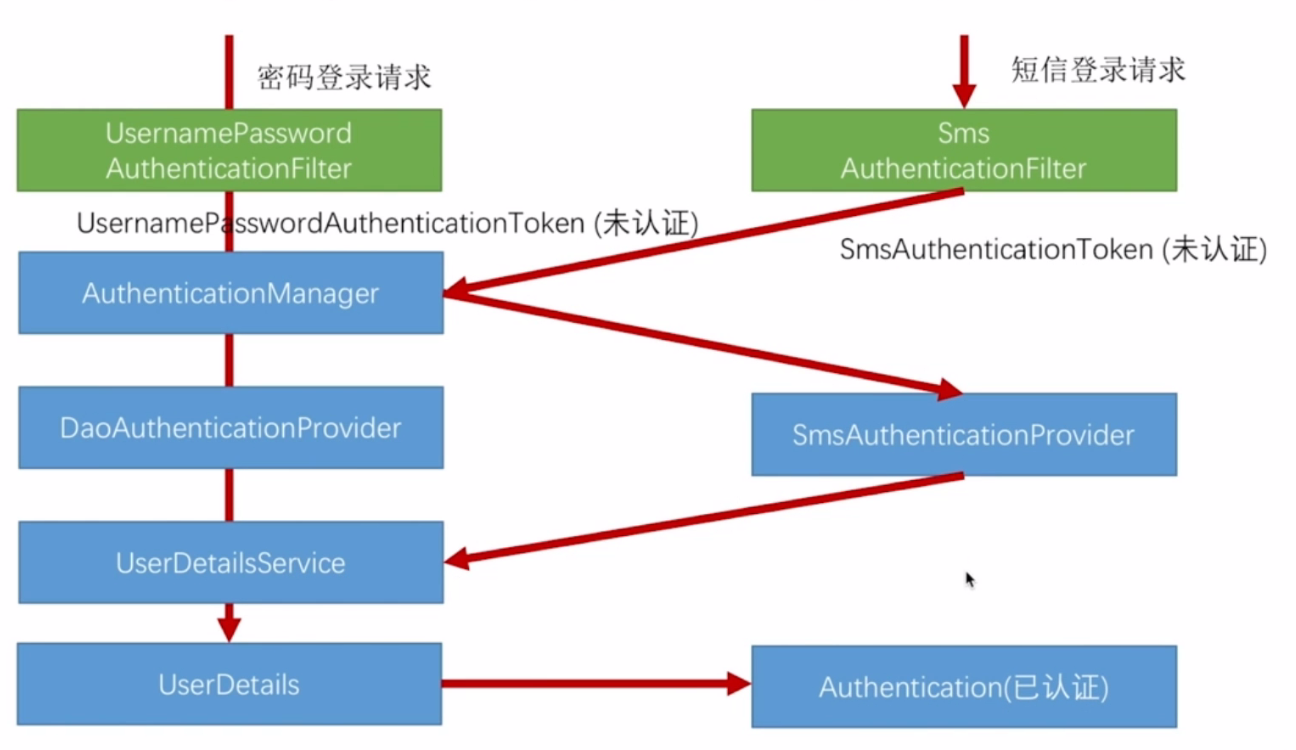
Controller接受手机号并发送

|  |
| --- |
| @GetMapping("/code/sms") public String createSmsCode(HttpServletRequest request, HttpServletResponse response) throws ServletRequestBindingException {  String mobile = ServletRequestUtils.getRequiredStringParameter(request, "mobile");  ValidateCode smsCode = validateCodeGenerator.generateSmsCode(mobile);  sessionStrategy.setAttribute(new ServletWebRequest(request), AppConst.SMS\_CODE, smsCode);  smsCodeSender.send(mobile,smsCode.getCode());  return smsCode.getCode(); } |

browser中配置过滤当前路径 不展示

### 3.5.2短信验证

短信认证流程



以上流程中没有验证短信验证码是否正确，如同图形验证码一样，在sms过滤器之前验证短信验证码。可以重用。

Core中创建，可以提供给browser和app使用

创建SmsCodeAuthenticationToken 可以参考UsernamePasswordAToken

不贴代码了

创建SmsCodeAuthenticationFilter 参考usernamepasswordAFilter

拷过来代码，原先存在username和password，现在只要mobile，修改Token。

创建SmsCodeAuthenticationProvider 实现接口AuthenticationProvider

|  |
| --- |
| @Data  public class SmsCodeAuthenticationProvider implements AuthenticationProvider {  private UserDetailsService userDetailsService;  /\*\*  \* 身份验证逻辑  \*  \* @param authentication  \* @return  \* @throws AuthenticationException  \*/  @Override  public Authentication authenticate(Authentication authentication) throws AuthenticationException {  SmsCodeAuthenticationToken authenticationToken= (SmsCodeAuthenticationToken) authentication;  String mobile = (String) authenticationToken.getPrincipal();  //将电话传入方法获取用户信息  UserDetails user=userDetailsService.loadUserByUsername("逢魔Zi-O");  if (user==null){  throw new InternalAuthenticationServiceException("无法获取用户信息");  }  SmsCodeAuthenticationToken resultAuthenticationToken=new SmsCodeAuthenticationToken(user,user.getAuthorities());  resultAuthenticationToken.setDetails(resultAuthenticationToken.getDetails());  return resultAuthenticationToken;  }  /\*\*  \* AuthenticationManager判断执行那个provider  \* @param authentication  \* @return  \*/  @Override  public boolean supports(Class<?> authentication) {  //判断传进来的是否是SmsCodeAuthenticationToken类型  //SmsCodeAuthenticationToken 继承 AbstractAuthenticationToken 实现 Authentication  return SmsCodeAuthenticationToken.class.isAssignableFrom(authentication);  }  } |

创建短信验证码逻辑 参考图形验证码 略过，需要手机号加验证码验证

配置短信验证码，在core中配置，方便browser和app使用

|  |
| --- |
| @Component  public class SmsCodeAuthenticationSecurityConfig extends SecurityConfigurerAdapter<DefaultSecurityFilterChain, HttpSecurity> {  @Autowired  @Qualifier(value = "swbSecurityAuthenticationSuccessHandler")  private AuthenticationSuccessHandler successHandler;  @Autowired  @Qualifier(value = "swbSecurityAuthenticationFailureHandler")  private AuthenticationFailureHandler failureHandler;  @Autowired  @Qualifier(value = "myUserDetailsService")  private UserDetailsService userDetailsService;  @Override  public void configure(HttpSecurity http) throws Exception {  //创建过滤器filter  SmsCodeAuthenticationFilter filter = new SmsCodeAuthenticationFilter();  //filter设置AuthenticationManager  filter.setAuthenticationManager(http.getSharedObject(AuthenticationManager.class));  //filter设置成功和失败的处理器 Autowired注入 在browser中写过，app中可以重写  filter.setAuthenticationSuccessHandler(successHandler);  filter.setAuthenticationFailureHandler(failureHandler);  //配置Provider  SmsCodeAuthenticationProvider provider = new SmsCodeAuthenticationProvider();  provider.setUserDetailsService(userDetailsService);  http.authenticationProvider(provider)  .addFilterAfter(filter, UsernamePasswordAuthenticationFilter.class);  }  } |

配置短信验证码的过滤器 在browser的BrowserSecurityConfig中

|  |
| --- |
| ……………………………  // configure方法中  SmsCodeFilter smsCodeFilter = new SmsCodeFilter(); smsCodeFilter.setAuthenticationFailureHandler(failureHandler); smsCodeFilter.setSecurityProperties(securityProperties); smsCodeFilter.afterPropertiesSet();  http.addFilterBefore(smsCodeFilter, UsernamePasswordAuthenticationFilter.class)  .addFilterBefore(validateCodeFilter, UsernamePasswordAuthenticationFilter.class)  .formLogin()  ………………………………… |

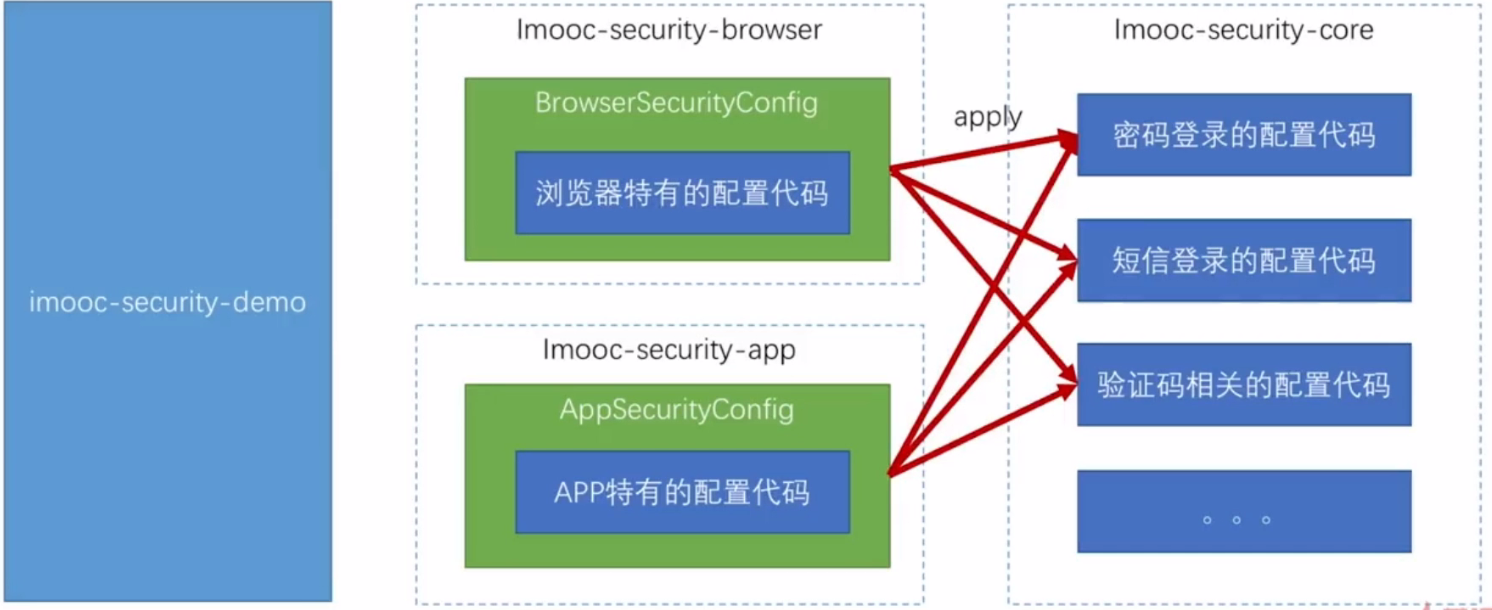
在BrowserSecurityConfig中应用core中的配置SmsCodeAuthenticationSecurityConfig

|  |
| --- |
| Autowired注入  ……………….  // configure方法最后添加配置  .and().csrf().disable() .apply(smsCodeAuthenticationSecurityConfig); |

验证短信和密码登录均可

重构

1.图形验证码和短信验证码存在重复代码，需要修改(包括类内逻辑和配置)



BrowserSecurityConfig配置。

图形验证码和短信验证码校验器合并 core中，可供browser和app使用

Rember-me仅供browser使用

提取部分公用方法以及变量

跳过

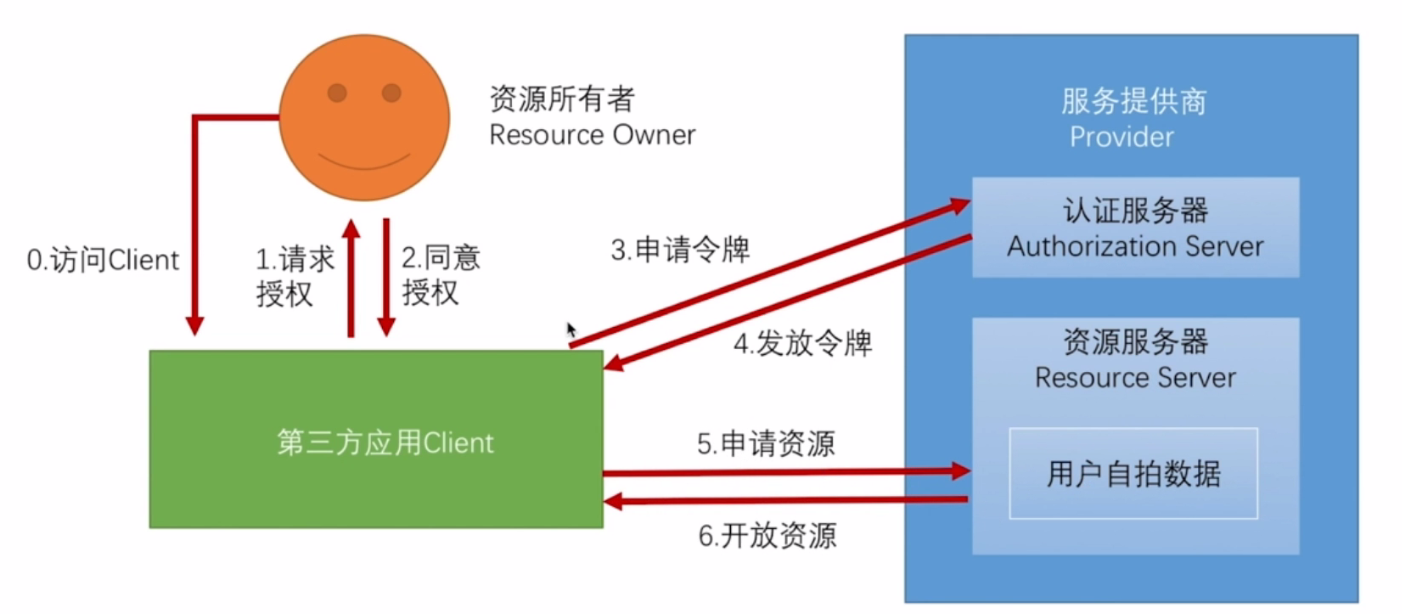
## 3.6总结

在服务中加了一条过滤器链，在过滤器链中添加了一些校验，可以自己配置新的校验

在通过校验后判断是否存在权限

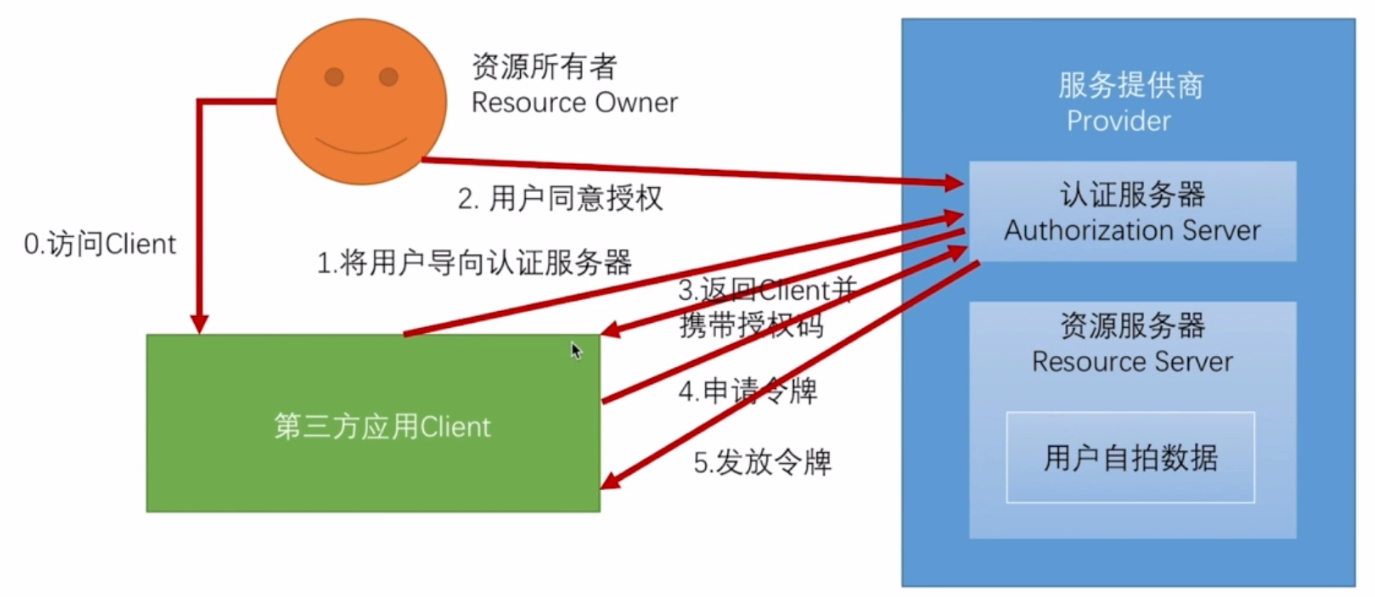
# 4.第三方登陆

## 4.1 OAuth协议



授权模式： 授权码，简化，密码，客户端。

授权码模式：



# 5.session管理

## 5.1单机session管理

Spring Boot2.2.5配置session过期时间

|  |
| --- |
| server.servlet.session.timeout=10S |

获取该值后，将转化为分钟单位，并与1比较，取最小值。也就是最少1分钟有效时间。

session过期后，访问接口提示，这与没有登陆访问接口的提示信息相同。不合理。

|  |
| --- |
| {"content":"访问信息需要身份认证"} |

在BrowserSecurityConfig配置HttpSecurity，指定session失效后的地址。并配置该路径不需要身份认证。

|  |
| --- |
| http.apply(smsCodeAuthenticationSecurityConfig)  ……………………  .and()  .sessionManagement()  .invalidSessionUrl(SecurityConstants.DEFAULT\_SESSION\_INVALID\_URL)  .and()  // 开始请求权限配置  .authorizeRequests()  //当前路径不需要认证  .antMatchers(…………………..  SecurityConstants.DEFAULT\_SESSION\_INVALID\_URL,  ……………………………… |
| 失效的url处理  @GetMapping("/session/invalid")  @ResponseStatus(HttpStatus.UNAUTHORIZED)  public SimpleResponse sessionInvalid(){  String message="session失效，用户已过期";  return new SimpleResponse(message);  } |

.invalidSessionUrl(URL)失效后的跳转路径(2者取一个)

.invalidSessionStrategy(InvalidSessionStrategy)失效后的处理

并发控制:

指定数量，并配置处理器，处理用户被踢掉后的逻辑。

|  |
| --- |
| .sessionManagement()  .invalidSessionUrl(SecurityConstants.DEFAULT\_SESSION\_INVALID\_URL)  //最大数量  .maximumSessions(1)  .expiredSessionStrategy(new BrowserSecuritySessionExpiredStrategy())  .and()  .and()  //开始请求权限配置  .authorizeRequests() |

BrowserSecuritySessionExpiredStrategy处理失败逻辑

|  |
| --- |
| @Slf4j  public class BrowserSecuritySessionExpiredStrategy implements SessionInformationExpiredStrategy {  @Override  public void onExpiredSessionDetected(SessionInformationExpiredEvent event) throws IOException, ServletException {  log.info("开始处理同一个用户不同机器多次登录逻辑");  Object principal = event.getSessionInformation().getPrincipal();  log.info(JSONUtil.toJsonStr(principal));  log.info("逻辑处理结束");  HttpServletResponse response = event.getResponse();  response.setContentType("application/json;charset=utf-8");  response.getWriter().write(  JSONUtil.toJsonStr(new SimpleResponse("你的账号在别的地方登录，该设备已下线")));  }  } |

也可以指定超出最大登录数量则无法访问

|  |
| --- |
| .and()  .sessionManagement()  .invalidSessionUrl(SecurityConstants.DEFAULT\_SESSION\_INVALID\_URL)  //最大数量  .maximumSessions(1)  //达到最大登录数时，不允许登录  .maxSessionsPreventsLogin(true)  .expiredSessionStrategy(new BrowserSecuritySessionExpiredStrategy())  .and().and() |

## 5.2集群session管理

加入Spring Session 使用redis管理

在spring boot2.x中，修改browser的依赖

|  |
| --- |
| compile "org.springframework.session:spring-session-core:2.3.0.RELEASE"  compile "org.springframework.session:spring-session-data-redis:2.3.0.RELEASE" |

配置文件中指定session的存放方式为redis，并配置redis的信息

|  |
| --- |
| spring:  session:  store-type: redis  redis:  host: 127.0.0.1  port: 6379  password: password |

分别启动2个端口8081，8082

在浏览器中访问8081并登录，访问/security/me 返回用户信息。

在同一个浏览器访问8082 /security/me 不需要登录，同意返回用户信息。

关闭浏览器后，重新登录用户，提示超过登录次数限制,即之前限制的1次登录。

## 5.3退出

在Security的配置文件中，HttpSecurity设置logout的退出逻辑

|  |
| --- |
| http.apply(smsCodeAuthenticationSecurityConfig)  .and()  //退出  .logout()  //指定退出的路径  .logoutUrl("/signOut")  //退出成功后跳转的url  .logoutSuccessUrl(SecurityConstants.DEFAULT\_LOGIN\_PAGE\_URL)  //自定义SuccessHandle指定退出成功的处理  .logoutSuccessHandler(new BrowserLogoutSuccessHandler())  .and() // 开始请求权限配置  .authorizeRequests() |

自定义SuccessHandle ，SuccessHandle与url只能生效一个

|  |
| --- |
| @Slf4j  public class BrowserLogoutSuccessHandler implements LogoutSuccessHandler {  @Override  public void onLogoutSuccess(HttpServletRequest request, HttpServletResponse response, Authentication authentication) throws IOException, ServletException {  log.info("退出登录");  log.info(JSONUtil.toJsonStr(authentication));  response.getWriter().write(JSONUtil.toJsonStr(new SimpleResponse("退出登录")));  }  } |

登录成功后访问/signOut 退出登录，但是由于之前设置了最大登录数为1,导致无法登录。暂时没找到解决方案。