# Spring Security

——开发企业级认证与授权

目录

[Spring Security 1](#_Toc533862989)

[——开发企业级认证与授权 1](#_Toc533862990)

[1. 开发环境与配置（Maven聚合工程） 2](#_Toc533862991)

[1-1. 开发环境介绍 2](#_Toc533862992)

[1-2. 工程结构介绍 2](#_Toc533862993)

[1-3. Hello Spring Security 6](#_Toc533862994)

[2.使用springMVC开发RESTful API 18](#_Toc533862995)

[2-1.RESTful简介 18](#_Toc533862996)

[2-2.查询请求 19](#_Toc533862997)

[2-3.用户详情请求 26](#_Toc533862998)

[2-4.用户创建请求 29](#_Toc533862999)

[2-5.修改和删除请求 36](#_Toc533863000)

[2-6.服务异常处理 38](#_Toc533863001)

[2-7.使用filter和interceptor拦截RESTful服务 45](#_Toc533863002)

[2-8.使用切面拦截RESTful服务 49](#_Toc533863003)

[2-9.使用rest方式处理文件服务 50](#_Toc533863004)

[2-10.使用多线程提高rest服务性能 54](#_Toc533863005)

[2-11.使用swagger自动生成文档 61](#_Toc533863006)

[2-12.使用WireMock伪造rest服务 64](#_Toc533863007)

[3. 使用spring security开发基于表单的登录 66](#_Toc533863008)

[3-1.简介 66](#_Toc533863009)

[3-2.Spring Security 基本原理 67](#_Toc533863010)

[3-3.自定义用户认证逻辑 71](#_Toc533863011)

[3-4.个性化用户认证流程 75](#_Toc533863012)

[3-5.认证流程源码详解 84](#_Toc533863013)

[3-6.图片验证码 89](#_Toc533863014)

[3-7.图片验证码重构 94](#_Toc533863015)

[3-8.添加记住我功能 95](#_Toc533863016)

[3-9.短信验证码接口开发 95](#_Toc533863017)

[3-10.短信登录开发 95](#_Toc533863018)

[3-11.短信登录配置及重构 95](#_Toc533863019)

[3-12.小结 95](#_Toc533863020)

[4. 使用Spring Social开发第三方登录 96](#_Toc533863021)

[4-1.OAuth协议简介 96](#_Toc533863022)

[4-2.SpringSocial简介 96](#_Toc533863023)

[4-3.开发QQ登录 96](#_Toc533863024)

[4-4.处理注册逻辑 96](#_Toc533863025)

[4-5.开发微信登录 96](#_Toc533863026)

[4-6.绑定和解绑处理 96](#_Toc533863027)

[4-7.单机Session管理 96](#_Toc533863028)

[4-8.集群Session管理 96](#_Toc533863029)

[4-9.退出登录 96](#_Toc533863030)

[5. 使用Spring Security Oauth开发APP认证框架 96](#_Toc533863031)

[5-1.Spirng Security OAuth简介 96](#_Toc533863032)

[5-2.实现标准的OAuth服务提供商 96](#_Toc533863033)

[5-3.Spirng Security OAuth 核心源码解析 96](#_Toc533863034)

[5-4.重构用户名密码登录 96](#_Toc533863035)

[5-5.重构短信登录 96](#_Toc533863036)

[5-6.重构社交登录 96](#_Toc533863037)

[5-7.重构注册逻辑 96](#_Toc533863038)

[5-8.令牌配置 96](#_Toc533863039)

[5-9.使用JWT替换默认令牌 96](#_Toc533863040)

[5-10.基于JWT实现SSO单点登录 96](#_Toc533863041)

[6. 使用Spirng Security 控制授权 97](#_Toc533863042)

[6-1.Spring Security 授权简介 97](#_Toc533863043)

[6-2.Spinrg Security 源码解析 97](#_Toc533863044)

[6-3.权限表达式 97](#_Toc533863045)

[6-4.基于数据库RBAC数据模型控制权限 97](#_Toc533863046)

[总结 97](#_Toc533863047)

## 开发环境与配置（Maven聚合工程）

### 开发环境介绍

本文旨在开发可复用的安全模块，故使用以下开发环境配合开发。

1.开发工具: RapidClipse 4.0

2.Java版本:1.8

3.Maven版本:3.3.9

代码使用SpringBoot开发。

### 工程结构介绍

本工程使用Maven聚合工程结构开发



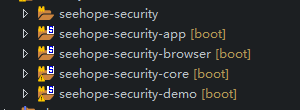
工程分为core，app，browser，demo 4个模块。

Core：作为工程的核心模块，分别被browser以及app模块引用，在core模块中存放通用代码以及通用依赖。

Browser：该模块用来描述基于浏览器端的应用服务，引用core模块，被demo模块引用。

App：该模块用来描述基于移动端的应用服务，引用core模块。

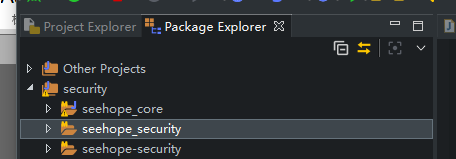
Demo：改模块引用browser模块，主要用来测试代码。



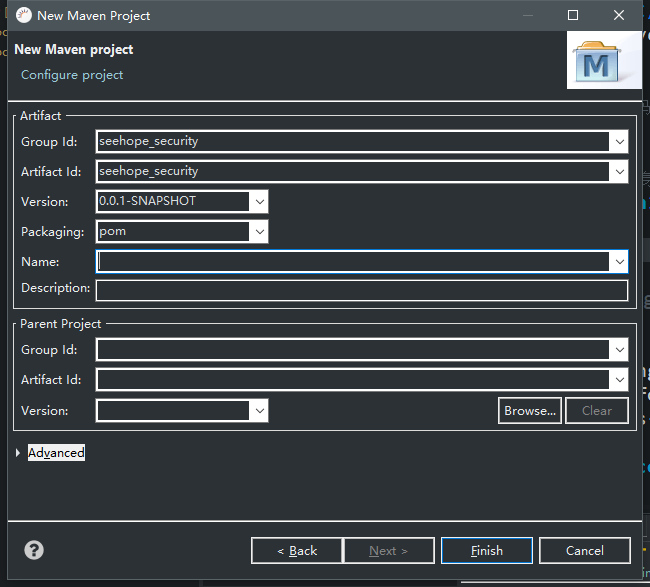
4个模块构成了整个Seehope Security工程。

创建步骤：

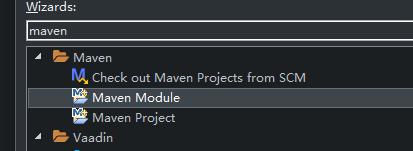
如果在创建的过程中新创建的工程无显示，请在eclipse的window->show view中选择Package Explorer 视窗，并使用该视窗查看工程 。

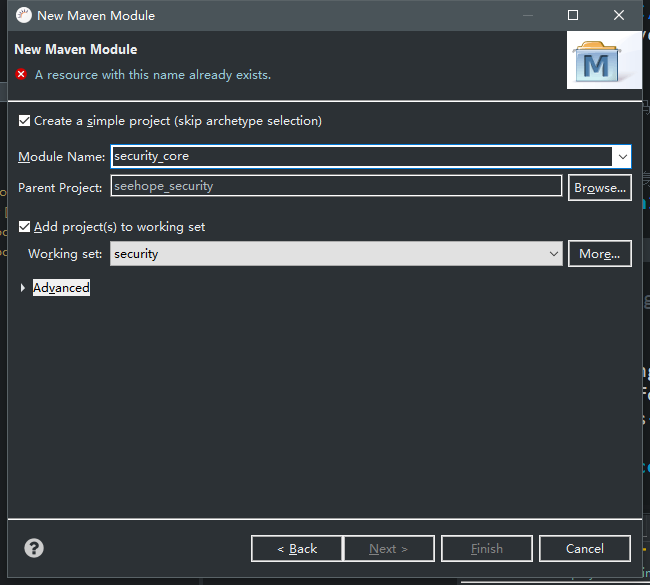


1. 创建MavenProject Seehope Security 归档成POM



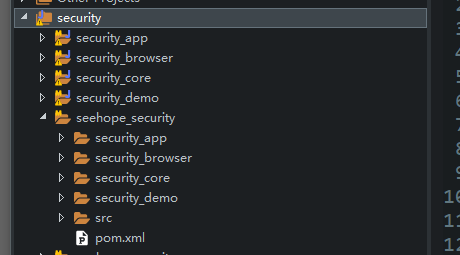
1. 选中已创建的seehope security ，创建Maven Module seehope-core模块





1. 重复以上步骤，选中seehope\_security 工程，创建borwser，app以及demo 模块

最终结果



在创建工程的过程中并没有体现出browser以及app模块依赖core的关系而是在之后两个模块的POM文件配置中依赖core模块。

使用以上方式创建工程之后，也就是选中seehope security（POM）工程之后创建Maven Module，那么打开POM工程的Pom文件中，我们可以看到自动添加了

<modules>

  <module>security\_core</module>

  <module>security\_browser</module>

  <module>security\_app</module>

  <module>security\_demo</module>

</modules>

这样一段代码，表示seehope security工程中包含这4个模块。

以上，工程环境创建完毕。

### Hello Spring Security

本节先配置好工程依赖，创建hello 工程。

上一节创建好了Maven聚合工程，本节配置好依赖环境以及工程属性。

* 1. 首先，统一不同模块之间的版本号，找到seehope security（总工程）下的pom.xml 文件

先定义Maven变量

    <!--用以整合子项目，本身没有任何java代码-->

    <properties>

        <!--定义项目版本号 整个项目，包括本身以及其子项目-->

        <seehope.security.version>1.0.0-SNAPSHOT</seehope.security.version>

    </properties>

然后在每个模块的pom.xml文件中，引用该版本变量，包括总工程的pom.xml文件本身

<version>${seehope.security.version}</version>

* 1. 在总工程的pom.xml文件中，引入jar包版本管理器

1. <dependencyManagement>
2. <!--引入spring io 以及 spirng cloud 项目进行依赖管理 -->
3. <dependencies>
4. <dependency>
5. <groupId>io.spring.platform</groupId>
6. <artifactId>platform-bom</artifactId>
7. <version>Brussels-SR4</version>
8. <type>pom</type>
9. <scope>import</scope>
10. </dependency>
11. <dependency>
12. <groupId>org.springframework.cloud</groupId>
13. <artifactId>spring-cloud-dependencies</artifactId>
14. <version>Dalston.SR2</version>
15. <type>pom</type>
16. <scope>import</scope>
17. </dependency>
18. </dependencies>
19. </dependencyManagement>

引用spring io 以及spring cloud 的jar包版本管理器来管理该工程的jar包版本管理器来管理本工程的jar包版本，当我们的工程使用的jar包在这两个spring的工程也用使用的情况下，我们不需要写jar包的版本号，最大程度避免了jar包版本冲突问题。

③ 引入springboot maven 编译插件 配置工程环境

<!--设置项目的编译以及运行版本 -->

<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>

至此，总工程pom.xml文件配置完毕

<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>net.seehope.security</groupId>

    <artifactId>seehope-security</artifactId>

    <version>${seehope.security.version}</version>

    <packaging>pom</packaging>

    <!--用以整合子项目，本身没有任何java代码-->

    <properties>

        <!--定义项目版本号 整个项目，包括本身以及其子项目-->

        <seehope.security.version>1.0.0-SNAPSHOT</seehope.security.version>

    </properties>

    <dependencyManagement>

        <!--引入spring io 以及 spirng cloud 项目进行依赖管理 -->

        <dependencies>

            <dependency>

                <groupId>io.spring.platform</groupId>

                <artifactId>platform-bom</artifactId>

                <version>Brussels-SR4</version>

                <type>pom</type>

                <scope>import</scope>

            </dependency>

            <dependency>

                <groupId>org.springframework.cloud</groupId>

                <artifactId>spring-cloud-dependencies</artifactId>

                <version>Dalston.SR2</version>

                <type>pom</type>

                <scope>import</scope>

            </dependency>

        </dependencies>

    </dependencyManagement>

    <!--设置项目的编译以及运行版本 -->

    <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>

    <!--整合子项目 创建过程中自动生成-->

    <modules>

        <module>../seehope-security-app</module>

        <module>../seehope-security-browser</module>

        <module>../seehope-security-core</module>

        <module>../seehope-security-demo</module>

    </modules>

</project>

④ 配置core 模块 pom.xml文件 （记得在第一步统一版本号）

该模块被browser以及app模块所引用，所以我们可以吧整个工程通用的依赖包放在core里面，根据maven依赖体系，如果browser模块引用core，那么browser也会引用core所依赖的所有依赖。

本工程旨在开发通用的权限管理模块，数据库操作并不是本工程描述重点，故没有引入数据库依赖。下文是所有核心依赖，除此之外，不同的模块可以根据不同的需要单独引入特殊的依赖，比如在browser环境中有Session环境，可以引入spring session来管理session对象。

下文是所有core模块中的pom.xml中的所有内容

<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>seehope-security-core</artifactId>

    <!--核心源码包，位于第二层 引入项目整体依赖环境-->

    <!-- 创建工程时自动生成 -->

    <parent>

        <groupId>net.seehope.security</groupId>

        <artifactId>seehope-security</artifactId>

        <version>${seehope.security.version}</version>

        <relativePath>../seehope-security</relativePath>

    </parent>

    <dependencies>

        <!--springbootAOP -->

        <dependency>

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

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

        </dependency>

        <dependency>

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

            <artifactId>spring-boot-starter-jta-narayana</artifactId>

        </dependency>

        <!--认证模块 -->

        <dependency>

            <groupId>org.springframework.cloud</groupId>

            <artifactId>spring-cloud-starter-oauth2</artifactId>

        </dependency>

        <!--缓存模块 -->

        <dependency>

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

            <artifactId>spring-boot-starter-data-redis</artifactId>

        </dependency>

        <!--数据库操作模块 -->

        <dependency>

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

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

        </dependency>

        <!--数据库连接驱动 -->

        <dependency>

            <groupId>mysql</groupId>

            <artifactId>mysql-connector-java</artifactId>

        </dependency>

        <!--spring 测试框架，方便测试RESTful风格API -->

        <dependency>

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

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

        </dependency>

        <!--社交模块， 第三方授权认证 开始 -->

        <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>

        <!--社交模块， 第三方授权认证 结束 -->

        <!--常用工具模块，字符串操作等 -->

        <dependency>

            <groupId>commons-lang</groupId>

            <artifactId>commons-lang</artifactId>

        </dependency>

        <!--集合操作相关 -->

        <dependency>

            <groupId>commons-collections</groupId>

            <artifactId>commons-collections</artifactId>

        </dependency>

        <!--对象操作相关-->

        <dependency>

            <groupId>commons-beanutils</groupId>

            <artifactId>commons-beanutils</artifactId>

        </dependency>

        <!--io相关操作 -->

        <dependency>

            <groupId>commons-io</groupId>

            <artifactId>commons-io</artifactId>

        </dependency>

        <!-- Google Json -->

        <!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->

        <dependency>

            <groupId>com.google.code.gson</groupId>

            <artifactId>gson</artifactId>

        </dependency>

        <!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger2 -->

        <dependency>

            <groupId>io.springfox</groupId>

            <artifactId>springfox-swagger2</artifactId>

            <version>2.8.0</version>

        </dependency>

        <!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger-ui -->

        <dependency>

            <groupId>io.springfox</groupId>

            <artifactId>springfox-swagger-ui</artifactId>

            <version>2.8.0</version>

        </dependency>

        <dependency>

            <groupId>com.github.tomakehurst</groupId>

            <artifactId>wiremock</artifactId>

        </dependency>

<!-- https://mvnrepository.com/artifact/org.apache.httpcomponents/httpclient -->

<dependency>

<groupId>org.apache.httpcomponents</groupId>

<artifactId>httpclient</artifactId>

</dependency>

    </dependencies>

</project>

以上，core中的pom.xml编写完毕。

⑤ 配置browser 以及 app 和 demo模块的 pom.xml 文件

首先还是在步骤①中的版本号配置，其次，引入 对 core 模块的依赖

Brower – pom.xml

<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>seehope-security-browser</artifactId>

    <!--浏览器端服务模块 -->

    <parent>

        <groupId>net.seehope.security</groupId>

        <artifactId>seehope-security</artifactId>

        <version>${seehope.security.version}</version>

        <relativePath>../seehope-security</relativePath>

    </parent>

    <dependencies>

        <!--依赖于核心包 -->

        <dependency>

            <groupId>net.seehope.security</groupId>

            <artifactId>seehope-security-core</artifactId>

            <version>${seehope.security.version}</version>

        </dependency>

        <!--session 创建分发以及管理模块-->

        <dependency>

            <groupId>org.springframework.session</groupId>

            <artifactId>spring-session</artifactId>

        </dependency>

        <!--session管理-->

        <dependency>

            <groupId>commons-httpclient</groupId>

            <artifactId>commons-httpclient</artifactId>

        </dependency>

    </dependencies>

</project>

App – pom.xml

<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>net.seehope</groupId>

    <artifactId>seehope-security-app</artifactId>

    <!--移动端服务模块-->

    <parent>

        <groupId>net.seehope.security</groupId>

        <artifactId>seehope-security</artifactId>

        <version>${seehope.security.version}</version>

        <relativePath>../seehope-security</relativePath>

    </parent>

    <dependencies>

        <!--依赖于核心包-->

        <dependency>

            <groupId>net.seehope.security</groupId>

            <artifactId>seehope-security-core</artifactId>

            <version>${seehope.security.version}</version>

        </dependency>

    </dependencies>

</project>

Demo – pom.xml

<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>seehope-security-demo</artifactId>

    <!--示例模块-->

    <parent>

        <groupId>net.seehope.security</groupId>

        <artifactId>seehope-security</artifactId>

        <version>${seehope.security.version}</version>

        <relativePath>../seehope-security</relativePath>

    </parent>

    <dependencies>

        <dependency>

            <groupId>net.seehope.security</groupId>

            <artifactId>seehope-security-browser</artifactId>

            <version>${seehope.security.version}</version>

        </dependency>

    </dependencies>

    <build>

        <plugins>

            <plugin>

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

                <artifactId>spring-boot-maven-plugin</artifactId>

                <version>1.3.3.RELEASE</version>

                <executions>

                    <execution>

                        <goals>

                            <!--重新打包，按照springboot打包 -->

                            <goal>repackage</goal>

                        </goals>

                    </execution>

                </executions>

            </plugin>

        </plugins>

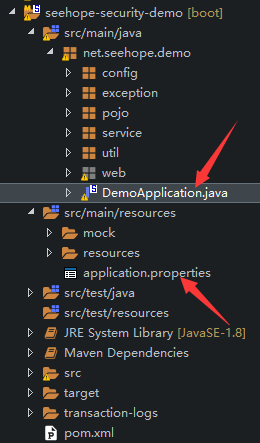
    </build>

</project>

至此，所有模块POM文件配置完毕。

⑥ 创建并启动hello spring security工程

在demo 模块中创建包结构以及主程序DemoApplication.class，在resources目录中创建application.properties配置文件。



编辑DemoApplication.class

package net.seehope.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.annotation.ComponentScan;

import springfox.documentation.swagger2.annotations.EnableSwagger2;

/\*\*

\* @author : MT

\* @date 创建时间：2018年3月22日 下午8:09:24

\* @version 1.0

\*/

@SpringBootApplication

@ComponentScan("net.seehope")

public class DemoApplication {

    public static void main(String[] args) {

        SpringApplication.run(DemoApplication.class, args);

    }

}

编辑配置文件application.properties

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/spring\_security?useUnicode=yes&characterEncoding=utf-8

spring.datasource.username=root

spring.datasource.password=123456

#在不使用session管理的时候先关闭该功能

spring.session.store-type=none

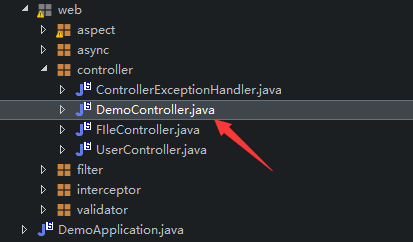
#在不适用security的时候先关闭该功能

security.basic.enabled=false

server.port=8080

在配置文件中，先关闭session以及security的相关功能，若不关闭security功能，则在访问当前工程中的所有资源都将需要用户名以及密码。若不关闭session，则可能在启动时报错。

在web.controller 包中创建DemoController.java进行hello测试



package net.seehope.demo.web.controller;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.GetMapping;

/\*\*

\* @author : MT

\* @date 创建时间：2018年12月24日 下午3:28:55

\* @version 1.0

\*/

@Controller

public class DemoController {

    private static final Logger log = LoggerFactory.getLogger(DemoController.class);

    @GetMapping("/hello")

    public void sayHello() {

        log.info(" hello spring security ~~~");

    }

}

之后选中DemoApplication，启动该主方法，打开浏览器。访问<http://localhost:8080/hello>即可看到控制台的日志输出信息。

至此。Hello工程创建完毕。

说明：在不同模块中，Spring容器并不是共享的关系，换句话说，每个模块需要有独立的启动文件，以及配置文件，用来生成Srping容器，

但是，因为在browser以及app模块中引入了core模块，在demo中引入了browser模块，所以，通过注解扫描包的方式，可以直接在demo中引用在core中写好并注解好的代码。换句话说，如果配置文件都是使用java文件+注解的方式配置的，那么只要在该模块扫描到此配置，那么此配置中的功能就会被注入到当前模块，所以在maven聚合工程中，包名和注解扫描包功能非常重要，在本工程，demo模块中，我们扫描所有net.seehop开头的包。

@ComponentScan("net.seehope")

可以试试在core模块中写一个创建一个net.seehope开头的包，创建一个HelloBean，并使用注解注册到spring容器中，在demo工程中扫描net.seehop包，引用该bean测试一下。

## 2.使用springMVC开发RESTful API

### 2-1.RESTful简介

RESTful一种软件架构风格、设计风格，而不是标准，只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。本章主要讲述使用Spring MVC开发RESTful风格的API。

|  |  |  |  |
| --- | --- | --- | --- |
| **行为** | **传统API** | **RESTful API** | **方法** |
| 查询 | /user/query?name=mt | /user | GET |
| 详情 | /user/getInfo?id=1 | /user/1 | GET |
| 创建 | /user/create?name=mt | /user | POST |
| 修改 | /user/update?id=1&name=mt | /user/1 | PUT |
| 删除 | /user/delete?id=1 | /user/1 | DELETE |

RESTful风格的API有如下几个特点：

* 使用URL描述资源
* 使用HTTP方法描述行为，使用HTTP状态码来表示不同的结果
* 使用JSON进行数据交互
* RESTful只是一种风格，并不是一种强制的标准

2-1.1常用注解复习

这里介绍几个常用的注解：在第3小结会有比较详细的介绍

* @RestController标明此Controller提供RESTful API
* @RequestMapping及其变体（@GetMapping、PostMapping等），映射HTTP请求到Java方法
* @RequestParam映射请求参数到Java方法的参数
* @PathVariable映射URL片段到Java方法的参数
* @PageableDefault指定默认分页参数
* @JsonView按照指定方式序列化Java对象

### 2-2.查询请求

2.2.1编写第一个RESTfulAPI（针对RestfAPI的测试用例）

在正式开发API之前，先来介绍一种编码规范——**测试驱动开发（TDD）**的编程。

在产品的研发过程中，测试一项至关重要。不论是软件还是硬件。

软件的测试先行，在研发过程中，就做到质量的保证，因为在出现Bug的时候，容易定位Bug,而且即使是在客户端出现Bug，也能够轻易的找到Bug出现的原因。

硬件的测试先行，即保证了研发过程中，方便及时发现出现问题的原因。同时，也为以后的批量生产做好铺垫。当进行批量生产的时候，如果测试步骤过于繁琐，会造成生产效率的低下，直接影响了产品的生产进度。

测试先行，我想更多的是指在开发过程中要考虑到测试。准确来说就是，在开发的过程中，应该知道怎样的测试结果，能够证明我们开发出来的产品是正确的。

以软件来说，就是我们的输入应该获得怎样的输出，输出的结果，在我们的软件开发之前就是已经确定的。而且，开发的过程中，应该考虑如何设计与实现，能够在检测输出的时候，更方便、更准确。

于硬件来说，就是我们的输入会得到怎样的输出，在合适的地方留下各个测量点，在检测的时候，能够方便的在每一个需要测试的节点，进行测试。而每一个测量点的预留，也是为了以后的大规模生产中，制作测试硬件的测试平台更方便。如果，没有留下相应的测量点，一来回提高测试的复杂度，二来降低生产效率。

无论软硬件，测试的效率，都会直接影响产品的开发效率。

可能硬件的历史长久，工程概念也更深入。所以虽然硬件的开发过程中，都要留下测试的测量点。虽然这并没有形成理论，但是作为硬件开发的经验或者说专业做法，却已经在产品的研发与制作中深入人心。

测试本质也是设计过程中的一环。考虑测试，也是为了进行更好的设计。测试不能解决质量问题，还要依靠良好的设计。

测试驱动开发是敏捷开发中的一项核心实践和技术，也是一种设计方法论。TDD的原理是在开发功能代码之前，先编写单元测试用例代码，测试代码确定需要编写什么产品代码。TDD虽是敏捷方法的核心实践，但不只适用于XP（Extreme Programming），同样可以适用于其他开发方法和过程。

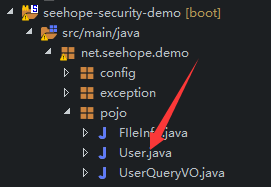
TDD的基本思路就是通过测试来推动整个开发的进行，但测试驱动开发并不只是单纯的测试工作，而是把需求分析，设计，质量控制量化的过程。

TDD的重要目的不仅仅是测试软件，测试工作保证代码质量仅仅是其中一部分，而且是在开发过程中帮助客户和程序员去除模棱两可的需求。TDD首先考虑使用需求（对象、功能、过程、接口等），主要是编写测试用例框架对功能的过程和接口进行设计，而测试框架可以持续进行验证。

综上所述，先编写测试用例，一方面是依据测试驱动开发的编程方式，一方面也是因为RESTful风格的API需要浏览器通过POST,DELETE,PUT等方式发送json形式的数据，而在浏览器上发送以上Method比较麻烦，后期可以借助相应工具，比如浏览器插件 RESTClient，

或者在代码中启用Swagger插件等。

目的是测试获取所有用户的API，依据RESTful风格，即为测试 GET 方式的 /user 请求 首先,构建User实体类。



/\*\*

\* @author : MT

\* @date 创建时间：2018年3月22日 下午10:59:57

\* @version 1.0

\*/

public class User {

    private String id;

    private String username;

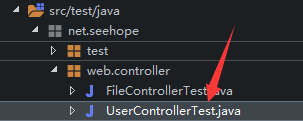
    private Date birthday;

    private String password;

}

用户对象非常简单，包含，String类型的用户ID，用户名，生日，以及密码

然后，在测试目录下构建UserControllerTest.java 测试用例



package net.seehope.web.controller;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;

import java.io.UnsupportedEncodingException;

import java.time.LocalDateTime;

import java.time.ZoneId;

import java.util.Date;

import org.junit.Before;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.http.MediaType;

import org.springframework.test.context.junit4.SpringRunner;

import org.springframework.test.web.servlet.MockMvc;

import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

import org.springframework.test.web.servlet.result.MockMvcResultMatchers;

import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import org.springframework.web.context.WebApplicationContext;

import com.google.gson.Gson;

import net.seehope.demo.pojo.User;

/\*\*

\* @author : MT

\* @date 创建时间：2018年3月22日 下午9:43:00

\* @version 1.0

\*/

@RunWith(SpringRunner.class)

@SpringBootTest(classes = DemoApplication.class)

public class UserControllerTest {

    @Autowired

    private WebApplicationContext webApplicationContext;

    // 模拟MVC环境相关

    private MockMvc mockMvc;

    /\*\*

     \* 每次方法执行之前都模拟一个MVC环境

     \* 每次运行都相当于服务器重启

     \*/

    @Before

    public void setUp() {

        mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();

    }

    @Test

    public void whenQuerySuccess() throws UnsupportedEncodingException, Exception {

        // 模拟发送一个get请求

        String result =

         mockMvc.perform(MockMvcRequestBuilders

                .get("/user")//设置访问路径

                .param("username", "mt")//设置访问参数 key,value

                .param("age", "18")

                .param("ageTo", "60")

                .param("xxx", "something")

                .param("size", "20")

                .param("page", "3")

                .param("sort", "age,desc")

                .contentType(MediaType.APPLICATION\_JSON\_UTF8))//设置访问方式是JSON形式字符串

                .andExpect(MockMvcResultMatchers.status().isOk())//期待的返回值状态码为200

                .andExpect(MockMvcResultMatchers.jsonPath("$.length()").value(3))

                //期待的返回值中，json返回数组长度为3

                .andReturn()//对响应的期待设置完毕

                .getResponse()//获取响应头

                .getContentAsString();//将响应头转化为String

        System.out.println(result);//输出响应中的json数据

}

}

解释一下代码。首先，在@Before中，初始化WEB环境，相当于每个方法执行之前重启Tomcat。

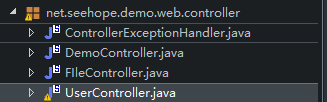
在测试用例whenQuerySuccess中，需要伪造一个请求，通过GET的方式发送到/user这个地址中。通过MockMvcRequestBuilders伪造请求，.get方法设置当前请求是GET方式，并配置访问路径为/user, .param方法配置当前GET请求参数，该方法可以被调用无数次，.contentType方法设置访问形式，在这里设置成UTF-8编码的Json形式字符串。至此，请求伪造完毕。.andExpect方法类似于断言的应用，即设置当前请求的响应。(MockMvcResultMatchers.status().isOk())表示期待的返回状态为成功，即状态码为200，MockMvcResultMatchers.jsonPath("$.length()").value(3))表示返回是一个Json形式的字符串，且一共有三个对象，在.jsonPath方法中填的字符串 是一个特殊的表达式，称之为JSONPath语法，本文不对改表达式深入探讨，有兴趣的可以自己查询相关资料。.andReturn()方法表示期待的响应设置完毕，.getResponse()方法得到响应，.getContentAsString();方法将得到的响应转化为String类型，并在后面输出。至此，测试用例编写完毕。运行测试用例，得到结果：

java.lang.AssertionError: Status expected:<200> but was:<404>

理所当然的出现了此异常，因为目前只编写了测试用例而没有编写对应的方法，所以测试用例中，断言响应状态未成功，但是实际上，此请求招不到对应的API，所以抛出了404（找不到当前资源异常）。

既然明确的知道此测试用例会跑出异常，为什么还要特意执行一次呢?因为测试用例本身也是一部分代码，既然是代码，那么必然会有出错的可能性，所以测试用例也需要执行一次来判断测试用例本身是否正常。

接下来，编写对应的API



/\*\*

\* @author : MT

\* @date 创建时间：2018年3月22日 下午10:55:43

\* @version 1.0

\*/

@RestController

@RequestMapping("/user")

public class UserController {

@GetMapping

public List<User> queryAll(User parmUser){

    System.out.println(ReflectionToStringBuilder.toString(parmUser, ToStringStyle.MULTI\_LINE\_STYLE));

    List<User> users = new ArrayList<>();

    User user = new User();

    user.setPassword("123456");

    user.setUsername("mt");

    users.add(user);

    users.add(user);

    users.add(user);

    return users;

}

}

此API接受一个user类型的参数，返回一个user类型的list集合，方法中，通过ReflectionToStringBuilder.toString(parmUser,ToStringStyle.MULTI\_LINE\_STYLE)); 方法将参数中的user转化为字符串类型，并通过多行的样式输出。最终，执行测试用例的结果如下，状态为success。

net.seehope.demo.pojo.User@213ceb4e[

id=<null>

username=mt

birthday=<null>

password=<null>

]

[{"id":null,"username":"mt","birthday":null,"password":"123456"},{"id":null,"username":"mt","birthday":null,"password":"123456"},{"id":null,"username":"mt","birthday":null,"password":"123456"}]

2.2.2使用注解声明RESTfulAPI

① @RestController 表明此Controller提供RestAPI

此注解位于类级别，相当于@Controller注解和@ResponseBody的结合，即标注当前类中的所有方法，返回结果都是json形式的字符串（遵循RESTful风格）。

② @RestquestMapping及其变体。映射HTTP请求URL到java方法

类似于RestquestMapping注解的功能，有4中变体，分别对应RESTful风格中的4个不同的请求方式，实际上就相当于在原始的RestquestMapping设置了请求方式，使用变种更加方便快捷。

@RequestMapping(path="/user",method=RequestMethod.GET)

@GetMapping("/user")

@PostMapping("/user")

@PutMapping("/user")

@DeleteMapping("/user")

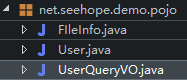
③ @RequestParm 映射请求参数到java方法的参数

public List<User> queryAll(@RequestParam(name="username",defaultValue="mt",required=true) String username,User parmUser)

依然参照UserController中的方法，如果方法接受一个8大基本类型或者是String类型的参数，那么可以使用@RequestParm注解来声明对参数的约束，其中，name参数的值表示从请求中获取的parm的key，即相当于调用了request.getParameter(name参数的值)然后将获取到的值，存放如被标注的方法参数username中，也就是说，在前端传入的参数名和后台方法的参数名不对应的情况下，可以使用该注解注入参数。Required参数的类型为boolean类型，表示被标注的参数是否是当前方法调用的必要参数，此参数和defaultValue参数之间有联系，当required设置为true，且没有设置defaultValue时，如果前端没有传入该方法参数，那么该方法会抛出异常，如果有设置defaultvalue，前端没有传入参数，那么该方法参数会成为默认值并被调用。

2.2.3在RESTfulAPI中传递参数

在实际业务中，查询User的时候可能会依据非常多的参数去查，比如按照年龄，按照生日，按照出生日期等等条件去查询，如果把所有的参数直接写在方法的参数列表中并通过@RequestParm标注并不是说不行，但是代码略显繁琐，所以一般情况下，可以封装一个对象，专门用来封装查询参数



/\*\*

\* @author : MT

\* @date 创建时间：2018年3月22日 下午11:09:25

\* @version 1.0

\*/

public class UserQueryVO {

    private String username;

    private int age;

    private int ageTo;

    private String xxx;

}

再对UserController中的 queryAll方法修改一下

    @GetMapping

    public List<User> queryAll(UserQueryVO userQueryVO,

            @PageableDefault(page = 3, size = 15, sort = { "age,desc" }) Pageable pageable) {

        System.out.println(ReflectionToStringBuilder.toString(userQueryVO, ToStringStyle.MULTI\_LINE\_STYLE));

        System.out.println(pageable.getPageSize());

        System.out.println(pageable.getPageNumber());

        System.out.println(pageable.getSort());

        List<User> users = new ArrayList<>();

        User user = new User();

        user.setPassword("123456");

        user.setUsername("mt");

        users.add(user);

        users.add(user);

        users.add(user);

        return users;

    }

其中，接受的参数使用了userqueryvo对象，同时，添加了@PageableDefault(page = 3, size = 15, sort = { "age,desc" }) Pageable pageable对象用以接受分页参数，如果使用spring data作为dao层实现，那么可以很方便的配合该方法进行分页操作，在此对象中，封装了3个参数，page:当前页码,size:每页记录条数sort:排序依据，此参数为String数组固定传入两个字符串，前一个数为排序依据字段，后一个参数为asc或者desc。大部分分页逻辑都可以通过这三个参数实现，dao层实现不是spring data也没关系。@PageableDefault此注解意义在于前端如果没有传入分页参数，则提供默认的分页参数。注意，在这里并没有真的调用dao层实现，所以此参数并没有实际意义，因为本文并非不关注dao层实现，故使用内存中对象模拟dao层。至此，新的API开发完毕，因为在之前的测试用例中就有传入userquery对象，以及分页参数，所以测试用例不需要更改，直接执行，得到如下结果。

net.seehope.demo.pojo.UserQueryVO@b267745[

username=mt

age=18

ageTo=60

xxx=something

]

20

3

age: DESC

[{"id":null,"username":"mt","birthday":null},{"id":null,"username":"mt","birthday":null},{"id":null,"username":"mt","birthday":null}]

查询所有请求开发完毕。

### 2-3.用户详情请求

2.3.1 @PathVariable 映射URL片段到Java方法的参数

同理，依据测试驱动编程的开发规范，先开发测试用例，在UserControllerTest中，添加新的测试方法。

@Test

    public void whenGetInfoSuccess() throws Exception {

        String result = mockMvc

                .perform(MockMvcRequestBuilders.get("/user/1").contentType(MediaType.APPLICATION\_JSON\_UTF8))

                .andExpect(MockMvcResultMatchers.status().isOk()).andExpect(jsonPath("$.username").value("mt"))

                .andReturn().getResponse().getContentAsString();

        System.out.println(result);

    }

whenGetInfoSuccess方法直接向/user/1发送请求，不携带任何参数，同时期待返回状态为200，以及返回的json数据中，第一层中key为username的字段的值为mt。并在最后将响应转化为字符串输出。

按照惯例执行测试代码，同样得到上述异常，表示测试用例编写无误。

于UserController中编写API

// 占位符接受参数同时使用正则表达式可以直接验证参数是否符合规则

@GetMapping("/{id:\\d+}")

public User getUserInfoById(@PathVariable String id) {

    System.out.println("innerControllerService");

    User user = new User();

    user.setUsername("mt");

    user.setPassword("123456");

    return user;

}

重新执行测试用例，得到结果 控制台输出：

innerControllerService

{"id":null,"username":"mt","birthday":null,"password":"123456"}

在本段代码中API 依据 RESTful 的设计风格，将所有的请求视作资源，那么，查询具体用户的请求应该分配为”/user/id” 含义为访问数据库中ID字段为id的资源，URL中的id对应数据中的ID,假设访问地址为/user/1即为访问User表中的第一条记录，将这种业务视为一种资源，即为RESTful标准。

同时，因为在类级别已经分配了RequestMapping("/user")地址，所以在方法级别只需要将URL中的ID参数拦截即可，故可以使用 @GetMappling（“/{id}”）来拦截URL中的参数，其中{id}表示占位符id两个字符可以替换成任意字符串，配合方法中的@PathVariable注解，将占位符中的拦截到的地址，注入到方法中的参数，以便在方法中使用。在此流程中，占位符中的字符串与参数名必须一致，不一致的话，需要通过参数注解声明占位符名。

@PathVariable(name="id")

2.3.2在URL声明中使用正则表达式

在此案例中：

@GetMapping("/{id:\\d+}")

占位符{id}中还有:\\d+字符串，冒号后面的字符串为正则表达式，\d+表示为1位或者1位以上的数字，即对前端传入的URL做了限制，当传入的URL不符合规则时，抛出异常，编写测试用例测试，把whenGetInfoSuccess方法复制并修改一下，得到whenGetInfoFail。

@Test

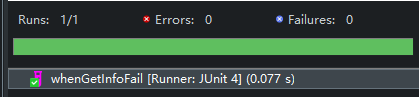
    public void whenGetInfoFail() throws Exception {

        mockMvc.perform(MockMvcRequestBuilders.get("/user/a").contentType(MediaType.APPLICATION\_JSON\_UTF8))

                .andExpect(MockMvcResultMatchers.status().is4xxClientError());

    }

在此用例中，期望方法的返回值为4xx错误，执行测试用例，测试通过。



2.3.3 @JsonView控制json输出内容

在实际的业务中，接受并返回json形式的字符串有着各种各样的问题，比如，如果直接返回User对象，那么存在User封装中的Password势必也会返回到前端，如果在数据库指令中，改成不查询password又太过麻烦，可重用性低，所以，通常，可以通过@JsonView注解指定返回前端的字段而不需要做其他任何更改。

使用@JsonView步骤

2.3.3.1 使用接口来声明多个视图

public class User {

    public interface UserSimpleView {

    };

    public interface UserDetailView extends UserSimpleView {

    };

}

对User类进行改造，先声明两个内部接口，UserSimpleView 此接口表示简单的视图，即，不包含有Password字段，其次，声明UserDetailView视图，集成simple视图，也就是说，detail视图除了拥有simple视图中的字段之外，还包含有其他字段。

2.3.3.2 在值对象的GET方法或者直接在成员变量上指定视图归属

@JsonView(UserSimpleView.class)

private String id;

@JsonView(UserSimpleView.class)

private String username;

@JsonView(UserSimpleView.class)

private Date birthday;

@JsonView(UserDetailView.class)

private String password;

声明完了两个视图之后，依据上述逻辑，除了password之外的字段归属于simple视图，password字段归属于detail视图，因为detail视图集成simple视图，所有改类设计中的detail视图包含有所有字段。至此，视图分配完毕。

2.3.3.3 在Controller方法上指定视图

修改getUserInfoById方法，在方法级别加上@JsonView(User.UserDetailView.class)注解表示该方法返回JSON形式字符串的时候，使用该视图，即不包含password

// 占位符接受参数同时使用正则表达式可以直接验证参数是否符合规则

@JsonView(User.UserSimpleView.class)

@GetMapping("/{id:\\d+}")

public User getUserInfoById(@PathVariable(name="id") String id) {

    // throw new UserNotExistException(id);

    System.out.println("innerControllerService");

    User user = new User();

    user.setUsername("mt");

    user.setPassword("123456");

    return user;

}

再次执行whenGetInfoSuccess测试用例，得到结果

innerControllerService

{"id":null,"username":"mt","birthday":null}

很明显，password字段已经不包含在返回结果中。

### 2-4.用户创建请求

2.4.1 @RequestBody 映射请求体到Java方法参数

依据RESTful风格，创建以及需要测试的API应该为POST方式的/user请求，表示在user表中添加一条资源。

依据测试驱动开发的规范，先编写测试用例，

@Test

    public void whenCreateUserSuccess() throws Exception {

        User user = new User();

        user.setUsername("mt");

        mockMvc.perform(MockMvcRequestBuilders

                .post("/user")

                .contentType(MediaType.APPLICATION\_JSON\_UTF8)

                .content(new Gson().toJson(user)))

                .andExpect(MockMvcResultMatchers.status().isOk())

                .andExpect(jsonPath("$.id").value(1));

    }

该测试用例向/user 发送post请求，创建一个新的User对象，并设置初始化测试数据，同时，将user转化为json形式字符串，加入请求中内容中，设置请求参数类型为json形式字符串，期待返回状态码为200，且在返回的json数据中，id字段的值为1。至此测试用例编写完成。

编写UserController中的API

// 如果是post请求接受参数，必须声明requestbody

@PostMapping

public User createUser(@RequestBody User user) {

    System.out.println(ReflectionToStringBuilder.toString(user));

    user.setId("1");

    return user;

}

此API接受一个User类型的对象，并最终将user作为json形式的字符串返回。注意，如果要接受一个json形式的字符串，作为参数，对应接受的参数，必须使用@RequestBody注解声明该参数是一个json形式的字符串。，执行测试用例

得到结果



net.seehope.demo.pojo.User@1939a394[id=<null>,username=mt,birthday=<null>,password=<null>]

API测试完毕。

2.4.2 日期类型参数的处理

在实际开发中，经常会遇到日期类型的处理，但是前端和后台之间日期类型的约定有很多种标准，但是在前后端完全分离的架构中，按照通常的“yyyy-MM-dd HH:mm:ss”这种约定，往往容易出现很多问题，因为后台API通常会被很多不同的终端调用，不同终端处理的方式不同，为了让API更加的通用，最好使用时间戳进行日期的传递，不带格式。前端在收到时间戳之后，自己决定如何显示，后台只负责提供数据。

修改测试用例

@Test

    public void whenCreateUserSuccess() throws Exception {

        Date date=new Date();

        String content = "{\"id\":\"1\", \"username\":\"mt\",\"password\":123456,\"birthday\":" + date.getTime() + "}";

        String result=  mockMvc.perform(MockMvcRequestBuilders

                .post("/user")

                .contentType(MediaType.APPLICATION\_JSON\_UTF8)

                .content(content))

                .andExpect(MockMvcResultMatchers.status().isOk())

                .andExpect(jsonPath("$.id").value(1))

                .andReturn()

                .getResponse()

                .getContentAsString();

        System.out.println(result);

    }

在此测试用例中，直接在请求内容中存放json形式的字符串，并将birthday设置成当前时间的时间戳，发送给后台。执行测试用例，得到结果。

net.seehope.demo.pojo.User@154f8280[

id=1

username=mt

birthday=Thu Dec 27 16:51:18 GMT+08:00 2018

password=<null>

]

{"id":"1","username":"mt","birthday":1545900678647,"password":null}

其中，第一条user输出在API中，SpringMVC自动将请求中获取的时间戳转换成了Date对象，第二条记录来自于测试用例中响应的输出，SpringMVC讲其自动转化为时间戳形式，而对于前端而言，任何终端都有能力将时间戳类型显示在前端。

2.4.3 @valid注解和BindingResult验证请求参数的合法性处理检验结果

在实际业务中，往往需要验证用户传入的参数，前端验证并不能完全的保证数据的正确性，后台也需要对用户输入进行验证，在此业务中，可以在POJO类User中的字段中，添加注解，说明验证逻辑，在此用例中，先设定好验证规则，之后使用该方法进行约定。

验证三条规则1.生日必须为过去的时间，2.密码非空，3.自定义用户名验证规则

以下为可供使用的预先定义好的验证注解:

@Null   限制只能为null

@NotNull    限制必须不为null

@AssertFalse    限制必须为false

@AssertTrue 限制必须为true

@DecimalMax(value)  限制必须为一个不大于指定值的数字

@DecimalMin(value)  限制必须为一个不小于指定值的数字

@Digits(integer,fraction)   限制必须为一个小数，且整数部分的位数不能超过integer，小数部分的位数不能超过fraction

@Future 限制必须是一个将来的日期

@Max(value) 限制必须为一个不大于指定值的数字

@Min(value) 限制必须为一个不小于指定值的数字

@Past   验证注解的元素值（日期类型）比当前时间早

@Pattern(value) 限制必须符合指定的正则表达式

@Size(max,min)  限制字符长度必须在min到max之间

@NotEmpty   验证注解的元素值不为null且不为空（字符串长度不为0、集合大小不为0）

@NotBlank   验证注解的元素值不为空（不为null、去除首位空格后长度为0），不同于@NotEmpty，@NotBlank只应用于字符串且在比较时会去除字符串的空格

@Email  验证注解的元素值是Email，也可以通过正则表达式和flag指定自定义的email格式

修改POJO中的User，执行以上验证逻辑，先使用预先定义好的验证逻辑

public class User {

    public interface UserSimpleView {

    };

    public interface UserDetailView extends UserSimpleView {

    };

    @JsonView(UserSimpleView.class)

    private String id;

    @JsonView(UserSimpleView.class)

    private String username;

    @Past(message = "生日必须是过去的时间")

    @JsonView(UserSimpleView.class)

    private Date birthday;

    @NotBlank(message = "密码不能为空")

    @JsonView(UserDetailView.class)

    private String password;

}

在birthday字段以及password字段上加上了验证注解，其中，message字段表示当用户输入不符合验证规则时返回给前端的提示。此时，在pojo上验证规则声明完毕，类似于JsonView的使用方式，声明完了规则，当时用的时候需要在API方法级别声明使用此规则，使得规则复用性很强，需要验证就应用规则，不需要验证的时候不应用也不会强制验证POJO，这种设计模式非常值得参考。

修改API，在API中应用对POJO的验证规则

// 如果是post请求接受参数，必须声明requestbody

@PostMapping

public User createUser(@Valid @RequestBody User user) {

    System.out.println(ReflectionToStringBuilder.toString(user));

    user.setId("1");

    return user;

}

即，在方法接受参数的时候，使用注解@Valid标注方法参数，那么，SpringMVC在接收该类型的参数时，会到User类下查询字段的验证逻辑。

修改测试用例whenCreateUserSuccess中的content字符串中的password字段的值为null，让此用例的参数不符合API需求。

String content = "{\"id\":\"1\", \"username\":\"mt\",\"password\":null,\"birthday\":" + date.getTime() + "}";

执行测试用例，得到异常

java.lang.AssertionError: Status expected:<200> but was:<400>

请求状态码400，说明请求参数错误。即密码为null。

很明显，此用例的业务不符合常态下的也无需求，当用户访问不符合需求，直接抛出异常，连当前API都没有进入，直接返回异常到被调用方，而且在没有添加异常处理通知的情况下，不符合业务要求，通常，需要对异常进行处理，例如记录异常日志，跳转到错误页面，并显示错误信息等，此时，SpringMVC默认的处理为直接抛出异常，那么，如何在代码中获取异常信息，让验证失败的请求进入代码中处理呢？

在使用了@Valid注解参数的方法中，添加BindingResult errors参数，当SpringMVC扫描到方法有该参数时，会将错误信息存放进该对象，同时不抛出异常，以便程序员在方法中处理该异常

修改API为

    // 如果是post请求接受参数，必须声明requestbody

    @PostMapping

    public User createUser(@Valid @RequestBody User user, BindingResult errors) {

        if (errors.hasErrors()) {

            errors.getAllErrors().stream().forEach(error -> {

                FieldError fieldError = (FieldError) error;

                System.out.println(fieldError.getField() + " : " + fieldError.getDefaultMessage());

            });

        }

        System.out.println(ReflectionToStringBuilder.toString(user));

        user.setId("1");

        return user;

    }

在当前API中，循环遍历errors对象，将错误信息打印到控制台上，在正式的业务场景中，可以针对异常做不同的处理，在后面的章节中，会介绍如何统一处理服务异常信息。

此时，再执行上一例中password为null的测试用例，得到结果

password : 密码不能为空

net.seehope.demo.pojo.User@3cb173db[id=1,username=mt,birthday=Thu Dec 27 17:43:02 GMT+08:00 2018,password=<null>]

{"id":"1","username":"mt","birthday":1545903782277,"password":null}

从结果可以得到，当前API已被调用，没有在异常中终止，而是等待用户自己处理异常，至此，springMVC的验证逻辑使用方法演示结束。

在实际的应用中，有的时候使用预先定义的验证逻辑有时候并不能完全的满足业务逻辑，比如说对用户名的验证，在实际业务中，除了需要验证用户名输入的字符串符合业务流程，同时，也需要验证当前用户注册输入的用户名在数据库中是否已经存在。很明显，此业务涉及到Dao层应用，预先定义好的验证器无法完成此业务，鉴于此类情况，SpringMVC也支持自定义Validator组件。

SpringMVC的验证都是基于注解的，如果需要添加验证器，那么也就是要定义新的注解，首先，创建注解



package net.seehope.demo.web.validator.annotation;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

import javax.validation.Constraint;

import javax.validation.Payload;

import net.seehope.demo.web.validator.MyValidator;

/\*\*

\* @author : MT

\* @date 创建时间：2018年4月10日 下午11:00:36

\* @version 1.0

\*/

@Target({ ElementType.METHOD, ElementType.FIELD })

@Retention(RetentionPolicy.RUNTIME)

@Constraint(validatedBy = MyValidator.class)

public @interface MyValidatorAnnotation {

    String message();

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}

其中注解@Target表示当前自定义注解可以应用在方法级别以及字段级别，@Retention注解表示当前注解为运行时注解，@Constraint(validatedBy = MyValidator.class注解表示当前注解的业务执行类，这个类目前还没有创建，先写上去，之后再创建此逻辑类，先看注解的字段，案例中的三个字段，都是SpringMVC 验证器标准中固定好必须要有的三个参数，其中，message参数在之前的案例中已经使用过，表示当验证逻辑错误时，返回给前端的信息。此标准可以在之前用例的POJO User中，点开跳转到任意一个验证注解中查看。

接着编写注解逻辑执行类



package net.seehope.demo.web.validator;

import javax.validation.ConstraintValidator;

import javax.validation.ConstraintValidatorContext;

import org.springframework.beans.factory.annotation.Autowired;

import net.seehope.demo.service.HelloService;

import net.seehope.demo.web.validator.annotation.MyValidatorAnnotation;

/\*\*

\* @author : MT

\* @date 创建时间：2018年4月10日 下午11:01:56

\* @version 1.0

\*/

public class MyValidator implements ConstraintValidator<MyValidatorAnnotation, Object> {

    // 可以在验证器中使用注解，当前类不需要注册到spring

    @Autowired

    private HelloService helloService;

    @Override

    public void initialize(MyValidatorAnnotation constraintAnnotation) {

        // TODO Auto-generated method stub

        System.out.println(" myValidator init ");

    }

    // context包含了注解中的信息

    @Override

    public boolean isValid(Object value, ConstraintValidatorContext context) {

        // TODO Auto-generated method stub

        helloService.sayHello();

        System.out.println("valid value" + value);

        return false;

    }

}

自定义验证逻辑类需要实现ConstraintValidator<MyValidatorAnnotation, Object>接口期中，第一个泛型表示应用此注解逻辑的注解，第二个泛型表示被当前注解标注的对象的类型。

实现此接口，需要覆盖两个方法initialize和isValid，初始化方法并没有太多的应用，主要是isValid逻辑，如果当前方法返回true，则验证通过，如果返回为false，则验证失败，同时，按照之前的逻辑，此注解需要应用在User.username上，所以，需要去Dao层中查询当前username是否存在，在isValid 方法参数中value表示被标注的字段，即username，可以在当前类的方法中注入Dao层方法，将username作为参数，查询，根据数据库返回的结果判断当前注解返回值，在本例中，并不讨论数据库实现，故只是象征性的调用UserService。在控制台中打印一句话。注意，此注解验证类不需要注册到Spring容器中。而是通过注解关联。此时，注解和注解验证逻辑类编写完成，开始使用和测试。

修改pojo类：

@MyValidatorAnnotation(message = "this is a test")

@JsonView(UserSimpleView.class)

private String username;

使用自定义注解，再次调用测试用例，得到返回结果

myValidator init

say hello

valid valuemt

username : this is a test

password : 密码不能为空

net.seehope.demo.pojo.User@13d10057[id=1,username=mt,birthday=Thu Dec 27 18:02:40 GMT+08:00 2018,password=<null>]

{"id":"1","username":"mt","birthday":1545904960124,"password":null}

自定义注解验证逻辑开发完成

请认真思考上面的一个自定义校验注解的流程，可以轻松掌握在后期的开发中，使用注解来实现校验，而不是写许多重复的校验逻辑代码。

### 2-5.修改和删除请求

本小节主要目的在于说明修改以及删除请求的编写，在实际业务中，测试用例断言的结果必须为成功，但是在本用例中，为了演示代码流程，断言并不是成功的，只是为了演示用例。

在上一小节的内容中，已经介绍了如何使用以及自定义验证逻辑，在开发中，这些逻辑都是可以服用的，首先，先编写修改请求测试用例。

@Test

public void whenUpdateUserSuccess() throws Exception {

/\*  User user = new User();

user.setId("1");

user.setPassword("123456");

user.setUsername("mtproject");\*/

Date date = new Date(

LocalDateTime.now().plusYears(1).atZone(ZoneId.systemDefault()).toInstant().toEpochMilli());

// System.out.println(new Gson().toJson(user));

String content = "{\"id\":\"1\", \"username\":\"mt\",\"password\":null,\"birthday\":" + date.getTime() + "}";

String result = mockMvc

.perform(MockMvcRequestBuilders.put("/user/1").contentType(MediaType.APPLICATION\_JSON\_UTF8)

.content(content))

.andExpect(MockMvcResultMatchers.status().isOk()).andReturn().getResponse().getContentAsString();

System.out.println(result);

}

在修改用例中，请求参数中的birthday特意被设置成了当前时间一年后的时间戳，以put请求的方式，发送到/user/1 API中，并期待返回状态为200。执行测试用例，得到400异常。

编写API接口：

@PutMapping("/{id:\\d+}")

    public User updateUser(@Valid @RequestBody User user, BindingResult errors) {

        if (errors.hasErrors()) {

            errors.getAllErrors().stream().forEach(error -> {

                FieldError fieldError = (FieldError) error;

                System.out.println(fieldError.getField() + " : " + fieldError.getDefaultMessage());

            });

        }

        System.out.println(ReflectionToStringBuilder.toString(user));

        return user;

    }

类似于增加服务，在修改服务中，处理PUT请求，验证URL规则，对请求参数应用验证逻辑，同时从JSON数据转化为user对象，如果发生校验异常，则将错误信息存放在errors对象中，框架本身不处理校验异常。

在POJO中，设置了birthday字段的校验逻辑，必须是过去的日期。此时，调用测试用例，得到结果如下：

myValidator init

say hello

valid valuetom

password : 密码不能为空

birthday : 生日必须是过去的时间

username : this is a test

net.seehope.demo.pojo.User@b144175[id=1,username=tom,birthday=Sat Dec 28 09:42:40 GMT+08:00 2019,password=<null>]

{"id":"1","username":"tom","birthday":1577497360140,"password":null}

控制台信息和预期相符合

接着处理删除服务，编写测试用例

@Test

public void whenDeleteUserSuccess() throws Exception {

mockMvc.perform(MockMvcRequestBuilders.delete("/user/1").contentType(MediaType.APPLICATION\_JSON\_UTF8))

.andExpect(MockMvcResultMatchers.status().isOk());

非常简单，向/user/1发送delete请求，并期待返回状态成功。

编写服务API

@DeleteMapping("/{id}")

    public void deleteUser(@PathVariable Integer id) {

        System.out.println(id);

    }

删除服务测试完毕，不包含Dao实现。

### 2-6.服务异常处理

当我们从浏览器访问不存在的Spring Boot的RESTful API的时候，往往会返回Spring Boot内置的404错误界面，但是作为前后端分离的应用，相同的API也许会在其他终端访问，比如手机APP等，那么也会是相同的处理方式吗?

2.6.1 SpringBoot 默认的错误处理机制

首先，先来看看SpringMVC对浏览器访问的默认处理。从DemoApplication.java处启动服务。

访问服务中不存在的API

http://localhost:8080/1

得到结果

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Fri Dec 28 10:11:15 GMT+08:00 2018

There was an unexpected error (type=Not Found, status=404).

No message available

以上结果即为Sringboot默认的404界面，同理，如果访问异常的API，得到结果

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

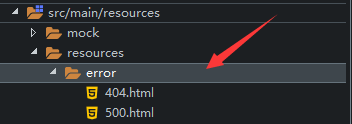
Fri Dec 28 10:12:55 GMT+08:00 2018

There was an unexpected error (type=Internal Server Error, status=500).

Circular view path [hello]: would dispatch back to the current handler URL [/hello] again. Check your ViewResolver setup! (Hint: This may be the result of an unspecified view, due to default view name generation.)

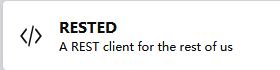
以上为默认的500状态页面，很明显，一般业务情况下，不可能直接跳转到这种默认的错误页面，直接将错误抛给用户。

按照springboot的默认约定，可以创建以下文件结构，取代默认的404以及500页面。

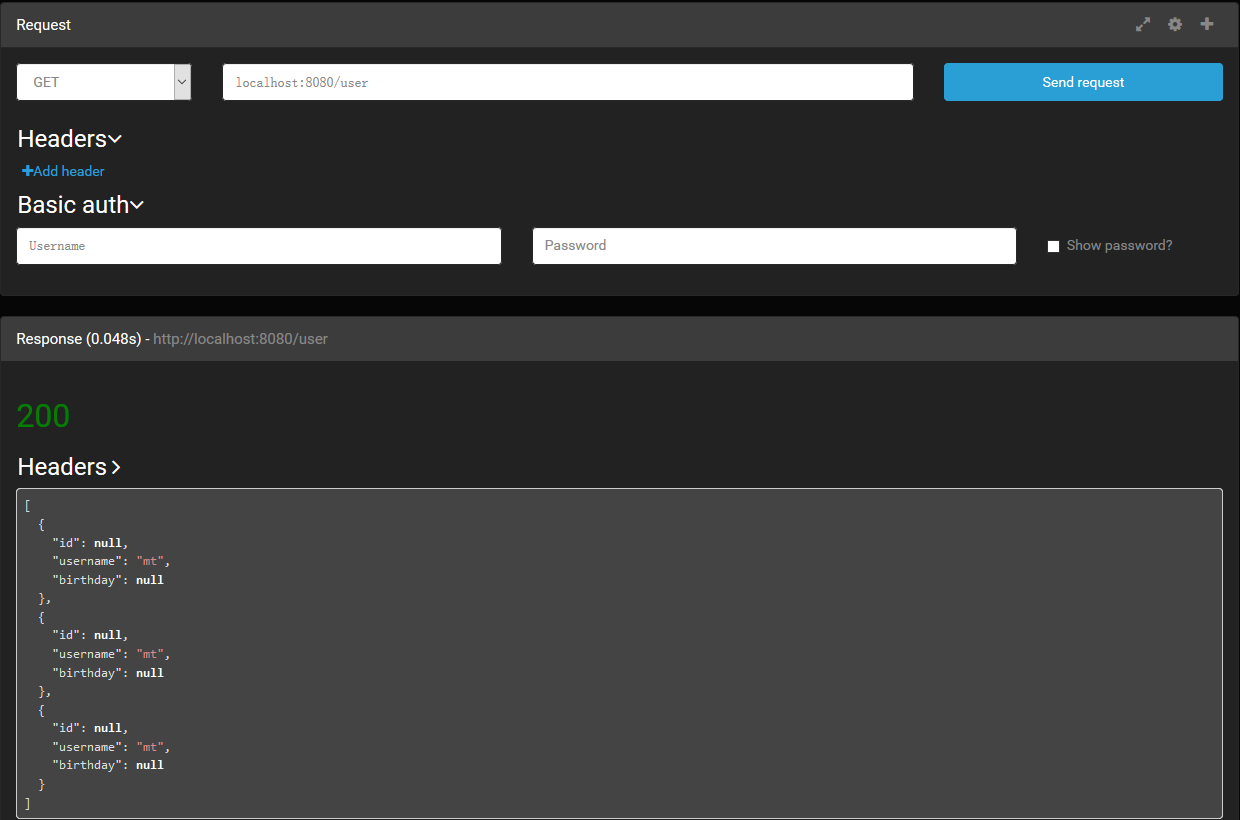


当Springboot扫描到resources源码路径下resources文件夹下有error文件夹，且其中有 状态码.html 文件，那么当服务抛出响应异常时，即跳转到对应HTML文件。以上即为Springboot在默认情况下对浏览器访问异常的处理。

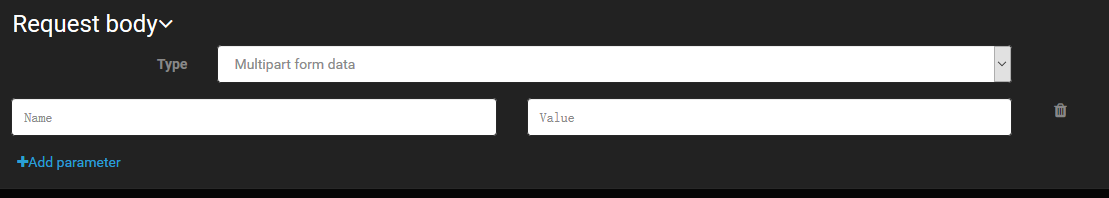
接着，模拟APP对服务进行访问，很多软件都可以做到，本例中使用Firefox的RESTED插件



界面非常简单



在头部中，可以对request进行设置，选择请求方式，以及请求地址，在Header选项卡中，可以对请求头进行设置，在Basic auth中可以对默认的权限信息进行设置，因为此插件转为RESTful风格API设置，所以在GET请求中，并不能设置请求参数，当请求方式设置为post,put之后，选项中会出现



这段内容，即对请求头中的参数进行设置。

当使用插件发送会导致404以及500请求的时候，看一下服务返回的响应会是什么情况。

当访问404

Get localhost:8080/1

得到结果

X-Application-Context: application:8080

Content-Type: application/json;charset=UTF-8

Transfer-Encoding: chunked

Date: Fri, 28 Dec 2018 02:30:23 GMT

{

"timestamp": 1545964223037,

"status": 404,

"error": "Not Found",

"message": "No message available",

"path": "/1"

}

当访问500

GET localhost:8080/hello

得到结果

X-Application-Context: application:8080

Content-Type: application/json;charset=UTF-8

Content-Language: zh-CN

Transfer-Encoding: chunked

Date: Fri, 28 Dec 2018 02:31:43 GMT

Connection: close

{

"timestamp": 1545964303383,

"status": 500,

"error": "Internal Server Error",

"exception": "javax.servlet.ServletException",

"message": "Circular view path [hello]: would dispatch back to the current handler URL [/hello] again. Check your ViewResolver setup! (Hint: This may be the result of an unspecified view, due to default view name generation.)",

"path": "/hello"

}

很明显，SpringMVC在默认情况下对浏览器以及其他终端的响应做了不同的处理，如果是浏览器访问，那么会得到一段HTML内容，如果是其他移动端，那么会得到一段JSON响应。那么，SpringMVC是如何识别请求是由浏览器还是移动终端发起的呢?可以浏览一下源码，这段源代码来自于Spring Boot的一个包org.springframework.boot.autoconfigure.web中的BasicErrorController，从类名就可以知道它是一个Controller，且处理的路径就是“/error”。找到错误处理的两个方法。

@RequestMapping(produces = "text/html")

    public ModelAndView errorHtml(HttpServletRequest request,

            HttpServletResponse response) {

        HttpStatus status = getStatus(request);

        Map<String, Object> model = Collections.unmodifiableMap(getErrorAttributes(

                request, isIncludeStackTrace(request, MediaType.TEXT\_HTML)));

        response.setStatus(status.value());

        ModelAndView modelAndView = resolveErrorView(request, response, status, model);

        return (modelAndView == null ? new ModelAndView("error", model) : modelAndView);

    }

    @RequestMapping

    @ResponseBody

    public ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {

        Map<String, Object> body = getErrorAttributes(request,

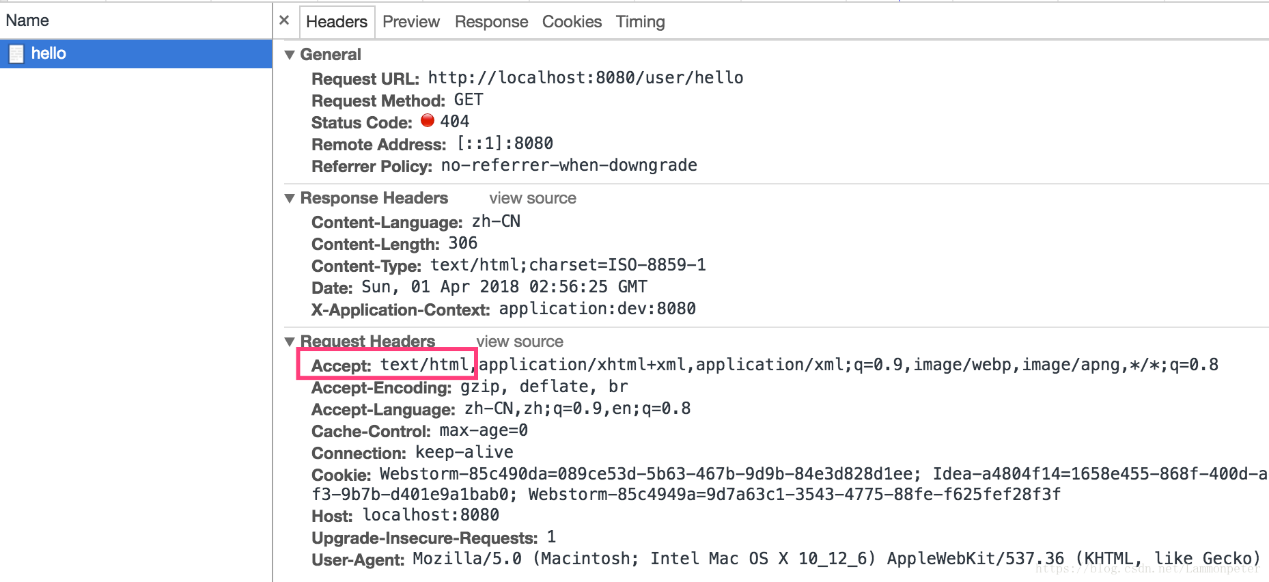
                isIncludeStackTrace(request, MediaType.ALL));

        HttpStatus status = getStatus(request);

        return new ResponseEntity<Map<String, Object>>(body, status);

    }

第一个方法和第二个方法处理的都是同一个API，区别就在于第一个方法的@RequestMapping里面包含一个produces属性，它表示将生成什么类型的资源返回给前端，很明显，第一个方法要返回的是一个HTML页面，而第二个方法返回的是JSON数据。这就很明了了，当浏览器访问错误的API的时候，会自动进入第一个方法处理错误，从客户端访问的时候，就会进入第二个方法处理错误。当然，从浏览器发送的请求的时候，我们可以看见请求头中看到浏览器要求的返回数据类型就包含了text/html，如下图所示：



从这两个方法中可以看出，默认情况下，当SpringMVC中服务发生了异常，那么SpringMVC会将当前请求转发到/error中，调用对应的BasicErrorController进行处理，如果请求头中包含text/html，那么调用errorHtml方法，返回一个HTML视图，如果不包含，那么调用error方法返回一个JSON形式字符串，以上即为SpringMVC对异常的默认处理机制。

2.6.2,自定义异常处理逻辑

先来看看上节描述的验证异常，修改UserController中的createUser方法，去掉BindingResult参数，看看当参数异常，SpringMVC如何处理。

// 如果是post请求接受参数，必须声明requestbody

@PostMapping

public User createUser(@Valid @RequestBody User user /\*, BindingResult errors\*/) {

/\*  if (errors.hasErrors()) {

errors.getAllErrors().stream().forEach(error -> {

FieldError fieldError = (FieldError) error;

System.out.println(fieldError.getField() + " : " + fieldError.getDefaultMessage());

});

}\*/

System.out.println(ReflectionToStringBuilder.toString(user));

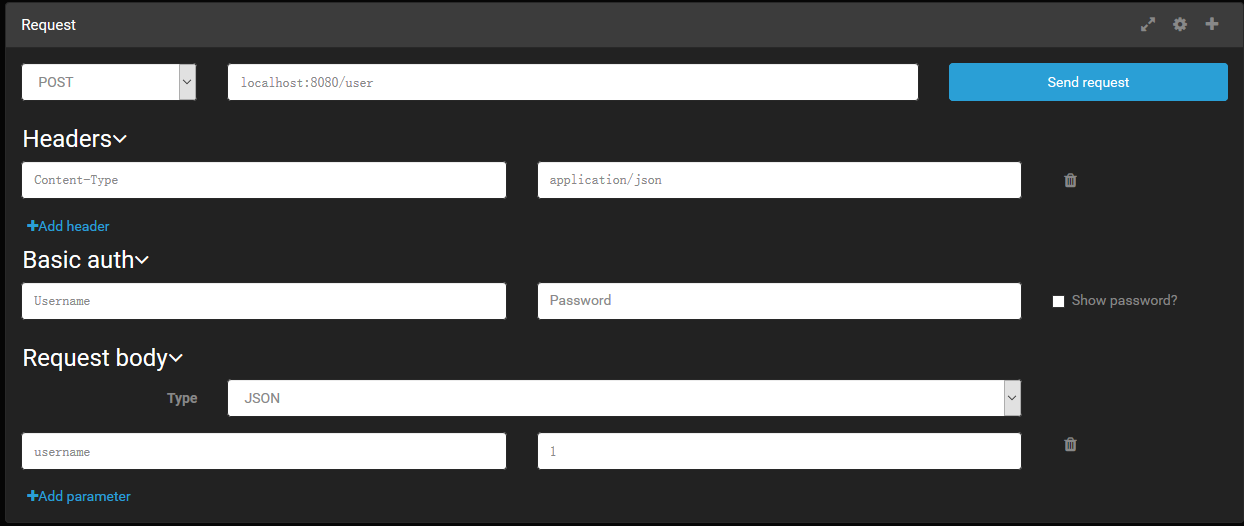
user.setId("1");

return user;

// throw new UserNotExistException("123");

}

在工具中访问当前API



得到结果

X-Application-Context: application:8080

Content-Type: application/json;charset=UTF-8

Transfer-Encoding: chunked

Date: Fri, 28 Dec 2018 02:51:55 GMT

Connection: close

{

"timestamp": 1545965515029,

"status": 400,

"error": "Bad Request",

"exception": "org.springframework.web.bind.MethodArgumentNotValidException",

"errors": [

{

"codes": [

"MyValidatorAnnotation.user.username",

"MyValidatorAnnotation.username",

"MyValidatorAnnotation.java.lang.String",

"MyValidatorAnnotation"

],

"arguments": [

{

"codes": [

"user.username",

"username"

],

"arguments": null,

"defaultMessage": "username",

"code": "username"

}

],

"defaultMessage": "this is a test",

"objectName": "user",

"field": "username",

"rejectedValue": "1",

"bindingFailure": false,

"code": "MyValidatorAnnotation"

},

{

"codes": [

"NotBlank.user.password",

"NotBlank.password",

"NotBlank.java.lang.String",

"NotBlank"

],

"arguments": [

{

"codes": [

"user.password",

"password"

],

"arguments": null,

"defaultMessage": "password",

"code": "password"

}

],

"defaultMessage": "密码不能为空",

"objectName": "user",

"field": "password",

"rejectedValue": null,

"bindingFailure": false,

"code": "NotBlank"

}

],

"message": "Validation failed for object='user'. Error count: 2",

"path": "/user"

}

很明显，SringMVC对服务验证错误异常的处理非常的完善，如果使用自定义的BindingResult参数进行异常处理，无非非常的麻烦，那么，能不能既使用框架的异常处理，又自定义异常处理之后的操作，而不是光抛出异常呢。

SpringMVC提供有异常处理通知标准，即异常切面可以自定义。

创建异常处理类ControllerExceptionHandler



@ControllerAdvice

public class ControllerExceptionHandler {

    @ResponseBody

    @ResponseStatus(HttpStatus.INTERNAL\_SERVER\_ERROR)

    @ExceptionHandler(Exception.class)

    public Map<String, Object> handleException(Exception exception) {

        Map<String, Object> result = new HashMap<>();

        result.put("message", exception.getMessage());

        return result;

    }

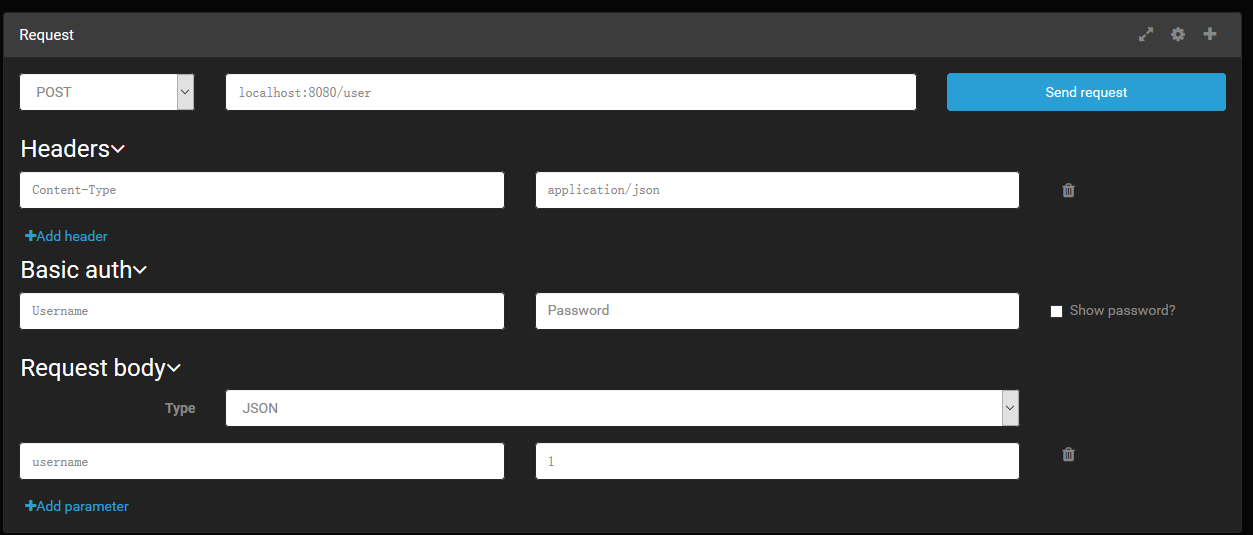
}

在此通知中，可以模仿springmvc的BasicErrorController，分别对浏览器和APP进行不同的响应，再本用例中，只处理非浏览器请求，返回json。

如果spirngboot有扫描到ControllerAdvice类，那么所有异常抛出的时候，会检查此类中是否有对当前异常的处理，如果有，使用自定义处理，如果没有，使用默认处理，在handleException方法中，ExceptionHandler注解表示当前方法对那个异常类进行处理，当前处理Exception类即，所有异常都使用当前方法处理，方法参数必须和待处理异常类匹配，即发生异常后，异常会作为参数传入当前方法，ResponseStatus注解表示当前相应返回状态码，本例中为500。

此方法的返回值为Map对象，即Json形式的键值对，本例中只有一个字段Message，对应的值为异常中的message字段。

此时，在此访问创建User的API



得到结果

{

"message": "Validation failed for argument at index 0 in method: public net.seehope.demo.pojo.User net.seehope.demo.web.controller.UserController.createUser(net.seehope.demo.pojo.User), with 2 error(s): [Field error in object 'user' on field 'username': rejected value [1]; codes [MyValidatorAnnotation.user.username,MyValidatorAnnotation.username,MyValidatorAnnotation.java.lang.String,MyValidatorAnnotation]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [user.username,username]; arguments []; default message [username]]; default message [this is a test]] [Field error in object 'user' on field 'password': rejected value [null]; codes [NotBlank.user.password,NotBlank.password,NotBlank.java.lang.String,NotBlank]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [user.password,password]; arguments []; default message [password]]; default message [密码不能为空]] "

}

自定义异常处理逻辑编写完成。

当然，实际业务中，情况往往要比案例复杂些许，经常需要对不同的异常进行不同的处理，此时，可以编写不同的方法捕获不同的异常，针对浏览器端以及app端进行不同的处理。异常处理通知类非常有用，让异常处理和业务逻辑解耦，一般情况下，BindingResult参数时非常少使用的。

### 2-7.使用filter和interceptor拦截RESTful服务

在某些业务中，往往需要对所有RESTful风格的API进行一些统一的处理，在Spring框架中，有3种处理方式，在本案例中，以记录所有API的访问时间为例，分别使用这三种方式测试。

1. 过滤器（Filter）

过滤器作为最JavaEE标准之一，可以拦截请求，决定当前请求是否能够访问到API，以及响应是否依照API的响应，换言之，拦截器在请求访问到具体API进去和出来的过程中都是起作用的，那么，理所当然的可以用来做对于API的统一处理。

在本案例中，先编写时间过滤器。



public class TimeFilter implements Filter {

    @Override

    public void init(FilterConfig filterConfig) throws ServletException {

        // TODO Auto-generated method stub

        System.out.println(" timeFilter init");

    }

    @Override

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

            throws IOException, ServletException {

        // TODO Auto-generated method stub

        long start = new Date().getTime();

        System.out.println(" filterStartAt " + start);

        chain.doFilter(request, response);

        System.out.println(" filter cost time: " + (new Date().getTime() - start));

    }

    @Override

    public void destroy() {

        // TODO Auto-generated method stub

        System.out.println("timeFilter Destory");

    }

}

在doFilter方法中请求放行之前和之后，截取请求调用时间，然后将当前Filter注册到Spring容器中，并使之对RESTful请求进行拦截。

编写配置文件，注册过滤器



@Configuration

public class WebConfig extends WebMvcConfigurerAdapter {

    /\*\*

     \* spring boot 注册filter

     \* @return

     \*/

    @Bean

    public FilterRegistrationBean timeFilter() {

        FilterRegistrationBean bean = new FilterRegistrationBean();

        TimeFilter filter = new TimeFilter();

        bean.setFilter(filter);

        List<String> urls = new ArrayList<>();

        urls.add("/user/\*");

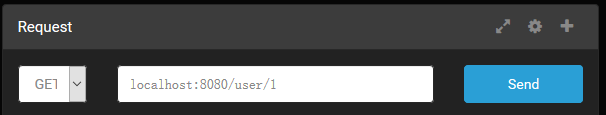
        bean.setUrlPatterns(urls);

        return bean;

    }

}

在配置文件中，配置类需要继承WebMvcConfigurerAdapter 类，表示对MVC环境进行配置，在此类中注册了过滤器。此时，启动服务，通过RESTED调用服务。



得到结果

filterStartAt 1545969776329

innerControllerService

filter cost time: 71

时间过滤器执行完毕。

1. 拦截器（Interceptor）

代理，拦截器，责任链等java设计模式，其中，代理模式作为基础，拦截器作为代理模式的应用层，责任链模式再作为拦截器的应用，有着层层递进的关系，在上述过滤器中，实际上过滤器链和责任链并没有太大的区别。

而拦截器作为过滤器链中的一环，拦截器和过滤器本质上区别也不大，只不过过滤器是JavaSE标准，而当前使用的拦截器，是基于SpringMVC的标准。

首先，编写时间拦截器



@Component

public class TimeIntercepter implements HandlerInterceptor {

    @Override

    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

            throws Exception {

        // TODO Auto-generated method stub

        System.out.println("preHandle");

        request.setAttribute("startTime", new Date().getTime());

        System.out.println(handler.getClass().getName());

        return true;

    }

    /\*\*

     \* 非正常完成不会被调用，例如抛出异常

     \*/

    @Override

    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

            ModelAndView modelAndView) throws Exception {

        // TODO Auto-generated method stub

        System.out.println("postHandle");

        System.out.println("costTime: " + (new Date().getTime() - (long) request.getAttribute("startTime")));

    }

    /\*\*

     \* 完成都会被调用，无论有没有正常完成

     \*/

    @Override

    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)

            throws Exception {

        // TODO Auto-generated method stub

        System.out.println("afterCompletion");

        System.out.println("costTime: " + (new Date().getTime() - (long) request.getAttribute("startTime")));

        System.out.println("ex: " + ex);

    }

}

可以看到，自定义的拦截器需要实现HandlerInterceptor接口并覆盖三个方法，preHandle，postHandle，afterCompletion分别表示，连接点前置处理，连接点后置处理，连接点最终处理。类似于切面的前置通知，后置通知，最终通知概念。

前置通知preHandle方法，返回boolean类型参数，需要传入object类型handler对象，此对象表示连接点对象，如果此方法返回为真，那么连接点方法会被执行，如果返回为false，则连接点方法都不会被执行。

通常，可以用来处理文件上传逻辑，先验证上传的文件是否符合后缀，大小等规则，不符合直接返回false或者抛出异常。

postHandle方法类似于后置通知，即连接点方法执行完毕，按照SpringMVC标准，就是API被调用完毕被执行，此方法不需要返回值，有一个modelAndView参数，也就是说，此方法可以修改目标API最终返回的结果和视图。同时，在这三个方法中都有request以及response对象，即可以修改req以及resp对象，在一次请求生命周期内共享数据，例如本例，在前置通知中，生成初始时间，并存入request对象，并在后置通知中取出该对象并计算。

最后一个afterCompletion方法，类似于最终通知，post方法只有目标API未发生异常之后才会被调用，而此方法，无论目标API是否正常结束都会被调用。同时，在此方法中，存有异常对象，至此，拦截器类编写完毕，同理，将拦截器注册到Spring容器中，在配置文件WebConfig.java即上节配置了filter的位置中添加

@Autowired

TimeIntercepter timeIntercepter;

@Override

public void addInterceptors(InterceptorRegistry registry) {

// TODO Auto-generated method stub

registry.addInterceptor(timeIntercepter);

}

注册时间拦截器。启动服务，在RESTED中访问localhost:8080/user/1得到结果

filterStartAt 1545978490388

preHandle

org.springframework.web.method.HandlerMethod

innerControllerService

postHandle

costTime: 63

afterCompletion

costTime: 64

ex: null

filter cost time: 82

从结果中分析一下，发现过滤器输出出现在首尾，拦截器输出出现在中间，很明显，无论使用什么框架，本质上都是对JavaEE标准中的WEB组件进行封装，注册的filter肯定是在所有应用之前被调用。

### 2-8.使用切面拦截RESTful服务

1. 切面 (Aspect)

切面也是springAOP定下的标准之一，其基本原理也是基于拦截器。本例中通过注解方式声明切面：



@Aspect

@Component

public class TimeAspect {

    @Pointcut("execution(\* net.seehope.demo.web.controller.UserController.\*(..))")

    public void timeJoinPoint() {

    }

    @Around(value = "timeJoinPoint()")

    public Object handlerControllerMethod(ProceedingJoinPoint pjp) throws Throwable {

        long begin = new Date().getTime();

        System.out.println(" aspectTimeBegin: " + begin);

        Object result = pjp.proceed();

        System.out.println(" aspectTimeEnd: " + (new Date().getTime() - begin));

        return result;

    }

}

运行服务，再次通过RESTED访问API，得到结果

filterStartAt 1545979134861

preHandle

org.springframework.web.method.HandlerMethod

aspectTimeBegin: 1545979134898

innerControllerService

aspectTimeEnd: 3

postHandle

costTime: 68

afterCompletion

costTime: 68

ex: null

filter cost time: 87

很明显，filter的优先级依然是最高的，其次是切面，最后是拦截器组件。

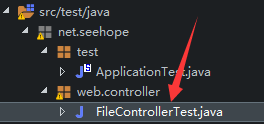
在实际工程中，三个组件的功能都差不多，程序员可以依据具体的业务需求选用相应的组件完成业务流程。

### 2-9.使用rest方式处理文件服务

在实际应用中，上传下载非常常见，同时，也有非常多的不同的处理策略，用户上传的文件，因为空间有限，一般不会存储在应用服务器中，多使用第三方存储服务，在目前前后端分离的架构中，前端多使用SPA架构，即单页服务架构，表单提交之后跳转的业务非常少见，本案例中，模拟异步文件提交请求。

2.9.1 文件上传

依据上述案例中的方法，先开发测试用例



@RunWith(SpringRunner.class)

@SpringBootTest(classes = DemoApplication.class)

public class FileControllerTest {

    @Autowired

    private WebApplicationContext webApplicationContext;

    // 模拟MVC环境相关

    private MockMvc mockMvc;

    /\*\*

     \* 每次方法执行之前都模拟一个MVC环境

     \* 每次运行都相当于服务器重启

     \*/

    @Before

    public void setUp() {

        mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();

    }

    @Test

    public void whenFileUploadSuccess() throws UnsupportedEncodingException, Exception {

        String result = mockMvc

                .perform(MockMvcRequestBuilders.fileUpload("/file")

                .file(new MockMultipartFile("file","test.txt","multipart/form-data", "hello springBoot FileUpload".getBytes("utf-8"))))

                .andExpect(MockMvcResultMatchers.status().isOk())

                .andReturn()

                .getResponse()

                .getContentAsString();

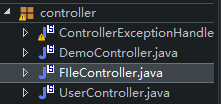
        System.out.println(result);

    }

}

在测试用例whenFileUploadSuccess中，像/file API发送文件请求，直接伪造文件file，其中MockMultipartFile方法第一个参数为文件名，第二个参数为文件内容。也就是说，像API发送了一个JSON形式的字符串，将文件也放入JSON中，以file为文件名，字符串为文件内容。执行测试用例，得到400结果，表示测试用例编码无误。

接着编写文件上传服务。



注意，在本案例中没有对文件上传做任何限制，在实际的业务中，文件上传存在非常多的问题。在后文中，会使用Spring Security处理文件上传服务。

@RestController

@RequestMapping("/file")

public class FIleController {

    @Autowired

    private HttpServletRequest request;

    @Autowired

    private HttpServletResponse response;

    @PostMapping()

    public FIleInfo fileUpload(MultipartFile file) throws IllegalStateException, IOException {

        String fileName = file.getOriginalFilename();

        System.out.println("parmName: " + file.getName());

        System.out.println("fileContentType: " +file.getContentType());

        System.out.println("fileName: " + fileName);

        File folder = new File("D:\\\\ ", "/upload");

        if (!folder.exists()) {

            folder.mkdirs();

        }

        String ext = FilenameUtils.getExtension(fileName);

        File uploadFile = new File(folder, UUID.randomUUID()+"."+ ext);

        file.transferTo(uploadFile);

        FIleInfo fIleInfo = new FIleInfo(uploadFile.getAbsolutePath());

        return fIleInfo;

    }

}

此方法中fileUpload方法接受MultipartFile对象作为参数，同时返回FIleInfo对象，此对象为自定义的，用来存放文件信息的实体类，在实际业务中，当文件上传之后，后台会经过验证处理，之后，会对成功上传的文件进行重命名，并存储到响应路径中，其中，新文件名，原始文件名，上传路径等信息需要保留下来，存入数据库，某些情况下需要返回给前端，此对象就是用来封装此类数据，因为在本案例中并不涉及Dao层实现，故这里只存放文件路径filepath

public class FIleInfo {

    private String filePath;

}

再接着看fileUpload方法，MultipartFile对象为springmvc默认的用来存放文件信息的对象在此方法中，将该对象中的常见字段输出，并将文件保存到D盘的upload文件夹中。

执行测试用例，得到结果

preHandle

org.springframework.web.method.HandlerMethod

parmName: file

fileContentType: multipart/form-data

fileName: test.txt

postHandle

costTime: 70

afterCompletion

costTime: 70

ex: null

{"filePath":"D:\\upload\\feb9b4c4-a957-40cd-8e88-a388eb6080f7.txt"}

从结果出看到，在上一小节中编写的组件依然在起作用，为了方便，可以先把组件注释掉，此外，文件上传服务成功。

2.9.2 文件下载

此服务不用编写测试用例，直接在浏览器中输入API地址测试服务是否完成。接着编写FileController.

@GetMapping("/{fileName}")

public void fileDownload(@PathVariable String fileName) throws IOException {

String path = "D:\\\\upload\\\\feb9b4c4-a957-40cd-8e88-a388eb6080f7.txt";

File downloadFile = new File(path);

System.out.println("downFilePath:" + downloadFile.getAbsolutePath());

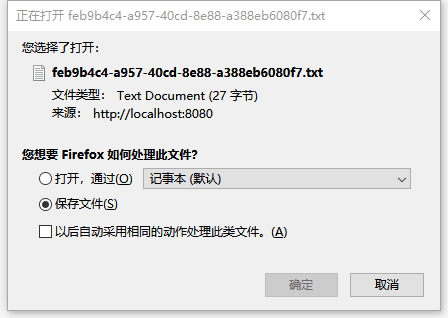
response.setContentType("appliocation/x-download");

response.addHeader("Content-Disposition", "attachment;filename=" + "feb9b4c4-a957-40cd-8e88-a388eb6080f7.txt");

FileUtils.copyFile(downloadFile, response.getOutputStream());

}

启动服务，在浏览器中访问/file/1，得到响应



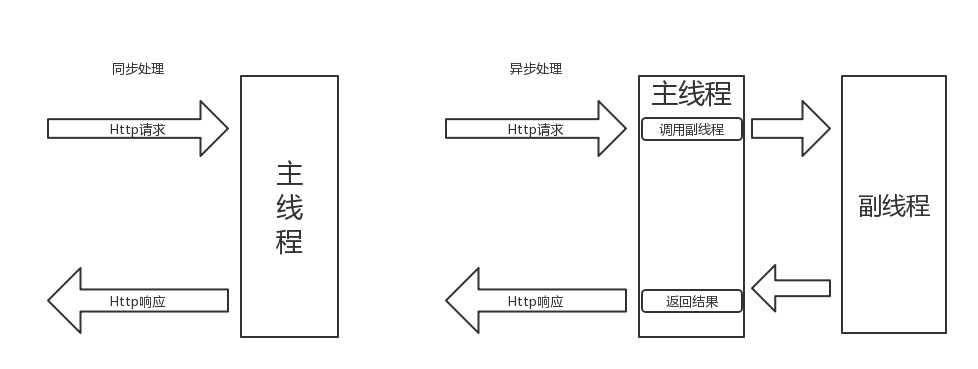
在本案例中，因为不涉及到DAO操作，所以无论收集到什么下载请求，都是返回固定的实例文件，在实际的工程中，上传的文件会记录原始文件名以及上传者ID，以供下载的时候，通过文件名查询上传时文件存放路径。

文件上传下载功能和传统的SSM并没有太多的区别，在当前用例中，并没有用上SpringSecurity的功能，只是介绍了SpringMVC的基本业务，在后续的章节中，会完善上传下载逻辑。

### 2-10.使用多线程提高rest服务性能

在实际开发的过程中，使用多线程来提高服务的性能是非常有必要的，Tomcat线程池默认情况下上限为200个，换言之，允许200个用户同时访问，第201个用户即开始排队等待服务，如果等待时间超过上限，则抛出超时异常，按照统计模型，整个网站注册用户大概在4000个左右，如果单纯的对tomcat线程池进行修改，其意义也并不大。并不能充分利用起服务器性能，而在SpringMVC框架中，有停供多线程的API执行方式。

2.10.1使用Runnable异步处理Rest服务



在演示案例之前，先说明一下同步处理与异步处理的区别，本节开篇有提到，tomcat线程池只有200个，而单方面提高线程池数量对性能并没有太大的帮助，那么思路就很简单了，在controller中，接收到请求，开启副线程处理，副线程处理完毕，返回结果，换言之，在这种模型下，tomcat线程池的功能非常简单，只需要分发请求到控制器，而具体业务的执行交由控制器开启的副线程处理，tomcat主线程直接返回，释放，等待下一个用户接入。

举个例子，如果业务处理需要1秒钟，那么，1秒之后才能给第201个用户提供服务，采用上述模型，那么，业务处理时间和主线程解耦，主线程不需要等待业务返回，极大地提高了服务器运行效率。

看一下代码实现。

@RestController

public class AsyncController {

    private static final Logger log = LoggerFactory.getLogger(AsyncController.class);

    @GetMapping("/order/{id}")

    public Callable<String> getOrder(@PathVariable String id) throws InterruptedException {

        log.info("------主线程开始------");

        Callable<String> result = new Callable<String>() {

            @Override

            public String call() throws Exception {

                // TODO Auto-generated method stub

                log.info("副线程开启");

                Thread.sleep(1000);

                log.info("副线程结束");

                return "success";

            }

        };

        log.info("------主线程返回------");

        return result;

    }

}

本用例中使用logger输出信息，使用日志框架输出信息时，会将当前系统时间精确到毫秒级别输出，用来观察API执行消耗。

当前方法返回一个Callable对象，如果要使用异步请求，那么固定会返回当前对象，将方法实际上的返回值放入泛型中，本例中，方法本来的返回值固定为“success”字符串。

在当前方法中，生成一个callable对象，将原本的返回值放入泛型，在callable对象的call方法中，执行API的业务逻辑，比如调用业务层方法等。此时，SpringMVC在扫描到当前方法返回值为callable对象时，当tomcat线程池将请求分配到API上，就直接返回了，进入方法中，call方法会创建另一个副线程执行业务逻辑，模拟业务处理流程，让线程休眠1秒钟，当副线程执行完毕，到getOrder方法返回的时候，将callable对象一起返回到前端。

执行测试用例来看看结果，启动服务，在浏览器中输入localhost:8080/order/1得到结果：



对于前端而言，异步API和同步API并没有太大的区别，都是消耗1秒多一点的时间，控制台输出

2018-12-29 10:15:37.638 INFO 13088 --- [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring FrameworkServlet 'dispatcherServlet'

2018-12-29 10:15:37.639 INFO 13088 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization started

2018-12-29 10:15:37.668 INFO 13088 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization completed in 29 ms

2018-12-29 10:15:37.717 INFO 13088 --- [nio-8080-exec-2] n.s.demo.web.async.AsyncController : ------主线程开始------

2018-12-29 10:15:37.718 INFO 13088 --- [nio-8080-exec-2] n.s.demo.web.async.AsyncController : ------主线程返回------

2018-12-29 10:15:37.726 INFO 13088 --- [ MvcAsync1] n.s.demo.web.async.AsyncController : 副线程开启

2018-12-29 10:15:38.726 INFO 13088 --- [ MvcAsync1] n.s.demo.web.async.AsyncController : 副线程结束

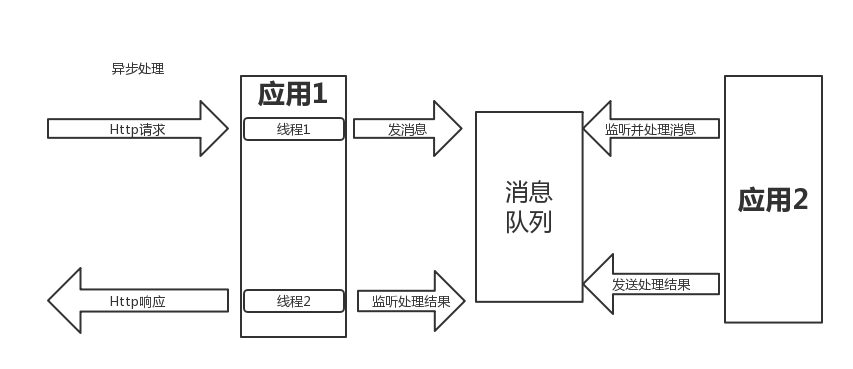
观察控制台输出结果，nio-8080-exec开头的线程为tomcat主线程，主线程从2018-12-29 10:15:37.717 开始到2018-12-29 10:15:37.718 结束，即1毫秒就释放了，然后副线程开始，副线程1000毫秒后结束，最终返回前端，很明显，当前架构，相对于同步架构，优秀了很多，此时，使用多线程处理，极大地提高了tomcat的负载。

2.10.2使用DeferredResult处理异步服务

在上述小节中，已经可以使用callable处理异步服务了，为什么还需要使用DeferredResult呢，有什么必要性？

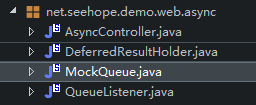
在使用callable中，服务的编写还是有其局限性，副线程一定要在主线程内被调起，然而在实际业务中，往往要复杂一些，依然使用下单业务举例，接受下单请求的应用，和处理下单业务的应用并不是一台服务器，那么http请求通过消息队列进行数据传递，应用2坚挺到消息队列中有业务，那么读取数据处理下单业务，处理完毕之后再把返回结果存放到消息队列中，应用1中有线程2在监听消息队列，当返现消息队列中有订单业务处理完毕的结果的时候，再将响应返回给前端。

在当前业务场景中，应用1和应用2完全是隔离的，甚至都不在一台服务器上，互相之间并不知道对方的存在。在这种情况下，callable的使用就不符合业务了。需要使用另一个对象DeferredResult来处理异步服务。



此时，编写代码来模拟应用场景，在此测试用例中，并不会真正的开发应用2并使用消息队列，而是使用代码来模拟消息队列，编写一个模拟队列，用来存放下单信息以及订单处理完成信息，其次，在主线程中，编写处理http请求逻辑，然后编写线程2中的代码，监听模拟队列中的队列完成消息。最后，编写DeferredResultHolder对象，用来存放线程1中的DeferredResult，在整个业务流程中，存储DeferredResult数据以供线程2使用，。

首先，编写模拟队列MockQueue：



@Component

public class MockQueue {

    private String placeOrder;

    private String completeOrder;

    private static final **Logger** log = LoggerFactory.getLogger(MockQueue.class);

    public String getPlaceOrder() {

        return placeOrder;

    }

    public void setPlaceOrder(String placeOrder) throws InterruptedException {

        new Thread(() -> {

            log.info("接收到下单请求" + placeOrder);

            try {

                Thread.sleep(1000);

            } catch (InterruptedException e) {

                // TODO Auto-generated catch block

                e.printStackTrace();

            }

            this.completeOrder = placeOrder;

            log.info("下单请求处理完毕" + placeOrder);

        }).start();

    }

    public String getCompleteOrder() {

        return completeOrder;

    }

    public void setCompleteOrder(String completeOrder) {

        this.completeOrder = completeOrder;

    }

}

因为当前系统中不存在应用2，即订单处理业务，所以在此消息队列中，设置placeOrder字段的时候，模拟应用2处理业务，并将处理结果存放回消息队列流程，在此方法中休眠1秒，处理完毕后，将状态设置到completeOrder字段以供应用1的线程2监听。

接着，编写DeferredResultHolder对象：

@Component

public class DeferredResultHolder {

    private Map<String, DeferredResult<String>> map = new HashMap<>();

    public Map<String, DeferredResult<String>> getMap() {

        return map;

    }

    public void setMap(Map<String, DeferredResult<String>> map) {

        this.map = map;

    }

}

此对象非常简单，只是用来在应用1的线程1以及线程2之间传递API返回的DeferredResultHolder对象结果。只包含有一个字段，hashmap集合，第一个String类型的参数类似于订单号。第二个参数用来存放DeferredResult即订单处理结果。

接着，修改API订单处理逻辑

@RestController

public class AsyncController {

    @Autowired

    private MockQueue mockQueue;

    @Autowired

    private DeferredResultHolder deferredResultHolder;

    private static final Logger log = LoggerFactory.getLogger(AsyncController.class);

    @GetMapping("/order/{id}")

    public DeferredResult<String> getOrder(@PathVariable String id) throws InterruptedException {

        log.info("主线程开始");

        String orderNumber = RandomStringUtils.randomNumeric(8);

        mockQueue.setPlaceOrder(orderNumber);

        DeferredResult<String> result = new DeferredResult<>();

        deferredResultHolder.getMap().put(orderNumber, result);

        /\*Callable<String> result = new Callable<String>() {

            @Override

            public String call() throws Exception {

                // TODO Auto-generated method stub

                log.info("副线程开启");

                Thread.sleep(1000);

                log.info("副线程结束");

                return "success";

            }

        };\*/

        log.info("主线程返回");

        return result;

    }

}

主线程业务非常简单，直接将生成订单请求通知消息队列，因为并不是真的存在有消息队列，所以需要手动调用队列的set方法，实际业务中只需要将请求存放到队列中即可，同时，订单号也是由应用2生成的，而不是在主线程中生成，这里为了模拟业务，所以由主线程生成订单号，最后，将生成的订单号以及结果放入DeferredResultHolder对象，方便在线程2中使用消息队列中的结果。最后，返回result，在此代码中，是看不到有另一个应用存在的。

最后，来编写线程2代码，即监听队列的消息。

@Component

public class QueueListener implements ApplicationListener<ContextRefreshedEvent> {

    @Autowired

    private MockQueue mockQueue;

    @Autowired

    private DeferredResultHolder deferredResultHolder;

    private static final Logger log = LoggerFactory.getLogger(QueueListener.class);

    @Override

    public void onApplicationEvent(ContextRefreshedEvent event) {

        // TODO Auto-generated method stub

        new Thread(() -> {

            while (true) {

                if (StringUtils.isNotBlank(mockQueue.getCompleteOrder())) {

                    String orderNumber = mockQueue.getCompleteOrder();

                    log.info("返回订单处理结果" + orderNumber);

                    deferredResultHolder.getMap().get(orderNumber).setResult("place order success");

                    mockQueue.setCompleteOrder(null);

                } else {

                    try {

                        Thread.sleep(100);

                    } catch (InterruptedException e) {

                        // TODO Auto-generated catch block

                        e.printStackTrace();

                    }

                }

            }

        }).start();

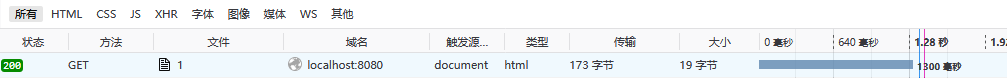
    }

}

在此监听器中，监听spring容器启动事件，当服务启动时，创建新的线程，在新的线程中，不断循环遍历模拟队列中的字段getCompleteOrder中是否有值，如果有值，那么从DeferredResultHolder中将应用2的返回结果取出，返回到前端。同时，将完成状态清空。如果字段为null，则休眠100毫秒之后继续监测。

至此，整个业务流程处理完毕，在当前流程中，实际上有3个线程，主线程接受HTTP请求，之后操作消息队列，第二个线程在消息队列中，即，应用2的处理业务，第三个线程在监听器中，监听消息队列的返回结果，并等待返回给前端，应用1和应用2之间通过holder来传递数据。

启动服务，测试代码，访问 localhost:8080/order/1此时，对于前端来说，依然是



控制台输出结果

2018-12-29 12:01:17.069 INFO 22536 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring FrameworkServlet 'dispatcherServlet'

2018-12-29 12:01:17.070 INFO 22536 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization started

2018-12-29 12:01:17.099 INFO 22536 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization completed in 29 ms

2018-12-29 12:01:17.160 INFO 22536 --- [nio-8080-exec-1] n.s.demo.web.async.AsyncController : 主线程开始

2018-12-29 12:01:17.163 INFO 22536 --- [nio-8080-exec-1] n.s.demo.web.async.AsyncController : 主线程返回

2018-12-29 12:01:17.163 INFO 22536 --- [ Thread-10] net.seehope.demo.web.async.MockQueue : 接收到下单请求11904903

2018-12-29 12:01:18.163 INFO 22536 --- [ Thread-10] net.seehope.demo.web.async.MockQueue : 下单请求处理完毕11904903

2018-12-29 12:01:18.228 INFO 22536 --- [ Thread-7] n.seehope.demo.web.async.QueueListener : 返回订单处理结果11904903

分析结果，和使用callable结果相同，但是代码从结构上来说完成了解耦，不需要再主线程中直接调用副线程。同时，一个共有3个线程参与了整个逻辑。流程处理完成。

以上业务流程较为复杂，如果不搞清楚能理解的话可以将本小结一开始的业务图，自己画一遍，加深一下对整个业务流程的理解。搞清楚哪个应用是用来做什么处理的，

2.10.3异步处理配置

针对于上一小节的应用，如果需要对异步API做拦截处理，需要单独在配置文件中配置，因为同步Controller与同步Controller所实现的类不同。

在WebConfig.java中继续编写

@Override

    public void configureAsyncSupport(AsyncSupportConfigurer configurer) {

        // TODO Auto-generated method stub

        configurer.setDefaultTimeout(0);

        configurer.registerCallableInterceptors(null);

        configurer.registerDeferredResultInterceptors(null);

        configurer.setTaskExecutor(null);

    }

本用例中并没有真的使用拦截器去拦截异步请求，在异步控制器的处理中，不同于同步控制器的配置主要就是这4个，setDefaultTimeout方法用来配置异步请求超时时间，registerCallableInterceptors方法用来给方法返回类型是callable类型的API添加拦截器，registerDeferredResultInterceptors方法用来给返回类型是DeferredResult类型的API添加拦截器，最后一个setTaskExecutor方法针对于返回callable类型的API，用来指定call方法中的线程池，springMvc提供给callable的线程池只是一个简单的，不可重用的线程池，如果需要对callable线程池进行优化，可以提供其他的线程池实现。

本小结只是描述了当前4个API的作用，在具体的业务中，可以根据具体的需求具体配置。

### 2-11.使用swagger自动生成文档

在实际的开发过程中，采用前后端分离的架构下，前后端需要沟通好API的参数以及返回结果，在传统过程中，这一工作非常复杂，需要前端后台先沟通好接口文档，如果后台或者前端在开发的过程中，如果有对API进行更改，那么还需要更新接口文档，而更新文档又有一些延时性，那么，在其中需要浪费较多的沟通成本，除此之外，当用户开发好API之后，如果不是使用测试驱动编程的开发规范，当需要测试RESTful风格API时，往往需要有很多额外的工作，此时，如果后台使用spring框架，那么可以引入swagger插件，生成文档。

当spring中引入swagger，配合注解扫描所有Controller，swagger插件会自动生成一篇基于HTML的接口文档，在此测试文档中，可以直接传入参数并调用对应API。

使用swagger的方法非常简单，在工程创建的初期，即第一张的内容中，已经引入了swagger的依赖。如果要开启swagger功能，只需要在主程序中加上注解即可。

@SpringBootApplication

@EnableSwagger2

@ComponentScan("net.seehope")

public class DemoApplication {

    public static void main(String[] args) {

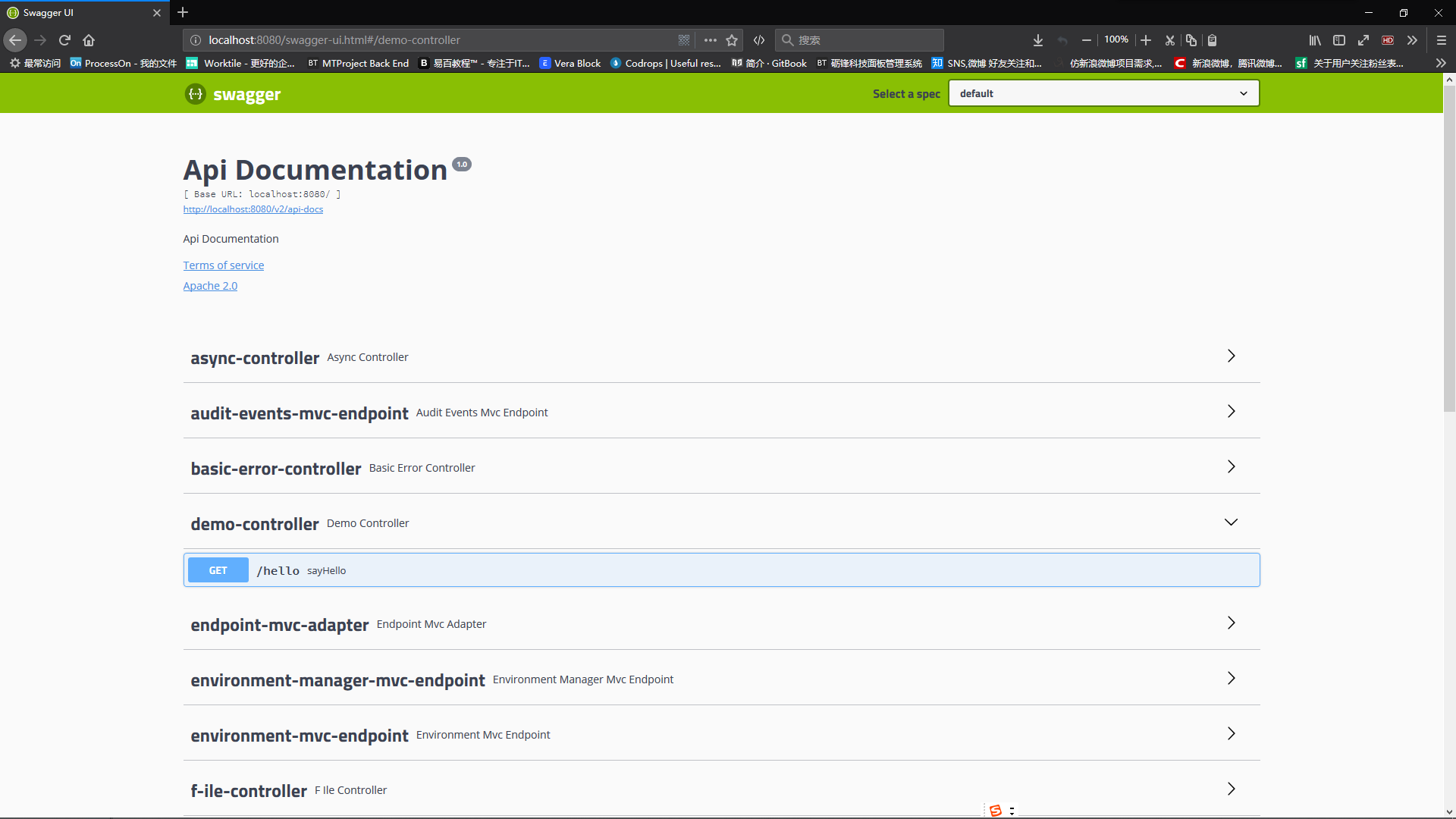
        SpringApplication.run(DemoApplication.class, args);

    }

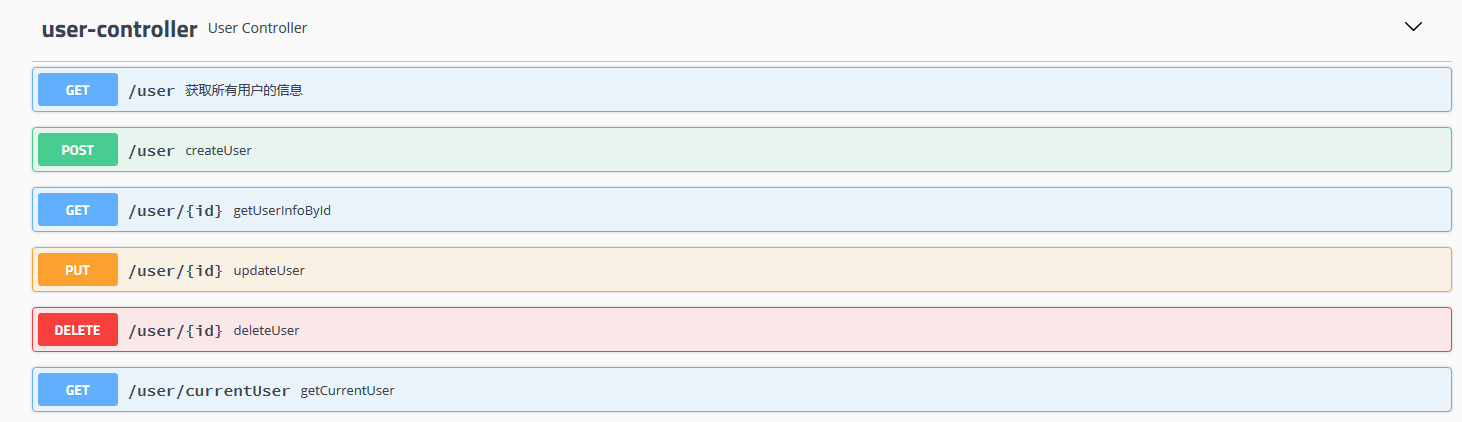
}

@EnableSwagger2注解表示此服务启用swagger组件，配合ComponentScan注解，swagger会在("net.seehope")包中扫描所有@Controller注解，生成接口文档。

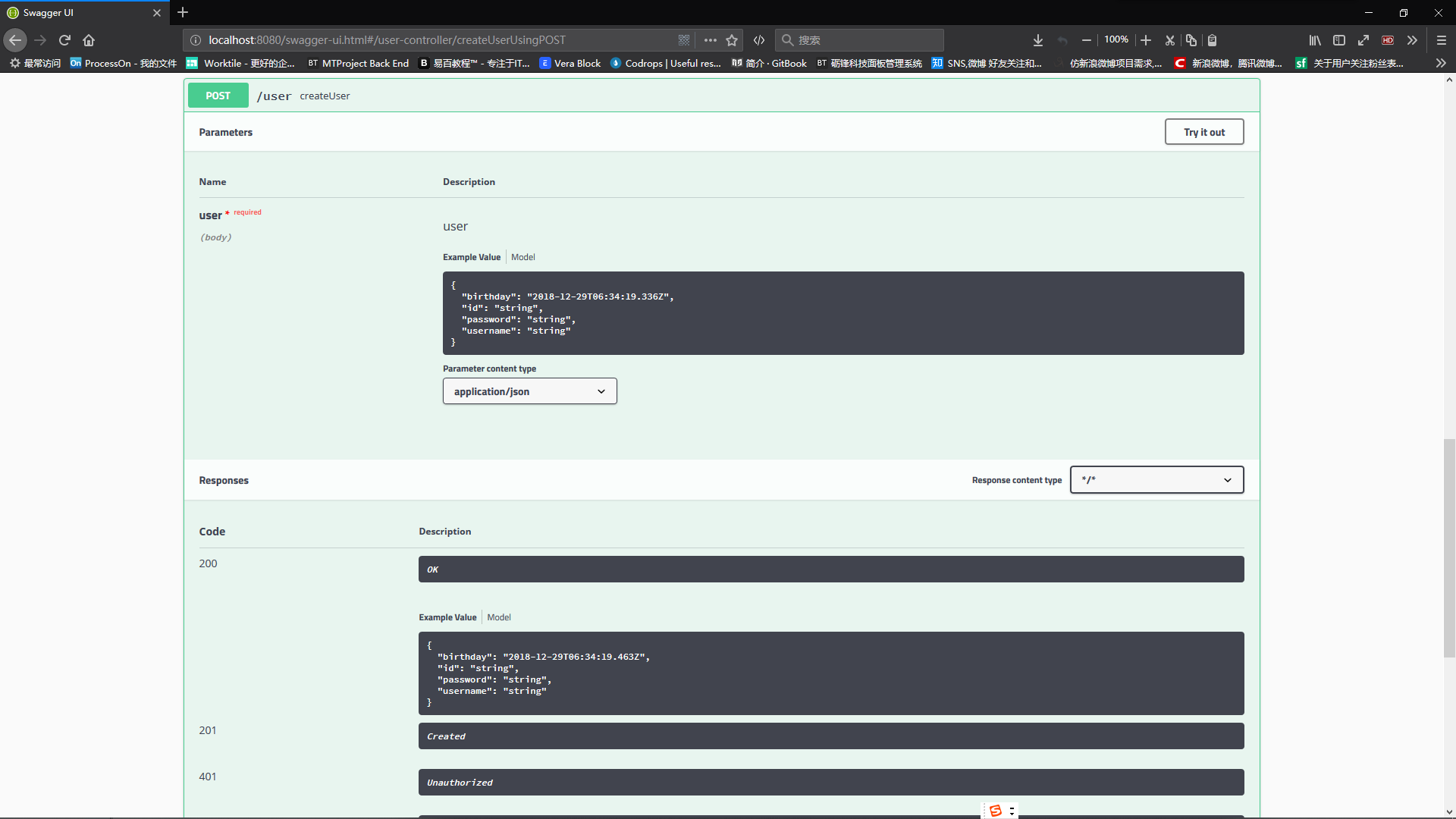
启动服务，访问<http://localhost:8080/swagger-ui.html>得到结果。



得到如下结果，其中，包含有一些springboot工程自带的Controller，先忽略这些系统自定的控制器，看看自定义的控制器UserController

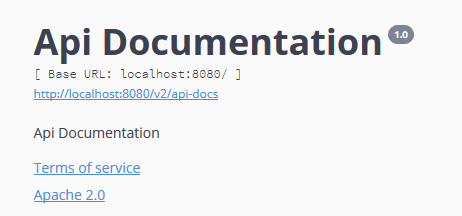


展开Post /user请求



得到如下界面，很明显，在当前html中，可以对此API进行测试，单击Try it out 按钮，发现参数变为可输入状态，同时提供了样板数据，修改样板之后，点击向此API发送请求，同时在下面可以看到服务响应结果，整个业务流程看起来非常的清晰，服务有什么API可以被访问，地址是多少，参数是多少，请求方式是什么，一目了然，同时，因为此文档是扫描工程自动生成，换言之，当服务代码更新之后，文档也会自动更新，大大降低了前后端工程师之间的沟通成本，同时，也可以借助此插件，对所有API进行测试。比使用RESTED插件要方便一些。

在本案例中，只是使用了swagger的基本功能，即生成API文档，并没有深入研究swagger的用法，实际上，Swagger也有提供一系列的注解，类级别，方法级别，参数级别，用来说明，类，方法，或者参数的意义，这些注解中的信息，会出现在文档的相应位置，可能读者已经发现在GET /user 这个API中，是中文说明的API，这个功能就是通过注解实现的。在文档中的头

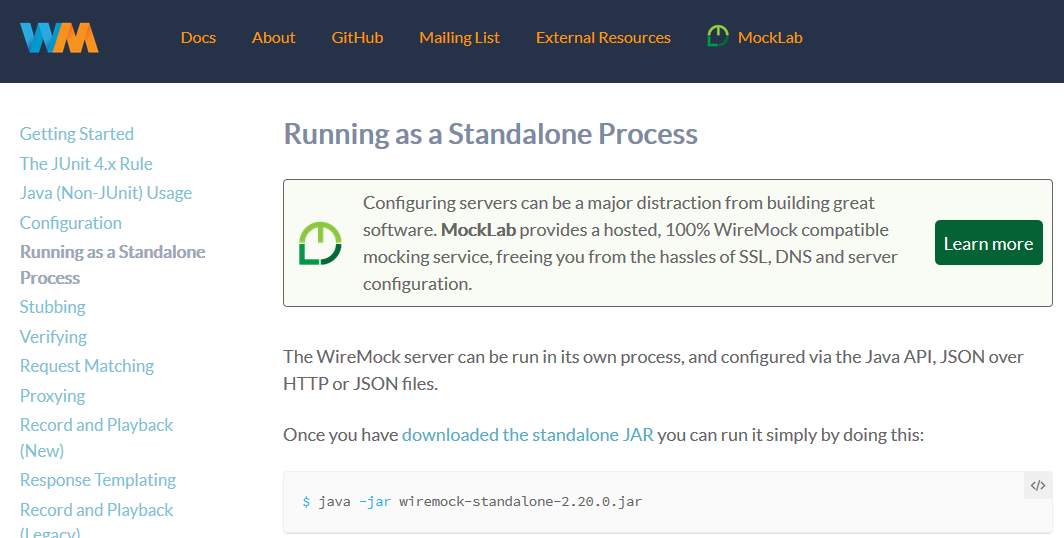


以及API中的描述，都是支持自定义的。本篇不做详细的探讨，有兴趣的可以自行搜索swagger注解使用。

### 2-12.使用WireMock伪造rest服务

使用WireMock快速伪造RESTful服务，在目前前后端分离的架构中，在后台API还没有开发好的时候，提供给前端访问，那么，既然真的要写API有何必提供一个伪造的API呢，前端在开发的过程中不能自行伪造测试数据吗？这是因为，后台的RESTful服务可能不光被一种终端所调用，比如APP，IOS,HTML，都要伪造数据，那么就相当于，伪造数据这个工作，每个人都要做一次，在此之中，又因为每个工程师对API的理解可能有所不同，写出来的服务良莠不齐，此时，最好还是由后台提供统一的数据，前端的开发效率就能得到极大的提升。那么，可不可以在开发之前，现在controller中把返回数据写死呢，这也是可以的，但是因为在开发过程中，返回数据的字段也可能是变化的，每当出现这种变化，就需要修改控制器并重新部署代码，而，WireMock是一个独立的客户端，在代码中，可以告诉WireMock服务收到什么请求时给出什么响应，而服务器是一直运行，不用反复重启的，当有这种服务器存在时，各个前端都不需要去模拟数据了，只需要访问WireMock服务即可，而不是真的就访问到后台API，当后端真正开发好之后，对于前端而言，可能只需要修改其中一个配置，就能对接上真正的服务。

实际上WireMock是一个非常强大的工具，登录官网，获取WireMock服务



点开DOCS选项卡，选择Running as a Standalone Process 点击下载，得到wiremock-standalone-2.20.0.jar。当jar包下载好之后，即可通过java -jar指令启动

得到结果，其中--port 8081参数表示伪造服务从8081端口启动。

D:\WorkSpace\Tools>java -jar wiremock-standalone-2.20.0.jar --port 8081

port: 8081

enable-browser-proxying: false

disable-banner: false

no-request-journal: false

verbose: false

接下来编写代码，交由wiremock服务器伪造API。代码之中，wiremock的依赖已经预先导入了。

public class MockServer {

    public static void main(String[] args) {

        WireMock.configureFor(8081);// 绑定正在运行的wiremock服务

        WireMock.removeAllMappings();// 先清空之前所有的API

        WireMock.stubFor(

                get(urlPathEqualTo("/order/1")).willReturn(aResponse().withBody("{\"id\":1}").withStatus(200)));

    }

}

创建API伪造类MockServer

public class MockServer {

    public static void main(String[] args) {

        WireMock.configureFor(8081);// 绑定正在运行的wiremock服务

        WireMock.removeAllMappings();// 先清空之前所有的API

        WireMock.stubFor(

                get(urlPathEqualTo("/order/1")).willReturn(aResponse().withBody("{\"id\":1}").withStatus(200)));

    }

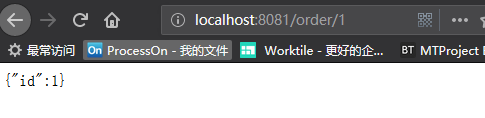
}

首先，先绑定wiremock服务，其次，清空之前所有的API，所有伪造的API卸载当前类中，然后通过stubFor方法伪造API，get方法表示当前请求方式为get，urlPathEqualTo方法表示伪造的请求地址willReturn表示伪造的响应状态withBody方法表示伪造的响应内容，withStatus方法表示伪造的响应状态码。执行代码，得到控制台输出

15:33:45.192 [main] DEBUG org.apache.http.impl.conn.PoolingHttpClientConnectionManager - Connection [id: 0][route: {}->http://localhost:8081] can be kept alive indefinitely

15:33:45.192 [main] DEBUG org.apache.http.impl.conn.PoolingHttpClientConnectionManager - Connection released: [id: 0][route: {}->http://localhost:8081][total kept alive: 1; route allocated: 1 of 5; total allocated: 1 of 50]

表示请求伪造完成。同时，在浏览器中访问



之后，将MockServer中的代码重构一下

public class MockServer {

    public static void main(String[] args) throws IOException {

        configureFor(8081);

        removeAllMappings();

        mockServer("/order", "01");

    }

    public static void mockServer(String url, String classPathObjectModelPath) throws IOException {

        ClassPathResource classPathResource = new ClassPathResource(

                "mock/response/" + classPathObjectModelPath + ".txt");

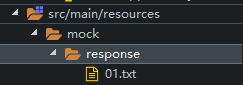
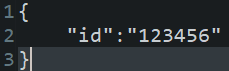
        String content = StringUtils.join(FileUtils.readLines(classPathResource.getFile(), "utf-8").toArray(), "\n");

        stubFor(get(urlPathEqualTo(url)).willReturn(aResponse().withBody(content).withStatus(200)));

    }

}

在工程中，以文件的形式封装伪造的请求，并将创建请求方法封装起来

在此执行主方法，创建请求伪造请求。

至此，使用wiremock伪造请求介绍完成。本节中对于wiremock的介绍并没有太过深入，如果对wiremock有兴趣的话，可以浏览官方文档或者自行查询资料。

## 使用spring security开发基于表单的登录

### 3-1.简介

经过第2章的讲解中，已经开发出了可供不同终端使用的RESTful风格的API，但是，在当前应用中，任何用户只要知道服务器路径，那么即可访问服务中的任何资源，相当于在“裸奔“”状态，这种情况，在真实企业级应用中，肯定是不允许存在的。而SpringSecurity的应用主要就是在于处理这个问题，即对用户的认证以及授权。这两个主要区域是Spring Security的两个目标。“认证”，是建立一个主体的过程（一个“主体”一般是指用户，设备或一些可以在你的应用程序中执行动作的其他系统）。“授权”指确定一个主体是否允许在你的应用程序执行一个动作的过程。为了抵达需要授权的点，主体的身份已经由认证过程建立。

SpringSecurity 核心功能

1.认证（你是谁）

2.授权（你能干什么）

3.攻击防护（防止伪造身份）

本章内容，主要解释SpringSecurity的认证功能，解释SpringSecurity基本原理，实现用户名+密码认证，实现手机号+短信认证方法。在本章中，也将解释部分SpringSecurity源码。

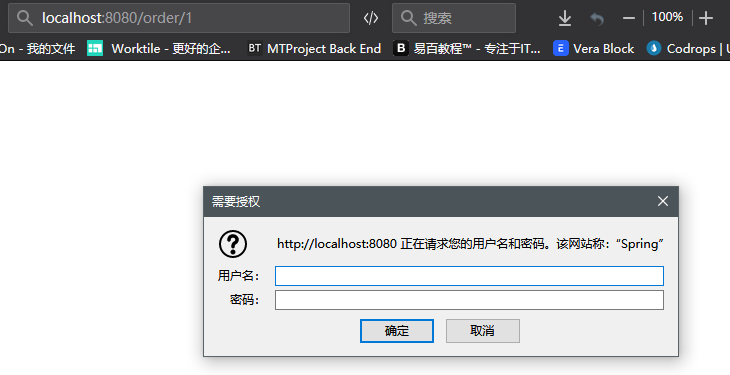
### 3-2.Spring Security 基本原理 SpringSecurity业务流程比较复杂，先从实际的案例开始编写代码，再逐渐讲解SpringSecurity的原理。

在第一章中，搭建SpringSecurity时配置的application.properties中，关闭了默认的安全的配置，先将配置修改为，之后直接启动服务

#在不使用security的时候先关闭该功能

security.basic.enabled=true

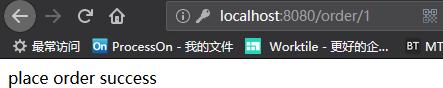
此时，无论访问应用中的哪个资源，都会出现



也就是说，默认情况下，SpringSecurity限制了所有资源的访问，访问任意资源都需要输入用户名与密码，此时用户名固定为user，密码在控制台日志中

Using default security password: a9cae6c7-1673-485f-aecc-ad2fd4e1c35e

输入用户名密码之后跳转到

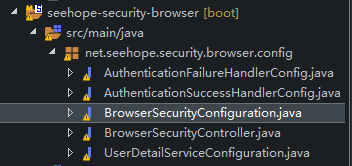


很明显，默认情况下的流程肯定是不满足实际情况的，先来分析一下整个业务。

当用户第一次访问/order/1的情况下，用户身份没有经过认证，访问请求被拦截，弹出basic认证窗口，等待用户认证身份信息，认证成功之后允许访问API。这就是basic认证。

再来看另外一种最简单的认证方式，基于表单的用户名密码验证，本章节最主要的目的是基于SpringSecurity开发出一个可重用的安全模块，换言之，熟悉业务之后，本模块可以直接复用到新的工程中，只需要更改少量代码，即可完成安全认证等功能，所以，在本例中，和安全配置有关的代码，配置在browser工程中，少量常量，配置在core工程中。使用Demo工程模拟真实工程，在demo工程中引用browser中的安全模块。

先在browser下创建SpringSecurity配置文件



@Configuration

public class BrowserSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override

    protected void configure(HttpSecurity http) throws Exception {

        // TODO Auto-generated method stub

        http.formLogin()//http请求使用表单登录方式验证

        .and()

        .authorizeRequests()//表示接下来都是对请求的授权配置

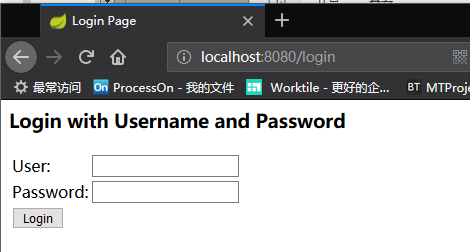
        .anyRequest()//任何请求

        .authenticated();//都需要身份认证

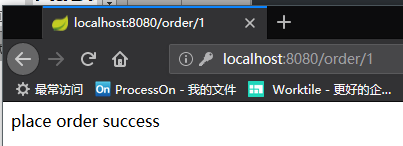
    }

}

在此配置文件中，配置了非常基础的安全环境，即对HTTP请求，使用表单认证方式，所有的请求，都需要进行身份认证。启动服务，再次访问API。得到结果



类似于basic认证，用户名固定为User，密码为默认密码，且出现在控制台上，输入用户名密码，此密码为随机密码，每次启动服务更新，之后得到结果。



成功访问到API，分析一下基于表单的认证流程，因为在配置文件中配置了所有请求都需要认证，所以在访问/order/1的时候，框架检测到没有认证信息，所以请求被拦截，框架记录用户本次访问的路径，同时，将请求重定向到/login，框架返回默认的同步表单提供用户认证，认证成功之后，框架再重定向到之前记录的用户访问地址。

那么，如果把代码中的formlogin改成如下所示，那么实际上就是SpringSecurity的默认配置。

http.httpBasic()

.and()

.authorizeRequests()//表示接下来都是对请求的授权配置

.anyRequest()//任何请求

.authenticated();//都需要身份认证

至此，SpringSecurity的基本配置使用完成了，在之后的章节中，会逐步扩展当前实例。

那么，解释一下安全框架在以上两个业务流程中起到的作用，在基于表单登陆的流程中。实质上，整个安全框架实质上就是一个过滤器连，也就是一组过滤器和拦截器，所有的请求，都会经过框架的过滤器，同理，响应也会经过该过滤器链，在服务部署的时候，Springboot会自动将过滤器链部署到应用中，而在链中，有很多的过滤器，大致上分为3类过滤器。

首先，最核心的，就是以下三个绿色的过滤器，UsernamePasswordAuthenticationFilter,

BasicAuthenticationFilter以及同类的其余过滤器，这些过滤器的作用就是用来验证用户的身份，每个过滤器负责处理一种认真方式，比如之前基于表单的身份认证，以及基本验证方式，分辨对应了第一第二个过滤器，换言之，如果需要添加其他的验证方式，那么，就需要在过滤器链中添加新的过滤器，绿色的过滤器中也负责检查当前的业务中，是否有当前过滤器所需要的信息，比如，对于UsernamePasswordAuthenticationFilter过滤器而言，就会检测，当前请求是否是登录请求，如果是，请求中是否有携带用户名和密码，如果带用户名和密码，如果有用户名密码，那么当前过滤器会尝试使用当前用户名密码认证，如果没有携带用户名密码，那么就会放行当前请求，交给下一个认证方式过滤器认证，BasicAuthenticationFilter会检查当前请求的请求头中，是否有BasicAuthentication的信息，如果有，会尝试取出数据，做BASE64解码，取出用户名密码，尝试身份认证，同时在SpringSecurity中还有提供很多其他的认证方式，按上以上说明的流程，请求会经过一个一个的过滤器，任何一个过滤器，成功的完成了认证之后，都会在当前请求上做一个标记，表示用户认证成功。

请求经过这些绿色的过滤器之后，最终会到橙色的FilterSecurityInterceptor拦截器中，这个过滤器一定在整个过滤器链最终一环，他是整个应用的守门人，在该过滤器之后，就是应用API了，这个过滤器就将决定当前请求是否能真的访问到后面的API，此过滤器会检查，在SpringSecurity中的配置文件中是否如何配置的。

在当前案例中。

@Override

protected void configure(HttpSecurity http) throws Exception {

// TODO Auto-generated method stub

http.formLogin()//http请求使用表单登录方式验证

.and()

.authorizeRequests()//表示接下来都是对请求的授权配置

.anyRequest()//任何请求

.authenticated();//都需要身份认证

}

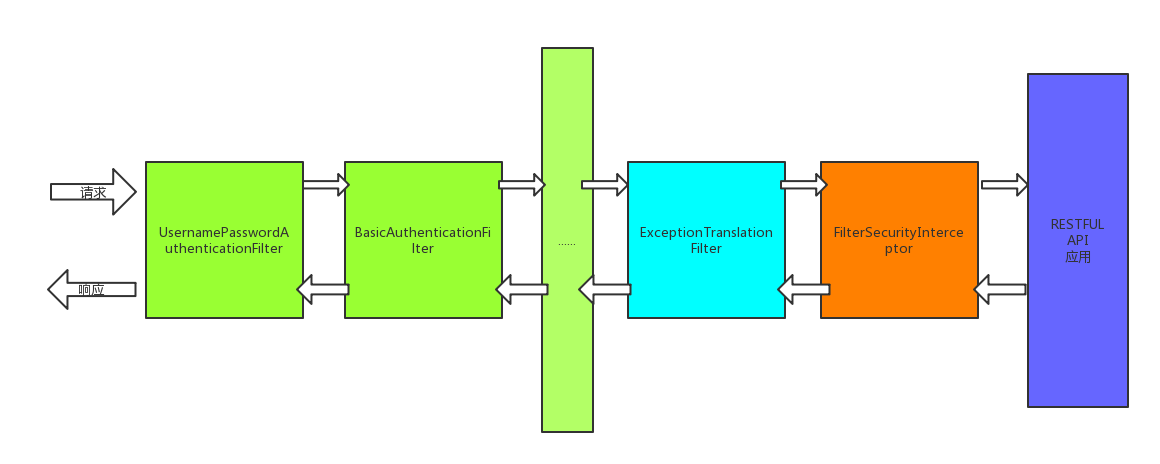
配置了所有的请求都需要身份认证，那么，此拦截器就会检测当前请求是否经过了绿色过滤器的认证，即检查标记，实际上，这一段配置可以写的非常复杂，比如可以配置某类请求只有VIP用户才允许访问等，这些配置都会被拦截器读取，拦截器会根据这些配置做判断。如果判断通过，那么访问到具体API，如果不过，根据不过的原因，会抛出不同的异常。

比如在当前配置中，配置了所有请求都需要身份认证，那么，如果当前请求没有认证，拦截器就会抛出一个用户没有经过身份认证异常。如果配置文件中配置了当前请求只有VIP用户才能访问，而用户在之前虽然验证了，但是并不是VIP用户，那么就会抛出用户权限不足异常，在异常抛出之后，就会被固定在拦截器前面一环的ExceptionTranslationFilter所捕获，这两个过滤器是固定在过滤器最终的两环之上。

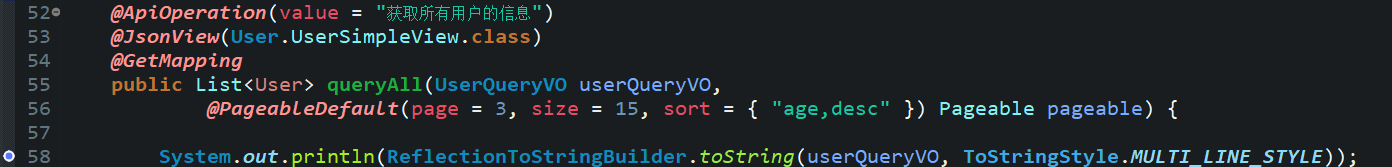
ExceptionTranslationFilter这个过滤器最主要的作用就是用来捕获守门人所抛出的异常，然后根据抛出的异常，作出相应的处理，比如说，如果是因为没有登录抛出的异常，那么就会根据配置文件中的配置，引导用户完成认证，比如，配置文件中配置了formLogin方式认证，那么该过滤器就会引导用户到默认的认证页面，如果配置了httpBasic方式，那么就会让浏览器弹出一个窗口，让用户进行认证。

至此，即为SpringSecurity整个核心业务的基本原理，整个框架提供的业务和特型，都是经过当前过滤器链实现的。在之后的章节中，会讲解如何使用手机验证码，或者微信qq等第三方登录，实际上，都是在这个过滤器链上加入这种绿色的拦截器，来支持不同的身份认证功能逻辑，在实际的业务中，过滤器链上过滤器链上过滤器不止这三种，还会有其他很多种，一般一个普通的应用都会有10几个这个过滤器，在后续的章节中，会讲解这些过滤器，目前，在讲解SpringSecurity基本原理的过程中，只需要知道有这三种过滤器即可。

注意，在拦截器上，这些绿色的拦截器是可以通过配置文件决定使用或者不是用的，但是其他两种过滤器，是肯定会出现在框架中，并且肯定是在最后的两个环节上。位置不可更改，也不能在过滤器链上去掉。



在实际用例中，可以使用断点调试演示一下业务流程，首先，在自定义应用UserController中的queryAll方法中。



然后在橘色的FilterSecurityInterceptor中，第124行，打上断点，当前拦截器为应用最后的守门人，所有的认证逻辑，都是在当前beforeInvocation方法中被调用。



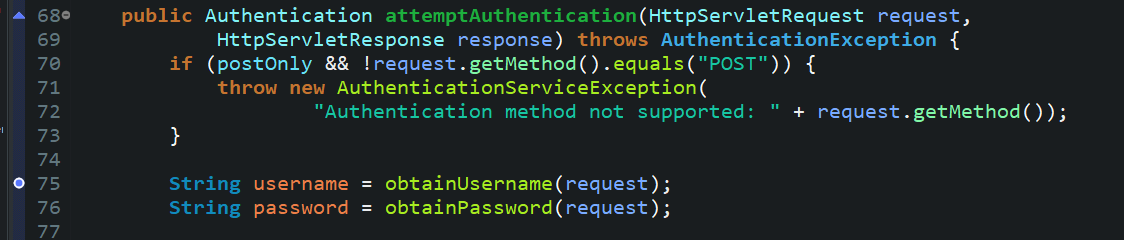
如果判断成功，允许执行的时候，那么

fi.getChain().doFilter(fi.getRequest(), fi.getResponse());

方法会被调用，将当前请求放行。并调用API，在往前，找到ExceptionTranslationFilter.



在123行打上断点，同时，来看一下此过滤器dofilter方法非常简单，如果没有异常，就直接放行，如果发生异常，则在catch中对不同的方法进行不同的处理。在往前，找到UsernamePasswordAuthenticationFilter，即用来做用户名密码认证逻辑的过滤器，可以在源码中看到，当前类只会处理Post方式的/login请求，收到对应的请求之后，会从request中获取username，password参数，然后做认证处理。



在75行打上断点之后，此时，请求经过的几个主要的过滤器都打上了断点，Debug模式启动服务，访问/user/1，理解一下整个流程。

### 3-3.自定义用户认证逻辑

很明显，在上节的配置中，不管是basic还是form都不符合实际业务，一般情况下，登录页面，登录所需字段，以及用户名密码校验逻辑都应该是与业务相符的，数据也应该从数据库中获取，而不是固定的user以及默认的密码，那么，着这种情况下，如何处理用户认证逻辑呢？

* + 1. 处理用户信息获取逻辑

在SpringSecurity中，用于认证的用户信息都被封装在UserDetail接口中，可以在工具中查看该类源码。

public interface UserDetails extends Serializable {

    Collection<? extends GrantedAuthority> getAuthorities();

    String getPassword();

    String getUsername();

    boolean isAccountNonExpired();

    boolean isAccountNonLocked();

    boolean isCredentialsNonExpired();

    boolean isEnabled();

}

发现该封装类一共就7个方法，都是用来表示用户的认证信息的，从上往下，依次为，用户权限列表，密码，用户名，账号本身是否过期，是否锁定，本次登录凭证是否过期，账号本身是否可用。基本上封装了大多数系统所需要的认证信息。如果系统中没有需要验证账号本身是否过期的业务，那么也可以将这些字段永远设置为真，如果为假，那么则认证失败。

那么，默认情况下，SpringSecurity会需要此类封装用户认真信息，接着，编写，用户认证业务处理逻辑。

在config包下接着创建UserDetailServiceConfiguration类用来自定义用户认证逻辑业务处理。

@Component

public class UserDetailServiceConfiguration implements UserDetailsService {

    // @Autowired

    // private Mapper mapper;

    private static final Logger log = LoggerFactory.getLogger(UserDetailServiceConfiguration.class);

    @Override

    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        // TODO Auto-generated methobd stub

        log.info("登录用户名 : " + username);

        String password = 123456

        log.info("password : " + password);

        User user = new User(username, password, true, true, true, true,

                AuthorityUtils.commaSeparatedStringToAuthorityList("admin"));

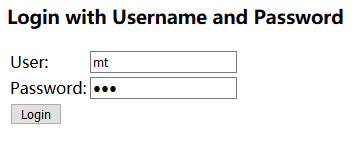
        return user;

    }

}

在这段代码中，实现了UserDetailsService接口，同时将本类注册到spring容器中，spring扫描到本类，会自动调用loadUserByUsername方法去，该方法会返回一个UserDetails对象，spring会验证该对象中的字段是否符合逻辑，即，用户输入的用户名，密码和返回的用户名，密码是否一致，以及对应的状态是否为真。同样的，由于本案例不想过多描述dao层操作，固定使用123作为密码，用用户输入的用户名作为数据库返回用户名。实际案例中，本类可以注入dao层实现，以便从数据库中获取用户信息。同时权限列表，固定为admin字符串，授权，鉴权相关业务，将会在之后的章节中进行详细的处理。

此时，用户获取逻辑及校验逻辑编写完毕，启动服务。访问/user/1，依然跳转到默认的注册页面/login，输入任意用户名，密码123456。



登录成功，自定义用户数据获取逻辑开发完成。



* + 1. 处理密码加密解密

在实际业务中，密码处理在安全系统中算是比较重要的一个部分，密码在数据库中不可能以明文方式存储，通常情况下，采用MD5加盐值方式进行加密，但是这种加密也是有着缺陷的，首先MD5到目前为止，已经不是不可以被破解的了，通过彩虹表匹配的方式，有存在破解的可能性，其次，通常情况下，盐值是固定的，那么久代表着，如果系统中存在有一个用户的密码泄露，被破解了，那么所有用户的密码都存在被破解的可能性，基于这个问题，SpringSecurity本身有提供密码加密解密类，PasswordEncoder，此类在加密的的时候，会在加密的结果中混入随机的盐值，换言之，同一个密码，每次加密的出的结果不同，在解密的时候，推算出对应的盐值，在进行解密。使用该密码加密解密类是一种比较好的处理方式，重构loadUserByUsername方法，使用PasswordEncoder加密。

@Component

public class UserDetailServiceConfiguration implements UserDetailsService {

    // @Autowired

    // private Mapper mapper;

    private static final Logger log = LoggerFactory.getLogger(UserDetailServiceConfiguration.class);

    @Autowired

    private PasswordEncoder encoder;

    @Override

    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        // TODO Auto-generated methobd stub

        log.info("登录用户名 : " + username);

        // 根据用户名查找用户信息

        // login 页面传入的username参数，数据库中查出的密码，以及权限列表

        // 根据查找出来的用户信息判断用户是否被锁定，删除，登录状态过期，密码过期

        /\*\*

         \* String username, String password, boolean enabled,

            boolean accountNonExpired, boolean credentialsNonExpired,

            boolean accountNonLocked, Collection<? extends GrantedAuthority> authorities

            用户名 密码 是否删除 是否登录状态过期 是否密码过期 是否被锁定， 有一个返回false即登录失效，并报出相应标志位异常

            这里不一定要使用spring自带的user类，只需要实现userdetail接口的对象即可，可以使用自定义的，符合项目的自定义user对象，实现该接口逻辑

            $2a$10$tE.bdVy3JXixNSJHshXgE.ifO1JqPYMJBGRC9QUumTOE7wvjdLv.G

            $2a$10$mrehzJi7IqMB88AeQ/UXZOGYe0tdW18ycMK6ilQj8oY2rmSkqaosy

         \*/

        // 此操作应该注册的时候完成，也就是说，存在数据库中的密码信息，应该是加密之后的信息

        String password = encoder.encode("123456");

        log.info("password encode: " + password);

        // 权限列表应该存在数据库中作为数据提取到security配置中 密码和权限列表，都要去读取

        User user = new User(username, password, true, true, true, true,

                AuthorityUtils.commaSeparatedStringToAuthorityList("admin"));

        return user;

    }

}

注意，此类中，加密类为下列包中的，不要导错包了。

import org.springframework.security.crypto.password.PasswordEncoder;

之后，在配置文件中，将此加密类注册到容器中，找到SpringSecurity配置文件，加入

    /\*\*

     \* 使用spring sec 自带的编码解码规则

     \* 也可以使用自定义的编码解码规则，实现passwordencoder类，编码以及解码接口

     \* @return

     \*/

    @Bean

    public PasswordEncoder getPasswordEncoder() {

        return new BCryptPasswordEncoder();

    }

在此方法中，密码还是固定为123456，只不过是使用加密之后的123456，再次启动服务，从控制台中，观察加密之后的密码。

2019-01-02 11:15:23.876 INFO 54896 --- [nio-8080-exec-4] n.s.s.b.c.UserDetailServiceConfiguration : 登录用户名 : mt

2019-01-02 11:15:23.999 INFO 54896 --- [nio-8080-exec-4] n.s.s.b.c.UserDetailServiceConfiguration : password encode: $2a$10$ljYYOLaaP3QqHZic1g2pq.EaloIGk5NxZLsbXMHEj140h.sX0Ka7i

2019-01-02 11:17:48.079 INFO 38252 --- [nio-8080-exec-3] n.s.s.b.c.UserDetailServiceConfiguration : 登录用户名 : mt

2019-01-02 11:17:48.209 INFO 38252 --- [nio-8080-exec-3] n.s.s.b.c.UserDetailServiceConfiguration : password encode: $2a$10$HiLSatAM1UIMiBRxz0sC4eyxvb9KqyEe.jDmYiIX0qqv1nCcrDkPK

这是两次启动服务之后，对123456编码加密之后得到的结果，发现两次对同一个字符串进行加密，所得到的结果不同，但是都能成功登录。SpringSecurity提供的加密类是非常好用的。同时，解密匹配的逻辑也不需要程序员自己处理，只需要返回一个对应的UserDetail类即可，在实际的业务中，用户在生成的时候，就应该使用加密方法，加密用户输入的密码存入数据库中，在取回的时候，就不用再加密解密了。

### 3-4.个性化用户认证流程

在上小节的内容中，已经完成了自定义的用户数据获取（UserDetailsService），自定义的用户信息校验（UserDetails），以及自定义的密码加密逻辑。在此小节中，将介绍如何自定用户的登录界面，即如何配置在未认证的情况下，访问某些资源，如何在用户认证成功之后执行一定的操作，比如，如果是今天第一次登录，那么给用户增加积分，记录登录日志等业务，如何进行用户登录失败处理，比如，超过3次之后在一段时间内不允许登录，同时，记录异常IP，等信息。

* + 1. 自定义登录页面

在SpringSecurity中，请求和静态资源都认为是资源，默认情况下，自定义的请求和资源都是不允许访问的，只有系统默认的/login以及login表单的请求是允许访问应用的，如果需要自定义登录界面，那么需要在配置文件中，接触对登录页面以及登录请求的限制，首先，编写security配置类。

对登录策略重新配置

    @Override

    protected void configure(HttpSecurity http) throws Exception {

        // TODO Auto-generated method stub

        http.formLogin()// http请求使用表单登录方式验证

                .loginPage("/demo-signIn.html")//设置登录之后跳转的URL

                .loginProcessingUrl("/authentication/form")//设置登录form

                .and()

                .authorizeRequests()// 表示接下来都是对请求的授权配置

                .antMatchers("/demo-signIn.html","/authentication/form")//设置请求地址

                .permitAll()//设置匹配的请求，允许访问

                .anyRequest()// 任何请求

                .authenticated()// 都需要身份认证

                .and()

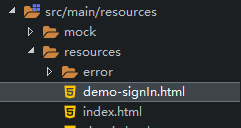
                .csrf()//暂时先关闭跨域请求拦截策略

                .disable();

    }

代码的解释都写在注释中，在这段配置中，配置了未认证情况下允许访问登录页面以及登录请求，以及登录策略从默认页面转到自定义的demo-signIn.html页面。并将登录请求放行。同时，在配置的最后，暂时的关闭了CSRF即跨域请求拦截策略，如果不关闭，那么在访问的时候会抛出403异常，在之后的章节中会讲解这个问题，在这里暂时先关闭。

编写登录页面，同样，登录页面也非常基础，在demo工程中创建



<body>

    <p>demo-signIn.html</p>

    <form action="/authentication/form" method="post">

        <input type="text" name="username" placeholder="username" /> <br />

        <br> <input type="text" name="password" placeholder="password" />

        <input type="submit" value="submit"> <br><!-- <input

            type="text" name="imageCode"> --> <!-- <img src="/code/image"> -->

    </form>

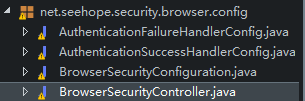
</body>

启动服务，在此访问/user/1,会发现，跳转到了自定义的登录页面，输入任意登录名，123456密码，成功访问到应用服务。自定义认证逻辑编写完毕。

但是，在此逻辑中还有着一些问题，重构并优化一下配置代码。解决几个问题，首先，本节的目的应该是开发一个可重用的，跨平台的安全模块，那么登录请求地址就不能写死在代码中，最好以配置文件的形式开放，同时提供默认值，其次，既然是跨平台的，那么肯定不能统一让用户在html登录，应该先判断用户访问终端，如果是终端访问，那么应该返回Json字符串，如果是浏览器访问，那么应该返回登录页面。

很明显，配置文件中无法判断请求来源，那么，将默认的登录界面，交给控制器处理，由控制器判断是应该跳转还是返回Json字符串。

在配置文件中创建控制器。



@RestController

public class BrowserSecurityController {

    private static final Logger log = LoggerFactory.getLogger(BrowserSecurityController.class);

    private RequestCache requestCache = new HttpSessionRequestCache();

    private RedirectStrategy redirectStrategy = new DefaultRedirectStrategy();

    @Autowired

    private SecurityProperties securityProperties;

    /\*\*

     \*

     \* 访问除了不需要认证的资源，都会进入当前方法

     \*

     \* 当前方法的目的是，根据入口的不同 （因为是通用后台，所以可以被移动端访问（返回json），或者web端访问（返回html））

     \* 从而分发不同的认证逻辑

     \* 当需要身份认证时，跳转到当前方法

     \* @param request

     \* @param response

     \* @return

     \* @throws IOException

     \*/

    @RequestMapping("/authentication/require")

    @ResponseStatus(code = HttpStatus.UNAUTHORIZED)

    public JSONResult requireAuthentication(HttpServletRequest request, HttpServletResponse response)

            throws IOException {

        // 去到引发当前跳转的请求

        SavedRequest savedRequest = requestCache.getRequest(request, response);

        if (savedRequest != null) {

            String target = savedRequest.getRedirectUrl();

            log.info("引发跳转的请求是: " + target);

            if (StringUtils.endsWith(target, ".html")) {

                redirectStrategy.sendRedirect(request, response, securityProperties.getBrowser().getLoginPage());

            }

        }

        return JSONResult.errorAuthorized("访问的服务需要身份认证，请引导用户到登录页面");

    }

}

按照上述小节的描写，此类中应该分辨用户端访问方式，但是，本例中处理有些不同，首先，获取用户的输入，判断请求地址，如果请求以.html结尾，那么直接跳转到登录页面，如果不是，那么，直接返回401状态码，以及Json形式的描述，让移动端自行引导到终端的认真页面。换言之，本类相当于一个拦截操作，只要用户没有登录，那么就引导用户跳转到登录操作，其中JSONResult类是一个工具类，封装了Json形式返回的常见方法，以及结果字段。

package net.seehope.core.util;

/\*\*

\* @Description: 自定义响应数据结构

\*              这个类是提供给门户，ios，安卓，微信商城用的

\*              门户接受此类数据后需要使用本类的方法转换成对于的数据类型格式（类，或者list）

\*              其他自行处理

\*              200：表示成功

\*              500：表示错误，错误信息在msg字段中

\*              501：bean验证错误，不管多少个错误都以map形式返回

\*              502：拦截器拦截到用户token出错

\*              555：异常抛出信息

\*              401：请求未授权

\*/

public class JSONResult {

    // 响应业务状态

    private Integer status;

    // 响应消息

    private String msg;

    // 响应中的数据

    private Object data;

    private String ok; // 不使用

    public static JSONResult build(Integer status, String msg, Object data) {

        return new JSONResult(status, msg, data);

    }

    public static JSONResult ok(Object data) {

        return new JSONResult(data);

    }

    public static JSONResult ok() {

        return new JSONResult(null);

    }

    public static JSONResult errorMsg(String msg) {

        return new JSONResult(500, msg, null);

    }

    public static JSONResult errorMap(Object data) {

        return new JSONResult(501, "error", data);

    }

    public static JSONResult errorTokenMsg(String msg) {

        return new JSONResult(502, msg, null);

    }

    public static JSONResult errorException(String msg) {

        return new JSONResult(555, msg, null);

    }

    public static JSONResult errorAuthorized(String msg) {

        return new JSONResult(401, msg, null);

    }

    public JSONResult() {

    }

    public JSONResult(Integer status, String msg, Object data) {

        this.status = status;

        this.msg = msg;

        this.data = data;

    }

    public JSONResult(Object data) {

        this.status = 200;

        this.msg = "OK";

        this.data = data;

    }

    public Boolean isOK() {

        return this.status == 200;

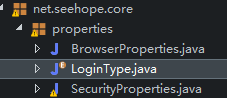
    }

    //get/set

}

同时，上节也说过，本模块应该做成通用模块，即可以在不同的系统中复用，那么，就应该把登录页面开放，让使用的系统自行配置登录页面。

在core工程中定义，当然，在browser模块中定义也可以（即封装在一个module内）



public class BrowserProperties {

    private String loginPage = "/demo-signIn.html";

    private LoginType loginType = LoginType.REDIREACT;

}

定义两个字段，登录页面，以及登录类型，同时，提供默认值，登录类型为枚举类，包含两个值。

public enum LoginType {

    REDIREACT, JSON

}

同时，定义SecurityProperties类封装所有和安全相关的配置

@ConfigurationProperties(prefix = "seehope.security")

public class SecurityProperties {

    private BrowserProperties browser = new BrowserProperties();

    //get/set

}

在此类中，使用Springboot从配置application配置中，寻找相应字段的配置，之后，在使用此安全模块的工程Demo中，配置好

seehope.security.browser.loginPage=/demo-signIn.html

seehope.security.browser.loginType=JSON

重构browser模块安全框架配置。

@Autowired

private SecurityProperties securityProperties;

http.formLogin()// http请求使用表单登录方式验证

                .loginPage("/authentication/require")//设置登录之后跳转的URL

                .loginProcessingUrl("/authentication/form")//设置登录form

                .and()

                .authorizeRequests()// 表示接下来都是对请求的授权配置

                .antMatchers(securityProperties.getBrowser().getLoginPage(),

                        "/authentication/form",

                        "/authentication/require")//设置请求地址

                .permitAll()//设置匹配的请求，允许访问

                .anyRequest()// 任何请求

                .authenticated()// 都需要身份认证

                .and()

                .csrf()//暂时关闭跨域请求拦截策略

                .disable();

将登录请求转发到控制器，同时，使用变量来配置登录页面，值得注意的是，登录请求也就是form表单的请求地址这时候还一定要是/authentication/form，同理，也可以开放此字段作为配置，让使用安全模块的人来配置。

至此，可重用的安全模块开发完成，启动服务，分别重浏览器和RESTED访问服务，也可以试试在Demo工程中自定义登录页面。

访问请求，得到结果，至此，可重用的安全配置编码完成

2019-01-02 15:57:18.822 INFO 21300 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring FrameworkServlet 'dispatcherServlet'

2019-01-02 15:57:18.822 INFO 21300 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization started

2019-01-02 15:57:18.857 INFO 21300 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization completed in 35 ms

2019-01-02 15:57:19.132 INFO 21300 --- [nio-8080-exec-2] n.s.s.b.c.BrowserSecurityController : 引发跳转的请求是: http://localhost:8080/index.html

2019-01-02 15:57:23.004 INFO 21300 --- [nio-8080-exec-4] n.s.s.b.c.UserDetailServiceConfiguration : 登录用户名 : mt

2019-01-02 15:57:23.137 INFO 21300 --- [nio-8080-exec-4] n.s.s.b.c.UserDetailServiceConfiguration : password encode: $2a$10$YtfgkJBkiJVnJZk76U0n3uZgGeiTIew3rg8Iki7KQNltFJ8SoYpTC

* + 1. 登陆成功/失败处理

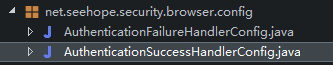
在上述案例中，对于浏览器端发起的请求已经处理的比较合适了，但是针对于移动终端，当前的处理业务还不够完善，那么，可以添加新的配置，完善认证业务。

对于上小结的代码，在配置文件中描述了下述逻辑。

用户在访问应用的时候，除了登录页面和登录请求，其他所有的资源都需要授权，换言之，所有的请求都会重定向到控制器，在控制器中判断请求是否以.html结尾，即，是否为静态资源文件，如果是，则重定向到登录页面，如果不是，抛出异常，同时返回状态码401，用户未授权。

自定义逻辑大体上处理完成。但是，对于认证成功之后，以及认证失败之后的处理有些不足，这时候，对于认证配置，添加新的逻辑处理，即登录成功处理和登录失败处理。

在配置包中创建新的处理类，成功处理，失败处理



/\*\*

\* 认证成功处理器，默认逻辑是跳转到之前的请求，比如用户输入 /user没有权限访问之后，跳转到/user请求，如果配置了登陆成功配置，可以自定义认证成功之后的跳转，

\* 比如说返回一个json形式的字符串，用在跨平台处理上面

\* @author : MT

\* @date 创建时间：2018年12月10日 下午2:26:41

\* @version 1.0

\*/

@Component

public class AuthenticationSuccessHandlerConfig extends SavedRequestAwareAuthenticationSuccessHandler {

    @Autowired

    private ObjectMapper objectMapper;

    @Autowired

    private SecurityProperties securityProperties;

    @Override

    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response,

            Authentication authentication) throws IOException, ServletException {

        // TODO Auto-generated method stub

        if (LoginType.JSON.equals(securityProperties.getBrowser().getLoginType())) {

            response.setContentType("application/json;charset=utf-8");

            response.getWriter().write(objectMapper.writeValueAsString(authentication));

        } else {

            super.onAuthenticationSuccess(request, response, authentication);

        }

    }

}

在成功处理中，如果登录控制器返回响应状态码为200，则进入成功处理器，处理器中判断，在配置文件中，设置好的响应式JSON形式还是重定向形式，如果是JSON，那么直接输出用户的权限列表Authentication对象，在本案例中作为示例。

@Component

public class AuthenticationFailureHandlerConfig extends SimpleUrlAuthenticationFailureHandler {

    @Autowired

    private ObjectMapper objectMapper;

    @Autowired

    private SecurityProperties securityProperties;

    @Override

    public void onAuthenticationFailure(HttpServletRequest request, HttpServletResponse response,

            AuthenticationException exception) throws IOException, ServletException {

        // TODO Auto-generated method stub

        if (LoginType.JSON.equals(securityProperties.getBrowser().getLoginType())) {

            response.setStatus(HttpStatus.INTERNAL\_SERVER\_ERROR.value());

            response.setContentType("application/json;charset=utf-8");

            response.getWriter().write(objectMapper.writeValueAsString(exception));

        } else {

            super.onAuthenticationFailure(request, response, exception);

        }

    }

}

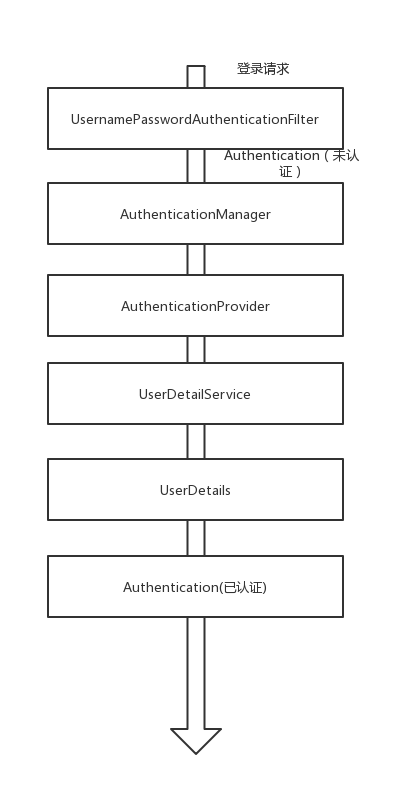
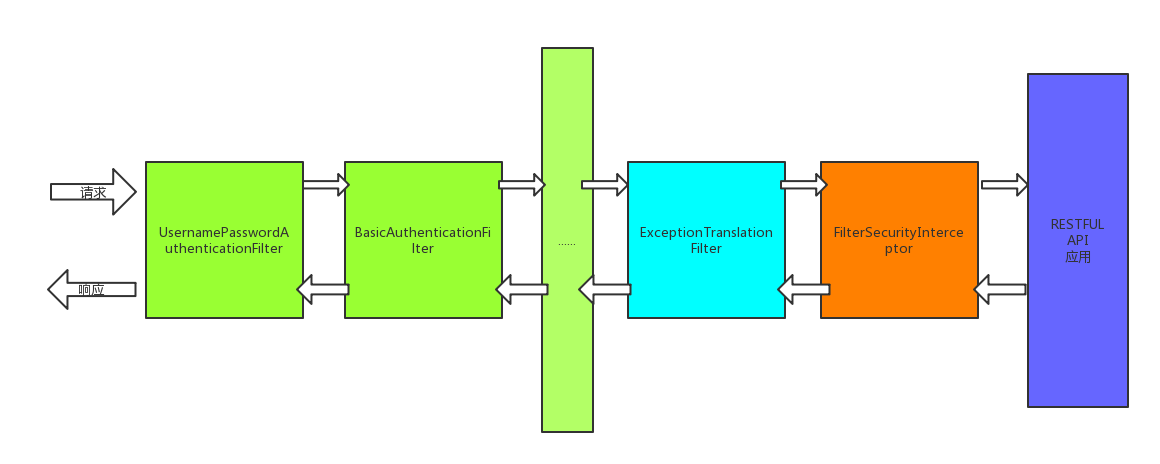
失败处理也是同理，如果控制器抛出异常，那么进入处理逻辑，如果是配置JSON形式返回，则直接输出异常信息，如果配置为重定向，那么保留控制器返回结果。在本案例中，很多业务逻辑是作为案例来说明应用效果的，在实际业务中，请根据不同的需要来使用组件。

### 3-5.认证流程源码详解

在之前小节的内容中，已经介绍了如何自定义登录页面，以及登录成功，失败处理，以及如何自定义用户信息获取逻辑（UserDetailService），处理用户校验逻辑（UserDetails），处理密码加密解密（PasswordEncoder）等配置。那么，在整个security流程中，是如何将所有组件组合起来使用呢？在本小结内容中，将会对解释一下security的源码流程。

3.5.1 认证处理流程说明

此时，在此回顾一下以下流程图，实际上，用户认证处理流程，就是请求从进入应用到返回相应的流程。



请求在访问到具体的应用之前，会经过这些核心类。同样的，可以使用Debug模式断点调试来一步步查看服务运行的具体流程，首先，将断点准备好。

UsernamePasswordAuthenticationFilter第70行，即在图1中，springsecurity过滤器链上的一环，负责处理用户名密码认证逻辑。

String username = obtainUsername(request);

String password = obtainPassword(request);

UsernamePasswordAuthenticationToken authRequest = new UsernamePasswordAuthenticationToken(username, password);

在这里解释一下该类的核心业务，首先从请求中获取到了用户名以及密码，值得注意的是，如果使用用户名密码过滤器，默认情况下，前端传入参数的字段名必须为username以及password，获取到这两个字段之后，构建了一个UsernamePasswordAuthenticationToken对象，这个对象，就是在认证成功处理器中提到过的Authentication对象的一种实现，封装了用户的认证信息。看一下此时创建的认证对象。发现

public UsernamePasswordAuthenticationToken(Object principal, Object credentials) {

    super(null);

    this.principal = principal;

    this.credentials = credentials;

    setAuthenticated(false);

}

很明显，在当前流程中，只是把用户名和密码赋值给成员变量，其中，调用了super将所有权限列表设置为空，同时，认证状态也是未认证。这是认证信息构建的第一步。在返回到过滤器，下一步执行

// Allow subclasses to set the "details" property

setDetails(request, authRequest);

也就是说，将上一部构建好的认证详情信息存放到请求中，以便在后续的过滤器中继续加工。

return this.getAuthenticationManager().authenticate(authRequest);

然后，调用了getAuthenticationManager的authenticate方法，这里就出现了，图二中的第二个类，这个类本身并不提供任何认证逻辑，他的作用是用来管理第三个类getAuthenticationProvider的，进入ProviderManager类查看当前方法。这是AuthenticationManager的实现类。找到第163行。

for (AuthenticationProvider provider : getProviders()) {

    if (!provider.supports(toTest)) {

        continue;

    }

发现这里有个for循环，很明显，这里使用了适配器模式，寻找支持当前处理的provider，调用相应的处理方法。对于不同的方式的认证，比如用户名密码，或者微信等第三方的登录，使用对应的provider处理认证逻辑。比方说在使用第三方登录的时候，可以查看SocialAuthenticationFilter类中的doAuthentication方法在调用认证方法的时候，传入的参数就是SocialAuthenticationToken对象，那么返回到providermanager中，就会使用对应的provider去处理相应的认证业务。最终执行。

result = provider.authenticate(authentication);

真正的去调用用户认证的逻辑。此时，可以观察一下此认证逻辑，找到AbstractUserDetailsAuthenticationProvider类，主要的认证都封装在此抽象类中的authenticate方法中。

UserDetails user = this.userCache.getUserFromCache(username);

try {

    user = retrieveUser(username,

            (UsernamePasswordAuthenticationToken) authentication);

}

在此方法中调用了retrieveUser方法返回了一个UserDetails对象也就是用户认证信息封装对象。retrieveUser方法是一个抽象方法，其实现类在DaoAuthenticationProvider类中。

UserDetails loadedUser;

try {

    loadedUser = this.getUserDetailsService().loadUserByUsername(username);

}

调用了用户认证逻辑处理方法，当然，这里实际上调用的，是用户自定义的UserDetailService的实现类。此方法的返回结果，在实际业务中，就是从数据库中取出来的一个包含用户认证所需要的所有信息的对象。在返回到抽象类中，执行了retrieveUser方法之后，返回的UserDetail，被preAuthenticationChecks.check(user)方法调用，在这此方法中，对User对象进行验证。在验证方法中

if (!user.isAccountNonLocked()) {

    logger.debug("User account is locked");

    throw new LockedException(messages.getMessage(

            "AbstractUserDetailsAuthenticationProvider.locked",

            "User account is locked"));

}

if (!user.isEnabled()) {

    logger.debug("User account is disabled");

    throw new DisabledException(messages.getMessage(

            "AbstractUserDetailsAuthenticationProvider.disabled",

            "User is disabled"));

}

if (!user.isAccountNonExpired()) {

    logger.debug("User account is expired");

    throw new AccountExpiredException(messages.getMessage(

            "AbstractUserDetailsAuthenticationProvider.expired",

            "User account has expired"));

}

对自定义返回的UserDetail对象中的3个字段，进行了验证，进行完预检查之后，又执行了附加检查：

additionalAuthenticationChecks(user,(UsernamePasswordAuthenticationToken) authentication);

该方法在DaoAuthenticationProvider类中有具体的实现，在这个附加检查内，实际上就是在使用用户提供的PasswordEncoder来验证数据库中返回的密码是否正确。预检查检查用户状态是否通过，附加检查检查密码是否正确，如果都通过之后，调用后检查

postAuthenticationChecks.check(user);

后检查就是检查预检查中最后一个没有检查的字段，即当前登录票据是否过期。当所有的验证通过，那么就认为用户认证成功，加工之前request中存放的数据字段。

return createSuccessAuthentication(principalToReturn, authentication, user);

在这个方法中，实际上也是重新创建了一下认证请求，只不过不是调用之前那个只有用户名密码的构造器，而是创建了一个新的包含有认证成功信息的对象。在回顾一下图2，认证逻辑全部在AuthenticationProvider中，调用了UserDetailsService中的load方法，获得了UserDetail对象，经过认证之后，最后返回了已认证的Authentication对象。返回到AbstractAuthenticationProcessingFilter中的doFilter逻辑，根据认真信息中记录的不同，分别可能被调用

unsuccessfulAuthentication(request, response, failed);

successfulAuthentication(request, response, chain, authResult);

方法。在这两个方法中，会调用自定义的登录成功或者登陆失败处理，如果认证流程中未发生异常，那么调用成功处理，如果在图2中任何一个类中发生异常，那么会调用失败处理。整个图二的认证处理器流程全部处理完成。

流程比较复杂，在实际的业务中，可能会添加不同的认证处理类用来处理不同的认证方式。

3.5.2 认证结果如何在多个请求之间共享

很明显，如果要在多个请求之间共享数据，肯定是通过session标准，在AbstractAuthenticationProcessingFilter类的successfulAuthentication方法中：

SecurityContextHolder.getContext().setAuthentication(authResult);

这部分代码中可以看到，整个认证信息和两个对象有关系，SecurityContextHolder以及SecurityContext对象，认证之后，框架会把认证之后的结果存放在SecurityContext中，并将SecurityContext存放在SecurityContextHolder中。SecurityContextImpl非常简单，只是封装了一下authentication即认证信息对象，之后覆盖了equals，hashcode等方法保证其唯一性。那么，SecurityContextHolder实际上是threadlocal的一种封装，负责将SecurityContext封装到当前线程中相当于线程级别的成员变量，一般情况下，从请求到响应的过程中，是一条线程完成工作的，那么，在任何一个地方存储了该对象，在下一个流程中就能取出来用。而这个类，最终是交由SecurityContextPersistenceFilter类使用，这个类也是个过滤器，它固定出现在SpringSecurity过滤器链的最前端，所以，用户请求进来，首先访问该类，响应返回，最后一个过他，在用户请求进来的时候，检查session中是否有SecurityContext，如果有，将SecurityContext取出，放入当前线程，如果没有，则放行，当请求处理完毕响应返回时，检查线程，线程中有没有认证信息，如果有认证信息放入session。这样的话，不同的请求，在同一个线程中就能拿到同一个认证信息，因为整个请求和响应都是在一个线程中完成的，所有在一次请求相应经过的所有类中，都可以操作这个认证信息。

3.5.3 获取认证用户信息

最后，试着编写一条服务来获取用户的认证信息，在整个框架中，随时可以在控制器中获取用户认证信息，而且可以使用多种方式，修改用户控制器UserController，添加方法

@GetMapping("/currentUser")

public Object getCurrentUser(@AuthenticationPrincipal UserDetails userDetails) {

    return userDetails;

}

@GetMapping("/user")

public Object getUser() {

    return SecurityContextHolder.getContext().getAuthentication();

}

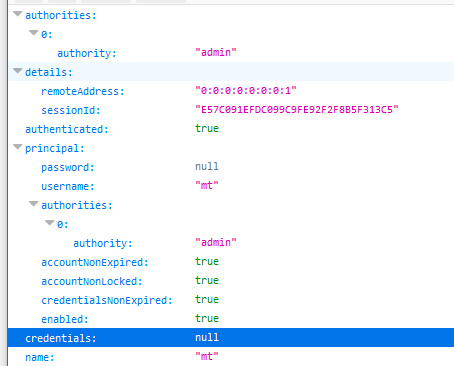
@GetMapping("/me")

public Object getCurrentUserDetail(Authentication authentication) {

    return authentication;

}

启动服务，访问相应API，得到结果

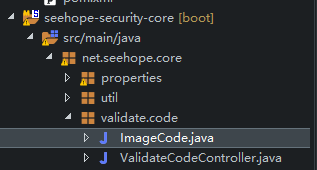


### 3-6.图片验证码

3.6.1 开发生成图形验证码接口

实际上就是开发一个能够根据随机数生成图片的API，同时将随机数存储到Session中，然后将生成的图片写到接口的响应中。

因为生成图形验证码这个需求，不管是移动端还是浏览器都是有可能用到的，所以生成二维码的逻辑放在core模块中。



import java.awt.image.BufferedImage;

import java.time.LocalDateTime;

public class ImageCode {

    private String code;

    private BufferedImage image;

    private LocalDateTime expireTime;

    /\*\*

     \*

     \* @param code

     \* @param image

     \* @param expireTime 过期的时间点

     \*/

    public ImageCode(String code, BufferedImage image, LocalDateTime expireTime) {

        this.code = code;

        this.image = image;

        this.expireTime = expireTime;

    }

    /\*\*

     \*

     \* @param code

     \* @param image

     \* @param expireIn 多少秒之后过期 单位是秒

     \*/

    public ImageCode(String code, BufferedImage image, int expireIn) {

        this.code = code;

        this.image = image;

        this.expireTime = LocalDateTime.now().plusSeconds(expireIn);

}

    public boolean isExpried() {

        return LocalDateTime.now().isAfter(expireTime);

    }

//get/set

}

该类有封装有三个字段，验证码，验证码图片，以及验证码过期时间，之后，编写验证码生成控制器。

@RestController

public class ValidateCodeController {

    public static final String SESSION\_KEY = "SESSION\_KEY\_IMAGE\_CODE";

    private SessionStrategy sessionStrategy = new HttpSessionSessionStrategy();

    /\*\*

     \* 图片流直接在响应中写出去，所以该方法不需要返回值

     \* @throws IOException

     \*/

    @GetMapping("/code/image")

    public void createValidateCode(HttpServletRequest request, HttpServletResponse response) throws IOException {

        // 生成图片二维码

        ImageCode imageCode = createImageCode(request);

        // 讲二维码放到session中

        sessionStrategy.setAttribute(new ServletWebRequest(request), SESSION\_KEY, imageCode);

        // 响应二维码到前端

        ImageIO.write(imageCode.getImage(), "JPEG", response.getOutputStream());

    }

    private ImageCode createImageCode(HttpServletRequest request) {

        // TODO Auto-generated method stub

        int width = 67;

        int height = 23;

        BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE\_INT\_RGB);

        Graphics g = image.getGraphics();

        Random random = new Random();

        g.setColor(getRandColor(200, 250));

        g.fillRect(0, 0, width, height);

        g.setFont(new Font("Times New Roman", Font.ITALIC, 20));

        g.setColor(getRandColor(160, 200));

        for (int i = 0; i < 155; i++) {

            int x = random.nextInt(width);

            int y = random.nextInt(height);

            int xl = random.nextInt(12);

            int yl = random.nextInt(12);

            g.drawLine(x, y, x + xl, y + yl);

        }

        StringBuilder sRand = new StringBuilder();

        for (int i = 0; i < 4; i++) {

            String rand = String.valueOf(random.nextInt(10));

            sRand.append(rand);

            g.setColor(new Color(20 + random.nextInt(110), 20 + random.nextInt(110), 20 + random.nextInt(110)));

            g.drawString(rand, 13 \* i + 6, 16);

        }

        g.dispose();

        return new ImageCode(sRand.toString(), image, 60);

    }

    /\*\*

    \* 生成随机背景条纹

    \*

    \* @param fc 前景色

    \* @param bc 背景色

    \* @return RGB颜色

    \*/

    private Color getRandColor(int fc, int bc) {

        Random random = new Random();

        if (fc > 255) {

            fc = 255;

        }

        if (bc > 255) {

            bc = 255;

        }

        int r = fc + random.nextInt(bc - fc);

        int g = fc + random.nextInt(bc - fc);

        int b = fc + random.nextInt(bc - fc);

        return new Color(r, g, b);

    }

}

本案例中并不关注图形如何生成，生成图形的方法，可以自行搜索。按照之前安全框架的配置，本响应应该放置于登录页面，即用户未登录状态下允许访问，在安全配置中授权。

同时，在登录页面加上验证码的引用。

<input type="text" name="imageCode">

<img src="/code/image">

启动服务，访问到登录页面，得到结果



3.6.2 在认证流程中加入图形验证码校验

在之前小节的编码中，都是使用或者实现SpringSecurity默认提供的类，但是在框架中，并没有哪个类是用于验证验证码是否正确的，基于框架的基本流程，即过滤器链模式，那么，用户可以自定义过滤器，并将过滤器加入在usernamepassword验证器之前，在请求进入的时候，进行验证码认证，如果通过，让请求继续访问，如果不通过，则抛出异常。

先编写验证码拦截器，添加验证码拦截规则

public class ValidateCodeFilter extends OncePerRequestFilter {

    private static final Logger log = LoggerFactory.getLogger(ValidateCodeFilter.class);

    private AuthenticationFailureHandler authenticationFailureHandler;

private SessionStrategy sessionStrategy = new HttpSessionSessionStrategy();

    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)

            throws ServletException, IOException {

        // TODO Auto-generated method stub

        // 如果当前请求是登录请求，那么进行验证码认证，如果不是登录请求，那么直接放行

        if (StringUtils.equals("/authentication/form", request.getRequestURI())

                && StringUtils.equalsIgnoreCase(request.getMethod(), "post")) {

            try {

                validate(new ServletWebRequest(request));

            } catch (ValidateCodeException e) {

                // TODO: handle exception

                authenticationFailureHandler.onAuthenticationFailure(request, response, e);

return;

            }

        }

        filterChain.doFilter(request, response);

    }

    private void validate(ServletWebRequest servletWebRequest) throws ServletRequestBindingException {

        // 在验证码生成API中，生成的验证码出现在了

        ImageCode codeInSession = (ImageCode) sessionStrategy.getAttribute(servletWebRequest,

                ValidateCodeController.SESSION\_KEY);

        // 从请求中将imageCode字段取出 即前端传入的参数名为imagecode

        String codeInRequest = ServletRequestUtils.getStringParameter(servletWebRequest.getRequest(), "imageCode");

        log.info(codeInRequest);

        if (StringUtils.isBlank(codeInRequest)) {

            throw new ValidateCodeException("验证码的值不能为空");

        }

        if (codeInSession == null) {

            throw new ValidateCodeException("验证码不存在");

        }

        if (codeInSession.isExpried()) {

            // 销毁验证码

            sessionStrategy.removeAttribute(servletWebRequest, ValidateCodeController.SESSION\_KEY);

            throw new ValidateCodeException("验证码已过期");

        }

        if (!StringUtils.equals(codeInSession.getCode(), codeInRequest)) {

            throw new ValidateCodeException("验证码不匹配");

        }

        // 每次登录请求提交，不管成功没成功都刷新验证码

        sessionStrategy.removeAttribute(servletWebRequest, ValidateCodeController.SESSION\_KEY);

}

}

当前拦截器需要继承OncePerRequestFilter类，此类确保在过滤器链中只会被调用一次处理，当登录页面一被访问的时候，img标签就会像控制器发送请求获取验证码图片，控制器除了返回验证码图片之外，还将验证码对象存放在了Session中，那么过滤器的验证逻辑就很简单了，只需要从Session中将控制器存放的验证码取出，并和前端传回来的验证码对比，如果相同，则通过即可。如果验证码异常，那么各自抛出。其中，ValidateCodeException类为AuthenticationException类的子类，后者表示所有验证异常的父类。

public class ValidateCodeException extends AuthenticationException {

    private static final long serialVersionUID = 1L;

    public ValidateCodeException(String msg) {

        super(msg);

        // TODO Auto-generated constructor stub

    }

}

验证逻辑编写完成，将自定义的过滤器添加到SpringSecurity过滤器链中。同时，给过滤器中的失败处理器赋值。

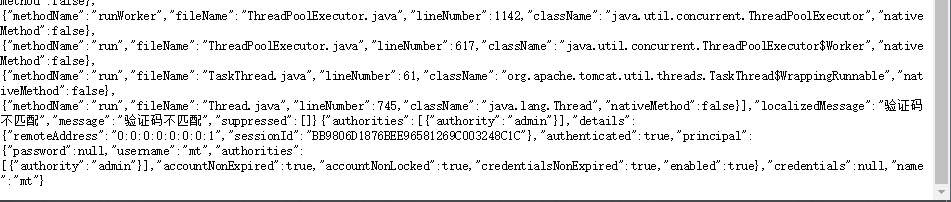
ValidateCodeFilter validateCodeFilter=new ValidateCodeFilter();

validateCodeFilter.setAuthenticationFailureHandler(authenticationFailureHandlerConfig);

http.addFilterBefore(validateCodeFilter, UsernamePasswordAuthenticationFilter.class)

        .formLogin()// http请求使用表单登录方式验证

此时，验证码功能编写完成，启动服务，测试代码。成功逻辑不用多说，当验证码异常，浏览器输出



在返回异常的时候，将所有的堆栈信息全都输出，实际业务中，前端并不需要堆栈信息，只需要将异常的提示信息返回即可，修改异常处理器。

response.getWriter().write(objectMapper.writeValueAsString(exception.getMessage()));

这时候，验证码异常即为



不在返回所有堆栈异常

### 3-7.图片验证码重构

在上小节的内容中，已经介绍了如何编写图形验证码功能，但是，光实现功能是非常简单的事，而本案例的目的，在于开发一个可重用的安全接口，那么很明显，在实际业务中，验证码的应用场景远比这复杂，比如验证码位数不同，生成的验证码方式不同等。在这种情况下，应该将验证码生成逻辑写成可配置，可重用的。

3.7.1 验证码基本参数可配置

在core模块中创建验证码配置封装类，此类中除了图形验证码配置，在之后的章节中也会存放短信验证码等配置，所以这样封装。

public class ValidateCodeProperties {

    ImageCodeProperties imageCode = new ImageCodeProperties();

}

以及图形验证码配置属性封装类

public class ImageCodeProperties {

    private int width = 67;

    private int height = 23;

    private int expireIn = 60;

    private int length = 4;

    /\*\*

     \* 存放所有需要验证嘛验证的请求，如有多个，逗号隔开

     \*/

    private String urls;

}

封装了验证码宽度，高度，长度以及过期时间，urls中存放所有需要调用验证码的请求，多个请求之间以逗号隔开。

然后，将ValidateCodeProperties放入SecurityProperties类中

@ConfigurationProperties(prefix = "seehope.security")

public class SecurityProperties {

    private BrowserProperties browser = new BrowserProperties();

    private ValidateCodeProperties validateCode = new ValidateCodeProperties();

}

修改生成验证码代码，用配置项取代固定参数。

    private SecurityProperties properties;

    /\* (non-Javadoc)

     \* @see net.seehope.core.validate.code.ValidateCodeGenerator#createImageCode(javax.servlet.http.HttpServletRequest)

     \*/

    @Override

    public ImageCode createImageCode(HttpServletRequest request) {

        // TODO Auto-generated method stub

        int width = ServletRequestUtils.getIntParameter(request, "width",

                properties.getValidateCode().getImageCode().getWidth());

        int height = ServletRequestUtils.getIntParameter(request, "height",

                properties.getValidateCode().getImageCode().getHeight());

        for (int i = 0; i < properties.getValidateCode().getImageCode().getLength(); i++) {

        return new ImageCode(sRand.toString(), image, properties.getValidateCode().getImageCode().getExpireIn());

    }

这里只贴出了需要修改的代码，在代码下面记得提供SecurityProperties类的get/set方法，方便在配置文件中使用过滤器的时候注入配置参数。

在本案例中，提供有3层配置逻辑，首先，在core包的properties文件中，定义了默认值，其次，读取使用该模块的工程的配置文件，获取参数，最终，读取请求中的参数。之后，修改配置逻辑。

最后，在security.core包下新建配置文件，将securityproperties注入到springboot容器中。

@Configuration

@EnableConfigurationProperties(SecurityProperties.class)

public class SecurityCoreConfig {

}

当springboot扫描到以上配置文件时，会将属性配置文件注入到spring容器中，以便在整个工程中使用。

ValidateCodeFilter validateCodeFilter = new ValidateCodeFilter();

        validateCodeFilter.setAuthenticationFailureHandler(authenticationFailureHandlerConfig);

        validateCodeFilter.setProperties(securityProperties);

然后，在demo模块中添加对验证码参数的配置

seehope.security.validateCode.imageCode.width=200

seehope.security.validateCode.imageCode.length=6

启动服务，观察验证码的生成是否遵循当前三层配置。

3.7.2 验证码拦截接口可配置

之所以要将验证码拦截借口可配置化，是因为在实际业务中，不光是登录请求会需要验证码，可能会有其他请求需要验证码，那么此时，具体使用的工程可以按需配置需要验证码的请求。在封装中已经添加了urls字段，那么接着修改过滤器，修改放行逻辑。

public class ValidateCodeFilter extends OncePerRequestFilter implements InitializingBean {

private AntPathMatcher machers = new AntPathMatcher();

    private Set<String> urls = new HashSet<>();

    /\* (non-Javadoc)

     \* @see org.springframework.web.filter.GenericFilterBean#afterPropertiesSet()

     \*/

    @Override

    public void afterPropertiesSet() throws ServletException {

        // TODO Auto-generated method stub

        super.afterPropertiesSet();

        String[] urlsConfig = StringUtils

                .splitByWholeSeparatorPreserveAllTokens(properties.getValidateCode().getImageCode().getUrls(), ",");

        for (String string : urlsConfig) {

            urls.add(string);

        }

        urls.add("/authentication/form");

    }

    @Override

    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)

            throws ServletException, IOException {

        // TODO Auto-generated method stub

        // 如果当前请求是登录请求，那么进行验证码认证，如果不是登录请求，那么直接放行

        boolean action = false;

        for (String string : urls) {

            if (machers.match(string, request.getRequestURI())) {

                action = true;

            }

}

}

}

首先，让过滤器实现InitializingBean接口，该接口提供一个afterPropertiesSet方法，这个方法并没有什么固定的流程，而是一个标准，在配置类中创建完过滤器对象时，手动调用下该方法，用作初始化，将配置文件中的urls字段取出，并存放到成员变量中，在过滤逻辑中，重写验证逻辑，如果用户传入的请求与配置文件符合，则需要验证。

在此修改配置文件

ValidateCodeFilter validateCodeFilter = new ValidateCodeFilter();

validateCodeFilter.setAuthenticationFailureHandler(authenticationFailureHandlerConfig);

validateCodeFilter.setProperties(securityProperties);

validateCodeFilter.afterPropertiesSet();

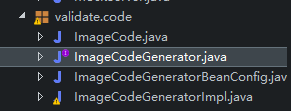
在demo工程中配置

seehope.security.validateCode.imageCode.urls=/user/\*

那么此时，除了登录逻辑，所有的/user开头的逻辑都会被验证。启动服务，验证逻辑。

3.7.3 验证码生成逻辑可配置

在实际业务中，不一定会使用这个基础的验证码生成逻辑，那么，就需要将验证码生成逻辑开放出接口，让使用验证码模块的用户去实现功能。



创建ImageCodeGenerator接口

public interface ImageCodeGenerator {

    /\*\*

     \* 创建验证码接口

     \* @param request

     \* @return

     \*/

    ImageCode createImageCode(HttpServletRequest request);

}

封装了创建验证码接口，构建默认实现类

public class ImageCodeGeneratorImpl implements ImageCodeGenerator {

    private SecurityProperties properties;

    /\* (non-Javadoc)

     \* @see net.seehope.core.validate.code.ValidateCodeGenerator#createImageCode(javax.servlet.http.HttpServletRequest)

     \*/

    @Override

    public ImageCode createImageCode(HttpServletRequest request) {

        // TODO Auto-generated method stub

        int width = ServletRequestUtils.getIntParameter(request, "width",

                properties.getValidateCode().getImageCode().getWidth());

        int height = ServletRequestUtils.getIntParameter(request, "height",

                properties.getValidateCode().getImageCode().getHeight());

        BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE\_INT\_RGB);

        Graphics g = image.getGraphics();

        Random random = new Random();

        g.setColor(getRandColor(200, 250));

        g.fillRect(0, 0, width, height);

        g.setFont(new Font("Times New Roman", Font.ITALIC, 20));

        g.setColor(getRandColor(160, 200));

        for (int i = 0; i < 155; i++) {

            int x = random.nextInt(width);

            int y = random.nextInt(height);

            int xl = random.nextInt(12);

            int yl = random.nextInt(12);

            g.drawLine(x, y, x + xl, y + yl);

        }

        StringBuilder sRand = new StringBuilder();

        for (int i = 0; i < properties.getValidateCode().getImageCode().getLength(); i++) {

            String rand = String.valueOf(random.nextInt(10));

            sRand.append(rand);

            g.setColor(new Color(20 + random.nextInt(110), 20 + random.nextInt(110), 20 + random.nextInt(110)));

            g.drawString(rand, 13 \* i + 6, 16);

        }

        g.dispose();

        return new ImageCode(sRand.toString(), image, properties.getValidateCode().getImageCode().getExpireIn());

    }

    /\*\*

    \* 生成随机背景条纹

    \*

    \* @param fc 前景色

    \* @param bc 背景色

    \* @return RGB颜色

    \*/

    public Color getRandColor(int fc, int bc) {

        Random random = new Random();

        if (fc > 255) {

            fc = 255;

        }

        if (bc > 255) {

            bc = 255;

        }

        int r = fc + random.nextInt(bc - fc);

        int g = fc + random.nextInt(bc - fc);

        int b = fc + random.nextInt(bc - fc);

        return new Color(r, g, b);

    }

    /\*\*

     \* @return the properties

     \*/

    public SecurityProperties getProperties() {

        return properties;

    }

    /\*\*

     \* @param properties the properties to set

     \*/

    public void setProperties(SecurityProperties properties) {

        this.properties = properties;

    }

}

实际上就是把控制器中的生成逻辑封装在了实现类中，同时，引入安全配置，并提供get/set方便在配置中使用。

添加配置文件

@Configuration

public class ImageCodeGeneratorBeanConfig {

    @Autowired

    private SecurityProperties properties;

    @Bean

    @ConditionalOnMissingBean(name = "imageCodeGeneratorImpl")

    public ImageCodeGenerator validateCodeGenerator() {

        ImageCodeGeneratorImpl generator = new ImageCodeGeneratorImpl();

        generator.setProperties(properties);

        return generator;

    }

}

在配置文件中，有ConditionalOnMissingBean注解，意味着，如果启动的时候，spring检测到有名字叫imageCodeGeneratorImpl的bean修改控制器，那么就不执行配置文件，也就是不注入默认的验证码创建逻辑，而是使用自定义的imageCodeGeneratorImpl。

当然，在当前业务中，如果硬性规定子类实现名字一定也并不是非常的合乎情理，也可以指定某类的实现，即直接在注解中填写父类.class

@Autowired

private ImageCodeGenerator validateCodeGenerator;

/\*\*

\* 图片流直接在响应中写出去，所以该方法不需要返回值

\* @throws IOException

\*/

@GetMapping("/code/image")

public void createValidateCode(HttpServletRequest request, HttpServletResponse response) throws IOException {

// 生成图片二维码

ImageCode imageCode = validateCodeGenerator.createImageCode(request);

// 讲二维码放到session中

sessionStrategy.setAttribute(new ServletWebRequest(request), SESSION\_KEY, imageCode);

// 响应二维码到前端

ImageIO.write(imageCode.getImage(), "JPEG", response.getOutputStream());

}

在demo模块中，自定义验证码生成逻辑，并将名字与上述配置对应，得到代码

package net.seehope.demo.util;

@Component

public class ImageCodeGeneratorImpl implements ImageCodeGenerator {

    /\* (non-Javadoc)

     \* @see net.seehope.core.validate.code.ValidateCodeGenerator#createImageCode(javax.servlet.http.HttpServletRequest)

     \*/

    @Override

    public ImageCode createImageCode(HttpServletRequest request) {

        // TODO Auto-generated method stub

        System.out.println("在使用的模块中自定义的验证码生成逻辑");

        return null;

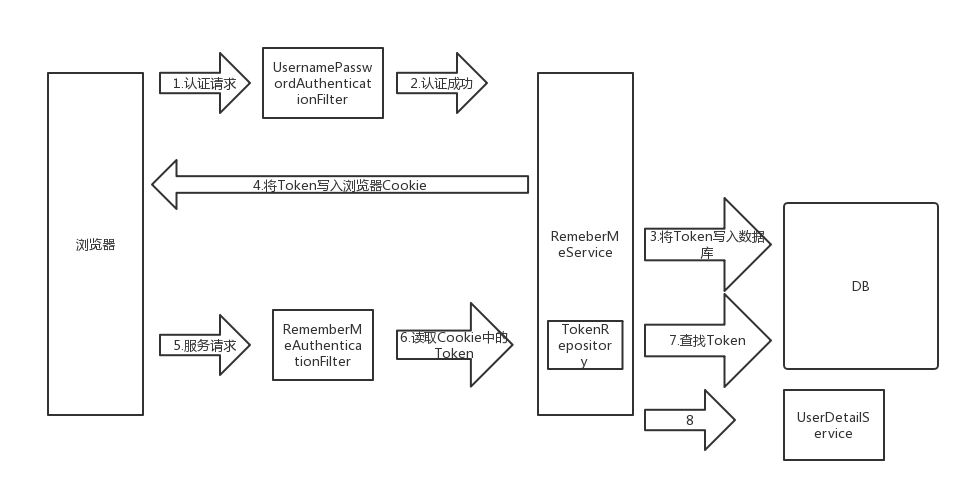
    }

}

启动服务，默认情况下，验证码不会生成，而是在控制台上输出并且抛出异常，此时，成功将验证码生成接口开放。

### 3-8.添加记住我功能

3.8.1 记住我功能基本原理



在SpringSecurity中，如果成功配置了记住我功能，那么在用户认证成功之后，会进入RemeberMeService中，交由ToeknRepository将Token存放进数据库中。这是第一次认证流程，当用户再次访问的时候，在过滤器链中即本章最开始的图中，框架会添加RememberMeAuthenticationFilter过滤器存在过绿色滤器链的最后一环 即异常捕获过滤器之前，当前面所有的过滤器无法认证用户身份的时候，来读取Cookie中的Token，之后再交由TokenRepository去DB中寻找对应的用户认证信息。找到之后，将读取出来的信息，交付给UserDetailService，之后的流程就和普通登录流程相同了。

3.8.2 记住我功能基本实现

首先，修改登录页面。添加复选框remember-me，这里注意一下，记住我功能复选框的name如果不做修改，一定设置为此字符。

记住我:<input type="checkbox" name="remember-me" checked="checked">

同时，在配置记住我功能之前，先把上小结自定义的图形验证码关闭，再修改自定义属性BrowserProperties。添加字段rememberMeSeconds，描述记住我功能保留时间。

修改主配置文件：

@Autowired

private DataSource dataSource;

@Autowired

private UserDetailsService userDetailsService;

@Bean

public PersistentTokenRepository persistentToeknRepository() {

    JdbcTokenRepositoryImpl tokenRepository = new JdbcTokenRepositoryImpl();

    tokenRepository.setDataSource(dataSource);

    tokenRepository.setCreateTableOnStartup(true);

    return tokenRepository;

}

http.addFilterBefore(validateCodeFilter, UsernamePasswordAuthenticationFilter.class).formLogin()// http请求使用表单登录方式验证

                .loginPage("/authentication/require")// 设置登录之后跳转的URL

                .loginProcessingUrl("/authentication/form")// 设置登录form

                .successHandler(authenticationSuccessHandlerConfig)

                .failureHandler(authenticationFailureHandlerConfig)

                .and()

                .rememberMe()

                .tokenRepository(persistentToeknRepository())

                .tokenValiditySeconds(securityProperties.getBrowser().getRememberMeSeconds())

                .userDetailsService(userDetailsService)

                .and()

                .authorizeRequests()// 表示接下来都是对请求的授权配置

                .antMatchers(securityProperties.getBrowser().getLoginPage(), "/authentication/form",

                        "/authentication/require", "/code/image")// 设置请求地址

                .permitAll()// 设置匹配的请求，允许访问

                .anyRequest()// 任何请求

                .authenticated()// 都需要身份认证

                .and().csrf()// 暂时关闭跨域请求拦截策略

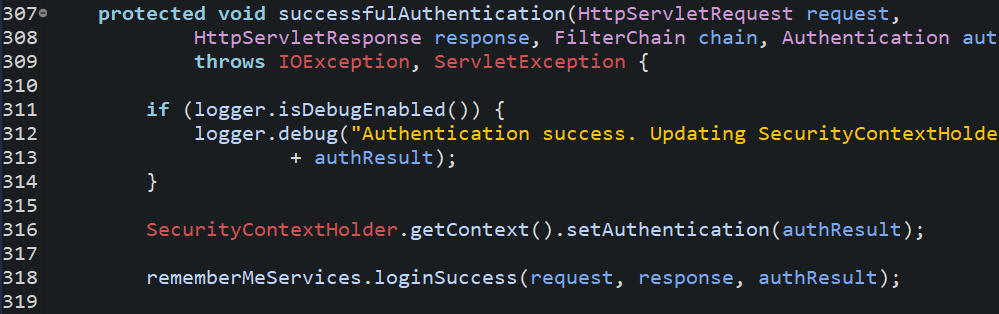
                .disable();

}

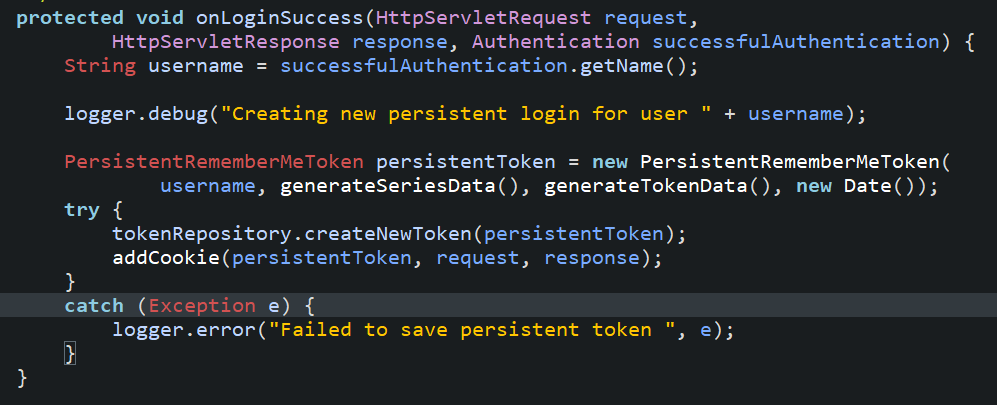
需要配置tokenRepository类，同时将数据源交付，进入该类源码，Token表脚本已经封装在类中，可以在方法使用setCreateTableOnStartup方法在启动的时候自动执行创建表的脚本，也可以自己在源码中将脚本取出执行，如果使用该方法，记得第二次启动的时候将该方法注释掉。在主配置中，添加相应配置。记住我功能配置完成，虽然在第一节图中的流程较长，但是实际应用起来，不过是几行配置而已，在SpringSecurity中已经有实现。启动服务，检测功能。

3.8.3 记住我功能SpringSecurity源码解析

回顾上小节的流程图内容认证成功后，会由AbstractAuthenticationProcessingFilter调用RememberMeService。

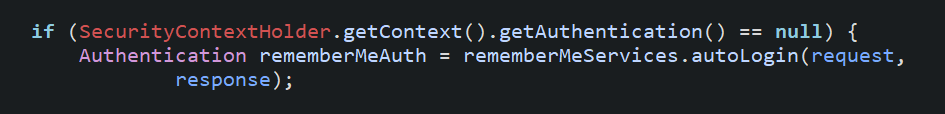


接着进入loginSuccess方法，实现类PersistentTokenBasedRememberMeServices中

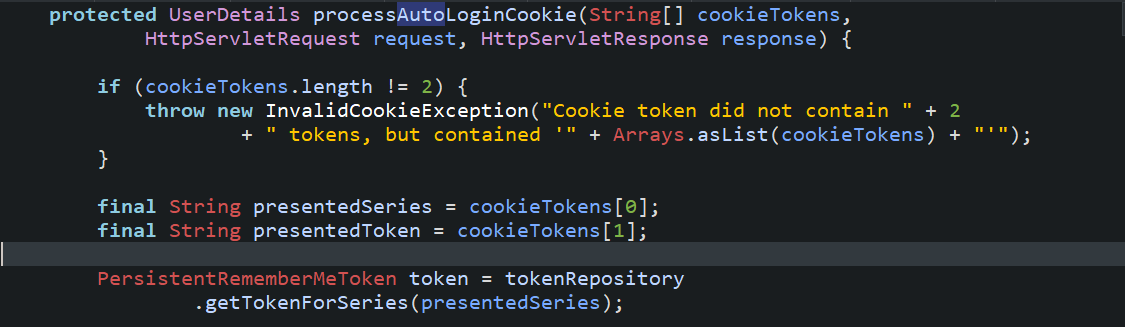


调用了了persistentToken，持久化到数据库，同时，添加cookie返回给浏览器。至此，第一次登录流程完成。

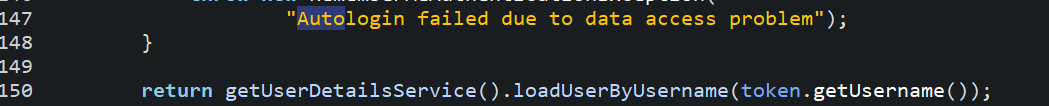
用户再次访问的时候，按照上小节的说明，绿色的过滤器前面无法认证，在最后的RememberMeAuthenticationFilter中



将Token取出，并交由数据库，取出对应用户名，完成认证流程。



然后，调用UserDetails



最终，返回，讲用户信息认证成功信息放回Session中。

### 3-9.短信验证码接口开发

3.9.1 开发短信验证码接口

先回顾一下短信验证流程，首先，在登录页面中，增加两个新的字段，用户手机以及验证码，在用户填写了手机并验证通过之后，使能按钮，点击按钮发送请求到后台，后台生成验证码放入session同时调用第三方接口，向用户填写的手机发送短信告知验证码内容。

    <fieldset>

    <legend>短信验证码登录</legend>

    <form action="/authentication/form" method="post">

        <input type="text" name="username" placeholder="username" /> <br />

        <br> <input type="text" name="password" placeholder="password" />

        <br>

        <!-- 记住我:<input type="checkbox" name="remember-me"

        checked="checked"> -->

        <br> <input type="text" name="validateCode"><br> <input type="text" name="mobile" placeholder="手机号" value="13012345678">

        <input type="button" value="发送验证码" id="smsCode"></input> <br>

        <input type="submit" value="submit">

    </form>

</fieldset>

</body>

<script src="http://libs.baidu.com/jquery/2.0.0/jquery.min.js"></script>

<script type="text/javascript">

$(function() {

    $("#smsCode").click(function() {

        $.ajax({

            type: "get",

            url: "/code/sms?mobile=13012345678",

            async: true,

            success: function(data) {

                console.log(data);

            },

            error: function() {

                console.log("system error");

            }

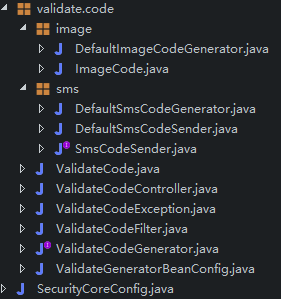
        });

    });

});

</script>

首先，将图片验证码重构一下。

提取出共有类ValidateCode，包含两个字段Code和expireTime用来表示普通验证码（短信），ImageCode继承ValidateCode，并添加独有字段image表示图片。修改验证码生成接口中方法名为createCode，返回类型为ValidateCode，修改实现类名为DefaultImageCodeGenerator，修改配置文件

@ConditionalOnMissingBean(ignored = ValidateCodeGenerator.class)

修改添加默认图形验证码实现类的判断逻辑为，检测spring容器中是否有validateCodeGenerator类的实现，如果有，则不添加默认实现。修改过滤器逻辑。

ValidateCode codeInSession = (ValidateCode) sessionStrategy.getAttribute(servletWebRequest,

                ValidateCodeController.SESSION\_KEY);

        // 从请求中将validateCode字段取出 即前端传入的参数名为validateCode

        String codeInRequest = ServletRequestUtils.getStringParameter(servletWebRequest.getRequest(), "validateCode");

注意，这里前端传入所有验证码的字段名要叫做validateCode，不管是图片验证码还是短信验证码。

修改控制器，验证码生成逻辑，添加新的接口以及成员变量

    @Autowired

    private ValidateCodeGenerator smsValidateCodeGenerator;

    @Autowired

    private SmsCodeSender smsCodeSender;

    @GetMapping("/code/sms")

    public void createSmsCode(HttpServletRequest request, HttpServletResponse response)

            throws IOException, ServletRequestBindingException {

        // 生成短信验证码

        ValidateCode smsValidateCode = smsValidateCodeGenerator.createCode(request);

        // 将短信验证码放到session中

        sessionStrategy.setAttribute(new ServletWebRequest(request), SESSION\_KEY, smsValidateCode);

        // 调用短信服务商提供的短信发送接口

        String mobile = ServletRequestUtils.getRequiredStringParameter(request, "mobile");

        smsCodeSender.send(mobile, smsValidateCode.getCode());

    }

当前接口提供给前端Ajax请求使用，当用户点击发送验证码，该方法被调用，接受前端验证过后的mobile字段，getRequiredStringParameter方法如果参数值为null则抛出异常，收到生成验证码请求之后，和图形验证码逻辑相同，创建验证码，并调用第三方接口发送验证码。

创建DefaultSmsCodeGenerator类，表示默认短信验证码生成逻辑

@Component("smsCodeGenerator")

public class DefaultSmsCodeGenerator implements ValidateCodeGenerator {

    private SecurityProperties securityProperties;

    /\* (non-Javadoc)

     \* @see net.seehope.core.validate.code.ValidateCodeGenerator#createImageCode(javax.servlet.http.HttpServletRequest)

     \*/

    @Override

    public ValidateCode createCode(HttpServletRequest request) {

        // TODO Auto-generated method stub

        String code = RandomStringUtils

                .randomNumeric(securityProperties.getValidateCode().getSmsValidateCodeProperties().getLength());

        return new ValidateCode(code,

                securityProperties.getValidateCode().getSmsValidateCodeProperties().getExpireIn());

    }

}

非常简单，直接返回配置文件中位数的随机数即可，因为短信验证码的生成逻辑可以说是唯一的，但是发送短信验证码的第三方可是不可预料的，所以发送短信的接口是开放的。创建短信发送接口SmsCodeSender：

public interface SmsCodeSender {

    /\*\*

     \* 发送短信验证码

     \* @param mobile 向那个手机号发送

     \* @param code 发送的验证码是什么

     \*/

    void send(String mobile, String code);

}

创建短信发送默认实现：DefaultSmsCodeSender

public class DefaultSmsCodeSender implements SmsCodeSender {

    @Override

    public void send(String mobile, String code) {

        // TODO Auto-generated method stub

        System.out.println("向手机发送短信验证码:" + mobile + "验证码:" + code);

    }

}

在本案例中并没有使用实际的短信服务商的借口去发送短信，这里只是直接在控制台上打印验证码而已。

修改配置文件，将验证码生成请求，放入不检测请求列表中。

.antMatchers(securityProperties.getBrowser().getLoginPage(), "/authentication/form",

"/authentication/require", "/code/image","/code/sms")// 设置请求地址

同时，添加短信验证码配置类

public class SmsValidateCodeProperties {

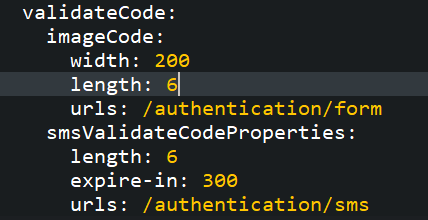
    private int expireIn = 300;

    private int length = 6;

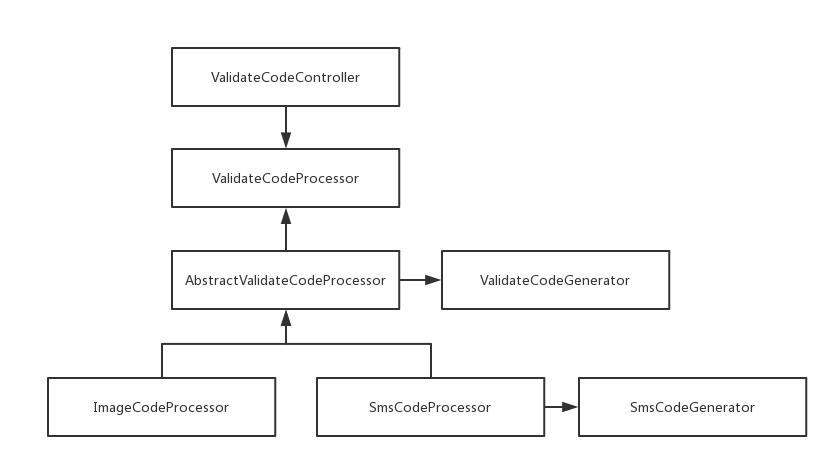
    private String urls;

}

和短信验证码字段类似。启动之前，在使用该模块的配置文件中添加短信验证码配置



同样的，这里没有添加短信服务商的API，在实际应用中，验证码发送类SmsCodeSender中的方法要实现。启动服务，验证接口。  
3.9.2 重构代码



按照上图的说明，仿照SpringSecurity的编写方式，在验证码生成控制器中，图片验证码，和短信验证码的生成逻辑存在有部分重复代码，可以抽取出抽象类，将通用方法放在抽象类中实现，将独有代码，放在具体实现类ImageCodeProcessor,SmsCodeProcessor中实现。

首先，跟随请求接受的流程，先重构controller

@RestController

public class ValidateCodeController {

  @Autowired

  private Map<String, ValidateCodeProcessor> validateCodeProcessors;

  @GetMapping("/code/{type}")

  public void createCode(HttpServletRequest request, HttpServletResponse response, @PathVariable String type)

      throws Exception {

    validateCodeProcessors.get(type + "CodeProcessor").create(new ServletWebRequest(request, response));

  }

}

短信验证生成和图片生成接口合并为有一个接口，同时，在成员变量中注入hashmap validateCodeProcessors对象，依据spring的机制，如果成员变量中有这种泛型的hashmap，那么会在spring容器中查找所有的ValidateCodeProcessor类的实现，以beanname作为key，以bean作为value，存放到map中，在控制器内，可以对key的name做判断，取出对应的处理逻辑。

依据上小结的图，所有的验证码处理逻辑封装在validateCodeProcessor接口中，该接口只有一个方法，即提供验证码。

public interface ValidateCodeProcessor {

  /\*\*

   \* 验证码放入Session时的前缀

   \*/

  String SESSION\_KEY\_PREFIX = "SESSION\_KEY\_FOR\_CODE\_";

  /\*\*

   \* 创建验证码

   \* @param request

   \* @throws Exception

   \*/

  void create(ServletWebRequest request) throws Exception;

}

同时，该类有一个抽象的实现AbstractValidateCodeProcessor，用来封装验证码生成逻辑。

/\*\*

\* 抽象出所有验证码的生成逻辑

\* 分为三步，即，生成验证码，存储验证码，发送验证码

\* 生成逻辑交由不同的实现类实现，发送验证码也同理

\* 如果验证码生成逻辑不会发生更改，只是需要更改验证码生成方式，那么此类不需要修改，只需要修改生成类的实现

\* 只有在三步不满足业务逻辑，不在停留在生成，存储，发送流程，此类才会被修改

\*/

public abstract class AbstractValidateCodeProcessor<C extends ValidateCode> implements ValidateCodeProcessor {

  private static final Logger log = LoggerFactory.getLogger(AbstractValidateCodeProcessor.class);

  /\*\*

   \* spring 框架再扫描到该类中有此注入类，那么会将所有ValidateCodeGenerator的实现注入到该表中

   \* 以BeanName作为Key

   \* 收集系统中所有的 {@link ValidateCodeGenerator} 接口的实现。

   \*/

  @Autowired

  private Map<String, ValidateCodeGenerator> validateCodeGenerators;

  /\*\*

   \* 操作session的工具类

   \*/

  private SessionStrategy sessionStrategy = new HttpSessionSessionStrategy();

  /\*

   \* (non-Javadoc)

   \*

   \* @see

   \* com.imooc.security.core.validate.code.ValidateCodeProcessor#create(org.

   \* springframework.web.context.request.ServletWebRequest)

   \*/

  @Override

  public void create(ServletWebRequest request) throws Exception {

    C validateCode = generate(request);

    save(request, validateCode);

    send(request, validateCode);

  }

  /\*\*

   \* 生成校验码

   \* 此处，暂时对验证码生成方法做了进一步限定，接受前台传入的参数判断/code/之后的字符串

   \* 通过字符串匹配，调用相应的验证码生成器

   \* 如果传入的是sms则调用defaultSmsValidateCodeGenerator Bean

   \* 如果传入的是image则调用defaultImageVlidateCodeGenerator Bean

   \* 换言之，前端验证码生成请求限定为 /code/sms 或者 /code/image 后期会在优化当前逻辑

   \* @param request

   \* @return

   \*/

  @SuppressWarnings("unchecked")

  private C generate(ServletWebRequest request) {

    String validateGeneratorName = "default" + captureName(getProcesserType(request)) + "CodeGenerator";

    ValidateCodeGenerator validateCodeGenerator = validateCodeGenerators.get(validateGeneratorName);

    log.info(validateCodeGenerator.getClass().getSimpleName());

    return (C) validateCodeGenerator.createCode(request.getRequest());

  }

  /\*\*

   \* 保存验证码，将生成的验证码保存在session中。

   \* 名字一定为 SESSION\_KEY\_FOR\_CODE\_SMS

   \* 或者 SESSION\_KEY\_FOR\_CODE\_IMAGE

   \* 也是初步逻辑。后期会再次重构

   \* @param request

   \* @param validateCode

   \*/

  private void save(ServletWebRequest request, C validateCode) {

    String sessionKey = SESSION\_KEY\_PREFIX + getProcesserType(request).toUpperCase();

    log.info(sessionKey);

    sessionStrategy.setAttribute(request, sessionKey, validateCode);

  }

  /\*\*

   \* 发送校验码，由子类实现

   \*

   \* @param request

   \* @param validateCode

   \* @throws Exception

   \*/

  protected abstract void send(ServletWebRequest request, C validateCode) throws Exception;

  private String getProcesserType(ServletWebRequest request) {

    return StringUtils.substringAfter(request.getRequest().getRequestURI(), "/code/");

  }

  // 首字母大写

  public static String captureName(String name) {

    char[] cs = name.toCharArray();

    cs[0] -= 32;

    return String.valueOf(cs);

  }

}

对于该类的说明，大多写在注释内了，该抽象类有两个实际的实现。ImageCodeProcessor用来生成图片验证码，SmsCodeProcessor用来生成短信验证码。

@Component

public class SmsCodeProcessor extends AbstractValidateCodeProcessor<ValidateCode> {

  private static final Logger log = LoggerFactory.getLogger(SmsCodeProcessor.class);

  @Autowired

  private SmsCodeSender smsCodeSender;

  @Override

  protected void send(ServletWebRequest request, ValidateCode smsValidateCode) throws Exception {

    // TODO Auto-generated method stub

    // 调用短信服务商提供的短信发送接口

    log.info("SmsCodeProcessor");

    String mobile = ServletRequestUtils.getRequiredStringParameter(request.getRequest(), "mobile");

    log.info(mobile);

    smsCodeSender.send(mobile, smsValidateCode.getCode());

  }

}

ImageCodeProcessor

@Component

public class ImageCodeProcessor extends AbstractValidateCodeProcessor<ImageCode> {

  /\*\*

   \* 发送图形验证码方法

   \*/

  @Override

  protected void send(ServletWebRequest request, ImageCode imageCode) throws Exception {

    // TODO Auto-generated method stub

    ImageIO.write(imageCode.getImage(), "JPEG", request.getResponse().getOutputStream());

  }

}

修改配置文件，让验证码请求允许在没有授权的情况下访问

.antMatchers(securityProperties.getBrowser().getLoginPage(), "/authentication/form",

            "/authentication/require", "/code/\*")// 设置请求地址

整个验证码生成流程重构完成，该流程中，具体实现的代码和之前两个小节的内容并没有什么太大的区别，但是从架构上来分析，模仿springSecurity的架构，将验证码生成的逻辑共有的部分抽取出抽象的实现，如果验证码的生成逻辑不发生变化（即，创建验证码，存储验证码，发送验证码），那么抽象类就不需要更改，同时，开放除了验证码生成实现，以及短信发送的实现。

从业务实现的角度上来看，这种重构无非是多此一举，但是从架构的角度上来看，对于代码来说，整个架构的可重用性产生了非常大的变化。

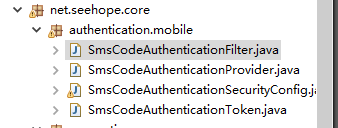
### 3-10.短信登录开发

在上述用例中，重构了验证码生成逻辑，将共有的生成逻辑抽取生成抽象类，而验证码的发送逻辑（图形验证码响应给前端，短信验证码调用第三方接口发送短信到客户手机）开放为可配置的接口给使用安全模块的程序猿使用，如果使用者有自己的实现，则不调用框架默认的实现，框架本身包含默认的实现，同时将部分字段设置成可配置项交由用户配置。在此用例中，还未生成短信验证码认证过滤器，在之后的过程中会重构验证码认证逻辑。

在验证码生成逻辑重构之后，接着编写短信登录逻辑，和用户名密码登录逻辑类似，回顾一下用户名密码登录逻辑，在过滤器链中的UsernamePasswordAuthenticationFilter负责检查请求，如果是登录请求，检查有无用户名以及密码，如果有，那么将用户名和密码包装成未认证状态的UsernamePasswordAuthenticationToken，并调用AuthenticationManager使用适配器模式，调用对应provider认证用户名密码登录逻辑业务。

那么，如果用户要自定义短信登录业务，很明显可以仿照此逻辑，像springboot容器中添加短信登录流程。SmsCodeAuthenticationFilter，短信认证票据实体类SmsCodeAuthenticationToken，短信认证适配器SmsAuthenticationProvider，去完成短信登录业务。

首先，构建短信认证过滤器，用来包装票据以及判断是否短信登录请求过滤器。



写法很简单,仿照系统中的UsernamePasswordAuthenticationFilter，将代码完整复制过来，之后，将所有跟密码参数相关的代码去掉，将用户信息直接改为手机号，而不是用户名，同时，在存放token的时候，将原来的usernameToken替换为自定义的SmsCodeToken。

public class SmsCodeAuthenticationFilter extends AbstractAuthenticationProcessingFilter {

    public static final String SEEHOPE\_SECURITY\_FORM\_MOBILE\_KEY = "mobile";

    private String mobileParameter = SEEHOPE\_SECURITY\_FORM\_MOBILE\_KEY;

    /\*\*

     \* 当前过滤器是否只处理post请求

     \*/

    private boolean postOnly = true;

    /\*\*

     \* 设置当前过滤器要拦截什么请求

     \* 验证手机表单提交请求

     \*/

    public SmsCodeAuthenticationFilter() {

        super(new AntPathRequestMatcher("/authentication/mobile", "POST"));

    }

    /\*\*

     \* 认证流程

     \*/

    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)

            throws AuthenticationException {

        /\*\*

         \* 判断当前请求是否为POST

         \*/

        if (postOnly && !request.getMethod().equals("POST")) {

            throw new AuthenticationServiceException("Authentication method not supported: " + request.getMethod());

        }

        // 获取手机号

        String mobile = obtainMobile(request);

        if (mobile == null) {

            mobile = "";

        }

        mobile = mobile.trim();

        SmsCodeAuthenticationToken authRequest = new SmsCodeAuthenticationToken(mobile);

        // Allow subclasses to set the "details" property

        setDetails(request, authRequest);

        return this.getAuthenticationManager().authenticate(authRequest);

    }

    protected String obtainMobile(HttpServletRequest request) {

        return request.getParameter(mobileParameter);

    }

    protected void setDetails(HttpServletRequest request, SmsCodeAuthenticationToken authRequest) {

        authRequest.setDetails(authenticationDetailsSource.buildDetails(request));

    }

    public void setUsernameParameter(String usernameParameter) {

        Assert.hasText(usernameParameter, "Username parameter must not be empty or null");

        this.mobileParameter = usernameParameter;

    }

    public void setPostOnly(boolean postOnly) {

        this.postOnly = postOnly;

    }

    public final String getUsernameParameter() {

        return mobileParameter;

    }

}

短信票据实体类封装SmsCodeAuthenticationToken，此类的编写逻辑和上述Filter编写逻辑相同，复制UsernamePasswordAuthenticationToken，并将其中和密码字段相关的代码删除。

public class SmsCodeAuthenticationToken extends AbstractAuthenticationToken {

    private static final long serialVersionUID = SpringSecurityCoreVersion.SERIAL\_VERSION\_UID;

    /\*\*

     \* 用户认证信息，认证之前存放前端发送的手机号，成功之后放认证成功的用户信息

     \*/

    private final Object principal;

    /\*\*

     \*

     \* @param mobile 手机号

     \*/

    public SmsCodeAuthenticationToken(String mobile) {

        super(null);

        this.principal = mobile;

        setAuthenticated(false);

    }

    public SmsCodeAuthenticationToken(Object principal, Collection<? extends GrantedAuthority> authorities) {

        super(authorities);

        this.principal = principal;

        super.setAuthenticated(true); // must use super, as we override

    }

    public Object getPrincipal() {

        return this.principal;

    }

    public void setAuthenticated(boolean isAuthenticated) throws IllegalArgumentException {

        if (isAuthenticated) {

            throw new IllegalArgumentException(

                    "Cannot set this token to trusted - use constructor which takes a GrantedAuthority list instead");

        }

        super.setAuthenticated(false);

    }

    @Override

    public void eraseCredentials() {

        super.eraseCredentials();

    }

    @Override

    public Object getCredentials() {

        // TODO Auto-generated method stub

        return null;

    }

}

短信认证Token编写完成。

提供短信登录业务适配器，交由AuthenticationManager控制

public class SmsCodeAuthenticationProvider implements AuthenticationProvider {

    private UserDetailsService userDetailService;

    /\*

     \* manager通过适配器模式，获取到当前类 （AuthenticationProvider 子类）

     \* 调用该方法验证是否支持sms处理

     \* (non-Javadoc)

     \* @see org.springframework.security.authentication.AuthenticationProvider#supports(java.lang.Class)

     \*/

    @Override

    public boolean supports(Class<?> authentication) {

        // TODO Auto-generated method stub

        return SmsCodeAuthenticationToken.class.isAssignableFrom(authentication);

    }

    @Override

    public Authentication authenticate(Authentication authentication) throws AuthenticationException {

        // TODO Auto-generated method stub

        SmsCodeAuthenticationToken authenticationToken = (SmsCodeAuthenticationToken) authentication;

        /\*\*

         \* 调用读取用户信息逻辑，这里注意，如果有多种登录方式，那么在获取信息的逻辑中应该判断当前是以什么逻辑登录的，或者在自己本身模块中配置登录认证逻辑

         \*/

        UserDetails user = userDetailService.loadUserByUsername((String) authenticationToken.getPrincipal());

        if (user == null) {

            throw new InternalAuthenticationServiceException("无法获取用户信息");

        }

        SmsCodeAuthenticationToken authenticationResult = new SmsCodeAuthenticationToken(user, user.getAuthorities());

        authenticationResult.setDetails(authenticationToken.getDetails());

        return authenticationResult;

    }

    public UserDetailsService getUserDetailService() {

        return userDetailService;

    }

    public void setUserDetailService(UserDetailsService userDetailService) {

        this.userDetailService = userDetailService;

    }

}

在认证处理器中调用认证方法，重新包装token，将userdetail放入token中，交由下一个过滤器处理。至此，3个自定义组件全部配置完成，最后，将配置好的组件添加到主配置中，创建SmsCodeAuthenticationSecurityConfig类，添加自定义配置文件，模仿主配置写法。

public class SmsCodeAuthenticationSecurityConfig

        extends SecurityConfigurerAdapter<DefaultSecurityFilterChain, HttpSecurity> {

    @Autowired

    private AuthenticationSuccessHandler authenticationSuccessHandler;

    @Autowired

    private AuthenticationFailureHandler authenticationFailureHandler;

    @Autowired

    private UserDetailsService userDetailService;

    @Override

    public void configure(HttpSecurity http) throws Exception {

        // TODO Auto-generated method stub

        SmsCodeAuthenticationFilter smsCodeAuthenticationFilter = new SmsCodeAuthenticationFilter();

        smsCodeAuthenticationFilter.setAuthenticationManager(http.getSharedObject(AuthenticationManager.class));

        smsCodeAuthenticationFilter.setAuthenticationSuccessHandler(authenticationSuccessHandler);

        smsCodeAuthenticationFilter.setAuthenticationFailureHandler(authenticationFailureHandler);

        SmsCodeAuthenticationProvider smsCodeAuthenticationProvider = new SmsCodeAuthenticationProvider();

        smsCodeAuthenticationProvider.setUserDetailService(userDetailService);

        http.authenticationProvider(smsCodeAuthenticationProvider).addFilterAfter(smsCodeAuthenticationFilter,

                UsernamePasswordAuthenticationFilter.class);

    }

}

当前配置是添加在core模块中的一段安全配置，注意下这段代码目前是有点问题的，比如userDetails在core中没有实现，而在browser中的实现，是取用户名密码，此时短信登录逻辑肯定是不通过的，这些逻辑业务会在之后重构。同时，这段配置是独立的，需要将这段配置添加到主配置，也就是browser模块的配置文件中。

http.addFilterBefore(smsCodeFilter, UsernamePasswordAuthenticationFilter.class)

.addFilterBefore(validateCodeFilter, UsernamePasswordAuthenticationFilter.class).formLogin()// http请求使用表单登录方式验证

.loginPage("/authentication/require")// 设置登录之后跳转的URL

.loginProcessingUrl("/authentication/form")// 设置登录form

.successHandler(authenticationSuccessHandlerConfig).failureHandler(authenticationFailureHandlerConfig)

.and().rememberMe().tokenRepository(persistentToeknRepository())

.tokenValiditySeconds(securityProperties.getBrowser().getRememberMeSeconds())

.userDetailsService(userDetailsService).and().authorizeRequests()// 表示接下来都是对请求的授权配置

.antMatchers(securityProperties.getBrowser().getLoginPage(), "/authentication/form",

"/authentication/require", "/code/\*")// 设置请求地址

.permitAll()// 设置匹配的请求，允许访问

.anyRequest()// 任何请求

.authenticated()// 都需要身份认证

.and().csrf()// 暂时关闭跨域请求拦截策略

.disable().apply(smsCodeAuthenticationSecurityConfig);

也就是在整段配置的最末尾添加了core中的配置类，springsecurity支持分段配置。

### 3-11.短信登录配置及重构

在之前的内容中，将整个验证码生成，发送，以及短信登录流程全部编写完成，但是在代码中存在有非常多的问题，代码质量并不高，在实际开发中，小到重复的字符串，比如表单的登录接口，以及controller中的API，两者应该是处理同一个请求，提供默认值并开放给用户配置，大到整段逻辑，比如图片验证码认证和短信验证码认证逻辑，也就是validateCodeFilter这两种认证方式的认证逻辑应该是相同的，这种代码，都应该重构一下。

重复字符串写在配置文件中，相同的业务逻辑重构成抽象类，按照这种逻辑重写代码，按照这种方式写的代码，当有配置发生改变的时候，只需要修改配置文件，而且，不需要在整个工程中找在那里用过这个字符串，只需要改一处代码就可以修改所有的位置。重构相同的业务逻辑可以大大提高代码的重用性，当需要添加新的相同的业务的时候，不需要再写重复的代码。同时，也将整个security的主配置文件分开来写，将通用配置写在core中，将模块相关配置写在相应的模块中，比如记住我这个功能，就是浏览器独有的，应该写在浏览器模块中，而其余配置都是浏览器和APP端通用的，可以直接写在core模块中。

在本案例中，代码都是先解释流程，再重构，而在实际的开发中，如果熟练之后，可以直接按照这种方式写代码。

将整个流程重构一次

1. 重构ValidateCodeController 验证生成请求处理

/\*\*

\* @author : MT

\* @date 创建时间：2018年12月13日 下午3:40:27

\* @version 1.0

\*/

@RestController

public class ValidateCodeController {

    @Autowired

    private Map<String, ValidateCodeProcessor> validateCodeProcessors;

    @GetMapping(SecurityConstants.DEFAULT\_VALIDATE\_CODE\_URL\_PREFIX + "/{type}")

    public void createCode(HttpServletRequest request, HttpServletResponse response, @PathVariable String type)

            throws Exception {

        validateCodeProcessors.get(type + "CodeProcessor").create(new ServletWebRequest(request, response));

    }

}

重构过程中，验证码生成请求封装为常量，并在控制器成员变量中注入了容器中所有validateCodeProcessor实现，即验证码处理器，根据请求中携带的参数名不同，调用不同的验证码生成处理器。

其中，将所有的常量封装在securityConstants中

public interface SecurityConstants {

    /\*\*

     \* 默认处理验证码的URL前缀

     \* 会通过验证码过滤器

     \*/

    static final String DEFAULT\_VALIDATE\_CODE\_URL\_PREFIX = "/code";

    /\*\*

     \* 当请求需要身份认证时，跳转的URL

     \* @see SecurityController

     \*/

    static final String DEFAULT\_UN\_AUTHENTICATION\_URL = "/authentication/require";

    /\*\*

     \* 默认用户名密码登录请求处理URL

     \*/

    static final String DEFAULT\_SIGN\_IN\_PROCESSING\_URL\_FORM = "/authentication/form";

    /\*\*

     \* 默认的手机登录请求处理URL

     \*/

    static final String DEFAULT\_SIGN\_IN\_PROCESSING\_URL\_MOBILE = "/authentication/mobile";

    /\*\*

     \* 默认的登录页面

     \*/

    static final String DEFAULT\_SIGN\_PAGE\_URL = "/demo-login.html";

    /\*\*

     \* 默认从请求中获取图片验证码的字段名，name

     \*/

    static final String DEFAULT\_PARAMETER\_NAME\_CODE\_IMAGE = "imageCode";

    /\*\*

     \* 默认从请求中获取短信验证码的字段名 name

     \*/

    static final String DEFAULT\_PARAMETER\_NAME\_CODE\_SMS = "smsCode";

    /\*\*

     \* 默认从请求中获取手机号的字段名 name

     \*/

    static final String DEFAULT\_PARAMETER\_NAME\_CODE\_MOBILE = "mobile";

    /\*\*

     \* 默认的使用安全模块的图片验证码生成逻辑类名

     \* 名字一定要是此名，不然代码逻辑或出现问题

     \* 必须是全小写+CodeGenerator 大小写敏感

     \*/

    static final String DEFAULT\_IMAGE\_CODE\_GENERATOR\_CLASS\_NAME = "imageCodeGenerator";

}

1. 重构AbstractValidateCodeProcessor

/\*\*

\* Title: ValidateCodeProcessor

\* Description: 验证码处理器接口，用来创建验证码以及验证验证码是否正确

\* Company: Seehope

\* @author MT

\* @date 2019年2月25日

\*/

public interface ValidateCodeProcessor {

    /\*\*

     \* 创建验证码

     \* 生成，存放，发送到客户端三步

     \* @param request

     \* @throws Exception

     \*/

    void create(ServletWebRequest request) throws Exception;

    /\*\*

     \* 验证当前请求中存放的验证按与生成时存放在session中的验证码是否匹配。

     \* @param request

     \* @throws Exception

     \*/

    void validate(ServletWebRequest request) throws Exception;

}

/\*\*

\* 抽象出所有验证码的生成逻辑

\* 分为三步，即，生成验证码，存储验证码，发送验证码

\* 生成逻辑交由不同的实现类实现，发送验证码也同理

\* 如果验证码生成逻辑不会发生更改，只是需要更改验证码生成方式，那么此类不需要修改，只需要修改生成类的实现

\* 只有在三步不满足业务逻辑，不在停留在生成，存储，发送流程，此类才会被修改

\*/

public abstract class AbstractValidateCodeProcessor<C extends ValidateCode> implements ValidateCodeProcessor {

    private static final Logger log = LoggerFactory.getLogger(AbstractValidateCodeProcessor.class);

    public static final String SESSION\_KEY\_PREFIX = "VALIDATE\_CODE\_IN\_SESSION\_FOR\_";

    /\*\*

     \* spring 框架再扫描到该类中有此注入类，那么会将所有ValidateCodeGenerator的实现注入到该表中

     \* 以BeanName作为Key

     \* 收集系统中所有的 {@link ValidateCodeGenerator} 接口的实现。

     \*/

    @Autowired

    private Map<String, ValidateCodeGenerator> validateCodeGenerators;

    /\*\*

     \* 验证码存储逻辑，在web环境下可以放session中，在安卓下可以用redis放入数据中

     \* 之后会将存储业务开放成接口交由使用模块实现

     \*/

    private SessionStrategy sessionStrategy = new HttpSessionSessionStrategy();

    @Override

    public void create(ServletWebRequest request) throws Exception {

        C validateCode = generate(request);

        save(request, validateCode);

        send(request, validateCode);

    }

    /\*\*

     \* 生成校验码

     \* 此处，暂时对验证码生成方法做了进一步限定，接受前台传入的参数判断/code/之后的字符串

     \* 通过字符串匹配，调用相应的验证码生成器

     \* 如果传入的是sms则调用defaultSmsValidateCodeGenerator Bean

     \* 如果传入的是image则调用defaultImageVlidateCodeGenerator Bean

     \* 换言之，前端验证码生成请求限定为 /code/sms 或者 /code/image 后期会在优化当前逻辑

     \* @param request

     \* @return

     \*/

    @SuppressWarnings("unchecked")

    private C generate(ServletWebRequest request) {

        String type = getValidateCodeType(request).toString().toLowerCase();

        String generatorName = type + "CodeGenerator";

        log.info(type + " " + generatorName);

        Set<String> keySet = validateCodeGenerators.keySet();

        for (String string : keySet) {

            log.info("key: " + string);

        }

        ValidateCodeGenerator validateCodeGenerator = validateCodeGenerators.get(generatorName);

        if (validateCodeGenerator == null) {

            throw new ValidateCodeException("验证码生成器 :" + generatorName + " 不存在");

        }

        return (C) validateCodeGenerator.createCode(request);

    }

    /\*\*

     \* 保存验证码，将生成的验证码保存在session中。

     \* 名字一定为 SESSION\_KEY\_FOR\_CODE\_SMS

     \* 或者 SESSION\_KEY\_FOR\_CODE\_IMAGE

     \* 也是初步逻辑。后期会再次重构

     \* @param request

     \* @param validateCode

     \*/

    private void save(ServletWebRequest request, C validateCode) {

        String sessionKey = SESSION\_KEY\_PREFIX + getValidateCodeType();

        log.info(sessionKey);

        sessionStrategy.setAttribute(request, sessionKey, validateCode);

    }

    /\*\*

     \* 发送校验码，由子类实现

     \*

     \* @param request

     \* @param validateCode

     \* @throws Exception

     \*/

    protected abstract void send(ServletWebRequest request, C validateCode) throws Exception;

    @SuppressWarnings("unchecked")

    @Override

    public void validate(ServletWebRequest request) {

        ValidateCodeType codeType = getValidateCodeType(request);

        String sessionKey = SESSION\_KEY\_PREFIX + getValidateCodeType();

        /\*\*

         \* 获取存在session中的验证码

         \*/

        C codeInSession = (C) sessionStrategy.getAttribute(request, sessionKey);

        /\*\*

         \* 获取从前端传回来的验证码

         \*/

        String codeInRequest;

        try {

            codeInRequest = ServletRequestUtils.getStringParameter(request.getRequest(),

                    codeType.getParamNameOnValidate());

        } catch (ServletRequestBindingException e) {

            throw new ValidateCodeException("获取验证码的值失败");

        }

        log.info(codeType.getParamNameOnValidate());

        if (StringUtils.isBlank(codeInRequest)) {

            throw new ValidateCodeException(codeType + "请填写验证码");

        }

        if (codeInSession == null) {

            throw new ValidateCodeException(codeType + "验证码不存在");

        }

        if (codeInSession.isExpried()) {

            sessionStrategy.removeAttribute(request, sessionKey);

            throw new ValidateCodeException(codeType + "验证码已过期，请重新获取");

        }

        if (!StringUtils.equals(codeInSession.getCode(), codeInRequest)) {

            throw new ValidateCodeException(codeType + "验证码不正确");

        }

        sessionStrategy.removeAttribute(request, sessionKey);

    }

    /\*\*

     \* 根据请求的url获取校验码的类型

     \* 实现类的在CodeProcessor之前的名字全大写

     \* @param request

     \* @return

     \*/

    private ValidateCodeType getValidateCodeType(ServletWebRequest request) {

        String type = StringUtils.substringBefore(getClass().getSimpleName(), "CodeProcessor");

        return ValidateCodeType.valueOf(type.toUpperCase());

    }

    private String getValidateCodeType() {

        String type = StringUtils.substringBefore(getClass().getSimpleName(), "CodeProcessor");

        return type.toUpperCase();

    }

}

重构的过程中，将验证码的验证逻辑也封装在验证码处理器中，将验证码在session中的key重构，同时，添加判断当前验证码类型方法，通过子类实现的类名+CodeGenerator作为map中的key，在create方法中，调用对应验证码generator创建验证码，之后调用save方法存储，在当前案例中，将验证码存入了session，但是在之后的业务中，因为手机端并没有session标准，所有这块代码后续要需要重构，暂时先用session管理，最后，调用发送逻辑，发送作为抽象方法，交由子类实现发送逻辑，该抽象类有两个子类，imageCodeProcessor和smsCodeProcessor，前者将图片响应给浏览器，后者调用短信发送接口，将短信发送给用户，同时，getValidateCodeType方法得到的值也会是image或者sms。为了在认证验证码方法中判断当前是何种验证码，所以引入枚举类validateCodeType:

public enum ValidateCodeType {

    /\*\*

     \* 短信验证码

     \*/

    SMS {

        @Override

        public String getParamNameOnValidate() {

            return SecurityConstants.DEFAULT\_PARAMETER\_NAME\_CODE\_SMS;

        }

    },

    /\*\*

     \* 图片验证码

     \*/

    IMAGE {

        @Override

        public String getParamNameOnValidate() {

            return SecurityConstants.DEFAULT\_PARAMETER\_NAME\_CODE\_IMAGE;

        }

    };

    /\*\*

     \* 校验时从请求中获取的参数的名字

     \* @return

     \*/

    public abstract String getParamNameOnValidate();

}

之后，重构验证码处理器processor和生成器generator，具体代码改动幅度不大，故在此章中不过多赘述，在小结中有提供完成之后的模块代码。之后，整个验证码生成逻辑重构完成。

接着，重构验证码验证逻辑，对生成的验证码进行验证

@Component

public class ValidateCodeFilter extends OncePerRequestFilter implements InitializingBean {

    @Autowired

    private AuthenticationFailureHandler authenticationFailureHandler;

    @Autowired

    private SecurityProperties properties;

    /\*\*

     \* 系统中的验证码处理器

     \* 包括构建验证码逻辑和验证验证码逻辑

     \*/

    @Autowired

    private ValidateCodeProcessorHolder validateCodeProcessorHolder;

    private AntPathMatcher machers = new AntPathMatcher();

    /\*\*

     \* 验证码映射表，映射了什么字符串的请求使用什么验证码

     \*

     \* key:url value:validateCodeType

     \*/

    private Map<String, ValidateCodeType> urlMap = new HashMap<String, ValidateCodeType>();

    /\* (non-Javadoc)

     \* @see org.springframework.web.filter.GenericFilterBean#afterPropertiesSet()

     \*/

    @Override

    public void afterPropertiesSet() throws ServletException {

        // TODO Auto-generated method stub

        super.afterPropertiesSet();

        urlMap.put(SecurityConstants.DEFAULT\_SIGN\_IN\_PROCESSING\_URL\_FORM, ValidateCodeType.IMAGE);

        addUrlToMap(properties.getValidateCode().getImageCode().getUrls(), ValidateCodeType.IMAGE);

        urlMap.put(SecurityConstants.DEFAULT\_SIGN\_IN\_PROCESSING\_URL\_MOBILE, ValidateCodeType.SMS);

        addUrlToMap(properties.getValidateCode().getSmsCode().getUrls(), ValidateCodeType.SMS);

    }

    /\*\*

     \* 将系统中配置的需要校验验证码的URL根据校验的类型放入map

     \*

     \* @param urlString

     \* @param type

     \*/

    protected void addUrlToMap(String urlString, ValidateCodeType type) {

        if (StringUtils.isNotBlank(urlString)) {

            String[] urls = StringUtils.splitByWholeSeparatorPreserveAllTokens(urlString, ",");

            for (String url : urls) {

                urlMap.put(url, type);

            }

        }

    }

    /\* (non-Javadoc)

     \* @see org.springframework.web.filter.OncePerRequestFilter#doFilterInternal(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse, javax.servlet.FilterChain)

     \*/

    @Override

    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)

            throws ServletException, IOException {

        // TODO Auto-generated method stub

        // 如果当前请求是登录请求，那么进行验证码认证，如果不是登录请求，那么直接放行

        ValidateCodeType type = getValidateCodeType(request);

        if (type != null) {

            logger.info("校验请求(" + request.getRequestURI() + ")中的验证码,验证码类型" + type);

            try {

                validateCodeProcessorHolder.findValidateCodeProcessor(type)

                        .validate(new ServletWebRequest(request, response));

                logger.info("验证码校验通过");

            } catch (ValidateCodeException exception) {

                authenticationFailureHandler.onAuthenticationFailure(request, response, exception);

                return;

            } catch (Exception e) {

                // TODO Auto-generated catch block

                e.printStackTrace();

            }

        }

        filterChain.doFilter(request, response);

    }

    /\*\*

     \* 获取校验码的类型，如果当前请求不需要校验，则返回null

     \*

     \* @param request

     \* @return

     \*/

    private ValidateCodeType getValidateCodeType(HttpServletRequest request) {

        ValidateCodeType result = null;

        if (!StringUtils.equalsIgnoreCase(request.getMethod(), "get")) {

            Set<String> urls = urlMap.keySet();

            for (String url : urls) {

                if (machers.match(url, request.getRequestURI())) {

                    result = urlMap.get(url);

                }

            }

        }

        return result;

    }

}

在重构之后的过滤器中，对ValidateCodeProcessor做了封装处理，将系统中的validateCode封装到了对象中，并对该Map提供了一些基本操作，方便取出相应的验证码处理器来验证验证码。

/\*\*

\* <b>Description:依据spring的机制，如果在spring组件的成员变量中存在Map<String, ValidateCodeProcessor>

\* 对象，那么会将容器中ValidateCodeProcessor的所有实现类作为value,beanName作为key,存入该成员变量中.

\* 在此类中，包含系统中所有的验证码 处理器，并提供相应的一些工具方法

\* </b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.validate.code

\* <br><b>ClassName:</b> ValidateCodeProcessorHolder

\* <br><b>Date:</b> 2019年2月25日 下午4:35:09

\*/

@Component

public class ValidateCodeProcessorHolder {

    @Autowired

    private Map<String, ValidateCodeProcessor> validateCodeProcessors;

    /\*\*

     \* @param type 传入验证码认证方式，以供调用不同的验证码处理机制

     \* @return

     \*/

    public ValidateCodeProcessor findValidateCodeProcessor(ValidateCodeType type) {

        return findValidateCodeProcessor(type.toString().toLowerCase());

    }

    /\*\*

     \* @param type

     \* @return

     \*/

    public ValidateCodeProcessor findValidateCodeProcessor(String type) {

        String name = type.toLowerCase() + "CodeProcessor";

        ValidateCodeProcessor processor = validateCodeProcessors.get(name);

        if (processor == null) {

            throw new ValidateCodeException("验证码处理器" + name + "不存在");

        }

        return processor;

    }

}

在过滤器中，判断当前请求是使用图片还是短信验证码登录，之后调用相应的处理器，处理，至此，验证码验证逻辑重构完成。

接下来重构配置文件，按照设计思路，core模块封装全局配置文件，其余模块封装根据当前模块环境下的配置文件，比如基于浏览器的rememberme服务，这种配置在App端的时候并没有效果。

首先，在core中创建validateCodeSecurityConfig

/\*\*

\* <b>Description:验证码相关安全配置

\* 会在最开始添加验证码过滤器，如果是需要验证码，图形或者短信验证码的请求，会过验证逻辑

\* </b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.validate.code

\* <br><b>ClassName:</b> ValidateCodeSecurityConfig

\* <br><b>Date:</b> 2019年2月26日 上午10:23:04

\*/

@Component

public class ValidateCodeSecurityConfig extends SecurityConfigurerAdapter<DefaultSecurityFilterChain, HttpSecurity> {

    @Autowired

    private Filter validateCodeFilter;

    @Override

    public void configure(HttpSecurity http) throws Exception {

        // TODO Auto-generated method stub

        http.addFilterBefore(validateCodeFilter, AbstractPreAuthenticatedProcessingFilter.class);

    }

}

单独配置验证码验证过滤器，之后将表单配置写成可选配置，放在core模块中，等需需要表单登录配置的模块使用配置

/\*\*

\* <b>Description:表单登录配置项</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.authentication

\* <br><b>ClassName:</b> FormAuthenticationConfig

\* <br><b>Date:</b> 2019年2月26日 上午10:19:09

\*/

@Component

public class FormAuthenticationConfig {

    @Autowired

    protected AuthenticationSuccessHandler authenticationSuccessHandler;

    @Autowired

    protected AuthenticationFailureHandler authenticationFailureHandler;

    public void configure(HttpSecurity http) throws Exception {

        http.formLogin().loginPage(SecurityConstants.DEFAULT\_UN\_AUTHENTICATION\_URL)

                .loginProcessingUrl(SecurityConstants.DEFAULT\_SIGN\_IN\_PROCESSING\_URL\_FORM)

                .successHandler(authenticationSuccessHandler).failureHandler(authenticationFailureHandler);

    }

}

这里将表单登录配置单独封装在一个类中，接着配置默认的用户详情服务和密码编码器

DefaultUserDetailsService

/\*\*

\* <b>Description:</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.authentication

\* <br><b>ClassName:</b> DefaultUserDetailService

\* <br><b>Date:</b> 2019年2月26日 上午10:15:38

\*/

public class DefaultUserDetailsService implements UserDetailsService {

    private static final Logger log = LoggerFactory.getLogger(DefaultUserDetailsService.class);

    /\*

     \* (non-Javadoc)

     \*

     \* @see org.springframework.security.core.userdetails.UserDetailsService#

     \* loadUserByUsername(java.lang.String)

     \*/

    @Override

    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        log.warn("请配置 UserDetailsService 接口的实现.");

        throw new UsernameNotFoundException(username);

    }

}

在默认的用户登录配置中，提醒用户需要根据使用模块的不同，自己配置userdetailService的实现，不然抛出异常，同时，将该类通过配置文件的方式注入springboot容器。

/\*\*

\* <b>Description:认证相关的拓展点配置，配置在此的bean,业务系统都可以通过声明同类型或同名的bean来安全覆盖</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.authentication

\* <br><b>ClassName:</b> AuthenticationBeanConfig

\* <br><b>Date:</b> 2019年2月26日 上午10:12:54

\*/

@Configuration

public class AuthenticationBeanConfig {

    /\*\*

     \* 默认密码处理器

     \* @return

     \*/

    @Bean

    @ConditionalOnMissingBean(PasswordEncoder.class)

    public PasswordEncoder passwordEncoder() {

        return new BCryptPasswordEncoder();

    }

    /\*\*

     \* 默认认证器

     \*

     \* @return

     \*/

    @Bean

    @ConditionalOnMissingBean(UserDetailsService.class)

    public UserDetailsService userDetailsService() {

        return new DefaultUserDetailsService();

    }

}

配置文件负责检测目前在容器中是否存在UserDetailsService的实现，如果有，则不做任何操作，如果没有，那么将注册默认的实现。

之后，添加授权配制



首先，配置用户在没有登录的情况下允许访问的请求，而这些配置在整个系统中应该有且只有一个，所以在配置的时候，需要使用manager进行分析配置。

/\*\*

\* <b>Description:授权信息管理器，用于收集容器中所有authorizeConfigProvider</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.authorize

\* <br><b>ClassName:</b> AuthorizeConfigManager

\* <br><b>Date:</b> 2019年2月27日 下午4:50:33

\*/

public interface AuthorizeConfigManager {

    /\*\*

    \* <b>Description:</b><br>

    \* @param config

    \* @Note

    \* <b>Author:</b> <a href="" target="\_blank">Monty</a>

    \* <br><b>Date:</b> 2019年2月27日 下午4:51:45

    \* <br><b>Version:</b> 1.0

     \*/

    void config(ExpressionUrlAuthorizationConfigurer<HttpSecurity>.ExpressionInterceptUrlRegistry config);

}

权限处理器 authorizeProvider

/\*\*

\* <b>Description:授权配置提供器，各个模块和业务系统可以通过实现此接口向系统中添加授权配置</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.authorize

\* <br><b>ClassName:</b> AuthorizeConfigProvider

\* <br><b>Date:</b> 2019年2月27日 下午4:07:44

\*/

public interface AuthorizeConfigProvider {

    /\*\*

     \*

    \* <b>Description:返回的boolean表示配置中是否有针对anyrequest的配置，在整个配置授权中，应该有且仅有一个队anyrequest的配置，

    \* 如果所有的实现都没有针对anyrequest的配置，系统会自动添加一个anyrequest.authenticated的配置，如果有多个，则会抛出异常</b><br>

    \* @param config

    \* @return

    \* @Note

    \* <b>Author:</b> <a href="" target="\_blank">Monty</a>

    \* <br><b>Date:</b> 2019年2月27日 下午4:24:04

    \* <br><b>Version:</b> 1.0

     \*/

    boolean config(ExpressionUrlAuthorizationConfigurer<HttpSecurity>.ExpressionInterceptUrlRegistry config);

}

编写默认的授权实现类

/\*\*

\* <b>Description:</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.authorize

\* <br><b>ClassName:</b> DefaultAuthorizeConfigProvider

\* <br><b>Date:</b> 2019年2月27日 下午4:21:25

\*/

public class DefaultAuthorizeConfigProvider implements AuthorizeConfigProvider {

    @Autowired

    private SecurityProperties properties;

    @Override

    public boolean config(ExpressionUrlAuthorizationConfigurer<HttpSecurity>.ExpressionInterceptUrlRegistry config) {

        // TODO Auto-generated method stub

        config.antMatchers(SecurityConstants.DEFAULT\_UN\_AUTHENTICATION\_URL,

                SecurityConstants.DEFAULT\_SIGN\_IN\_PROCESSING\_URL\_MOBILE,

                SecurityConstants.DEFAULT\_VALIDATE\_CODE\_URL\_PREFIX + "/\*", properties.getBrowser().getLoginPage())

                .permitAll();

        return false;

    }

}

将常量中配置的，允许用户未登录情况下访问的地址添加到配置文件中，之后，添加处理器，检查系统中是否只存在一个授权配置。

/\*\*

\* <b>Description:用来管理授权配置，验证容器中是否只有一段对anyRequest的配置</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.authorize

\* <br><b>ClassName:</b> DefaultAuthorizeConfigManager

\* <br><b>Date:</b> 2019年2月27日 下午4:49:06

\*/

public class DefaultAuthorizeConfigManager implements AuthorizeConfigManager {

    @Autowired

    private List<AuthorizeConfigProvider> authorizeConfigProviders;

    @Override

    public void config(ExpressionUrlAuthorizationConfigurer<HttpSecurity>.ExpressionInterceptUrlRegistry config) {

        // TODO Auto-generated method stub

        boolean existAnyRequestConfig = false;

        String existAnyRequestConfigName = null;

        for (AuthorizeConfigProvider authorizeConfigProvider : authorizeConfigProviders) {

            boolean currentIsAnyRequestConfig = authorizeConfigProvider.config(config);

            if (existAnyRequestConfig && currentIsAnyRequestConfig) {

                throw new RuntimeException("重复的anyRequest配置:" + existAnyRequestConfigName + ","

                        + authorizeConfigProvider.getClass().getSimpleName());

            } else if (currentIsAnyRequestConfig) {

                existAnyRequestConfig = true;

                existAnyRequestConfigName = authorizeConfigProvider.getClass().getSimpleName();

            }

        }

        if (!existAnyRequestConfig) {

            config.anyRequest().authenticated();

        }

    }

}

管理器负责验证系统中是否只存在一个授权配制，如果已经有授权配制了，那么抛出异常。

最后，配置实用模块browser中的安全配置

/\*\*

\*

\* @author : MT

\* @date 创建时间：2018年5月5日 上午10:51:15

\* @version 1.0

\*/

@Configuration

public class BrowserSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Autowired

    private SecurityProperties securityProperties;

    @Autowired

    private DataSource dataSource;

    @Autowired

    private UserDetailsService userDetailsService;

    @Autowired

    private SmsCodeAuthenticationSecurityConfig smsCodeAuthenticationSecurityConfig;

    @Autowired

    private ValidateCodeSecurityConfig validateCodeSecurityConfig;

    @Autowired

    private FormAuthenticationConfig formAuthenticationConfig;

    /\*\*

     \* 使用spring sec 自带的编码解码规则

     \* 也可以使用自定义的编码解码规则，实现passwordencoder类，编码以及解码接口

     \* @return

     \*/

    @Bean

    public PasswordEncoder getPasswordEncoder() {

        return new BCryptPasswordEncoder();

    }

    @Bean

    public PersistentTokenRepository persistentToeknRepository() {

        JdbcTokenRepositoryImpl tokenRepository = new JdbcTokenRepositoryImpl();

        tokenRepository.setDataSource(dataSource);

        return tokenRepository;

    }

    @Override

    protected void configure(HttpSecurity http) throws Exception {

        // TODO Auto-generated method stub

        formAuthenticationConfig.configure(http);

        http.apply(validateCodeSecurityConfig).and().apply(smsCodeAuthenticationSecurityConfig).and().rememberMe()

                .tokenRepository(persistentToeknRepository())

                .tokenValiditySeconds(securityProperties.getBrowser().getRememberMeSeconds())

                .userDetailsService(userDetailsService).and().authorizeRequests()// 表示接下来都是对请求的授权配置

                .antMatchers(securityProperties.getBrowser().getLoginPage(), "/authentication/form",

                        "/authentication/require", "/code/\*")// 设置请求地址

                .permitAll()// 设置匹配的请求，允许访问

                .anyRequest()// 任何请求

                .authenticated()// 都需要身份认证

                .and().csrf()// 暂时关闭跨域请求拦截策略

                .disable().apply(smsCodeAuthenticationSecurityConfig);

    }

}

还有BrowserSecurityController中，的访问路径，将所有的重复出现的字符串配置成securityConstants中的常量。

至此，整个模块中，抽取出了所有的重复逻辑以及重复字符串，整个验证码生成，验证，登录流程配置完成，当然，也可以将默认情况下允许用户访问列表配置为配置项，注入到授权处理器中。

### 3-12.小结

在第三章的讲解过程中，实现了springSecurity的基本配置，基于表单用户名与密码的登录实现，通过自定义SpringSecurity组件，实现了短信登录以及图片验证码，短信验证码逻辑，并提出了一种高可重用性的设计方法，在写完之后将整个结构重构，最后，在章节结束之前，提供第三章最后完成的代码，有需要的可以参考一下。

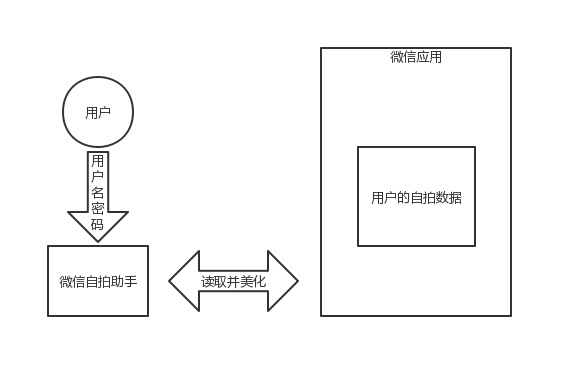
## 使用Spring Social开发第三方登录

### 4-1.OAuth协议简介

1. OAuth协议要解决的问题

在早期的业务场景中，举个例子，如果说我要在微信中开发一个微信拍照助手，用于美化用户的自拍数据，本来这对于微信以及第三方应用来说都是一件好事，对于微信来说，有项目组帮忙开发功能，对于第三方应用来说，可以使用微信的用户平台，但是，在没有Oauth协议的情况下，用户如何能将自拍数据交给第三方应用呢，只能通过用户名密码，先不说用户愿不愿意将用户名与密码交给服务商，就算用户同意，那么也存在非常多的问题。

首先，第三方应用可以访问用户微信上的所有数据，并不能限制第三方应用只能访问自拍数据，其次，用户只有修改密码才能收回授权，如果同时有非常多的第三方应用，那么如果你只想我不能用，如果修改密码，那么所有的第三方应用都不能用了，最后，密码泄露的可能性大大提高，如果使用这种方式授权，那么如果第三方应用中任何一个出现了安全问题，那么密码就会被泄漏，而，Oauth就是为了解决这个场景而诞生的。



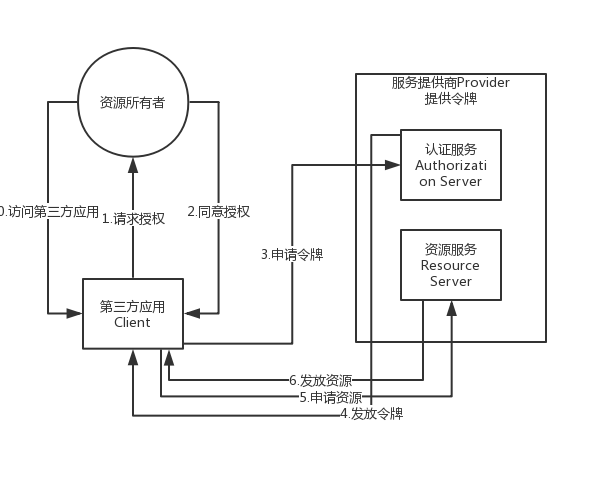
而在使用了OAuth协议的情况下，用户将不再将用户名密码授予服务商，而是提供OAuth票据，即Token，在使用token的情况下，以上问题都将得到解决，首先，Token中有明确指定服务商只能够访问自拍数据，其次token有有效期，只能在固定的时间内登录，超过时间无法登录，最后，根本就没有把用户名和密码交付给服务商，所有的问题都得到处理。

1. OAuth协议中的各种角色

按照上述业务场景，整个Oauth中大致可以分为3个角色，资源所有者，即上述用户，第三方应用提供商，即上述Client，服务提供商，即上述微信客户端，而在服务提供商之中，还包含有认证服务AuthorizationServer 以及资源服务ResourceServer。

其中，认证服务负责用户第三方应用发送的令牌申请请求是否真的经过了用户的授权，而资源服务负责认证用户的token是否合法以及发送资源给第三方服务。

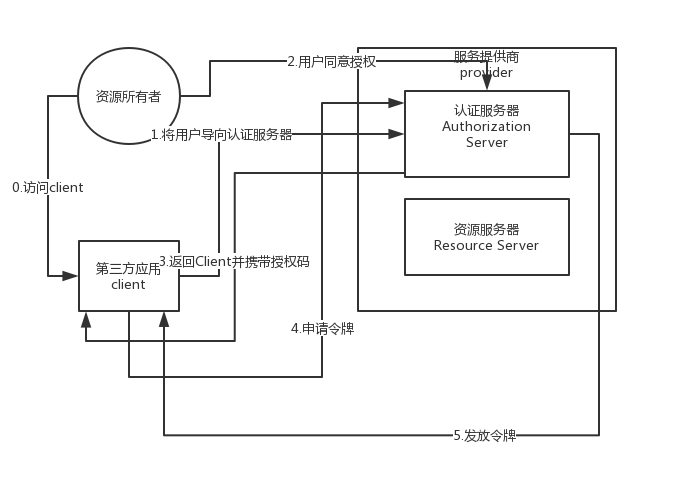
1. OAuth协议运行流程



将上述流程整理成图，大致情况下，认证流程如下图所示，首先，用户访问第三方应用，此时，第三方应用像客户请求授权，用户同意授权之后，将授权信息提交给服务提供商的认证服务，认证服务认证用户确实授权之后，发送令牌给第三方应用，第三方应用收到令牌之后，将令牌发送给服务提供商的资源服务，资源服务认证当前令牌合法之后，负责将令牌请求的资源发送给第三方应用。

大致上分为这6步，而在这6步中，第二步，请求授权非常的重要，在Oauthh协议中的授权模式，分为4种，即授权码模式，authorization，简化模式implicit，密码模式resource owner password credentials，客户端模式 client credentials，在之后的章节中，会详细的对授权码模式以及密码模式讲解，而简化模式和客户端模式由于使用的业务场景不多，所以在后续的章节中不详细的进行讲解，其中，授权码模式是最严密最完善的授权方式，而在目前市场上成熟的产品中，比如qq，微信，微博等第三方授权，全部都是使用授权码认证方式。

在授权码认证模式下，大致的流程如下图



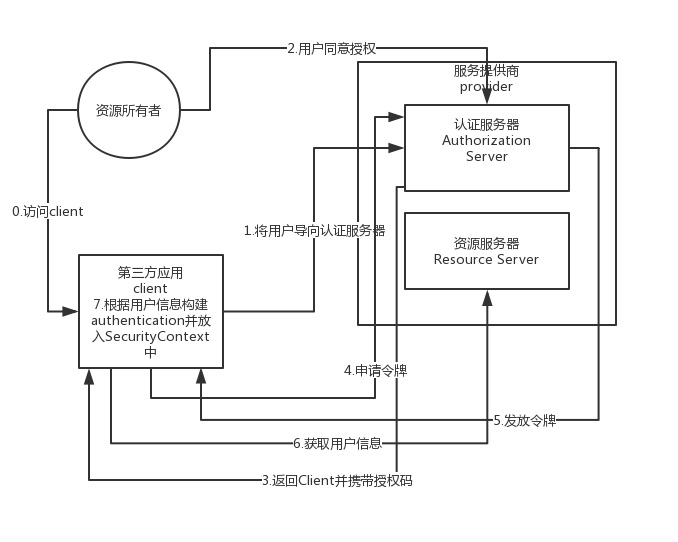
很明显的看出，授权码认证模式与上述流程有少许不同，在授权码认证模式下，资源所有者访问第三方应用时，第三方应用需要将用户导向认证服务器，用户在认证服务器提供的页面上完成授权，之后，认证服务器将验证码返回给第三方应用的接口中（需第三方应用和服务提供商约定好回调接口），第三方应用再讲授权码携带在请求中，像认证服务器申请令牌，最后，认证服务器通过之后，将令牌发放给第三方应用。

在这种情况下，用户同意授权是在服务商提供的页面上完成，而不是由第三方应用提交，防止了第三方应用伪造授权，之后，将授权码交由第三方应用的接口，再由第三方应用提交请求获取TOKEN，在这个流程里，需要第三方应用中存在回调服务，如果此时第三方应用为静态资源网站，那么将无法使用授权码授权模式，可以使用简化模式，在简化模式中，此时返回给第三方应用的直接就是Token，静态资源网站可以通过脚本获取到Token。

授权码模式将授权码发给第三方应用服务器，而不是浏览器，安全性能相对简化模式较高。这种模式，是4中模式中，流程最完整，最严密的。这也是为什么主流服务提供商，都是使用授权码模式完成授权的原因。

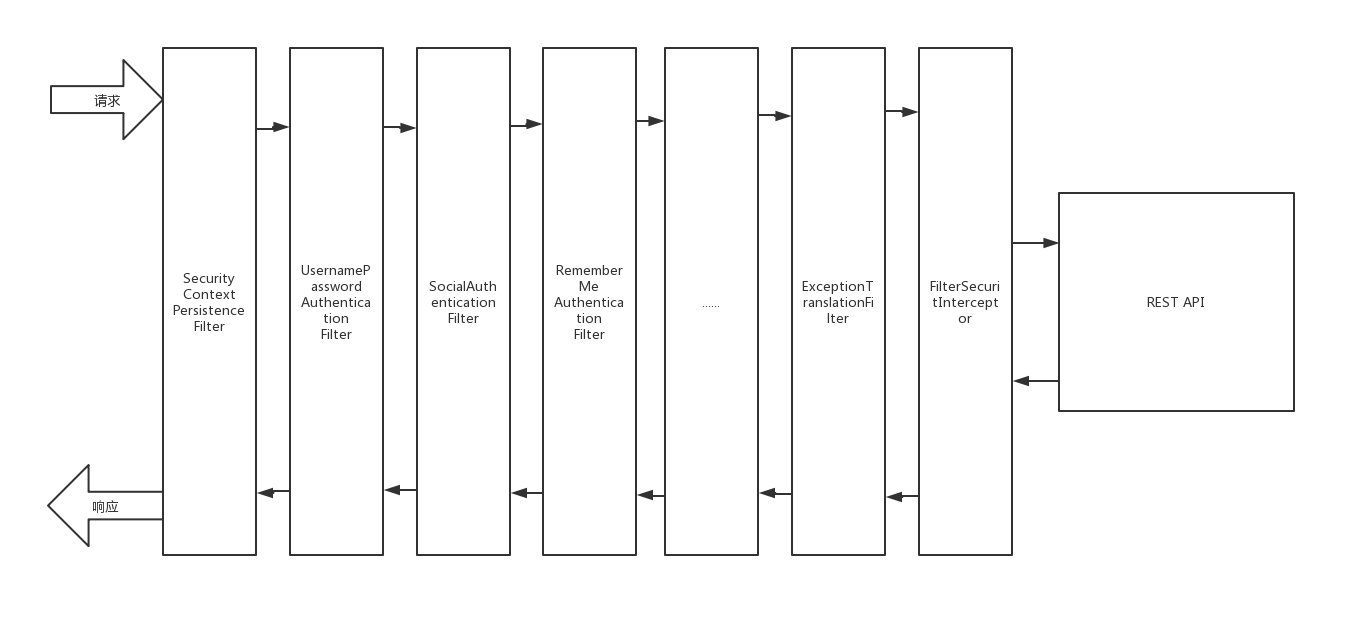
### 4-2.SpringSocial简介

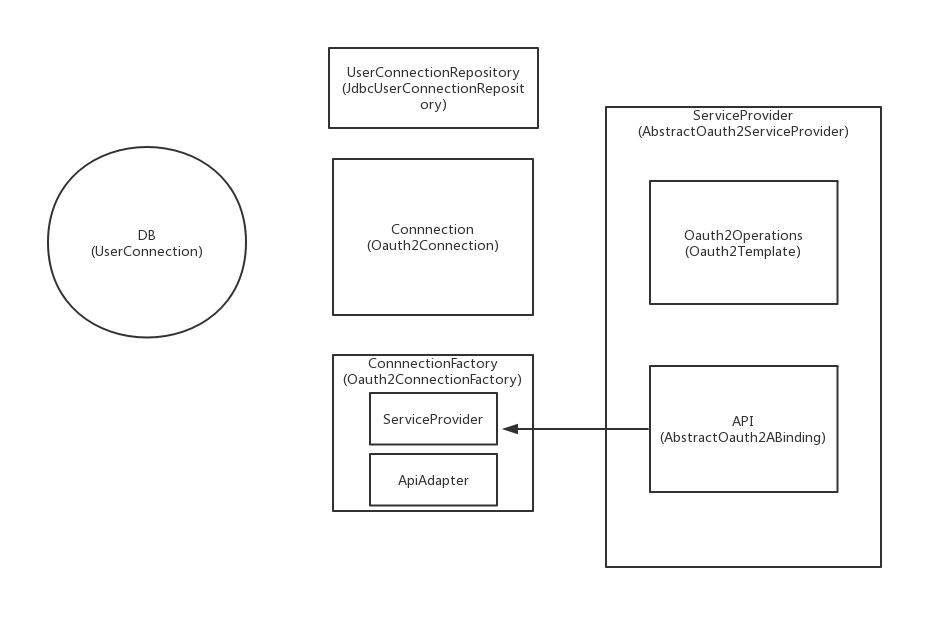
在上述小节中，简略的介绍了一下Oauth协议，那么Oauth协议和spring social之间有什么关系呢。



实际上，spring social在 Oauth 之后添加了两步，在上述场景中，尝试获取资源所有者自拍信息，但是如果不光获取用户自拍数据，同时获取用户基本信息，比如昵称，头像，简介等共有数据，同时，在第三方应用中，构建以认证的authentication并存入securityContext中，那么就相当于使用服务提供商的用户系统，在第三方应用中完整了用户登录，这就是所谓的第三方登录了。

在使用了spring social之后，会在spring security的拦截器链中添加Social AuthenticationFilter ，此过滤器中封装了上述认证流程与逻辑，引导第三方应用完成第三方登录流程。

 很明显，在UsernamePasswordAuthenticationFilter之后，添加了SocialAuthenticationFilter过滤器。增加了第三方登录业务流程支持，实现第三方登录。在整个认证流程中，使用的是Oauth2协议，目前，国内基本上因为使用Oauth协议的时间比较晚，可以说是目前来说，全部都是使用的Oauth2协议，即上述流程，而在国外，Twitter，fackbook，linkedin等，目前使用的是Oauth1协议，在本章中，暂不对Oauth做介绍，在Oauth2协议中，1-5步流程都是固定的，而在获取到用户信息，用户信息的封装可能会由于服务提供商的不同而有所变化，所以，这是个可配置项。



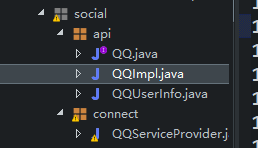
其中Oauth2Operatios会帮助我们完成1-5验证流程，API接口会帮我们完整第六步，因为，每个服务商返回的用户信息并没有统一的标准，所以这里只是一个抽象类，交由用户实现获取服务提供商用户信息的接口，从第七步开始，就跟服务提供商无关了，都是在第三方应用中完成，Connection对象是为了获取从服务提供商中获取到的用户信息，从ConnectionFactory中获取，其中包含用户信息，也就是说，整个serviceProvider被封装在ConnectionFactory中，CconnectionFactory调用ServiceProvider获取用户信息，将用户信息封装成Connection，但是这里返回的Connection对象是一个固定的数据类型，而用户信息返回数据并没有固定规范，所以需要使用ApiAdapter将服务提供商返回的Connection信息，适配为业务系统中自定义的数据结构，最后，将业务数据通过UserConnectionRepository存到DB中，在DB中，会创建一个UserConnection表，用来管理系统用户user，以及服务方提供的用户数据，例如wechat\_user或者qq\_user之间的映射关系，以上的所有类和抽象类，都是我们在编写springsocial中需要接触到的业务逻辑相关类。

以上即为springSocial登录的大致流程，熟悉了大致流程之后，就可以开始编写social代码了，同样的，在本章代码编写的过程中，也是先将代码的流程实现，之后在讲解框架的核心实现流程原理，最后再将实现的代码重构，提高代码质量。

首先，按照流程图，先构建ConnectionFactory，构建的过程中，需要提供ServiceProvider以及ApiAdapter接口的实现，而ServiceProvider需要Oauth2Operations以及API的实现。

### 4-3.开发QQ登录

首先，编写和QQ登录相关的代码，创建的新的package用于封装spring social相关内容。



首先，构建QQ接口，用来封装用户信息获取接口，处于上图中的ServiceProvider中的API接口，即AbstractOAuth2Binding的实现类，该实现类负责向服务提供商发送请求获取用户信息。

/\*\*

\* <b>Description:</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.social.api

\* <br><b>ClassName:</b> QQ

\* <br><b>Date:</b> 2019年3月1日 下午2:30:11

\*/

public interface QQ {

    QQUserInfo getUserInfo() throws JsonParseException, JsonMappingException, IOException;

}

设计getUserInfo方法，获取用户信息，用户信息具体字段有服务提供商提供，根据服务商提供的接口返回信息，封装UserInfo类型。

package net.seehope.core.social.api;

/\*\*

\* <b>Description:</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.social.api

\* <br><b>ClassName:</b> QQUserInfo

\* <br><b>Date:</b> 2019年3月1日 下午2:30:30

\*/

public class QQUserInfo {

    /\*\*

     \*  返回码

     \*/

    private String ret;

    /\*\*

     \* 如果ret<0，会有相应的错误信息提示，返回数据全部用UTF-8编码。

     \*/

    private String msg;

    /\*\*

     \*

     \*/

    private String openId;

    /\*\*

     \* 不知道什么东西，文档上没写，但是实际api返回里有。

     \*/

    private String is\_lost;

    /\*\*

     \* 省(直辖市)

     \*/

    private String province;

    /\*\*

     \* 市(直辖市区)

     \*/

    private String city;

    /\*\*

     \* 出生年月

     \*/

    private String year;

    /\*\*

     \*  用户在QQ空间的昵称。

     \*/

    private String nickname;

    /\*\*

     \*  大小为30×30像素的QQ空间头像URL。

     \*/

    private String figureurl;

    /\*\*

     \*  大小为50×50像素的QQ空间头像URL。

     \*/

    private String figureurl\_1;

    /\*\*

     \*  大小为100×100像素的QQ空间头像URL。

     \*/

    private String figureurl\_2;

    /\*\*

     \*  大小为40×40像素的QQ头像URL。

     \*/

    private String figureurl\_qq\_1;

    /\*\*

     \*  大小为100×100像素的QQ头像URL。需要注意，不是所有的用户都拥有QQ的100×100的头像，但40×40像素则是一定会有。

     \*/

    private String figureurl\_qq\_2;

    /\*\*

     \*  性别。 如果获取不到则默认返回”男”

     \*/

    private String gender;

    /\*\*

     \*  标识用户是否为黄钻用户（0：不是；1：是）。

     \*/

    private String is\_yellow\_vip;

    /\*\*

     \*  标识用户是否为黄钻用户（0：不是；1：是）

     \*/

    private String vip;

    /\*\*

     \*  黄钻等级

     \*/

    private String yellow\_vip\_level;

    /\*\*

     \*  黄钻等级

     \*/

    private String level;

    /\*\*

     \* 标识是否为年费黄钻用户（0：不是； 1：是）

     \*/

    private String is\_yellow\_year\_vip;

 /\*\*

     \* get/set

     \*/

编写QQImpl即API接口的实现，用户获取用户信息，该类需要继承AbstractOAuth2ApiBinding类。

/\*\*

\* <b>Description:

\* 相当于整个流程中ServiceProvier中的api接口，用户实现第6步，即获取用户信息，需要提供服务提供商所需要的参数

\* 在AbstractOAuth2ApiBinding中存在的String token,

\* 是一个常量所以qqImpl在系统中不是单例存在，而是原型，

\* 每一个用户都有一个独立的token。

\* </b>

\* <br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.social.api

\* <br><b>ClassName:</b> QQImpl

\* <br><b>Date:</b> 2019年3月1日 下午2:30:48

\*/

public class QQImpl extends AbstractOAuth2ApiBinding implements QQ {

    private static final Logger log = LoggerFactory.getLogger(QQImpl.class);

    /\*\*

     \* 获取openId的URL 在像qq申请openid的时候需要令牌，而令牌的获取以及认证的步骤在第前5步，目前交由spirng social的默认实现，最终能够获取到openID

     \*/

    private static final String URL\_GET\_OPENID = "https://graph.qq.com/oauth2.0/me?access\_token=%S";

    /\*\*

     \* 获取到用户的在服务商中的OpenID之后，调用该URL，可以获取用户信息，在访问该API时，需要提供access\_token，这个交由springsocial实现

     \* 以及oauth\_consumer\_key即appId，和在上一个URL中获取到的OpenId

     \*/

    private static final String URL\_GET\_USERINFO = "https://graph.qq.com/user/get\_user\_info?oauth\_consumer\_key=%S&openid=%S";

    private ObjectMapper objectMapper = new ObjectMapper();

    /\*\*

     \* 服务商给第三方应用接入用的ID

     \*/

    private String appId;

    /\*\*

     \* 和服务商中用户系统的ID相对应，如果是qq，则openID相当QQ号

     \*/

    private String openId;

    /\*\*

     \* 在调用父类方法的时候，需要由Oauth2Operatios实现中获取的accessToken，

     \* 同时，因为qq服务商要求accessToken必须携带在浏览器参数中，所以不能使用默认的父类构造器

     \* 默认的父类构造器会将accessToken存放在RequestHeader中所以，在调用父类构造器时，需要

     \* 提供参数，告知将accessToken存放在浏览器参数中

     \* @param accessToken

     \* @param appId

     \*/

    public QQImpl(String accessToken, String appId) {

        super(accessToken, TokenStrategy.ACCESS\_TOKEN\_PARAMETER);

        this.appId = appId;

        String url = String.format(URL\_GET\_OPENID, accessToken);

        /\*\*

         \* 按照qq的返回结果，返回字符串中包含appid以及 对应授权accesstoken的用户的openid

         \*/

        String result = getRestTemplate().getForObject(url, String.class);

        log.info(result);

        this.openId = StringUtils.substringBetween(result, "\"openid\":", "}");

    }

    @Override

    public QQUserInfo getUserInfo() throws JsonParseException, JsonMappingException, IOException {

        // TODO Auto-generated method stub

        String url = String.format(URL\_GET\_USERINFO, appId, openId);

        String result = getRestTemplate().getForObject(url, String.class);

        log.info(result);

        return objectMapper.readValue(result, QQUserInfo.class);

    }

}

之后，配置oauth2template，且qq提供的第三方登录服务是基于标准的OAuth2流程，所以覆盖该类的时候，只需要重写一下编码集即可

public class QQOAuth2Template extends OAuth2Template {

    /\*\*

     \* @param clientId

     \* @param clientSecret

     \* @param authorizeUrl

     \* @param accessTokenUrl

     \*/

    public QQOAuth2Template(String clientId, String clientSecret, String authorizeUrl, String accessTokenUrl) {

        super(clientId, clientSecret, authorizeUrl, accessTokenUrl);

        // TODO Auto-generated constructor stub

    }

    /\* (non-Javadoc)

     \* @see org.springframework.social.oauth2.OAuth2Template#createRestTemplate()

     \*/

    @Override

    protected RestTemplate createRestTemplate() {

        // TODO Auto-generated method stub

        RestTemplate restTemplate = super.createRestTemplate();

        restTemplate.getMessageConverters().add(new StringHttpMessageConverter(Charset.forName("UTF-8")));

        return restTemplate;

    }

}

至此构建ServiceProvider需要的两个组件构建完成，开始构建ServiceProvider

/\*\*

\* <b>Description:ServiceProvider实现类，负责处理前6步流程默认情况下，

\* 使用Oauth2Template默认实现</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.social.connect

\* <br><b>ClassName:</b> QQServiceProvider

\* <br><b>Date:</b> 2019年3月1日 下午3:51:22

\*/

public class QQServiceProvider extends AbstractOAuth2ServiceProvider<QQ> {

    private String appId;

    /\*\*

     \* 服务商提供的认证接口相当于第一步操作中第三方应用将用户导向认证服务器地址

     \*/

    private static final String URL\_AUTHORIZE = "https://graph.qq.com/oauth2.0/authorize";

    /\*\*

     \* 服务商提供的获取Token接口，需要将授权码提交到该接口中，相当于授权码模式中的第4步

     \*/

    private static final String URL\_ACCESS\_TOKEN = "https://graph.qq.com/oauth2.0/token";

    /\*\*

     \* 调用模块中API的实现，获取用户资源，因为用户资源在系统中存在有很多个，而不是默认spring封装的单例模式，所以在这部分代码中，

     \* 每个用户都是个新的封装类

     \*/

    @Override

    public QQ getApi(String accessToken) {

        // TODO Auto-generated method stub

        return new QQImpl(accessToken, appId);

    }

    /\*\*

     \* @param oauth2Operations

     \*/

    public QQServiceProvider(String appId, String appSecret) {

        // TODO Auto-generated constructor stub

        /\*\*

         \* 构造QQServiceProvider中，完整钱5步认证流程需要提供的4个参数,完整构造之后会获取到AccessToken，然后调用getApi方法，将accessToken传入到

         \* api接口中，去获取用户在服务商中的信息

         \*/

        super(new OAuth2Template(appId, appSecret, URL\_AUTHORIZE, URL\_ACCESS\_TOKEN));

    }

}

至此，ServiceProvider编写完成，serviceProvider负责完成用户的认证操作，以及获取用户的信息。当获取到用户信息之后，存放如QQUserInfo类中，可以在QQ互联中心查看相关文档，获取到用户信息之后，需要将qq互联返回的用户信息，转化为本系统中的用户，提供APIAdapter类负责将返回的数据转化为系统userconnection对象。

/\*\*

\* <b>Description:主要用于处理服务商返回的用户数据，并将用户数据与springSocial中的

\* 默认标准Connection相适配</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.social.connect

\* <br><b>ClassName:</b> QQAdapter

\* <br><b>Date:</b> 2019年3月1日 下午5:15:20

\*/

public class QQAdapter implements ApiAdapter<QQ> {

    /\*\*

     \* 用来测试当前API服务是否可用

     \* 这里直接返回为True即认为QQAPI永远可用，且参数不变

     \*/

    @Override

    public boolean test(QQ api) {

        // TODO Auto-generated method stub

        return true;

    }

    /\*\*

     \* 适配qq返回的用户信息以及SpringSocial标准connection对象的数据

     \*/

    @Override

    public void setConnectionValues(QQ api, ConnectionValues values) {

        // TODO Auto-generated method stub

        QQUserInfo qqUserInfo = api.getUserInfo();

        values.setDisplayName(qqUserInfo.getNickname());

        values.setImageUrl(qqUserInfo.getFigureurl\_qq\_1());

        // qq中并没有用，因为qq并没有个人主页如果是微博，则为个人主页信息;

        values.setProfileUrl(null);

        // 用户在服务商中的唯一标识

        values.setProviderUserId(qqUserInfo.getOpenId());

    }

    /\*\*

     \* 在QQ业务中，这段业务逻辑并没有用

     \*/

    @Override

    public UserProfile fetchUserProfile(QQ api) {

        // TODO Auto-generated method stub

        return null;

    }

    /\*\*

     \*

     \*/

    @Override

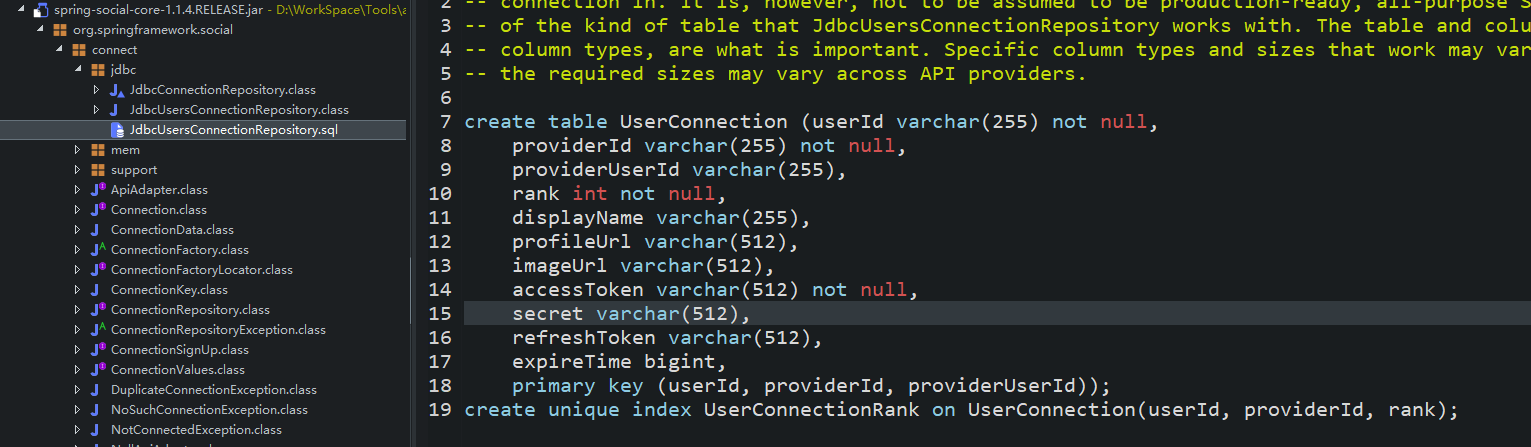
    public void updateStatus(QQ api, String message) {

        // TODO Auto-generated method stub

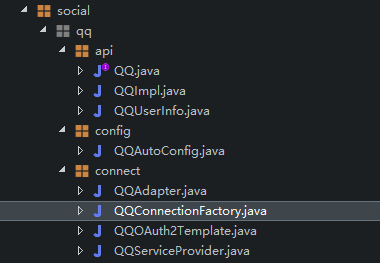
    }

}

当转化逻辑成功之后，会调用数据源，插入系统中的userconnection表，此表在springsocial源码中有定义



使用者可以将此sql源码考出，并导入自己的数据库中，创建userconnection表，如果用户系统中有规范，每个表必须有模块前缀，那么，用户也可以自定义表名前缀，但是必须以userconnection结尾。至此，可以接着编写jdbc模块，将connection中用户插入到数据库中。



/\*\*

\* <b>Description:</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.social.connect

\* <br><b>ClassName:</b> QQConnectionFactory

\* <br><b>Date:</b> 2019年3月1日 下午5:28:45

\*/

public class QQConnectionFactory extends OAuth2ConnectionFactory<QQ> {

    /\*\*

     \* @param providerId 服务商的唯一标识

     \* @param serviceProvider

     \* @param apiAdapter

     \*/

    public QQConnectionFactory(String providerId, String appId, String appSecret) {

        super(providerId, new QQServiceProvider(appId, appSecret), new QQAdapter());

        // TODO Auto-generated constructor stub

    }

}

最后，在通过配置类，将所有组件装配如spring容器

/\*\*

\* <b>Description: qq第三方登录配置，@ConditionalOnProperty，只有当配置文件中出现了这个配置项，此配置文件才生效</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.social.qq.config

\* <br><b>ClassName:</b> QQAutoConfig

\* <br><b>Date:</b> 2019年3月4日 下午2:49:12

\*/

@Configuration

@ConditionalOnProperty(prefix = "seehope.security.social.qq", name = "appId")

public class QQAutoConfig extends SocialAutoConfigurerAdapter {

    @Autowired

    private SecurityProperties properties;

    @Override

    protected ConnectionFactory<?> createConnectionFactory() {

        // TODO Auto-generated method stub

        return new QQConnectionFactory(properties.getSocial().getQq().getProviderId(),

                properties.getSocial().getQq().getAppId(), properties.getSocial().getQq().getAppSecret());

    }

}

在装配的时候，先判断配置文件中用户是否有配置qq的appid，如果没有配置，则不将qq登录相关配置注入spring容器中。

最后，配置jdbcUserConnectionFactory，该类会查找系统中所有的connectionFactory实现，最终调用适配当前登录方式的factory，并调用起第三方登录方式

/\*\*

\* <b>Description:</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.social

\* <br><b>ClassName:</b> SocialConfig

\* <br><b>Date:</b> 2019年3月1日 下午5:30:50

\*/

@Configuration

@EnableSocial

public class SocialConfig extends SocialConfigurerAdapter {

    @Autowired

    DataSource dataSource;

    @Autowired

    private SecurityProperties properties;

    /\*\*

     \* 交由用户实现的，如果使用模块没实现该接口则不注入

     \* 此接口用于当用户点击服务提供商登录的时候，如果服务提供商返回的用户在本系统没有值，那么直接用服务商提供的用户属性完成登录

     \*

     \*/

    @Autowired(required = false)

    private ConnectionSignUp ConnectionSignUp;

    /\* 如果要用微博登录，spirng Socia 开源社区中已经有微博登录模块，可以在spring social中下载

     \* connectionFactoryLocator查找系统中的ConnectionFactory当前应该用哪个connnectionFactory,在系统中可能会有很多ConnectionFactory，比如微信的，qq的，微博的

     \* Encryptors选择用户信息存入数据库中使用的加密方式，在当前案例中，为了演示方便，并不使用加密逻辑，方便查看用户数据

     \* (non-Javadoc)

     \* @see org.springframework.social.config.annotation.SocialConfigurerAdapter#getUsersConnectionRepository(org.springframework.social.connect.ConnectionFactoryLocator)

     \*/

    @Override

    public UsersConnectionRepository getUsersConnectionRepository(ConnectionFactoryLocator connectionFactoryLocator) {

        // TODO Auto-generated method stub

        JdbcUsersConnectionRepository jdbcUsersConnectionRepository = new JdbcUsersConnectionRepository(dataSource,

                connectionFactoryLocator, Encryptors.noOpText());

        jdbcUsersConnectionRepository.setTablePrefix("social\_");

        if (ConnectionSignUp != null) {

            jdbcUsersConnectionRepository.setConnectionSignUp(ConnectionSignUp);

        }

        return jdbcUsersConnectionRepository;

    }

    @Bean

    public SpringSocialConfigurer seehopeSocialSecurityConfig() {

        SeehopeSocialConfigurer configurer = new SeehopeSocialConfigurer(properties.getSocial().getFilterProcessUrl());

        configurer.signupUrl(properties.getBrowser().getSignUpUrl());

        return configurer;

    }

    @Bean

    public ProviderSignInUtils providerSignInUtils(ConnectionFactoryLocator connectionFactoryLocator) {

        return new ProviderSignInUtils(connectionFactoryLocator,

                getUsersConnectionRepository(connectionFactoryLocator));

    }

}

在配置该类的过程中，需要停供getFilterProcessUrl请求地址，即，前端触发所有第三方登录请求的前缀，将该字符串配置一下

/

\* <b>Description:</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.social

\* <br><b>ClassName:</b> SeehopeSocialConfigurer

\* <br><b>Date:</b> 2019年3月4日 下午4:30:07

\*/

public class SeehopeSocialConfigurer extends SpringSocialConfigurer {

    private String filterProcessUrl;

    public SeehopeSocialConfigurer(String filterProcessUrl) {

        this.filterProcessUrl = filterProcessUrl;

    }

    @SuppressWarnings("unchecked")

    @Override

    protected <T> T postProcess(T object) {

        // TODO Auto-generated method stub

        SocialAuthenticationFilter filter = (SocialAuthenticationFilter) super.postProcess(object);

        filter.setFilterProcessesUrl(filterProcessUrl);

        return (T) filter;

    }

    public String getFilterProcessUrl() {

        return filterProcessUrl;

    }

    public void setFilterProcessUrl(String filterProcessUrl) {

        this.filterProcessUrl = filterProcessUrl;

    }

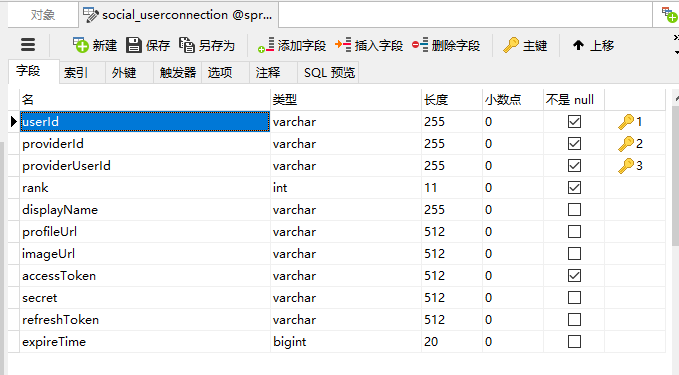
}

至此，qq登录的第三方应用代码编写完成，接着在qq互联中注册开发者身份，同时创建应用，在这一部中，创建应用可能需要提供企业信息，来获取权限，这里就不详细说明，了，同时，出于安全考虑，在配置文件中也不讲appid与appsecret放入了，需要测试环境的，可以自己操作，同时在，qq互联中也需要配置授权回调域名和接口，可以参照qq互联的文档进行配置，而在自己的应用中，如果需要发起第三方登录请求，

<!-- auth 开头请求会被socialAuthenticationFilter拦截 qq为providerid -->

<a href="/social/qq">qq登录</a>

可以通过该请求触发，其中/social开头是因为在filterProcessUrl中配置的social请求拦截请求前缀，当SocialAuthentication拦截到该请求之后，请求参数中包含/social/qq，即/filterPorcessUrl/providerId，会调用到JdbcUsersConnectionRepository对请求该调用的factory进行适配，之后，交由qqOAuthTemplate发送请求到配置的URL\_AUTHORIZE中请求授权码，同时，跳转到QQ的页面，用户同意授权之后，携带授权码返回到/filterProcessUrl/providerId，之后，在向URL\_ACCESS\_TOKEN中请求Token，返回token之后，调用用户写的api接口，即QQImpl，向URL\_GET\_OPENID请求该用户的OpenId，获取到OpenID之后，再向URL\_GET\_USERINFO发送请求，以获得userInfo，成功获取到UserInfo之后，调用apiAdapter，将服务商返回的用户信息，转化为系统自带的userconnection对象



并借由该表，完成之后的一系列，解绑，绑定，注册，第三方登录等操作。最后，要完成用户登录操作，还差最重要的一步，重构UserDeatilService

/\*\*

\* 自定义配置用户认证逻辑

\* @author : MT

\* @date 创建时间：2018年5月5日 下午12:02:37

\* @version 1.0

\*/

@Component

public class UserDetailServiceConfiguration implements UserDetailsService, SocialUserDetailsService {

    // @Autowired

    // private Mapper mapper;

    private static final Logger log = LoggerFactory.getLogger(UserDetailServiceConfiguration.class);

    @Autowired

    private PasswordEncoder encoder;

    @Override

    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        // TODO Auto-generated methobd stub

        // 根据用户名查找用户信息

        // login 页面传入的username参数，数据库中查出的密码，以及权限列表

        // 根据查找出来的用户信息判断用户是否被锁定，删除，登录状态过期，密码过期

        /\*\*

         \* String username, String password, boolean enabled,

            boolean accountNonExpired, boolean credentialsNonExpired,

            boolean accountNonLocked, Collection<? extends GrantedAuthority> authorities

            用户名 密码 是否删除 是否登录状态过期 是否密码过期 是否被锁定， 有一个返回false即登录失效，并报出相应标志位异常

            这里不一定要使用spring自带的user类，只需要实现userdetail接口的对象即可，可以使用自定义的，符合项目的自定义user对象，实现该接口逻辑

            $2a$10$tE.bdVy3JXixNSJHshXgE.ifO1JqPYMJBGRC9QUumTOE7wvjdLv.G

            $2a$10$mrehzJi7IqMB88AeQ/UXZOGYe0tdW18ycMK6ilQj8oY2rmSkqaosy

         \*/

        // 此操作应该注册的时候完成，也就是说，存在数据库中的密码信息，应该是加密之后的信息

        // 权限列表应该存在数据库中作为数据提取到security配置中 密码和权限列表，都要去读取

        log.info("表单 登录用户名 : " + username);

        return buildUser(username);

    }

    @Override

    public SocialUserDetails loadUserByUserId(String userId) throws UsernameNotFoundException {

        // TODO Auto-generated method stub

        log.info("社交 登录用户名" + userId);

        return buildUser(userId);

    }

    public SocialUserDetails buildUser(String userId) {

        String password = encoder.encode("123456");

        log.info("数据库密码为" + password);

        return new SocialUser(userId, password, true, true, true, true,

                AuthorityUtils.commaSeparatedStringToAuthorityList("admin"));

    }

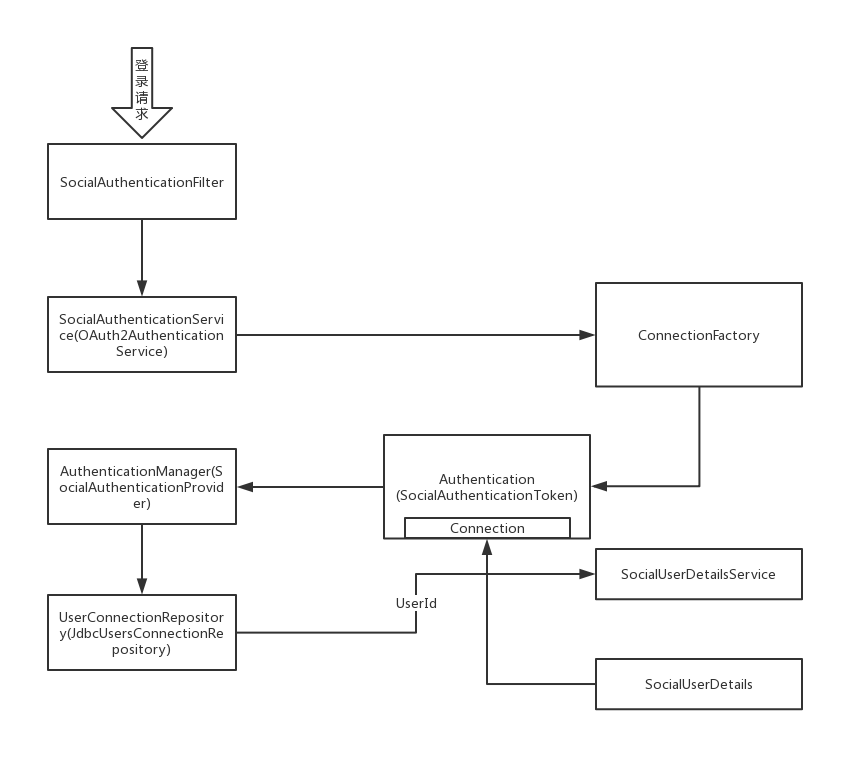
}

即重构用户返回逻辑，此时，userconnection表中存储了系统用户和服务商用户之间的对应关系，表中userId字段指本系统主键，providerId指服务商id，如qq，wechat，微博等，providerUserId值在服务商中，用户的唯一标识，即openid。

换句话说如果，服务商返回的openid在本系统有对应的userid，则表示该用户已经在本系统注册过，或者说处于绑定状态，如果没有，则表示该用户没有在本系统注册。最后，如果对应的userid在系统中有找到，则将用户对象返回到securityContext中，标识用户已经完成了登录逻辑。

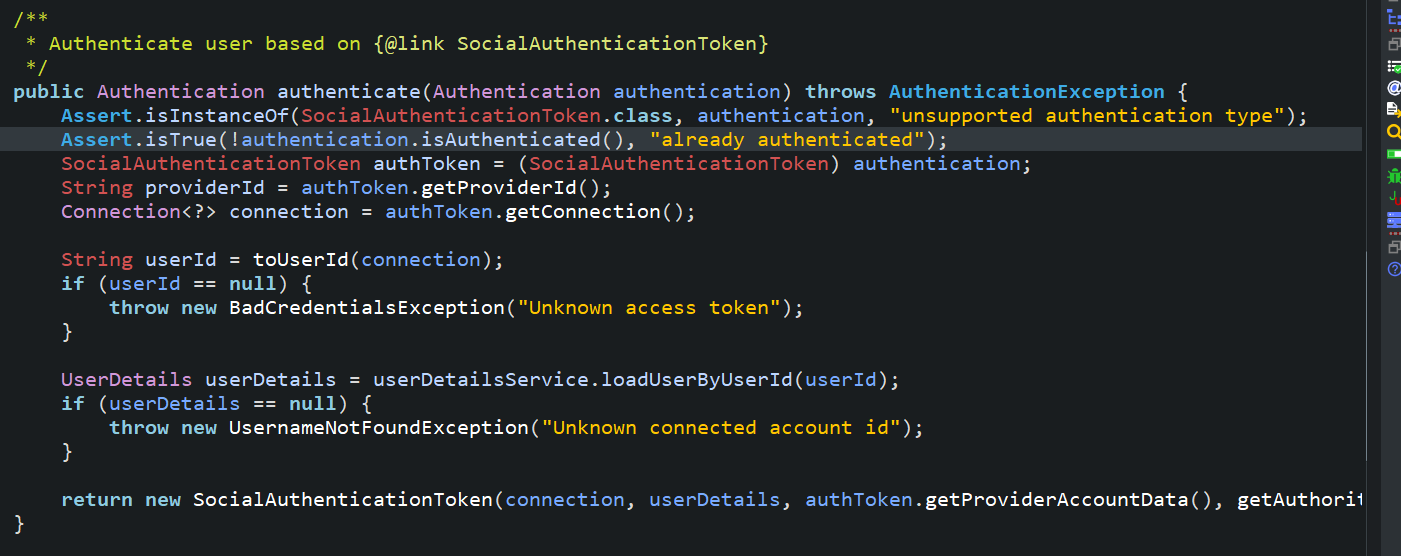
实际上不管是第三方登录还是别的一些逻辑，基本都是依靠于springsecurity的拦截器链模式，如果能完全理解该拦截器链模型，那么对springsecurity的很多扩展标准都会很容易上手。

### 4-4.处理注册逻辑



再次回顾一下整个流程，首先，/social/qq请求进入整个应用，被socialAuthenticationFilter拦截器捕获，之后调用SocialAuthenticationService中的方法，调用用户自己配置的ConnectionFactory，完整授权流程，即请引导用户跳转到服务商提供的授权页面，用户点击授权之后，携带授权码，返回在服务商开放平台中填写的回调接口，即/social/qq，收到授权码之后，oauth2template再将授权码发送到服务提供商的token申请接口，服务提供商验证授权码有效性之后，返回token到回调接口/social/qq，之后，应用就已经获取到Token了，应用在通过API接口的实现，本例中QQImpl，发送请求到服务商，获取用户ID，得到openID，之后在像服务商申请获取用户基本资源，获取到基本资源QQUserInfo之后，经过APIAdapter接口的实现，将QQUserInfo包装为系统userConnection对象，方便用户后续操作，同时，将connection对象，加入到未认证的SocialAuthentication对象中并将socialAuthentication放入securityContext中，交由下个过滤器处理，之后authenticationManager使用适配器模式，适配到SocialAuthenticationProvider之后，调用相

展开socialAuthenticationProvider类，查看social源码其中的



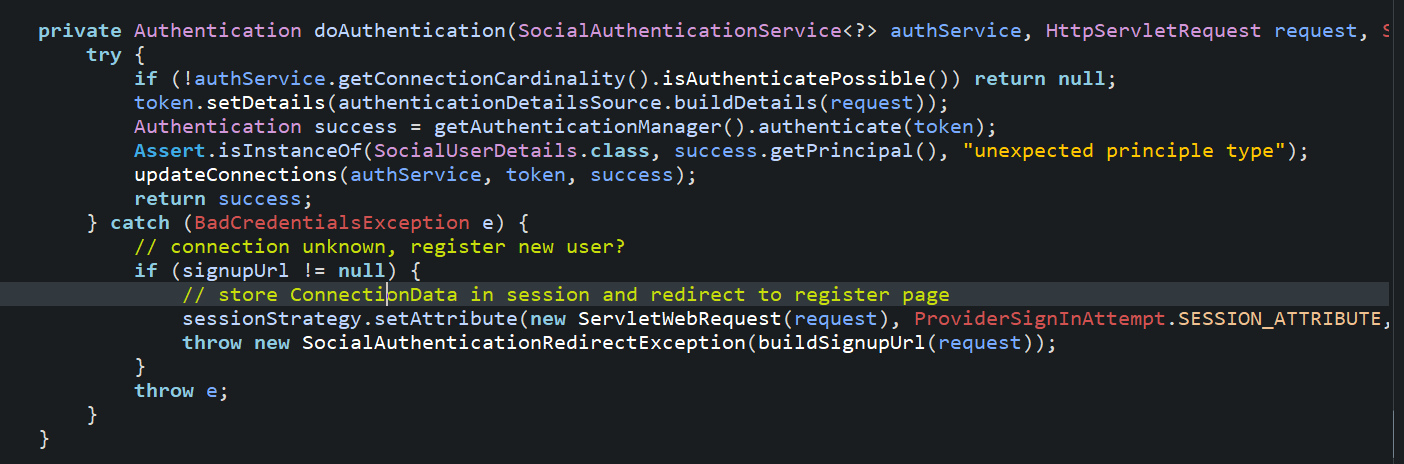
应处理方法，首先，

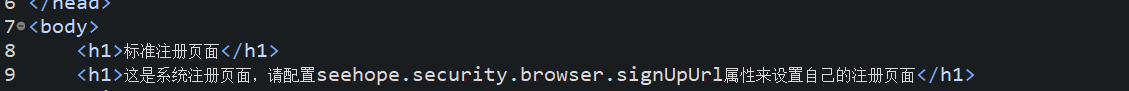
String userId = toUserId(connection);

方法会将connection中的userid取出，也就是说,使用userid字段作为用户已经登录的依据，之后，判断

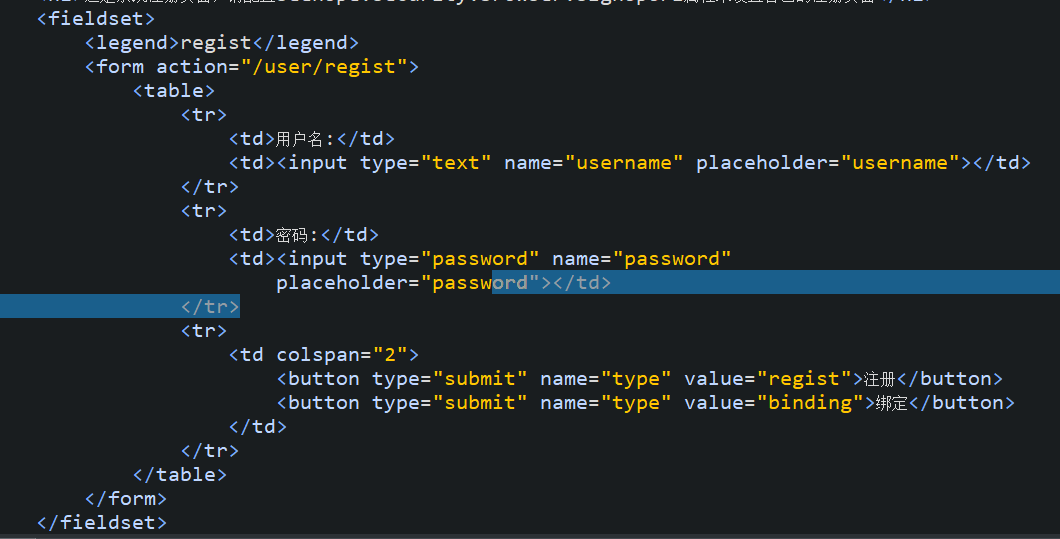


即，从数据库中查找，对应字段中是否有userid了，如果有，则表示用户已经登陆过，如果没有则表示用户第一次登录，如果第一次登录，会抛出badCredentialsException异常，该异常会被SocialAuthenticationFilter所捕获，捕获到之后，会判断系统中是否有signupUrl，如果不为空，则认为用户配置了注册页面，则跳到注册页面，引导用户完成注册逻辑，默认注册页为/signup，即，默认跳转到/signup,同时，又因为signup在系统中未授权，所以在这里添加系统注册页面，同时将地址授权，然而在大部分系统之中，注册业务逻辑都是由各自模块完成的，所以在core工程或者browser工程，提供的默认系统注册页面，都不需要填写具体的表单，只需要弹出提示，让各自模块实现注册页面即可





之后，像系统中添加注册页面



此时，跳转到注册页面就可能产生两种情况，即当前服务方用户在本系统中已经存在用户，或者当前服务方用户在本系统中并没有用户，对应两种逻辑，注册和绑定，不管是注册还是绑定，最终都会提交到/user/regist请求控制器中，此时，如果是第一次使用第三方登录，会跳到登录页面，此时，存在两个问题，其一，如何在html中显示用户信息，比如提示用户目前在用第三方登录，并显示用户的头像，其二，如何在点注册或者绑定之后，请求提交到方法中，如何处理，如何将userid传递回给springsocial，让spirngsocial将userid放到之前的userconnection对象中，组成一条完整的记录，存放如数据库，为了完整这一系列操作，我们需要一个工具类，首先，在配置文件socialConfig中注册该工具，需要提供两个参数

@Bean

public ProviderSignInUtils providerSignInUtils(ConnectionFactoryLocator connectionFactoryLocator) {

return new ProviderSignInUtils(connectionFactoryLocator,

getUsersConnectionRepository(connectionFactoryLocator));

}

之后，开放接口并封装vo对象提供给浏览器以供获取connection中的字段：

public class SocialUserInfo {

    private String providerId;

    private String providerUserId;

    private String nickName;

    private String headImg;

同时，在BrowserSecurityController中开放该接口，让前端浏览器能够获取从第三方中返回的用户信息

@GetMapping("/social/user")

public SocialUserInfo getUserInfo() {

Connection<?> connection = providerSignInUtils

.getConnectionFromSession(new ServletWebRequest(request, response));

SocialUserInfo socialUserInfo = new SocialUserInfo(connection.getKey().getProviderId(),

connection.getKey().getProviderUserId(), connection.getDisplayName(), connection.getImageUrl());

return socialUserInfo;

}

至此，当前端在注册页面中需要获取到用户服务商来源信息的时候，可以向该api发送请求以获取用户信息。

然后在demo模块中编写该请求处理 最后，将该请求配置到放行路径

@PostMapping("/regist")

public void regist(User user, HttpServletRequest request) {

log.info("------user regist------");

/\*

不管是注册用户还是绑定用户，都会拿到从前端传一个用户的唯一标识

从前端获取到用户输入的用户名之后，经过验证逻辑，将用户名以及在之前userConnection中的数据一起传入userconnection表中，

下次用户在登录的时候，social就能获取到当前用户在数据库中的映射，从而第三方用户就有值，可以直接登录了

\*/

String userId = user.getUsername();

// 注册或者绑定逻辑

//该方法会将从前端注册后返回的userId获取之后，传入到springsocial中，springsocial将对应的userconnection表中userid插入，表示绑定或者注册成功

providerSignInUtils.doPostSignUp(userId, new ServletWebRequest(request, response));

log.info(ReflectionToStringBuilder.toString(user, ToStringStyle.MULTI\_LINE\_STYLE));

}

最后，当下一次登陆的时候，

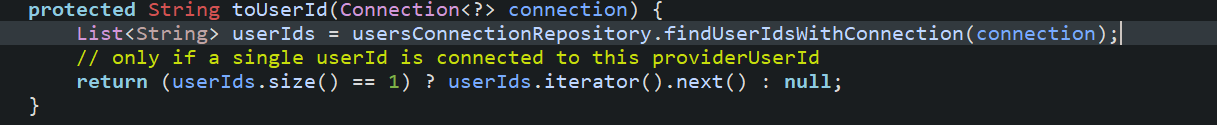


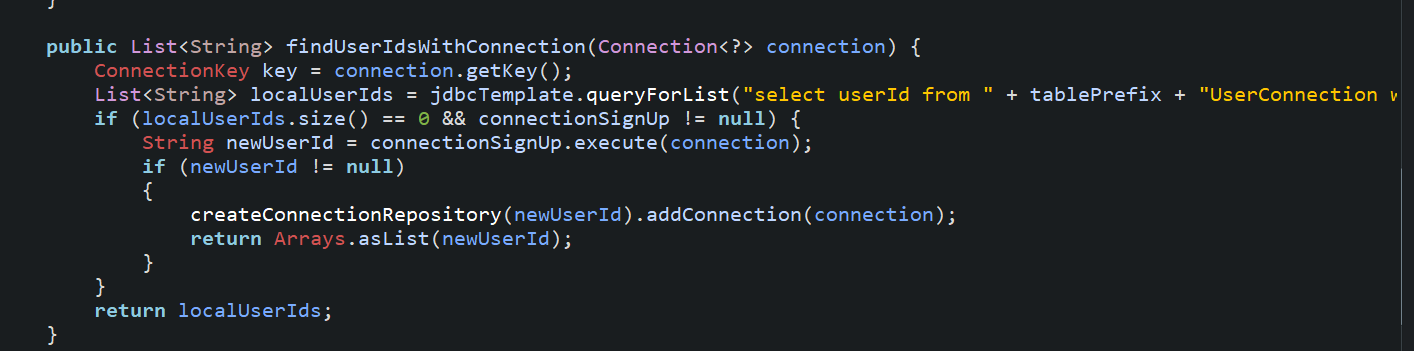
还是回到该方法，这时候，userid有值，则调用UserDetailsService，获取到用户信息之后，包装成token之后，放入securityContext中，表示用户已经登陆了。

至此，用户注册流程配置完成。

在目前的很多系统之中，第三方登录的流程有些不同，在有的系统中，当用户使用第三方登录之后，如果在系统中没有找到对应的绑定记录，那么则直接给用户注册一个默认的账号，之后跳转入系统，而不是引导用户到登录页，在springsocial中也可以实现该业务逻辑。

还是在socialAuthenticationprovider中，的toUserId方法，在该方法中





在该方法中，先试检查数据库中是否有对应的userid，其次，检测容器中是否有connectionSignUp的实现，如果有，则调用该实现的execute方法，从而获取newUserId，并最后返回newUserId到provider，所以，只需要实现connectionSignUp，即可让用户自动注册。

在demo项目中添加

@Component

public class DemoConnectionSignUp implements ConnectionSignUp {

    @Override

    public String execute(Connection<?> connection) {

        // TODO Auto-generated method stub

        // 根据社交用户信息 默认创建用户对象并返回用户唯一标识，在实际业务中，肯定不会服务商返回的用户昵称作为唯一标示，这里是示范，之后在具体的业务中需要更改

        return connection.getDisplayName();

    }

}

之后，将该类配置到socialConfig中

/\*\*

     \* 交由用户实现的，如果使用模块没实现该接口则不注入

     \* 此接口用于当用户点击服务提供商登录的时候，如果服务提供商返回的用户在本系统没有值，那么直接用服务商提供的用户属性完成登录

     \*

     \*/

    @Autowired(required = false)

    private ConnectionSignUp ConnectionSignUp;

    @Override

    public UsersConnectionRepository getUsersConnectionRepository(ConnectionFactoryLocator connectionFactoryLocator) {

        // TODO Auto-generated method stub

        JdbcUsersConnectionRepository jdbcUsersConnectionRepository = new JdbcUsersConnectionRepository(dataSource,

                connectionFactoryLocator, Encryptors.noOpText());

        jdbcUsersConnectionRepository.setTablePrefix("social\_");

        if (ConnectionSignUp != null) {

            jdbcUsersConnectionRepository.setConnectionSignUp(ConnectionSignUp);

        }

        return jdbcUsersConnectionRepository;

    }

至此，整个注册流程完成。

### 4-5.开发微信登录

微信登录的操作和qq登录的操作十分类似，但是因为微信和qq的协议有所不同，所以有些代码需要重构一下，同时，值得注意的是，如果在微信中需要开放第三方登录功能，需要在微信开放平台中注册应用而不是公众平台，微信登录接口和获取用户信息接口不在同一个位置。

仿照qq登陆的流程，重构微信登录，先构建微信返回用户模型

/\*\*

\* <b>Description:微信接口返回的用户基本信息

\* \* subscribe   用户是否订阅该公众号标识，值为0时，代表此用户没有关注该公众号，拉取不到其余信息。

    openid  用户的标识，对当前公众号唯一

    nickname    用户的昵称

    sex 用户的性别，值为1时是男性，值为2时是女性，值为0时是未知

    city    用户所在城市

    country 用户所在国家

    province    用户所在省份

    language    用户的语言，简体中文为zh\_CN

    headimgurl  用户头像，最后一个数值代表正方形头像大小（有0、46、64、96、132数值可选，0代表640\*640正方形头像），用户没有头像时该项为空。若用户更换头像，原有头像URL将失效。

    subscribe\_time  用户关注时间，为时间戳。如果用户曾多次关注，则取最后关注时间

    unionid 只有在用户将公众号绑定到微信开放平台帐号后，才会出现该字段。

    remark  公众号运营者对粉丝的备注，公众号运营者可在微信公众平台用户管理界面对粉丝添加备注

    groupid 用户所在的分组ID（兼容旧的用户分组接口）

    tagid\_list  用户被打上的标签ID列表

    subscribe\_scene 返回用户关注的渠道来源，ADD\_SCENE\_SEARCH 公众号搜索，ADD\_SCENE\_ACCOUNT\_MIGRATION 公众号迁移，ADD\_SCENE\_PROFILE\_CARD 名片分享，ADD\_SCENE\_QR\_CODE 扫描二维码，ADD\_SCENEPROFILE LINK 图文页内名称点击，ADD\_SCENE\_PROFILE\_ITEM 图文页右上角菜单，ADD\_SCENE\_PAID 支付后关注，ADD\_SCENE\_OTHERS 其他

    qr\_scene    二维码扫码场景（开发者自定义）

    qr\_scene\_str    二维码扫码场景描述（开发者自定义）</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.social.wechat.api

\* <br><b>ClassName:</b> WechatUserInfo

\* <br><b>Date:</b> 2019年3月5日 下午4:52:37

\*/

public class WechatUserInfo {

    private String openid;

    private String nickname;

    private String city;

    private String country;

    private String province;

    private String language;

    private String headimgurl;

    private String unionid;

    private String remark;

    private String subscribe\_scene;

    private String qr\_scene;

    private String qr\_scene\_str;

    private Integer subscribe;

    private Integer sex;

    private Integer groupid;

    private List<Integer> tagid\_list;

    private Date subscribe\_time;

之后，构建获取服务商返回用户信息接口API的实现类

public interface Wechat {

    WechatUserInfo getUserInfo(String openId);

}

/\*\*

\* <b>Description:在已经通过template获取到accesstoken的情况下获取用户的信息</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.social.wechat.api

\* <br><b>ClassName:</b> WechatImpl

\* <br><b>Date:</b> 2019年3月5日 下午5:02:50

\*/

public class WechatImpl extends AbstractOAuth2ApiBinding implements Wechat {

    private static final Logger log = LoggerFactory.getLogger(WechatImpl.class);

    // 微信获取用户详情API

    private static final String URL\_GET\_USERINFO = "https://api.weixin.qq.com/cgi-bin/user/info?openid=%s&lang=zh\_CN";

    private ObjectMapper objectMapper = new ObjectMapper();

    /\*\*

     \* 微信也要求accessToken放入请求参数之中

     \* @param accessToken

     \* @param appId

     \*/

    public WechatImpl(String accessToken) {

        super(accessToken, TokenStrategy.ACCESS\_TOKEN\_PARAMETER);

    }

    /\* resttemplate获取微信请求返回的时候

     \* 默认注册的字符转化器编码格式为ISO-8859-1 而微信返回的UTF-8的，所以覆盖了原来的方法

     \* (non-Javadoc)

     \* @see org.springframework.social.oauth2.AbstractOAuth2ApiBinding#getMessageConverters()

     \*/

    @Override

    protected List<HttpMessageConverter<?>> getMessageConverters() {

        // TODO Auto-generated method stub

        List<HttpMessageConverter<?>> messageConverters = super.getMessageConverters();

        messageConverters.remove(0);

        messageConverters.add(new StringHttpMessageConverter(Charset.forName("UTF-8")));

        return messageConverters;

    }

    @Override

    public WechatUserInfo getUserInfo(String openId) {

        // TODO Auto-generated method stub

        String url = String.format(URL\_GET\_USERINFO, openId);

        String result = getRestTemplate().getForObject(url, String.class);

        log.info("wechatUserInfo: " + result);

        try {

            return objectMapper.readValue(result, WechatUserInfo.class);

        } catch (Exception e) {

            log.warn(e.getMessage());

            throw new RuntimeException("获取用户信息失败 ");

        }

    }

}

构建WechatOauth2Template，在构建oauth的过程中，以为微信在获取授权码的流程中，直接将用户的openid也同样返回了，所以需要重新封装一下这一步返回的accessToken

/\*\*

\* <b>Description:

\* 微信的access-token信息，与标准的oauth2协议不同，在获取access-token的同时，就会返回openid

\* 所以这里集成了标准accessGrant，添加了openid字段，作为微信accesstoken封装策略

\*

\* </b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.social.wechat.connect

\* <br><b>ClassName:</b> WechatAccessGrant

\* <br><b>Date:</b> 2019年3月5日 下午7:58:36

\*/

public class WechatAccessGrant extends AccessGrant {

    /\*\*

     \* @param accessToken

     \* @param scope

     \* @param refreshToken

     \* @param expiresIn

     \*/

    public WechatAccessGrant(String accessToken, String scope, String refreshToken, Long expiresIn) {

        super(accessToken, scope, refreshToken, expiresIn);

        // TODO Auto-generated constructor stub

    }

    /\*\*

     \*

     \*/

    public WechatAccessGrant() {

        // TODO Auto-generated constructor stub

        super("");

    }

    /\*\*

     \*

     \*/

    private static final long serialVersionUID = 1L;

    private String openId;

}

/\*\*

\*

\* 完成微信的OAuth2认证流程的模板类。国内厂商实现的OAuth2每个都不同, spring默认提供的OAuth2Template适应不了，只能针对每个厂商自己微调。

\*

\* 用来完成OAuth2协议的前面几部

\* 微信的接口实现和标准的OAuth协议有比较大的不同，包括，请求accessToken的参数，以及接口返回的参数，需要对参数重新进行拼装

\* 重构整个流程

\*

\*/

public class WechatOAuth2Template extends OAuth2Template {

    private String clientId;

    private String clientSecret;

    private String accessTokenUrl;

    private static final String REFRESH\_TOKEN\_URL = "https://api.weixin.qq.com/sns/oauth2/refresh\_token";

    private static final Logger logger = LoggerFactory.getLogger(WechatOAuth2Template.class);

    /\*\*

     \* 微信获取授权码的url

        private static final String URL\_AUTHORIZE = "https://open.weixin.qq.com/connect/qrconnect";

     \* 微信获取accessToken的url

        private static final String URL\_ACCESS\_TOKEN = "https://api.weixin.qq.com/sns/oauth2/access\_token";

     \* @param clientId

     \* @param clientSecret

     \* @param authorizeUrl

     \* @param accessTokenUrl

     \*/

    public WechatOAuth2Template(String clientId, String clientSecret, String authorizeUrl, String accessTokenUrl) {

        super(clientId, clientSecret, authorizeUrl, accessTokenUrl);

        setUseParametersForClientAuthentication(true);

        this.clientId = clientId;

        this.clientSecret = clientSecret;

        this.accessTokenUrl = accessTokenUrl;

    }

    /\*  适配微信需要的获取token参数

     \* (non-Javadoc)

     \* @see org.springframework.social.oauth2.OAuth2Template#exchangeForAccess(java.lang.String, java.lang.String, org.springframework.util.MultiValueMap)

     \*/

    @Override

    public AccessGrant exchangeForAccess(String authorizationCode, String redirectUri,

            MultiValueMap<String, String> parameters) {

        StringBuilder accessTokenRequestUrl = new StringBuilder(accessTokenUrl);

        accessTokenRequestUrl.append("?appid=" + clientId);

        accessTokenRequestUrl.append("&secret=" + clientSecret);

        accessTokenRequestUrl.append("&code=" + authorizationCode);

        accessTokenRequestUrl.append("&grant\_type=authorization\_code");

        accessTokenRequestUrl.append("&redirect\_uri=" + redirectUri);

        logger.info(accessTokenUrl);

        return getAccessToken(accessTokenRequestUrl);

    }

    /\*\*

     \*  刷新token的方法

     \*/

    public AccessGrant refreshAccess(String refreshToken, MultiValueMap<String, String> additionalParameters) {

        StringBuilder refreshTokenUrl = new StringBuilder(REFRESH\_TOKEN\_URL);

        refreshTokenUrl.append("?appid=" + clientId);

        refreshTokenUrl.append("&grant\_type=refresh\_token");

        refreshTokenUrl.append("&refresh\_token=" + refreshToken);

        return getAccessToken(refreshTokenUrl);

    }

    /\*\*

     \*

    \* <b>Description:获取accessToken的方法

    \* accessToken 获取请求已经在上一步中适配过了

    \* </b><br>

    \* @param accessTokenRequestUrl

    \* @return

    \* @Note

    \* <b>Author:</b> <a href="" target="\_blank">Monty</a>

    \* <br><b>Date:</b> 2019年3月5日 下午10:27:51

    \* <br><b>Version:</b> 1.0

     \*/

    @SuppressWarnings("unchecked")

    private AccessGrant getAccessToken(StringBuilder accessTokenRequestUrl) {

        logger.info("获取access\_token, 请求URL: " + accessTokenRequestUrl.toString());

        // 获取到微信的返回值之后，会使用消息转化器进行转换 这里微信返回的html/text会被转化为 String 即，json数据

        String response = getRestTemplate().getForObject(accessTokenRequestUrl.toString(), String.class);

        logger.info("获取access\_token, 响应内容: " + response);

        Map<String, Object> result = null;

        try {

            // 将返回的json转化为map

            result = new ObjectMapper().readValue(response, Map.class);

        } catch (Exception e) {

            e.printStackTrace();

        }

        // 返回错误码时直接返回空

        if (StringUtils.isNotBlank(MapUtils.getString(result, "errcode"))) {

            String errcode = MapUtils.getString(result, "errcode");

            String errmsg = MapUtils.getString(result, "errmsg");

            throw new RuntimeException("获取access token失败, errcode:" + errcode + ", errmsg:" + errmsg);

        }

        // 拼接Token，除了OAuth2标准需要的参数之外，额外添加openId参数

        WechatAccessGrant accessToken = new WechatAccessGrant(MapUtils.getString(result, "access\_token"),

                MapUtils.getString(result, "scope"), MapUtils.getString(result, "refresh\_token"),

                MapUtils.getLong(result, "expires\_in"));

        accessToken.setOpenId(MapUtils.getString(result, "openid"));

        return accessToken;

    }

    /\*\*

     \*

     \* 该方法被调用了用来构建URL

     \* 构建获取授权码的请求。也就是引导用户跳转到微信的地址。

     \*/

    @Override

    public String buildAuthenticateUrl(OAuth2Parameters parameters) {

        String url = super.buildAuthenticateUrl(parameters);

        url = url + "&appid=" + clientId + "&scope=snsapi\_login";

        return url;

    }

    @Override

    public String buildAuthorizeUrl(OAuth2Parameters parameters) {

        return buildAuthenticateUrl(parameters);

    }

    /\*\*

     \* 微信返回的contentType是html/text，添加相应的HttpMessageConverter来处理。

     \*/

    protected RestTemplate createRestTemplate() {

        RestTemplate restTemplate = super.createRestTemplate();

        restTemplate.getMessageConverters().add(new StringHttpMessageConverter(Charset.forName("UTF-8")));

        return restTemplate;

    }

}

凑齐了oauth2template和api的实现之后，构建ServiceProvider

public class WechatServiceProvider extends AbstractOAuth2ServiceProvider<Wechat> {

    /\*\*

     \* 微信获取授权码的url

     \*/

    private static final String URL\_AUTHORIZE = "https://open.weixin.qq.com/connect/qrconnect";

    /\*\*

     \* 微信获取accessToken的url

     \*/

    private static final String URL\_ACCESS\_TOKEN = "https://api.weixin.qq.com/sns/oauth2/access\_token";

    /\*\*

     \* @param appId

     \* @param appSecret

     \*/

    public WechatServiceProvider(String appId, String appSecret) {

        super(new WechatOAuth2Template(appId, appSecret, URL\_AUTHORIZE, URL\_ACCESS\_TOKEN));

    }

    /\* (non-Javadoc)

     \* @see org.springframework.social.oauth2.AbstractOAuth2ServiceProvider#getApi(java.lang.String)

     \*/

    @Override

    public Wechat getApi(String accessToken) {

        return new WechatImpl(accessToken);

    }

}

之后，构建apiadapter用以将微信用户转化为connection对象

public class WechatAdapter implements ApiAdapter<Wechat> {

    /\*\*

     \* 和QQ不同的是，在微信协议中 没有使用accessToken换取用户openID流程

     \* 获取到accessToken的时候，直接能获取到用户的ID，就可以直接拿accessToken和openId换取用户的详情信息

     \* 所以不需要吧accessToken放到api接口的实现之中，而是放在适配器这里

     \*/

    private String openId;

    public WechatAdapter() {

    }

    public WechatAdapter(String openId) {

        this.openId = openId;

    }

    /\*\*

     \* @param api

     \* @return

     \*/

    @Override

    public boolean test(Wechat api) {

        return true;

    }

    /\*\*

     \* @param api

     \* @param values

     \*/

    @Override

    public void setConnectionValues(Wechat api, ConnectionValues values) {

        WechatUserInfo profile = api.getUserInfo(openId);

        values.setProviderUserId(profile.getOpenid());

        values.setDisplayName(profile.getNickname());

        values.setImageUrl(profile.getHeadimgurl());

    }

    /\*\*

     \* @param api

     \* @return

     \*/

    @Override

    public UserProfile fetchUserProfile(Wechat api) {

        return null;

    }

    /\*\*

     \* @param api

     \* @param message

     \*/

    @Override

    public void updateStatus(Wechat api, String message) {

        // do nothing

    }

}

之后，组件完成，构建connnectionFactory

/\*\*

\* <b>Description:</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.social.wechat.connect

\* <br><b>ClassName:</b> WechatConnectionFactory

\* <br><b>Date:</b> 2019年3月5日 下午10:48:53

\*/

public class WechatConnectionFactory extends OAuth2ConnectionFactory<Wechat> {

    /\*\*

     \* @param appId

     \* @param appSecret

     \*/

    public WechatConnectionFactory(String providerId, String appId, String appSecret) {

        super(providerId, new WechatServiceProvider(appId, appSecret), new WechatAdapter());

    }

    /\*\*

     \* 由于微信的openId是和accessToken一起返回的，所以在这里直接根据accessToken设置providerUserId即可，不用像QQ那样通过QQAdapter来获取

     \*/

    @Override

    protected String extractProviderUserId(AccessGrant accessGrant) {

        if (accessGrant instanceof WechatAccessGrant) {

            return ((WechatAccessGrant) accessGrant).getOpenId();

        }

        return null;

    }

    /\* (non-Javadoc)

     \* @see org.springframework.social.connect.support.OAuth2ConnectionFactory#createConnection(org.springframework.social.oauth2.AccessGrant)

     \*/

    public Connection<Wechat> createConnection(AccessGrant accessGrant) {

        return new OAuth2Connection<Wechat>(getProviderId(), extractProviderUserId(accessGrant),

                accessGrant.getAccessToken(), accessGrant.getRefreshToken(), accessGrant.getExpireTime(),

                getOAuth2ServiceProvider(), getApiAdapter(extractProviderUserId(accessGrant)));

    }

    /\* (non-Javadoc)

     \* @see org.springframework.social.connect.support.OAuth2ConnectionFactory#createConnection(org.springframework.social.connect.ConnectionData)

     \*/

    public Connection<Wechat> createConnection(ConnectionData data) {

        return new OAuth2Connection<Wechat>(data, getOAuth2ServiceProvider(), getApiAdapter(data.getProviderUserId()));

    }

    private ApiAdapter<Wechat> getApiAdapter(String providerUserId) {

        return new WechatAdapter(providerUserId);

    }

    private OAuth2ServiceProvider<Wechat> getOAuth2ServiceProvider() {

        return (OAuth2ServiceProvider<Wechat>) getServiceProvider();

    }

}

最终，将connectionFactory注册到spring容器中。

/\*\*

\* 微信登录配置

\*

\* @author zhailiang

\*

\*/

@Configuration

@ConditionalOnProperty(prefix = "seehope.security.social.wechat", name = "appId")

public class WechatAutoConfig extends SocialAutoConfigurerAdapter {

    @Autowired

    private SecurityProperties securityProperties;

    /\*

     \* (non-Javadoc)

     \*

     \* @see

     \* org.springframework.boot.autoconfigure.social.SocialAutoConfigurerAdapter

     \* #createConnectionFactory()

     \*/

    @Override

    protected ConnectionFactory<?> createConnectionFactory() {

        WechatProperties wechatProperties = securityProperties.getSocial().getWechat();

        return new WechatConnectionFactory(wechatProperties.getProviderId(), wechatProperties.getAppId(),

                wechatProperties.getAppSecret());

    }

    /\*\*

    \* <b>Description:绑定成功后返回视图，非常简单，只是返回一个只有一句绑定成功的html页面，如果用户有自己配置名字为wechatConnectedView的bean

    \* 则使用用户自定义的实现

    \* </b><br>

    \* @return

    \* @Note

    \* <b>Author:</b> <a href="" target="\_blank">Monty</a>

    \* <br><b>Date:</b> 2019年3月6日 下午4:57:26

    \* <br><b>Version:</b> 1.0

     \*/

    @Bean({ "connect/wechatConnected", "connect/wecahtConnect" })

    @ConditionalOnMissingBean(name = "wechatConnectedView")

    public View wechatConnectedView() {

        return new ConnectionResultView();

    }

}

至此，微信登录流程适配完成，可以发现，微信登录流程和qq登录流程十分相似，同时，在构建好一个第三方登录模块之后，再构建新的服务，会变得十分简单。大部分的代码都可以重用。同时，如果要集成微博登录，也可以从spring social官网上下载spring social weibo项目进行业务。

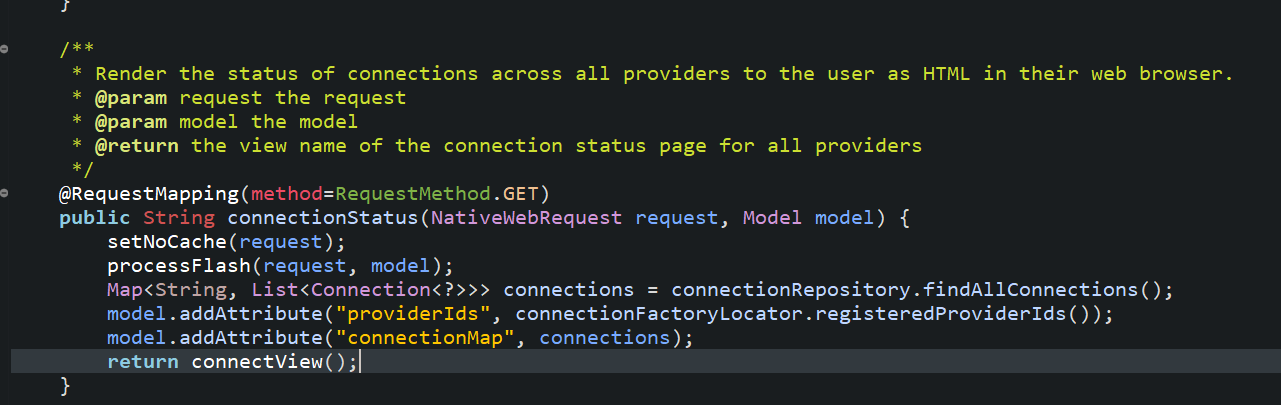
### 4-6.绑定和解绑处理

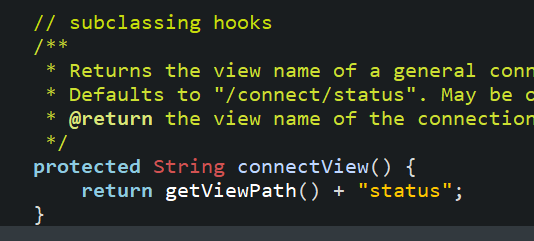
先来回顾一下当前成熟产品中的解绑和绑定业务。当用户在本系统中已经有注册之后，跳转到绑定页面，可以查看当前用户有绑定那些系统支持的服务商，将所有系统支持的服务商列表列出，并显示当前绑定状态，已绑定的可以选择解绑，未绑定的可以选择绑定。

如果整个系统流程都是基于spring social开发的，那么，所有用和服务商相关的业务流程都是通过userconnection表完成，期中，userid字段和其他两个主键的关联，显示了当前用户的绑定以及关联关系。同时，这里的providerId也是用户自己填写的。用于表示服务提供方。UserID表示本系统中的用户，如果本系统userid没有对应的provider则表示当前用户没有绑定，反之，则有绑定。

那么，此时需要解决的问题有两个，即如果获取当前用户绑定状态，以及如何添加绑定关系，以及如何解除绑定关系。

实际上，这些业务功能，在springsocial中已经封装好了。在connectController中

在该方法中，也就是说，只需要在用户登陆之后访问/connect get请求，则能够获取当前用户绑定状态，最后，返回一个connectView，实际上，这个connectview是一个



这里返回的是一个视图的路径，也就是说，会正在spring容器中找该视图，如果开发者没有配置该视图，最终会返回异常，此时，自定义该视图

/\*\*

\* <b>Description:

\* 交由springsocial使用，返回用户在系统中第三方登录关系视图

\*

\* 名字必须为connect/status social会检测容器中是否存在此bean

\* </b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.social

\* <br><b>ClassName:</b> ConnectionStatusView

\* <br><b>Date:</b> 2019年3月6日 下午4:27:14

\*/

@Component("connect/status")

public class ConnectionStatusView extends AbstractView {

    @Autowired

    private ObjectMapper objectMapper;

    @Override

    protected void renderMergedOutputModel(Map<String, Object> model, HttpServletRequest request,

            HttpServletResponse response) throws Exception {

        // TODO Auto-generated method stub

        @SuppressWarnings("unchecked")

        Map<String, List<Connection<?>>> maps = (Map<String, List<Connection<?>>>) model.get("connectionMap");

        Map<String, Boolean> result = new HashMap<String, Boolean>();

        for (String key : maps.keySet()) {

            result.put(key, CollectionUtils.isNotEmpty(maps.get(key)));

        }

        response.setContentType("application/json;charset=utf-8");

        response.getWriter().write(objectMapper.writeValueAsString(result));

    }

}

该类主要是用来简化connection中的数据模型，只需要将绑定与否的信息返回给前端即可，不需要返回太多信息。此时，当前端像/connect/status发送请求的时候，就能够获取用户是否已经绑定的信息。例如qq:true wechat:false这样的信息。

之后，处理第二个问题，即，用户如何绑定以及解绑

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Insert title here</title>

</head>

<body>

    <fieldset>

        <legend>标准绑定页面</legend>

        <form action="/connect/wechat" method="post">

            <button type="button">binding wechat</button>

        </form>

        <!-- 和绑定请求相同，只不过解绑是delete请求，因为在纯html种不能发起delete，所以这里解绑业务需要用ajax实现 -->

        <form action="/connect/wechat" method="delete">

            <button type="button">解绑 微信</button>

        </form>

    </fieldset>

</body>

</html>

在connectionController中也已经定义好了该业务逻辑，并且最终返回的视图名为

protected String connectView(String providerId) {

return getViewPath() + providerId + "Connect";

}

即，如果要配置微信绑定返回结果视图，那么应该配置

public class ConnectionResultView extends AbstractView {

    @Override

    protected void renderMergedOutputModel(Map<String, Object> model, HttpServletRequest request,

            HttpServletResponse response) throws Exception {

        // TODO Auto-generated method stub

        response.setContentType("text/html;charset=UTF-8");

        if (model.get("connection") == null) {

            response.getWriter().write("<h3>解绑成功</h3>");

        } else {

            response.getWriter().write("<h3>绑定成功</h3>");

        }

    }

}

注意，不需要将该结果视图配置为/connect/wechat或者/connect/qq 因为该结果视图不会只有qq或者微信使用，而是应该根据服务商的不同，通过不同的服务商名字返回，同时，在该视图中，也不再是返回一段json数据了，因为绑定成功或者解绑逻辑，也是通过服务商实现的，调用了服务商的接口，最终应该返回一个页面提示用户绑定成功或者解绑成功。

之后，将该类注册到wechat配置中

/\*\*

    \* <b>Description:绑定成功后返回视图，非常简单，只是返回一个只有一句绑定成功的html页面，如果用户有自己配置名字为wechatConnectedView的bean

    \* 则使用用户自定义的实现

    \* </b><br>

    \* @return

    \* @Note

    \* <b>Author:</b> <a href="" target="\_blank">Monty</a>

    \* <br><b>Date:</b> 2019年3月6日 下午4:57:26

    \* <br><b>Version:</b> 1.0

     \*/

    @Bean({ "connect/wechatConnected", "connect/wecahtConnect" })

    @ConditionalOnMissingBean(name = "wechatConnectedView")

    public View wechatConnectedView() {

        return new ConnectionResultView();

    }

名字和微信绑定。之后，spring social会自动帮我们完成绑定和解绑功能。

之后，所有有关服务方登录的逻辑全部完成。

### 4-7.单机Session管理

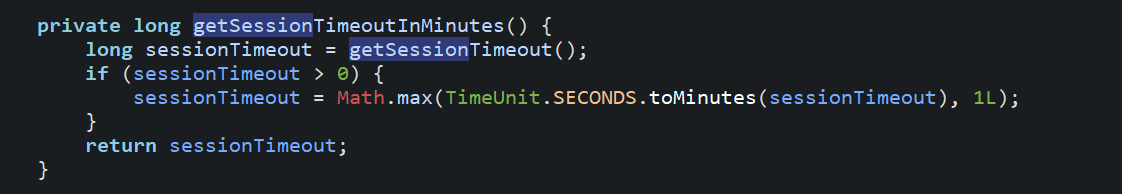
从之前的小结中，完成了多种方式的登录操作，但是，所有在spring security中的登录操作，最终的逻辑都是一样的，就是认证之后交给用户一个authenticationToken，表示用户的登录状态以及权限。而这一系列的操作全部是依赖于session完成的，在浏览器环境下这样的结构问题不大，但是在app中，并不能依赖默认的实现。

所以对于session管理非常的重要，而在session的管理中，主要关注这几个方面

1. session的超时处理

在spring框架中，要控制session超时异常非常简单，只需要在配置文件中添加

配置单位为s，最短超时时间为1分钟，默认情况下，如果不设置是30分钟。即，用户超过这个时间不进行操作，则当前session超时。这些操作在

中获取并进行管理。此时，超过1分钟登录，用户身份认证会失效。身份失效之后，访问到认证失败处理器，返回处理其结果，也就是说，session超时异常处理默认情况下和认证失败处理是相同的，而在实际业务流程中，这样的响应对于用户来说并不友好，需要将异常的原因返回给用户看。

http.apply(validateCodeSecurityConfig).// 应用验证码配置

                and().apply(smsCodeAuthenticationSecurityConfig)// 应用短信登录配置

                .and().apply(seehopeSocialSecurityConfigurer)// 应用第三方登录配置

                .and().rememberMe().tokenRepository(tokenRepository)

                .tokenValiditySeconds(securityProperties.getBrowser().getRememberMeSeconds())// 记住我功能配置，浏览器独有

                .userDetailsService(userDetailsService)// 配置登录认证逻辑

                .and().sessionManagement().invalidSessionStrategy(invalidSessionStrategy)// session失效之后跳转的路径，给用户较好的体验

                .maximumSessions(securityProperties.getBrowser().getSession().getMaxMumSessions())// 控制session并发，同时只允许1个用户访问

                .maxSessionsPreventsLogin(securityProperties.getBrowser().getSession().getMaxSessionPreventsLogin())// true则当并发数量打到最大的时候，后来的用户无法登录

                .expiredSessionStrategy(expiredSessionStrategy)// 并发session处理

                // 可以处理当并发访问发生时，服务器做什么响应，也可以记录请求IP,返回给用户

                .and().and().logout().logoutUrl(securityProperties.getBrowser().getLogOutUrl())// 和注销相关所有配置 注销之后，会1

                // 使当前Session失效，2会清楚与当前用户相关的rememberme记录，最后清空当前的securityContext

                // 最后，从定向到login page

                .logoutSuccessHandler(logoutSuccessHandler)// 退出成功之后的跳转

                .deleteCookies("JSESSIONID")// 退出成功之后，将session删除

                .and().csrf().disable();// 暂时关闭跨域请求拦截策略

其中，invalidSessionStrategy负责配置session超时异常处理，在browser中创建

/\*\*

\* <b>Description:抽象的session失效处理</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-browser

\* <br><b>PackageName:</b> net.seehope.security.browser.session

\* <br><b>ClassName:</b> AbstractSessionStrategyConfig

\* <br><b>Date:</b> 2019年3月6日 下午8:55:13

\*/

public abstract class AbstractSessionStrategyConfig {

    private static final Logger logger = LoggerFactory.getLogger(AbstractSessionStrategyConfig.class);

    /\*\*

     \* 跳转的url

     \*/

    private String destinationUrl;

    /\*\*

     \* 系统配置信息

     \*/

    private SecurityProperties securityPropertie;

    /\*\*

     \* 重定向策略

     \*/

    private RedirectStrategy redirectStrategy = new DefaultRedirectStrategy();

    /\*\*

     \* 跳转前是否创建新的session

     \*/

    private boolean createNewSession = true;

    private ObjectMapper objectMapper = new ObjectMapper();

    /\*\*

     \* @param invalidSessionUrl

     \* @param invalidSessionHtmlUrl

     \*/

    public AbstractSessionStrategyConfig(SecurityProperties securityPropertie) {

        String invalidSessionUrl = securityPropertie.getBrowser().getSession().getSessionInvalidUrl();

        Assert.isTrue(UrlUtils.isValidRedirectUrl(invalidSessionUrl), "url must start with '/' or with 'http(s)'");

        Assert.isTrue(StringUtils.endsWithIgnoreCase(invalidSessionUrl, ".html"), "url must end with '.html'");

        this.destinationUrl = invalidSessionUrl;

        this.securityPropertie = securityPropertie;

    }

    /\*

     \* session 失效处理方法，同时处理超时失效和并发失效等策略

     \*

     \* session 失效第一次触发请求，可以获取到触发该异常的request以及response对象

     \* (non-Javadoc)

     \*

     \* @see org.springframework.security.web.session.InvalidSessionStrategy#

     \* onInvalidSessionDetected(javax.servlet.http.HttpServletRequest,

     \* javax.servlet.http.HttpServletResponse)

     \*/

    protected void onSessionInvalid(HttpServletRequest request, HttpServletResponse response) throws IOException {

        logger.info("session失效");

        if (createNewSession) {

            request.getSession();

        }

        String sourceUrl = request.getRequestURI();

        String targetUrl;

        // 如果是html请求，即终端为浏览器

        if (StringUtils.endsWithIgnoreCase(sourceUrl, ".html")) {

            // 且，源路径不为登录或者注销路径

            if (StringUtils.equals(sourceUrl, securityPropertie.getBrowser().getLoginPage())

                    || StringUtils.equals(sourceUrl, securityPropertie.getBrowser().getLogOutUrl())) {

                // 那么目标路径为源路径

                targetUrl = sourceUrl;

            } else {

                // 那么目标路径为要跳转URL

                targetUrl = destinationUrl;

            }

            logger.info("跳转到:" + targetUrl);

            // 重定向到目标页面

            redirectStrategy.sendRedirect(request, response, targetUrl);

            // 如果终端为APP

        } else {

            Object result = buildResponseContent(request);

            response.setStatus(HttpStatus.UNAUTHORIZED.value());

            response.setContentType("application/json;charset=UTF-8");

            response.getWriter().write(objectMapper.writeValueAsString(result));

        }

    }

    /\*\*

     \* @param request

     \* @return

     \*/

    protected Object buildResponseContent(HttpServletRequest request) {

        String message = "session已失效";

        if (isConcurrency()) {

            message = message + "，有可能是并发登录导致的";

        }

        return new SimpleResponse(message);

    }

    /\*\*

     \* session失效是否是并发导致的

     \*

     \* @return

     \*/

    protected boolean isConcurrency() {

        return false;

    }

    /\*\*

     \* Determines whether a new session should be created before redirecting (to

     \* avoid possible looping issues where the same session ID is sent with the

     \* redirected request). Alternatively, ensure that the configured URL does

     \* not pass through the {@code SessionManagementFilter}.

     \*

     \* @param createNewSession

     \* defaults to {@code true}.

     \*/

    public void setCreateNewSession(boolean createNewSession) {

        this.createNewSession = createNewSession;

    }

}

之后，编写超时失效处理

/\*\*

\* <b>Description:

\*   当session 超时失效之后的处理

\* </b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-browser

\* <br><b>PackageName:</b> net.seehope.security.browser.session

\* <br><b>ClassName:</b> InvalidSessionStrategyConfig

\* <br><b>Date:</b> 2019年3月6日 下午8:52:39

\*/

public class InvalidSessionStrategyConfig extends AbstractSessionStrategyConfig implements InvalidSessionStrategy {

    public InvalidSessionStrategyConfig(SecurityProperties securityProperties) {

        super(securityProperties);

    }

    @Override

    public void onInvalidSessionDetected(HttpServletRequest request, HttpServletResponse response)

            throws IOException, ServletException {

        onSessionInvalid(request, response);

    }

}

1. session的并发处理

所谓session并发处理，即指假如说有两台机器，A,B,用户在A上登录了自己的账户，之后又在B中登陆了自己的账号，这时候，需不需要对用户进行处理，这时候，给用户设置一个最大链接数，当用户超过最大链接出之后，触发并发策略判断，此时，有两种策略，其一，B中登陆的账号顶替掉A中登陆的账号，即A中账号失效，B中账号启用，其二，限制用户在B中登录，即不允许登录。这两种策略在spring social中也已经实现完成了，用户只需要经过少量的配置即可启用该功能

.maximumSessions(securityProperties.getBrowser().getSession().getMaxMumSessions())// 控制session并发，同时只允许1个用户访问

.maxSessionsPreventsLogin(securityProperties.getBrowser().getSession().getMaxSessionPreventsLogin())// true则当并发数量打到最大的时候，后来的用户无法登录

.expiredSessionStrategy(expiredSessionStrategy)// 并发session处理

// 可以处理当并发访问发生时，服务器做什么响应，也可以记录请求IP,返回给用户

这三段配置，即，允许最大并发数，这里设置为1个，以及并发策略，这里设置为true，达到1条连接之后不允许用户在登录，并设置了并发异常处理机制。

/\*\*

\* <b>Description:

\* 当session并发策略出现异常时处理

\* </b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-browser

\* <br><b>PackageName:</b> net.seehope.security.browser.session

\* <br><b>ClassName:</b> ExpiredSessionStrategyConfig

\* <br><b>Date:</b> 2019年3月6日 下午7:55:05

\*/

public class ExpiredSessionStrategyConfig extends AbstractSessionStrategyConfig

        implements SessionInformationExpiredStrategy {

    public ExpiredSessionStrategyConfig(SecurityProperties securityPropertie) {

        super(securityPropertie);

    }

    /\* (non-Javadoc)

     \* @see org.springframework.security.web.session.SessionInformationExpiredStrategy#onExpiredSessionDetected(org.springframework.security.web.session.SessionInformationExpiredEvent)

     \*/

    @Override

    public void onExpiredSessionDetected(SessionInformationExpiredEvent event) throws IOException, ServletException {

        onSessionInvalid(event.getRequest(), event.getResponse());

    }

    /\* (non-Javadoc)

     \* @see com.imooc.security.browser.session.AbstractSessionStrategy#isConcurrency()

     \*/

    @Override

    protected boolean isConcurrency() {

        return true;

    }

}

最后，是该流程中出现的相关配置文件

package net.seehope.core.properties;

/\*\*

\* <b>Description:</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-core

\* <br><b>PackageName:</b> net.seehope.core.properties

\* <br><b>ClassName:</b> SessionProperties

\* <br><b>Date:</b> 2019年3月6日 下午8:46:25

\*/

public class SessionProperties {

    /\*

     \* 同一个用户最大session连接数

     \*/

    private Integer maxMumSessions = 1;

    /\*\*

     \* 当同一个用户session达到最大用户时，是后一个用户替换前一个用户，还是不允许再登录，为false则为后一个替换前一个，为true则为不允许登录

     \*/

    private Boolean maxSessionPreventsLogin = false;

    /\*\*

     \* session失效时跳转的地址

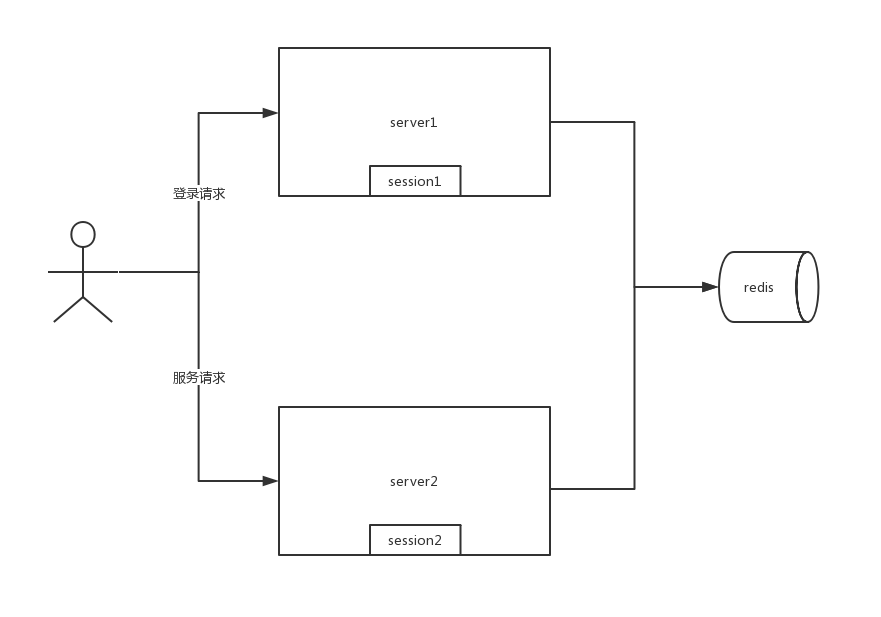
     \*/

    private String sessionInvalidUrl = "/demo-session-invalid.html";

}

### 4-8.集群Session管理

首先，描述一下业务场景，在大部分的公司业务生产中，服务都不是由一台服务器管理的，如果在负载均衡服务器nginx中没有对session做处理，那么经常会遇到这种情况，假如说公司有两台服务器A,B，用户第一次在A服务器中登录，session被记录了，第二次访问应用，被nginx分配到了B服务器进行业务处理，那么此时，由于session是跟随服务器内存的，此时，用户在B服务器中没有记录，需要重新登录。那么在这种情况下，如何进行session集群管理呢，很简单，将session从内存中抽取出来，存放到独立的存储中redis，然后所有的应用都访问该redis服务即可。这样每个服务器获取到的都是同一个数据对象了。’



要实现上述功能，在spring中是非常简单的，因为spring已经提供有一个专门的项目来管理这个功能了，就是在browser工程中独立引入的springsession工程，开发者只需要告诉spirng框架，存储类型是什么，以及redis服务的端口以及用户名密码即可。

/\*

\* Copyright 2012-2016 the original author or authors.

\*

\* Licensed under the Apache License, Version 2.0 (the "License");

\* you may not use this file except in compliance with the License.

\* You may obtain a copy of the License at

\*

\* http://www.apache.org/licenses/LICENSE-2.0

\*

\* Unless required by applicable law or agreed to in writing, software

\* distributed under the License is distributed on an "AS IS" BASIS,

\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

\* See the License for the specific language governing permissions and

\* limitations under the License.

\*/

package org.springframework.boot.autoconfigure.session;

/\*\*

\* Supported Spring Session data store types.

\*

\* @author Tommy Ludwig

\* @author Eddú Meléndez

\* @since 1.4.0

\*/

public enum StoreType {

    /\*\*

     \* Redis backed sessions.

     \*/

    REDIS,

    /\*\*

     \* Mongo backed sessions.

     \*/

    MONGO,

    /\*\*

     \* JDBC backed sessions.

     \*/

    JDBC,

    /\*\*

     \* Hazelcast backed sessions.

     \*/

    HAZELCAST,

    /\*\*

     \* Simple in-memory map of sessions.

     \*/

    HASH\_MAP,

    /\*\*

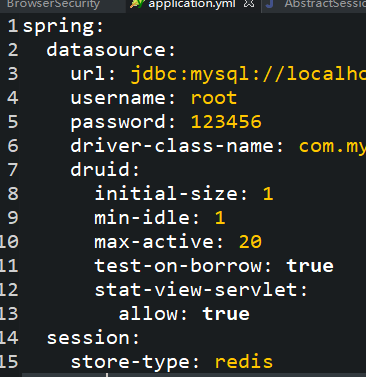
     \* No session data-store.

     \*/

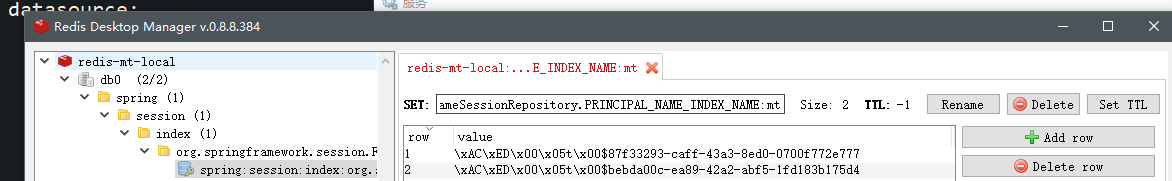
    NONE;

}

当然，在spring中，session的存储方式不光是redis，市场上常见的nosql都有实现，之后，用户可以在配置文件中选择该实现方式是什么，在本节的内容中，我们使用redis作为session存储服务，因为session本身具有超时时间，同时，在springsecurity中，所有的操作都要依赖于session即session存取非常的频繁，这种种特性都和redis非常相符，用redis来实现session存储非常的方便高效。

在配置文件中添加，这里大小写都可以。

之后，spring的session会默认存储到redis中，配置好之后，再次登录，发生异常，会提示验证码实现类没有序列化，在这里，如果需要将对象存储到磁盘中，对象需要实现序列化接口，而这里是图片验证码没有实现序列化接口，而bufferedimage本身不可更改，那么在重构一下代码逻辑，即validateCodeProcessor中的验证码存储逻辑，不将整个validatecode存入session，而将验证码code存到session中，其实，也没有必要将整个图片放入session，至此，重构完成。会发现spring像redis中添加了记录，在本例中没有加入redis的配置，那么spring会默认加入到本地的redis服务中。



那么，最后还有一个问题，在使用了集群配置之后，在上小结中配置的超时异常，并发策略在spring中是否还生效。可以尝试一下，最后，可以发现这些功能都是起作用的。

### 4-9.退出登录

SpringSecurity中，退出登录页非常简单，首先，配置好退出登录请求地址，之后，配置退出成功处理器，最后，在退出之后，将session中的用户信息清空即可。Spring 默认的退出登录处理，首先，使当前登录失效，然后，清空与当前用户相关的remember-me记录，之后清空当前的securityContext，最后，重定向到登录页面，这里配置的logouturl值得就是退出登录触发请求，退出成功之后由logoutSuccessHandler处理。

.and().and().logout().logoutUrl(securityProperties.getBrowser().getLogOutUrl())// 和注销相关所有配置 注销之后，会1

// 使当前Session失效，2会清楚与当前用户相关的rememberme记录，最后清空当前的securityContext

// 最后，重定向到login page

.logoutSuccessHandler(logoutSuccessHandler)// 退出成功之后的跳转

.deleteCookies("JSESSIONID")// 退出成功之后，将session删除

之后，配置logoutsuccesshandler

/\*\*

\* <b>Description:</b><br>

\* @author <a href="" target="\_blank">Monty</a>

\* @version 1.0

\* @Note

\* <b>ProjectName:</b> seehope-security-browser

\* <br><b>PackageName:</b> net.seehope.security.browser.logout

\* <br><b>ClassName:</b> LogoutSuccessHandler

\* <br><b>Date:</b> 2019年3月7日 下午8:40:34

\*/

public class LogoutSuccessHandlerConfig implements LogoutSuccessHandler {

    private static final Logger log = LoggerFactory.getLogger(LogoutSuccessHandlerConfig.class);

    private SecurityProperties properties;

    @Autowired

    private ObjectMapper mapper;

    public LogoutSuccessHandlerConfig(SecurityProperties properties) {

        this.properties = properties;

    }

    @Override

    public void onLogoutSuccess(HttpServletRequest request, HttpServletResponse response, Authentication authentication)

            throws IOException, ServletException {

        // TODO Auto-generated method stub

        log.info("触发推出成功逻辑");

        if (StringUtils.isNotBlank(properties.getBrowser().getLogOutSuccessUrl())) {

            response.sendRedirect(properties.getBrowser().getLogOutSuccessUrl());

        } else {

            response.setContentType("application/json;charset=utf-8");

            response.getWriter().write(mapper.writeValueAsString(new SimpleResponse("退出登录成功")));

        }

    }

    public SecurityProperties getProperties() {

        return properties;

    }

    public void setProperties(SecurityProperties properties) {

        this.properties = properties;

    }

}

在成功触发了退出逻辑之后，判断用户是否有配置logouturl，如果有，则重定向到退出页面，如果没有，则返回json形式数据。

之后。将退出相关类配置到spring容器中

@Bean

@ConditionalOnMissingBean(LogoutSuccessHandlerConfig.class)

public LogoutSuccessHandlerConfig logoutSuccessHandlerConfig() {

    return new LogoutSuccessHandlerConfig(properties);

}

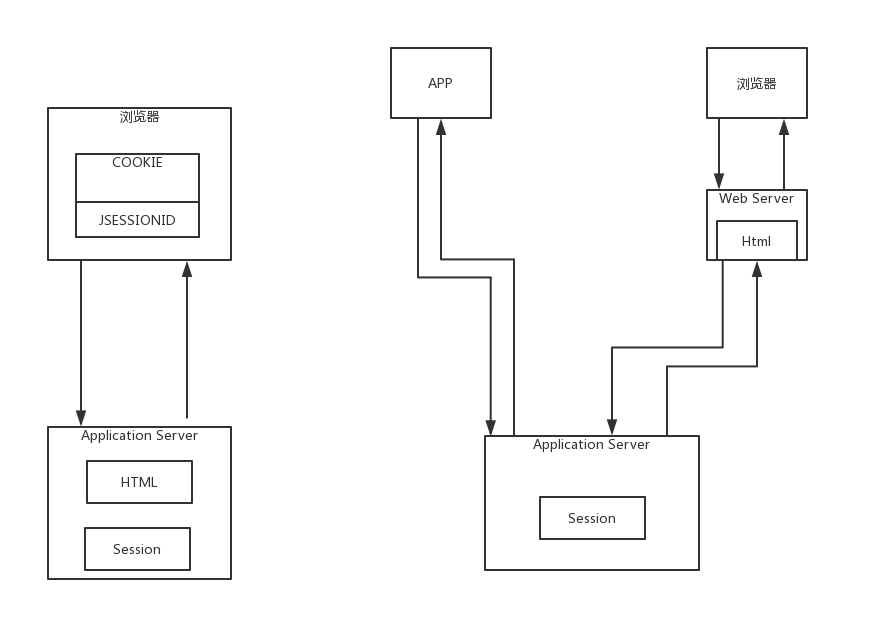
登录逻辑开发完成，最后，附上第4张完成之后的代码。



## 使用Spring Security Oauth开发APP认证框架

### 5-1.Spirng Security OAuth简介

在之前的登录方式中，当用户登陆之后的信息，都是存在session中的，也就是说，所有的检测都是依赖于session。





### 5-2.实现标准的OAuth服务提供商

### 5-3.Spirng Security OAuth 核心源码解析

### 5-4.重构用户名密码登录

### 5-5.重构短信登录

### 5-6.重构社交登录

### 5-7.重构注册逻辑

### 5-8.令牌配置

### 5-9.使用JWT替换默认令牌

### 5-10.基于JWT实现SSO单点登录

## 使用Spirng Security 控制授权

### 6-1.Spring Security 授权简介

### 6-2.Spinrg Security 源码解析

### 6-3.权限表达式

### 6-4.基于数据库RBAC数据模型控制权限

## 总结