1. 导入maven坐标

<parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.0.1.RELEASE</version>  
 <relativePath />  
</parent>  
  
<properties>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  
 <java.version>1.8</java.version>  
 <spring-cloud.version>Finchley.SR1</spring-cloud.version>  
</properties>  
  
<dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-dependencies</artifactId>  
 <version>${spring-cloud.version}</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 </dependencies>  
</dependencyManagement>

1. 建立Eureka服务

建立一个独立的子模块

取名为eureka

导入坐标，导入eureka的坐标，并且导入Ribbon的坐标，Ribbon是负载均衡的,导入spring的安全框架坐标，配置eureka对应的访问权限

<dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-netflix-eureka-server</artifactId>  
 </dependency>

<!--spring 的安全中心-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>  
</dependency>  
<dependency>  
 <groupId>com.squareup.okhttp3</groupId>  
 <artifactId>okhttp</artifactId>  
</dependency>

</dependencies>

并且写入以下代码：

package com.sc;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;  
  
@SpringBootApplication  
@EnableEurekaServer  
public class EurekaApplication {  
 public static void main(String[] args) {  
 SpringApplication.*run*(EurekaApplication.class, args);  
 }  
}

新版本的eureka要求增加一个对应的类在eureka服务中，我们起名为WebSecurityConfig，类的信息如下：

package com.sceureka;  
  
import org.springframework.context.annotation.Configuration;  
import org.springframework.security.config.annotation.web.builders.HttpSecurity;  
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  
  
@EnableWebSecurity  
@Configuration  
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {  
  
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 http.csrf().disable();  
 super.configure(http);  
 }  
}

配置完成后，需要在yml文件中加入对应的配置，如下：

*#Eureka服务注册中心  
#作为一个实例也包括定期心跳到注册表（通过客户端的serviceUrl），  
#默认持续时间为30秒。在实例，服务器和客户端在其本地缓存中都具有相同的元数据（因此可能需要3个心跳）之前，  
#客户端才能发现服务。您可以使用eureka.instance.leaseRenewalIntervalInSeconds更改期限，这将加快客户端连接到其他服务的过程。  
#在生产中，最好坚持使用默认值，因为服务器内部有一些计算可以对租赁更新期进行假设。  
#  
#  
#***server:  
 port:** 9000  
**spring:  
 application:  
 name:** sc-euruka  
 *# 配置eureka的安全中心* **security:  
 user:  
 name:** 123qwe  
 **password:** 123qwe  
*# Eureka的默认注册地址***eureka:  
 client:  
 serviceUrl:  
 defaultZone:** http://${**spring.security.user.name**}:${**spring.security.user.password**}@${**eureka.instance.hostname**}:${**server.port**}/eureka/  
*# http://name:pwd@localhost:9001/eureka  
 # 禁止eureka向自己注册，eureka会自己向自己注册服务，如果禁止那么就是单机版* **register-with-eureka: false  
 fetch-registry: false***# 初始化Eureka名称，如果不初始化名称，那么就以application的name为名称* **instance:  
 hostname:** localhost  
 **instanceId:** ${**spring.application.name**}  
 **server:***# eureka 关闭自我保护机制，如果是网络原因，那么eureka可以有自己的缓存能够维持继续访问* **enable-self-preservation: false**

解释：

Server.port:应用启动的端口号

Spring.application.name:应用名称

Spring.security.user.name:登入或注册eureka的用户名称

Spring.security.user.password:登入或注册eureka的密码

Eureka.client.serviceUrl.defaultZone:注册的eureka地址

Register-with-eureka:是否注册到eureka上

Fetch.registery:是否从eureka上获取注册客户端的信息

Enable.self.preservation:是否启动自我保护机制

Ip-address:如果配置地址，那么注册到的eureka中心就是对应的ipi了

Prefer-ip-address:注册到eureka上的地址是否展示

1. 配置完成eureka后，我们写入两个客户端

3.1第一个客户端为order模块

导入pom.xml文件，pom文件的坐标有三个，一个是eureka的客户端，一个是ribbon的负载均衡，一个是hystrix的熔断器

<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-openfeign</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>com.squareup.okhttp3</groupId>  
 <artifactId>okhttp</artifactId>  
 </dependency>  
</dependencies>

新建一个类为OrderApplication.java

其中的主类为信息为：

package com.sc;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;  
import org.springframework.cloud.netflix.hystrix.EnableHystrix;  
import org.springframework.cloud.openfeign.EnableFeignClients;  
  
@SpringBootApplication  
@EnableDiscoveryClient  
@EnableHystrix  
@EnableFeignClients  
public class OrderApplication {  
 public static void main(String[] args) {  
 SpringApplication.*run*(OrderApplication.class, args);  
 }  
}

注解：一个是注册到eureka上，一个是启动熔断器，一个是使用feign客户端。

可进行重试和相应的次数：配置如下：

package com.sc;  
  
import feign.Retryer;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class FeignConfig {  
 @Bean  
 public Retryer feignRetryer () {  
 return new Retryer.Default(100, 1000, 2);  
 }  
}

new Retryer.Default(100, 1000, 2); 100是超时的毫秒数，1000最大毫秒数，2尝试2次

使用方式，在使用注解@FeignClient(value=””, configureation=FeignConfig.class)即可

同时配置RestTemplate的注解，在主类对应的方法下，提供如下配置，并且在pom.xml中导入对应的okhttp远程访问，对应的restTemplate的访问就可以进行使用okhttp进行访问了。

所以配置如下（如果需要的话）:

Pom.xml文件配置

<dependency>  
 <groupId>com.squareup.okhttp3</groupId>  
 <artifactId>okhttp</artifactId>  
</dependency>

对应的主类加入：

@Bean  
@LoadBalanced

@Primary  
RestTemplate restTemplate () {  
 return new RestTemplate(new OkHttp3ClientHttpRequestFactory());  
}

1. 以同样的方式建立一个商品端：
2. 我们在双方的项目下，建立一个controller提供对外的访问。

package com.sc.controller;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.cloud.client.discovery.DiscoveryClient;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class OrderController {  
  
 @Autowired  
 DiscoveryClient discoveryClient;  
 @Autowired  
 private OrderInterface orderInterface;  
  
 @GetMapping("/order/{id}")  
 public String goods (@PathVariable("id") String id) {  
  
 return orderInterface.goodsById(id);  
 }  
  
}

以上是order的controller

接下来是feign客户端的接口，也就是order中调用goods中的对应的类

package com.sc.controller;  
  
import com.sc.FeignConfig;  
import org.springframework.cloud.openfeign.FeignClient;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
  
@FeignClient(value = "goodsApplication", configuration = FeignConfig.class)  
public interface OrderInterface {  
  
 @GetMapping("/goods/{id}")  
 String goodsById (@PathVariable("id") String id);  
  
}

goods中有一个相当于springmvc的映射，直接映射对应即可。

1. RestTemplate结合Hystrix
2. Feign结合Hystrix，feign结合它的话，首先在yml文件中进行配置，配置如下：

当我们调用失败的时候，进行对应的调用错误的方法，最开始要加如对应的配置文件信息，为：

**feign:  
 hystrix:  
 enabled:** true

首先建立一个实现feign接口调用其他客户端的实现类，具体的话，如下：

首先feign调用的接口如下：

package com.sc.controller;  
  
import com.sc.FeignConfig;  
import org.springframework.cloud.openfeign.FeignClient;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
  
@FeignClient(value = "goodsApplication", configuration = FeignConfig.class, fallback = ErrorOrder.class)  
public interface OrderInterface {  
  
 @GetMapping("/goods/{id}")  
 String goodsById (@PathVariable("id") String id);  
  
}

在此处，fallback是指的回调的类，那么我们就要建立对应的类的信息，如下;

package com.sc.controller;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class ErrorOrder implements OrderInterface {  
 @Override  
 public String goodsById(String id) {  
 return "error";  
 }  
}

此类要放到容器中，使用对应的注解，同时要实现feign调用的客户端的接口即可。然后再实现对应的方法的内容就OK了。当我们调用失败的时候，我们就可以对应的调用此类的对应的方法了。

Spring boot 结合swagger2

首先引入pom.xml文件，入下内容：

<dependency><!--添加Swagger依赖 -->  
 <groupId>io.springfox</groupId>  
 <artifactId>springfox-swagger2</artifactId>  
 <version>2.7.0</version>  
</dependency>  
<dependency><!--添加Swagger-UI依赖 -->  
 <groupId>io.springfox</groupId>  
 <artifactId>springfox-swagger-ui</artifactId>  
 <version>2.7.0</version>  
</dependency>

添加完成后，需要配置swagger2的配置类信息，入下：

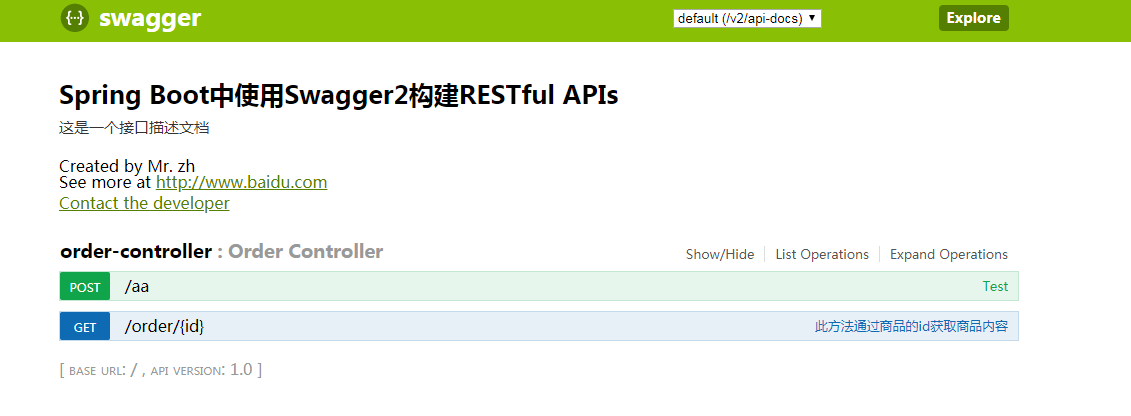
package com.sc;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import springfox.documentation.builders.ApiInfoBuilder;  
import springfox.documentation.builders.PathSelectors;  
import springfox.documentation.builders.RequestHandlerSelectors;  
import springfox.documentation.service.ApiInfo;  
import springfox.documentation.service.Contact;  
import springfox.documentation.spi.DocumentationType;  
import springfox.documentation.spring.web.plugins.Docket;  
import springfox.documentation.swagger2.annotations.EnableSwagger2;  
  
@Configuration  
@EnableSwagger2  
public class Swagger2 {  
  
 @Bean  
 public Docket createRestApi() {  
 return new Docket(DocumentationType.*SWAGGER\_2*)  
 .apiInfo(apiInfo())  
 .select()  
 .apis(RequestHandlerSelectors.*basePackage*("com.sc.controller"))  
 .paths(PathSelectors.*any*())  
 .build();  
 }  
  
 private ApiInfo apiInfo() {

return new ApiInfoBuilder()  
 .title("Spring Boot中使用Swagger2构建RESTful APIs")  
 .description("这是一个接口描述文档")  
 .version("1.0")  
 .contact(new Contact("Mr. zh", "http://www.baidu.com", "1220041602"))  
 .build();  
 }  
  
}

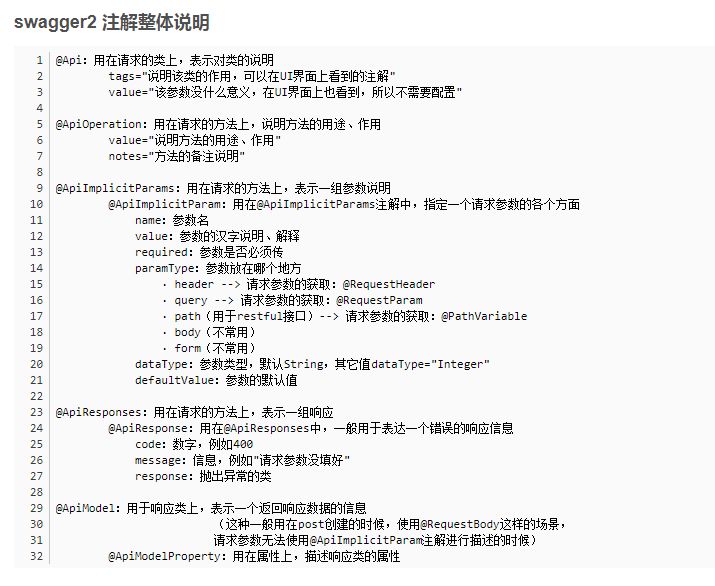
其中注意：在createRestApi中，apis是扫描包的类

ApiInfo是创建页面ui的信息类，其中以此为标题，描述，版本号链接信息等。所以创建出来的接口页面入下：

访问网址：http://localhost:9001/swagger-ui.html#



完成后，描述对应的注解信息，入下：







Spring security

首先引入pom.xml文件，如下:

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

Application.yml文件的配置：

**server:  
 port:** 9999  
**spring:  
 application:  
 name:** securityApplication  
 **thymeleaf:  
 mode:** HTML5  
 **cache:** false

建立一个普通的映射类，如下：

package com.sc.controller;  
  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestMethod;  
  
*/\*\*  
 \** ***@Author*** *ZhPJ  
 \** ***@Date*** *2018/11/6 000610:24  
 \** ***@Version*** *1.0  
 \** ***@Description:*** *\*/*@Controller  
public class SecurityController {  
 @RequestMapping(path = "/security", method = RequestMethod.*GET*)  
 public String security() {  
 return "login";  
 }  
}

建立一个简单的登录页面，如下：

此spring的安全中心是结合spring boot的。

其次 就是 配置spring security 的配置文件，SecurityConfig.java，具体如下：

package com.sc;  
  
import org.springframework.context.annotation.Configuration;  
import org.springframework.security.config.annotation.web.builders.HttpSecurity;  
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  
  
@Configuration  
public class SecurityCore extends WebSecurityConfigurerAdapter {  
  
 @Override  
 protected void configure (HttpSecurity httpSecurity) throws Exception {  
 httpSecurity.formLogin()  
 .and()  
 .authorizeRequests()  
 .anyRequest()  
 .authenticated();  
 }  
}

在这里我们能看到，继承了WebSecurityConfigurerAdapter并重写了configure方法，

看其API，是http的，方法说明，

formLogin是有一个独立页面进行登陆的，

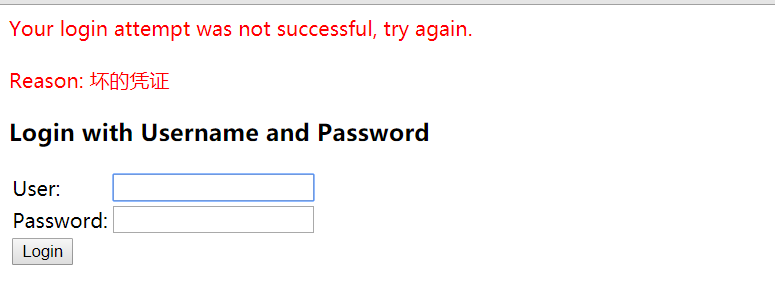
and()并且

authorizeRequests()认证请求，

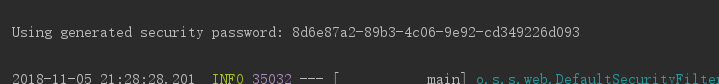
anyRequest()所有的请求，

authenticated()所有请求都被认证才可以

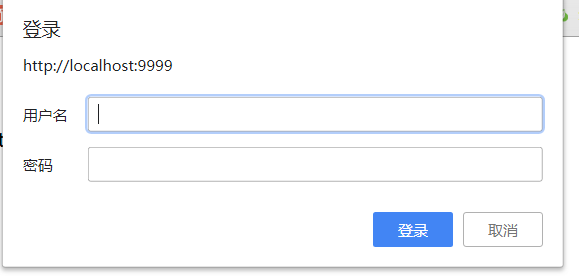
formLogin登陆的显示结果是这样子的：



会在控制台生成一个对应的密码，用户名为user，密码在控制台如下：



httpBasic () 继承登陆,显示如下



会有一个弹出框，用户名为user，密码在控制台和上面的一样。

自定义密码配置

首先要定义一个类并放入到spring容器中（和上面的一样，只不过密码不是由控制台生成，而是自定义了密码），然后定义一个类，继承UserDetailsService并将这个类放到spring容器中，重写UserDetails方法，具体实现如下：

package com.sc;  
  
import org.springframework.security.core.authority.AuthorityUtils;  
import org.springframework.security.core.userdetails.User;  
import org.springframework.security.core.userdetails.UserDetails;  
import org.springframework.security.core.userdetails.UserDetailsService;  
import org.springframework.security.core.userdetails.UsernameNotFoundException;  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
import org.springframework.stereotype.Component;  
  
@Component  
public class UserDetailServiceConfig implements UserDetailsService {  
 @Override  
 public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
 return new User(username, "{noop}123", AuthorityUtils.*commaSeparatedStringToAuthorityList*("admin"));  
 }  
}

在这里new User ()

Username：是前台传过来的值，拿到username，去数据库中获取password并且进行匹配，然后获取对应的权限admin即可。由于spring boot整合了spring security5.0所以如下图：



使用spring security框架实现密码加密，首先注入一种加密的方式在容器中，如下:

// 注入一个密码加密的一种方式  
@Bean  
public PasswordEncoder passwordEncoder () {  
 return new BCryptPasswordEncoder();  
}

其次就是在加密或者判等的地方进行解密，如下:

package com.sc;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.security.core.authority.AuthorityUtils;  
import org.springframework.security.core.userdetails.User;  
import org.springframework.security.core.userdetails.UserDetails;  
import org.springframework.security.core.userdetails.UserDetailsService;  
import org.springframework.security.core.userdetails.UsernameNotFoundException;  
import org.springframework.security.crypto.password.PasswordEncoder;  
import org.springframework.stereotype.Component;  
  
@Component  
public class UserDetailServiceConfig implements UserDetailsService {  
 @Autowired  
 private PasswordEncoder passwordEncoder;  
 @Override  
 public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
 System.*out*.println(passwordEncoder.encode("123"));  
 return new User(username, passwordEncoder.encode("123"), AuthorityUtils.*commaSeparatedStringToAuthorityList*("admin"));  
 }  
}

即可。