Online Bank Design Scheme

# 业务选择

网上银行设计业务体系较为复杂，比较适合使用分布式系统，目前系统设计三块业务。

1. 查询、转账：转账可为跨行转账
2. 借贷服务：借贷涉及模块、风控（校验，调用接口查看征信信息，或者本行校验，是否有未完成借贷服务）
3. 理财产品：包括行内的以及代理的。

目前仅仅涉及三块内容。

# 技术选型

分布式系统，微服务架构，选择spring-cloud。包括服务治理、熔断、声明式调用、restful调用、消息驱动、网关、鉴权中心、分布式服务跟踪，分布式配置中心。

# 系统架构设计

系统：

OnlineBank-geteway:api网关 spring-cloud-zuul

OnlineBank-sso：单点登录 –前期、只做seesion管理、后期、做角色权限管理。

-----OnlineBank-exceptionHandler:统一错误处理，消息驱动

-----OnlineBank-router:路由yue

OnlineBank-customer-search:查询余额（接口调用频繁）

OnlineBank- customer-handler-web:用户操作、转账、借贷、理财、支付

onlineBank-manage-transfer:转账后台服务-区分跨行和本行（本行直接操作数据库，跨行的调用其他银行提供的接口）

onlineBank-manage-outward-pay:转账接口，这个以消息队列的形式接收跨行转账的消息

转账逻辑：转账操作-🡪后台校验是否本行-🡪本行直接行内进行账户加减操作—>非本行，调用其他银行接口-🡪其他银行进行账户加操作，并且记录下操作银行，操作账户，操作金额—>返回给本行—>操作成功，则进行账户减操作，并且发送一条消息给outward-pay系统—>outward-pay系统将消息归类，统一进行转账操作（按不同的银行进行转账），并记录表操作记录

OnlineBank-manage-loans:借贷后台服务，用户操作调用该系统接口（1、接口返回借贷资质，根据风控中心返回的数据来判断该用户的额度）

OnlineBank-manage-products:理财产品，行内

OnlineBank-manage-out-products:理财产品外部代理（接口调用restTemplate,有自己的表，方便生成季度报表）

理财产品逻辑：客户选择理财产品—>后台评测理财产品的来源方—>本行

onlineBank-manage-message-push:消息推送中心(websockt、feign、netty)

onlineBank-manage-message-web:消息中心，编辑消息，统一在消息推送推送，定时消息和即时消息

onlineBank-manage-job:定时任务中心(zookeeper选举)

onlineBank-manage-pay:支付中心，小额支付中心（类似于支付宝的付款）

onlineBank-manage-risk:风控中心（借贷调用风控中心-风控返回结果-根据结果判断是否放款操作-是->放款）

onlineBank-manage-loans-pay:放款中心，借贷后台

借贷操作：用户->资质（先查表，看用户是否已经经过资质评测，如果没有，则进行风控资质评测，和后面的风控调用同一接口，验证：1、当前用户是否有其他未还贷款。2、当前用户是否有过逾期记录）->借款操作->风控（验证：1、当前用户是否有其他未还贷款。2、当前用户是否有过逾期记录。）-）是否放款->评测通过则调用放贷中心

onlineBank-manage-common:公共类

onlineBank-restful-commom-web:公共框架中心，包括请求进入的logger日志，操作日志记录操作

onlineBank-restful-common-service

技术汇总：

Springboot:1.5.9

Springcloud—eurake、feign、ribben、分布式配置中心、stream、zuul、熔断

Zookeeper:选举、分布式锁

Redis:分布式锁：redisTeplate、springboot的redis仓库

Mongo:报表、理财产品报表

Netty:sso和message

Websocket:message消息推送

Shiro—security—smart:三个

Mybatis

rabbitMQ:

kafka:

fastjson:

Quartz：定时任务

数据连接池：druid

日志管理：slf4j、logback按时间拆分日志

前端：vue.js（简单的操作）

# 技术补充

## swaager使用、对外暴露接

### 1、依赖（可在maven仓库查找最新依赖）

<**dependency**>  
 <**groupId**>io.springfox</**groupId**>  
 <**artifactId**>springfox-swagger2</**artifactId**>  
 <**version**>2.8.0</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>io.springfox</**groupId**>  
 <**artifactId**>springfox-swagger-ui</**artifactId**>  
 <**version**>2.8.0</**version**>  
</**dependency**>

### 配置类

@Configuration  
@EnableSwagger2  
**public class** Swagger2 {  
  
 @Bean  
 **public** Docket createRestApi() {  
 **return new** Docket(DocumentationType.***SWAGGER\_2***)  
 .apiInfo(apiInfo())  
 .select()  
 .apis(RequestHandlerSelectors.*basePackage*(**"com.didispace.web"**)) –扫描的controller包  
 .paths(PathSelectors.*any*())  
 .build();  
 }  
 **private** ApiInfo apiInfo() {  
 **return new** ApiInfoBuilder()  
 .title(**"Spring Boot中使用Swagger2构建RESTful APIs"**)  
 .description(**"更多Spring Boot相关文章请关注：http://blog.didispace.com/"**)  
 .termsOfServiceUrl(**"http://blog.didispace.com/"**)  
 .contact(**"程序猿DD"**)  
 .version(**"1.0"**)  
 .build();  
 }  
}

### 常用注解

@Api：修饰整个类，描述Controller的作用

@ApiOperation：描述一个类的一个方法，或者说一个接口

@ApiParam：单个参数描述

@ApiModel：用对象来接收参数

@ApiProperty：用对象接收参数时，描述对象的一个字段

@ApiResponse：HTTP响应其中1个描述

@ApiResponses：HTTP响应整体描述

@ApiIgnore：使用该注解忽略这个API

@ApiError ：发生错误返回的信息

@ApiParamImplicitL：一个请求参数

@ApiParamsImplicit 多个请求参数

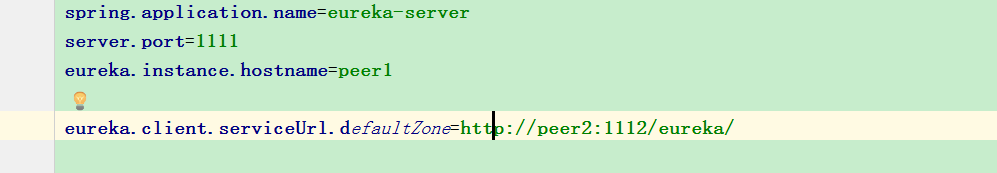
@ApiImplicitParams 多个参数

@ApiImplicitParams({  
 @ApiImplicitParam(name = **"id"**, value = **"用户ID"**, required = **true**, dataType = **"Long"**),  
 @ApiImplicitParam(name = **"user"**, value = **"用户详细实体user"**, required = **true**, dataType = **"User"**)  
})

## 2、spring-cloud

### 1、eurake

#### 1、 高可用架构：多个配置文件，多个服务互相注册



#### 2、命令行启动时启动命令后添加server.port=??可指定启动后的port

#### 3、服务心跳检测是失效时间可在配置文件控制

**eureka.instance.lease-renewal-interval-in-seconds**=30 默认服务心跳检测时间30秒  
**eureka.instance.lease-expiration-duration-in-seconds**=90 默认服务失效时间90秒

#### 4、维护一份制度的服务清单来返回给客户端，默认30秒更新一次

**eureka.client.registry-fetch-interval-seconds**=30

#### 5、自我保护机制、开启之后将不会即使剔除掉心跳检测失败的机器，生产会因为网络波动产生，单机时为了及时剔除机器，可设置关闭自我保护机制

**eureka.server.enable-self-preservation**=true 默认开启

#### 6、默认将自己注册到eureka服务，并且从服务器定时读取服务清单（客户端负载均衡）

**eureka.client.register-with-eureka**=**false 注册  
eureka.client.fetch-registry**=**false 读取注册**

#### 7、服务注册类配置

org.springframework.cloud.netflix.eureka.EurekaClientConfigBean；类，以eureka.client开头

#### 8、服务实例类配置

org.springframework.cloud.netflix.eureka.EurekaInstanceConfigBean：以eureka.instance开头

元数据

实例名

端点

### 2、Ribbon

#### 1、Ribbon将面向服务的rest模板请求自动转换成客户端负载均衡的服务调用

RestTemplate可以独自使用，没有ribbon的情况下将不具备负载均衡的功能

#### 2、在RestTemplate的bean添加 @LoadBalanced即可开启

@Bean  
@LoadBalanced  
**public** RestTemplate restTemplate() {  
 **return new** RestTemplate();  
}

#### 4、更改负载均衡策略时不能被ComponentScan扫描到（排除）

@ComponentScan(excludeFilters = { @ComponentScan.Filter(type = FilterType.***ANNOTATION***, value = ExcludeFromComponentScan.**class**) })

@Configuration  
@ExcludeFromComponentScan（自定义注解）  
**public class** TestConfiguration {  
 *// @Autowired  
 // IClientConfig config;* @Bean  
 **public** IRule ribbonRule() {  
 **return new** RandomRule();  
 }  
}

#### 5、利用属性自定义配置更改负载均衡策略

**microservice-provider-user:  
 ribbon:  
 NFLoadBalancerRuleClassName:** com.netflix.loadbalancer.RandomRule

#### ribbon可定义重试策略

**ribbon.ConnectTimeout**=**250  
ribbon.ReadTimeout**=**1000  
ribbon.OkToRetryOnAllOperations**=**true  
ribbon.MaxAutoRetriesNextServer**=**2  
ribbon.MaxAutoRetries**=**0**

### 3、Hystrix (https://www.cnblogs.com/yepei/p/7169127.html)

#### 1、依赖来自于spring-cloud

<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-hystrix</**artifactId**>  
</**dependency**>

#### 2、@EnableCircuitBreaker开启断路器功能

@springCloudApplication标准应用包含服务发现和断路器

#### 3、@HystrixCommand指定回调方法

@HystrixCommand(fallbackMethod = **"helloFallback"**, commandKey = **"helloKey"**)

#### 4、Hystrix默认超时时间为2000毫秒

#### 5、自定义HystrixCommand

**import** com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;  
**import** com.netflix.hystrix.contrib.javanica.annotation.ObservableExecutionMode;  
**import** com.netflix.hystrix.contrib.javanica.command.AsyncResult;  
**import** org.springframework.http.ResponseEntity;  
**import** org.springframework.web.client.RestTemplate;  
**import** rx.Observable;  
**import** rx.Subscriber;  
**import** javax.swing.plaf.metal.MetalTheme;  
**import** java.util.concurrent.Future;  
*/\*\*  
 \** ***@author*** *qinhaohao  
 \** ***@create*** *2018-01-18 15:21  
 \*\*/***public class** UserCommand {  
 @CacheResult(cacheKeyMethod = **"getBasic"**)  
 @HystrixCommand(fallbackMethod = **"defaultString"**,groupKey = **"StringGroup"**,threadPoolKey = **"getStringThread"**)  
 **public** String getBasic(){  
 **return "同步"**;  
 }  
 */\*\*  
 \** ***@Author:*** *qinhaohao  
 \** ***@Param*** *: No such property: code for class: Script1  
 \** ***@Description:*** *\** ***@Return:*** *//异步  
 \** ***@Date*** *:2018/1/18 15:53  
 \*/*

@CacheRemove(commandKey = **"getBasic"**)@HystrixCommand(fallbackMethod = **"defaultString"**)  
 **public** Future<String> getBasic1(){  
 **return new** AsyncResult<String>() {  
 @Override  
 **public** String invoke() {  
 **return "异步"**;  
 }  
 };  
 };  
 */\*\*  
 \** ***@Author:*** *qinhaohao  
 \** ***@Param*** *: No such property: code for class: Script1  
 \** ***@Description:*** *Observe()方式  
 \** ***@Return:*** *\** ***@Date*** *:2018/1/18 15:53  
 \*/* @HystrixCommand(observableExecutionMode = ObservableExecutionMode.***EAGER***,fallbackMethod = **"defaultString"**)  
 **public** Observable<String> getBasic2(){  
 **return** Observable.*create*(**new** Observable.OnSubscribe<String>() {  
 @Override  
 **public void** call(Subscriber<? **super** String> subscriber) {  
 **try** {  
 **if**(!subscriber.isUnsubscribed()){  
 String body = **"observe"**;  
 subscriber.onNext(body);  
 subscriber.onCompleted();  
 }  
 }**catch** (Exception e){  
 subscriber.onError(e);  
 }  
 }  
 });  
 }  
 */\*\*  
 \** ***@Author:*** *qinhaohao  
 \** ***@Param*** *: No such property: code for class: Script1  
 \** ***@Description:*** *observable方式  
 \** ***@Return:*** *\** ***@Date*** *:2018/1/18 15:52  
 \*/* @HystrixCommand(observableExecutionMode = ObservableExecutionMode.***LAZY***,fallbackMethod = **"defaultString"**)  
 **public** Observable<String> getBasic3(){  
  
 **return** Observable.*create*(**new** Observable.OnSubscribe<String>() {  
 @Override  
 **public void** call(Subscriber<? **super** String> subscriber) {  
 **try** {  
 **if**(!subscriber.isUnsubscribed()){  
 String body = **"toObserve"**;  
 subscriber.onNext(body);  
 subscriber.onCompleted();  
 }  
 }**catch** (Exception e){  
 subscriber.onError(e);  
 }  
  
 }  
 });  
 }  
  
 */\*\*  
 \* fallBack方法定义无特殊要求，可以级联进行降级  
 \** ***@return*** *\*/* **private** String defaultString(Throwable e){  
 System.***out***.println(e);  
 **return "fallbackMethod"**;  
}

}

测试类

**import** org.springframework.web.client.RestTemplate;  
**import** rx.Observable;  
**import** rx.Observer;  
**import** rx.functions.Action1;  
**import** java.util.concurrent.TimeUnit;  
*/\*\*  
 \** ***@author*** *qinhaohao  
 \** ***@create*** *2018-01-18 15:33  
 \*\*/***public class** Test {  
 **public static void** main(String[] args) **throws** Exception{  
 RestTemplate restTemplate = **new** RestTemplate();  
 UserCommand userCommand = **new** UserCommand();  
 System.***out***.println(userCommand.getBasic());  
*// String s = userCommand.getBasic1().get(100, TimeUnit.MILLISECONDS);  
// System.out.println(s);* Observable<String> fs = userCommand.getBasic2();  
 fs.subscribe(**new** Action1<String>() {  
 @Override  
 **public void** call(String s) {  
 *//执行结果处理,result 为getBasic2返回的结果  
 //用户对结果做二次处理.* System.***out***.println(**"getBasic2回调函数"**);  
 }  
 });  
 fs.subscribe(**new** Observer<String>() {  
 @Override  
 **public void** onCompleted() {  
 *// onNext/onError完成之后最后回调* System.***out***.println(**"execute onCompleted"**);  
 }  
 @Override  
 **public void** onError(Throwable e) {  
 *// 当产生异常时回调* System.***out***.println(**"onError "** + e.getMessage());  
 e.printStackTrace();  
 }  
 @Override  
 **public void** onNext(String s) {  
 *// 获取结果* System.***out***.println(**"onNext: "** + s);  
 }  
 });  
 System.***out***.println(userCommand.getBasic2().toBlocking().toFuture().get().toString());  
 System.***out***.println(userCommand.getBasic3().toBlocking().toFuture().get().toString());  
 }  
}

#### 6、Hystrix配置

##### Command属性

HystrixCommand.run()隔离策略（通过线程池隔离还是通过信号量隔离）

HystrixCommand超时时间，默认1000毫秒

是否启用超时时间

执行超时的时候是否要将他中断

执行被取消的时候是否中断

使用信号量隔离的时候配置信号量大小，线程数

使用信号量隔离时候并发请求数

##### Fallback配置

GetFallback方法执行的最大并发数，默认10

是否启用fallback

##### circuitBreaker配置

当服务请求命令失败时，是否使用断路器来跟踪其健康指标和熔断请求

滚动时间窗中，断路器熔断的最小请求数:默认20（如果滚动时间窗【默认10秒】内仅收到19个请求，即使这19个请求都失败了，断路器也不会打开））

断路器打开之后的休眠时间窗。休眠时间窗结束之后，会将断路器设置为半开状态，尝试熔断的请求命令，如果依然失败，就设置为打开的状态。(默认5000ms)

熔断器打开的错误百分比（默认50）

断路器强制打开或者关闭

##### requestContext配置

是否开启请求缓存

##### Collapser属性

请求合并批处理中允许的最大请求数

批处理过程中每个命令延迟的时间

##### threadPool属性

执行命令线程池核心线程数 （默认10）

线程池的最大队列大小（默认-1，使用SynchronousQueue实现，不为-1时使用LinkedBlockingQueue实现）

队列设置拒绝阈值(默认5)

设置滚动时间窗长度（默认1000ms）

### 4、feign

声明式调用，结合了ribbon和hystrix

#### 1、使用继承特性来定义双方的接口

##### 1、在公共类定义接口

@RequestMapping(**"/refactor"**)  
**public interface** HelloService {  
 @RequestMapping(value = **"/hello4"**, method = RequestMethod.***GET***)  
 String hello(@RequestParam(**"name"**) String name) ;  
 @RequestMapping(value = **"/hello5"**, method = RequestMethod.***GET***)  
 User hello(@RequestHeader(**"name"**) String name, @RequestHeader(**"age"**) Integer age);  
 @RequestMapping(value = **"/hello6"**, method = RequestMethod.***POST***)  
 String hello(@RequestBody User user);  
}

##### 2、在服务提供方的controller实现该接口，并重写服务具体逻辑

@RestController  
**public class** RefactorHelloController **implements** HelloService {  
 @Override  
 **public** String hello(@RequestParam(**"name"**) String name) {  
 **return "Hello "** + name;  
 }  
 @Override  
 **public** User hello(@RequestHeader(**"name"**) String name, @RequestHeader(**"age"**) Integer age) {  
 **return new** User(name, age);  
 }  
 @Override  
 **public** String hello(@RequestBody User user) {  
 **return "Hello "**+ user.getName() + **", "** + user.getAge();  
 }  
}

##### 3、服务消费方继承该接口

@FeignClient(value = **"HELLO-SERVICE"**)  
**public interface** RefactorHelloService **extends** com.didispace.service.HelloService {  
}

##### 4、好处

服务提供方维护公共类接口和本身具体实现，消费方不需要关注接口的具体实现，也不需要做更改。只需要在消费时调用相应的方法即可。

#### 2、Feign可禁用Hystrix,可全局禁用也可局部禁用

#### 3、针对熔断机制的配置

##### 1、继承服务类接口，重写方法，实现每个方法，方法体内配置每个方法的fallback实现

@Component  
**public class** HelloServiceFallback **implements** HelloService {  
 @Override  
 **public** String hello() {  
 **return "error"**;  
 }  
 @Override  
 **public** String hello(@RequestParam(**"name"**) String name) {  
 **return "error"**;  
 }  
 @Override  
 **public** User hello(@RequestHeader(**"name"**) String name, @RequestHeader(**"age"**) Integer age) {  
 **return new** User(**"未知"**, 0);  
 }  
 @Override  
 **public** String hello(@RequestBody User user) {  
 **return "error"**;  
 }  
}

##### 2、在服务类接口的@FeignClient注解上加上fallback，并指定类

@FeignClient(name=**"HELLO-SERVICE"**, fallback = HelloServiceFallback.**class**)  
**public interface** HelloService {  
 @RequestMapping(**"/hello"**)  
 String hello();

@RequestMapping(value = **"/hello1"**, method = RequestMethod.***GET***)  
 String hello(@RequestParam(**"name"**) String name) ;

@RequestMapping(value = **"/hello2"**, method = RequestMethod.***GET***)  
 User hello(@RequestHeader(**"name"**) String name, @RequestHeader(**"age"**) Integer age);

@RequestMapping(value = **"/hello3"**, method = RequestMethod.***POST***)  
 String hello(@RequestBody User user);  
}

### 5、zuul

api网关，给服务提供统一入口，并且可以在入口处做统一的鉴权中心

1. 作为系统的同一入口，屏蔽了系统内部各个微服务的细节
2. 和服务治理框架结合，实现自动化的服务实例维护以及负载均衡的路由转发
3. 它可以实现接口权限校验与微服务业务逻辑的解耦
4. 通过服务网关中的过滤器，在各个生命周期中去校验请求的内容，将原本在对外服务做的校验前移，保证了微服务的无状态性，同时降低了微服务的测试难度，让服务本身更集中关注业务逻辑的处理

#### 1、配合eureka实现网关（serviceId）

**zuul.routes.api-a.path**=**/api/a/\*\*  
zuul.routes.api-a.serviceId**=**hello-service  
  
zuul.routes.api-b.path**=**/api-b/\*\*  
zuul.routes.api-b.serviceId**=**hello-service**

#### 2、单独使用（url）

**zuul.routes.api-a-url.path**=**/api-a-url/\*\*  
zuul.routes.api-a-url.url**=**http://localhost:8001/**

#### 3、单独使用并且配置多实例

**zuul.routes.api-d.path**=**/ddd/\*\*  
zuul.routes.api-d.serviceId**=**hello  
ribbon.eureka.enabled**=**false 关闭eureka  
hello.ribbon.listOfServers**=**http://localhost:8001/,http://localhost:8002/**

**不使用eureke时，不具备线程隔离和断路器，也不具备负载均衡的能力**

#### 4、请求过滤（ZuulFilter）

**public class** AccessFilter **extends** ZuulFilter {  
  
 **private static** Logger *log* = LoggerFactory.*getLogger*(AccessFilter.**class**);

//过滤器类型，此处定义为前置

//pre 请求被路由前

//routing 路由请求时被调用

//post：routing和error过滤器之后被调用（获取到请求返回的结果，针对结果加工）

//error：处理请求时发生错误时被调用  
 @Override  
 **public** String filterType() {   
 **return "pre"**;  
 }

//过滤器的执行顺序，多个过滤器时按照顺序执行（数值越小优先级越高）  
 @Override  
 **public int** filterOrder() {  
 **return** 0;  
 }

//判断该过滤器是否需要被执行，使用该函数来指定过滤器的有效范围  
 @Override  
 **public boolean** shouldFilter() {  
 **return true**;  
 }

//过滤器的具体逻辑  
 @Override  
 **public** Object run() {  
 RequestContext ctx = RequestContext.*getCurrentContext*();  
 HttpServletRequest request = ctx.getRequest();  
  
 *log*.info(**"send {} request to {}"**, request.getMethod(), request.getRequestURL().toString());  
  
 Object accessToken = request.getParameter(**"accessToken"**);  
 **if**(accessToken == **null**) {  
 *log*.warn(**"access token is empty"**);  
 ctx.setSendZuulResponse(**false**);//过滤该请求，为false不进行路由  
 ctx.setResponseStatusCode(401);//设置返回结果码  
 **return null**;  
 }  
 *log*.info(**"access token ok"**);  
 **return null**;  
 }  
  
}

@EnableZuulProxy  
@SpringCloudApplication  
**public class** Application {  
  
 **public static void** main(String[] args) {  
 **new** SpringApplicationBuilder(Application.**class**).web(**true**).run(args);  
 }  
  
 @Bean  
 **public** AccessFilter accessFilter() {//创建bean使过滤器生效  
 **return new** AccessFilter();  
 }  
  
 @Bean  
 **public** PatternServiceRouteMapper serviceRouteMapper() {  
 **return new** PatternServiceRouteMapper(  
 **"(?<name>^.+)-(?<version>v.+$)"**,//自定义路由映射规则：1、服务名称是否匹配该正则表达式  
 **"${version}/${name}"**);//根据服务的内容转换出路径表达式  
 }  
  
}

#### 服务路由的默认规则

##### 1、当和Eureka结合的时候，它会为eureka上的每个服务都自动创建一个默认路由规则，规则的path会使用serviceId配置的服务名作为请求前缀

##### 2、当不希望被外部访问的服务被访问到，可以通过zuul.ignored-serivces去配置

zuul.ignored-serivces=\*时，将不自动创建，需要全部手动添加

#### 路径匹配

##### 1、书写规则

? 匹配任意单个字符

\* 匹配任意数量的字符

\*\* 匹配任意数量的字符，支持多级目录

##### 2、使用yaml文件来进行有序的配置

从上往下匹配

##### 3、忽略表达式

**zuul.ignored-patterns**=**/\*\*/hello/\*\* 针对所有路由**

#### 7、路由前缀

**zuul.prefix**=**/api**

**zuul.strip-prefix**=**false 关闭前缀**

#### 8、本地跳转

**zuul.routes.api-b-url.path**=**/api-b-url/\*\*  
zuul.routes.api-b-url.url**=**forward:/local 需要在本地提供接口**

@RestController  
**public class** HelloController {  
  
 @RequestMapping(**"/local/hello"**)  
 **public** String hello() {  
 **return "Hello World Local"**;  
 }  
}

#### 9、过滤器详解

##### 1、通过zuul/\*路径访问的请求会绕过dispatcherServlet，被ZuulServlet处理，应对大文件上传，可通过zuul.servletPath修改默认路径

##### 2、没有默认的错误处理的过滤器，需要自己定义

@Component  
**public class** ThrowExceptionFilter **extends** ZuulFilter {  
  
 **private static** Logger *log* = LoggerFactory.*getLogger*(ThrowExceptionFilter.**class**);  
  
 @Override  
 **public** String filterType() {  
 **return "pre"**;  
 }  
  
 @Override  
 **public int** filterOrder() {  
 **return** 0;  
 }  
  
 @Override  
 **public boolean** shouldFilter() {  
 **return true**;  
 }  
  
 @Override  
 **public** Object run() {  
 *log*.info(**"This is a pre filter, it will throw a RuntimeException"**);  
 RequestContext ctx = RequestContext.*getCurrentContext*();  
 **try** {  
 doSomething();  
 } **catch** (Exception e) {  
 ctx.set(**"error.status\_code"**, HttpServletResponse.***SC\_INTERNAL\_SERVER\_ERROR***);  
 ctx.set(**"error.exception"**, e);  
*// ctx.set("error.message", "有一些错误发生");* }  
 **return null**;  
 }  
  
 **private void** doSomething() {  
 **throw new** RuntimeException(**"Exist some errors..."**);  
 }  
  
}

##### 3、ErrorFilter处理

*\*\*  
 \* 捕获为处理的异常统一做一些处理，让`SendErrorFilter`可以正确的返回异常信息  
 \*/  
//@Component***public class** ErrorFilter **extends** ZuulFilter {  
  
 Logger **log** = LoggerFactory.*getLogger*(ErrorFilter.**class**);  
  
 @Override  
 **public** String filterType() {  
 **return "error"**;  
 }  
  
 @Override  
 **public int** filterOrder() {  
 **return** 10;  
 }  
  
 @Override  
 **public boolean** shouldFilter() {  
 **return true**;  
 }  
  
 @Override  
 **public** Object run() {  
 RequestContext ctx = RequestContext.*getCurrentContext*();  
 Throwable throwable = RequestContext.*getCurrentContext*().getThrowable();  
 **log**.error(**"this is a ErrorFilter : {}"**, throwable.getCause().getMessage());  
 ctx.set(**"error.status\_code"**, HttpServletResponse.***SC\_INTERNAL\_SERVER\_ERROR***);  
 ctx.set(**"error.exception"**, throwable.getCause());  
 **return null**;  
 }  
  
}

#### 10、禁用过滤器

*# zuul.<SimpleClassName>.<filterType>.disable=true***zuul.AccessFilter.pre.disable**=**true**

#### 11、post请求抛出异常时没有下家接手，此时需要自定义异常处理

*/\*\*  
 \** ***@author*** *翟永超  
 \** ***@create*** *2016/12/15.  
 \** ***@blog*** *http://blog.didispace.com  
 \*/***public class** DidiFilterProcessor **extends** FilterProcessor {  
  
 @Override  
 **public** Object processZuulFilter(ZuulFilter filter) **throws** ZuulException {  
 **try** {  
 **return super**.processZuulFilter(filter);  
 } **catch** (ZuulException e) {  
 RequestContext ctx = RequestContext.*getCurrentContext*();  
 ctx.set(**"failed.exception"**, e);  
 ctx.set(**"failed.filter"**, filter);//记录下请求filter阶段  
 **throw** e;  
 }  
 }  
}

*/\*\*  
 \* 捕获为处理的异常统一做一些处理，让`SendErrorFilter`可以正确的返回异常信息  
 \* 当是`post`过滤器抛出的话，错误信息会让`ErrorExtFilter`返回异常信息  
 \*/*@Component  
**public class** ErrorFilter **extends** ZuulFilter {  
  
 Logger **log** = LoggerFactory.*getLogger*(ErrorFilter.**class**);  
  
 @Override  
 **public** String filterType() {  
 **return "error"**;  
 }  
  
 @Override  
 **public int** filterOrder() {//先于*ErrorExtFilter`执行*  
 **return** 20;  
 }  
  
 @Override  
 **public boolean** shouldFilter() {  
 **return true**;  
 }  
  
 @Override  
 **public** Object run() {  
 RequestContext ctx = RequestContext.*getCurrentContext*();  
 Throwable throwable = RequestContext.*getCurrentContext*().getThrowable();  
 **log**.error(**"this is a ErrorFilter : {}"**, throwable.getCause().getMessage());  
 ctx.set(**"error.status\_code"**, HttpServletResponse.***SC\_INTERNAL\_SERVER\_ERROR***);  
 ctx.set(**"error.exception"**, throwable.getCause());  
 **return null**;  
 }  
  
}

*/\*\*  
 \* 从POST抛出的异常，使用该过滤器返回错误信息  
 \*/*@Component  
**public class** ErrorExtFilter **extends** SendErrorFilter {  
  
 Logger **log** = LoggerFactory.*getLogger*(ErrorExtFilter.**class**);  
  
 @Override  
 **public** String filterType() {  
 **return "error"**;  
 }  
  
 @Override  
 **public int** filterOrder() {  
 **return** 30;  
 }  
  
 @Override  
 **public boolean** shouldFilter() {  
 RequestContext ctx = RequestContext.*getCurrentContext*();  
 ZuulFilter failedFilter = (ZuulFilter) ctx.get(**"failed.filter"**);//获取请求filter阶段  
 **if**(failedFilter != **null** && failedFilter.filterType().equals(**"post"**)) {  
 **return true**;  
 }  
 **return false**;  
 }  
}

//自定义异常信息

*/\*\*  
 \** ***@author*** *翟永超  
 \** ***@create*** *2016/12/15.  
 \** ***@blog*** *http://blog.didispace.com  
 \*/***public class** DidiErrorAttributes **extends** DefaultErrorAttributes {  
  
 @Override  
 **public** Map<String, Object> getErrorAttributes (  
 RequestAttributes requestAttributes, **boolean** includeStackTrace){  
 Map<String, Object> result = **super**.getErrorAttributes(requestAttributes, includeStackTrace);  
 result.remove(**"exception"**);  
 **return** result;  
 }  
  
}

### 6、spring cloud config

#### 1、服务端配置

**spring:  
 application:  
 name:** config-server  
 **cloud:  
 config:  
 server:  
 git:  
 uri:** https://github.com/qinhaohao1114/springCloud-Learning/  
 **username:** qinhaohao1114  
 **password:** qhh18260639556  
 **search-paths:** config-repo  
**server:  
 port:** 9991  
**eureka:  
 client:  
 service-url:  
 defaultZone:** http://localhost:9990/eureka/（根据是否使用eureka来动态配置）

#### 2、客户端配置

**spring:  
 application:  
 name:** qinhh  
 **cloud:  
 config:  
 discovery:  
 enabled:** true 开启eureka配置  
 **service-id:** config-server  
 **profile:** dev  
**server:  
 port:** 9992  
**eureka:  
 client:  
 service-url:  
 defaultZone:** <http://localhost:9990/eureka/>

**management:  
 security:  
 enabled:** false

不配合euraka

**spring:  
 application:  
 name:** qinhh  
 **cloud:  
 config:  
 profile:** dev  
 **label:** master  
 **uri:** http://localhost:9991/  
**server:  
 port:** 9992

**management:  
 security:  
 enabled:** false

#### 3、开启动态刷新

##### 1、导入actuator包

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

##### 2、配置刷新注解

@RefreshScope  
@RestController  
**public class** TestController {  
  
 @Value(**"${from}"**)  
 **private** String **from**;  
  
 @RequestMapping(**"/from"**)  
 **public** String from() {  
 **return this**.**from**;  
 }  
 **public void** setFrom(String from) {  
 **this**.**from** = from;  
 }  
 **public** String getFrom() {  
 **return from**;  
 }  
}

##### 3、关闭security的安全认证

**management:  
 security:  
 enabled:** false

##### git仓库更改文件之后，像客户端发送/ refresh请求