Spring Cloud Zuul：API网关服务

- 通过使用 Spring Cloud Eureka 实现高可用的服务注册中心以及实现微服务的注册与发现
- 通过 Spring Cloud Ribbon 或 Feign 实现服务间负载均衡的接口调用
- 为了使分布式系统更为健壮，对于依赖的服务调用使用 Spring Cloud Hystrix来进行包装实现线程隔离并加入熔断机制，以避免在微服务架构中因个别服务出现异常而引起级联故障蔓延
- 它的存在就像是整个微服务架构系统的门面一样，所有的外部客户端访问都需要经过它来进行调度和过滤。它除了要实现请求路由、 负载均衡、 校验过滤等功能之外， 还需要更多能力， 比如与服务治理框架的结合、 请求转发时的熔断机制、 服务的聚合等一系列高级功能
- 路由规则与服务实例的维护问题
    - SpringCloudZuul通过与SpringCloudEureka进行整合， 将自身注册为Eureka服务治理下的应用， 同时从Eureka中获得了所有其他微服务的实例信息。这样的设计非常巧妙地将服务治理体系中维护的实例信息利用起来， 使得将维护服务实例的工作交给了服务治理框架自动完成， 不再需要人工介入
    - 而对于路由规则的维护， Zuul默认会将通过以服务名作为ContextPath的方式来创建路由映射， 大部分情况下， 这样的默认设置已经可以实现我们大部分的路由需求
- 对于类似签名校验、 登录校验在微服务架构中的冗余问题
    - 完全可以独立成一个单独的服务存在，只是它们被剥离和独立出来之后， 并不是给各个微服务调用， 而是在API网关服务上进行统一调用来对微服务接口做前置过滤， 以实现对微服务接口的拦截和校验
    - SpringCloudZuul提供了一套过滤器机制， 它可以很好地支持这样的任务。 开发 者可以通过使用Zuul来创建各种校验过滤器，然后指定哪些规则的请求需要执行校验逻辑，只有通过校验的才会被路由到具体的微服务接口，不然就返回错误提示
    - 通过这样的改造，各个业务层的微服务应用就不再需要非业务性质的校验逻辑了， 这使得我们的微服务应用可以更专注千业务逻辑的开发， 同时微服务的自动化测试也变得更容易实现

入门

- `api-gateway`
- zuul依赖，其还包含了
    - hystrix，该依赖用来在网关服务中实现对微服务转发时候的保护机制， 通过线程隔离和断路器，防止微服务的故障引发API网关资源无法释放， 从而影响其他应用的对外服务
    - ribbon，用来实现在网关服务进行路由转发时候的客户端负载均衡以及请求重试
    - actuator，用来提供常规的微服务管理端点，在Spring Cloud Zuul中还特别提供了/routes端点来返回当前的所有路由规则
    - 所以 Zuul天生就拥有线程隔离和断路器的自我保护功能，以及对服务调用的客户端负载均衡功能
        - 但是需要注意， 当使用path与url的映射关系来配置路由规则的时候，对于路由转发的请求不会采用HystrixCommand来包装， 所以这类路由请求没有线程隔离和断路器的保护，并且也不会有负载均衡的能力。因此，我们在使用Zuul 的时候尽量使用path和

serviceId的组合来进行配置， 这样不仅可以保证API网关的健壮和稳定，也能用到Ribbon的客户端负载均衡功能

- 主类，@EnableZuulProxy注解开启Zuul的API网关服务功能
- application.properties，服务名称（api-gateway），端口（5555）
- 启动

传统路由

- 不依赖千服务发现机制的情况下， 通过在配置文件中具体指定每个 路由表达式与服务实例的映射关系来实现API网关对外部请求的路由
- 传统路由的映射方式比较直观且容易理解， API网关直接根据请求的URL路径找到最 匹配的 path 表达式， 直接转发给该表达式对应的 url或对应 serviceId下配置的实例地址， 以实现外部请求的路由
- 只需要对 api-gateway服务增加一些关于路由规则的配置， 就能实现传统的路由转发功能
  - 单实例配置：通过zuul.routes..path与zuul.routes..url参数对的方式进行配置

```
# 所有符合/api-a-url/**规则的访问都将被路由转发到http://localhost:8080/地址上
# 当访问 http://localhost:5555/api-a-url/hello的时候，API网关服务会将该请求路由到http://localhost:8080/hello提供的微服务接口上
# api-a-url部分为路由的名字，可以任意定义，但是一组path和url映射关系的路由名要相同
zuul.routes.api-a-url.path=/api-a-url/**
zuul.routes.api-a-url.url=http://localhost:8080/
```

  - 多实例配置：通过zuul.routes. .path与zuul.routes. .serviceId参数对的方式进行配置

```
zuul.routes.user-service.path=/user-service/**
# 与面向服务的路由的区别是serviceid是由用户手工命名的服务名称，配合ribbon.listOfServers参数实现服务与实例的维护
zuul.routes.user-service.serviceId=user-service
# 由于zuul.routes.<route>.serviceId指定的是服务名称，默认清况下Ribbon会根据服务发现机制来获取配置服务名对应的实例清单。但是，该示例并没有整合类似Eureka之类的服务治理框架，所以需要将该参数设置为false，否则配置的serviceId获取不到对应实例的清单
ribbon.eureka.enabled=false
# 由于存在多个实例，API网关在进行路由转发时需要实现负载均衡策略，于是这里还需要SpringCloudRibbno的配合
# 符合/user-service/** 规则的请求路径转发到http://localhost:8080/和http://localhost:8081/两个实例地址的路由规则
# 参数内容与zuul.routes.<route>.serviceId的配置相对应，开头的user-service对应了serviceId的值， 这两个参数的配置相当于在该应用内部手工维护了服务与实例的对应关系
user-service.ribbon.listOfServers=http://localhost:8080/,http://localhost:8081/
```

  - 不论是单实例还是多实例的配置方式，我们都需要为每一对映射关系指定一个名称，也就是上面配置中的 , 每 一个对应了 一条路由规则。 每条路由规则都需要通过path属性来定义一个用来匹配客户端请求的路径表达式， 并通过url或service工d属性来指定请求表达式映射具体实例地址或服务名

面向服务的路由

- SpringCloud Zuul实现了与 SpringCloudEureka的无缝整合， 我们可以让路由的path不是映射具体的url, 而是让它映射到某个具体的服务，而具体的 url则交给Eureka的服务发现机制去自动维护，不需要向传统路由配置方式那样为serviceId指定具体的服务实例地 址，只需要通过zuul.routes. .path与zuul.routes. .serviceId 参数对的方式进行配置即可， 其中可以指定为任意的路由名称
- 与Eureka整合，eureka
- 在api-gateway的application.properties 配置文件中指定Eureka注册中心的位置， 并且配置服务路由

```
# http://localhost: 5555/api-a/hello，发送到hello-service服务的某个实例上去
zuul.routes.api-a.path=/api-a/**
zuul.routes.api-a.serviceId=hello-service
# http://localhost:5555/api-b/feign-consumer，发送到feign-consumer服务的某个实例上去
zuul.routes.api-b.path=/api-b/**
zuul.routes.api-b.serviceId=feign-consumer
# 指定服务注册中心
eureka.client.serviceUrl.defaultZone=http://localhost:llll/eureka/

# 更简洁的配置方式：zuul.routes.<serviceid>=<path>，其中<serviceid> 用来指定路由的具体服务名， <path>用来配置匹配的请求表达式
zuul.routes.user-service=/user-service/**
```

- Zuul巧妙地整合了 Eureka 来实现面向服务的路由。 实际上，我们可以直接将API网关也看作Eureka服务治理下的一个普通微服务应用。 它除了 会将自己注册到Eureka服务注册中心上之外，也会从注册中心获取所有服务以及它们的实例清单。所以，在Eureka的帮助下，API网关服务本身就已经维护了系统中所有 serviceId 与实例地址的映射关系。
- 当有外部请求到达 API 网关的时候，根据请求的 URL 路径找到最佳匹配的path规则， API 网关就可以知道要将该请求路由到哪个具体的serviceId上去。
- 由于在 API 网关中已经知道serviceid对应服务实例的地址清单， 那么只需要通过Ribbon的负载均衡策略， 直接在这些清单中选择一个具体的实例进行转发就能完成路由工作了
- 默认规则
  - 当为Spring CloudZuul构建的 API 网关服务引入SpringCloudEureka之后， 它为Eureka中的每个服务都自动创建一个默认路由规则， 这些默认规则的path会使用serviceId配置的服务名作为请求前缀。
  - 由于默认情况下所有 Eureka上的服务都会被 Zuul自动地创建映射关系来进行路由，这会使得一些我们不希望对外开放的服务也可能被外部访问到
  - 可以使用zuul.ignored-services参数来设置一个服务名匹配表达式来定义不自动创建路由的规则。Zuul在自动创建服务路由的时候会根据该表达式来进行判断， 如果服务名匹配表达式，那么 Zuul 将跳过该服务， 不为其创建路由规则
  - zuul.ignored-services=*的时候，Zuul将对所有的服务都不自动创建路由规则，就要在配置文件中逐个为需要路由的服务添加映射规则，只有在配置文件中出现的映射规则会被创建路由， 而从Eureka中获取的其他服务，Zuul将不会再为它们创建路由规则
- 自定义服务路由规则

- 有时候会需要为一组互相配合的微服务定义一个版本标识来方便管理它们的版本关系，根据这个标识我们可以很容易地知道这些服务需要一起启动并配合使用
- 默认情况下，Zuul自动为服务创建的路由表达式会采用服务名作为前缀， 比如针对上面的userservice-vl 和userservice-v2, 它会产生/userservice-vl 和/userservice-v2两个路径表达式来映射，但是这样生成出来的表达式规则较为单一，不利于通过路径规则来进行管理
- 通常的做法是为这些不同版本的微服务应用生成以版本代号作为路由前缀定义的路由规则，比如 /vl/userservice/。 这时候，通过这样具有版本号前缀的URL路径，就可以很容易地通过路径表达式来归类和管理这些具有版本信息的微服务了
- 如果我们的各个微服务应用都遵循了类似userservice-vl 这样的命名规则， 通过-分隔的规范来定义服务名和服务版本标识的话，那么，我们可以使用Zuul中自定义服务与路由映射关系的功能，来实现为符合上述规则的微服务自动化地创 建类似/vl/userservice/** 的路由匹配规则
- 实现步骤非常简单， 只需在 API 网关程序中， 增加如下Bean的创建 即可

```
@Bean
public PatternServiceRouteMapper serviceRouteMapper() {
  //PatternServiceRouteMapper对象通过正则表达式来自定义服务与 路由映射的生成关系
  //第一个参数，匹配服务名称是否符合该自定义 规则的正则表达式
  //第二个参数，定义根据服务名中定义的内容转换出的路径表达式 规则
  return new PatternServiceRouteMapper("(?<name>A .+)-(?<version>v.+$)", "${version}/${name}");
}
//当开发者在 API 网关中定义了PatternServiceRouteMapper 实现之后，只要符合第一个参数定义规则的服务名，都会优先使用该实现构建出的路径表达式，如果没有匹配上的服务则还是会使用默认的路由映射规则，即采用完整服务名作为前缀的路径表达式
```

- 路径匹配
  - path参数，Ant风格，
    - ?-匹配任意个单字符、*-匹配任意数量的字符、**-匹配任意数量的字符， 支待多级目录
  - 当存在多个匹配的路由规则时， 匹配结果完全取决于路由规则的保存顺序
- 忽略表达式
  - 可以用来设置 不希望被 API 网关进行路由的 URL 表达式
  - 它的范围并不是对某个路由， 而是对所有路由。 所以在设置的时候需要全面考虑 URL 规则，防止忽略了不该被忽略的 URL 路径

```
# 不希望/hello 接口被路由
zuul.ignored-patterns=/**/hello/**
zuul.routes.api-a.path=/api-a/**
zuul.routes.api-a.serviceid=hello-service
```

- 路由前缀
  - 为了方便全局地为路由规则增加前缀信息，Zuul 提供了 zuul.prefix 参数来进行设置
  - 可以通过设置 zuul.stripPrefix= false 来关闭该移除代理前缀的动作

    - 也可以通过 zuul.routes..strip-prefix=true 来对指定路由关闭移除代理前缀的动作
- 本地跳转
    - 通过 url 中使用 forward 来指定需要跳转的服务器资源路径

```
zuul.routes.api-a.path=/api-a/**
zuul.routes.api-a.url=http://localhost:8001/
# 将符合 /api-b/**规则的请求转发到API网关中以/local为前缀的请求上，由API网关进行本地处理
# 请求/api-b/hello，它符合 api-b 的路由规则， 所以该请求会被 API 网关转发到网关的/local/hello 请求上进行本地处理
# 在 API 网关上还需要增加一个/local/hello 的接口实现才能让 api-b 路由规则生效
zuul.routes.api-b.path=/api-b/**
zuul.routes.api-b.url=forward:/local
```

- Cookie与头信息
    - 默认情况下， SpringCloud Zuul在请求路由时， 会过滤掉HTTP请求头信息中的 一些敏感信息， 防止它们被传递到下游的外部服务器
    - 默认的敏感 头信息通过 zuul.sensitiveHeaders参数定义，包括Cookie、Set-Cookie、Authorization 三个属性
    - 如果我们要将使用了Spring Security、 Shiro 等安全框架构建的Web应用通过SpringCloud Zuul构建的 网关来进行路由时，由千Cookie信息无法传递， 我们的Web应用将无法实现登录和鉴权
    - 解决
        - 通过设置全局参数为空来覆盖默认值`zuul.sensitiveHeaders=`，不推荐
        - 通过指定路由的参数来配置
            - 方法一：对指定路由开启自定义敏感头
                - `zuul.routes.<router>.customSensitiveHeaders=true`
            - 方法二：将指定路由的敏感头设置为空
                - `zuul.routes.<router>.sensitiveHeaders=`
- 重定向问题
    - 虽然可以通过网关访问登录页面并发起登录请求，但是登录成功之后， 我们跳转到的页面URL却是具体Web应用实例的地址， 而不是通过网关的路由地址
        - 这个问题非常严重，因为使用API网关的一个重要原因就是要将网关作为统一入口， 从而不暴露所有的内部服务细节
    - 引起问题的大致原因是由于SpringSecurity或Shiro在登录完成之后，通过重定向的方式跳转到登录后的页 面， 此时登录后的请求结果状态码为302，请求响应头信息中的 Location指向了具体的服务实例地址，而请求头信息中的 Host也指向了具体的服务实例IP地址和端口。 所以，该问题的根本原因在于Spring CloudZuul在路由请求时，并没有将最初的Host信息设置正确
    - 解决，参数配置，使得网关在进行路由转发前为请求设置Host头信息，以标识最初的服务端请求地址`zuul.addHostHeader=true`

请求过滤

- 在实现了请求路由功能之后，微服务应用提供的接口 就可以通过统一的 API网关入口被客户端访问到了
- 对客户端请求的安全校验和权限控制，通过前置的网关服务来完成这些非业务性质的校验，在请求到达的时候就完成校验和过滤， 而不是转发后再过滤而导致更长的请求延迟。 同时，通过在网关中完成校验和过滤，微服务应用端就可以去除各种复杂的过滤器和拦截器了， 这使得微服务应用接口的开发和测试复杂度也得到了相应降低
- Zuul 允许开发者在API网关上通过定义过滤器来实现对请求的拦截与过滤，实现的方法非常简单，只需要继承 ZuulFiter 抽象类并实现它定义的4个抽象函数就可以完成对请求的拦截和过滤了

```
//实现了在请求被路由之前检查HttpServletRequet中是否有accessToken参数，若有就进行路由，若没有就拒绝访问，返回401Unauthorized错误
@Component
//或者在主类中，定义@Bean public AccessFilter accessFilter()return new AccessFilter();}
public class AccessFilter extends ZuulFilter {
    private static Logger log = LoggerFactory.getLogger(AccessFilter.class);

    //过滤器的类型，  它决定过滤器在请求的哪个生命周期中执行
    @Override
    public String filterType() {return "pre";}  //pre，代表会在请求被路由之前执行

    //过滤器的执行顺序
    @Override
    public int filterOrder() {return 0;}

    //断该过滤器是否需要被执行，true为过滤器需要被执行
    // 实际运用中可以利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.getRequestURI().toString());

        String accessToken = request.getParameter("accessToken");
        if(accessToken == null) {
            log.warn("access token is empty");
            //不对其进行路由
```

```
            ctx.setSendZuulResponse(false);
            ctx.setResponseStatusCode(401);
            return null;
        }
        log.info("access token ok");
        return null;
    }
}
//访问
//http://localhost:5555/api-a/hello: 返回401错误
//http://localhost:5555/api-a/hello&accessToken=token: 正确路由 到hello-service的/hello接口
```

作用

- 它作为系统的统一入口， 屏蔽了系统内部各个微服务的细节
- 它可以与服务治理框架结合，实现自动化的服务实例维护以及负载均衡的路由转发
- 它可以实现接口权限校验与微服务业务逻辑的解耦
- 通过服务网关中的过炖器， 在各生命周期中去校验请求的内容， 将原本在对外服务层做的校验前移， 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

Hystrix 和 Ribbon 支持

- 在使用Zuul搭建API网关的时候，可以通过Hystrix和Ribbon的参数来调整路由请求的各种超时时间等配置
- `hystrix.command.default.execution.isolation.thread.timeoutinMilliseconds`
    - 用来设置API 网关中路由转发请求的 HystrixCommand 执行超时时间（毫秒），当路由转发请求的命令执行时间超过该配置值之后，Hystrix会将该执行命令标记为TIMEOUT并抛出异常，Zuul会对该异常进行处理并返回错误JSON信息给外部调用方
- `ribbon.ConnectTimeout`：设置路由转发请求的时候， 创建请求连接的超时时间
    - 当其值小于hystrix.command.default.execution.isolation.thread.timeoutlnMilliseconds 配置值的时候， 若出现路由请求出现连接超时，会自动进行重试路由请求，如果重试依然失败，Zuul会返回JSON信息给外部调用方
    - 如果 ribbon.ConnectTimeot的配置值大千hystrix.command.default.execution.isolation.thread.timeoutlnMilliseconds配置值的时候， 当出现路由请求连接超时时， 由于此时对于路由转发的请求命令已经超时， 所以不会进行重试路由请求， 而是直接按请求命令超时处理， 返回TIMEOUT的错误信息。
- `ribbon.ReadTimeout`：用来设置路由转发请求的超时时间。它的超时是对请求连接建立之后的处理时间
    - 当其值小于 hystrix.command.default.execution.isolation.thread.timeoutinMilliseconds配置值的时候，若路由请求的处理时间超过该配置值且依赖服务的请求还未响应的时候， 会自动进行重试路由请求。如果重试后依然没有获得请求响应，Zuul会返回NUMBEROF_RETRIES_NEXTSERVER_EXCEEDED 错误 。
    - 当其值大于 hystrix.command.default.execution.isolation.thread.timeoutinMilliseconds，

若路由请求的处理时间超过该配置 值且依赖服务的请求还未响应时，不会进行重试路由请求，而是直接按请求命令超时处理，返回TIMEOUT 的错误信息

- 在使用Zuul的服务路由时，如果路由转发请求发生超时(连接超时或处理超时)， 只要超时时间的设置小于Hystrix的命令超时时间，那么它就会自动发起重试。 但是在有些情况下，我们可能需要关闭该 重试机制，那么可以通过下面的两个参数来进行设置

```
# 用来全局关闭重试机制
zuul.retryable= false
# 指定路由关闭重试机制
zuul.routes.<route>.retryable= false
```

过滤器

- Zuul的第一印象通常是这样的：它包含了对请求的路由和过滤两个功能，其中路由功能负责将外部请求转发到具体的微服务实例上，是实现外部访问统一入口的基础；而过滤器功能则负责对请求的处理过程进行干预，是实现请求校验、服务聚合等功能的基础。然而实际上， 路由功能在真正运行时， 它的路由映射和请求转发都 是由几个不同的过滤器完成的
  - 其中，路由映射主要通过pre类型的过滤器完成，它将请求路径与配置的路由规则进行匹配，以找到需要转发的目标地址
  - 而请求转发的部分则是由route类型的过滤器来完成，对pre类型过滤器获得的路由地址进行转发
  - 所以，过滤器可以说是Zuul实现API网关功能最为核心的部件，每一个进入Zuul的HTTP 请求都会经过一 系列的过滤器处理链得到请求响应并返回给客户端
- 4个基本特征: 过滤类型、 执行顺序、 执行条件、 具体操作
  - FilterType：该函数需要返回一个字符串来代表过滤器的类型，而这个类型就是在HTTP请求过程中定义的各个阶段。在 Zuul 中默认定义了4 种不同生命周期的过滤器类型
    - pre: 可以在请求被路由之前调用
    - routing: 在路由请求时被调用。
    - post: 在 routing 和 error 过滤器之后被调用
    - error: 处理请求时发生错误时被调用
  - filterOrder: 通过 int 值来定义过滤器的执行顺序， 数值越小优先级越高
  - shouldFilter: 返回一个 boolean 值来判断该过滤器是否要执行。 可以通过此方法来指定过滤器的有效围
  - run: 过滤器的具体逻辑。 在该函数中， 我们可以实现自定义的过滤逻辑， 来确定是否要拦截当前的请求， 不对其进行后续的路由， 或是在请求路由返回结果之后，对处理结果做一 些加工等
- 请求生命周期
  - Zuul 默认定义了4种不同的过滤器类型， 它们覆盖了一个外部HTTP请求到达API网关， 直到返回请求结果的全部生命周期
  - 当外部 HTTP 请求到达 API 网关服务的时候， 首先它会进入第一个阶段pre, 在这里它会被pre类型的过滤器进行处理，该类型过滤器的主要目的是在进行请求路由之前做一些前置加工， 比如请求的校验等
  - 第二个阶段routing, 也就是之前说的路由请求转发阶段，请求将会被routing类型过滤器处理。这里的具体处理内容就是将外部请求转发到具体服务实例上去的过程，当服务实例将请

求结果都返回之后， routing 阶段完成
    - 第三个阶段 post。此时请求将会被post类型的过滤器处理，这些过滤器在处理的时候不仅可以获取到请求信息， 还能获取到服务实例的返回信息， 所以在 post类型的过滤器 中，我们可以对处理结果进行 一 些加工或转换等内容
    - 特殊的阶段error, 该阶段只有在上述三个阶段中发生异常的时候才会触发，但是它的最后流向还是 po江类型的过滤器，因为它需要通过post过滤器将最终结果返回给请求客户端
- 核心过滤器
    - 在Spring Cloud Zuul中， 为了让 API 网关组件可以被更方便地使用， 它在 HTTP 请求生命周期的各个阶段默认实现了一批核心过滤器， 它们会在 API 网关服务启动的时候被自动加载和启用
    - pre过滤器
    - route过滤器
    - post过滤器
- 异常处理
    - 先创建pre 类型的过滤器RibbonRoutingFiter ，并在该过滤器的 run 方法实现中直接抛出一个异常
    - try catch处理
        - 有一个 post 过滤器SendErrorFilter 是用来处理异常信息
        - error.status_code 参数就是 SendErrorFilter 过滤器用来判断是否需要执行的重要参数

```
//RibbonRoutingFiter extends ZuulFiler
public Object run() {
        log.info("This is a pre filter, it will throw a
RuntimeException");
  RequestContext ctx = RequestContex七.getCurrentCon七ext();
        try {
                doSomething();
        ) catch (Exception e) {
                ctx.set("error.status_code",
HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
ctx.set("error.exception", e);
  }
                return null;
}
```

        - 对于message 的信息，我们在过滤器中还可以通过 ctx.set("error.message", " 自定义异常消息") ; 来定义更友好的错误信息。 SendErrorFilter会优先取error.message作为返回的 message 内容，如果没有的话才会使用 Exception中的 message 信息
    - ErrorFilter 处理
        - 在过滤器中使用 try-catch 来处理业务逻辑并向请求上下文中添加异常信息，但是不可控的人为因素、 意料之外的程序因素等， 依然会使得一些异常从过滤器中抛出， 对于意外抛出的异常又会导致没有控制台输出也没有任何响应信息的情况出现，可以用到 error 类型的过滤器了

- 由于在请求生命周期的 pre、route 、 post 三个阶段中有异常抛出的时候都会进入 error 阶段的处理， 所以可以通过创建一个 error 类型的过滤器来捕获这些异常信息，并根据这些异常信息在请求上下文中注入需要返回给客户端的错误描述

```
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= ctx.getThrowable();
                log.error("this is a ErrorFilter : {}",
throwable.getCause().ge七Message());
ctx.set("error.status_code", HttpServletResponse.SC_INTERNAL_SERVER—
ERROR);
    Ctx.set("error.exception", throwable.getCause());
    return null;
  }
}
```

- 不足与优化
  - 对自定义过滤器中处理异常的两种基本解决方法：
    - 一种是通过在各个阶段的过滤器中增加 try-catch 块，实现过滤器内部的异常处理;
    - 另一种是利用 error 类型过滤器的生命周期特性，集中处理pre、route、post阶段抛出的异常信息。
    - 通常情况下，我们可以将这两种手段同时使用，其中第一种是对开发人员的基本要求; 而第二种是对第一种处理方式的补充， 以防止意外情况的发生
  - 不足根源：对于从 post 过滤器中抛出异常的清况， 在经过error过滤器处理之后， 就没有其他类型的过滤器来接手了
  - 上面两种处理方法都在异常处理时向请求上下文中添加了一系列的 error.*参数，而这些参数真正起作用的地方是在 post阶段的SendErrorFilter, 在该过滤器中会使用这些参数来组织内容返回给客户端。
    - 而对 于 post阶段抛出异常的情况下， 由error过滤器处理之后 并不会再调用 post 阶段的请求 ， 自然这些error.*参数也就不会被SendErrorFilter消费输出
    - 所以， 如果我们在自定义post过滤器的时候， 没有正确处理异常，就依然有可能出现日志中没有异常但请求响应内容为空的问题
  - 解决上述问题的方法有很多种

- 最直接的是可以在实现 error过滤器的时候，直接组织结果返回就能实现效果。但是这样做的缺点也很明显，对于错误信息组织和返回的代码实现会存在多份，这样非常不利于日后的代码维护工作。所以为了保持对异常返回处理逻辑的一致性，还是希望将post过滤器抛出的异常交给SendErrorFilter来处理。
- 在前文中，已经实现了一个ErrorFilter来捕获pre、 route、 post过滤器抛出的异常， 并组织 error.*参数保存到请求的上下文中。
- 由于我们的目标是沿用SendErrorFi让er, 这些error.*参数依然对我们有用，所以可以继续沿用该过滤器，让它在 post过滤器抛出异常的时候，继续组织error.*参数，只是这里我们已经无法将 这些 error.*参数再传递给SendErrorFilter 过滤器来处理了
- 所以，我们需要在 ErrorFilter 过滤器之后再定义一 个 error 类型的过滤器， 让它来实现 SendErrorFilter的功能， 但是这个error过滤器并不需要处理所有出现异常的情况， 它仅对 post 过滤器抛出的异常有效
- 根据上面的思路， 我们完全可以创建一个继承自 SendErrorFilter的过滤器，复用它的 run方法，然后重写它的类型、顺序以及执行条件，实现对原有逻辑的复用

```
@Component
public class ErrorExtFilter extends SendErrorFilter {
  @Override
        public String filterType () { return "error";}

  @Override
        public int filterOrder () {
    return 30; //大于 ErrorF过ter 的值
  }

  @Override
        public boolean shouldFilter() {
                // TODO判断:仅处理来自post过滤器引起的异常
    RequestCon七ext ctx = RequestContext.getCurrentContext();
                ZuulFilter failedFilter = (ZuulFilter)
ctx.get("failed.filter");
    if(failedFilter != null &&
failedFiter.filterType().equals("post")) {
                        return true;
    }
                return false;
  }
}
```

- 怎么判断引起异常的过滤器来自什么阶段呢? shouldF辽ter方法该如何实现，不得不扩展原来的过滤器处理逻辑。 当有异常抛出的时候， 记录下抛出异常的过滤器， 这样我们就可以在ErrorExtFilter 过滤器的shouldFilter方法中获取并以此判断异常是否来自post 阶段的过滤器了
- 可以直接扩展processZuulFilter(ZuulFilter filter)，当过滤器执行抛出异常的时候， 我们捕获它，并向请求上下中记录一些信息

```
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.filter", filter);
                        throw e;
    }
  }
}
```

    - 因为扩展的过滤器处理类并还没有生效。 最后，需要在应用主类中，通过调用 FilterProcessor.setProcessor (new DidiFilterProcessor()); 方法来启用自定义的核心处理器以完成我们的优化目标

- 自定义异常信息
    - 如果不采用重写过滤器的方式，依然想要使用 SendErrorFilter 来处理异常返回的话，我们要如何定制返回的响应结果呢?
    - 这个 时候，我们的关注点就不能放在 Zuul 的过滤器上了， 因为错误信息的生成实际上并不是由 Spring Cloud Zuul完成的。 SendErrorFilter 它会根据请求上下中保存的错误信息来组织 一 个 forward 到/error 端点的请求来获取错误响应， 所以我们的扩展目标转移到了对/error端点的实现
    - 只需要自己编写一个自定义的 ErrorAttributes 接口实现类，并创建它的实例就能替代这个默认的 实现， 从而达到自定义错误信息的效果了
    - 比如我们不希望将 excep巨on 属性返回给客户端， 那么就可以编写 一 个自定 义 的 实现 ， 它可以基于 DefaultErrorAttributes , 然后重写getErrorAttributes 方法，从原来的结果中将 exception 移除即可

```
public class DidiErrorA七tributes 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;
  }
}
```

    - 最后， 为了让自定义的错误信息生成逻辑生效， 需要在应用主类中加入如下代码， 为其创建实例来替代默认的实现

```
@Bean
public DefaultErrorAttributes errorAttributes () {
  return new DidiErrorAttributes ();
}
```

        - 通过上面的方法就能基于Zuul的核心过滤器来灵活地自定义错误返回信息，以满足实际应用系统的响应格式了

- 禁用过滤器
    - `zuul.<SimpleClassName>.<filterType>.disable=true`
        - 代表过滤器的类名
        - 代表过滤器类型
        - `zuul.AccessFilter.pre.disable=true`
    - 可以用它来禁用Spring CloudZuul中默认定义的核心过滤器，实现一套更符合我们实际需求的处理机制

动态加载

- 作为最外部的网关， 它必须具备动态更新内部逻辑的能力， 比如动态修改路由规则、 动态添加/删除过滤器等
- 通过Zuul实现的API网关服务当然也具备了动态路由和动态过滤器的能力。可以在不重启 API网关服务的前提下， 为其动态修改路由规则和添加或删除过滤器
- 对于路由规则的控制几乎都可以在配置文件application.properties或application.yaml中完成，只需将API网关服务的配置文件通过Spring Cloud Config连接的Git仓库存储和管理， 就能轻松实现动态刷新路由规则的功能
- `api-gateway-dynamic-route`，引入`zuul`、`config`、`eureka`
- 在/resource目录下创建配置文件bootstrap.properties, 并在该文件中指定config-server和eureka-server的具体地址， 以获取应用的配置文件和实现服务注册与发现

```
spring.application.name=api-gateway-dynamic
server.port=5556
spring.cloud.config.uri=http://localhost:7001/
eureka.client.serviceUrl.defaultZone=http://localhost:1110/eureka/
```

- 主类，@EnableZuulProxy，需要使用@RefreshScope注解来将Zuul的配置内容动态化

```
@Bean
@RefreshScope
@ConfigurationProperties("zuul")
public ZuulProperties zuulproperties() {
  return new ZuulProperties();
}
```

- Git 仓库中增加网关的配置文件

    - 对于 API 网关服务在 Git 仓库中的配置文件名称完全取决于网关应用配置文件 bootstrap.properties 中 spring.application.name 属性的配置值

```
#  api-gateway-dynamic.properties
zuul.routes.service-a.path=/service-a/**
zuul.routes.service-a.serviceid=hello-service
zuul.routes.service-b.path=/service-b/**
zuul.routes.service-b.url=http://localhost:8001/
```

- 启动，通过对 API 网关服务调用/routes 接口来获取当前网关上的路由规则
- 先访问原来的路有规则，然后在git中修改路有规则，通过向api-gateway-dynamic-route 的/refresh 接口发送POST 请求来刷新配置信息。 当配置文件有修改的时候，该接口会返回被修改的属性名称
- 就已经完成了路由规则的动态刷新， 可以继续通过 API 网关服务的/routes 接口来查看当前的所有路由规则

动态过滤器

- 对于实现请求过滤器的动态加载， 需要借助基于NM实现的动态语言的帮助， 比如Groovy