Spring-Cloud组件：

zuul:

zuul是什么?

Zuul包含了对请求的路由和过滤两个最主要的功能：

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.

Zuul和Eureka进行整合，将Zuul自身注册为Eureka服务治理下的应用，同时从Eureka中获得其他微服务的消息，也即以后的访问微服务都是通过Zuul跳转后获得。

注意：Zuul服务最终还是会注册进Eureka

路由：

项目加入依赖:

```
<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-zuul</artifactId>
</dependency>
```

因为上文说过Zuul最终会注册进eureka 所以我们这里也依赖了eureka

yml文件:

```
server:
  port: 9000

eureka:

  client:
    serviceUrl:
        defaultZone: http://localhost:3000/eureka/
  instance:
    instance-id: zuul-1
    prefer-ip-address: true

spring:
  application:
```

```
    name: zuul
```

启动类:

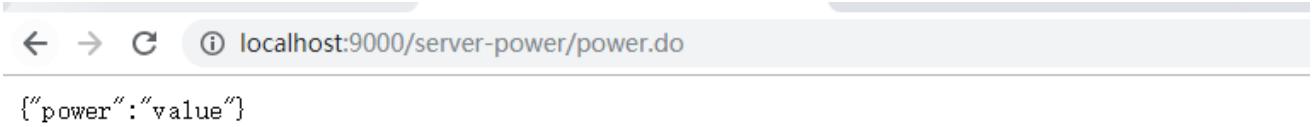
```
/**
 * 想要咨询vip课程相关的同学加一下安其拉老师QQ：3164703201
 * 想要往期视频的同学加一下妮可老师QQ：2860884084
 * author：鲁班学院-商鞅老师
 */
@SpringBootApplication
@EnableZuulProxy
public class AppZuul {

    public static void main(String[] args) {
        SpringApplication.run(AppZuul.class);
    }
}
```

这样 简单的zuul就搭建好了， 启动项目 我们就可以通过zuul然后加上对应的微服务名字访问微服务:

看看eureka上面的微服务名称

Application	AMIs	Availability Zones	Status
SERVER-ORDER	n/a (1)	(1)	UP (1) - order-2
SERVER-POWER	n/a (1)	(1)	UP (1) - power-1
ZUUL	n/a (1)	(1)	UP (1) - zuul-1

调用:



{"power":"value"}

到这里 一个简单的zuul已经搭建好了

在实际开发当中我们肯定不会是这样通过微服务调用，比如我要调用power 可能只要一个/power就好了 而不是/server-power

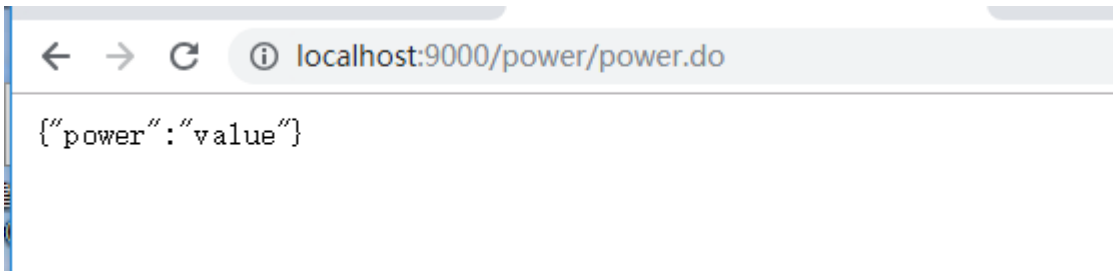
在yml加入以下配置即可:

```
zuul:
  routes:
    mypower:
      serviceId: server-power
      path: /power/**
    myorder:
      serviceId: server-order
      path: /order/**
```

讲道理看意思都看得出来把，my***是自己制定的名字 这个就不解释了

注意/ **代表是所有层级 / * 是代表一层。 如果是/ * 的话 /power/admin/getUser.do 就不会被路由 。

来看效果：

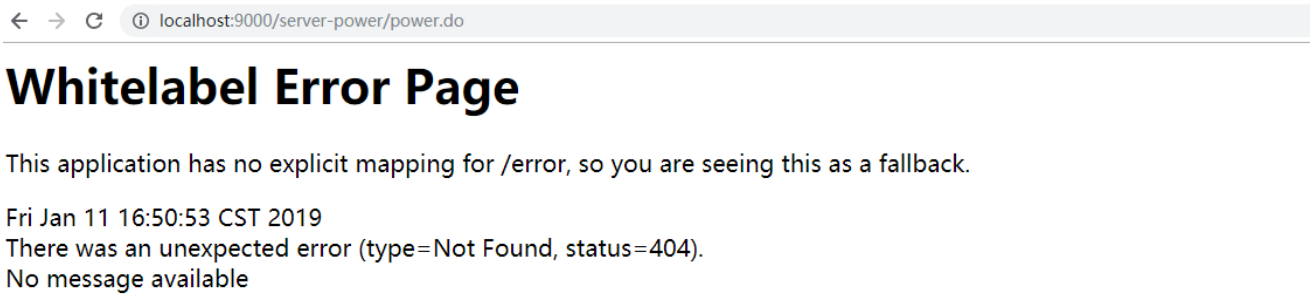


这时候我们能通过我们自定义的规则来访问了，但是还有一个问题，就是我们现在依然能用之前的微服务名调用，这样子是不合理的，第一是有多重地址了， 第二，一般微服务名这种最好不要暴露在外。所以我们一般会禁用微服务名方式调用。

加入配置:

```
ignored-services: server-power
```

这里咱们先禁用power的看看：



这里能发现我们不能通过微服务名来调用了， 不过这个配置

如果一个一个通过微服务名来配置难免有点复杂，所以一般这样配置来禁用所有:

```
ignored-services: "*"
```

可能有时候我们的接口调用需要一定的规范，譬如调用微服务的API URL前缀需要加上/api 对于这种情况， zuul也考虑到了并给出了解决方案：

```
zuul:
  prefix: /api
  ignored-services: "*"
  routes:
    mypower:
      serviceId: server-power
      path: /power/**
    myorder:
      serviceId: server-order
      path: /order/**
```

加上一个prefix 即定义好了一个前缀， 那么我们每次需要路由的时候需要加上一个/api的前缀

但是 这样有一个问题，就是这个/api前缀 会不会出现在我们路由后的IP地址中呢？因为有可能我们微服务提供的接口也是含有/api前缀的

答案是不会的。 但是可以进行配置

```
zuul:
  prefix: /api
  strip-prefix: false
  ignored-services: "*"
  routes:
    mypower:
      serviceId: server-power
      path: /power/**
    myorder:
      serviceId: server-order
      path: /order/**
```

过滤器:

过滤器(filter)是zuul的核心组件 zuul大部分功能都是通过过滤器来实现的。 zuul中定义了4种标准过滤器类型，这些过滤器类型对应于请求的典型生命周期。 PRE：这种过滤器在请求被路由之前调用。可利用这种过滤器实现身份验证、在 集群中选择请求的微服务、记录调试信息等。 ROUTING：这种过滤器将请求路由到微服务。这种过滤器用于构建发送给微服 务的请求，并使用 Apache HttpClient或 Netfilx Ribbon请求微服务 POST:这种过滤器在路由到微服务以后执行。这种过滤器可用来为响应添加标准 的 HTTP Header、收集统计信息和指标、将响应从微服务发送给客户端等。 ERROR：在其他阶段发生错误时执行该过滤器。

如果要编写一个过滤器，则需继承ZuulFilter类 实现其中方法:

```
@Component
public class LogFilter extends ZuulFilter {
```

```
    @Override
    public String filterType() {
        return FilterConstants.ROUTE_TYPE;
    }

    @Override
    public int filterOrder() {
        return FilterConstants.PRE_DECORATION_FILTER_ORDER;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() throws ZuulException {
        RequestContext currentContext = RequestContext.getCurrentContext();
        HttpServletRequest request = currentContext.getRequest();
        String remoteAddr = request.getRemoteAddr();
        System.out.println("访问者IP: "+remoteAddr+"访问地址:"+request.getRequestURI());
        return null;
    }
}
```

由代码可知，自定义的 zuul Filter需实现以下几个方法。

filterType:返回过滤器的类型。有 pre、 route、 post、 error等几种取值，分别对应上文的几种过滤器。

详细可以参考 com.netflix.zuul.ZuulFilter.filterType()中的注释。

filter0rder:返回一个 int值来指定过滤器的执行顺序，不同的过滤器允许返回相同的数字。

shouldFilter：返回一个 boolean值来判断该过滤器是否要执行， true表示执行， false表示不执行。

run：过滤器的具体逻辑。

禁用zuul过滤器 Spring Cloud默认为Zuul编写并启用了一些过滤器，例如DebugFilter、 FormBodyWrapperFilter等，这些过滤器都存放在spring-cloud-netflix-core这个jar包 里，一些场景下，想要禁用掉部分过滤器，该怎么办呢？ 只需在application.yml里设置zuul...disable=true 例如，要禁用上面我们写的过滤器，这样配置就行了：zuul.LogFilter.pre.disable=true

zuul容错与回退

zuul默认是整合了hystrix和ribbon的， 提供降级回退，那如何来使用hystrix呢?

我们自行写一个类，继承FallbackProvider 类 然后重写里面的方法

```
@Override
public String getRoute() {
    return null;
}

@Override
public ClientHttpResponse fallbackResponse(String route, Throwable cause) {
    return null;
}
```

这里 会发现有这2个方法需要重写， 那么如何来写呢？ 我们可以查阅官方文档：

If you would like to provide a default fallback for all routes, you can create a bean of type `FallbackProvider` and have the `getRoute` method return `*` or `null`, as shown in the following example:

```
class MyFallbackProvider implements FallbackProvider {
    @Override
    public String getRoute() {
        return "*";
    }

    @Override
    public ClientHttpResponse fallbackResponse(String route, Throwable throwable) {
        return new ClientHttpResponse() {
            @Override
            public HttpStatus getStatusCode() throws IOException {
                return HttpStatus.OK;
            }

            @Override
            public int getRawStatusCode() throws IOException {
                return 200;
            }

            @Override
            public String getStatusText() throws IOException {
                return "OK";
            }

            @Override
            public void close() {

            }

            @Override
            public InputStream getBody() throws IOException {
                return new ByteArrayInputStream("fallback".getBytes());
            }

            @Override
            public HttpHeaders getHeaders() {
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.APPLICATION_JSON);
                return headers;
            }
        };
    }
}
```

这是官方提供的demo

代码：

```
class MyFallbackProvider implements FallbackProvider {
    @Override
    public String getRoute() {
        //制定为哪个微服务提供回退（这里写微服务名 写*代表所有微服务）
        return "*";
    }

    //此方法需要返回一个ClientHttpResponse对象  ClientHttpResponse是一个接口，具体的回退逻辑要实现此接口
    //route：出错的微服务名      cause：出错的异常对象
  @Override
    public ClientHttpResponse fallbackResponse(String route, final Throwable cause) {
        //这里可以判断根据不同的异常来做不同的处理， 也可以不判断
        //完了之后调用response方法并根据异常类型传入HttpStatus
        if (cause instanceof HystrixTimeoutException) {
            return response(HttpStatus.GATEWAY_TIMEOUT);
```

```
        } else {
            return response(HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }



    private ClientHttpResponse response(final HttpStatus status) {
        //这里返回一个ClientHttpResponse对象 并实现其中的方法，关于回退逻辑的详细，便在下面的方法中
        return new ClientHttpResponse() {
            @Override
            public HttpStatus getStatusCode() throws IOException {
                //返回一个HttpStatus对象 这个对象是个枚举对象， 里面包含了一个status code 和
reasonPhrase信息
                return status;
            }

            @Override
            public int getRawStatusCode() throws IOException {
                //返回status的code  比如 404，500等
                return status.value();
            }

            @Override
            public String getStatusText() throws IOException {
                //返回一个HttpStatus对象的reasonPhrase信息
                return status.getReasonPhrase();
            }

            @Override
            public void close() {
               //close的时候调用的方法， 讲白了就是当降级信息全部响应完了之后调用的方法
            }

            @Override
            public InputStream getBody() throws IOException {
                //吧降级信息响应回前端
                return new ByteArrayInputStream("降级信息".getBytes());
            }

            @Override
            public HttpHeaders getHeaders() {
                //需要对响应报头设置的话可以在此设置
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.APPLICATION_JSON);
                return headers;
            }
        };
    }
}
```