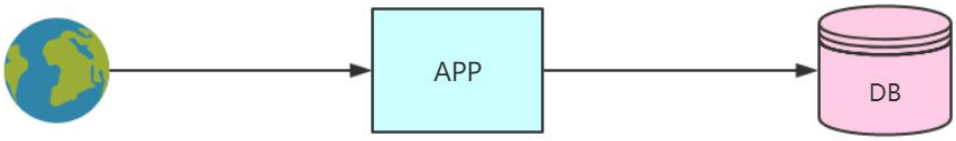
1、架构演变过程

我们最先接触的单体架构，整个系统就只有一个工程，打包往往是打成了 war 包，然后部署到单一 tomcat 上面，这种就是单体架构，如图：



假如系统按照功能划分了，商品模块，购物车模块，订单模块，物流模块等等模块。那么所有模块都会在一个工程里面，这就是单体架构。

单体架构优点

1、结构简单，部署简单

2、所需的硬件资源少

3、节省成本

缺点

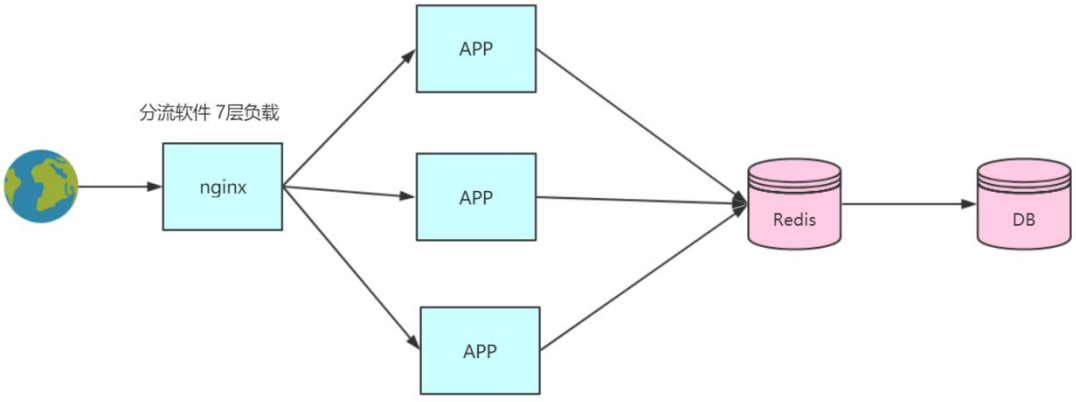
1、版本迭代慢，往往改动一个代码会影响全局

2、不能满足一定并发的访问

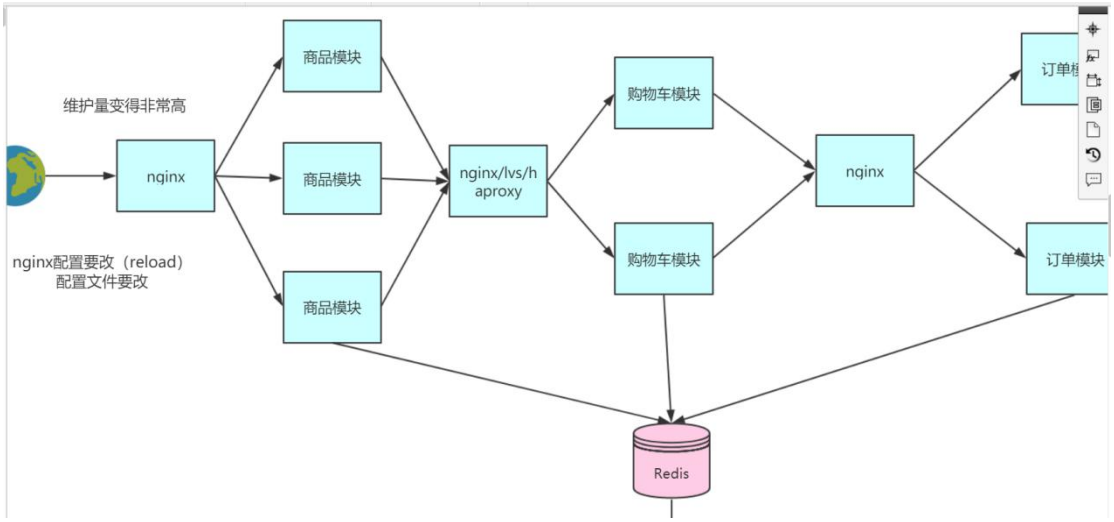
3、代码维护困难，所有代码在一个工程里面，存在被其他人修改的风险

随着业务的拓展，公司的发展，单体架构慢慢的不能满足我们的需求，我们需要对架构进行变动，我们能够想到的最简单的办法就是加机器，对应用横向扩展。

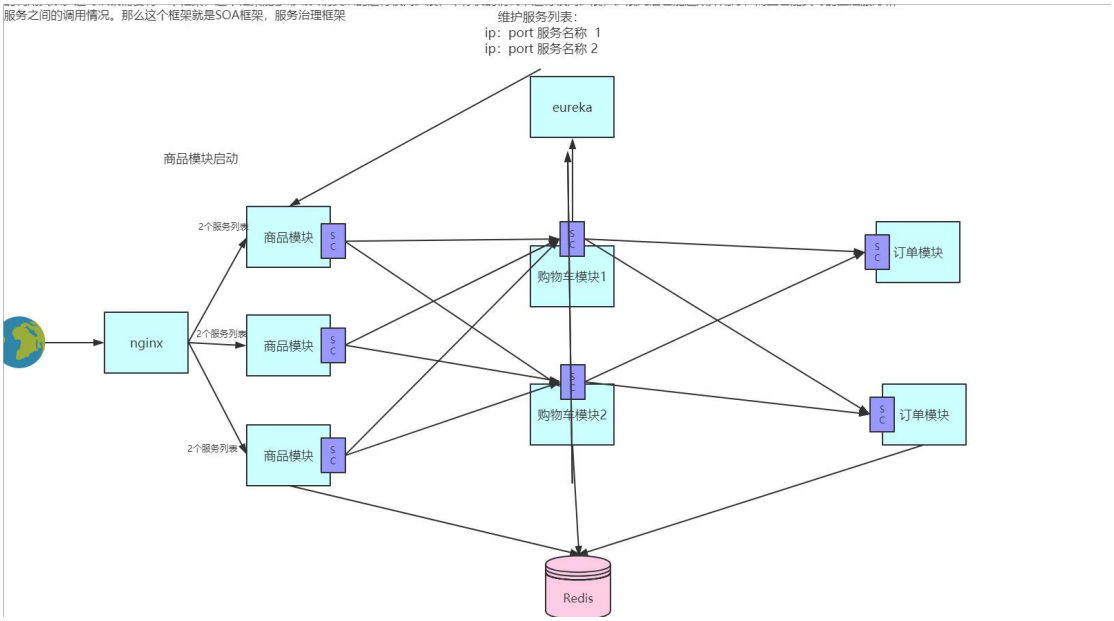
如图：



这种架构貌似暂时解决了我们的问题，但是用户量慢慢增加后，我们只能通过横向加机器来解决，还是会存在版本迭代慢，代码维护困难的问题。而且用户请求往往是读多写少的情况，所以可能真正需要扩容的只是商品模块而已，而现在是整个工程都扩容了，这无形中是一种资源的浪费，因为其他模块可能根本不需要扩容就可以满足需求。所以我们有必要对整个工程按照模块进行拆分，拆分后的架构图如下：



模块拆分后，模块和模块之间是需要通过接口调用的方式进行通信，模块和模块之间通过分流软件进行负载均衡。这个架构解决前面的资源浪费问题和代码管理问题，因为我们是对系统拆分了，各个模块都有单独的工程，比如我修改商品模块，就不需要担心会不会影响购物车模块。但是这种架构扩展非常麻烦，一旦需要横向加机器，或者减机器都需要修改 nginx 配置，一旦机器变多了以后，nginx 的配置量就是一个不能完成的工作。OK，这时候 SOA 服务治理框架就应运而生，架构图如下：



基于注册中心的 SOA 框架，扩展是非常方便的，因为不需要维护分流工具，但我们启动应用的时候就会把服务通过 http 的方式注册到注册中心。

在 SOA 框架中一般会有三种角色：1、注册中心 2、服务提供方 3、服务消费方

1、注册中心

在注册中心维护了服务列表

2、服务提供方

服务提供方启动的时候会把自己注册到注册中心

3、服务消费方

服务消费方启动的时候，把获取注册中心的服务列表，然后调用的时候从这个服务列表中选择某一个去调用。

微服务工程的特点：
1、扩展灵活
2、每个应用都规模不大
3、服务边界清晰，各司其职
4、打包应用变多，往往需要借助 CI 持续集成工具

2、搭建简单的微服务工程

1、注册中心搭建
Springcloud 中，我们选择 eureka 作为注册中心，springcloud 工程是基于 springboot 工程的。
pom.xml 中 jar 包依赖：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.2.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Springcloud 的版本

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
    <spring-cloud.version>Hoxton.SR1</spring-cloud.version>
</properties>
```

Eureka 服务端启动器导入

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

Springcloud 的依赖仓库导入

```
<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>
```

Application.properties 配置文件

```
server.port=8763
eureka.instance.hostname=localhost

#是否注册到eureka
eureka.client.registerWithEureka=false
#是否从eureka中拉取注册信息
eureka.client.fetchRegistry=false
##暴露eureka服务的地址
eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${server.port}/eurel
#自我保护模式，当出现出现网络分区、eureka在短时间内丢失过多客户端时，会进入自我保护模式，即一个服务长时间没有
eureka.server.enable-self-preservation=true

#eureka server清理无效节点的时间间隔，默认60000毫秒，即60秒
#eureka.server.eviction-interval-timer-in-ms=60
```

配置文件属性后续会详解

启动类

```
@SpringBootApplication
//开启eurekaServer服务注册功能
@EnableEurekaServer
public class EurekaApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaApplication.class,args);
    }
}
```

2、服务提供方

Pom 的 jar 包依赖，其他都跟 eureka 服务端是一样的，只是服务提供方要把服务注册到 eureka 服务端，所以服务提供方就是 eureka 的客户端，所以需要导入 eureka 客户端的启动器。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

bootstrap.properties

```
spring.application.name=micro-order
server.port=8084

eureka.client.serviceUrl.defaultZone=http\://localhost\:8763/eureka/
```

启动类

```
@SpringBootApplication

//开启eureka客户端功能
@EnableEurekaClient
public class MicroOrderApplication {
    public static void main(String[] args) {
        SpringApplication.run(MicroOrderApplication.class,args);
    }
}
```

3、服务消费方

pom 和属性配置文件基本上差不多，消费要负责调用服务提供方，所以需要调用客户端

```java
@SpringBootApplication

@EnableEurekaClient
public class MicroWebApplication {

    @Bean
    //负载均衡注解
    @LoadBalanced
    RestTemplate restTemplate() {
        return new RestTemplate();
    }

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

服务调用的时候就根据服务提供方的服务名称来调用的

```java
@Slf4j
@Service
@Scope(proxyMode = ScopedProxyMode.INTERFACES)
public class UserServiceImpl implements UserService {

    private AtomicInteger s = new AtomicInteger();
    private AtomicInteger f = new AtomicInteger();

    public static String SERVIER_NAME = "micro-order";

    @Autowired
    private RestTemplate restTemplate;

    @Override
    public String queryContents() {
        s.incrementAndGet();
        String results = restTemplate.getForObject( url: "http://"
                + SERVIER_NAME + "/queryUser", String.class);
        return results;
    }
}
```

服务提供方的名称，再加上服务提供方的接口名就可以完成调用了。
服务提供方和服务消费启动的时候都会往服务注册中心注册服务，eureka 服务端也可以通过界面查看到服务注册情况：

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MICRO-ORDER	n/a (3)	(3)	UP (3) - localhost:micro-order:8086 , localhost:micro-order:8085 , localhost:micro-order:8084
MICRO-WEB	n/a (1)	(1)	UP (1) - localhost:micro-web:8083

General Info

3、Eureka 用户认证

连接到 eureka 的时候需要带上连接的用户名和密码
eureka 服务端改造

添加启动器

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

代码配置，关闭 csrf 验证

```
@EnableWebSecurity
public class WebSecurityConfigurer extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        //关闭csrf
        http.csrf().disable();
        //开启认证：URL格式登录必须是httpBasic
        http.authorizeRequests().anyRequest().authenticated().and().httpBasic();
    }
}
```

Application.properties 配置

开启 basic 校验,设置登录用户名密码

```
security.basic.enabled=true
spring.security.user.name=admin
spring.security.user.password=admin
```

Eureka 客户端改造

跟 eureka 连接的时候要带上用户名密码

```
eureka.client.serviceUrl.defaultZone=http://admin:admin@localhost:8763/eureka/
```

4、服务续约保活

当客户端启动想 eureka 注册了本身服务列表后，需要隔段时间发送一次心跳给 eureka 服务端来证明自己还活着，当 eureka 收到这个心跳请求后才会知道客户端还活着，才会维护该客户端的服务列表信息。一旦因为某些原因导致客户端没有按时发送心跳给 eureka 服务端，这时候 eureka 可能会认为你这个客户端已经挂了，它就有可能把该服务从服务列表中删除掉。

有关续约保活的配置

客户端配置

```
#服务续约，心跳的时间间隔
eureka.instance.lease-renewal-interval-in-seconds=30
#如果从前一次发送心跳时间起，90 秒没接受到新的心跳，讲剔除服务
eureka.instance.lease-expiration-duration-in-seconds=90
#表示 eureka client 间隔多久去拉取服务注册信息，默认为 30 秒
eureka.client.registry-fetch-interval-seconds=30
```

服务端配置

```
#自我保护模式，当出现出现网络分区、eureka 在短时间内丢失过多客户端时，
会进入自我保护模式，即一个服务长时间没有发送心跳，eureka 也不会将其删
除，默认为 true
eureka.server.enable-self-preservation=true
#Eureka Server 在运行期间会去统计心跳失败比例在 15 分钟之内是否低
于 85%，如果低于 85%，Eureka Server 会将这些实例保护起来
eureka.server.renewal-percent-threshold=0.85
#eureka server 清理无效节点的时间间隔，默认 60000 毫秒，即 60 秒
eureka.server.eviction-interval-timer-in-ms=60000
```

5、Eureka 健康检测

Eureka 默认的健康检测只是你校验服务连接是否是 UP 还是 DOWN 的，然后客户端只会调用状态为 UP 状态的服务，但是有的情况下，虽然服务连接是好的，但是有可能这个服务的某些接口不是正常的，可能由于需要连接 Redis，mongodb 或者 DB 有问题导致接口调用失败，所以理论上服务虽然能够正常调用，但是它不是一个健康的服务。所以我们就有必要对这种情况做自定义健康检测。

Application.properties 配置

开启健康检测

```
#健康检测
eureka.client.healthcheck.enabled=true
```

自定义健康检测代码

```
@Configuration
public class MicroWebHealthIndicator implements HealthIndicator {
    @Override
    public Health health() {
        if(UserController.canVisitDb) {
            return new Health.Builder(Status.UP).build();
        } else {
            return new Health.Builder(Status.DOWN).build();
        }
    }
}
```

我们可以在 health 方法里面去连接数据库，如果连接异常了则返回 DOWN，如果没异常则方法 UP，这个 health 方法是线程去掉的，隔一段时间掉一次

检测检测依赖的 jar 包

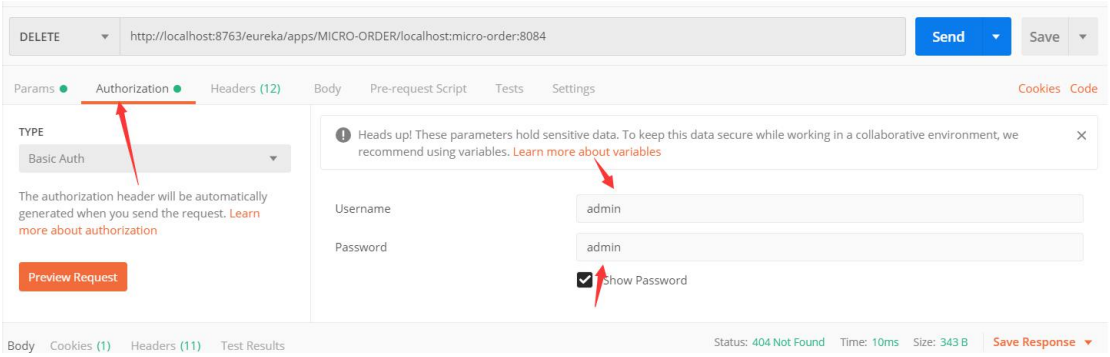
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

6、服务下线

比如有些情况是服务主机意外宕机了，也就意味着服务没办法给 eureka 心跳信息了，但是 eureka 在没有接受到心跳的情况下依赖维护该服务 90s，在这 90s 之内可能会有客户端调用到该服务，这就可能会导致调用失败。所以我们必须要有一个机制能手动的立马把宕机的服务从 eureka 服务列表中清除掉，避免被服务调用方调用到。

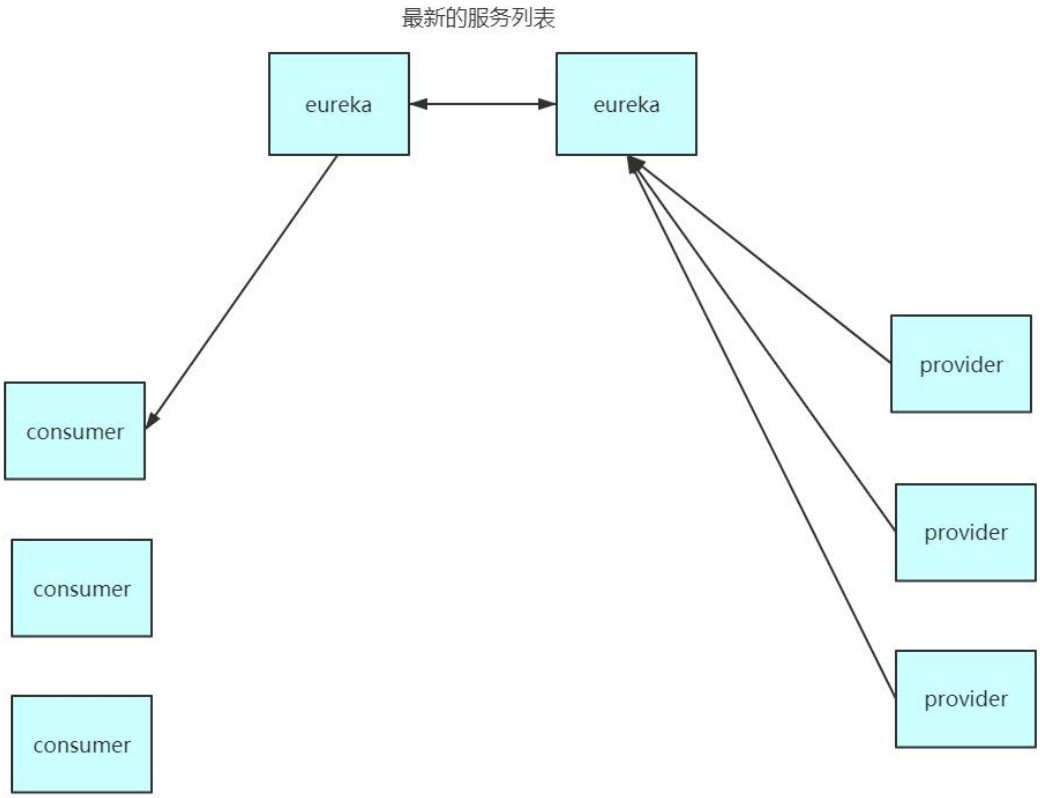
调用服务下线的接口：
这个接口是调用 eureka 服务端的接口

http://localhost:8763/eureka/apps/MICRO-ORDER/localhost:micro-order:8084



7、Eureka 高可用

Eureka 热备份的架构图如下：



整个微服务中存在多个 eureka 服务，每个 eureka 服务都是相互复制的，会把客户端注册进来的服务复制到 eureka 集群中的其他节点里面来。其实简单来说就是 eureka 每个节点相互复制。

具体配置如下：

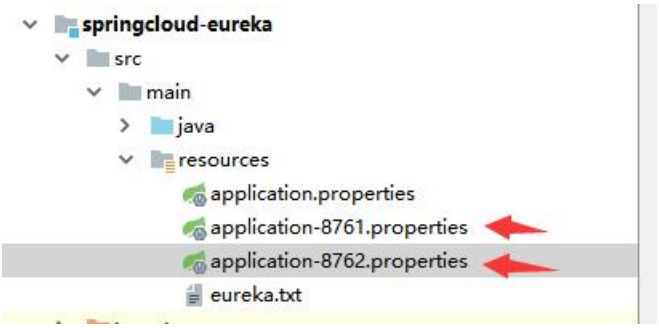
端口为 8761 的 eureka 服务端把自己注册到 8762 的 eureka 服务端

```
server.port=8761
eureka.instance.hostname=Eureka8761
#是否注册到eureka
eureka.client.registerWithEureka=true
#是否从eureka中拉取注册信息
eureka.client.fetchRegistry=true
##暴露eureka服务的地址
eureka.client.serviceUrl.defaultZone=http://admin:admin@Eureka8762.com:8762/eureka/

#自我保护模式，当出现出现网络分区、eureka在短时间内丢失过多客户端时，会进入自我保护模式，即一个服务长时间没有发送心跳，eureka也不会将
eureka.server.enable-self-preservation=false

security.basic.enabled=true
spring.security.user.name=admin
spring.security.user.password=admin
```

同样的道理，8762 注册到 8761

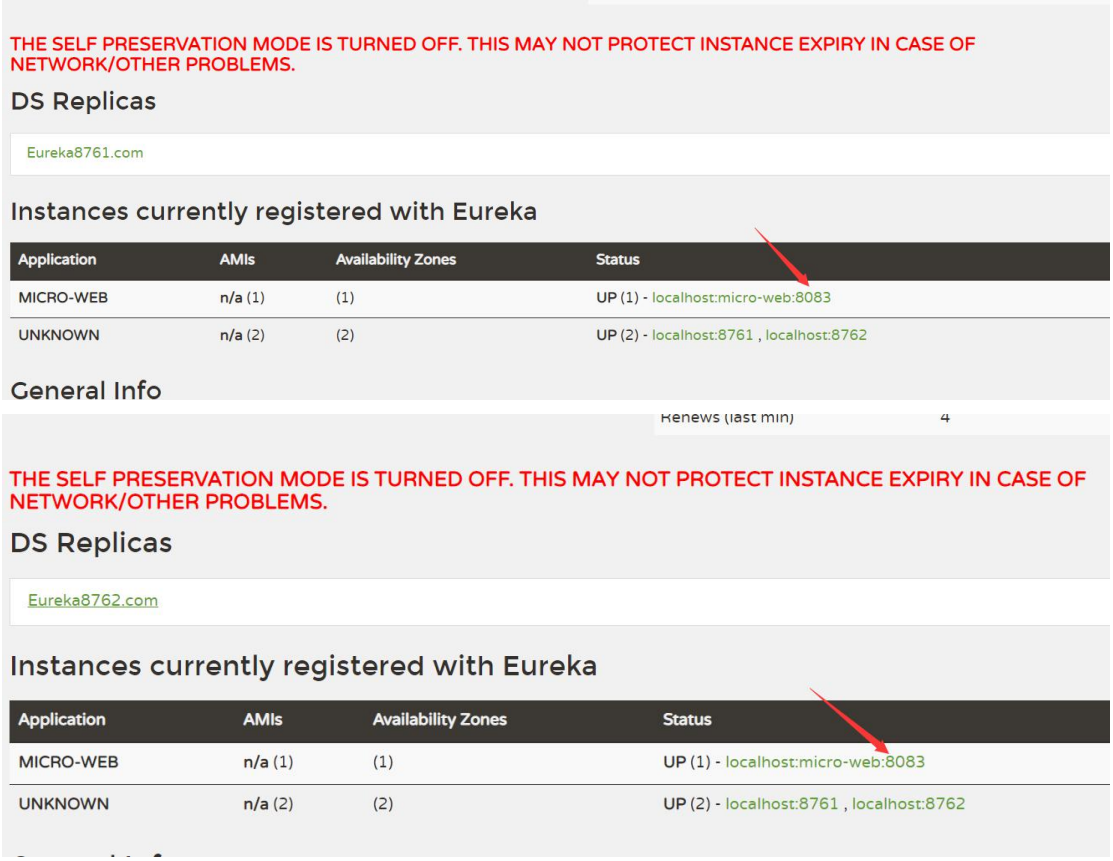
```
server.port=8762
eureka.instance.hostname=Eureka8762
#是否注册到eureka
eureka.client.registerWithEureka=true
#是否从eureka中拉取注册信息
eureka.client.fetchRegistry=true
##暴露eureka服务的地址
eureka.client.serviceUrl.defaultZone=http://admin:admin@Eureka8761.com:8761/eureka/

#自我保护模式，当出现出现网络分区、eureka在短时间内丢失过多客户端时，会进入自我保护模式，即一个服务长时间没有发送心跳，eureka也不会将其
eureka.server.enable-self-preservation=false

security.basic.enabled=true
spring.security.user.name=admin
spring.security.user.password=admin
```

配置存在在两个配置文件中



启动的时候按照指定配置文件启动

java -jar springcloud-eureka.jar --spring.profiles.active=8761



8、Ribbon API

Ribbon 是一个独立的组件，是用来进行远程接口调用的，代码如下

```java
@Slf4j
@Service
@Scope(proxyMode = ScopedProxyMode.INTERFACES)
public class UserServiceImpl implements UserService {

    private AtomicInteger s = new AtomicInteger();
    private AtomicInteger f = new AtomicInteger();

    public static String SERVIER_NAME = "micro-order";

    @Autowired
    private RestTemplate restTemplate;

    @Override
    public String queryContents() {
        s.incrementAndGet();
        String results = restTemplate.getForObject( url: "http://"
                + SERVIER_NAME + "/queryUser", String.class);
        return results;
    }
}
```

通过 getForObject 方法可以掉到用 micro-order 服务的，queryUser 接口。然后在调用期间会存在负载均衡，micro-order 服务对应有几个服务实例就会根据负载均衡算法选择某一个去调用。

Get 请求

getForEntity：此方法有三种重载形式，分别为：

getForEntity(String url, Class<T> responseType)

getForEntity(String url, Class<T> responseType, Object... uriVariables)

getForEntity(String url, Class<T> responseType, Map<String, ?> uriVariables)

getForEntity(URI url, Class<T> responseType)

注意：此方法返回的是一个包装对象 ResponseEntity<T>其中 T 为 responseType 传入类型，想拿到返回类型需要使用这个包装类对象的 getBody()方法

getForObject:此方法也有三种重载形式，这点与 getForEntity 方法相同：

getForObject(String url, Class<T> responseType)

getForObject(String url, Class<T> responseType, Object... uriVariables)

getForObject(String url, Class<T> responseType, Map<String, ?> uriVariables)

getForObject(URI url, Class<T> responseType)

注意：此方法返回的对象类型为 responseType 传入类型

Post 请求

post 请求和 get 请求都有*ForEntity 和*ForObject 方法，其中参数列表有些不同，除了这两个方法外，还有一个 postForLocation 方法，其中 postForLocation 以 post 请求提交资源，并返回新资源的 URI

postForEntity：此方法有三种重载形式，分别为：

postForEntity(String url, Object request, Class<T> responseType, Object... uriVariables)

postForEntity(String url, Object request, Class<T> responseType, Map<String, ?> uriVariables)

postForEntity(URI url, Object request, Class<T> responseType)

注意：此方法返回的是一个包装对象 ResponseEntity<T>其中 T 为 responseType 传入类型，想拿到返回类型需要使用这个包装类对象的 getBody()方法

postForObject:此方法也有三种重载形式，这点与 postForEntity 方法相同：

postForObject(String url, Object request, Class<T> responseType, Object... uriVariables)

postForObject(String url, Object request, Class<T> responseType, Map<String, ?> uriVariables)

postForObject(URI url, Object request, Class<T> responseType)

注意：此方法返回的对象类型为 responseType 传入类型

postForLocation:此方法中同样有三种重载形式，分别为：

postForLocation(String url, Object request, Object... uriVariables)

postForLocation(String url, Object request, Map<String, ?> uriVariables)

postForLocation(URI url, Object request)

注意：此方法返回的是新资源的 URI，相比 getForEntity、getForObject、postForEntity、postForObject 方法不同的是这个方法中无需指定返回类型，因为返回类型就是 URI，通过 Object... uriVariables、Map<String, ?> uriVariables 进行传参依旧需要占位符，参看 postForEntity 部分代码

9、负载均衡算法

```
@Bean
public IRule ribbonRule() {
    //线性轮训
    new RoundRobinRule();
    //可以重试的轮训
    new RetryRule();
    //根据运行情况来计算权重
    new WeightedResponseTimeRule();
    //过滤掉故障实例，选择请求数最小的实例
    new BestAvailableRule();
    return new RandomRule();
}
```

```
//线性轮训
new RoundRobinRule();
```

```
//可以重试的轮训
new RetryRule();
//根据运行情况来计算权重
new WeightedResponseTimeRule();
//过滤掉故障实例，选择请求数最小的实例
new BestAvailableRule();
//随机
new RandomRule();
```

10、Ribbon 配置

Application.properties 配置

```
#点对点直连测试配置
# 关闭 ribbon 访问注册中心 Eureka Server 发现服务，但是服务依旧会注册。
#true 使用 eureka false 不使用
ribbon.eureka.enabled=true
spring.cloud.loadbalancer.retry.enabled=true
#指定调用的节点
micro-order.ribbon.listOfServers=localhost:8001
#单位 ms ,请求连接超时时间
micro-order.ribbon.ConnectTimeout=1000
#单位 ms ,请求处理的超时时间
micro-order.ribbon.ReadTimeout=2000

micro-order.ribbon.OkToRetryOnAllOperations=true
#切换实例的重试次数
micro-order.ribbon.MaxAutoRetriesNextServer=2
#对当前实例的重试次数 当 Eureka 中可以找到服务，但是服务连不上时将会重试
micro-order.ribbon.MaxAutoRetries=2
```

```
micro-order.ribbon.NFLoadBalancerRuleClassName=com.netflix.loadbalancer.RandomRule
micro-order.ribbon.NFLoadBalancerPingClassName=com.netflix.loadbalancer.PingUrl
```

代码配置

使用@RibbonClients 加载配置

```
/*
* 这个是针对 micro-order服务的 ribbon配置
* */
@Configuration
@RibbonClients(value = {
        @RibbonClient(name = "micro-order",configuration = RibbonLoadBalanceMicroOrderConfig.class)
})
public class LoadBalanceConfig {

}
```

这个配置类只针对 micro-order 服务，微服务系统里面有很多服务，这就可以区别化配置。

配置 configuration 配置类的时候，一定要注意，配置类不能陪@ComponentScan 注解扫描到，如果被扫描到了则该配置类就是所有服务共用的配置了。

```
//    @RibbonClientName
    private String name = "micro-order";

    @Bean
    @ConditionalOnClass
    public IClientConfig defaultClientConfigImpl() {
        DefaultClientConfigImpl config = new DefaultClientConfigImpl();
        config.loadProperties(name);
        config.set(CommonClientConfigKey.MaxAutoRetries,2);
        config.set(CommonClientConfigKey.MaxAutoRetriesNextServer,2);
        config.set(CommonClientConfigKey.ConnectTimeout,2000);
        config.set(CommonClientConfigKey.ReadTimeout,4000);
        config.set(CommonClientConfigKey.OkToRetryOnAllOperations,true);
        return config;
    }
    /*
    * 判断服务是否存活
    * */
//    @Bean
    public IPing iPing() {
        //这个实现类会去调用服务来判断服务是否存活
        return new PingUrl();
    }

    @Bean
    public IRule ribbonRule() {
        //线性轮训
        new RoundRobinRule();
        //可以重试的轮训
        new RetryRule();
        //根据运行情况来计算权重
        new WeightedResponseTimeRule();
        //过滤掉故障实例，选择请求数最小的实例
        new BestAvailableRule();
        return new RandomRule();
    }
```

11、Ribbon 单独使用

Ribbon 是一个独立组件，可以脱离 springcloud 使用的

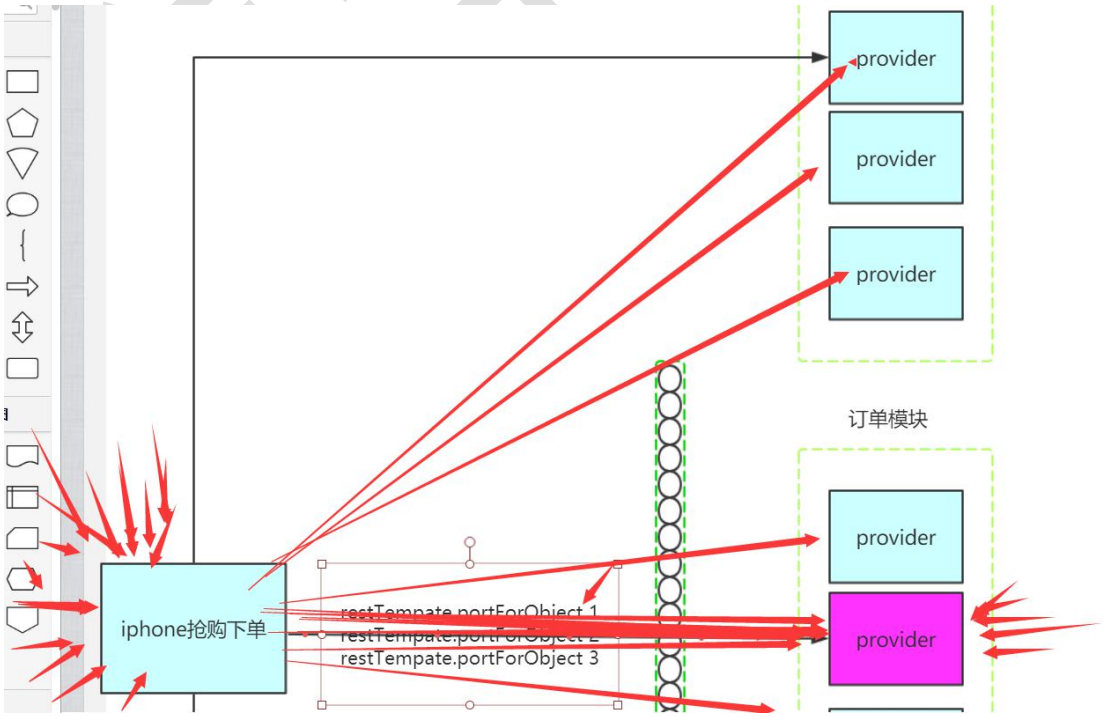
```
/*
* ribbon作为调用客户端，可以单独使用
* */
@Test
public void test1() {
    try {
        ConfigurationManager.getConfigInstance().setProperty("myClients.ribbon.listOfServers","localhost:8001,localhost:8002");
        RestClient client = (RestClient)ClientFactory.getNamedClient("myClients");
        HttpRequest request = HttpRequest.newBuilder().uri(new URI( str: "/user/queryContent")).build();

        for (int i = 0; i < 10; i++) {
            HttpResponse httpResponse = client.executeWithLoadBalancer(request);
            String entity = httpResponse.getEntity(String.class);
            System.out.println(entity);
        }
    } catch (URISyntaxException e) {
        e.printStackTrace();
    } catch (ClientException e) {
        e.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

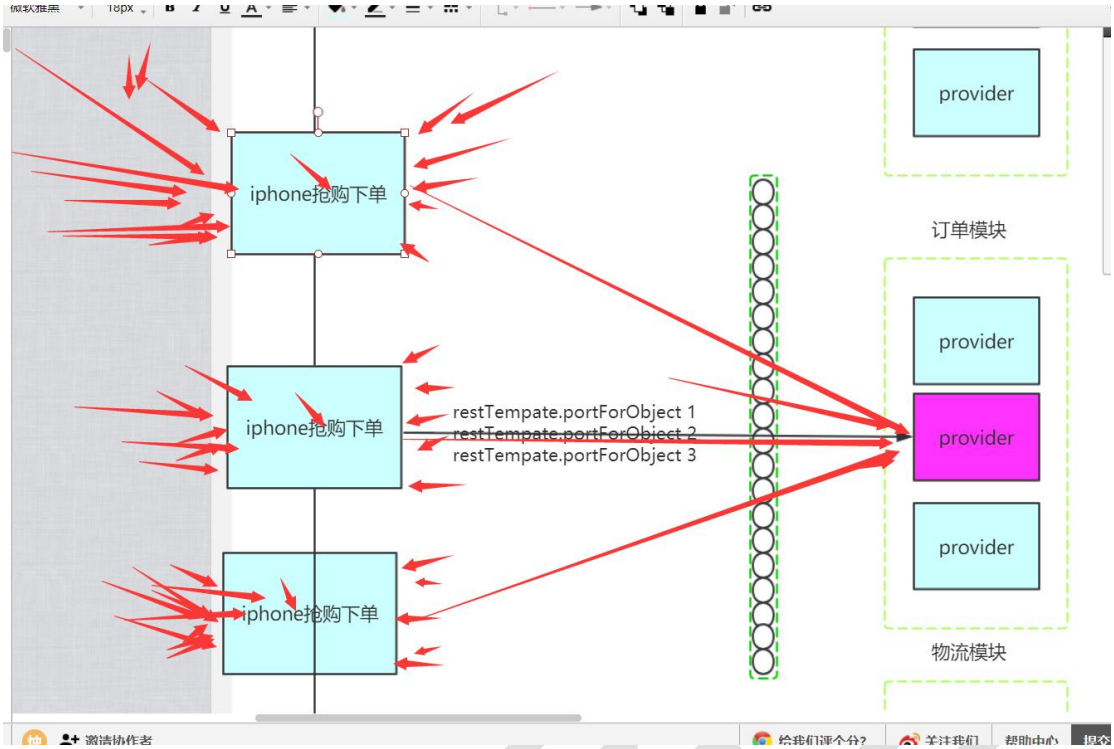
需要依赖两个 jar

```
com.netflix.ribbon:ribbon-core:2.3.0
com.netflix.ribbon:ribbon-httpclient:2.3.0
```

12、服务雪崩

雪崩是系统中的蝴蝶效应导致其发生的原因多种多样，有不合理的容量设计，或者是高并发下某一个方法响应变慢，亦或是某台机器的资源耗尽。从源头上我们无法完全杜绝雪崩源头的发生，但是雪崩的根本原因来源于服务之间的强依赖，所以我们可以提前评估。当整个微服务系统中，有一个节点出现异常情况，就有可能在高并发的情况下出现雪崩，导致调用它的上游系统出现响应延迟，响应延迟就会导致 tomcat 连接本耗尽，导致该服务节点不能正常的接收到正常的情况，这就是服务雪崩行为。





13、服务隔离

如果整个系统雪崩是由于一个接口导致的，由于这一个接口响应不及时导致问题，那么我们就有必要对这个接口进行隔离，就是只允许这个接口最多能接受多少的并发，做了这样的限制后，该接口的主机就会空余线程出来接收其他的情况，不会被哪个坏了的接口占用满。Hystrix 就是一个不错的服务隔离框架。

Hystrix 的导入

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

启动类开启 hystrix 功能

```
@SpringBootApplication(scanBasePackages = {"com.xiangxue.jack"})
//注册到eureka
@EnableEurekaClient
//开启断路器功能
@EnableCircuitBreaker
//开启feign支持，clients指定哪个类开启feign
@EnableFeignClients(clients = {StudentService.class,TeacherServiceFeign.class})
//开启重试功能
@EnableRetry
public class MicroWebApplication {
```

代码使用

```java
@HystrixCommand
@Override
public String queryTicket() {
    return "queryTicket";
}
```

14、Hystrix 服务隔离策略

1、线程池隔离

THREAD 线程池隔离策略　独立线程接收请求　　默认的

默认采用的就是线程池隔离

代码配置

```java
@HystrixCommand(fallbackMethod = "queryContentsFallback",
        commandKey = "queryContents",
        groupKey = "querygroup-one",
        commandProperties = {
                @HystrixProperty(name = "execution.isolation.semaphore.maxConcurrentRequests",value = "100"),
                @HystrixProperty(name = "execution.isolation.strategy", value = "THREAD"),
                @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "1000000000"
        },
        threadPoolKey = "queryContentshystrixJackpool", threadPoolProperties = {
          @HystrixProperty(name = "coreSize", value = "100")
})
@Override
public List<ConsultContent> queryContents() {
    log.info(Thread.currentThread().getName() + "========queryContents=========");
    s.incrementAndGet();
    List<ConsultContent> results = restTemplate.getForObject( url: "http://"
            + SERVIER_NAME + "/user/queryContent", List.class);
    return results;
}
```

默认线程池中有 10 个线程，可以配置线程池中线程大小

```java
@HystrixCommand(fallbackMethod = "queryContentsFallback",
        commandKey = "queryContents",
        groupKey = "querygroup-one",
        commandProperties = {
                @HystrixProperty(name = "execution.isolation.semaphore.maxConcurrentRequests",value = "100"),
                @HystrixProperty(name = "execution.isolation.strategy", value = "THREAD"),
                @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "1000000000"
        },
        threadPoolKey = "queryContentshystrixJackpool", threadPoolProperties = {
        @HystrixProperty(name = "coreSize", value = "100")
})
@Override
public List<ConsultContent> queryContents() {
    log.info(Thread.currentThread().getName() + "========queryContents=========");
    s.incrementAndGet();
    List<ConsultContent> results = restTemplate.getForObject( url: "http://"
            + SERVIER_NAME + "/user/queryContent", List.class);
    return results;
}
```

线程池隔离策略，hystrix 是会单独创建线程的，单元测试如下：

```
rverList@4b7586c7
t.MyTest            : Thread-7==>[{id=1, itemIndex=1, content=是否经常头昏、眼花
t.MyTest            : Thread-11==>[{id=1, itemIndex=1, content=是否经常头昏、眼
t.MyTest            : Thread-12==>[{id=1, itemIndex=1, content=是否经常头昏、眼
t.MyTest            : Thread-9==>[{id=1, itemIndex=1, content=是否经常头昏、眼花
t.MyTest            : Thread-5==>[{id=1, itemIndex=1, content=是否经常头昏、眼花
t.MyTest            : Thread-10==>[{id=1, itemIndex=1, content=是否经常头昏、眼
t.MyTest            : Thread-15==>[{id=1, itemIndex=1, content=是否经常头昏、眼
t.MyTest            : Thread-14==>[{id=1, itemIndex=1, content=是否经常头昏、眼
t.MyTest            : Thread-13==>[{id=1, itemIndex=1, content=是否经常头昏、眼
t.MyTest            : Thread-6==>[{id=1, itemIndex=1, content=是否经常头昏、眼花
```

```
: Thread-11==>null
erviceImpl  : hystrix-queryContentshystrixJackpool-4========queryContents=========
erviceImpl  : hystrix-queryContentshystrixJackpool-6========queryContents=========
erviceImpl  : hystrix-queryContentshystrixJackpool-2========queryContents=========
erviceImpl  : hystrix-queryContentshystrixJackpool-3========queryContents=========
erviceImpl  : hystrix-queryContentshystrixJackpool-1========queryContents=========
erviceImpl  : hystrix-queryContentshystrixJackpool-8========queryContents=========
erviceImpl  : hystrix-queryContentshystrixJackpool-9========queryContents=========
erviceImpl  : hystrix-queryContentshystrixJackpool-10========queryContents=========
erviceImpl  : hystrix-queryContentshystrixJackpool-7========queryContents=========
erviceImpl  : hystrix-queryContentshystrixJackpool-5========queryContents=========
ledTimer    : Shutdown hook installed for: NFLoadBalancer-PingTimer-micro-order
```

可以看到，用户线程和业务类中的线程是不一样的

2、信号量隔离

信号量隔离是采用一个全局变量来控制并发量，一个请求过来全局变量加 1，单加到跟配置中的大小相等是就不再接受用户请求了。

代码配置

```
@HystrixCommand(fallbackMethod = "queryContentsFallback",
        commandKey = "queryContents",
        groupKey = "querygroup-one",
        commandProperties = {
                @HystrixProperty(name = "execution.isolation.semaphore.maxConcurrentRequests",value = "100"),
                @HystrixProperty(name = "execution.isolation.strategy", value = "SEMAPHORE"),
                @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "1000000000"
        },
        threadPoolKey = "queryContentshystrixJackpool", threadPoolProperties = {
        @HystrixProperty(name = "coreSize", value = "100")
})
@Override
public List<ConsultContent> queryContents() {
    log.info(Thread.currentThread().getName() + "========queryContents=========");
    s.incrementAndGet();
    List<ConsultContent> results = restTemplate.getForObject( url: "http://"
            + SERVIER_NAME + "/user/queryContent", List.class);
    return results;
}
```

execution.isolation.semaphore.maxConcurrentRequests

这参数是用来控制信号量隔离级别的并发大小的。

单元测试

```
actingServerList@6aab1538
ack.test.MyTest            : Thread-11==>[{id=1, itemIndex=1, content=是否经
ack.test.MyTest            : Thread-12==>[{id=1, itemIndex=1, content=是否经
ack.test.MyTest            : Thread-13==>[{id=1, itemIndex=1, content=是否经
ack.test.MyTest            : Thread-10==>[{id=1, itemIndex=1, content=是否经
ack.test.MyTest            : Thread-7==>[{id=1, itemIndex=1, content=是否经常
ack.test.MyTest            : Thread-15==>[{id=1, itemIndex=1, content=是否经
ack.test.MyTest            : Thread-5==>[{id=1, itemIndex=1, content=是否经常
ack.test.MyTest            : Thread-8==>[{id=1, itemIndex=1, content=是否经常
ack.test.MyTest            : Thread-14==>[{id=1, itemIndex=1, content=是否经
ack.test.MyTest            : Thread-9==>[{id=1, itemIndex=1, content=是否经常
```

```
e.UserServiceImpl   : Thread-12========queryContents=========
e.UserServiceImpl   : Thread-8========queryContents=========
e.UserServiceImpl   : Thread-11========queryContents=========
e.UserServiceImpl   : Thread-9========queryContents=========
e.UserServiceImpl   : Thread-10========queryContents=========
e.UserServiceImpl   : Thread-14========queryContents=========
e.UserServiceImpl   : Thread-7========queryContents=========
e.UserServiceImpl   : Thread-5========queryContents=========
e.UserServiceImpl   : Thread-15========queryContents=========
e.UserServiceImpl   : Thread-13========queryContents=========
```

可以看到，单元测试中的线程和业务类中的线程是一样的，没有单独开启线程。

15、Hystrix 服务降级

服务降级是对服务调用过程的出现的异常的友好封装，当出现异常时，我们不希望直接把异常原样返回，所以当出现异常时我们需要对异常信息进行包装，抛一个友好的信息给前端。

Hystrix 降级的使用比较简单
代码示例

```
@HystrixCommand(fallbackMethod = "queryContentsFallback",
        commandKey = "queryContents",
        groupKey = "querygroup-one",
        commandProperties = {
                @HystrixProperty(name = "execution.isolation.semaphore.maxConcurrentRequests",value = "100"),
                @HystrixProperty(name = "execution.isolation.strategy", value = "SEMAPHORE"),
                @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "1000000000")
        },
        threadPoolKey = "queryContentshystrixJackpool", threadPoolProperties = {
        @HystrixProperty(name = "coreSize", value = "100")
})
@Override
public List<ConsultContent> queryContents() {
    log.info(Thread.currentThread().getName() + "========queryContents=========");
    s.incrementAndGet();
    List<ConsultContent> results = restTemplate.getForObject( url: "http://"
            + SERVIER_NAME + "/user/queryContent", List.class);
    return results;
}
```

指定降级方法
定义降级方法，降级方法的返回值和业务方法的方法值要一样

```
public List<ConsultContent> queryContentsFallback() {
    f.incrementAndGet();
    log.info("================queryContentsFallback=================");

    return null;
}
```

16、Hystrix 数据监控

Hystrix 进行服务熔断时会对调用结果进行统计，比如超时数、bad 请求数、降级数、异常数等等都会有统计，那么统计的数据就需要有一个界面来展示，hystrix-dashboard 就是这么一个展示 hystrix 统计结果的服务。

Dashboard 工程搭建

pom.xml

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
</dependency>
```

启动类

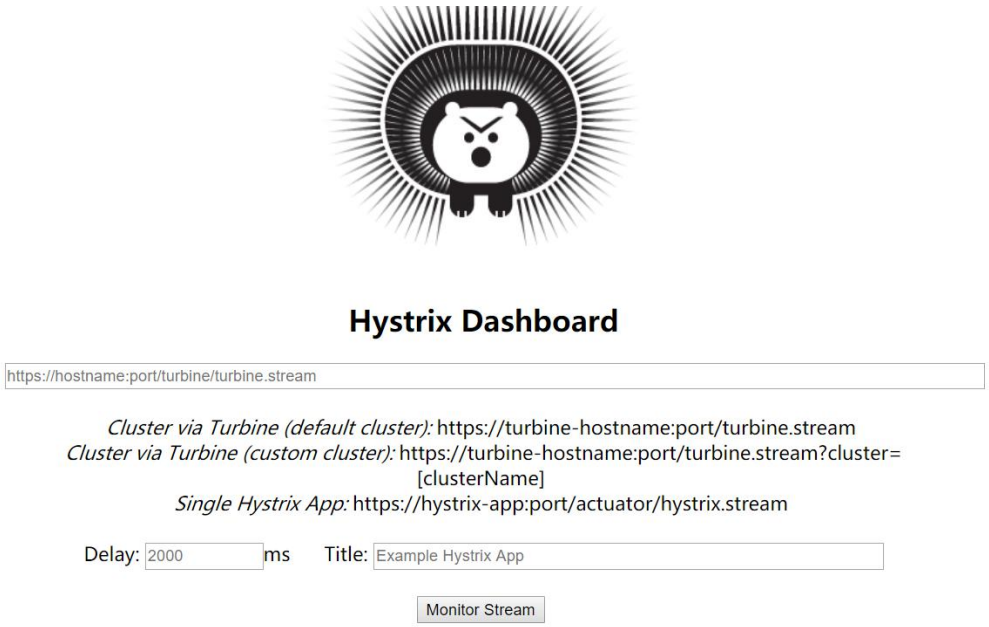
```
/*
* 监控界面：http://localhost:9990/hystrix
* 需要监控的端点（使用了hystrix组件的端点）：http://localhost:8083/actuator/hystrix.stream
 *
 * */
@SpringBootApplication
@EnableHystrixDashboard
public class HystrixDashboardApplication {
    public static void main(String[] args) {
        SpringApplication.run(HystrixDashboardApplication.class,args);
//        new SpringApplicationBuilder(HystrixDashboardApplication.class).web(true).run(args);
    }
}
```

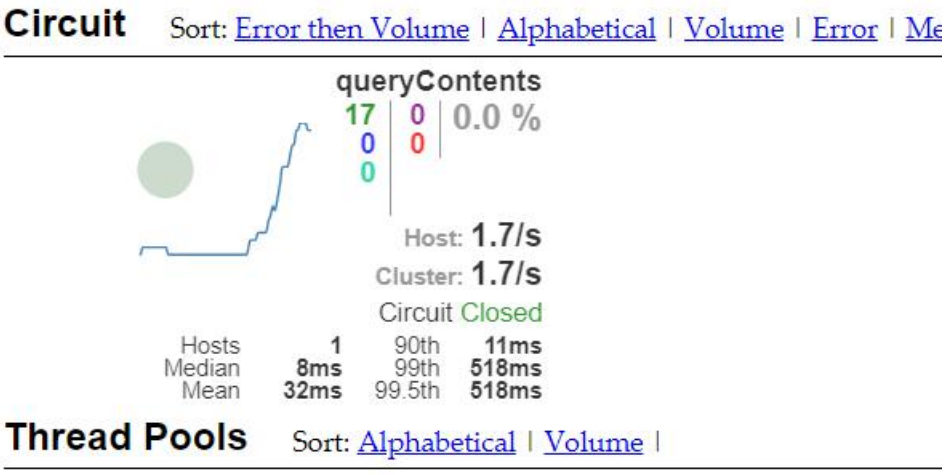
Dashboadr 界面

http://localhost:9990/hystrix

然后在界面中输入需要监控的端点 url：

http://localhost:8083/actuator/hystrix.stream





17、Hystrix 熔断

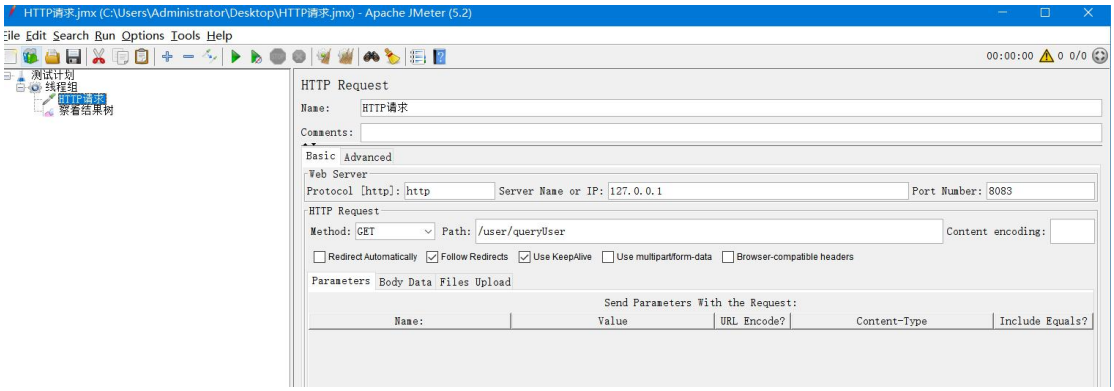
熔断就像家里的保险丝一样，家里的保险丝一旦断了，家里就没点了，家里用电器功率高了就会导致保险丝端掉。在我们 springcloud 领域也可以这样理解，如果并发高了就可能触发 hystrix 的熔断。

熔断发生的三个必要条件：
1、有一个统计的时间周期，滚动窗口
相应的配置属性
metrics.rollingStats.timeInMilliseconds
默认 10000 毫秒

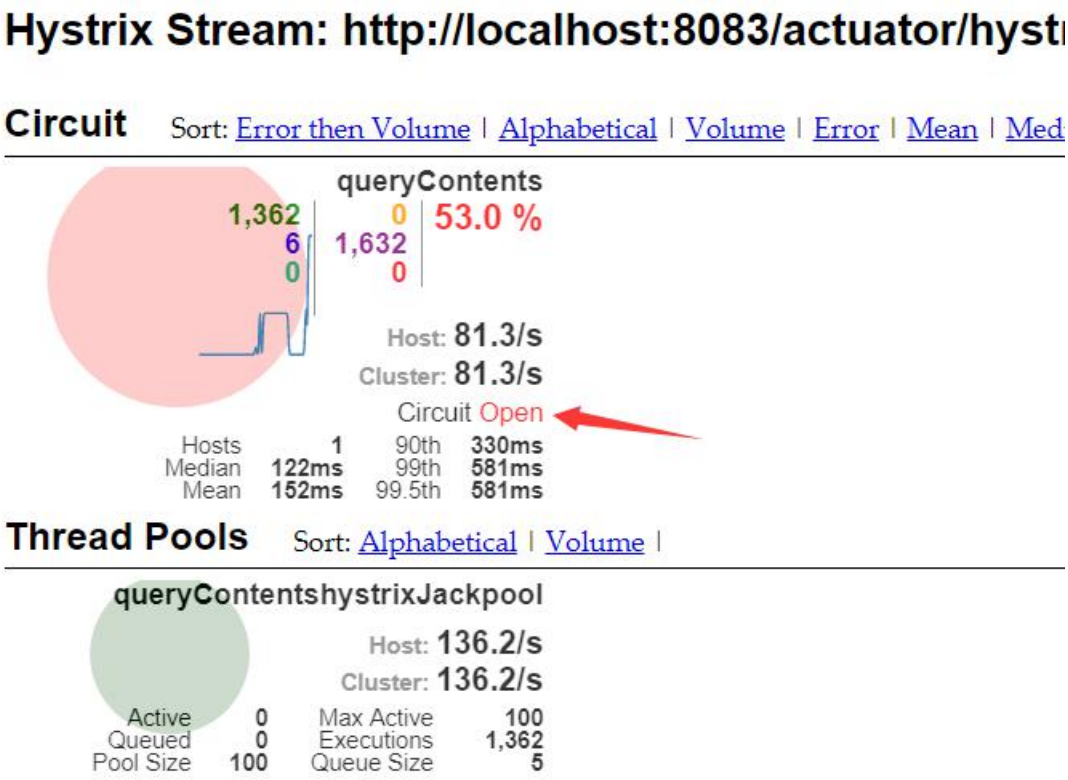
2、请求次数必须达到一定数量
相应的配置属性
circuitBreaker.requestVolumeThreshold
默认 20 次

3、失败率达到默认失败率
相应的配置属性
circuitBreaker.errorThresholdPercentage
默认 50%

上述 3 个条件缺一不可，必须全部满足才能开启 hystrix 的熔断功能。
当我们的对一个线程池大小是 100 的方法压测时看看 hystrix 的熔断效果：
Jmeter 压测：



Hystrix dashboard 界面



可以看到失败率超过 50%时，circuit 的状态是 open 的。

熔断器的三个状态：

1、关闭状态

关闭状态时用户请求是可以到达服务提供方的

2、开启状态

开启状态时用户请求是不能到达服务提供方的，直接会走降级方法

3、半开状态

当 hystrix 熔断器开启时，过一段时间后，熔断器就会由开启状态变成半开状态。

半开状态的熔断器是可以接受用户请求并把请求传递给服务提供方的，这时候如果远程调用返回成功，那么熔断器就会有半开状态变成关闭状态，反之，如果调用失败，熔断器就会有半开状态变成开启状态。

Hystrix 功能建议在并发比较高的方法上使用，并不是所有方法都得使用的。

18、Feign 的使用

Feign 是对服务端和客户端通用接口的封装，让代码可以复用做到统一管理。

1、jar 包导入

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

2、启动类导入 feign 客户端

```
@SpringBootApplication(scanBasePackages = {"com.xiangxue.jack"})
//注册到eureka
@EnableEurekaClient
//开启断路器功能
@EnableCircuitBreaker
//开启feign支持，clients指定哪个类开启feign
@EnableFeignClients(clients = {StudentService.class,TeacherServiceFeign.class})
//开启重试功能
//@EnableRetry
public class MicroWebApplication {

    @Bean
```

3、feign 客户端

```
@FeignClient(name = "MICRO-ORDER",path = "/feign"
        /*fallback = StudentServiceFallback.class,*/
        ,fallbackFactory = StudentServiceFallbackFactory.class)
public interface StudentService {

    @GetMapping("/student/getAllStudent")
    String getAllStudent();

    @PostMapping("/student/saveStudent")
    String saveStudent(@RequestBody Student student);

    @GetMapping("/student/getStudentById")
    String getStudentById(@RequestParam("id") Integer id);

    @GetMapping("/student/errorMessage")
    String errorMessage(@RequestParam("id") Integer id);

    @GetMapping("/student/queryStudentTimeout")
    String queryStudentTimeout(@RequestParam("millis") int millis);
}
```

4、服务端接口

```java
@Slf4j
@RestController
public class StudentController implements StudentService {

    @Autowired
    private StudentService studentService;

    @RequestMapping("/feign/student/getAllStudent")
    @Override
    public String getAllStudent() {
        return studentService.getAllStudent();
    }

    @RequestMapping("/feign/student/getStudentById")
    @Override
    public String queryStudentById(@RequestParam("id") Integer id) {
        return studentService.queryStudentById(id);
    }

    @RequestMapping("/feign/student/saveStudent")
    @Override
    public String saveStudent(@RequestBody Student student) {
        return studentService.saveStudent(student);
    }
```

服务端接口必须定义跟 feign 客户端相同的 url。

5、参数传递

对象类型参数

调用方

```java
@PostMapping("/student/saveStudent")
String saveStudent(@RequestBody Student student);
```

服务方

```java
@RequestMapping("/feign/student/saveStudent")
@Override
public String saveStudent(@RequestBody Student student) {
    return studentService.saveStudent(student);
}
```

实际上传的是字符串

其他参数

调用方

```java
@GetMapping("/student/getStudentById")
String getStudentById(@RequestParam("id") Integer id);
```

这里必须指定参数名称

服务方

```java
@RequestMapping("/feign/student/getStudentById")
@Override
public String queryStudentById(@RequestParam("id") Integer id) {
    return studentService.queryStudentById(id);
}
```

6、Feign 的服务降级

```
@Slf4j
@Component
public class StudentServiceFallbackFactory implements FallbackFactory<StudentService> {

    @Override
    public StudentService create(Throwable throwable) {

        if(throwable == null) {
            return null;
        }
        final String msg = throwable.getMessage();
        log.info("exception:" + msg);
        return new StudentService() {
            @Override
            public String getAllStudent() {
                log.info("exception=====getAllStudent==========" + msg);
                return msg;
            }

            @Override
            public String saveStudent(Student student) {
                log.info("exception=====saveStudent==========" + msg);
                return msg;
            }
```

这里在调用对应 feign 客户端方法出现异常了，就会回调到 create 方法中，最终会回调到对应的客户端方法中。

7、Feign 的异常过滤器

这个过滤器是对异常信息的再封装，把 feign 的异常信息封装成我们系统的通用异常对象

```
@Configuration
public class FeignErrMessageFilter {

    @Bean
    public ErrorDecoder errorDecoder() { return new FeignErrorDecoder(); }

    /*
    * 当调用服务时，如果服务返回的状态码不是200，就会进入到Feign的ErrorDecoder中
    * {"timestamp":"2020-02-17T14:01:18.080+0000","status":500,"error":"Internal Server Error","message":"/ by zer
    * 只有这种方式才能获取所有的被feign包装过的异常信息
    *
    * 这里如果创建的Exception是HystrixBadRequestException
    * 则不会走熔断逻辑，不记入熔断统计
    * */
    class FeignErrorDecoder implements ErrorDecoder {
        private Logger logger = LoggerFactory.getLogger(FeignErrorDecoder.class);
        @Override
        public Exception decode(String s, Response response) {
            RuntimeException runtimeException = null;
            try {
                String retMsg = Util.toString(response.body().asReader());
                logger.info(retMsg);

                runtimeException = new RuntimeException(retMsg);

            } catch (IOException e) {
                e.printStackTrace();
            }
```

过滤器把异常返回后，feign 前面定义的降级方法就会调到 create 方法。

Feign　我不建议大家使用，流程简单并发不高的方法可以用一用。

19、分布式配置中心

分布式配置中心解决了什么问题

1、抽取出各模块公共的部分，做到一处修改各处生效的目标

2、做到系统的高可用，修改了配置文件后可用在个模块动态刷新，不需要重启服务器

Springcloud 配置中心服务端搭建

1、jar 包导入

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

2、启动类

```
/*
* 加密: http://localhost:8085/encrypt?data=123456
* 解密: http://localhost:8085/decrypt
* */
@SpringBootApplication(scanBasePackages = {"com.xiangxue.jack"})
@EnableConfigServer
// 注册到eureka
@EnableEurekaClient
public class MicroConfigServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(MicroConfigServerApplication.class,args);
    }
}
```

3、分布式配置中心配置规则

分布式配置中心服务端是需要远程连接代码仓库的，比如 GitHub，gitlab 等，把需要管理的配置文件放到代码仓库中管理，所以服务端就需要有连接代码仓库的配置：

```
spring.cloud.config.server.git.uri=https://github.com/zg-jack/zg-config-repo
spring.cloud.config.server.git.search-paths=config-repo
spring.cloud.config.server.git.username=zg-jack
spring.cloud.config.server.git.password=zg0001jack
```

4、客户端使用配置中心

客户端只需要指定连接的服务端就行了，从服务端拉取配置信息

1、jar 包依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
    <version>LATEST</version>
</dependency>
```

2、Properties 配置

```
spring.cloud.config.profile=dev
spring.cloud.config.label=master

#这种配置是 configserver 还单机情况，直接连接这个单机服务就行

spring.cloud.config.uri=http://localhost:8085/
```

5、客户端快速失败和重试
当客户端连服务端失败时，客户端就快速失败，不进行加载其他的 spring 容器
快速失败

```
#如果连接不上获取配置有问题，快速响应失败
spring.cloud.config.fail-fast=true
```

客户端也有重试功能，连不上服务端是有重试机制
重试功能 jar 包导入

```
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
</dependency>
```

重试配置

```
#默认重试的间隔时间，默认 1000ms
spring.cloud.config.retry.multiplier=1000
#下一间隔时间的乘数，默认是 1.1
#spring.cloud.config.retry.initial-interval=1.1
#最大间隔时间，最大 2000ms
spring.cloud.config.retry.max-interval=2000
#最大重试次数，默认 6 次
spring.cloud.config.retry.max-attempts=6
```

重试效果

```
: Fetching config from server at : http://localhost:8085/
: Connect Timeout Exception on Url - http://localhost:8085/. Will be trying the next url i
: Fetching config from server at : http://localhost:8085/
: Connect Timeout Exception on Url - http://localhost:8085/. Will be trying the next url i
: Fetching config from server at : http://localhost:8085/
: Connect Timeout Exception on Url - http://localhost:8085/. Will be trying the next url i
: Fetching config from server at : http://localhost:8085/
: Connect Timeout Exception on Url - http://localhost:8085/. Will be trying the next url i
: Fetching config from server at : http://localhost:8085/
: Connect Timeout Exception on Url - http://localhost:8085/. Will be trying the next url i
: Fetching config from server at : http://localhost:8085/
: Connect Timeout Exception on Url - http://localhost:8085/. Will be trying the next url i
: Application run failed

  failing
```

6、配置信息的加密
在配置中心中，有些信息是比较敏感的，比如密码信息，在配置密码信息的时候有必要对密码信息加密以免密码信息泄露，springcloud 配置中心也支持配置信息加密的，这里一 RSA 非对称加密举例。

1、本地生成秘钥对

cd 到 jdk 的 keytool 目录:D:\Program Files\Java\jdk1.8.0_92\jre\bin

里面有一个 keytool.exe 可执行文件

2、执行指令生成秘钥文件

keytool -genkeypair -alias config-server -keyalg RSA -keystore config-server.keystore -validity 365

指令执行成功后会在 bin 目录生成一个 config-server.keystore 文件，把该文件 copy 到配置中心服务工程中 resources 目录下。

3、服务端工程配置

Properties 配置文件

添加秘钥配置

```
#加密配置
encrypt.key-store.location=config-server.keystore
encrypt.key-store.alias=config-server
encrypt.key-store.password=123456
encrypt.key-store.secret=123456
```

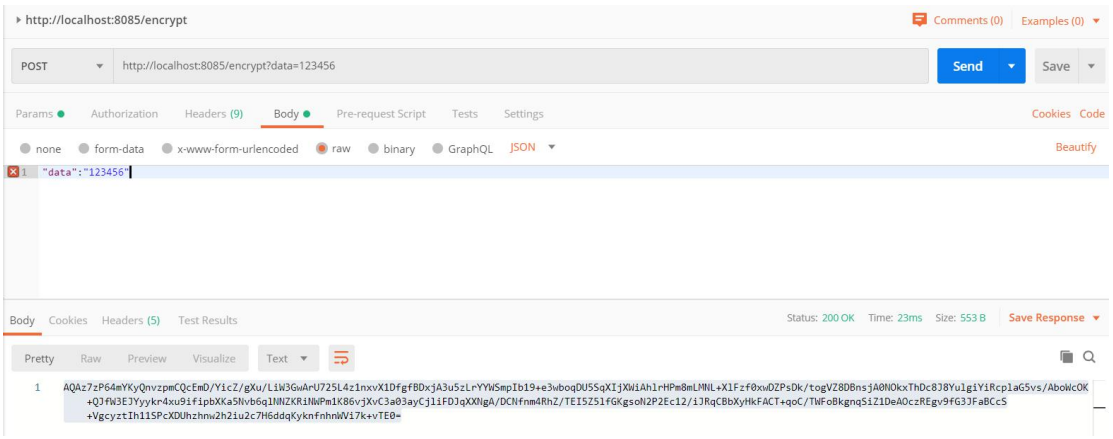
pom 中添加静态文件扫描，让能够扫描到.keystore 文件

```
<resources>
    <resource>
        <directory>src/main/java</directory>
        <includes>
            <include>**/*.properties</include>
            <include>**/*.xml</include>
        </includes>
        <filtering>false</filtering>
    </resource>
    <resource>
        <directory>src/main/resources</directory>
        <includes>
            <include>**/*.properties</include>
            <include>**/*.xml</include>
            <include>**/*.txt</include>
            <include>**/*.keystore</include>
        </includes>
        <filtering>false</filtering>
    </resource>
</resources>
```

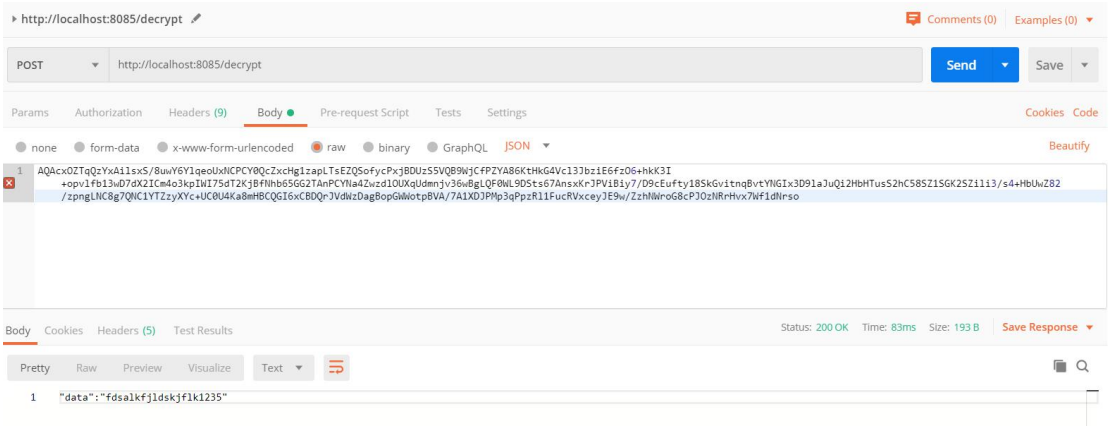
4、密码加密和解密接口

在服务端中有提供对信息加密和解密接口的

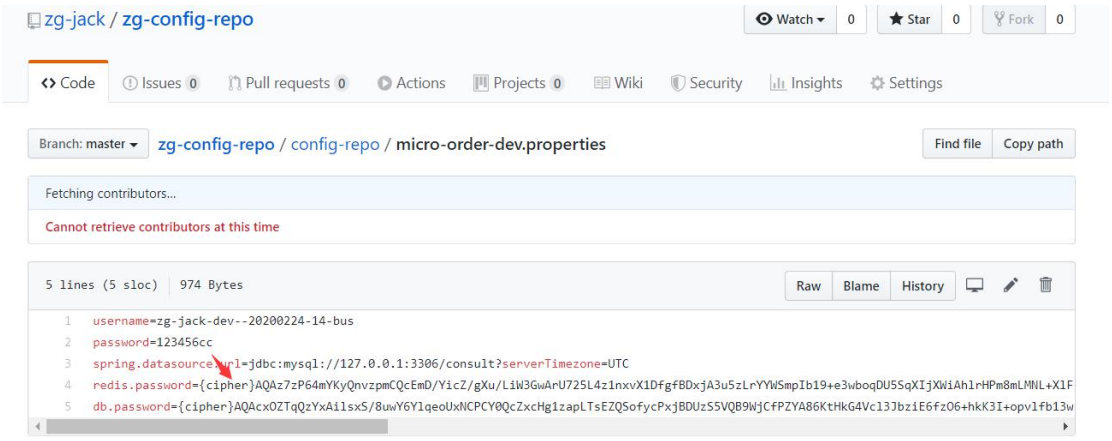
加密接口：http://localhost:8085/encrypt?data=123456，post 请求



解密接口：http://localhost:8085/decrypt，post 请求



5、代码仓库密文配置



密文前面一定要加上{cipher}标识这个是密文配置，需要服务端来解密的。

6、配置动态加载刷新

这个是一个革命性的功能，在运行期修改配置文件后，我们通过这个动态刷新功能可以不重启服务器，这样我们系统理论上可以保证 7*24 小时的工作

1、Environment 的动态刷新

动态刷新其实要有一个契机，其实这个契机就是手动调用刷新接口，如果你想刷新哪台主机的配置，就调用哪台注解的刷新接口
刷新接口为：http://localhost:8088/actuator/refresh

2、@Value 注入的属性动态刷新

其实前面在调用刷新接口后，@Value 注入的属性是没有刷新的还是老的配置，这个也好理解，@Value 注入的属性是项目启动就已经定了的。如果要使@Value 属性也刷新，就必须要在类上面加上：@RefreshScope 注解。

但是调用每台主机的刷新接口显然太麻烦了，如果需要刷新的集群机器有几百台，是不是就需要手动调用几百次呢，这几乎是一个不能完成的工作量。
Springcloud 中也提供了消息总线的东西，借助 mq 来完成消息的广播，当需要刷新时我们就只要调用一次刷新接口即可。

7、消息总线

消息总线其实很简单，就是为了解决一点刷新的功能，在一个点调用请求刷新接口，然后所有的在消息总线中的端点都能接到刷新的消息，所有我们必须把每一个端点都拉入到消息总线中来。

1、jar 导入

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

2、properties 配置

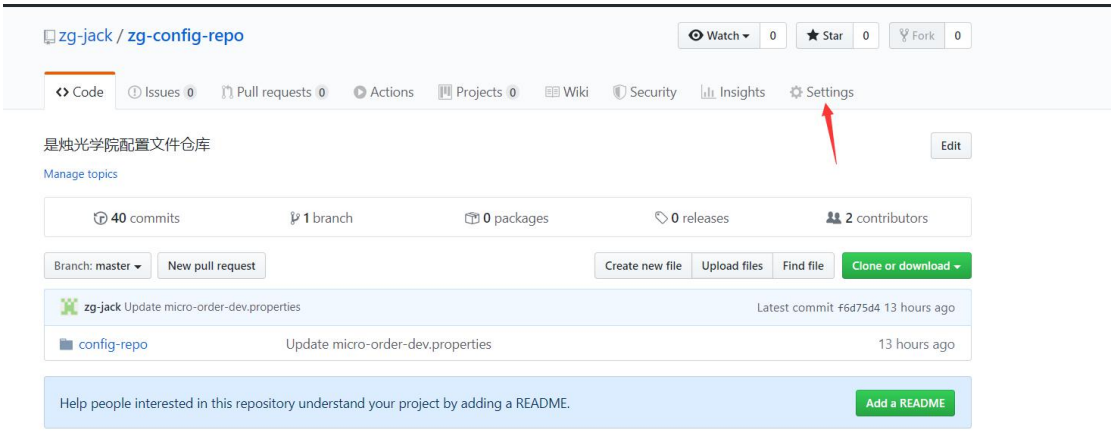
其实就是连接 mq 的配置

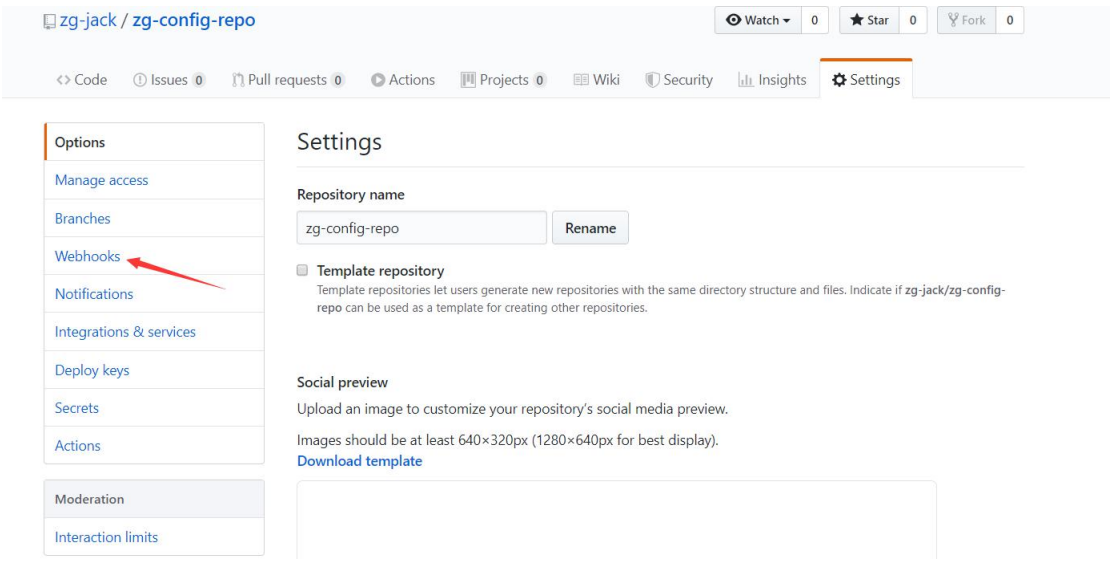
```
spring.rabbitmq.host=192.168.67.139
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=admin

# 刷新配置 url  http://localhost:8081/actuator/bus-refresh
spring.cloud.bus.refresh.enabled=true
spring.cloud.bus.trace.enabled=true
```

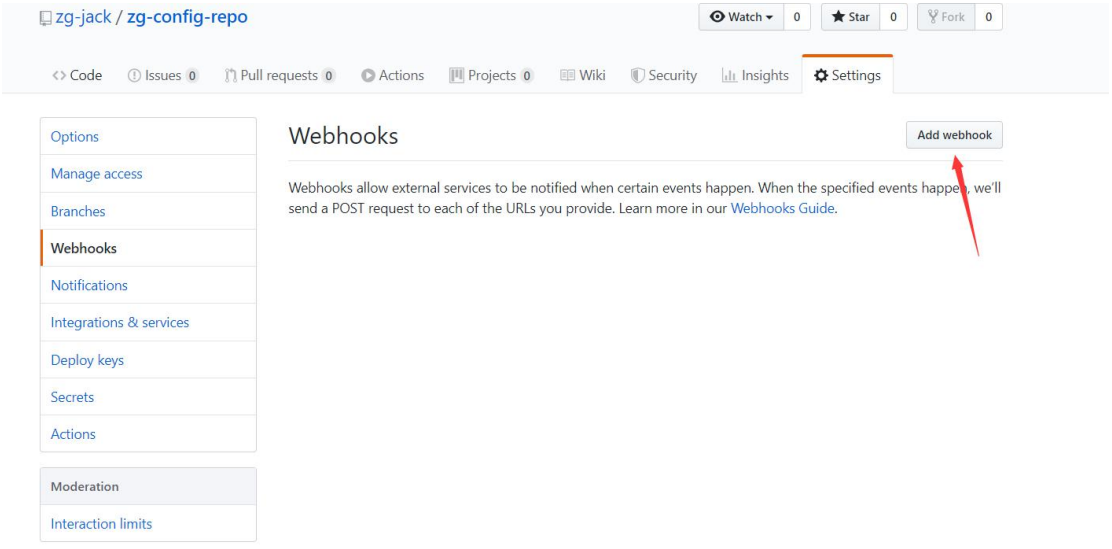
通过这两步就已经完成了拉入消息总线的工作了。

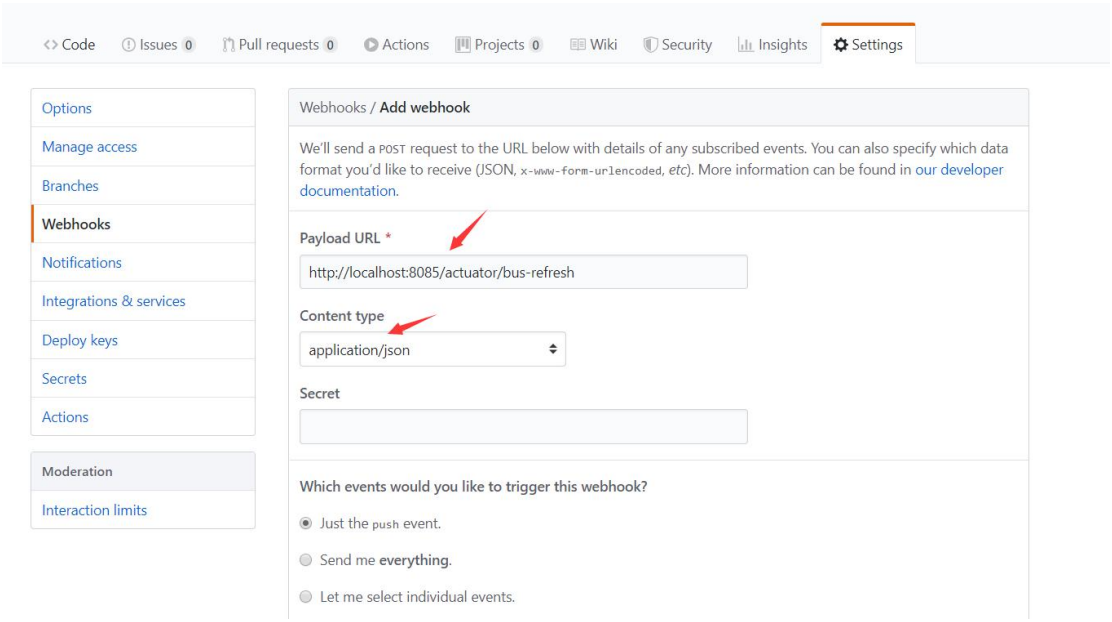
如果要刷新配置，就只要调用任意一个消息总线端点调用刷新接口即可，其他的端点就会收到刷新配置的消息。
刷新接口：http://localhost:8085/actuator/bus-refresh

这个接口也可以配置到 GitHub 中，只要 GitHub 有代码变更，就会调用这个接口









当然这里配置的 URL 是你生产环境的外网地址，这样当你 GitHub 中修改了配置后，就会主动调用消息总线的刷新接口。

消息总线弊端就是太重了，一个集群通知刷新配置功能，还得用一个 rabbitmq 来做，如果项目中根本用不到 rabbitmq 呢？就加大了项目负担，加大了维护成本。差评。

20、自定义分布式配置中心

1、zookeeper 创建节点及事件监听

```
RefreshScopeRegistry refreshScopeRegistry = (RefreshScopeRegistry)applicationContext.getBean( name: "refre
beanDefinitionRegistry = refreshScopeRegistry.getBeanDefinitionRegistry();

client = CuratorFrameworkFactory.
        builder().
        connectString(connnectStr).
        sessionTimeoutMs(5000).
        retryPolicy(new ExponentialBackoffRetry( baseSleepTimeMs: 1000,  maxRetries: 3)).
        build();

client.start();
try {
    Stat stat = client.checkExists().forPath(path);
    if (stat == null) {
        client.create().creatingParentsIfNeeded().withMode(CreateMode.PERSISTENT).
                forPath(path, "zookeeper config".getBytes());
        TimeUnit.SECONDS.sleep( timeout: 1);
    } else {
        //1、把config下面的子节点加载到spring容器的属性对象中
        addChildToSpringProperty(client, path);
    }
```

2、如果 zookeeper 里面有配置了把配置加载到 spring 中

```
private void createZookeeperSpringProperty() {
    MutablePropertySources propertySources = applicationContext.getEnvironment().getPropertySources();
    OriginTrackedMapPropertySource zookeeperSource = new OriginTrackedMapPropertySource(zkPropertyName, map);
    propertySources.addLast(zookeeperSource);
}
```

```
private void addChildToSpringProperty(CuratorFramework client, String path) {
    if (!checkExistsSpringProperty()) {
        //如果不存在zookeeper的配置属性对象则创建
        createZookeeperSpringProperty();
    }

    //把config目录下的子节点添加到 zk的PropertySource对象中
    MutablePropertySources propertySources = applicationContext.getEnvironment().getPropertySources();
    PropertySource<?> propertySource = propertySources.get(zkPropertyName);
    ConcurrentHashMap zkmap = (ConcurrentHashMap) propertySource.getSource();
    try {
        List<String> strings = client.getChildren().forPath(path);
        for (String string : strings) {
            zkmap.put(string, client.getData().forPath(path + "/" + string));
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

3、为配置节点添加事件

```
private void childNodeCache(CuratorFramework client, String path) {
    try {
        final PathChildrenCache pathChildrenCache = new PathChildrenCache(client, path, cacheData: false);
        pathChildrenCache.start(PathChildrenCache.StartMode.POST_INITIALIZED_EVENT);

        pathChildrenCache.getListenable().addListener((client, event) -> {
                switch (event.getType()) {
                    case CHILD_ADDED:
                        System.out.println("增加了节点");
                        addEnv(event.getData(), client);
                        break;
                    case CHILD_REMOVED:
                        System.out.println("删除了节点");
                        delEnv(event.getData());
                        break;
                    case CHILD_UPDATED:
                        System.out.println("更新了节点");
                        addEnv(event.getData(), client);
                        break;
                    default:
                        break;
                }
                //对refresh作用域的实例进行刷新
                refreshBean();
        });
    } catch (Exception e) {
```

当有节点新增、修改、删除时触发对应的事件，事件方法：

```
private void addEnv(ChildData childData, CuratorFramework client) {
    ChildData next = childData;
    String childpath = next.getPath();
    String data = null;
    try {
        data = new String(client.getData().forPath(childpath));
    } catch (Exception e) {
        e.printStackTrace();
    }
    MutablePropertySources propertySources = applicationContext.getEnvironment().getPropertySources();
    for (PropertySource<?> propertySource : propertySources) {
        if (zkPropertyName.equals(propertySource.getName())) {
            OriginTrackedMapPropertySource ps = (OriginTrackedMapPropertySource) propertySource;
            ConcurrentHashMap chm = (ConcurrentHashMap) ps.getSource();
            chm.put(childpath.substring(path.length() + 1), data);
        }
    }
}

private void delEnv(ChildData childData) {
    ChildData next = childData;
    String childpath = next.getPath();
    MutablePropertySources propertySources = applicationContext.getEnvironment().getPropertySources();
    for (PropertySource<?> propertySource : propertySources) {
        if (zkPropertyName.equals(propertySource.getName())) {
            OriginTrackedMapPropertySource ps = (OriginTrackedMapPropertySource) propertySource;
            ConcurrentHashMap chm = (ConcurrentHashMap) ps.getSource();
            chm.remove(childpath.substring(path.length() + 1));
        }
    }
}
```

4、对象重新创建触发@Value 的依赖注入

1、自定义 Scope，自己来管理 bean

```
public class RefreshScope implements Scope {

    private ConcurrentHashMap map = new ConcurrentHashMap();

    @Override
    public Object get(String name, ObjectFactory<?> objectFactory) {
        if(map.containsKey(name)) {
            return map.get(name);
        }

        Object object = objectFactory.getObject();
        map.put(name,object);
        return object;
    }

    @Override
    public Object remove(String name) {
        return map.remove(name);
    }
```

如果缓存中有则直接返回，如果缓存没用则调用 getObject 方法创建对象

2、把自定义作用域注册到 spring 中

```
@Data
//@Component
public class RefreshScopeRegistry implements BeanDefinitionRegistryPostProcessor {

    private BeanDefinitionRegistry beanDefinitionRegistry;


    @Override
    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException {
        this.beanDefinitionRegistry = registry;
    }

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {
        beanFactory.registerScope( scopeName: "refresh",new RefreshScope());
    }
}
```

3、在有节点更新修改时重新创建节点

```
private void refreshBean() {
    String[] beanDefinitionNames = applicationContext.getBeanDefinitionNames();
    for (String beanDefinitionName : beanDefinitionNames) {
        BeanDefinition beanDefinition = beanDefinitionRegistry.getBeanDefinition(beanDefinitionName);
        if(scopeName.equals(beanDefinition.getScope())) {
            //先删除,,,,思考，如果这时候删除了bean，有没有问题?
            applicationContext.getBeanFactory().destroyScopedBean(beanDefinitionName);
            //在实例化
            applicationContext.getBean(beanDefinitionName);
        }
    }
}
```

这里就会把自己管理的 bean 从缓存中删除，并且调用 getBean 创建新的对象

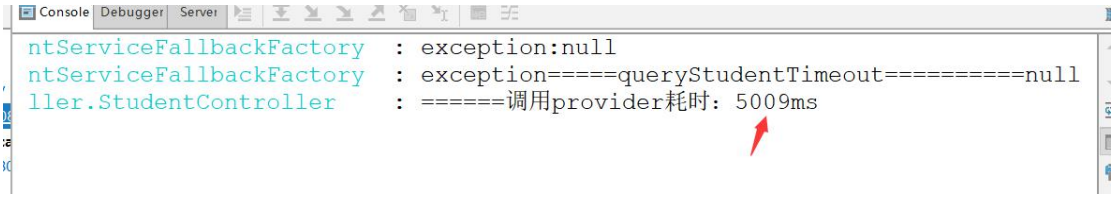
21、超时配置

```
#全局超时时间
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=5000
#hystrix.command.default.metrics.rollingStats.timeInMilliseconds=4000
#hystrix.command.default.metrics.healthSnapshot.intervalInMilliseconds=2000
#hystrix.command.<commandKey>作为前缀,默认是采用Feign的客户端的方法名字作为标识
hystrix.command.saveStudent.execution.isolation.thread.timeoutInMilliseconds=6000
hystrix.command.queryContents.circuitBreaker.sleepWindowInMilliseconds=20000

hystrix.command.errorMessage.execution.isolation.thread.timeoutInMilliseconds=100000000000

#ribbon.restclient.enabled=true
ribbon.client.name=micro-order
#点对点直连测试配置
# 关闭ribbon访问注册中心Eureka Server发现服务，但是服务依旧会注册。
#true使用eureka false不使用
ribbon.eureka.enabled=true
spring.cloud.loadbalancer.retry.enabled=true
#指定调用的节点
#micro-order.ribbon.listOfServers=localhost:8001
#单位ms ,请求连接超时时间
ribbon.ConnectTimeout=1000
#单位ms ,请求处理的超时时间
ribbon.ReadTimeout=3000
```

超时时间配置，如果 hystrix 超时时间大于 ribbon 超时时间，那么就以 hystrix 超时时间为准。

http://localhost:8083/student/timeOut?millis=2999

总共耗时：



Ribbon 重试

```
io-8086-exec-8] c.x.jack.controller.StudentController    : provider--->2999
io-8086-exec-7] c.x.jack.controller.StudentController    : provider--->2999
```

这种请求是 hystrix 超时了，触发了 ribbon 的重试，ribbon 调用还没结束，但是超过了 hystrix 的超时，所以 hystrix 拿不到异常。

22、Zuul 服务网关

Zuul 是分布式 springcloud 项目的流量入口，理论上所有进入到微服务系统的请求都要经过 zuul 来过滤和路由。

1、zuul 服务网关的搭建

Jar 包导入

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
```

启动类加注解

```java
@SpringBootApplication
@EnableZuulProxy
//@EnableZuulServer
public class MicroZuulApplication {
    public static void main(String[] args) {
        SpringApplication.run(MicroZuulApplication.class,args);
    }

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

2、配置详解

配置拦截的 url 和该 url 路由的服务

```
# 使用路径方式匹配路由规则。
# 参数key结构:  zuul.routes.customName.path=xxx
# 用于配置路径匹配规则。
# 其中customName自定义。通常使用要调用的服务名称，方便后期管理
# 可使用的通配符有:  * ** ?
# ? 单个字符
# * 任意多个字符，不包含多级路径
# ** 任意多个字符，包含多级路径
zuul.routes.micro-web.path=/web/**

# 参数key结构:  zuul.routes.customName.url=xxx
# url用于配置符合path的请求路径路由到的服务地址。
#zuul.routes.micro-order.url=http://localhost:8080/

# key结构 :  zuul.routes.customName.serviceId=xxx
# serviceId用于配置符合path的请求路径路由到的服务名称。
zuul.routes.micro-web.serviceId=micro-web-no
```

配置敏感请求头过滤

```
#针对某个服务传输指定的headers信息 ，默认是过滤掉 Cookie,Set-Cookie,Authorization 这三个信息的
#这里置空就是不要过滤掉这三个
zuul.routes.micro-web.sensitive-headers=
```

Zuul 本地跳转

```
zuul.routes.zuul-server.path=/local/**
zuul.routes.zuul-server.url=forward:/local
```

Zuul 跳转到指定地址

```
zuul.routes.blog.path=/blog/**
zuul.routes.blog.url=http://localhost:8003/
```

3、路由动态刷新

路由动态刷新其实就是分布式配置中心的思路，就是路由配置规则我们可以在项目启动后，通过修改远程 GitHub 上的配置，让配置在运行时生效，不需要重启机器。所以就需要加入配置中心的配置。

配置中心客户端 jar

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
    <version>LATEST</version>
</dependency>
```

配置信息

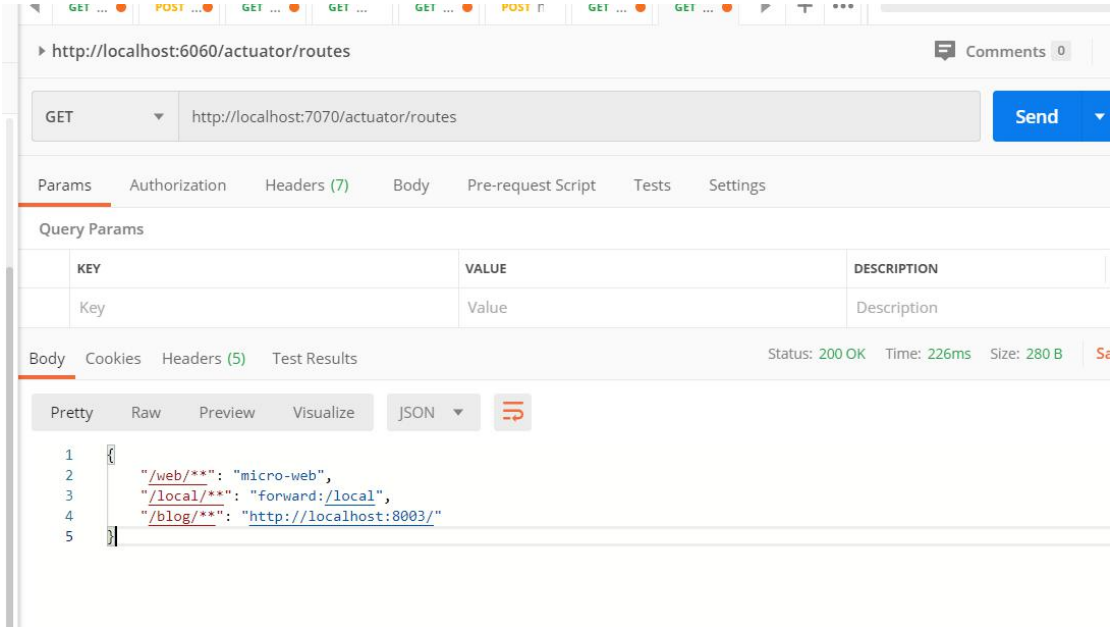
```
spring.cloud.config.profile=dev
spring.cloud.config.label=master
#这种配置是configserver还单机情况，直接连接这个单机服务就行
spring.cloud.config.uri=http://localhost:8085/
#configserver高可用配置
#开启configserver服务发现功能
#spring.cloud.config.discovery.enabled=true
#服务发现的服务名称
#spring.cloud.config.discovery.service-id=config-server

#如果连接不上获取配置有问题，快速响应失败
spring.cloud.config.fail-fast=true
#默认重试的间隔时间，默认1000ms
spring.cloud.config.retry.multiplier=1000
#下一间隔时间的乘数，默认是1.1
#spring.cloud.config.retry.initial-interval=1.1
#最大间隔时间，最大2000ms
spring.cloud.config.retry.max-interval=2000
#最大重试次数，默认6次
spring.cloud.config.retry.max-attempts=6
```

配置动态刷新类

```java
@SpringBootApplication
@EnableZuulProxy
//@EnableZuulServer
public class MicroZuulApplication {
    public static void main(String[] args) {
        SpringApplication.run(MicroZuulApplication.class,args);
    }

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

获取路由规则的接口

http://localhost:7070/actuator/routes



4、zuul 过滤器

Zuul 大部分功能都是通过过滤器来实现的，Zuul 定义了 4 种标准的过滤器类型，这些过滤器类型对应于请求的典型生命周期。

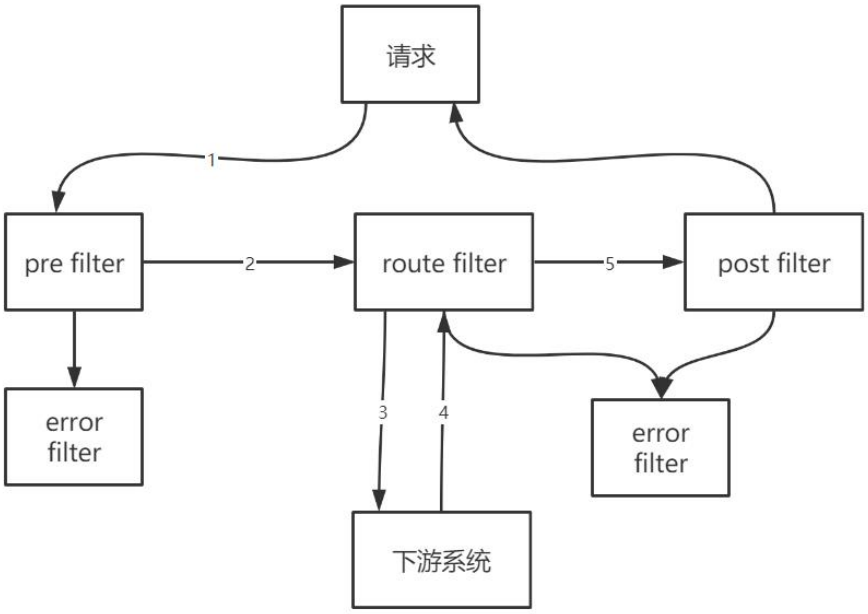
a、pre: 这种过滤器在请求被路由之前调用。可利用这种过滤器实现身份验证、在集群中选择请求的微服务，记录调试信息等。

b、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 netflix ribbon 请求微服务。

c、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。

e、error: 在其他阶段发送错误时执行该过滤器。

执行流程



23、Springcloud admin

Springcloud admin 是基于

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

把 actuator 负责统计数据，admin 是根据统计出来的数据来进行展示的，可以很好的监控整个微服务系统中的实例运行情况信息。

1、jar 导入

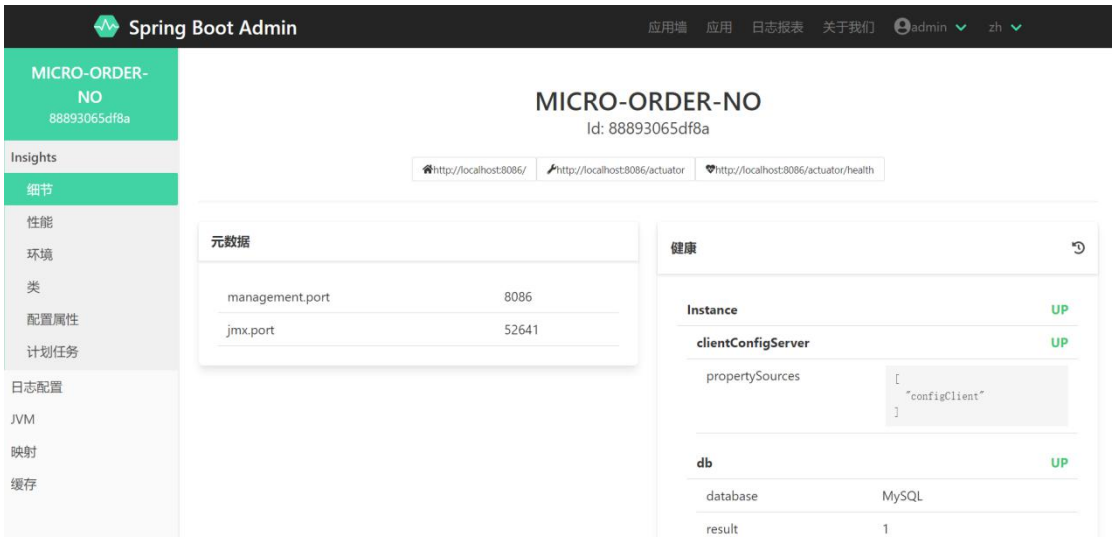
```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
    <version>LATEST</version>
</dependency>
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-server-ui</artifactId>
    <version>LATEST</version>
</dependency>
```

2、启动类加注解

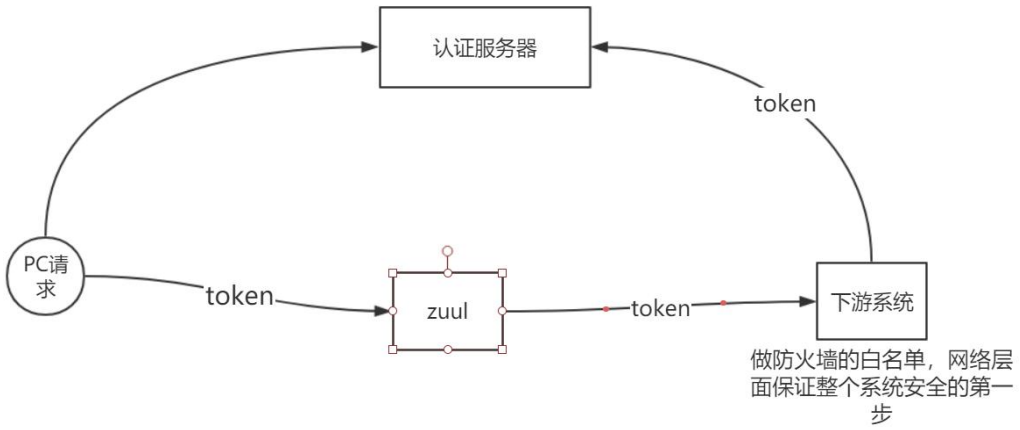
```java
@SpringBootApplication
@EnableEurekaClient
@EnableAdminServer
public class MicroAdminApplication {
    public static void main(String[] args) {
        SpringApplication.run(MicroAdminApplication.class,args);
    }
}
```

3、安全配置

```properties
# 安全配置
spring.security.user.name=admin
spring.security.user.password=admin
eureka.instance.metadata-map.user.name=${spring.security.user.name}
eureka.instance.metadata-map.user.password=${spring.security.user.password}
```



24、微服务系统的权限校验



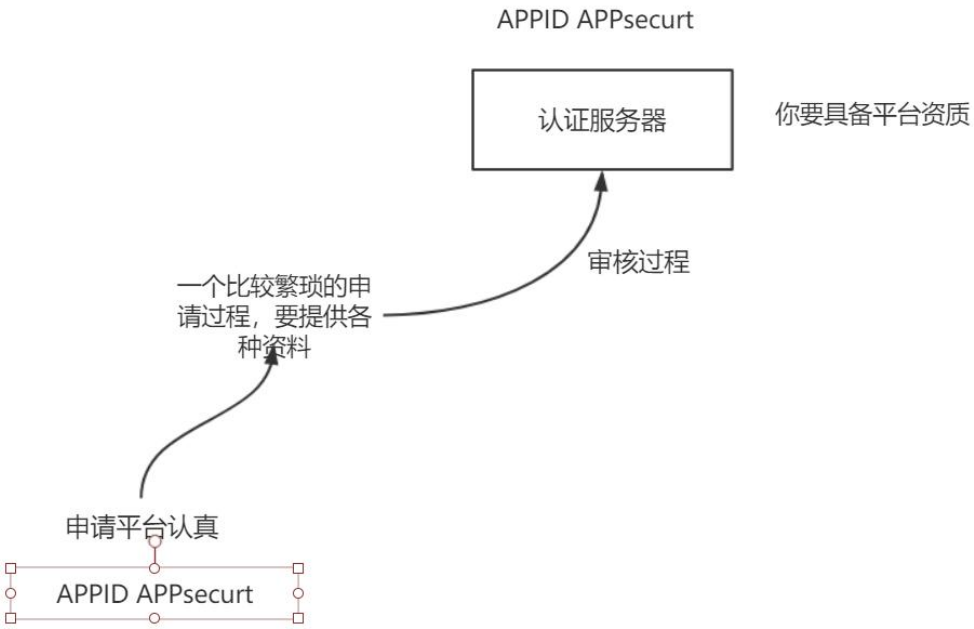
采用 token 认证的方式校验是否有接口调用权限，然后在下游系统设置访问白名单只允许 zuul 服务器访问。理论上 zuul 服务器是不需要进行权限校验的，因为 zuul 服务器没有接口，不需要从 zuul 调用业务接口，zuul 只做简单的路由工作。下游系统在获取到 token 后，通过过滤器把 token 发到认证服务器校验该 token 是否有效，如果认证服务器校验通过就会携带这个 token 相关的验证信息传回给下游系统，下游系统根据这个返回结果就知道该 token 具有的权限是什么了。所以校验 token 的过程，涉及到下游系统和认证服务器的交互，这点不好。占用了宝贵的请求时间。

25、获取 token 的过程

Token 是通过一个单独的认证服务器来颁发的，只有具备了某种资质认证服务器才会把 token 给申请者。

1、平台认证申请

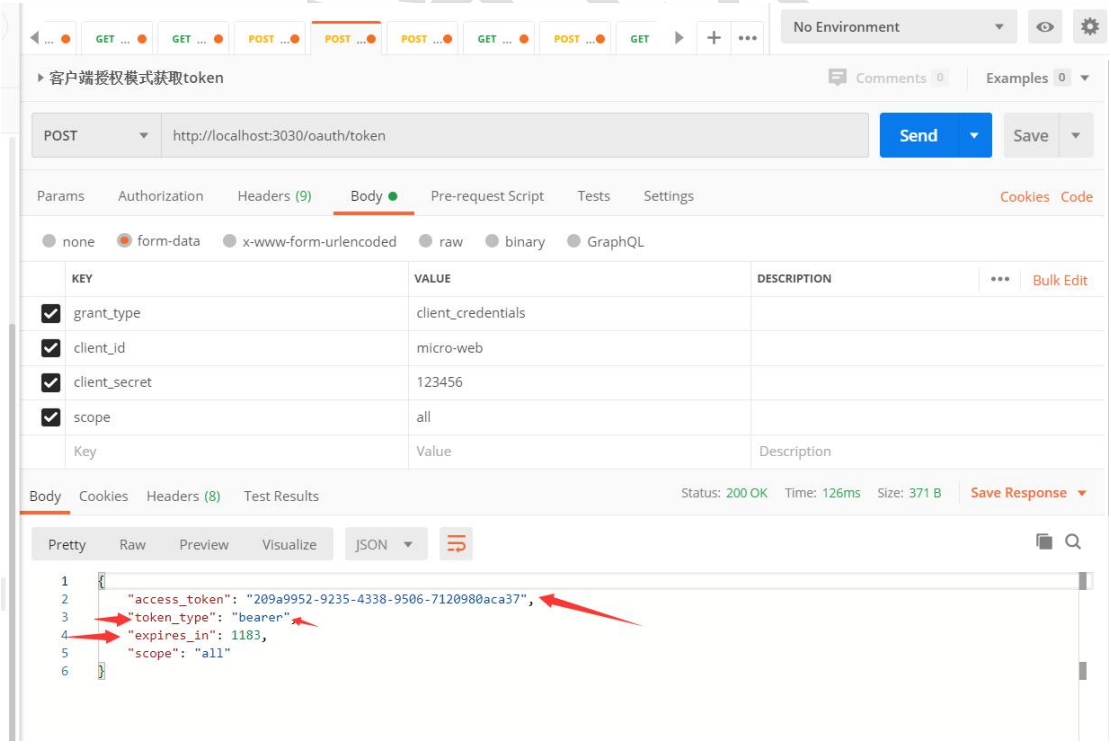
平台认证申请往往是一个比较繁琐的过程，需要申请方提供比较完整的认证申请材料，比如公司资质，营业执照等等信息，提交申请后认证方审核通过后，该平台才会允许申请 token。



2、获取 token

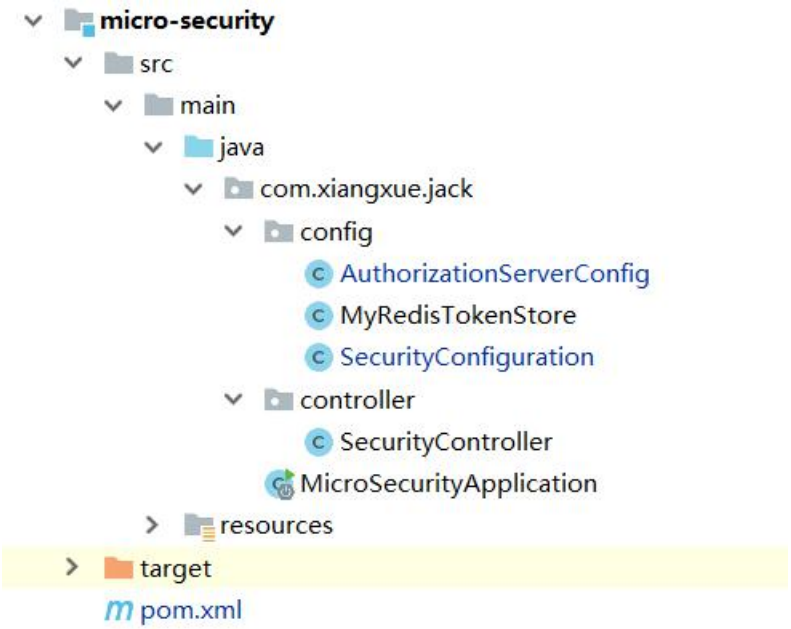
获取 token 在 oauth2.0 里面有 4 中模式，这里我们讲三种

1、客户端模式



我们可以看到客户端模式申请 token，只要带上有平台资质的客户端 id、客户端密码、然后带上授权类型是客户端授权模式，带上 scope 就可以了。这里要注意的是客户端必须是具有资质的。

基于 redis 存储 token 信息的认证服务器搭建在 micro-security 工程



1、认证服务器注解

```
@Configuration
@EnableAuthorizationServer
public class AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {
```

2、客户端信息配置

```
@Override
public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
    String finalSecret = "{bcrypt}" + new BCryptPasswordEncoder().encode( rawPassword: "123456");
    clients.
            jdbc(dataSource).
            inMemory().
            withClient( clientId: "micro-web")
            .resourceIds("micro-web")
            .authorizedGrantTypes("client_credentials", "refresh_token")
            .scopes("all","read", "write","aa")
            .authorities("client_credentials")
            .secret(finalSecret)
            .accessTokenValiditySeconds(1200)
            .refreshTokenValiditySeconds(50000)
            .and()
            .withClient( clientId: "micro-zuul")
            .resourceIds("micro-zuul")
            .authorizedGrantTypes("password", "refresh_token")
            .scopes("server")
            .authorities("password")
            .secret(finalSecret)
            .accessTokenValiditySeconds(1200)
            .refreshTokenValiditySeconds(50000);
}
```

3、配置 token 存储方式

```java
@Bean
public TokenStore tokenStore() {
    return new MyRedisTokenStore(connectionFactory);
}

/*
* AuthorizationServerEndpointsConfigurer: 用来配置授权（authorization）以及令牌（token）的访问端点和令牌服务(token ser
* */
@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
    endpoints
            .authenticationManager(authenticationManager)
            .userDetailsService(userDetailsService)//若无，refresh_token会有UserDetailsService is required错误
            .tokenStore(tokenStore()).
            allowedTokenEndpointRequestMethods(HttpMethod.GET,HttpMethod.POST);
}
```

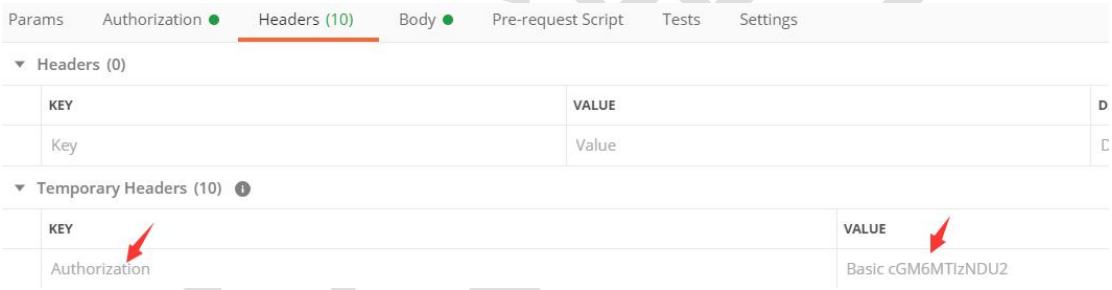
Oauth2.0 权限校验认证服务器代码配置和客户端代码配置基本上是固定写法，这里知道如何使用即可，关键是理解认证授权过程，oauth2.0 授权流程基本上是行业认证授权的标准了。

26、密码模式获取 token

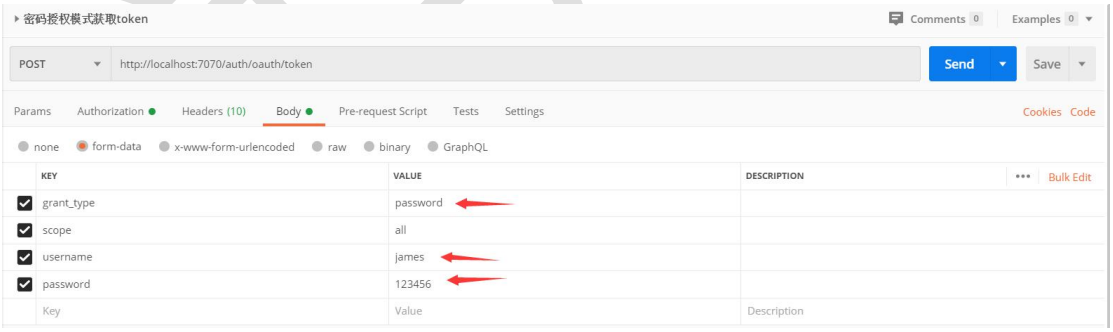
密码模式获取 token，也就是说在获取 token 过程中必须带上用户的用户名和密码，获取到的 token 是跟用户绑定的。

密码模式获取 token：

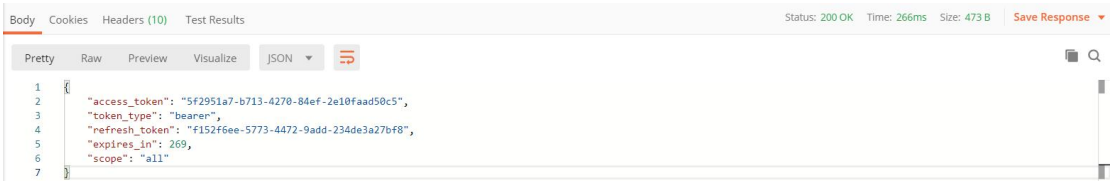
客户端 id 和客户端密码必须要经过 base64 算法加密，并且放到 header 中，加密模式为 Base64(clientId:clientPassword),如下：



其他参数：



获取到的 token



1、密码模式认证服务器代码配置

加上注解，说明是认证服务器

```
@EnableAuthorizationServer
@Configuration
@EnableAuthorizationServer
public class AuthorizationServerConfiguration extends AuthorizationServerConfigurerAdapter {
```

1、客户端配置，客户端是存储在表中的

```
@Bean // 声明 ClientDetails实现
public ClientDetailsService clientDetailsService() {
    return new JdbcClientDetailsService(dataSource);
}
```

对应的客户端表为：oauth_client_details

表中数据：

client_id	resource_	client_secret	scope
android	(Null)	{bcrypt}$2a$10$y.iyrX9c7lotqEqs4JF8ZuDfr06l.7Ryg7y7aUYFO.rqlhca32PBi	all
pc	(Null)	{bcrypt}$2a$10$y.iyrX9c7lotqEqs4JF8ZuDfr06l.7Ryg7y7aUYFO.rqlhca32PBi	all

authorized_grant_types	web_server_	authorities	access_toke	refresh_tok	additional_i	autoapprove
client_credentials,refresh_token	(Null)	oauth2	(Null)	(Null)	(Null)	
password,refresh_token,client_credentials,	(Null)	oauth2	(Null)	(Null)	(Null)	true

把客户端信息加入到 oauth2.0 框架中

```
@Override
public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
    clients.withClientDetails(clientDetailsService);
}
```

2、token 的保存方式，token 是也存储在数据库中

```
@Bean
public TokenStore tokenStore() {
    return new JdbcTokenStore(dataSource);
}
```

对应的 token 存储表为：**oauth_access_token**

表中数据：

token_id	token	authentication_id	user_name	client_id	authenticat	refresh_token
100bcc768b	(BLOB)	588dfa124da8d2728(	(Null)	pc	(BLOB)	1c09a8a7d1d364b19130(
129f2d6f6f3	(BLOB)	d68a31f8612ea056c6	james	pc	(BLOB)	0eb86596de180dc1d6d5(
4eca9d3b51	(BLOB)	e1a03d88cd328a9f8(	jack	pc	(BLOB)	2010c5fe95b1aa380082e(

可以看到 token 是跟用户绑定的。

设置 token 的属性

```
@Override
public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
    // redisTokenStore
      endpoints.tokenStore(new MyRedisTokenStore(redisConnectionFactory))
              .authenticationManager(authenticationManager)
              .allowedTokenEndpointRequestMethods(HttpMethod.GET, HttpMethod.POST);

    // 存数据库
    endpoints.tokenStore(tokenStore)
            .authenticationManager(authenticationManager)
            .userDetailsService(userServiceDetail);

    // 配置tokenServices参数
    DefaultTokenServices tokenServices = new DefaultTokenServices();
    tokenServices.setTokenStore(endpoints.getTokenStore());
    //支持refreshtoken
    tokenServices.setSupportRefreshToken(true);
    tokenServices.setClientDetailsService(endpoints.getClientDetailsService());
    tokenServices.setTokenEnhancer(endpoints.getTokenEnhancer());
    tokenServices.setAccessTokenValiditySeconds(60 * 5);
    //重复使用
    tokenServices.setReuseRefreshToken(false);
    tokenServices.setRefreshTokenValiditySeconds(60 * 10);
    endpoints.tokenServices(tokenServices);
}
```

3、认证服务器 token 校验和校验结果返回接口

```
@Slf4j
@RestController
@RequestMapping("/security")
public class SecurityController {

    @RequestMapping(value = "/check", method = RequestMethod.GET)
    public Principal getUser(Principal principal) {
        log.info("security server check=================>>>" + principal.toString());
        return principal;
    }
}
```

2、密码模式下游系统配置

1、properties 配置

指定客户端请求认证服务器接口

```
#security.oauth2.resource.service-id=api-gateway
security.oauth2.resource.user-info-uri=http://127.0.0.1:7070/auth/security/check
security.oauth2.resource.prefer-token-info=false
#security.oauth2.client.id=micro-web
security.oauth2.client.clientId=micro-web
security.oauth2.client.client-secret=123456
security.oauth2.client.access-token-uri=http://api-gateway/auth/oauth/token
security.oauth2.client.grant-type=client_credentials
security.oauth2.client.scope=all
```

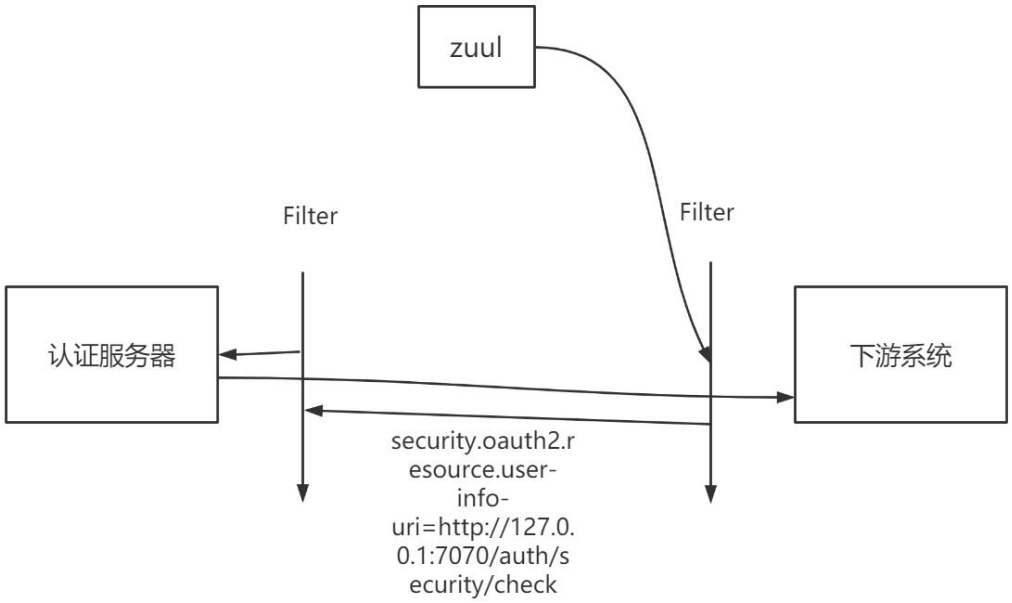
2、声明使用 oauth2.0 框架并说明这是一个客户端

```java
@EnableOAuth2Client
@EnableConfigurationProperties
@Configuration
public class OAuth2ClientConfig {


    @Bean
    @ConfigurationProperties(prefix = "security.oauth2.client")
    public ClientCredentialsResourceDetails clientCredentialsResourceDetails() {
        return new ClientCredentialsResourceDetails();
    }

//    @Bean
    public RequestInterceptor oauth2FeignRequestInterceptor(ClientCredentialsResourc
        return new OAuth2FeignRequestInterceptor(new DefaultOAuth2ClientContext(), c
    }

    @Bean
    public OAuth2RestTemplate clientCredentialsRestTemplate() {
        return new OAuth2RestTemplate(clientCredentialsResourceDetails());
    }
}
```

3、开启权限的方法级别注解和指定拦截路径

```java
@Configuration
@EnableResourceServer
//启用全局方法安全注解，就可以在方法上使用注解来对请求进行过滤
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ResourceServerConfiguration extends ResourceServerConfigurerAdapter {

//    @Autowired
//    private TokenStore tokenStore;

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.csrf().disable();
        // 配置order访问控制，必须认证后才可以访问
        http.authorizeRequests()
                .antMatchers( ...antPatterns: "/order/**","/user/**").authenticated();
    }
```

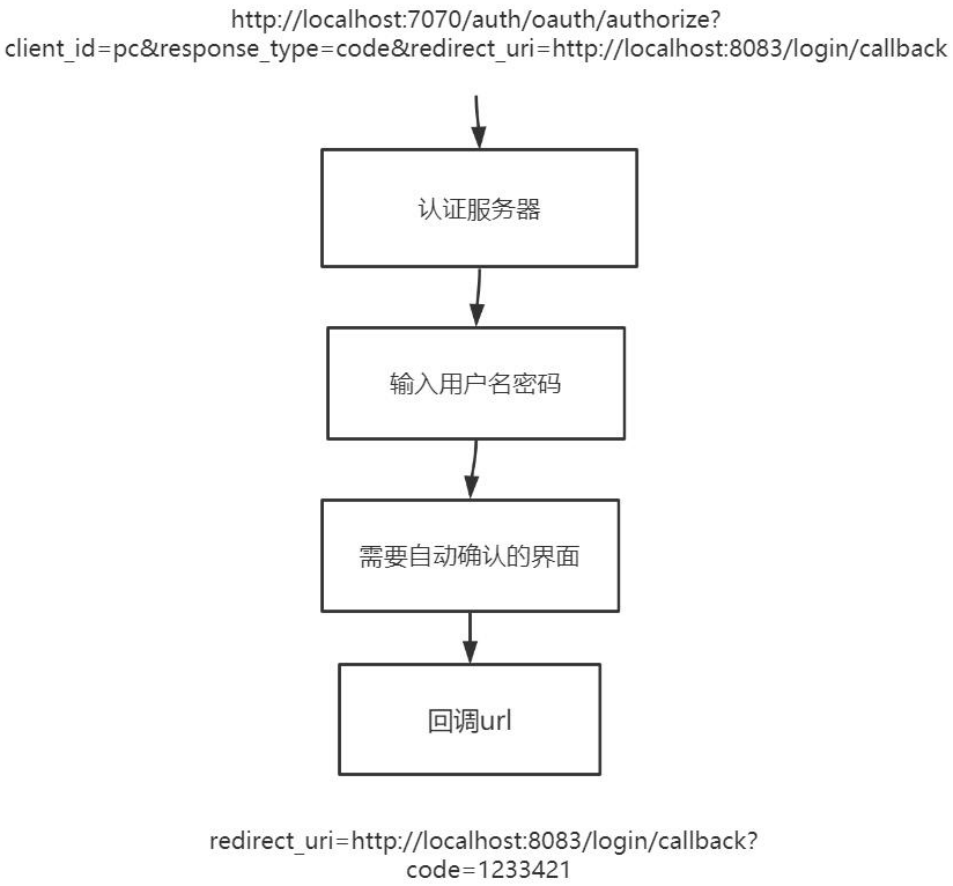
认证服务器和下游系统权限校验流程

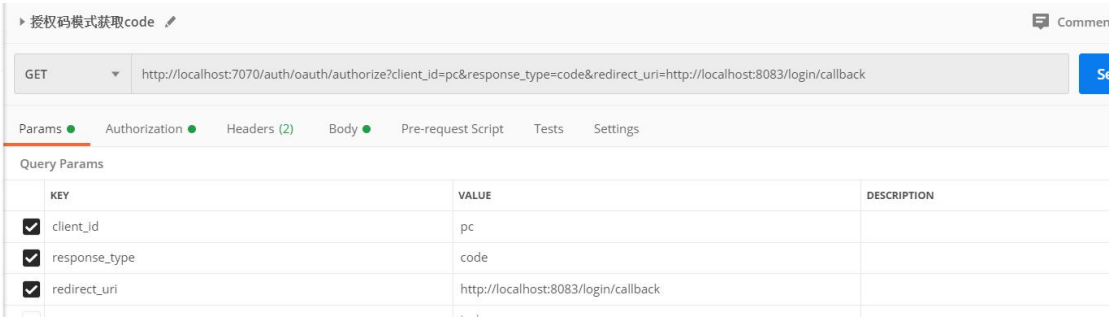


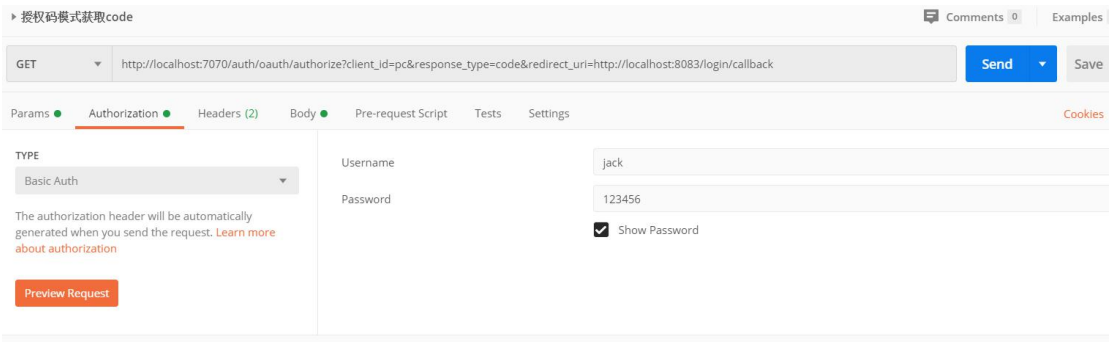
1、zuul 携带 token 请求下游系统，被下游系统 filter 拦截
2、下游系统过滤器根据配置中的 user-info-uri 请求到认证服务器
3、请求到认证服务器被 filter 拦截进行 token 校验，把 token 对应的用户、和权限从数据库查询出来封装到 Principal
4、认证服务器 token 校验通过后过滤器放行执行 security/check 接口，把 principal 对象返回
5、下游系统接收到 principal 对象后就知道该 token 具备的权限了，就可以进行相应用户对应的 token 的权限执行

27、授权码模式获取 token

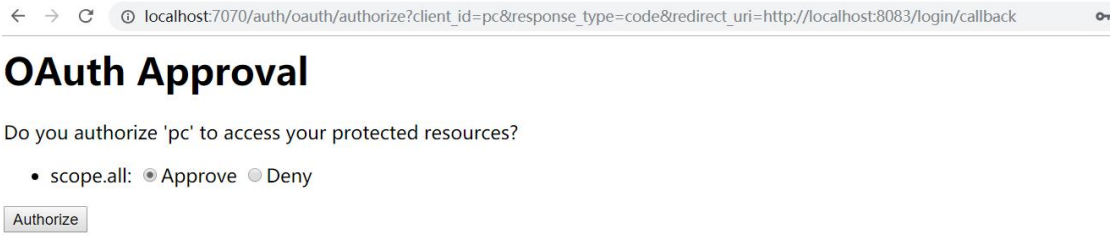
授权码模式获取 token，在获取 token 之前需要有一个获取 code 的过程。
1、获取 code 的流程如下：





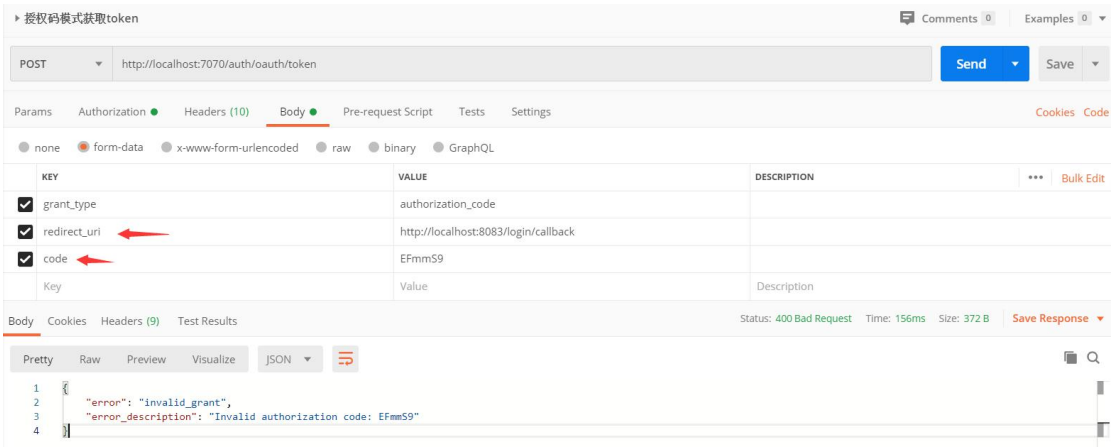


1、用户请求获取 code 的链接
http://localhost:7070/auth/oauth/authorize?client_id=pc&response_type=code&redirect_uri=http://localhost:8083/login/callback
2、提示要输入用户名密码
3、用户名秘密成功则会弹出界面



4、点击 approve 则会回调 redirect_uri 对应的回调地址并且把 code 附带到该回调地址里面

2、根据获取到的 code 获取 token
这里必须带上 redirect_uri 和 code，其他就跟前面的类似



其他配置跟密码模式的是一样的，拿到 token 后就可以访问了。

1、客户端模式
一般用在无需用户登录的系统做接口的安全校验，因为 token 只需要跟客户端绑定，控制粒度不够细

2、密码模式

密码模式，token 是跟用户绑定的，可以根据不同用户的角色和权限来控制不同用户的访问权限，相对来说控制粒度更细

3、授权码模式

授权码模式更安全，因为前面的密码模式可能会存在密码泄露后，别人拿到密码也可以照样的申请到 token 来进行接口访问，而授权码模式用户提供用户名和密码获取后，还需要有一个回调过程，这个回调你可以想象成是用户的手机或者邮箱的回调，只有用户本人能收到这个 code，即使用户名密码被盗也不会影响整个系统的安全。

28、JWT 模式

JWT：json web token 是一种无状态的权限认证方式，一般用于前后端分离，时效性比较端的权限校验，jwt 模式获取 token 跟前面的，客户端，密码，授权码模式是一样的，只是需要配置秘钥：

1、生成秘钥文件

cd 到 jdk 的 bin 目录执行该指令，会在 bin 目录下生成 micro-jwt.jks 文件，把该文件放到认证服务工程里面的 resources 目录下：

```
keytool -genkeypair -alias micro-jwt
        -validity 3650
        -keyalg RSA
        -dname "CN=jwt,OU=jtw,O=jwt,L=zurich,S=zurich, C=CH"
        -keypass 123456
        -keystore micro-jwt.jks
        -storepass 123456
```

2、生成公钥

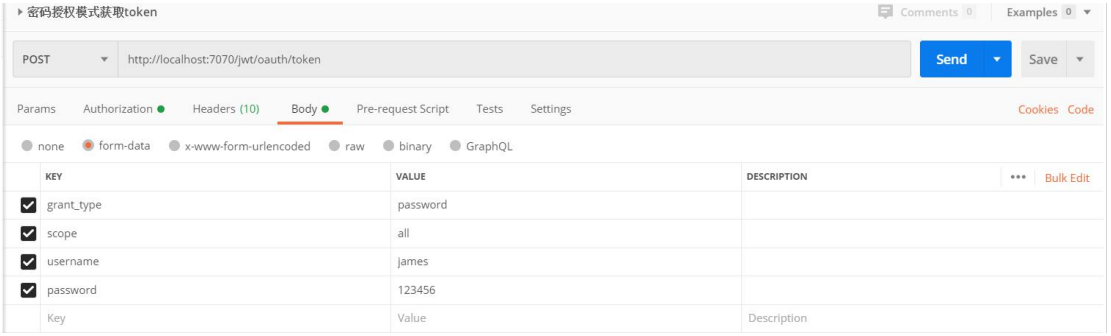
```
keytool -list -rfc --keystore micro-jwt.jks | openssl x509 -inform pem -pubkey
```

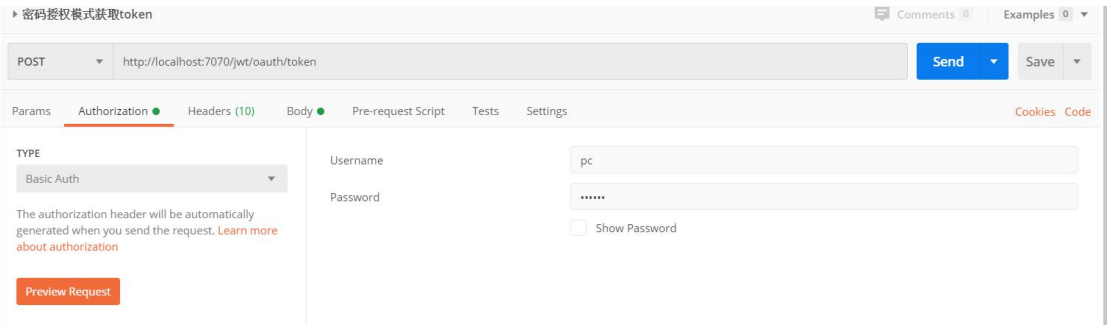
把生成的公钥内容放到 public.cert 文件中，内容如下：

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEApgAw3XUjR29upvmFIyaq
IaX8xKHbv27vGwDluG/LYVrwhpxZGyM5XEfTsDicnL8YMjCxATELoB5H45cofb0E
eBh/WXBZymndqm6QTKlg9xaMEPtvG9BPE+G8pzb/lA54gpAS68dQRzeTq4eVSYcL
z0mgjGqXdXIcyExDDitEhlx/z/lqGlaV1XA8O0RIBQoOyZCPIYk+tkq2l4ip7aBq
fgAOlox8HmMZ/z6Iic1EXG9deSSLZSHPcnF9iaaU2LdXmYuVWaBVro7zH+n/q3Km
Obmq34EqCX80tesS+tI1VDKWCzg+9reHMPtIAjABOjx0aUy/+KAMC8gOGLBnpIKs
1QIDAQAB
-----END PUBLIC KEY-----
```

把公钥文件放到客户端的 resources 目录下。

密码模式获取 jwt token：






```
{
    "access_token":
        "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1ODQ5ODc0ODMsInVzZXJfbmFtZSI6ImphbWVzIiwiYXV0aG9yaXRpZXMiOlsiUk9MRV9BRE1JTiJdLCJqdGkiOiIxNTAwNDcwOC1kZjVlLTQ1NzMtODExZi0xOWExMDA2ZjI3NmMiLCJjbGllbnRfaWQiOiJwYyIsInNjb3BlIjpbImFsbCJdfQ.
        HoUFEnGVG2FLCOvtIK02RmZGovpWUvcsH0TO-jyes1rj1ZqT_GeQ5uU0LMHIddZ0nYOBXCYJgR5vQkC-OOT64LpP0ypLbp9mPbEtYrzl3iT91cqpb_gcBFDZR7Wzi5eW9_B7BtfF9BvgEp51KicnpYgsN7yb4t5OXcn1Ves4uYSeNG96N9Yt0bgiA34-r8cZfA8_UePMY1sZRS3jgmBt--TcjXqJy-GRcL6_ilGgbwQyt-znOqxOxUg7glm9Zixbf27FmPkB0mqJ2qsNqqLz3Cc_RMTi24myRMVW6vS1x789s6tEh74lIwdEAzO73q_HPAvmOJO0RQNow9LhXFve6g",
    "token_type": "bearer",
    "refresh_token":
        "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX25hbWUiOiJqYW1lcyIsInNjb3BlIjpbImFsbCJdLCJhdGkiOiIxNTAwNDcwOC1kZjVlLTQ1NzMtODExZi0xOWExMDA2ZjI3NmMiLCJleHAiOjE1ODc1MzYyODMsImF1dGhvcml0aWVzIjpbIlJPTEVfQURNSU4iXSwianRpIjoiMmJiYmQ3YzUtNjg1MC00ZjBiLWI1OTQtYTIwMmNhM2M1ZDE0IiwiY2xpZW50X2lkIjoicGMifQ.
        NQsHEAXayqDGOLx2EhyalRZiRIEA8tUs6qqqwjRXgOBg4ayhzniqymgwcwRuo2ft-pXWRaCICC7rSCd1lgXJ3S57IhPmn4N3lAiffSJMa6CDTY4clwu2Os6khzrZvpZ819voA6YK75xN3thW0SV5k_s5ESoVu2LU1CpDprQArgARD5C99bb-Nths01wJgvMG8EDmftu2gMX5WI4x0c7CbS28Ei8s34LMRtdA1gQJM5h_Ze5w9cowTAbP6ZdcrK5qP9w_f2pwxFPyasxlhwcWcAlf-cl-58iRHwb7Ptsc22uPEyFD5eXTZHcGkkB4avbNWIqEGhmqgavhxuloATBCAw",
    "expires_in": 43199,
    "scope": "all",
    "jti": "15004708-df5e-4573-811f-19a1006f276c"
}
```



```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1ODQ5ODc0ODMsInVzZXJfbmFtZSI6ImphbWVzIiwiYXV0aG9yaXRpZXMiOlsiUk9MRV9BRE1JTiJdLCJqdGkiOiIxNTAwNDcwOC1kZjVlLTQ1NzMtODExZi0xOWExMDA2ZjI3NmMiLCJjbGllbnRfaWQiOiJwYyIsInNjb3BlIjpbImFsbCJdfQ.HoUFEnGVG2FLCOvtIK02RmZGovpWUvcsH0TO-jyes1rj1ZqT_GeQ5uU0LMHIddZ0nYOBXCYJgR5vQkC-OOT64LpP0ypLbp9mPbEtYrzl3iT91cqpb_gcBFDZR7Wzi5eW9_B7BtfF9BvgEp51KicnpYgsN7yb4t5OXcn1Ves4uYSeNG96N9Yt0bgiA34-r8cZfA8_UePMY1sZRS3jgmBt--TcjXqJy-GRcL6_ilGgbwQyt-znOqxOxUg7glm9Zixbf27FmPkB0mqJ2qsNqqLz3Cc_RMTi24myRMVW6vS1x789s6tEh74lIwdEAzO73q_HPAvmOJO0RQNow9LhXFve6g
```


Jwt 的 token 信息分成三个部分，用“.”号分割的。

第一步部分：头信息，通过 base64 加密生成

第二部分：有效载荷，通过 base64 加密生成

第三部分：签名，根据头信息中的加密算法通过，RSA(base64(头信息) + “.” + base64(有效载荷))生成的第三部分内容

可以到 jwt 的官网看看这三部分信息的具体内容：jwt 官网　jwt.io

29、链路追踪

其实链路追踪就是日志追踪，微服务下日志跟踪，微服务系统之间的调用变得非常复杂，往往一个功能的调用要涉及到多台微服务主机的调用，那么日志追踪也就要在多台主机之间进行，人为的去每台主机查看日志这种工作几乎是不能完成的工作，所以需要有专门的日志监控工具，这里讲的就是 zipkin 工具。

客户端添加 pom 依赖

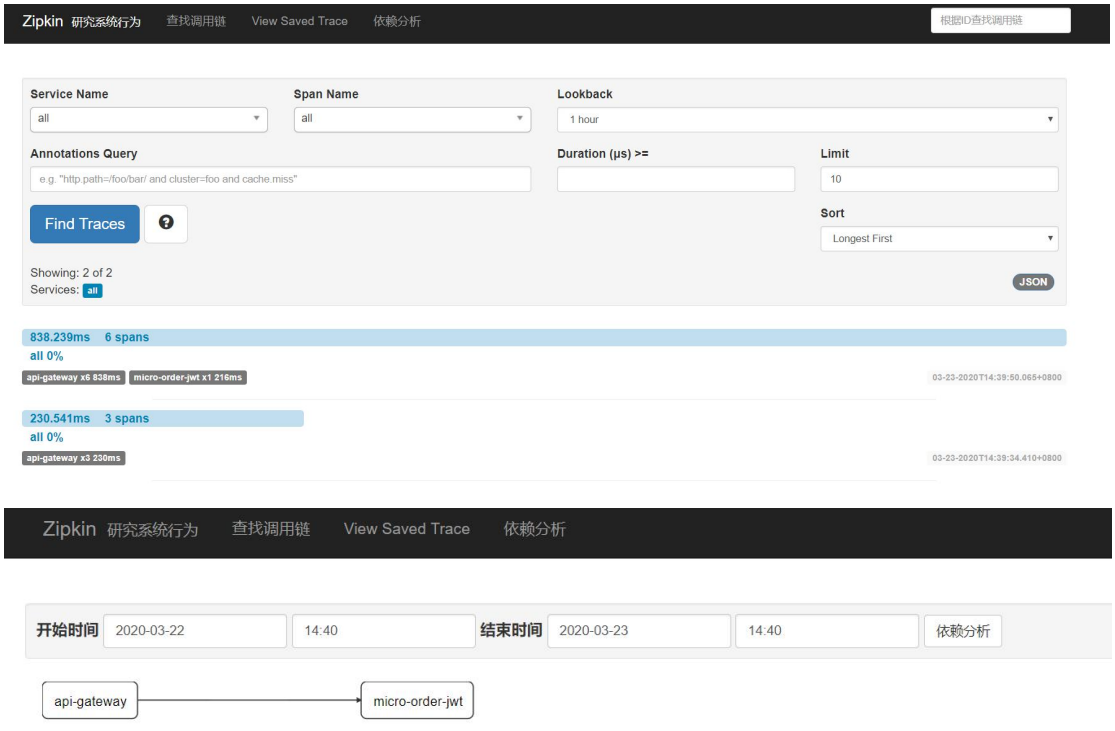
```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

客户端使用链路追踪只需要修改 properties 配置即可

```
#收集日志的百分比发送给 zipkin
spring.sleuth.sampler.percentage=1.0
# 若在同一个注册中心的话可以启用自动发现，省略 base-url
#spring.zipkin.locator.discovery.enabled=true
spring.zipkin.base-url=http://localhost:9411/
```

Zipkin 收集链路追踪日志，zipkin 启动

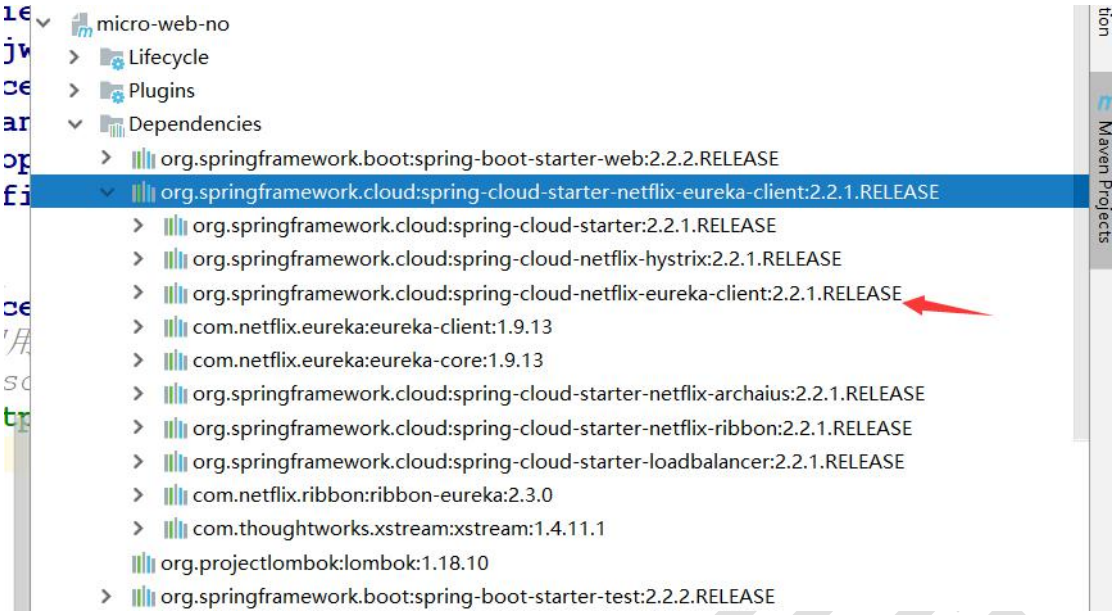
Java -jar zipkin.jar

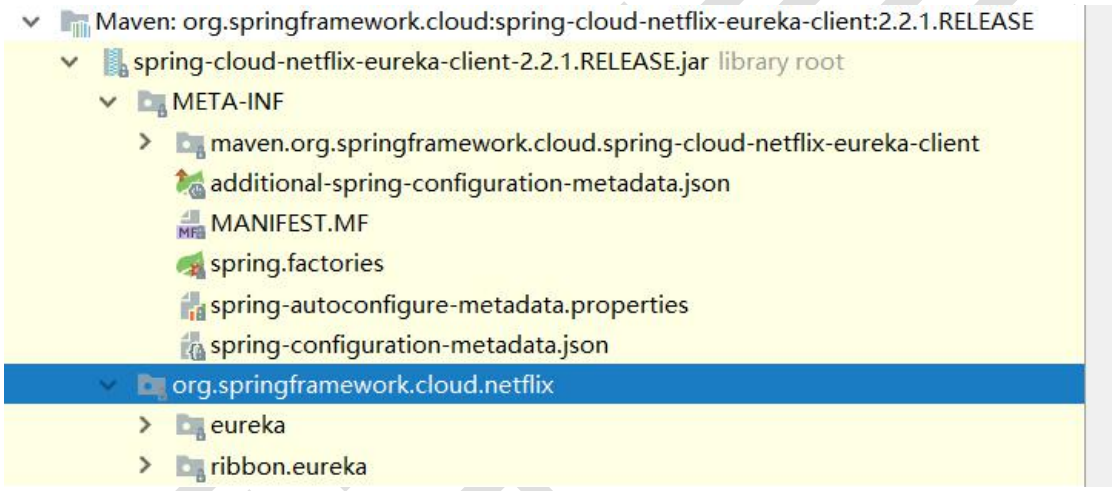


30、Eureka 源码

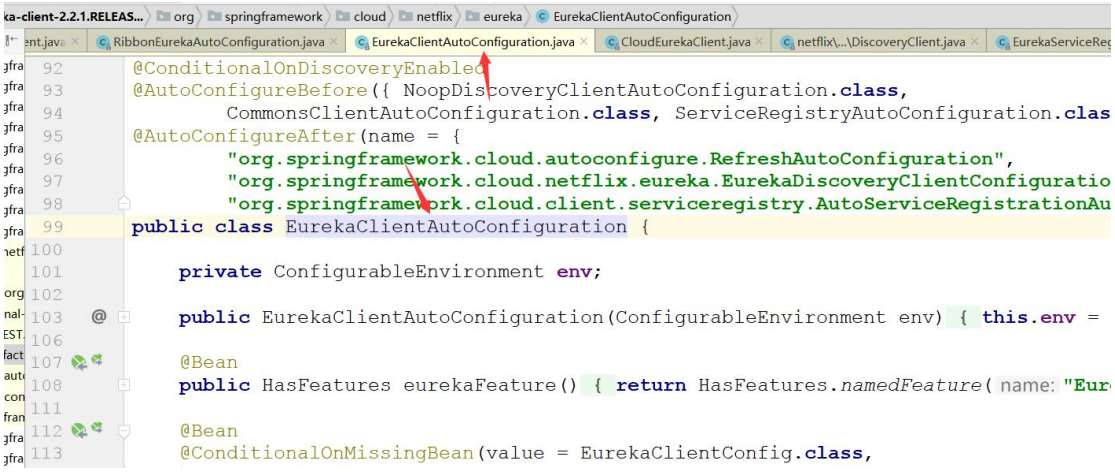
Eureka 客户端源码

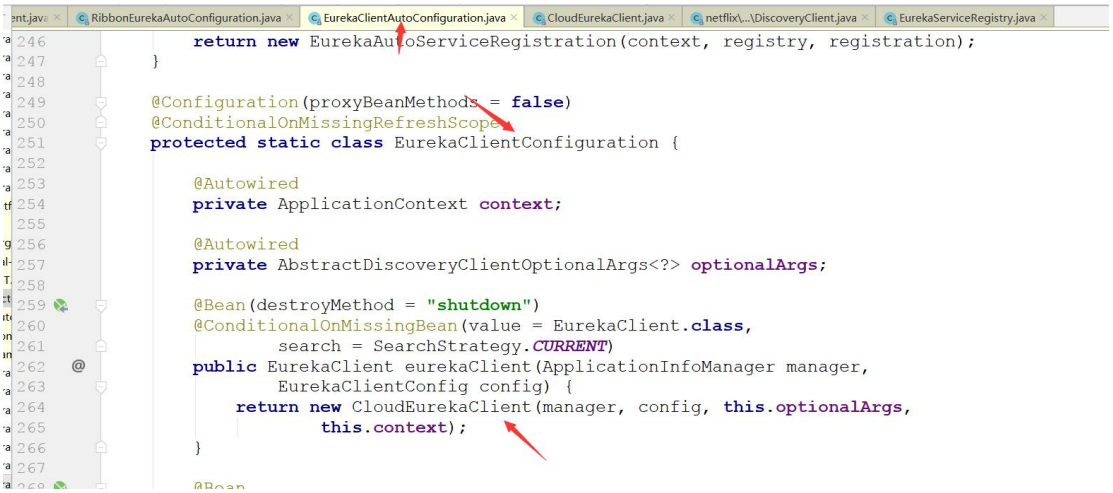
SPI 加载 spring.factories


micro-web-no
Lifecycle
Plugins
Dependencies
org.springframework.boot:spring-boot-starter-web:2.2.2.RELEASE
org.springframework.cloud:spring-cloud-starter-netflix-eureka-client:2.2.1.RELEASE
org.springframework.cloud:spring-cloud-starter:2.2.1.RELEASE
org.springframework.cloud:spring-cloud-netflix-hystrix:2.2.1.RELEASE
org.springframework.cloud:spring-cloud-netflix-eureka-client:2.2.1.RELEASE
com.netflix.eureka:eureka-client:1.9.13
com.netflix.eureka:eureka-core:1.9.13
org.springframework.cloud:spring-cloud-starter-netflix-archaius:2.2.1.RELEASE
org.springframework.cloud:spring-cloud-starter-netflix-ribbon:2.2.1.RELEASE
org.springframework.cloud:spring-cloud-starter-loadbalancer:2.2.1.RELEASE
com.netflix.ribbon:ribbon-eureka:2.3.0
com.thoughtworks.xstream:xstream:1.4.11.1
org.projectlombok:lombok:1.18.10
org.springframework.boot:spring-boot-starter-test:2.2.2.RELEASE
Maven Projects

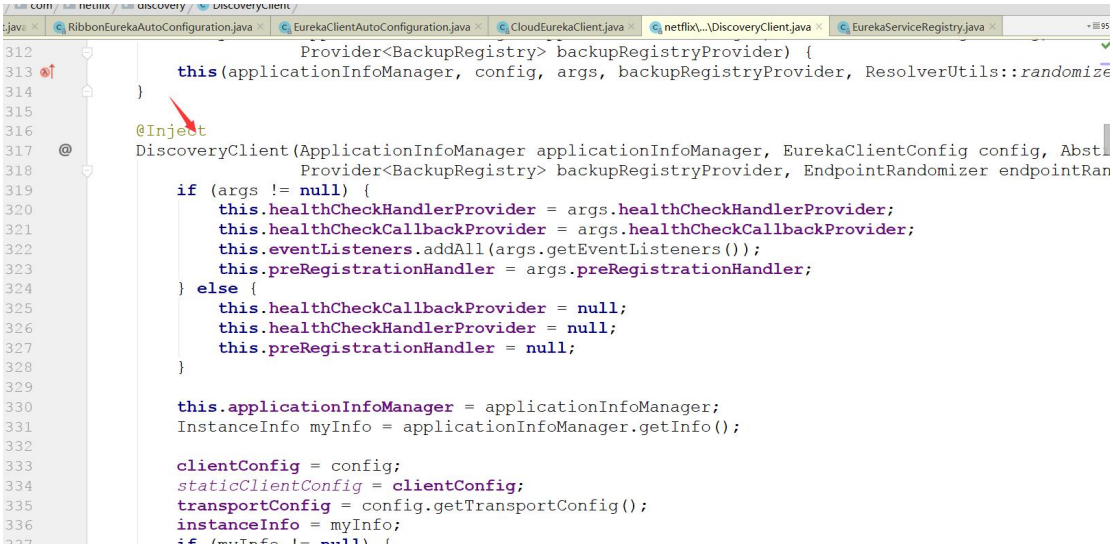

Maven: org.springframework.cloud:spring-cloud-netflix-eureka-client:2.2.1.RELEASE
spring-cloud-netflix-eureka-client-2.2.1.RELEASE.jar library root
META-INF
maven.org.springframework.cloud.spring-cloud-netflix-eureka-client
additional-spring-configuration-metadata.json
MANIFEST.MF
spring.factories
spring-autoconfigure-metadata.properties
spring-configuration-metadata.json
org.springframework.cloud.netflix
eureka
ribbon.eureka

着重看该类


org
springframework
cloud
netflix
eureka
EurekaClientAutoConfiguration
RibbonEurekaAutoConfiguration.java
EurekaClientAutoConfiguration.java
CloudEurekaClient.java
@ConditionalOnDiscoveryEnabled
@AutoConfigureBefore({ NoopDiscoveryClientAutoConfiguration.class,
CommonsClientAutoConfiguration.class, ServiceRegistryAutoConfiguration.clas
@AutoConfigureAfter(name = {
"org.springframework.cloud.autoconfigure.RefreshAutoConfiguration",
"org.springframework.cloud.netflix.eureka.EurekaDiscoveryClientConfiguratio
"org.springframework.cloud.client.serviceregistry.AutoServiceRegistrationAu
public class EurekaClientAutoConfiguration {
private ConfigurableEnvironment env;
public EurekaClientAutoConfiguration(ConfigurableEnvironment env) { this.env =
@Bean
public HasFeatures eurekaFeature() { return HasFeatures.namedFeature(name: "Eur
@Bean
@ConditionalOnMissingBean(value = EurekaClientConfig.class,



看该类的父类 DiscoveryClient



这里面有三个比较重要的定时器，负责服务注册、续约和服务拉取

```
// default size of 2 - 1 each for heartbeat and cacheRefresh
scheduler = Executors.newScheduledThreadPool( corePoolSize: 2,
        new ThreadFactoryBuilder()
                .setNameFormat("DiscoveryClient-%d")
                .setDaemon(true)
                .build());

heartbeatExecutor = new ThreadPoolExecutor(
         corePoolSize: 1, clientConfig.getHeartbeatExecutorThreadPoolSize(),  keepAliveTime:
        new SynchronousQueue<Runnable>(),
        new ThreadFactoryBuilder()
                .setNameFormat("DiscoveryClient-HeartbeatExecutor-%d")
                .setDaemon(true)
                .build()
);  // use direct handoff

cacheRefreshExecutor = new ThreadPoolExecutor(
         corePoolSize: 1, clientConfig.getCacheRefreshExecutorThreadPoolSize(),  keepAliveTi
        new SynchronousQueue<Runnable>(),
        new ThreadFactoryBuilder()
                .setNameFormat("DiscoveryClient-CacheRefreshExecutor-%d")
                .setDaemon(true)
                .build()
);  // use direct handoff
```

接着看下面的一个方法

```
// finally, init the schedule tasks (e.g. cluster resolvers, heartbeat,
initScheduledTasks();

try {
    Monitors.registerObject(this);
} catch (Throwable e) {
    logger.warn("Cannot register timers", e);
}
```

1、服务拉取

```
if (clientConfig.shouldFetchRegistry()) {
    // registry cache refresh timer
    int registryFetchIntervalSeconds = clientConfig.getRegistryFetchIntervalSeconds();
    int expBackOffBound = clientConfig.getCacheRefreshExecutorExponentialBackOffBound();
    scheduler.schedule(
            new TimedSupervisorTask(
                    name: "cacheRefresh",
                    scheduler,
                    cacheRefreshExecutor,
                    registryFetchIntervalSeconds,
                    TimeUnit.SECONDS,
                    expBackOffBound,
                    new CacheRefreshThread()
            ),
            registryFetchIntervalSeconds, TimeUnit.SECONDS);
}
```

```
class CacheRefreshThread implements Runnable {
    public void run() {
        refreshRegistry();
    }
}
```

refreshRegistry 里面的 fetchRegistry 方法

```
boolean success = fetchRegistry(remoteRegionsModified);
if (success) {
```

全量拉取和增量拉取

```
private boolean fetchRegistry(boolean forceFullRegistryFetch) {
    Stopwatch tracer = FETCH_REGISTRY_TIMER.start();

    try {
        // If the delta is disabled or if it is the first time, get all
        // applications
        Applications applications = getApplications();

        if (clientConfig.shouldDisableDelta()
                || (!Strings.isNullOrEmpty(clientConfig.getRegistryRefreshSingleVipAddres
                || forceFullRegistryFetch
                || (applications == null)
                || (applications.getRegisteredApplications().size() == 0)
                || (applications.getVersion() == -1)) //Client application does not have
        {
            logger.info("Disable delta property : {}", clientConfig.shouldDisableDelta())
            logger.info("Single vip registry refresh property : {}", clientConfig.getRegi
            logger.info("Force full registry fetch : {}", forceFullRegistryFetch);
            logger.info("Application is null : {}", (applications == null));
            logger.info("Registered Applications size is zero : {}",
                    (applications.getRegisteredApplications().size() == 0));
            logger.info("Application version is -1: {}", (applications.getVersion() == -1
            getAndStoreFullRegistry();
        } else {
            getAndUpdateDelta(applications);
        }
```

```
private void getAndStoreFullRegistry() throws Throwable {
    long currentUpdateGeneration = fetchRegistryGeneration.get();

    logger.info("Getting all instance registry info from the eureka server");

    Applications apps = null;
    EurekaHttpResponse<Applications> httpResponse = clientConfig.getRegistryRefreshSingleVipAddress() == null
            ? eurekaTransport.queryClient.getApplications(remoteRegionsRef.get())
            : eurekaTransport.queryClient.getVip(clientConfig.getRegistryRefreshSingleVipAddress(), remoteRegi
    if (httpResponse.getStatusCode() == Status.OK.getStatusCode()) {
        apps = httpResponse.getEntity();
    }
    logger.info("The response status is {}", httpResponse.getStatusCode());

    if (apps == null) {
        logger.error("The application is null for some reason. Not storing this information");
    } else if (fetchRegistryGeneration.compareAndSet(currentUpdateGeneration, update: currentUpdateGeneration +
        localRegionApps.set(this.filterAndShuffle(apps));
        logger.debug("Got full registry with apps hashcode {}", apps.getAppsHashCode());
    } else {
        logger.warn("Not updating applications as another thread is updating it already");
    }
}
```

从服务器获取服务列表后换成到 localRegionApps 中，后面 ribbon、feign、config、zuul 都用到了这个 localRegionApps

getApplications 方法调用，最终会走到这个类用 get 请求调用到 eureka 服务端

```
flix\...\DiscoveryClient.java | EurekaHttpClientDecorator.java | AbstractJerseyEurekaHttpClient.java | EurekaClientConfigBean.java | EurekaServiceRegistry.java
}

private EurekaHttpResponse<Applications> getApplicationsInternal(String urlPath, String[] regions) {
    ClientResponse response = null;
    String regionsParamValue = null;
    try {
        WebResource webResource = jerseyClient.resource(serviceUrl).path(urlPath);
        if (regions != null && regions.length > 0) {
            regionsParamValue = StringUtil.join(regions);
            webResource = webResource.queryParam("regions", regionsParamValue);
        }
        Builder requestBuilder = webResource.getRequestBuilder();
        addExtraHeaders(requestBuilder);
        response = requestBuilder.accept(MediaType.APPLICATION_JSON_TYPE).get(ClientResponse.class);

        Applications applications = null;
        if (response.getStatus() == Status.OK.getStatusCode() && response.hasEntity()) {
            applications = response.getEntity(Applications.class);
        }
        return anEurekaHttpResponse(response.getStatus(), Applications.class)
                .headers(headersOf(response))
                .entity(applications)
                .build();
    } finally {
```

对应的请求地址：

```
requestBuilder = {WebResource$Builder@8229}
  this$0 = {WebResource@8228} "http://admin:admin@localhost:8763/eureka/apps/"
  entity = null
  metadata = {OutBoundHeaders@8262} size = 0
```

2、服务注册和续约

```
if (clientConfig.shouldRegisterWithEureka()) {
    int renewalIntervalInSecs = instanceInfo.getLeaseInfo().getRenewalIntervalInSecs();
    int expBackOffBound = clientConfig.getHeartbeatExecutorExponentialBackOffBound();
    logger.info("Starting heartbeat executor: " + "renew interval is: {}", renewalIntervalIn

    // Heartbeat timer
    scheduler.schedule(
            new TimedSupervisorTask(
                    name: "heartbeat",
                    scheduler,
                    heartbeatExecutor,
                    renewalIntervalInSecs,
                    TimeUnit.SECONDS,
                    expBackOffBound,
                    new HeartbeatThread()
            ),
            renewalIntervalInSecs, TimeUnit.SECONDS);

    // InstanceInfo replicator
    instanceInfoReplicator = new InstanceInfoReplicator(
            discoveryClient: this,
            instanceInfo,
            clientConfig.getInstanceInfoReplicationIntervalSeconds(),
            burstSize: 2); // burstSize

    statusChangeListener = new ApplicationInfoManager.StatusChangeListener() {
        @Override
```

```
private class HeartbeatThread implements Runnable {

    public void run() {
        if (renew()) {
            lastSuccessfulHeartbeatTimestamp = System.currentTimeMillis();
        }
    }
}
```

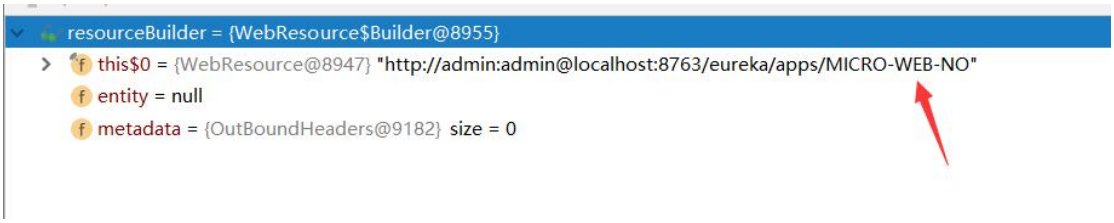
```
boolean renew() {
    EurekaHttpResponse<InstanceInfo> httpResponse;
    try {
        httpResponse = eurekaTransport.registrationClient.sendHeartBeat(instanceInfo.getAppName(), instance
        logger.debug(PREFIX + "{} - Heartbeat status: {}", appPathIdentifier, httpResponse.getStatusCode())
        if (httpResponse.getStatusCode() == Status.NOT_FOUND.getStatusCode()) {
            REREGISTER_COUNTER.increment();
            logger.info(PREFIX + "{} - Re-registering apps/{}", appPathIdentifier, instanceInfo.getAppName(
            long timestamp = instanceInfo.setIsDirtyWithTime();
            boolean success = register();
            if (success) {
                instanceInfo.unsetIsDirty(timestamp);
            }
            return success;
        }
        return httpResponse.getStatusCode() == Status.OK.getStatusCode();
    } catch (Throwable e) {
        logger.error(PREFIX + "{} - was unable to send heartbeat!", appPathIdentifier, e);
        return false;
    }
}
```

先看看 register 方法，服务注册，最终会掉到该类：

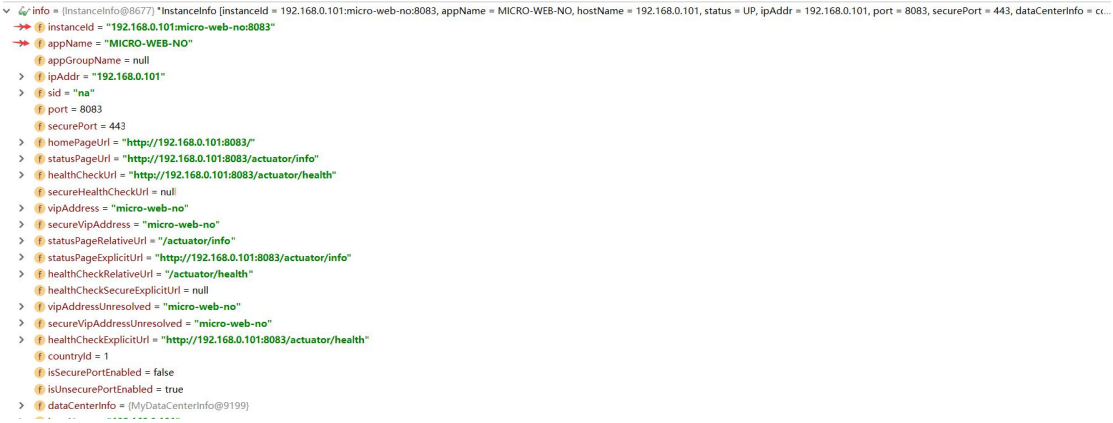
Post 请求到服务端

```
public EurekaHttpResponse<Void> register(InstanceInfo info) {
    String urlPath = "apps/" + info.getAppName();
    ClientResponse response = null;
    try {
        Builder resourceBuilder = jerseyClient.resource(serviceUrl).path(urlPath).getRequestBuilder();
        addExtraHeaders(resourceBuilder);
        response = resourceBuilder
                .header( name: "Accept-Encoding",  value: "gzip")
                .type(MediaType.APPLICATION_JSON_TYPE)
                .accept(MediaType.APPLICATION_JSON)
                .post(ClientResponse.class, info);
        return anEurekaHttpResponse(response.getStatus()).headers(headersOf(response)).build();
    } finally {
        if (logger.isDebugEnabled()) {
            logger.debug("Jersey HTTP POST {}/{} with instance {}; statusCode={}", serviceUrl, urlPath, inf
                    response == null ? "N/A" : response.getStatus());
        }
        if (response != null) {
            response.close();
        }
    }
}
```

服务注册的地址



服务注册的节点信息

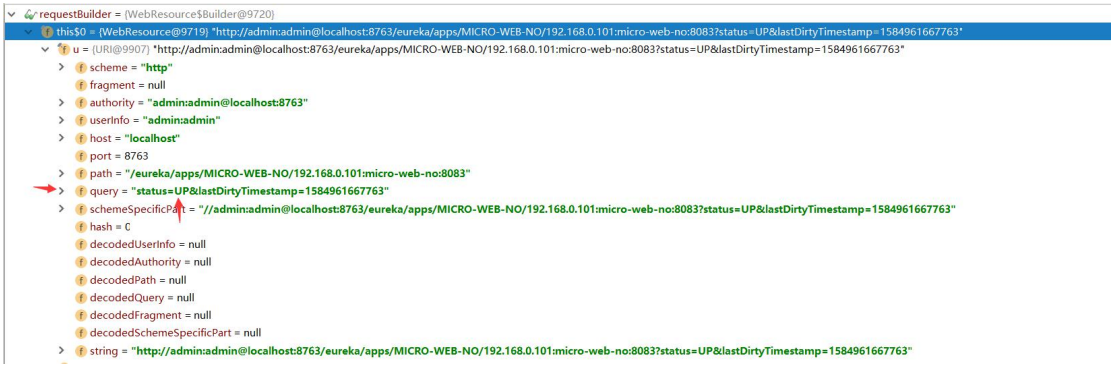


在看看服务续约中的 sendHeartBeat 方法

Put 请求

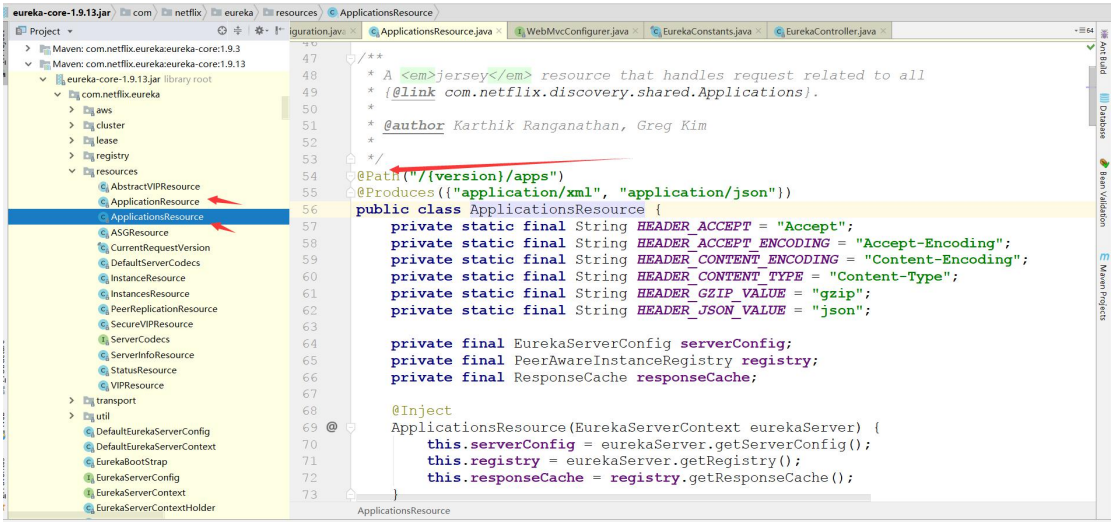
```
public EurekaHttpResponse<InstanceInfo> sendHeartBeat(String appName, String id, InstanceInfo info, Instance
    String urlPath = "apps/" + appName + '/' + id;
    ClientResponse response = null;
    try {
        WebResource webResource = jerseyClient.resource(serviceUrl)
                .path(urlPath)
                .queryParam("status", info.getStatus().toString())
                .queryParam("lastDirtyTimestamp", info.getLastDirtyTimestamp().toString());
        if (overriddenStatus != null) {
            webResource = webResource.queryParam("overriddenstatus", overriddenStatus.name());
        }
        Builder requestBuilder = webResource.getRequestBuilder();
        addExtraHeaders(requestBuilder);
        response = requestBuilder.put(ClientResponse.class);
        EurekaHttpResponseBuilder<InstanceInfo> eurekaResponseBuilder = anEurekaHttpResponse(response.getSta
        if (response.hasEntity()) {
            eurekaResponseBuilder.entity(response.getEntity(InstanceInfo.class));
        }
        return eurekaResponseBuilder.build();
    } finally {
        if (logger.isDebugEnabled()) {
            logger.debug("Jersey HTTP PUT {}/{}; statusCode={}", serviceUrl, urlPath, response == null ? "N/
        }
```

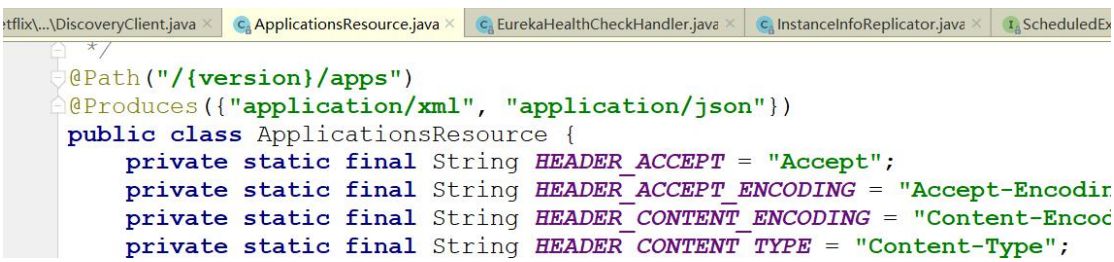
请求信息，心跳只是把节点状态发送到了服务端



Eureka 服务端源码

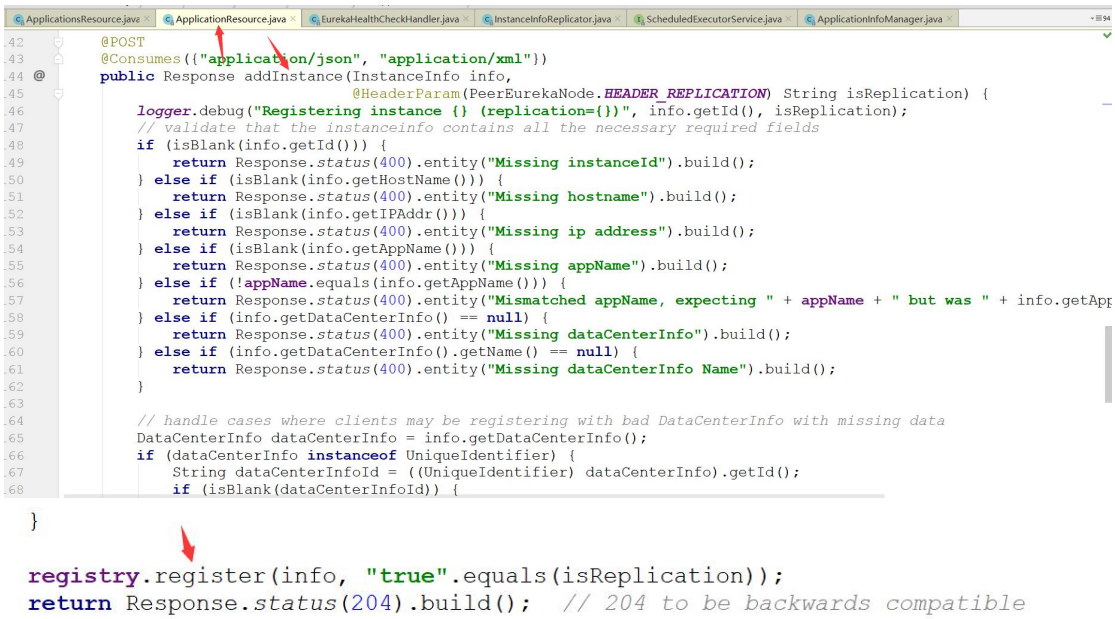
eureka 服务端是通过 JAX-RS 规范接收客户端的请求的





接收服务注册的方法

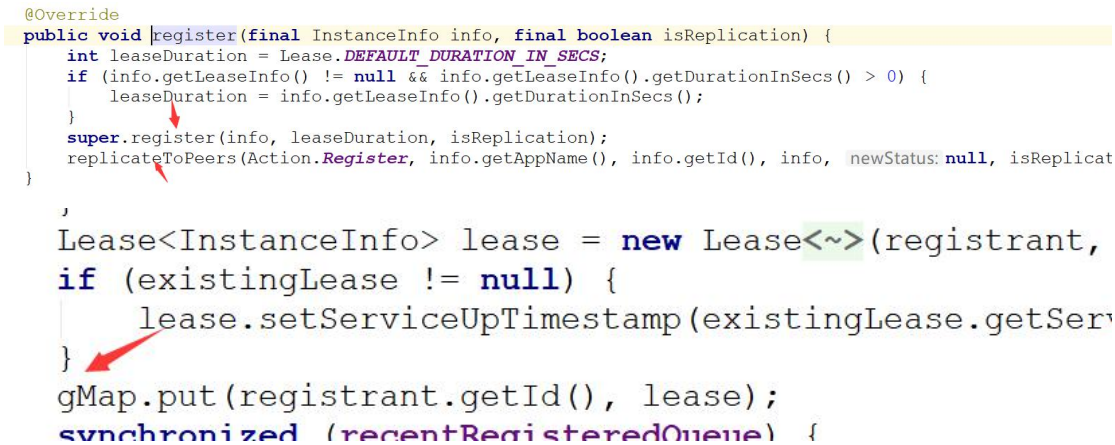
```
@Path("{appId}")
public ApplicationResource getApplicationResource(
        @PathParam("version") String version,
        @PathParam("appId") String appId) {
    CurrentRequestVersion.set(Version.toEnum(version));
    return new ApplicationResource(appId, serverConfig, registry);
}
```



接着看 register 方法

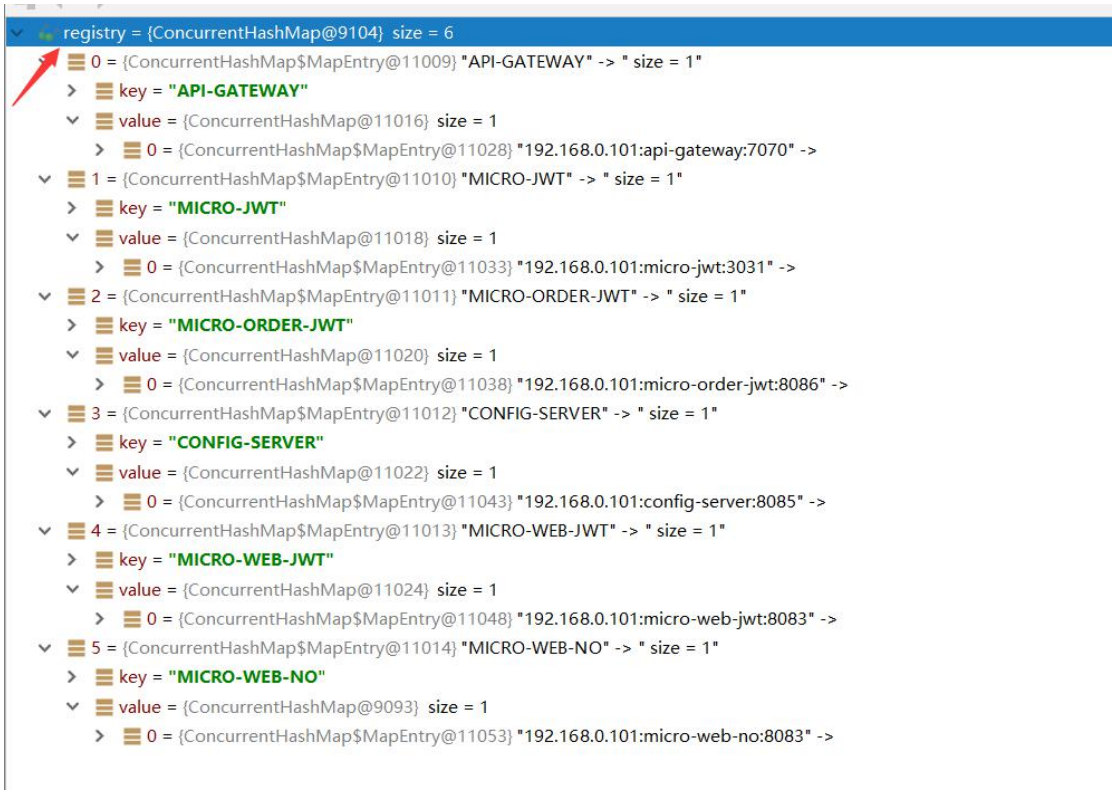
两件事，1、把客户端传递过来的节点信息报错到本地服务列表

2、把该 eureka 的信息复制到其他 eureka 节点上



对应的服务端注册信息如下：

Key 是对应的服务名称，value 则是这个服务对应的服务列表信息



复制 eureka 的节点信息到其他节点

所有节点循环

```
private void replicateToPeers(Action action, String appName, String id,
                              InstanceInfo info /* optional */,
                              InstanceStatus newStatus /* optional */, boolean isReplication) {
    Stopwatch tracer = action.getTimer().start();
    try {
        if (isReplication) {
            numberOfReplicationsLastMin.increment();
        }
        // If it is a replication already, do not replicate again as this will create a poison repl
        if (peerEurekaNodes == Collections.EMPTY_LIST || isReplication) {
            return;
        }

        for (final PeerEurekaNode node : peerEurekaNodes.getPeerEurekaNodes()) {
            // If the url represents this host, do not replicate to yourself.
            if (peerEurekaNodes.isThisMyUrl(node.getServiceUrl())) {
                continue;
            }
            replicateInstanceActionsToPeers(action, appName, id, info, newStatus, node);
        }
```

```
private void replicateInstanceActionsToPeers(Action action, String appName,
                                             String id, InstanceInfo info, InstanceStatus newStatus,
                                             PeerEurekaNode node) {
    try {
        InstanceInfo infoFromRegistry = null;
        CurrentRequestVersion.set(Version.V2);
        switch (action) {
            case Cancel:
                node.cancel(appName, id);
                break;
            case Heartbeat:
                InstanceStatus overriddenStatus = overriddenInstanceStatusMap.get(id);
                infoFromRegistry = getInstanceByAppAndId(appName, id, includeRemoteRegions: false);
                node.heartbeat(appName, id, infoFromRegistry, overriddenStatus, primeConnection: false);
                break;
            case Register:
                node.register(info);
                break;
            case StatusUpdate:
                infoFromRegistry = getInstanceByAppAndId(appName, id, includeRemoteRegions: false);
                node.statusUpdate(appName, id, newStatus, infoFromRegistry);
```

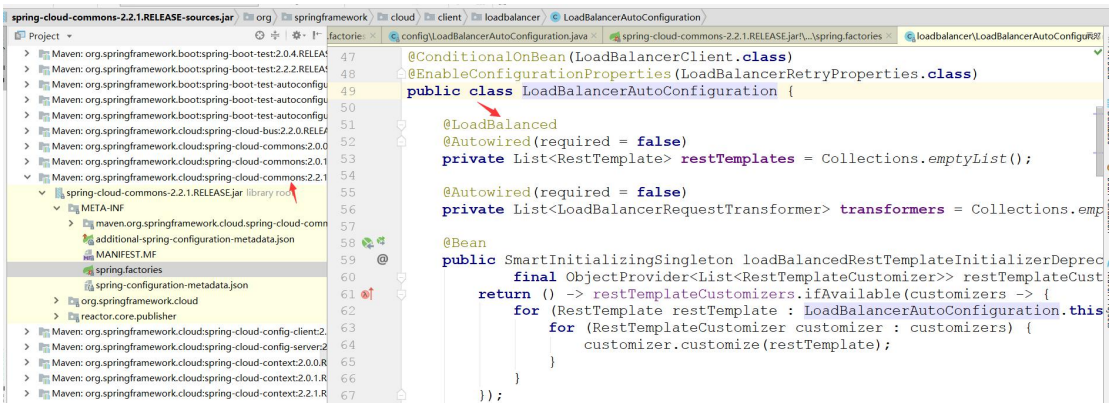
接收续约心跳的方法

```
@Path("{id}")
public InstanceResource getInstanceInfo(@PathParam("id") String id) {
    return new InstanceResource( app: this, id, serverConfig, registry);
}
```

```
@PUT
public Response renewLease(
        @HeaderParam(PeerEurekaNode.HEADER_REPLICATION) String isReplication,
        @QueryParam("overriddenstatus") String overriddenStatus,
        @QueryParam("status") String status,
        @QueryParam("lastDirtyTimestamp") String lastDirtyTimestamp) {
    boolean isFromReplicaNode = "true".equals(isReplication);
    boolean isSuccess = registry.renew(app.getName(), id, isFromReplicaNode);

    // Not found in the registry, immediately ask for a register
    if (!isSuccess) {
        logger.warn("Not Found (Renew): {} - {}", app.getName(), id);
        return Response.status(Status.NOT_FOUND).build();
    }
    // Check if we need to sync based on dirty time stamp, the client
    // instance might have changed some value
    Response response;
    if (lastDirtyTimestamp != null && serverConfig.shouldSyncWhenTimestampDiffers()) {
        response = this.validateDirtyTimestamp(Long.valueOf(lastDirtyTimestamp), isFromReplicaNode);
        // Store the overridden status since the validation found out the node that replicates wins
        if (response.getStatus() == Response.Status.NOT_FOUND.getStatusCode()
                && (overriddenStatus != null)
```

31、Ribbon 源码

老套路先看包依赖，通过 spi 加载进来



这里加上@LoadBalanced 注解的依赖注入，那么就只有加了@LoadBalanced 的实例才能注入进来，其实里面有一个@Qualifier

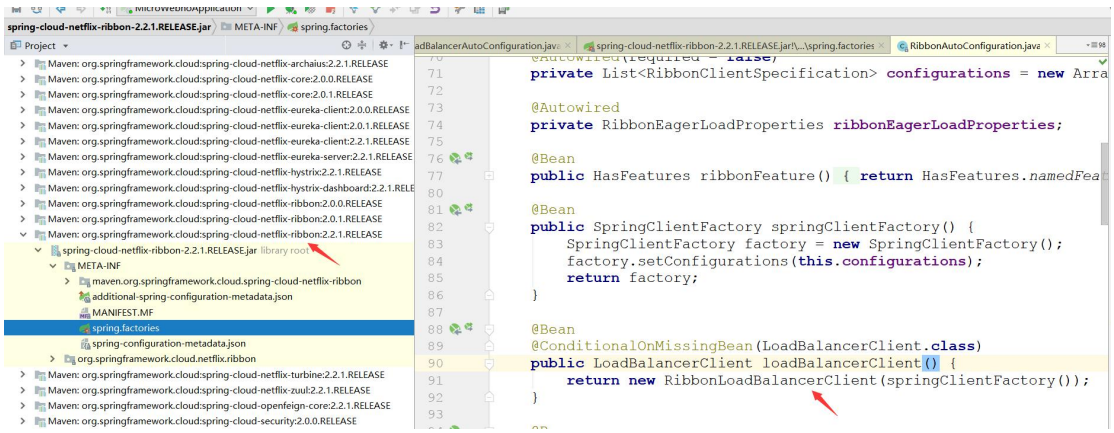
```
    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
//        HttpComponentsClientHttpRequestFactory factory = new HttpComponentsCl
//        factory.setConnectionRequestTimeout(2000);
//        factory.setReadTimeout(4000);
        return new RestTemplate();
    }
```

只是这里加上@LoadBalanced 注解才能依赖注入到源码中的，如果不加这个注解则没有 ribbon 功能，因为源码里面获取不到这个实例了

把拦截器设置到 restTemplate 中

```
                    LoadBalancerClient loadBalancerClient,
                    LoadBalancerRequestFactory requestFactory) {
                return new LoadBalancerInterceptor(loadBalancerClient, requestFactory);
            }

            @Bean
            @ConditionalOnMissingBean
            public RestTemplateCustomizer restTemplateCustomizer(
                    final LoadBalancerInterceptor loadBalancerInterceptor) {
                return restTemplate -> {
                    List<ClientHttpRequestInterceptor> list = new ArrayList<>(
                            restTemplate.getInterceptors());
                    list.add(loadBalancerInterceptor);
                    restTemplate.setInterceptors(list);
                };
            }

        }
```

创建 Ribbon 客户端对象




```
@Autowired(required = false)
private List<RibbonClientSpecification> configurations = new ArrayList<>();

@Autowired
private RibbonEagerLoadProperties ribbonEagerLoadProperties;

@Bean
public HasFeatures ribbonFeature() { return HasFeatures.namedFeature( name: "Ribbon", Ribbon.class); }

@Bean
public SpringClientFactory springClientFactory() {
    SpringClientFactory factory = new SpringClientFactory();
    factory.setConfigurations(this.configurations);
    return factory;
}
```

configurations 引用的实例化对象过程

```
@Configuration
@Conditional(RibbonAutoConfiguration.RibbonClassesConditions.class)
@RibbonClients
@AutoConfigureAfter(
        name = "org.springframework.cloud.netflix.eureka.EurekaClientAutoConf
@AutoConfigureBefore({ LoadBalancerAutoConfiguration.class,
        AsyncLoadBalancerAutoConfiguration.class })
@EnableConfigurationProperties({ RibbonEagerLoadProperties.class,
        ServerIntrospectorProperties.class })
public class RibbonAutoConfiguration {
```

```java
@Configuration(proxyBeanMethods = false)
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE })
@Documented
@Import(RibbonClientConfigurationRegistrar.class)
public @interface RibbonClients {

    RibbonClient[] value() default {};

    Class<?>[] defaultConfiguration() default {};

}
```

```java
private void registerClientConfiguration(BeanDefinitionRegistry registry, Object name,
        Object configuration) {
    BeanDefinitionBuilder builder = BeanDefinitionBuilder
            .genericBeanDefinition(RibbonClientSpecification.class);
    builder.addConstructorArgValue(name);
    builder.addConstructorArgValue(configuration);
    registry.registerBeanDefinition( beanName: name + ".RibbonClientSpecification",
            builder.getBeanDefinition());
}
```

restTemplate 的方法调用源码

```java
@HystrixCommand(fallbackMethod = "queryContentsAsynFallback")
@Override
public Future<String> queryContentsAsyn() {
    return (AsyncResult) () -> {
            log.info("========queryContents==========");
            List<ConsultContent> results = restTemplate.getForObject( url: "http://"
                    + SERVIER_NAME + "/user/queryContent", List.class);
            return JSONObject.toJSONString(results);
    };
}
```

restTemplate 中设置的拦截器，拦截器先调用

```java
private class InterceptingRequestExecution implements ClientHttpRequestExecution {

    private final Iterator<ClientHttpRequestInterceptor> iterator;

    public InterceptingRequestExecution() { this.iterator = interceptors.iterator(); }

    @Override
    public ClientHttpResponse execute(HttpRequest request, byte[] body) throws IOException {
        if (this.iterator.hasNext()) {
            ClientHttpRequestInterceptor nextInterceptor = this.iterator.next();
            return nextInterceptor.intercept(request, body, execution: this);
        }
            if (this.iterator.hasNext()) {
                ClientHttpRequestInterceptor nextInterceptor = this.iterator.next();   nextIr
                return nextInterceptor.intercept(request, body, execution: this);   nextInterce
            }
            else {
```

{LoadBalancerInterceptor@10032}

来到核心源码

```
 * @param request to be executed  request: LoadBalancerRequestFactory$lambda@10045
 * @param hint used to choose appropriate {@link Server} instance  hint: null
 * @return request execution result
 * @throws IOException executing the request may result in an {@link IOException}
 */
public <T> T execute(String serviceId, LoadBalancerRequest<T> request, Object hint)  serviceId: "micro-order-no
        throws IOException {
    ILoadBalancer loadBalancer = getLoadBalancer(serviceId);  serviceId: "micro-order-no"
    Server server = getServer(loadBalancer, hint);
    if (server == null) {
        throw new IllegalStateException("No instances available for " + serviceId);
    }
    RibbonServer ribbonServer = new RibbonServer(serviceId, server,
            isSecure(server, serviceId),
            serverIntrospector(serviceId).getMetadata(server));

    return execute(serviceId, ribbonServer, request);
}
```

获取 ILoadBalancer 类型的实例

```
public ILoadBalancer getLoadBalancer(String name) {
    return getInstance(name, ILoadBalancer.class);
}
```

根据服务名称创建一个容器，然后把容器根据服务名称缓存，这里在调用的时候获取容器的目的是为了拿到最新的服务列表，所以 ribbon 在第一次服务名称调用的时候是比较慢的，涉及到创建容器过程，第二次就直接从缓存里面拿容器对象了。

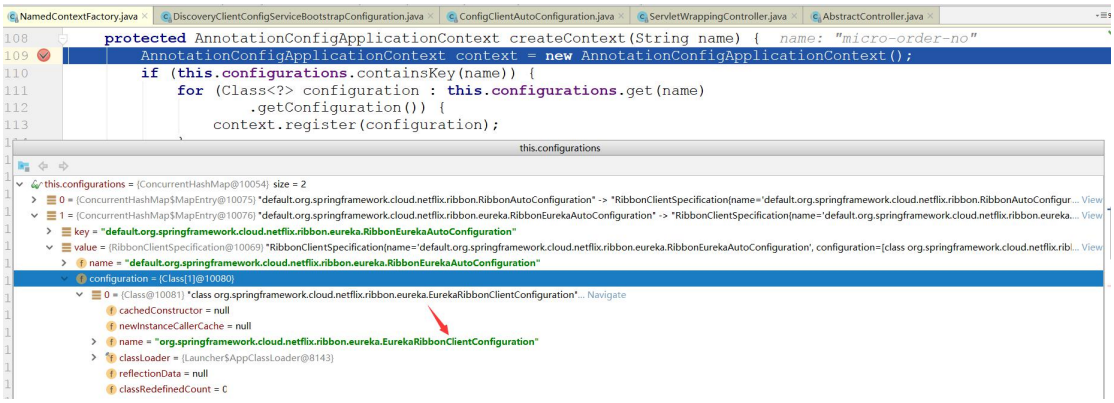
```
public <T> T getInstance(String name, Class<T> type) {  name: "micro-order-no"  type:
    AnnotationConfigApplicationContext context = getContext(name);  name: "micro-orde
    if (BeanFactoryUtils.beanNamesForTypeIncludingAncestors(context,
            type).length > 0) {
        return context.getBean(type);
    }
    return null;
}
```

```
protected AnnotationConfigApplicationContext getContext(String name) {
    if (!this.contexts.containsKey(name)) {
        synchronized (this.contexts) {
            if (!this.contexts.containsKey(name)) {
                this.contexts.put(name, createContext(name));
            }
        }
    }
    return this.contexts.get(name);
}
```

这里创建容器对象，并且把两个比较重要的类 register 进去了，register 其实就是把类变成 beanDefinition 对象，其实就是加载这两个类到容器中，触发这两个类重新获取服务列表

```
108     protected AnnotationConfigApplicationContext createContext(String name) {  name: "micro-order-no"
109         AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
110         if (this.configurations.containsKey(name)) {
111             for (Class<?> configuration : this.configurations.get(name)
112                     .getConfiguration()) {
113                 context.register(configuration);
114             }
115         }
116         for (Map.Entry<String, C> entry : this.configurations.entrySet()) {
117             if (entry.getKey().startsWith("default.")) {
118                 for (Class<?> configuration : entry.getValue().getConfiguration()) {
119                     context.register(configuration);
120                 }
121             }
122         }
123         context.register(PropertyPlaceholderAutoConfiguration.class,
124                 this.defaultConfigType);
125         context.getEnvironment().getPropertySources().addFirst(new MapPropertySource(
126                 this.propertySourceName,
127                 Collections.<~>singletonMap(this.propertyName, name)));
128         if (this.parent != null) {
129             // Uses Environment from parent as well as beans
```

第一个注册的类：

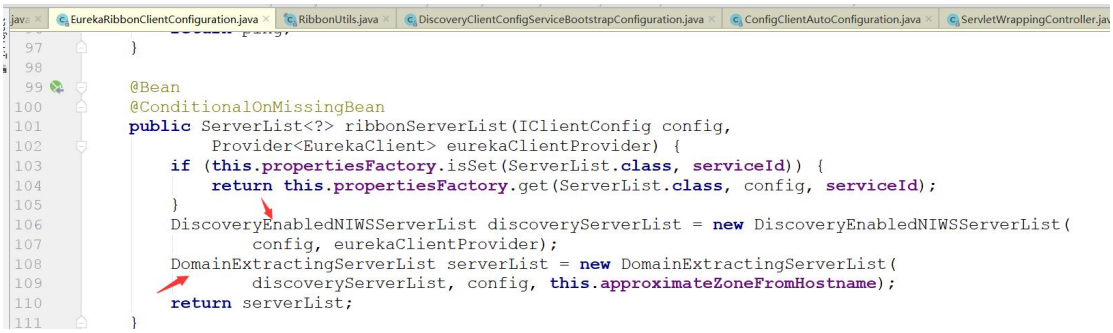


第二个注册的类：

我们看看这两个类：

EurekaRibbonClientConfiguration

创建 serverList 对象



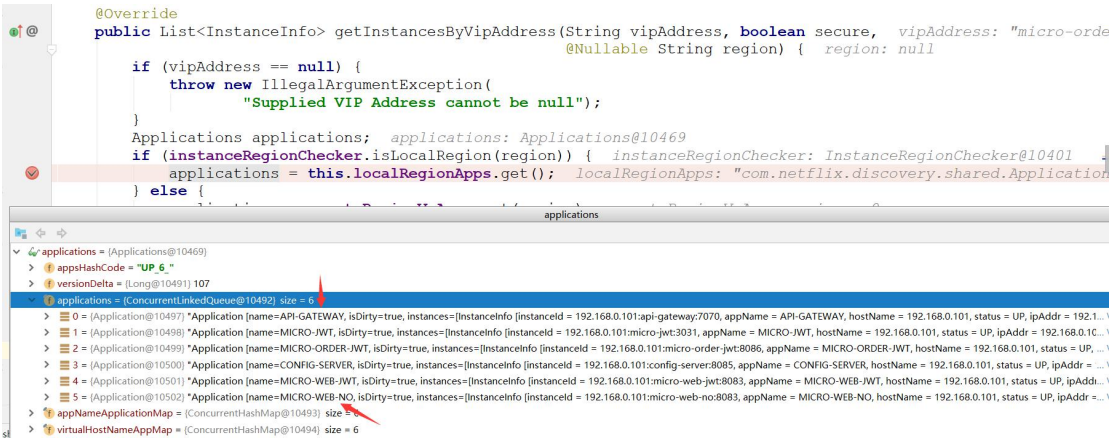
在 DiscoveryEnabledNIWSServerList 类中有一个特别重要的方法

```
@Override
public List<DiscoveryEnabledServer> getUpdatedListOfServers(){
    return obtainServersViaDiscovery();
}

private List<DiscoveryEnabledServer> obtainServersViaDiscovery() {
    List<DiscoveryEnabledServer> serverList = new ArrayList<~>();

    if (eurekaClientProvider == null || eurekaClientProvider.get() == null) {
        logger.warn("EurekaClient has not been initialized yet, returning a
        return new ArrayList<DiscoveryEnabledServer>();
    }

    EurekaClient eurekaClient = eurekaClientProvider.get();
    if (vipAddresses!=null){
        for (String vipAddress : vipAddresses.split( regex: ",")) {
            // if targetRegion is null, it will be interpreted as the same
```

这个方法里面有一个方法是从本地从 eureka 服务端拉取到的服务列表的变量中获取到服务列表

```
private List<DiscoveryEnabledServer> obtainServersViaDiscovery() {
    List<DiscoveryEnabledServer> serverList = new ArrayList<~>();

    if (eurekaClientProvider == null || eurekaClientProvider.get() == null) {
        logger.warn("EurekaClient has not been initialized yet, returning an empty list");
        return new ArrayList<DiscoveryEnabledServer>();
    }

    EurekaClient eurekaClient = eurekaClientProvider.get();
    if (vipAddresses!=null){
        for (String vipAddress : vipAddresses.split( regex: ",")) {
            // if targetRegion is null, it will be interpreted as the same region of client
            List<InstanceInfo> listOfInstanceInfo = eurekaClient.getInstancesByVipAddress(vipAddress, isSecure,
            for (InstanceInfo ii : listOfInstanceInfo) {
```

```
@Override
public List<InstanceInfo> getInstancesByVipAddress(String vipAddress, boolean secure,
                                                   @Nullable String region) {
    if (vipAddress == null) {
        throw new IllegalArgumentException(
                "Supplied VIP Address cannot be null");
    }
    Applications applications;
    if (instanceRegionChecker.isLocalRegion(region)) {
        applications = this.localRegionApps.get();
    } else {
        applications = remoteRegionVsApps.get(region);
        if (null == applications) {
            logger.debug("No applications are defined for region {}, so returning an emp
                    + "address {}.", region, vipAddress);
            return Collections.emptyList();
        }
    }
```



另外一个类

RibbonClientConfiguration

这个类中有一个非常重要的方法，serverList 是从第一个注册的类中实例化的，在这里依赖注入进来了

```
@ConditionalOnMissingBean
public ServerListUpdater ribbonServerListUpdater(IClientConfig config) { return new PollingServerListUpdater(co

@Bean
@ConditionalOnMissingBean
public ILoadBalancer ribbonLoadBalancer(IClientConfig config,
        ServerList<Server> serverList, ServerListFilter<Server> serverListFilter,
        IRule rule, IPing ping, ServerListUpdater serverListUpdater) {
    if (this.propertiesFactory.isSet(ILoadBalancer.class, name)) {
        return this.propertiesFactory.get(ILoadBalancer.class, config, name);
    }
    return new ZoneAwareLoadBalancer<>(config, rule, ping, serverList,
            serverListFilter, serverListUpdater);
}

@Bean
```

返回的实例就是 ILoadBalanced 类型的，所以我们刚刚看到的

```
public ILoadBalancer getLoadBalancer(String name) {
    return getInstance(name, ILoadBalancer.class);
}
```

获取到的实例就是 ZoneAwareLoadBalancer 实例。在该类实例化的构造函数中，最终会调到其父类的构造函数，里面有一个方法

```
                filter,
                new PollingServerListUpdater()
        );
    }

    public DynamicServerListLoadBalancer(IClientConfig clientConfig, IRule rule, IPing ping,
                                         ServerList<T> serverList, ServerListFilter<T> filter,
                                         ServerListUpdater serverListUpdater) {
        super(clientConfig, rule, ping);
        this.serverListImpl = serverList;
        this.filter = filter;
        this.serverListUpdater = serverListUpdater;
        if (filter instanceof AbstractServerListFilter) {
            ((AbstractServerListFilter) filter).setLoadBalancerStats(getLoadBalancerStats());
        }
        restOfInit(clientConfig);
    }
```

```
void restOfInit(IClientConfig clientConfig) {
    boolean primeConnection = this.isEnablePrimingConnections();
    // turn this off to avoid duplicated asynchronous priming done in BaseLoadBalanc
    this.setEnablePrimingConnections(false);
    enableAndInitLearnNewServersFeature();

    updateListOfServers();
    if (primeConnection && this.getPrimeConnections() != null) {
        this.getPrimeConnections()
                .primeConnections(getReachableServers());
    }
    this.setEnablePrimingConnections(primeConnection);
    LOGGER.info("DynamicServerListLoadBalancer for client {} initialized: {}", clien
}

@VisibleForTesting
public void updateListOfServers() {
    List<T> servers = new ArrayList<T>();
    if (serverListImpl != null) {
        servers = serverListImpl.getUpdatedListOfServers();
        LOGGER.debug("List of Servers for {} obtained from Discovery client: {}",
                getIdentifier(), servers);

        if (filter != null) {
            servers = filter.getFilteredListOfServers(servers);
            LOGGER.debug("Filtered List of Servers for {} obtained from Discovery client: {}"
                    getIdentifier(), servers);
        }
    }
    updateAllServerList(servers);
}
```

```
@Override
public List<DiscoveryEnabledServer> getUpdatedListOfServers(){
    return obtainServersViaDiscovery();
}
```

OK 这里就掉到了这个核心方法，从本地服务列表中获取到了服务列表信息

在看看获取到这个实例化的接下来的操作

```
Server server = getServer(loadBalancer, hint);
```

这里就是根据前面从本地服务列表中获取到的列表信息，根据负载均衡算法从中获取到一个

```
public Server chooseServer(Object key) {
    if (counter == null) {
        counter = createCounter();
    }
    counter.increment();
    if (rule == null) {
        return null;
    } else {
        try {
            return rule.choose(key);
        } catch (Exception e) {
            logger.warn("LoadBalancer [{}]:  Error choosing server for key {}", name, key, e);
            return null;
        }
    }
}
```

接下来就是具体 http 调用了，又会走回去，进到没有拦截器的实例中

```
public <T> T execute(String serviceId, LoadBalancerRequest<T> request, Object hint)
        throws IOException {
    ILoadBalancer loadBalancer = getLoadBalancer(serviceId);
    Server server = getServer(loadBalancer, hint);
    if (server == null) {
        throw new IllegalStateException("No instances available for " + serviceId);
    }
    RibbonServer ribbonServer = new RibbonServer(serviceId, server,
            isSecure(server, serviceId),
            serverIntrospector(serviceId).getMetadata(server));

    return execute(serviceId, ribbonServer, request);
}
```

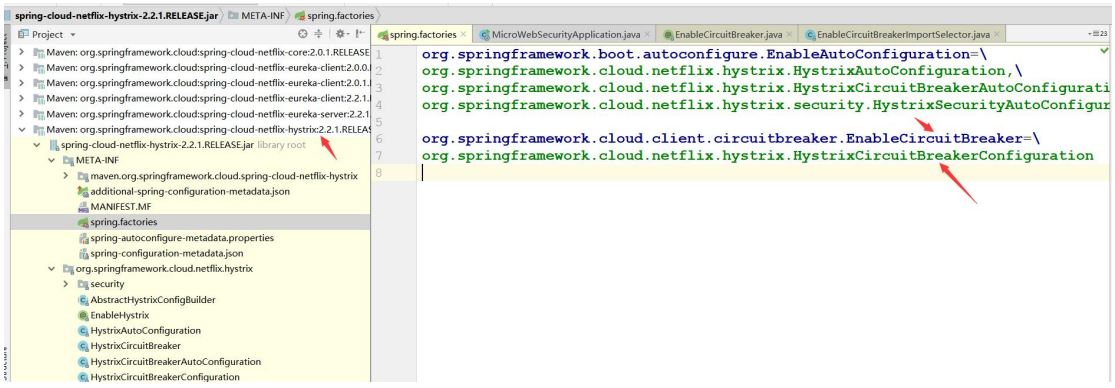
```
@Override
public <T> T execute(String serviceId, ServiceInstance serviceInstance,
        LoadBalancerRequest<T> request) throws IOException {
    Server server = null;
    if (serviceInstance instanceof RibbonServer) {
        server = ((RibbonServer) serviceInstance).getServer();
    }
    if (server == null) {
        throw new IllegalStateException("No instances available for " + serviceId);
    }

    RibbonLoadBalancerContext context = this.clientFactory
            .getLoadBalancerContext(serviceId);
    RibbonStatsRecorder statsRecorder = new RibbonStatsRecorder(context, server);

    try {
        T returnVal = request.apply(serviceInstance);
        statsRecorder.recordStats(returnVal);
        return returnVal;
    }
    // catch IOException and rethrow so RestTemplate behaves correctly
```

32、Hystrix 源码

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import(EnableCircuitBreakerImportSelector.class)
public @interface EnableCircuitBreaker {

}
```

```java
        if (!isEnabled()) {
            return new String[0];
        }
        AnnotationAttributes attributes = AnnotationAttributes.fromMap(
                metadata.getAnnotationAttributes(this.annotationClass.getName(), classValuesAsString: true));

        Assert.notNull(attributes, message: "No " + getSimpleName() + " attributes found. Is "
                + metadata.getClassName() + " annotated with @" + getSimpleName() + "?");

        // Find all possible auto configuration classes, filtering duplicates
        List<String> factories = new ArrayList<>(new LinkedHashSet<>(SpringFactoriesLoader
                .loadFactoryNames(this.annotationClass, this.beanClassLoader)));

        if (factories.isEmpty() && !hasDefaultFactory()) {
            throw new IllegalStateException("Annotation @" + getSimpleName()
                    + " found, but there are no implementations. Did you forget to include a starter?");
        }
```

SPI 加载 EnableCircuitBreaker 类型的类



创建 hystrixCommand 注解的切面类

```java
import ...

/**
 * @author Spencer Gibb
 * @author Christian Dupuis
 * @author Venil Noronha
 */
@Configuration(proxyBeanMethods = false)
public class HystrixCircuitBreakerConfiguration {

    @Bean
    public HystrixCommandAspect hystrixCommandAspect() { return new HystrixCommandAspect(); }

    @Bean
    public HystrixShutdownHook hystrixShutdownHook() { return new HystrixShutdownHook(); }
```

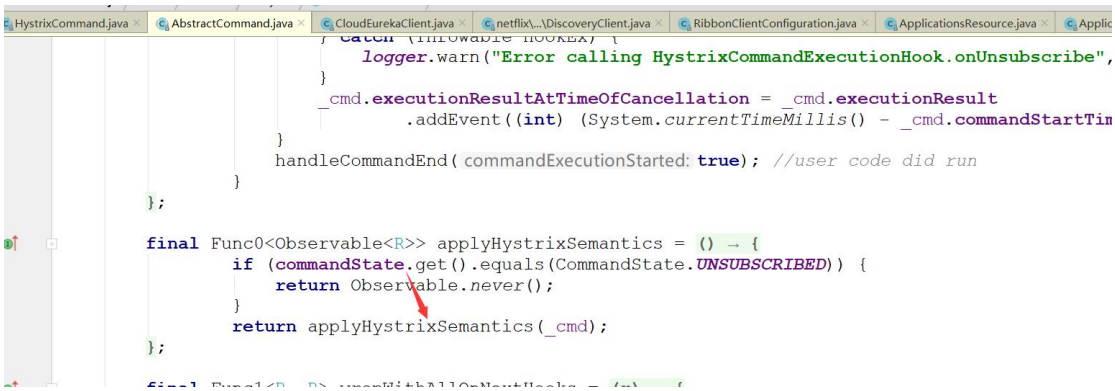
```
@Around("hystrixCommandAnnotationPointcut() || hystrixCollapserAnnotationPointcut()")
public Object methodsAnnotatedWithHystrixCommand(final ProceedingJoinPoint joinPoint) throws Throwabl
    Method method = getMethodFromTarget(joinPoint);
    Validate.notNull(method, message: "failed to get method from joinPoint: %s", joinPoint);
    if (method.isAnnotationPresent(HystrixCommand.class) && method.isAnnotationPresent(HystrixCollaps
        throw new IllegalStateException("method cannot be annotated with HystrixCommand and HystrixCo
                "annotations at the same time");
    }
    MetaHolderFactory metaHolderFactory = META_HOLDER_FACTORY_MAP.get(HystrixPointcutType.of(method))
    MetaHolder metaHolder = metaHolderFactory.create(joinPoint);
    HystrixInvokable invokable = HystrixCommandFactory.getInstance().create(metaHolder);
    ExecutionType executionType = metaHolder.isCollapserAnnotationPresent() ?
            metaHolder.getCollapserExecutionType() : metaHolder.getExecutionType();

    Object result;
    try {
        if (!metaHolder.isObservable()) {
            result = CommandExecutor.execute(invokable, executionType, metaHolder);
        } else {
            result = executeObservable(invokable, executionType, metaHolder);
```

同步调用

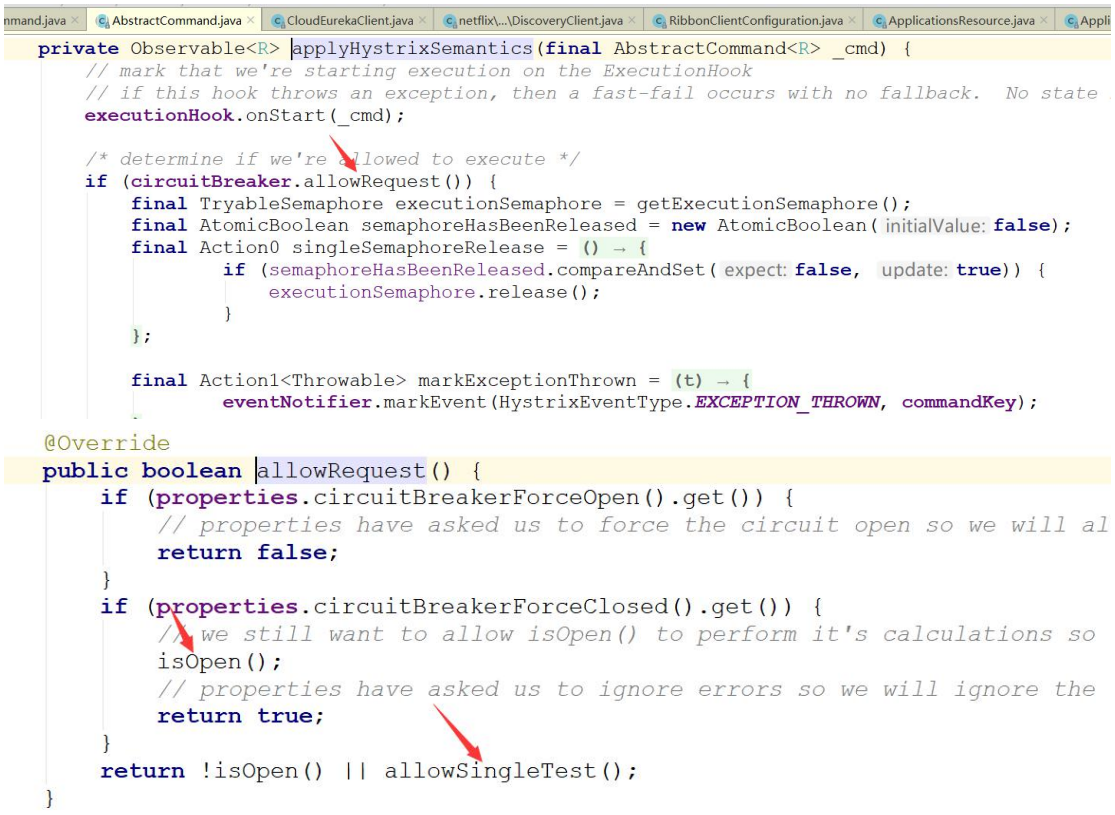
```
CommandExecutor.java | CloudEurekaClient.java | netflix\...\DiscoveryClient.java | RibbonClientConfiguration.java | ApplicationsResource.java | ApplicationResource.java | InstanceResource.java
46 @    public static Object execute(HystrixInvokable invokable, ExecutionType executionType, MetaHolder metaHolder) t
47          Validate.notNull(invokable);
48          Validate.notNull(metaHolder);
49
50          switch (executionType) {
51              case SYNCHRONOUS: {
52                  return castToExecutable(invokable, executionType).execute();
53              }
54              case ASYNCHRONOUS: {
55                  HystrixExecutable executable = castToExecutable(invokable, executionType);
56                  if (metaHolder.hasFallbackMethodCommand()
57                          && ExecutionType.ASYNCHRONOUS == metaHolder.getFallbackExecutionType()) {
58                      return new FutureDecorator(executable.queue());
59                  }
60                  return executable.queue();
61              }
62              case OBSERVABLE: {
63                  HystrixObservable observable = castToObservable(invokable);
64                  return ObservableExecutionMode.EAGER == metaHolder.getObservableExecutionMode() ? observable.obser
65              }
66              default:
67                  throw new RuntimeException("unsupported execution type: " + executionType);
68          }
```

```
    public R execute() {
        try {
            return queue().get();
        } catch (Exception e) {
            throw Exceptions.sneakyThrow(decomposeException(e));
        }
    }
```

```
.java | HystrixCommand.java | CloudEurekaClient.java | netflix\...\DiscoveryClient.java | RibbonClientConfiguration.java | ApplicationsResource.j
     * @throws IllegalStateException
     *             if invoked more than once
     */
    public Future<R> queue() {
        /*
         * The Future returned by Observable.toBlocking().toFuture() does not implement
         * interruption of the execution thread when the "mayInterrupt" flag of Future.
         * thus, to comply with the contract of Future, we must wrap around it.
         */
        final Future<R> delegate = toObservable().toBlocking().toFuture();

        final Future<R> f = new Future<R>() {
```



熔断器判断是否允许请求



判断请求数和失败率是否达标，如果都达标就开启熔断器

```
ommand.java | HystrixCircuitBreaker.java | CloudEurekaClient.java | netflix\...\DiscoveryClient.java | RibbonClientConfiguration.java | ApplicationsResource.java | ApplicationResource.ja
    public boolean isOpen() {
        if (circuitOpen.get()) {
            // if we're open we immediately return true and don't bother attempting to 'close' ourself as t
            return true;
        }

        // we're closed, so let's see if errors have made us so we should trip the circuit open
        HealthCounts health = metrics.getHealthCounts();

        // check if we are past the statisticalWindowVolumeThreshold
        if (health.getTotalRequests() < properties.circuitBreakerRequestVolumeThreshold().get()) {
            // we are not past the minimum volume threshold for the statisticalWindow so we'll return false
            return false;
        }

        if (health.getErrorPercentage() < properties.circuitBreakerErrorThresholdPercentage().get()) {
            return false;
        } else {
            // our failure rate is too high, trip the circuit
            if (circuitOpen.compareAndSet( expect: false,  update: true)) {
                // if the previousValue was false then we want to set the currentTime
                circuitOpenedOrLastTestedTime.set(System.currentTimeMillis());
                return true;
            } else {
                // How could previousValue be true? If another thread was going through this code at the sa
                // caused another thread to set it to true already even though we were in the process of do
```

当当前时间超过了时间窗口则允许一次请求

```
public boolean allowSingleTest() {
    long timeCircuitOpenedOrWasLastTested = circuitOpenedOrLastTestedTime.get();
    // 1) if the circuit is open
    // 2) and it's been longer than 'sleepWindow' since we opened the circuit
    if (circuitOpen.get() && System.currentTimeMillis() > timeCircuitOpenedOrWasLastTested + properties.circ
        // We push the 'circuitOpenedTime' ahead by 'sleepWindow' since we have allowed one request to try.
        // If it succeeds the circuit will be closed, otherwise another singleTest will be allowed at the en
        if (circuitOpenedOrLastTestedTime.compareAndSet(timeCircuitOpenedOrWasLastTested, System.currentTime
            // if this returns true that means we set the time so we'll return true to allow the singleTest
            // if it returned false it means another thread raced us and allowed the singleTest before we di
            return true;
        }
    }
    return false;
}
```

如果是采用的信号量隔离级别

```
if (executionSemaphore.tryAcquire()) {
    try {
        /* used to track userThreadExecutionTime */
        executionResult = executionResult.setInvocationStartTime(System.currentTimeMillis());
        return executeCommandAndObserve(_cmd)
                .doOnError(markExceptionThrown)
                .doOnTerminate(singleSemaphoreRelease)
                .doOnUnsubscribe(singleSemaphoreRelease);
    } catch (RuntimeException e) {
        return Observable.error(e);
    }
} else {
    return handleSemaphoreRejectionViaFallback();
}
```

如果全局变量 count 大于配置的最大请求数，则返回 false，不让请求走降级

```java
/* package */static class TryableSemaphoreActual implements TryableSer
    protected final HystrixProperty<Integer> numberOfPermits;
    private final AtomicInteger count = new AtomicInteger( initialValue: 0)

    public TryableSemaphoreActual(HystrixProperty<Integer> numberOfPe
        this.numberOfPermits = numberOfPermits;
    }

    @Override
    public boolean tryAcquire() {
        int currentCount = count.incrementAndGet();
        if (currentCount > numberOfPermits.get()) {
            count.decrementAndGet();
            return false;
        } else {
            return true;
        }
    }
```

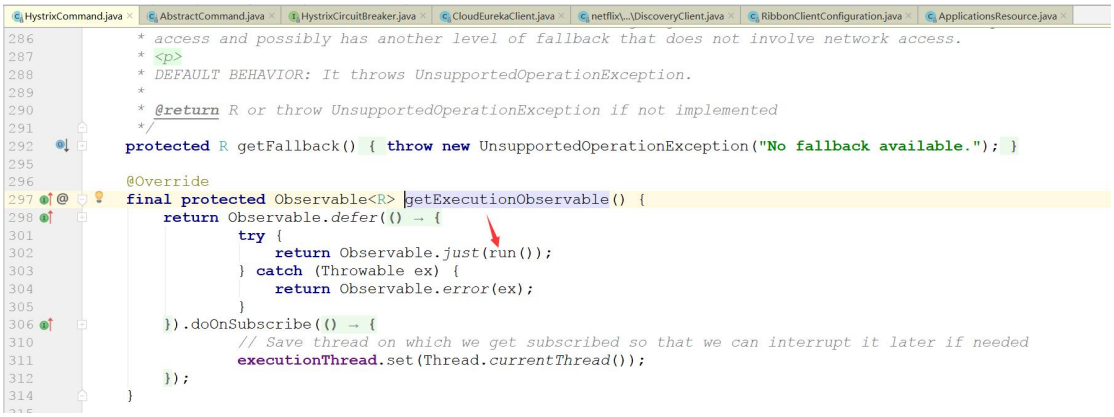
如果是线程池隔离级别，则这个 tryAcquire 方法就会返回 true

```java
                return handleFailureViaFallback(e);
            }
        };

        final Action1<Notification<? super R>> setRequestContext = (rNotification) -> {
                setRequestContextIfNeeded(currentRequestContext);
        };

        Observable<R> execution;
        if (properties.executionTimeoutEnabled().get()) {
            execution = executeCommandWithSpecifiedIsolation(_cmd)
                    .lift(new HystrixObservableTimeoutOperator<R>(_cmd));
        } else {
            execution = executeCommandWithSpecifiedIsolation(_cmd);
        }

        return execution.doOnNext(markEmits)
                .doOnCompleted(markOnCompleted)
                .onErrorResumeNext(handleFallback)
                .doOnEach(setRequestContext);
    }
```

```java
            }

            metrics.markCommandStart(commandKey, threadPoolKey, ExecutionIsolationStrategy.THREAD);

            if (isCommandTimedOut.get() == TimedOutStatus.TIMED_OUT) {
                // the command timed out in the wrapping thread so we will return immediately
                // and not increment any of the counters below or other such logic
                return Observable.error(new RuntimeException("timed out before executing run()"));
            }
            if (threadState.compareAndSet(ThreadState.NOT_USING_THREAD, ThreadState.STARTED)) {
                //we have not been unsubscribed, so should proceed
                HystrixCounters.incrementGlobalConcurrentThreads();
                threadPool.markThreadExecution();
                // store the command that is being run
                endCurrentThreadExecutingCommand = Hystrix.startCurrentThreadExecutingCommand(getCommandKe
                executionResult = executionResult.setExecutedInThread();
                /**
                 * If any of these hooks throw an exception, then it appears as if the actual execution th
                 */
                try {
                    executionHook.onThreadStart(_cmd);
                    executionHook.onRunStart(_cmd);
                    executionHook.onExecutionStart(_cmd);
                    return getUserExecutionObservable(_cmd);
                } catch (Throwable ex) {
```

```
private Observable<R> getUserExecutionObservable(final AbstractCommand<R> _cmd) {
    Observable<R> userObservable;

    try {
        userObservable = getExecutionObservable();
    } catch (Throwable ex) {
        // the run() method is a user provided implementation so can throw instead of using
        // so we catch it here and turn it into Observable.error
        userObservable = Observable.error(ex);
    }

    return userObservable
            .lift(new ExecutionHookApplication(_cmd))
            .lift(new DeprecatedOnRunHookApplication(_cmd));
}
```

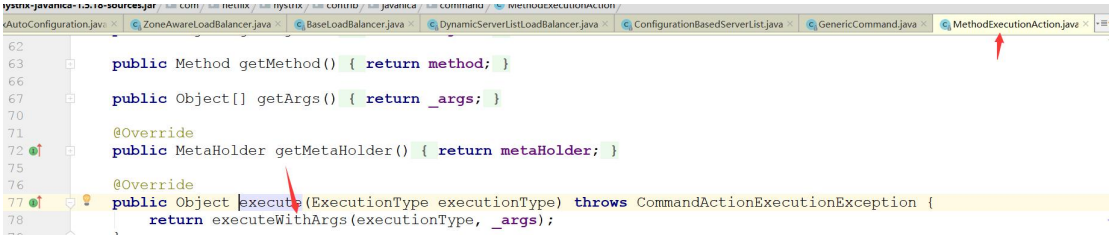
这里一切都合格，则 hystrix 就判断可以调用后端服务接口，则会反射调用被代理方法



这个钩子返回会掉到

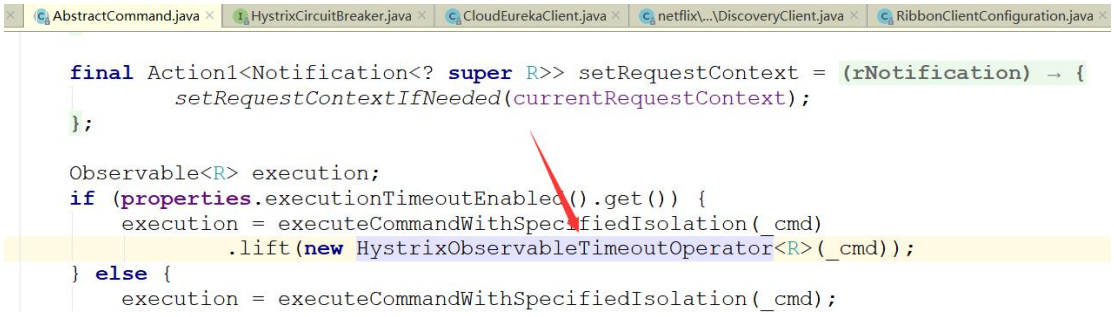
```
@ThreadSafe
public class GenericCommand extends AbstractHystrixCommand<Object> {

    private static final Logger LOGGER = LoggerFactory.getLogger(GenericCommand.class);

    public GenericCommand(HystrixCommandBuilder builder) { super(builder); }

    /**
     * {@inheritDoc}
     */
    @Override
    protected Object run() throws Exception {
        LOGGER.debug("execute command: {}", getCommandKey().name());
        return process(new Action() {
            @Override
            Object execute() { return getCommandAction().execute(getExecutionType()); }
        });
    }

    /**
```

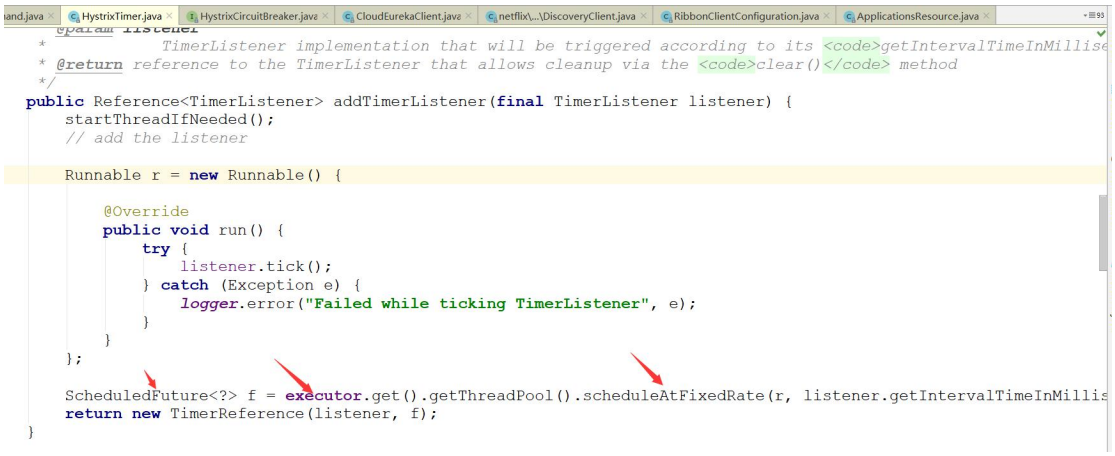
接下来就是反射调用了



```java
private Object execute(Object o, Method m, Object... args) throws CommandActionExecutionException {
    Object result = null;
    try {
        m.setAccessible(true); // suppress Java language access
        if (isCompileWeaving() && metaHolder.getAjcMethod() != null) {
            result = invokeAjcMethod(metaHolder.getAjcMethod(), o, metaHolder, args);
        } else {
            result = m.invoke(o, args);
        }
    } catch (IllegalAccessException e) {
        propagateCause(e);
    } catch (InvocationTargetException e) {
        propagateCause(e);
    }
    return result;
}
```

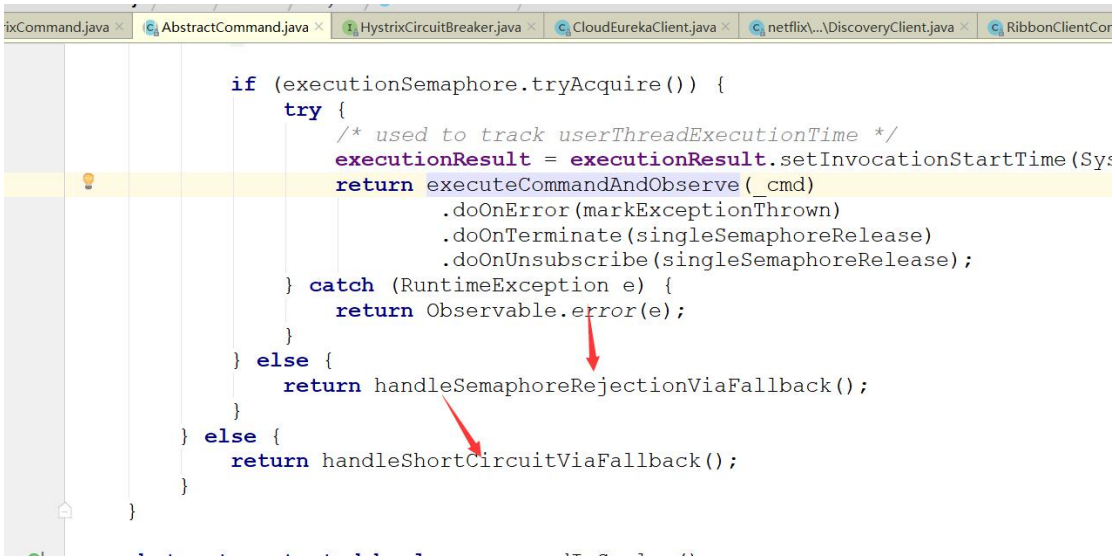
线程池的创建及超时控制





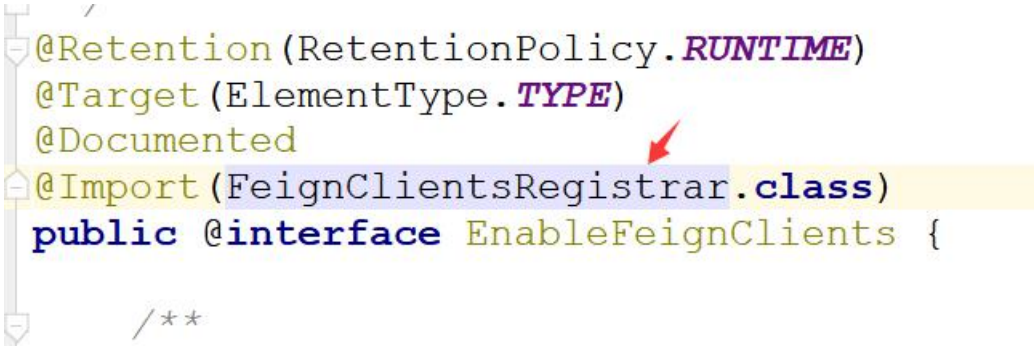
这里创建了线程

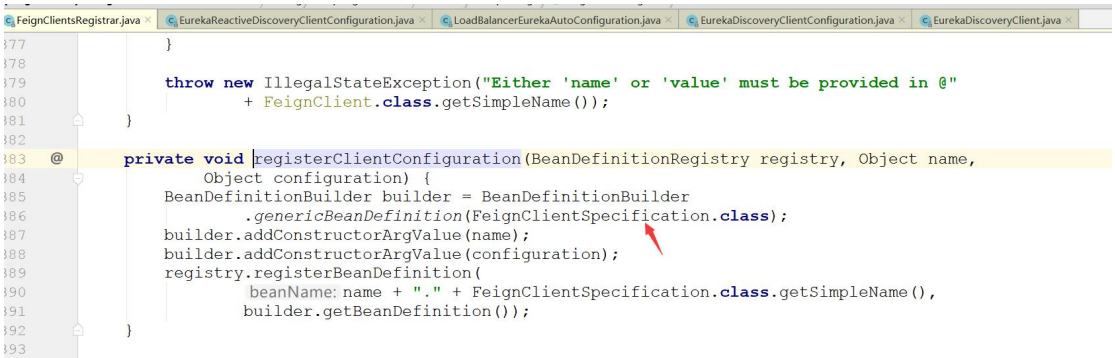
其他的，比如熔断器开启，线程池，信号量都满了，则会走到降级方法



这里也是会反射调用到 fallback 方法，fallback 降级方法也是有信号量和线程池的大小控制的，也就是信号量或线程池是多少大小，fallback 降级方法也会接收多少降级的请求。

33、Feign 源码





FeignClientsRegistrar.java | DefaultListableBeanFactory.java | EurekaReactiveDiscoveryClientConfiguration.java | LoadBalancerEurekaAutoConfiguration.java | EurekaDiscoveryClientConfiguration.j

```
225  @      private void registerFeignClient(BeanDefinitionRegistry registry,
226                  AnnotationMetadata annotationMetadata, Map<String, Object> attributes) {
227              String className = annotationMetadata.getClassName();
228              BeanDefinitionBuilder definition = BeanDefinitionBuilder
229                      .genericBeanDefinition(FeignClientFactoryBean.class);
230              validate(attributes);
231              definition.addPropertyValue( name: "url", getUrl(attributes));
232              definition.addPropertyValue( name: "path", getPath(attributes));
233              String name = getName(attributes);
234              definition.addPropertyValue( name: "name", name);
235              String contextId = getContextId(attributes);
236              definition.addPropertyValue( name: "contextId", contextId);
237              definition.addPropertyValue( name: "type", className);
238              definition.addPropertyValue( name: "decode404", attributes.get("decode404"));
239              definition.addPropertyValue( name: "fallback", attributes.get("fallback"));
240              definition.addPropertyValue( name: "fallbackFactory", attributes.get("fallbackFactory"));
241              definition.setAutowireMode(AbstractBeanDefinition.AUTOWIRE_BY_TYPE);
242
243              String alias = contextId + "FeignClient";
244              AbstractBeanDefinition beanDefinition = definition.getBeanDefinition();
245
246              boolean primary = (Boolean) attributes.get("primary"); // has a default, won't be
247                                                                     // null
248
249              beanDefinition.setPrimary(primary);
```

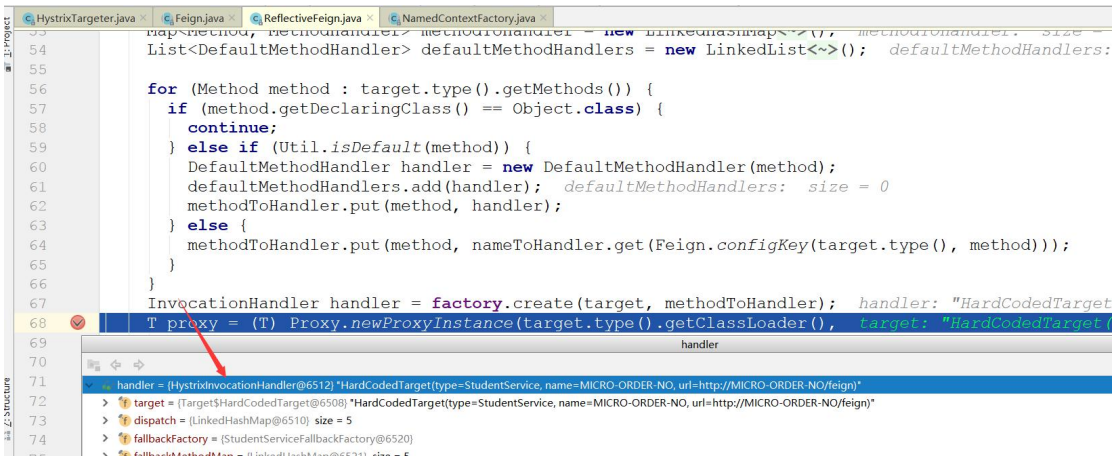
跟 mybatis 差不多，在 FeignClientFactoryBean 对有@FeignClient 注解的接口生成接口的代理 FeignClientFactoryBean 实现了 factoryBean 接口，里面有 getObject 方法

```
@Override
public Object getObject() throws Exception {
    return getTarget();
}
```

feign-core-10.4.0-sources.jar | feign | ReflectiveFeign

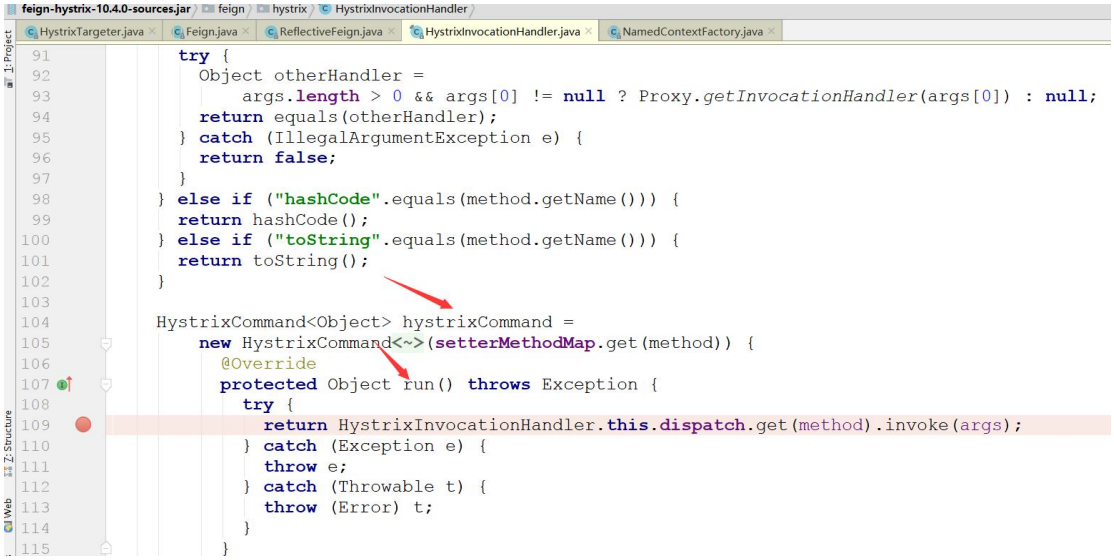
HystrixTargeter.java | Feign.java | ReflectiveFeign.java

```
51  @  public <T> T newInstance(Target<T> target) {
52       Map<String, MethodHandler> nameToHandler = targetToHandlersByName.apply(target);
53       Map<Method, MethodHandler> methodToHandler = new LinkedHashMap<~>();
54       List<DefaultMethodHandler> defaultMethodHandlers = new LinkedList<~>();
55
56       for (Method method : target.type().getMethods()) {
57         if (method.getDeclaringClass() == Object.class) {
58           continue;
59         } else if (Util.isDefault(method)) {
60           DefaultMethodHandler handler = new DefaultMethodHandler(method);
61           defaultMethodHandlers.add(handler);
62           methodToHandler.put(method, handler);
63         } else {
64           methodToHandler.put(method, nameToHandler.get(Feign.configKey(target.type(), method)));
65         }
66       }
67       InvocationHandler handler = factory.create(target, methodToHandler);
68       T proxy = (T) Proxy.newProxyInstance(target.type().getClassLoader(),
69           new Class<?>[] {target.type()}, handler);
70
71       for (DefaultMethodHandler defaultMethodHandler : defaultMethodHandlers) {
72         defaultMethodHandler.bindTo(proxy);
73       }
74       return proxy;
75     }
```

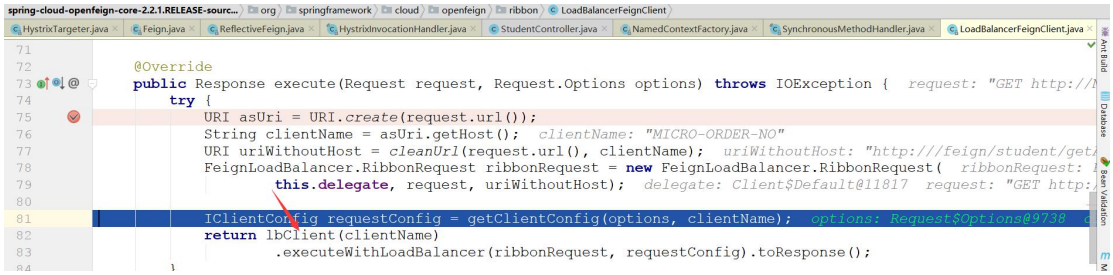


最终会生成有@FeignClient 注解的接口的代理

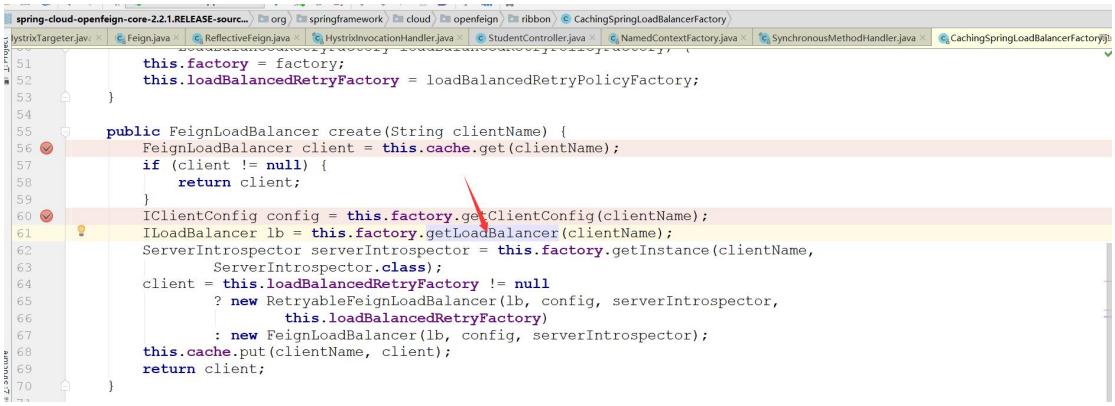
所以 Controller 中获取到代理实例后，则会掉到 HystrixInvocationHandler 的 invoke 方法

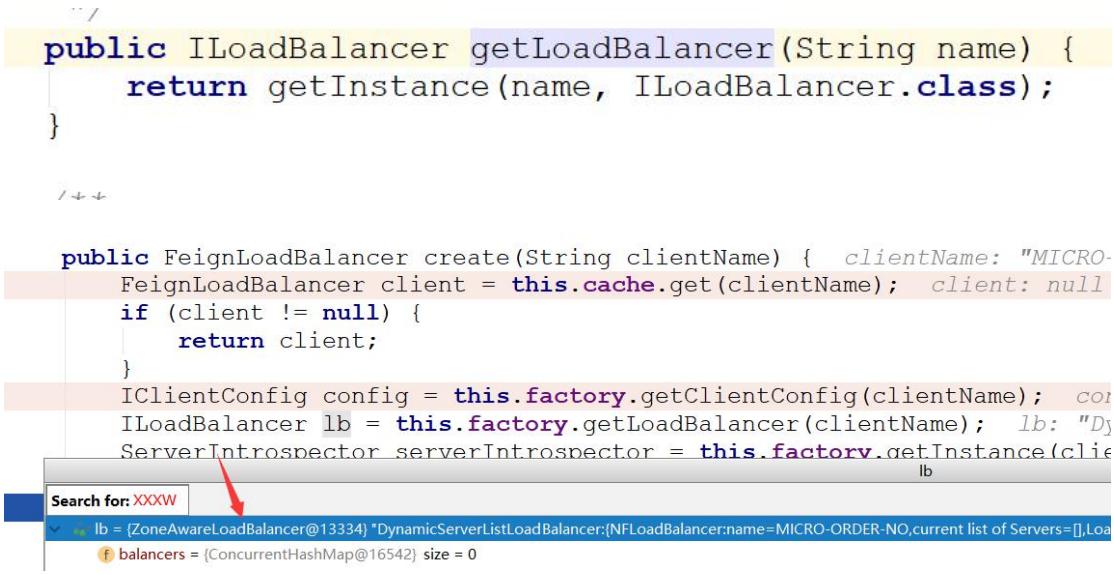


还是走的 hystrix 那一套，最后会走到上面的 run 钩子方法中



在这个方法里面获取服务列表类进行调用

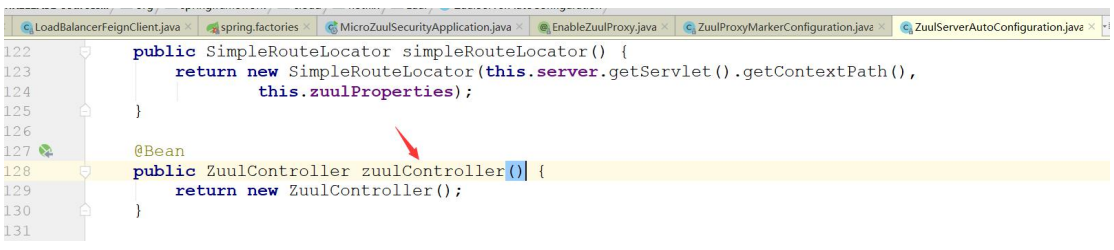




这里熟不熟属性了，跟上面 hystrix 的分析是一样的

34、Zuul 源码

对 zuul 工程的请求是有一个 controller 来接收的



```java
public class ZuulController extends ServletWrappingController {

    public ZuulController() {
        setServletClass(ZuulServlet.class);
        setServletName("zuul");
        setSupportedMethods((String[]) null); // Allow all
    }

    @Override
    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        try {
            // We don't care about the other features of the base class, just want to
            // handle the request
            return super.handleRequestInternal(request, response);
        }
        finally {
            // @see com.netflix.zuul.context.ContextLifecycleFilter.doFilter
            RequestContext.getCurrentContext().unset();
        }
    }
```

所有请求都会走到这个方法

对应的 HandlerMapping

```java
@Bean
public ZuulHandlerMapping zuulHandlerMapping(RouteLocator routes,
        ZuulController zuulController) {
    ZuulHandlerMapping mapping = new ZuulHandlerMapping(routes, zuulController);
    mapping.setErrorController(this.errorController);
    mapping.setCorsConfigurations(getCorsConfigurations());
    return mapping;
}
```

所有的请求会有这个 zuulServlet 来处理

```java
@Override
protected ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletResponse respo
        throws Exception {

    Assert.state( expression: this.servletInstance != null,  message: "No Servlet instance");
    this.servletInstance.service(request, response);
    return null;
}
```

```java
@Bean
@ConditionalOnMissingBean(name = "zuulServlet")
@ConditionalOnProperty(name = "zuul.use-filter", havingValue = "false",
        matchIfMissing = true)
public ServletRegistrationBean zuulServlet() {
    ServletRegistrationBean<ZuulServlet> servlet = new ServletRegistrationBean<>(
            new ZuulServlet(), this.zuulProperties.getServletPattern());
    // The whole point of exposing this servlet is to provide a route that doesn't
    // buffer requests.
    servlet.addInitParameter( name: "buffer-requests",  value: "false");
    return servlet;
}
```

在 zuulServlet 中就涉及到 pre、route、post、error 过滤器的调用

com › netflix › zuul › http › ZuulServlet

FeignClient.jav × | ZuulHandlerMapping.java × | ZuulController.java × | ServletWrappingController.java × | ZuulServlet.class × | ZuulServlet.java ×

```
68          // Marks this request as having passed through the "Zuul engine", as opposed t
69          // explicitly bound in web.xml, for which requests will not have the same data
70          RequestContext context = RequestContext.getCurrentContext();
71          context.setZuulEngineRan();
72
73          try {
74              preRoute();
75          } catch (ZuulException e) {
76              error(e);
77              postRoute();
78              return;
79          }
80          try {
81              route();
82          } catch (ZuulException e) {
83              error(e);
84              postRoute();
85              return;
86          }
87          try {
88              postRoute();
89          } catch (ZuulException e) {
90              error(e);
```

其中分析一个 route 过滤器，这个过滤器是 zuul 做路由的调用的

ing-cloud-netflix-zuul-2.2.1.RELEASE-sources... › org › springframework › cloud › netflix › zuul › ZuulProxyAutoConfiguration

ZuulHandlerMapping.java × | ZuulController.java × | ServletWrappingController.java × | ZuulServlet.class × | ZuulServlet.java × | spring.factories × | ZuulProxyAutoConfiguration.java ×

```
0           return new PreDecorationFilter(routeLocator,
1                   this.server.getServlet().getContextPath(), this.zuulProperties,
2                   proxyRequestHelper);
3       }
4
5       // route filters
6       @Bean
7       @ConditionalOnMissingBean(RibbonRoutingFilter.class)
8       public RibbonRoutingFilter ribbonRoutingFilter(ProxyRequestHelper helper,
9               RibbonCommandFactory<?> ribbonCommandFactory) {
0           RibbonRoutingFilter filter = new RibbonRoutingFilter(helper, ribbonCommandFactory,
1                   this.requestCustomizers);
2           return filter;
3       }
4
```

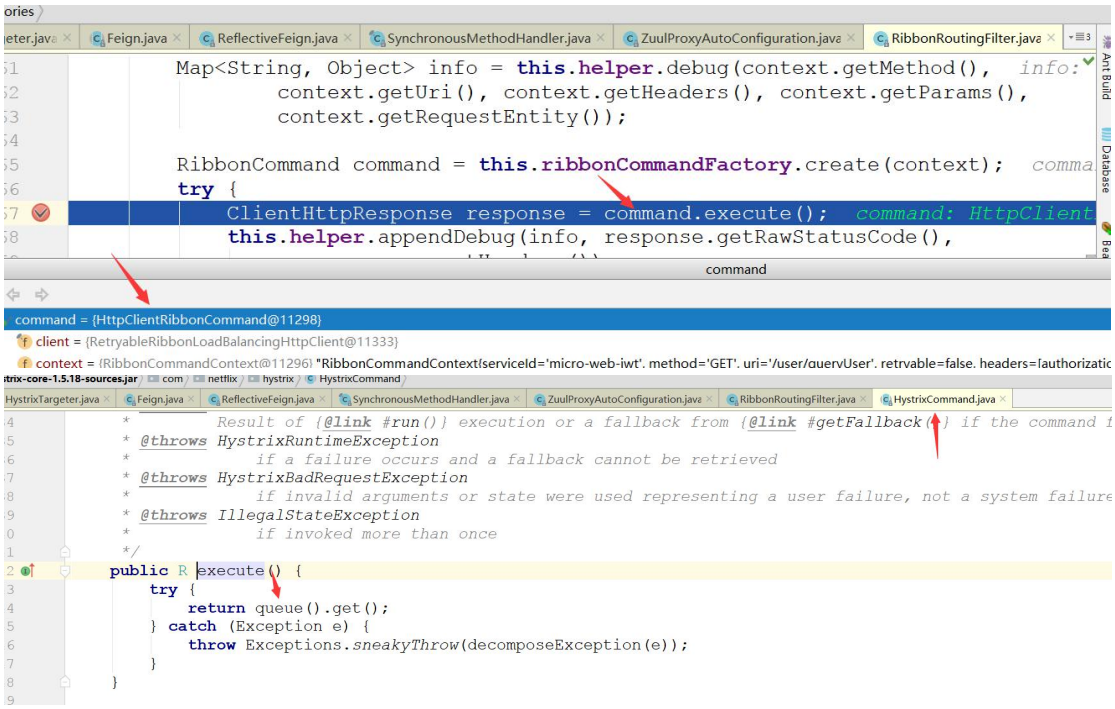
cloud-netflix-zuul-2.2.1.RELEASE-sources... › org › springframework › cloud › netflix › zuul › filters › route › RibbonRoutingFilter

tiveFeign.jav × | SynchronousMethodHandler.java × | ZuulServlet.class × | ZuulServlet.java × | spring.factories × | ZuulProxyAutoConfiguration.java × | RibbonRoutingFilter.java ×

```
    @Override
    public Object run() {
        RequestContext context = RequestContext.getCurrentContext();
        this.helper.addIgnoredHeaders();
        try {
            RibbonCommandContext commandContext = buildCommandContext(context);
            ClientHttpResponse response = forward(commandContext);
            setResponse(response);
            return response;
        }
        catch (ZuulException ex) {
            throw new ZuulRuntimeException(ex);
        }
        catch (Exception ex) {
            throw new ZuulRuntimeException(ex);
        }
    }
```

Feign.java × | ReflectiveFeign.java × | SynchronousMethodHandler.java × | ZuulServlet.class × | ZuulServlet.java × | spring.factories × | ZuulProxyAutoConfiguration.java × | RibbonRoutingFilter.java ×

```
    }

    protected ClientHttpResponse forward(RibbonCommandContext context) throws Exception {
        Map<String, Object> info = this.helper.debug(context.getMethod(),
                context.getUri(), context.getHeaders(), context.getParams(),
                context.getRequestEntity());

        RibbonCommand command = this.ribbonCommandFactory.create(context);
        try {
            ClientHttpResponse response = command.execute();
            this.helper.appendDebug(info, response.getRawStatusCode(),
                    response.getHeaders());
            return response;
        }
        catch (HystrixRuntimeException ex) {
            return handleException(info, ex);
        }

    }
```



这里走的依然是 hystrix 那一套

35、Config 源码

Config 服务端

接收获取配置信息请求的 Controller

```
@Bean
@RefreshScope
public EnvironmentController environmentController(
        EnvironmentRepository envRepository, ConfigServerProperties server) {
    EnvironmentController controller = new EnvironmentController(
            encrypted(envRepository, server), this.objectMapper);
    controller.setStripDocumentFromYaml(server.isStripDocumentFromYaml());
    controller.setAcceptEmpty(server.isAcceptEmpty());
    return controller;
}
```

以这个为例

```
@RequestMapping("/{name}-{profiles}.properties")
public ResponseEntity<String> properties(@PathVariable String name,
        @PathVariable String profiles,
        @RequestParam(defaultValue = "true") boolean resolvePlaceholders)
        throws IOException {
    return labelledProperties(name, profiles, label: null, resolvePlaceholders);
}
```

```
public Environment getEnvironment(String name, String profiles, String label,
        boolean includeOrigin) {
    if (name != null && name.contains("(_)")) {
        // "(_)" is uncommon in a git repo name, but "/" cannot be matched
        // by Spring MVC
        name = name.replace( target: "(_)", replacement: "/");
    }
    if (label != null && label.contains("(_)")) {
        // "(_)" is uncommon in a git branch name, but "/" cannot be matched
        // by Spring MVC
        label = label.replace( target: "(_)", replacement: "/");
    }
    Environment environment = this.repository.findOne(name, profiles, label,
            includeOrigin);
    if (!this.acceptEmpty
            && (environment == null || environment.getPropertySources().isEmpty()
        throw new EnvironmentNotFoundException("Profile Not found");
    }
    return environment;
}
```

在这里就从 github 上 clone，和更新配置信息并且保存到了本地

```
@Override
public synchronized Environment findOne(String application, String profile,
        String label, boolean includeOrigin) {
    NativeEnvironmentRepository delegate = new NativeEnvironmentRepository(
            getEnvironment(), new NativeEnvironmentProperties());
    Locations locations = getLocations(application, profile, label);
    delegate.setSearchLocations(locations.getLocations());
    Environment result = delegate.findOne(application, profile, label: "", includeOrigin);
    result.setVersion(locations.getVersion());
    result.setLabel(label);
    return this.cleaner.clean(result, getWorkingDirectory().toURI().toString(),
            getUri());
}
```

```
@Override
public synchronized Locations getLocations(String application, String profile,
        String label) {
    if (label == null) {
        label = this.defaultLabel;
    }
    String version = refresh(label);
    return new Locations(application, profile, label, version,
            getSearchLocations(getWorkingDirectory(), application, profile, label));
}
```

```
public String refresh(String label) {
    Git git = null;
    try {
        git = createGitClient();
        if (shouldPull(git)) {
            FetchResult fetchStatus = fetch(git, label);
            if (this.deleteUntrackedBranches && fetchStatus != null) {
                deleteUntrackedLocalBranches(fetchStatus.getTrackingRefUpdate
                        git);
            }
            // checkout after fetch so we can get any new branches, tags, ect
            checkout(git, label);
            tryMerge(git, label);
        }
        else {
            // nothing to update so just checkout and merge.
            // Merge because remote branch could have been updated before
            checkout(git, label);
            tryMerge(git, label);
        }
```

```
private Git createGitClient() throws IOException, GitAPIException {
    File lock = new File(getWorkingDirectory(), child: ".git/index.lock");
    if (lock.exists()) {
        // The only way this can happen is if another JVM (e.g. one that
        // crashed earlier) created the lock. We can attempt to recover by
        // wiping the slate clean.
        this.logger.info( o: "Deleting stale JGit lock file at " + lock);
        lock.delete();
    }
    if (new File(getWorkingDirectory(), child: ".git").exists()) {
        return openGitRepository();
    }
    else {
        return copyRepository();
    }
}
```

```
// request).
private synchronized Git copyRepository() throws IOException, GitAPIException {
    deleteBaseDirIfExists();
    getBasedir().mkdirs();
    Assert.state(getBasedir().exists(), message: "Could not create basedir: " + getBasedir());
    if (getUri().startsWith(FILE_URI_PREFIX)) {
        return copyFromLocalRepository();
    }
    else {
        return cloneToBasedir();
    }
}
```

```
private Git cloneToBasedir() throws GitAPIException {
    CloneCommand clone = this.gitFactory.getCloneCommandByCloneRepository()
            .setURI(getUri()).setDirectory(getBasedir());
    configureCommand(clone);
    try {
        return clone.call();
    }
    catch (GitAPIException e) {
        this.logger.warn( o: "Error occured cloning to base directory.", e);
        deleteBaseDirIfExists();
        throw e;
    }
}
```

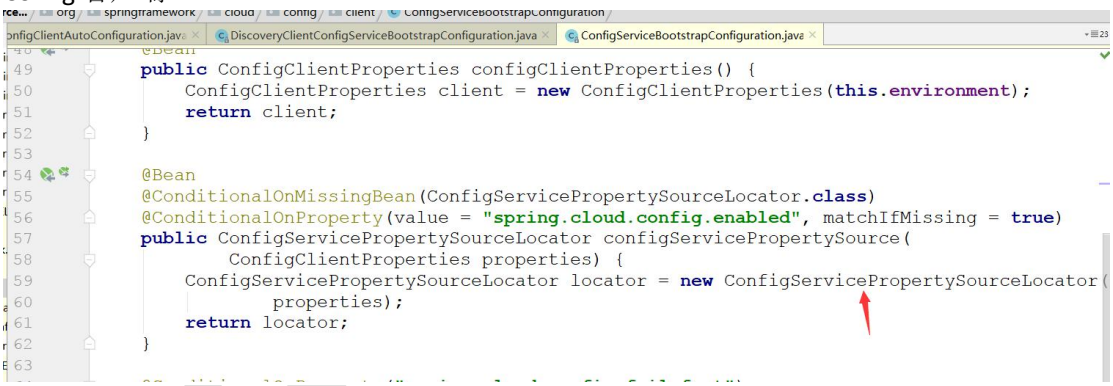
服务端的加密 controller

```java
 */
@Configuration(proxyBeanMethods = false)
public class ConfigServerEncryptionConfiguration {

    @Autowired(required = false)
    private TextEncryptorLocator encryptor;

    @Autowired
    private ConfigServerProperties properties;

    @Bean
    public EncryptionController encryptionController() {
        EncryptionController controller = new EncryptionController(this.encryptor);
        controller.setDefaultApplicationName(this.properties.getDefaultApplicationName()
        controller.setDefaultProfile(this.properties.getDefaultProfile());
        return controller;
    }
```

```java
@RequestMapping(value = "encrypt", method = RequestMethod.POST)
public String encrypt(@RequestBody String data,
        @RequestHeader("Content-Type") MediaType type) {
    return encrypt(defaultApplicationName, defaultProfile, data, type);
}
```

```java
@RequestMapping(value = "decrypt", method = RequestMethod.POST)
public String decrypt(@RequestBody String data,
        @RequestHeader("Content-Type") MediaType type) {
    return decrypt(defaultApplicationName, defaultProfile, data, type);
}
```

Config 客户端



```
ientConfigServiceBootstrapConfiguration.java | ConfigServiceBootstrapConfiguration.java | ConfigServicePropertySourceLocator.java
85
86      @Override
87      @Retryable(interceptor = "configServerRetryInterceptor")
88      public org.springframework.core.env.PropertySource<?> locate(
89              org.springframework.core.env.Environment environment) {
90          ConfigClientProperties properties = this.defaultProperties.override(environment);
91          CompositePropertySource composite = new OriginTrackedCompositePropertySource(
92                  name: "configService");
93          RestTemplate restTemplate = this.restTemplate == null
94                  ? getSecureRestTemplate(properties) : this.restTemplate;
95          Exception error = null;
96          String errorBody = null;
97          try {
98              String[] labels = new String[] { "" };
99              if (StringUtils.hasText(properties.getLabel())) {
100                 labels = StringUtils
101                         .commaDelimitedListToStringArray(properties.getLabel());
102             }
103             String state = ConfigClientStateHolder.getState();
104             // Try all the labels until one works
105             for (String label : labels) {
106                 Environment result = getRemoteEnvironment(restTemplate, properties,
107                         label.trim(), state);
108                 if (result != null) {
109                     log(result);
```

这个方法就发起了对 config 服务端的调用获取服务列表

```
// try all the labels until one works
for (String label : labels) {
    Environment result = getRemoteEnvironment(restTemplate, properties,
            label.trim(), state);
    if (result != null) {
```

```
cryptionConfiguration.java | DiscoveryClientConfigServiceBootstrapConfiguration.java | ConfigServiceBootstrapConfiguration.java | ConfigServicePropertySourceLocator.java
3           for (int i = 0; i < noOfUrls; i++) {
4               Credentials credentials = properties.getCredentials(i);
5               String uri = credentials.getUri();
6               String username = credentials.getUsername();
7               String password = credentials.getPassword();
8
9               logger.info( o: "Fetching config from server at : " + uri);
0
1               try {
2                   HttpHeaders headers = new HttpHeaders();
3                   headers.setAccept(
4                           Collections.singletonList(MediaType.parseMediaType(V2_JSON)));
5                   addAuthorizationToken(properties, headers, username, password);
6                   if (StringUtils.hasText(token)) {
7                       headers.add(TOKEN_HEADER, token);
8                   }
9                   if (StringUtils.hasText(state) && properties.isSendState()) {
0                       headers.add(STATE_HEADER, state);
1                   }
2
3                   final HttpEntity<Void> entity = new HttpEntity<>((Void) null, headers);
4                   response = restTemplate.exchange( url: uri + path, HttpMethod.GET, entity,
5                           Environment.class, args);
```

配置的动态刷新

当调用 http://localhost:8086/actuator/refresh 接口的时候就会调用到这个类

```
 */
@Endpoint(id = "refresh")
public class RefreshEndpoint {

    private ContextRefresher contextRefresher;

    public RefreshEndpoint(ContextRefresher contextRefresher) { this.

    @WriteOperation
    public Collection<String> refresh() {
        Set<String> keys = this.contextRefresher.refresh();
        return keys;
    }
```

```
public synchronized Set<String> refresh() {
    Set<String> keys = refreshEnvironment();
    this.scope.refreshAll();
    return keys;
}
```

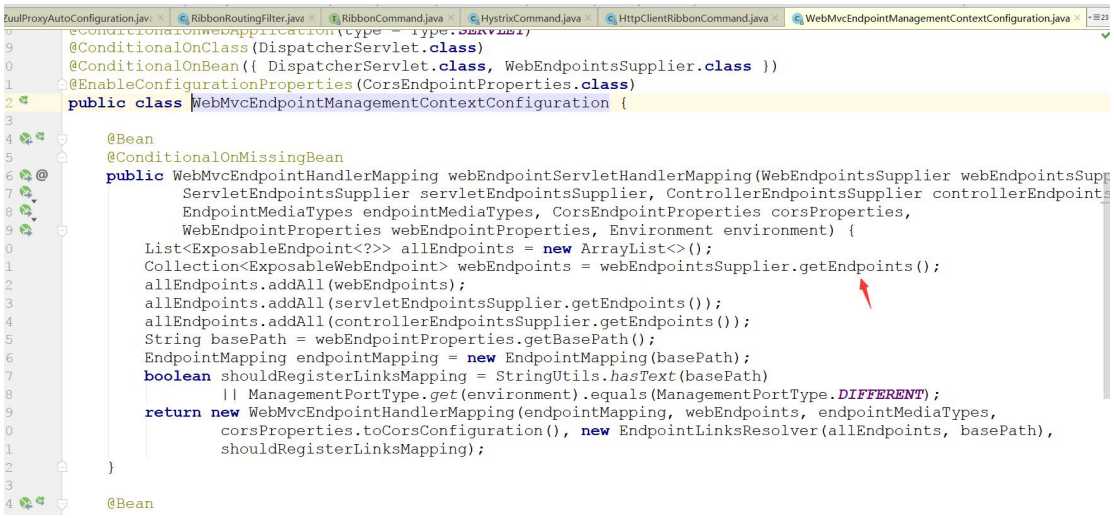
跟我们写分布式配置中心思路是一样的，先更新 spring 容器的属性，然后再更新 @RefreshScope 注解的类刷新，其实就是自定义 scope，由自己维护实例，这里就不赘述

36、actuator 源码

actuator 这个就是通过各种 endpoint 的注解来实现对各种监控信息的接口的调用

handlerMapping 对象

获取所有有@EndPoint 注解的对象



```
@ConditionalOnClass(DispatcherServlet.class)
@ConditionalOnBean({ DispatcherServlet.class, WebEndpointsSupplier.class })
@EnableConfigurationProperties(CorsEndpointProperties.class)
public class WebMvcEndpointManagementContextConfiguration {

    @Bean
    @ConditionalOnMissingBean
    public WebMvcEndpointHandlerMapping webEndpointServletHandlerMapping(WebEndpointsSupplier webEndpointsSupp
            ServletEndpointsSupplier servletEndpointsSupplier, ControllerEndpointsSupplier controllerEndpoints
            EndpointMediaTypes endpointMediaTypes, CorsEndpointProperties corsProperties,
            WebEndpointProperties webEndpointProperties, Environment environment) {
        List<ExposableEndpoint<?>> allEndpoints = new ArrayList<>();
        Collection<ExposableWebEndpoint> webEndpoints = webEndpointsSupplier.getEndpoints();
        allEndpoints.addAll(webEndpoints);
        allEndpoints.addAll(servletEndpointsSupplier.getEndpoints());
        allEndpoints.addAll(controllerEndpointsSupplier.getEndpoints());
        String basePath = webEndpointProperties.getBasePath();
        EndpointMapping endpointMapping = new EndpointMapping(basePath);
        boolean shouldRegisterLinksMapping = StringUtils.hasText(basePath)
                || ManagementPortType.get(environment).equals(ManagementPortType.DIFFERENT);
        return new WebMvcEndpointHandlerMapping(endpointMapping, webEndpoints, endpointMediaTypes,
                corsProperties.toCorsConfiguration(), new EndpointLinksResolver(allEndpoints, basePath),
                shouldRegisterLinksMapping);
    }

    @Bean
```

```
private Collection<EndpointBean> createEndpointBeans() {
    Map<EndpointId, EndpointBean> byId = new LinkedHashMap<>();
    String[] beanNames = BeanFactoryUtils.beanNamesForAnnotationIncludingAncestors(this.applicationCont
            Endpoint.class);
    for (String beanName : beanNames) {
        if (!ScopedProxyUtils.isScopedTarget(beanName)) {
            EndpointBean endpointBean = createEndpointBean(beanName);
            EndpointBean previous = byId.putIfAbsent(endpointBean.getId(), endpointBean);
            Assert.state( expression: previous == null, () -> "Found two endpoints with the id '" + endpoi
                    + endpointBean.getBeanName() + "' and '" + previous.getBeanName() + "'");
        }
    }
    return byId.values();
}
```

根据前面收集的所有有@EndPoint 注解的类，然后根据这些 EndPoint 建立 url 和 HandlerMethod 的映射关系，这个 url 就是 EndPoint 注解的 id

```
-5.2.2.RELEASE-sources.jar / org / springframework / web / servlet / handler / AbstractHandlerMethodMapping
JGitEnvironmentRepository.java | AbstractScmEnvironmentRepository.java | EncryptionAutoConfiguration.java | ConfigServerBootstrapConfiguration.java | AbstractHandlerMethodMapping.java
    }


    // Handler method detection

    /**
     * Detects handler methods at initialization.
     * @see #initHandlerMethods
     */
    @Override
    public void afterPropertiesSet() {
        initHandlerMethods();
    }

    /**
```

```
@Override
protected void initHandlerMethods() {
    for (ExposableWebEndpoint endpoint : this.endpoints) {
        for (WebOperation operation : endpoint.getOperations()) {
            registerMappingForOperation(endpoint, operation);
        }
    }
    if (this.shouldRegisterLinksMapping) {
        registerLinksMapping();
    }
}
```

```
private void registerMappingForOperation(ExposableWebEndpoint endpoint, WebOperation operation) {
    WebOperationRequestPredicate predicate = operation.getRequestPredicate();
    String path = predicate.getPath();
    String matchAllRemainingPathSegmentsVariable = predicate.getMatchAllRemainingPathSegmentsVariable()
    if (matchAllRemainingPathSegmentsVariable != null) {
        path = path.replace( target: "{*" + matchAllRemainingPathSegmentsVariable + "}", replacement: "**");
    }
    ServletWebOperation servletWebOperation = wrapServletWebOperation(endpoint, operation,
            new ServletWebOperationAdapter(operation));
    registerMapping(createRequestMappingInfo(predicate, path), new OperationHandler(servletWebOperation
            this.handleMethod);
}
```

其中 handler 类和 method 方法就是

```
        new ServletWebOperationAdapter(operation));
registerMapping(createRequestMappingInfo(predicate, path), new OperationHandler(servletWebOperation),
        this.handleMethod);
```

所以所有针对 actuator 的调用都会走到这个类和 handle 方法中

```
private final class OperationHandler {

    private final ServletWebOperation operation;

    OperationHandler(ServletWebOperation operation) {
        this.operation = operation;
    }

    @ResponseBody
    Object handle(HttpServletRequest request, @RequestBody(required = false) Map<String, String> body) {
        return this.operation.handle(request, body);
    }

    @Override
    public String toString() { return this.operation.toString(); }
```

这里 handle 方法就会调用到各自的有@EndPoint 注解的类中