链滴

Gradle 多模块 +SpringCloud 微服务实践

作者：someone27889
原文链接：https://ld246.com/article/1578357593309
来源网站：链滴

1.Project

REP3_CLOUD

2.Cloud 简单配置

2.1.父模块配置

主要就是配置一下通用依赖和基本插件

```
buildscript {
    ext {
        springBootVersion = '2.2.2.RELEASE'
        springBootManagementVersion = '1.0.8.RELEASE'
        springCloudVersion = 'Hoxton.SR1'
    }
    repositories {
        maven { url 'http://maven.aliyun.com/nexus/content/groups/public/' }
        mavenCentral()
        maven { url 'https://repo.spring.io/snapshot' }
        maven { url 'https://repo.spring.io/milestone' }
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
        classpath("io.spring.gradle:dependency-management-plugin:${springBootManagement
ersion}")
    }
}

allprojects {
```

```
    group "com.github.rep3.cloud"
    version "1.0.0"
}

subprojects {
    apply plugin: 'java'
    apply plugin: 'application'
    apply plugin: 'idea'
    apply plugin: 'eclipse'
    apply plugin: 'war'
    apply plugin: 'org.springframework.boot'
    apply plugin: 'io.spring.dependency-management'
    sourceCompatibility = JavaVersion.VERSION_1_8
    targetCompatibility = JavaVersion.VERSION_1_8
    jar {
        enabled = true
    }
    bootJar {
        classifier = 'boot'
    }
    repositories {
        maven { url 'http://maven.aliyun.com/nexus/content/groups/public/' }
        mavenCentral()
        maven { url 'https://repo.spring.io/snapshot' }
        maven { url 'https://repo.spring.io/milestone' }
    }
    dependencies {
        compile(
                'org.springframework.boot:spring-boot-starter-web',
                'org.springframework.boot:spring-boot-starter-tomcat',
                'org.springframework.boot:spring-boot-starter-actuator',
                'org.springframework.cloud:spring-cloud-starter',
                'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client',
                'com.google.guava:guava:23.0'

        )
        testCompile(
                "org.springframework:spring-test",
                "junit:junit:4.12"
        )
    }
    dependencyManagement {
        imports { mavenBom("org.springframework.boot:spring-boot-dependencies:${springBoo
Version}") }
        imports { mavenBom "org.springframework.cloud:spring-cloud-dependencies:${springCl
udVersion}" }
    }
}
```

2.2.EurekaServer 服务注册中心

配置一下 EurekaServer 端依赖

```
mainClassName = 'com.github.rep3.cloud.eureka.EurekaApplication'
```

```
springBoot {
    mainClassName = 'com.github.rep3.cloud.eureka.EurekaApplication'
    buildInfo {
        properties {
            artifact = 'eureka'
            version = '1.0.0'
            group = 'com.github.rep3.cloud'
            name = 'eureka'
        }
    }
}
bootJar {
    classifier = 'boot'
}
dependencies {
    compile 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-server'
}
```

2.3.Zuul 网关

配置 zuul 依赖

```
mainClassName = 'com.github.rep3.cloud.zuul.Rep3ZuulApplication'
springBoot {
    mainClassName = 'com.github.rep3.cloud.zuul.Rep3ZuulApplication'
    buildInfo {
        properties {
            artifact = 'zuul'
            version = '1.0.0'
            group = 'com.github.rep3.cloud'
            name = 'zuul'
        }
    }
}
bootJar {
    classifier = 'boot'
}
dependencies {
    compile('org.springframework.cloud:spring-cloud-starter-netflix-zuul')
}
```

Cloud 基本的依赖就配置完成了，接下来就配置配置 YAML 跑起来试试看

3.Springboot

SpringBoot 不做过多赘述，主要看一下配置文件加载的优先级

配置加载顺序 1->12 越小优先级越高

```
1.命令行传入参数
2.SPRING_APPLICATION_JSON中的属性
3.java:comp/env中的JNDI属性
4.Java的系统属性:System.getProperties()
5.操作系统环境变量
```

6.random*配置的随机属性
7.位于当前应用jar包之外的application-profile.yaml配置文件
8.位于当前应用jar包之内的application-profile.yaml配置文件
9.位于jar包之外的appication.yml配置
10.位于jar包之内的application.yml配置
11.@Configuration配置类中
12.应用默认属性比如port默认8080

4.SpringBootActutor

微服务的监控和管理，将依赖配置在根模块中，所有的子模块 SpringBoot 就可以使用了

'org.springframework.boot:spring-boot-starter-actuator'

启动以后查看 /health 就可以看到监控信息了

放开所有监控点

```
management:
  endpoints:
    web:
      exposure:
        include: '*'
```

查看所有监控点

http://192.168.0.101:8002/actuator

返回了所有监控点信息

{"_links":{"self":{"href":"http://192.168.0.101:8002/actuator","templated":false},"archaius":{"href":"http://192.168.0.101:8002/actuator/archaius","templated":false},"beans":{"href":"http://192.18.0.101:8002/actuator/beans","templated":false},"caches-cache":{"href":"http://192.168.0.101:002/actuator/caches/{cache}","templated":true},"caches":{"href":"http://192.168.0.101:8002/acuator/caches","templated":false},"health-path":{"href":"http://192.168.0.101:8002/actuator/heath/{*path}","templated":true},"health":{"href":"http://192.168.0.101:8002/actuator/health","teplated":false},"info":{"href":"http://192.168.0.101:8002/actuator/info","templated":false},"condiions":{"href":"http://192.168.0.101:8002/actuator/conditions","templated":false},"configprops":"href":"http://192.168.0.101:8002/actuator/configprops","templated":false},"env":{"href":"http:/192.168.0.101:8002/actuator/env","templated":false},"env-toMatch":{"href":"http://192.168.0.01:8002/actuator/env/{toMatch}","templated":true},"loggers-name":{"href":"http://192.168.0.11:8002/actuator/loggers/{name}","templated":true},"loggers":{"href":"http://192.168.0.101:800/actuator/loggers","templated":false},"heapdump":{"href":"http://192.168.0.101:8002/actuatorheapdump","templated":false},"threaddump":{"href":"http://192.168.0.101:8002/actuator/threddump","templated":false},"metrics-requiredMetricName":{"href":"http://192.168.0.101:8002/ctuator/metrics/{requiredMetricName}","templated":true},"metrics":{"href":"http://192.168.0.11:8002/actuator/metrics","templated":false},"scheduledtasks":{"href":"http://192.168.0.101:802/actuator/scheduledtasks","templated":false},"mappings":{"href":"http://192.168.0.101:8002/ctuator/mappings","templated":false},"refresh":{"href":"http://192.168.0.101:8002/actuator/refesh","templated":false},"features":{"href":"http://192.168.0.101:8002/actuator/features","templted":false},"service-registry":{"href":"http://192.168.0.101:8002/actuator/service-registry","temlated":false}}}

1./beans: 所有bean信息

2./caches:缓存信息
3./health:服务状态
4./info:服务自定义信息
5./configprops:应用配置的属性信息
6./env:应用所有可用环境信息
7./mappings:所有SpringMvc映射关系信息
8./metrics:内存信息,线程信息,垃圾回收信息等
9./dump:运行中的线程信息
10./trace:http跟踪信息
...

5.Eureka 服务治理中心

服务注册 :主要用来开微服务架构中有多少服务都部署在哪里

服务发现 :服务之间调用关系的处理，比如 A 服务调用 C 服务，会直接像注册中心发出咨询请求，然注册中心将 C服务的位置清单 返回给 A 服务，A 便获得了 多个C服务的位置 ,然后就可以调用其一。

SpringCloudEureka 使用 Netflix Eureka 来实现服务注册与发现，既包括了注册端也包括了服务端

服务端依赖

```
compile 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-server'
```

客户端依赖

```
compile 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client',
```

服务端 Application 开启 @EnableEurekaServer 注解标识为服务中心
同时要配置配置文件，给出 hostname 地址，客户端策略等

```
eureka:
  instance:
    hostname: localhost
  client:
    register-with-eureka: false
    fetch-registry: false
```

客户端需要开启 @EnableEurekaClient 标示为 Eureka 客户端，同时配置出服务中心的地址

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8000/eureka/
```

5.1 高可用

上面搭建的只有一个服务注册中心 1 Eureka : N Springboot ,当这一台 Eureka 崩掉以后，所有服务不能正常访问

在 Eureka 中，所有服务既是服务提供者，也是服务消费者，Eureka 本身也不例外，现在放开一下配

，并新建一个 Eureka 服务，两者互相注册成服务

```
  client:
    register-with-eureka: false
    fetch-registry: false
```

EurekaA

```
server:
  port:8001
spring:
  application:
    name:eureka1
eureka:
  instance:
    hostname: peer1
  client:
    serviceUrl:
      defaultZone: http://peer2:8000/eureka
```

EurekaB

```
server:
  port:8000
spring:
  application:
    name:eureka2
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone: http://peer1:8001/eureka
```

最后将 peer 的 ip 地址解析在/etc/hosts 中即可完成 eureka 高可用部署

高可用服务注册

```
spring:
  profiles: dev
  application:
    name: auth
server:
  port: 8002
eureka:
  client:
    serviceUrl:
      defaultZone: http://peer1/eureka/,http://peer2/eureka/
management:
  endpoints:
    web:
      exposure:
        include: '*'
```

5.2 负载均衡

ribbon 已经在 zuul 里了所以不必添加依赖

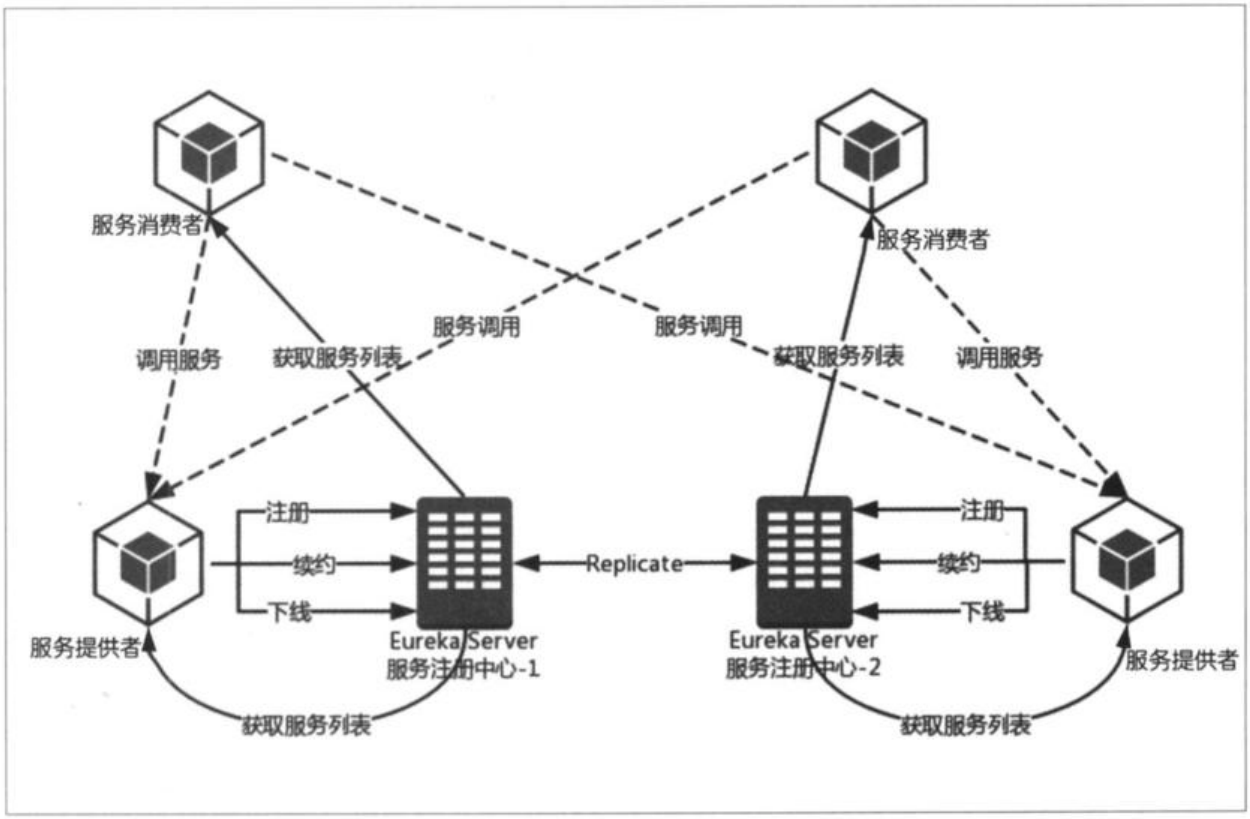
ribbon 和 eureka 联合使用时，在 eureka 的服务发现基础上实现了一套 服务选择 的策略，从而达每个服务负载均衡

将 @EnableEurekaClient 替换为 @EnableDiscoveryClient

当使用服务时，ribbon 会轮询服务从而走负载较低的服务达到负载均衡的效果

5.3 Eureka 基础架构

服务注册中心: Eureka提供的服务端,提供服务注册与发现功能
服务提供: Eureka提供的客户端,将自己的服务注册到Eureka中
服务消费者: 消费者从应用中心获取服务列表,从而使消费者知道到何处调用服务,ribbon来实现服务消,另外还有Feign消费方式



服务提供者
1.服务注册
　服务提供者在启动的时候`发送REST`请求到Eureka,其中包含了自身服务的信息
Eureka接收到请求后,将元数据信息存储在`双层Map`中,第一层key为服务名称,第二层是具体服务实例称.

2.服务同步
服务提供者可能注册到了不同的注册中心,他们的信息分别被多个eureka维护
所以访问其中一台eureka是无法调用到其余服务的,那么将eureka构建成集群模式
当服务提供者发送`REST请求`到其中一个注册中心时
eureka将请求转发给集群中其他的eureka中,从而实现同步

3.服务续约
心跳检测来观测哪些服务死掉,死掉的服务会被剔除出注册中心集群中去
其中有两个配置项
服务续约任务调用时间间隔,默认为30秒
eureka.instance.lease-renewal-interval-in-seconds=30
服务失效时间默认为90秒
eureka.instance.lease-expiration-duration-in-seconds=90

服务消费者
1.获取服务
消费者访问Eureka中的服务时,Eureka会提供一份只读的服务清单,该清单被eureka缓存并每30秒更新次
eureka.client.registry-fetch-interval-seconds配置缓存服务清单时间
2.服务调用
服务消费者获取服务清单后,通过服务名可以获得具体的提供服务的实例名称和服务元数据信息
客户端决定需要调用哪个服务实例,如果使用ribbon,则使用了轮询方式调用,从而实现客户端的负载均衡
3.服务下线
正常关闭服务实例时,服务实例发送`REST请求给EurekaServer`告诉服务中心我要下线了,服务中心收后将服务状态设置为`Down`并传播下线事件

服务注册中心
1.失效剔除
发生不可预料的错误时导致服务崩溃但eureka没有收到下线消息的时候,由于服务没有及时续约,所以剔除
2.自我保护
请求重试,断路器等机制

部分源码解析

```
//DiscoveryClient 为接口,抽象了一个服务中心基本需要实现的功能
// 所以注册中心可以换位任一实现了该接口的类,但不用修改代码
public class EurekaDiscoveryClient implements DiscoveryClient {

    // 一段描述
    public static final String DESCRIPTION = "Spring Cloud Eureka Discovery Client";
    // 客户端
    private final EurekaClient eurekaClient;
    // 客户端配置
    private final EurekaClientConfig clientConfig;

    // 根据服务ID获取服务实例列表
    @Override
    public List<ServiceInstance> getInstances(String serviceId) {
        List<InstanceInfo> infos = this.eurekaClient.getInstancesByVipAddress(serviceId,
                false);
        List<ServiceInstance> instances = new ArrayList<>();
        for (InstanceInfo info : infos) {
            instances.add(new EurekaServiceInstance(info));
        }
        return instances;
    }

    //获取服务名称
    public List<String> getServices() {
```

```
        Applications applications = this.eurekaClient.getApplications();
        if (applications == null) {
            return Collections.emptyList();
        }
        List<Application> registered = applications.getRegisteredApplications();
        List<String> names = new ArrayList<>();
        for (Application app : registered) {
            if (app.getInstances().isEmpty()) {
                continue;
            }
            names.add(app.getName().toLowerCase());

        }
        return names;
    }
}
// EurekaClient抽象了许多接口来定义Client的行为
public interface EurekaClient extends LookupService {
   // 通过Region获取Applications,Region为yaml中设置的字符串
   // Applicaions中包含了appNameApplicationMap等
  // Applications 包装eureka服务器返回的所有注册表信息的类
    public Applications getApplicationsForARegion(@Nullable String region);
  // 通过地址获取 服务注册表
    public Applications getApplications(String serviceUrl);
  // 注册监听
    public void registerEventListener(EurekaEventListener eventListener);
  // 取消监听
    public boolean unregisterEventListener(EurekaEventListener eventListener);
  // 心跳检测
    public HealthCheckHandler getHealthCheckHandler();
  // 停止服务
    public void shutdown();
  // 获取Client的配置
    public EurekaClientConfig getEurekaClientConfig();
    public ApplicationInfoManager getApplicationInfoManager();
}
........
```

EndpointUtils.java

```
// 从属性文件中获取要与eureka客户端对话的所有eureka服务URL的列表。
 public static Map<String, List<String>> getServiceUrlsMapFromConfig(EurekaClientConfig cl
entConfig, String instanceZone, boolean preferSameZone) {
        Map<String, List<String>> orderedUrls = new LinkedHashMap<>();
        // 从配置文件中读取Region返回
    // 通过eureka.client.region属性来定义
    String region = getRegion(clientConfig);
    // 读取 eureka.client.defaultZone
        String[] availZones = clientConfig.getAvailabilityZones(clientConfig.getRegion());
        if (availZones == null || availZones.length == 0) {
            availZones = new String[1];
            availZones[0] = DEFAULT_ZONE;
        }
        logger.debug("The availability zone for the given region {} are {}", region, availZones);
```

```
    int myZoneOffset = getZoneOffset(instanceZone, preferSameZone, availZones);

    String zone = availZones[myZoneOffset];
// 获取zone中所有配置的eurekaServer地址
    List<String> serviceUrls = clientConfig.getEurekaServerServiceUrls(zone);
    if (serviceUrls != null) {
        orderedUrls.put(zone, serviceUrls);
    }
    int currentOffset = myZoneOffset == (availZones.length - 1) ? 0 : (myZoneOffset + 1);
    while (currentOffset != myZoneOffset) {
        zone = availZones[currentOffset];
        serviceUrls = clientConfig.getEurekaServerServiceUrls(zone);
        if (serviceUrls != null) {
// 最后放入 服务列表中
            orderedUrls.put(zone, serviceUrls);
        }
        if (currentOffset == (availZones.length - 1)) {
            currentOffset = 0;
        } else {
            currentOffset++;
        }
    }
    if (orderedUrls.size() < 1) {
        throw new IllegalArgumentException("DiscoveryClient: invalid serviceUrl specified!");
    }
    return orderedUrls;
}
```

服务注册

在 DiscoveryClient 的构造函数中调用了 initScheduledTasks 函数，初始化一些任务

```java
if (clientConfig.shouldFetchRegistry() && !fetchRegistry( forceFullRegistryFetch: false)) {
    fetchRegistryFromBackup();
}

// call and execute the pre registration handler before all background tasks (inc registration) is started
if (this.preRegistrationHandler != null) {
    this.preRegistrationHandler.beforeRegistration();
}

if (clientConfig.shouldRegisterWithEureka() && clientConfig.shouldEnforceRegistrationAtInit()) {
    try {
        if (!register() ) {
            throw new IllegalStateException("Registration error at startup. Invalid server response.");
        }
    } catch (Throwable th) {
        logger.error("Registration error at startup: {}", th.getMessage());
        throw new IllegalStateException(th);
    }
}

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

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

// This is a bit of hack to allow for existing code using DiscoveryManager.getInstance()
```

函数中实力化了一个 InstanceInfoReplicator

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

该类中的 run 函数，这句就是调用注册了，

```java
}

public void run() {
    try {
        discoveryClient.refreshInstanceInfo();

        Long dirtyTimestamp = instanceInfo.isDirtyWithTime();
        if (dirtyTimestamp != null) {
            discoveryClient.register();
            instanceInfo.unsetIsDirty(dirtyTimestamp);
        }
    } catch (Throwable t) {
        logger.warn("There was a problem with the instance info replicator", t);
    } finally {
        Future next = scheduler.schedule( command: this, replicationIntervalSeconds, TimeUnit.SECONDS);
        scheduledPeriodicRef.set(next);
    }
}

}
```

```java
    */
    boolean register() throws Throwable {
        logger.info(PREFIX + "{}: registering service...", appPathIdentifier);
        EurekaHttpResponse<Void> httpResponse;
        try {
            httpResponse = eurekaTransport.registrationClient.register(instanceInfo);
        } catch (Exception e) {
            logger.warn(PREFIX + "{} - registration failed {}", appPathIdentifier, e.getMessage(), e);
            throw e;
        }
        if (logger.isInfoEnabled()) {
            logger.info(PREFIX + "{} - registration status: {}", appPathIdentifier, httpResponse.getStatusCode());
        }
        return httpResponse.getStatusCode() == Status.NO_CONTENT.getStatusCode();
    }

    /**
```

在往下就是 http 层了，层层专递的 InstanceInfo 就是实例的一些配置信息了

```java
@Override
public EurekaHttpResponse<Void> register(InstanceInfo info) {
    String urlPath = serviceUrl + "apps/" + info.getAppName();

    HttpHeaders headers = new HttpHeaders();
    headers.add(HttpHeaders.ACCEPT_ENCODING,  headerValue: "gzip");
    headers.add(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE);

    ResponseEntity<Void> response = restTemplate.exchange(urlPath, HttpMethod.POST,
            new HttpEntity<>(info, headers), Void.class);

    return anEurekaHttpResponse(response.getStatusCodeValue())
            .headers(headersOf(response)).build();
}
```

服务获取与续约

还在 initScheduledTasks 中

```java
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);
}
```

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

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

    // InstanceInfo replicator
    instanceInfoReplicator = new InstanceInfoReplicator(
```

registryFetchIntervalSeconds:eureka.client.registry-fetch-interval-seconds

renewalIntervalInSecs:eureka.instance.lease-renewal-interval-in-seconds

其最后也是通过 RestTemplate 去请求服务端

6.Ribbon

负载均衡设备/模块 都会维护一个下挂可用的服务清单，发送心跳检测
剔除不可用设备，当客户端发送请求到负载均衡设备时，该设备按照某种
算法，从维护的清单中抽取出一台服务器，并进行转发。

SpringCloudRibbon 客户端使用负载均衡

1.服务提供者向服务中心注册
2.服务消费者通过@LoadBalanced注解修饰过的RestTemplate访问服务

@LoadBalanced 注解给 RestTemplate 做标记，以使用负载均衡的客户端来配置它 LoadBalancerClient

```java
public interface LoadBalancerClient extends ServiceInstanceChooser {

    /**
     * Executes request using a ServiceInstance from the LoadBalancer for the specified
     * service.
     * @param serviceId The service ID to look up the LoadBalancer.
     * @param request Allows implementations to execute pre and post actions, such as
     * incrementing metrics.
     * @param <T> type of the response
     * @throws IOException in case of IO issues.
     * @return The result of the LoadBalancerRequest callback on the selected
     * ServiceInstance.
     */
    <T> T execute(String serviceId, LoadBalancerRequest<T> request) throws IOException;

    /**
     * Executes request using a ServiceInstance from the LoadBalancer for the specified
     * service.
```

```java
public interface ServiceInstanceChooser {

    /**
     * Chooses a ServiceInstance from the LoadBal
     * @param serviceId The service ID to look up
     * @return A ServiceInstance that matches the
     */
    ServiceInstance choose(String serviceId);

}
```

ServiceInstance choose(String serviceId) 根据传入的服务名，从负载均衡器中挑选一个对应的服实例

T execute(String serviceId,LoadBalancerRequest request): 从负载均衡器中提取服务来 执行 reque t

URI reconstructURI(ServiceInstance instance,URI original): 拼接请求地址

```java
@Configuration(proxyBeanMethods = false)
@ConditionalOnMissingClass("org.springframework.retry.support.RetryTemplate")
static class LoadBalancerInterceptorConfig {

    @Bean
    public LoadBalancerInterceptor ribbonInterceptor(
            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);
        };
    }

}
```

LoadBalancerInterceptor 的 Bean,用于对客户端发起的请求实时拦截

RestTemplateCstomizer 的 Bean,用于给 RestTemplate 增加 LoadBalancerInterceptor

```java
@LoadBalanced
@Autowired(required = false)
private List<RestTemplate> restTemplates = Collections.emptyList();
```

维护了一个 RestTemplate 列表，然后给每一个 RestTemplate 实例去初始化拦截器

```java
51      @Override
52 @    public ClientHttpResponse intercept(final HttpRequest request, final byte[] body,
53              final ClientHttpRequestExecution execution) throws IOException {
54          final URI originalUri = request.getURI();
55          String serviceName = originalUri.getHost();
56          Assert.state( expression: serviceName != null,
57                  message: "Request URI does not contain a valid hostname: " + originalUri);
58          return this.loadBalancer.execute(serviceName,
59                  this.requestFactory.createRequest(request, body, execution));
60      }
61
```

拦截器中注入了 LoadBalancerClient 和 LoadBalancerRequestFactory

最后交给 LoadBalancerClient 去执行 request

最后走到了 RibbonLoadBalancerClient

```
     * @throws IOException executing the request may result in an {@link IOException}
     */
    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
```

Server server = getServer(loadBalacer,hint);

getLoadBalancer通过Factory构造了默认的ZoneAwaerLoadBalaner

```
    if (loadBalancer == null) {
        return null;
    }
    // Use 'default' on a null hint, or just pass it on?
    return loadBalancer.chooseServer(hint != null ? hint : "default");
}

protected ILoadBalancer getLoadBalancer(String serviceId) {
```

然后调用了 ZoneAwaerLoadBalaner 中的 chooseServer 来选择一个服务，继而执行 request,选择略在 ZoneAwaerLoadBalaner的chooseServer中

负载均衡器

AbstractLoadBalancer:
ILoadBalancer的抽象实现,定义了
分组枚举:ALL所有服务实例,STATUS_UP正常服务实例,STATUS_NOT_UP停止服务实例.
实现了一个chooseServer函数,key为null,表示在选择具体服务实例时忽略key的条件判断.
getServerList(ServerGroup goup): 根据不同的分组选择不同的服务实例列表.
getLoadBalancerStats():获取LoadBalancerStats对象,储存负载均衡器中各个服务实例当前的属性和计信息.

BaseLoadBalancer:Ribbon负载均衡器的基础实现类
定义并维护了两个存储服务实例Server的对象列表,分别存储了所有服务实例和所有正常运行服务实例
定义了LoadBalancerStats
定义了IPing检查服务是否运行正常
定义了IPingStrategy Iping的执行策略对象
定义了IRule负载均衡的处理规则
...服务实例列表相关函数

DynamicServerListLoadBalancer继承BaseLoadBalancer,扩展了基础负载均衡器

负载均衡策略

AbstractLoadBalancerRule

RandomRule:随机选择策略
RoundRobinRule:线性轮询策略
RetryRule:重试机制策略
WeightedResponseTimeRule:权重计算策略

7.Hystrix 服务容错保护

单个服务出错断路

'org.springframework.cloud:spring-cloud-starterhystrix:1.4.7.RELEASE',

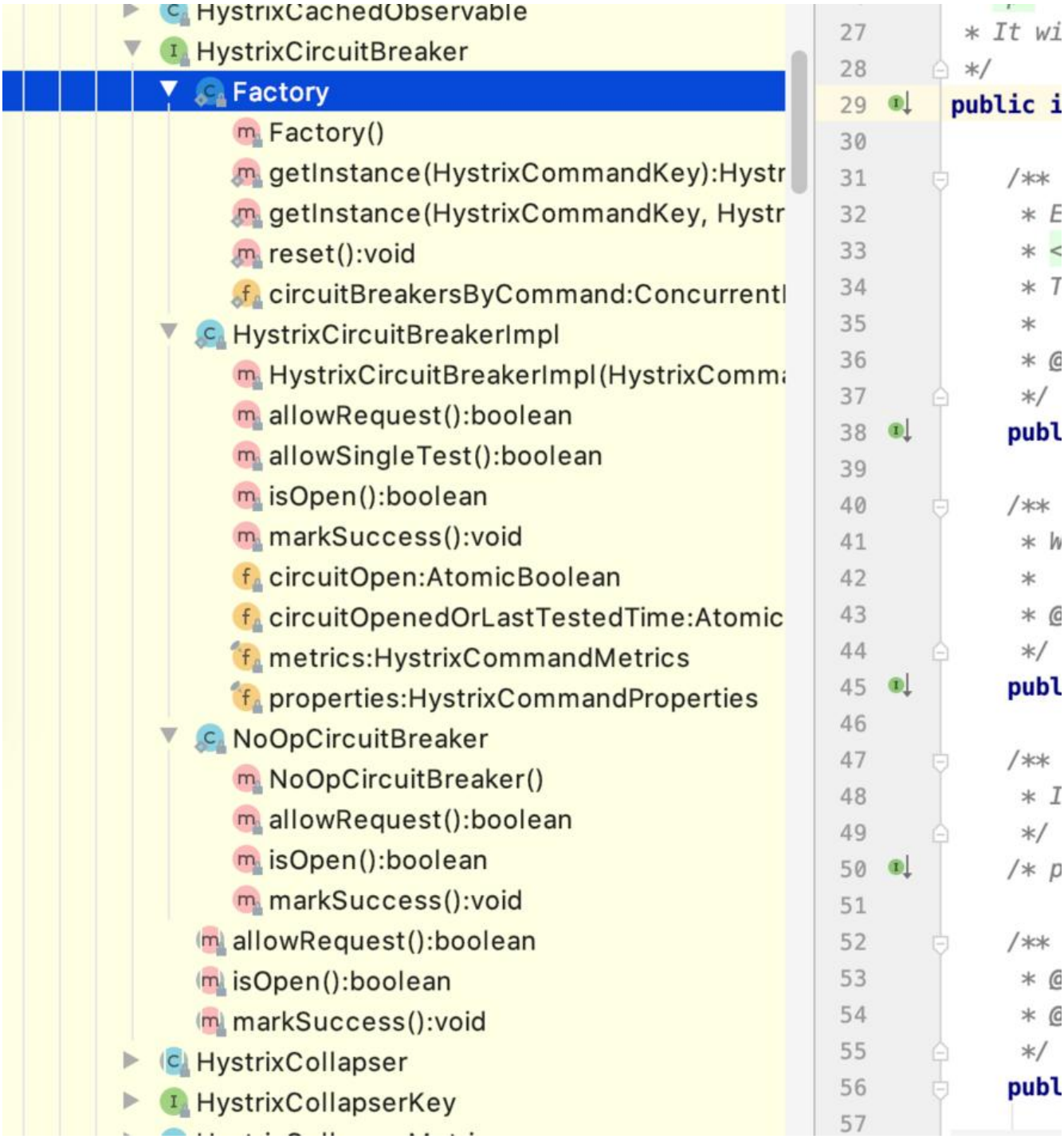
SpringCloudApplication 注解聚合了，服务注册负载均衡熔断机制等注解，可以直接标示到 Applicat on 上面

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker

@Service
public class DemoService {

    @Autowired
    RestTemplate restTemplate;
    // 如果函数出现错误,或者请求超时,那么将会出发熔断请求,并调用callback,返回结果
    @HystrixCommand(fallbackMethod = "helloFallback")
    public String toAuth() {
        return restTemplate.getForEntity("http://auth/home", String.class).getBody();
    }

    public String helloFallback(){
        return "error";
    }

}
```

工作流程

1.创建HystrixCommand/HystrixObservbleCommand Bean,用来表示对依赖服务的操作请求,同时递所有需要的参数.
2.命令执行,execute同步执行,queue异步执行,observe()HotOb返回Ob对象,toObServable返回Ob象ColdOb
3.结果是否被缓存
4.断路器是否打开
5.线程池/请求队列是否被占满
6.HystrixCommand.run()返回单一结果或跑一场
HystrixObservableCommand.construct()返回一个Ob来发射多个结果/通过onError发送错误通知
7.计算断路器健康程度,根据健康程度熔断/断路服务
8.fallback处理
9.返回成功的Response

断路器原理



HystrixCircuitBreaker 维护了三个抽象函数和三个类

allowRequest():每个Hystrix命令的请求通过该函数判断是否被执行

isOpen():断路器是否打开

markSuccess():闭合断路器

Factory:维护了一个Hystrix命令和HystrixCircuitBreaker的关系集合,key通过HystrixCommandKey定义,每一个Hystrix命令都有一个key标识,同时一个Hystrix命令也会在该合中找到它对应的断路器HystrixCirucuitBreaker

NoOpCirutiBreaker定义了一个什么都不做的断路器实现,允许所有请求,断路器始终闭合

HystrixCircuitBreakerImpl是本接口实现,定义了4个核心对象
HystrixCommandProperties:对应实例对象
HystrixCommandMetrics:记录度量指标对象
circuitOpen:是否打开标志
circuitOpenedOrLastTestedTime:打开或是上一次测试的时间

HystrixCircuitBreakerImpl 的 isOpen

```
@Override
public boolean isOpen() {
    if (circuitOpen.get()) {  打开标志为true那就直接返回true
        // if we're open we immediately return true and don't bother attempting to 'close' ourself
        return true;
    }
                通过健康指标运算
    // we're closed, so let's see if errors have made us so we should trip the circuit open
    HealthCounts health = metrics.getHealthCounts();
                请求总数是否在阀值内,在之内返回false
    // 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 f
        return false;
    }
                        错误百分比是否在阀值内,在之内返回false
    if (health.getErrorPercentage() < properties.circuitBreakerErrorThresholdPercentage().get()) {
        return false;
    } else {            打开断路器
        // 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
                        记录本次测试/打开时间
            circuitOpenedOrLastTestedTime.set(System.currentTimeMillis());
            return true;
        } else {
            // How could previousValue be true? If another thread was going through this code at th
            // caused another thread to set it to true already even though we were in the process o
            // In this case, we know the circuit is open, so let the other thread set the currentTi
            return true;
        }
    }
}

}
```

HystrixCircuitBreakerImpl 的 allowRequest,通过 isOpen 判断是否允许被请求

```java
@Override
public boolean allowRequest() {
    if (properties.circuitBreakerForceOpen().get()) {
        // properties have asked us to force the circuit open so we will a
        return false;
    }
    if (properties.circuitBreakerForceClosed().get()) {
        // we still want to allow isOpen() to perform it's calculations so
        isOpen();
        // properties have asked us to ignore errors so we will ignore the
        return true;
    }
    return !isOpen() || allowSingleTest();
}
```

!isOpen()||allowSingleTest() 通过配置文件中的 circuiBreakerSleepWindowInMilliseconds 属性眠时间，对比时间戳判断请求是否可以访问，如果休眠时间到达后请求访问失败，则断路器再次进入开状态，如果成功，断路器关闭，这句是切换断路器打开关闭状态的切换的实现。

```java
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.circuitBreakerS
        // 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 end of the 's
        if (circuitOpenedOrLastTestedTime.compareAndSet(timeCircuitOpenedOrWasLastTested, System.currentTimeMillis())
            // 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 did
            return true;
        }
    }
    return false;
}
```

8.Zuul路由管理

单独使用可以进行路由转发,类似nginx

```yaml
zuul:
  routes:
    {serviceId}:
      path: /user/**
      url: http://www.baidu.com/user/
```

上面将 /user的路由全部转发到下面对应的url中

多实例配置

```yaml
zuul:
  routes:
    {serviceId}:
      path: /user/**
```

```
      serviceId: {serviceId}
// 没有整合eureka这里应该关闭
ribbon:
  eureka:
    enabled: false
{serviceId}:
  ribbon:
    listOfServers:  http://www.baidu1.com/,http://www.baidu2.com/
```

忽略路由

```
zuul.ignored.parttens=/**/hello/**
```

本地跳转

```
zuul.routes.{serviceId}.url=forward:/local
```

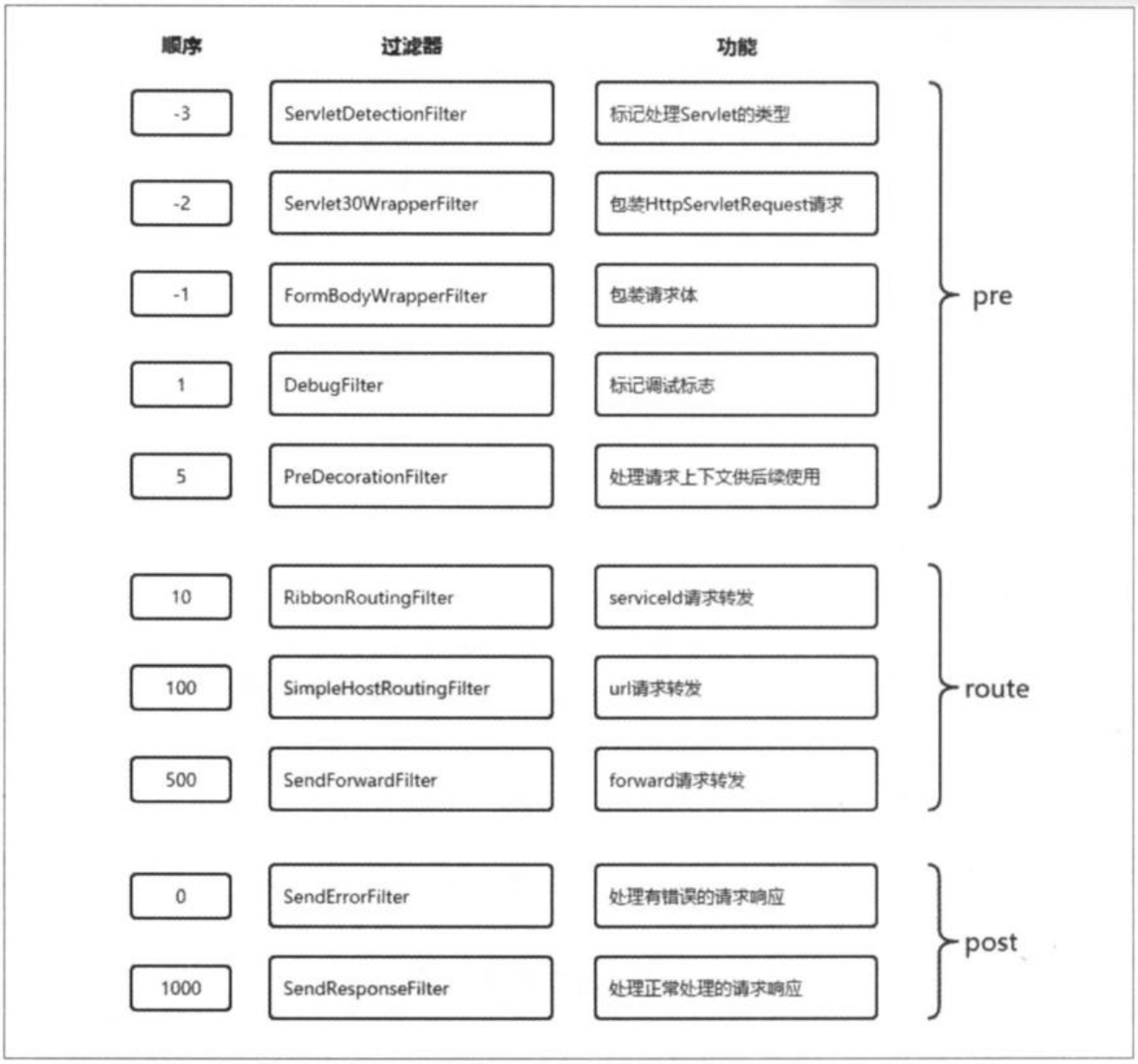
传递Http Header信息

```
zuul:
  routes:
    {router}:
      // 对制定路由开启自定义敏感头信息
      customSenstiveHeaders:true
      // 将制定路由敏感信息头设置为空
      senstiveHeaders
```

当遇到302跳转到具体微服务而不是网关时

```
zuul:
  // 跳转时设置Host为zuul
  addHostHeader : true
```

8.1四个阶段的核心过滤器



Pre: 路由转发之前

name	index	desc
ServletDetectionFilter	-3	检测前请求为DispatcherServelet/ZuulServlet处理运行
Servlet30WrapperFilter	-2	HttpservletRequest->Servlet30RequestWrapper
FormBodyWrapperFillter	-1	将ontentType为x-www-form-urlencoded/multipart-form-data请求->FormBodyRequestWrapper
DebugFilter	1	debug信息
PreDecorationFilter	5	设置请信息,进行路由匹配

route:路由转发中

name	index	desc
RibbonRouterFilter	10	通过serviceId和Ribbon/Hystrix对应用实例发起请求并将结果返回
SimpleHostRoutingFiler	100	过routesHost和路由规则向物理机器发送请求
SendForwardFillter	500	对forward.to 参数的请求进行处理,forward跳转

post route和error过滤器调用之后进入

name	index	desc
SendErrorFillter	0	包装错误信组成forward发送给网关来报错
SendResponseFiller	1000	返回应发送给客户端