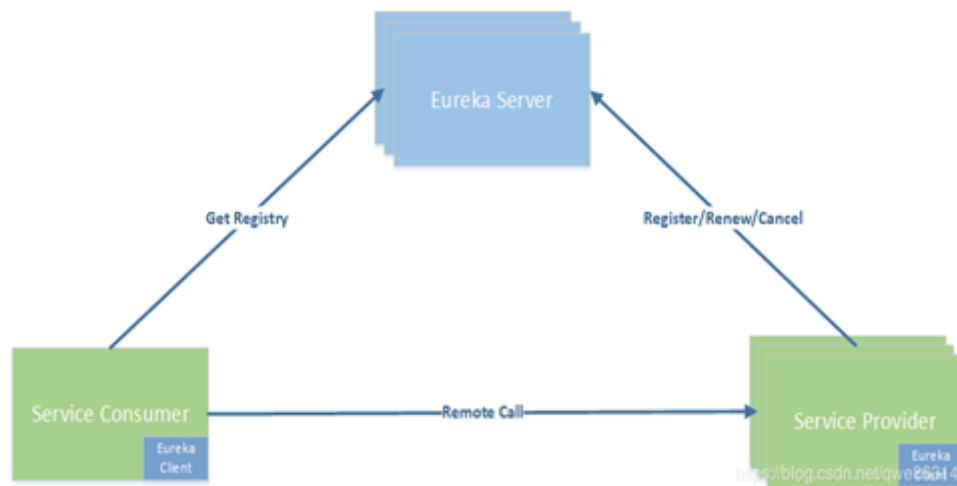


SpringCloud

1 eureka

1.1 eureka基本概念



eureka主要包含两个组件：Eureka Server 和 Eureka Client.

eureka server(注册中心)，提供了三个功能：

1、服务注册

服务提供者启动时，会通过 Eureka Client 向 Eureka Server 注册信息，Eureka Server 会存储该服务的信息，Eureka Server 内部有二层缓存机制来维护整个注册表

2、提供注册表

服务消费者在调用服务时，如果 Eureka Client 没有缓存注册表的话，会从 Eureka Server 获取最新的注册表

3、同步状态

Eureka Client 通过注册、心跳机制和 Eureka Server 同步当前客户端的状态

Eureka Client：注册中心客户端

Eureka Client 会拉取、更新和缓存 Eureka Server 中的信息。因此当所有的 Eureka Server 节点都宕掉，服务消费者依然可以使用缓存中的信息找到服务提供者，但是当服务有更改的时候会出现信息不一致

Register: 服务注册

服务的提供者，将自身注册到注册中心，服务提供者也是一个 Eureka Client。当 Eureka Client 向 Eureka Server 注册时，它提供自身的元数据，比如 IP 地址、端口，运行状况指示符 URL，主页等

Renew: 服务续约

Eureka Client 会每隔 30 秒发送一次心跳来续约。通过续约来告知 Eureka Server 该 Eureka Client 运行正常，没有出现问题。默认情况下，如果 Eureka Server 在 90 秒内没有收到 Eureka Client 的续约，Server 端会将实例从其注册表中删除

Eviction 服务剔除

当 Eureka Client 和 Eureka Server 不再有心跳时，Eureka Server 会将该服务实例从服务注册列表中删除，即服务剔除

Cancel: 服务下线

Eureka Client 在程序关闭时向 Eureka Server 发送取消请求。发送请求后，该客户端实例信息将从 Eureka Server 的实例注册表中删除。该下线请求不会自动完成，它需要调用以下内容：

```
DiscoveryManager.getInstance().shutdownComponent()
```

GetRegistry: 获取注册列表信息

Eureka Client 从服务器获取注册表信息，并将其缓存在本地。客户端会使用该信息查找其他服务，从而进行远程调用。该注册列表信息定期（每30秒钟）更新一次。每次返回注册列表信息可能与 Eureka Client 的缓存信息不同，Eureka Client 自动处理

Remote Call: 远程调用

当 Eureka Client 从注册中心获取到服务提供者信息后，就可以通过 Http 请求调用对应的服务；服务提供者有多个时，Eureka Client 客户端会通过 Ribbon 自动进行负载均衡

1.2 编写eureka server

(1) 引入依赖

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

(2) 编写启动类，在启动类上添加@EnableEurekaServer注解

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

(3) 编写配置文件

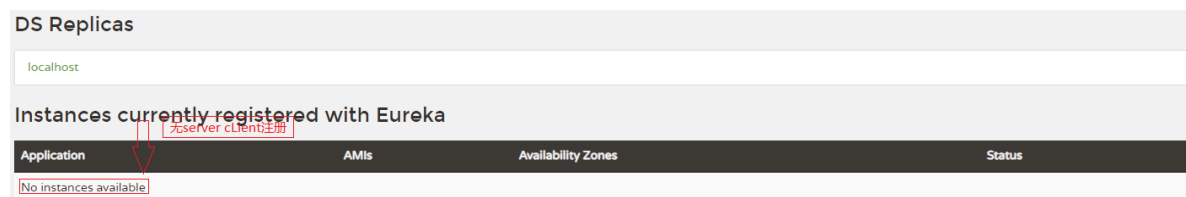
```
#应用名
spring.application.name=security
#端口号
server.port=8333
#主机名
eureka.instance.hostname=server2
#eureka server 地址
eureka.client.service-url.defaultZone=http://localhost:8333/eureka/
#是否开启自我保护
eureka.server.enableSelfPreservation=false
#是否将自己注册到eureka server
eureka.client.register-with-eureka=false
#是否从eureka server获取注册信息
eureka.client.fetch-registry=false
```

eureka.client.register-with-eureka=false：表示本应用是一个注册中心，当搭建eureka集群时，将此值改为true，与其他注册中心相互注册。

eureka.client.service-url.defaultZone: 与eureka server交互的地址，查询服务和注册服务都需要这个地址。

eureka.client.fetch-registry: 是否从其他节点获取注册信息，默认为true,因为此节点为单节点，所以不需要从其他节点获取信息，所以设置为false。

查看注册好之后的ui界面: <http://localhost:8333/>



这便是定义好的注册中心，从界面中可以看出，现在没有实例注册进去。

1.3 编写eureka client

(1) 引入依赖

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

(2) 编写启动类，在启动类上添加@EnableEurekaClient注解

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

(3) 编写配置文件

```
#注册到eureka server的应用名
spring.application.name=client-1
#端口号
server.port=8340
#将自己的ip注册到eureka server，若为false，则注册的为hostname
eureka.instance.prefer-ip-address=true
#注册到eureka server（eureka server的地址）
eureka.client.service-url.defaultZone = http://localhost:8333/eureka/
```

现已将服务名为client-1的微服务注册到地址为<http://localhost:8333/eureka/>的注册中心中，进去<http://localhost:8333/eureka/>此地址，UI端显示为：

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

DS Replicas

eureka的自我保护模式开启

localhost

Instances currently registered with Eureka

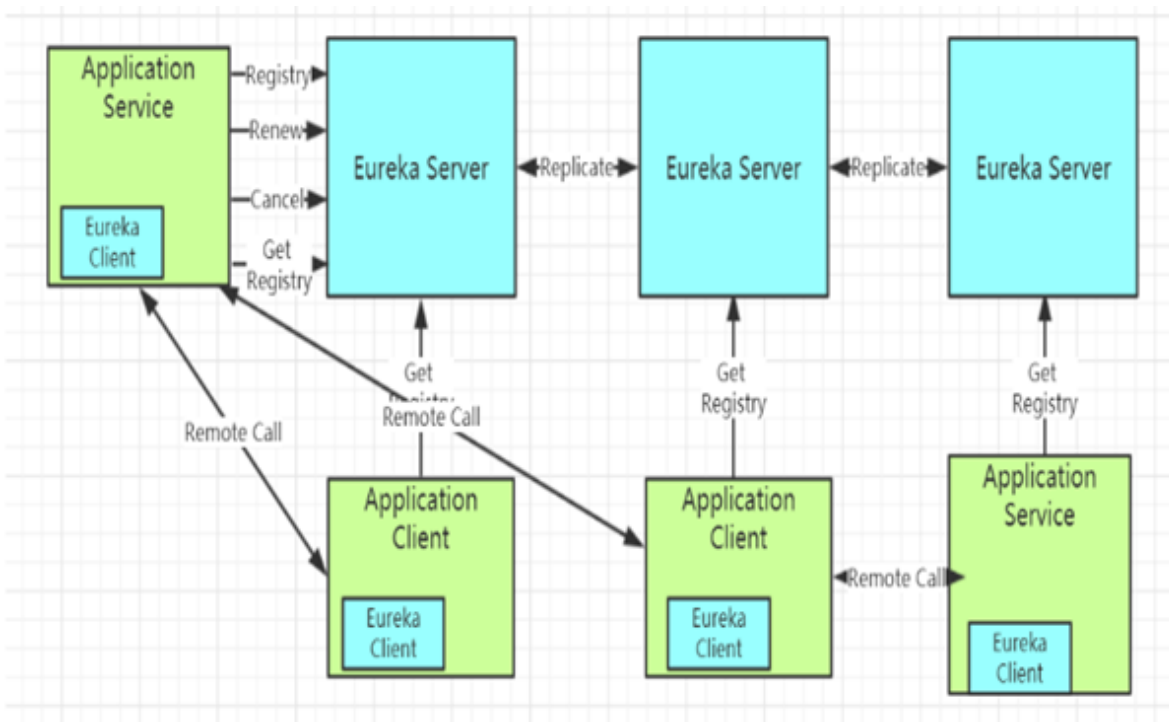
服务名为client-1的服务已注册

Application	AMIs	Availability Zones	Status
CLIENT-1	n/a (1)	(1)	UP (1) - localhost:client-1:8340

从图中可以看出服务名为client-1,端口号为8340的微服务已注册到注册中心中。

1.4 eureka的高可用（eureka集群）

1.4.1 eureka集群框架及概念



图中由三个Eureka Server组成一个集群

从图中可以看出 Eureka Server 集群相互之间通过 Replicate 来同步数据，相互之间不区分主节点和从节点，所有的节点都是平等的。在这种架构中，节点通过彼此互相注册来提高可用性，每个节点需要添加一个或多个有效的 serviceUrl 指向其他节点。

如果某台 Eureka Server 宕机，Eureka Client 的请求会自动切换到新的 Eureka Server 节点。当宕机的服务器重新恢复后，Eureka 会再次将其纳入到服务器集群管理之中。当节点开始接受客户端请求时，所有的操作都会进行节点间复制，将请求复制到其它 Eureka Server 当前所知的所有节点中

1.4.2 集群之间相互注册

首先：

将eureka.client.register-with-eureka和eureka.client.fetch-registry的值都改为true，或者不进行配置，因为默认为true。

配置如下：

第一个eureka server将前文配置修改，将其注册到另外两个eureka server中：

```
#应用名
spring.application.name=security
#端口号
```

```
server.port=8233
#主机名
eureka.instance.hostname=server2
#eureka server 地址
eureka.client.service-url.defaultZone=http://server3:8333/eureka/
#是否开启自我保护
#eureka.server.enableSelfPreservation=false
#是否将自己注册到eureka server
eureka.client.register-with-eureka=true
#是否从eureka server获取注册信息
eureka.client.fetch-registry=true
```

进入<http://localhost:8333/>可以看到:

DS Replicas

server2

Instances currently registered with Eureka

Application	AMIs	Availability Zones
No instances available		

General Info

Name	Value
total-avail-memory	343mb
environment	test
num-of-cpus	6
current-memory-usage	98mb (28%)
server-uptime	00:04
registered-replicas	http://server2:8233/eureka/
unavailable-replicas	http://server2:8233/eureka/

可以发现server2的服务已经注册<http://localhost:8333/>的注册中心中

再查看<http://localhost:8233/>:

DS Replicas

server3

Instances currently registered with Eureka

Application	AMIs	Availability Zones
No instances available		

General Info

Name	Value
total-avail-memory	335mb
environment	test
num-of-cpus	6
current-memory-usage	102mb (30%)
server-uptime	00:07
registered-replicas	http://server3:8333/eureka/
unavailable-replicas	http://server3:8333/eureka/

同样，可以发现server3的服务已经注册<http://localhost:8233/>的注册中心中
两个注册中心已经相互注册成为一个小集群。

1.5 微服务之间调用

注册两个微服务至注册中心：client-1,service。如图所示：

CLIENT-1	n/a (1)	(1)	UP (1) - localhost:client-1:8340
SECURITY	n/a (1)	(1)	UP (1) - localhost:security:8333
SERVICE	n/a (1)	(1)	UP (1) - localhost:service:8336

service作为服务提供者，client-1作为服务调用者。

服务调用者代码为：

编写服务调用者的controller代码为：

```
@RestController
@RequestMapping("/client")
public class UserController {
    @Autowired
    private RestTemplate restTemplate;
    @GetMapping("/getUser")
    public List<User> getUser(){
        List<User> userList =
restTemplate.getForObject("http://localhost:8336/services/getUser",List.class);
        return userList;
    }
}
```

restTemplate.getForObject("<http://localhost:8336/services/getUser>",List.class);方法解析：

```
public <T> T getForObject(URI url, Class<T> responseType)
```

url: 服务提供者的接口地址

编写服务提供者的代码:

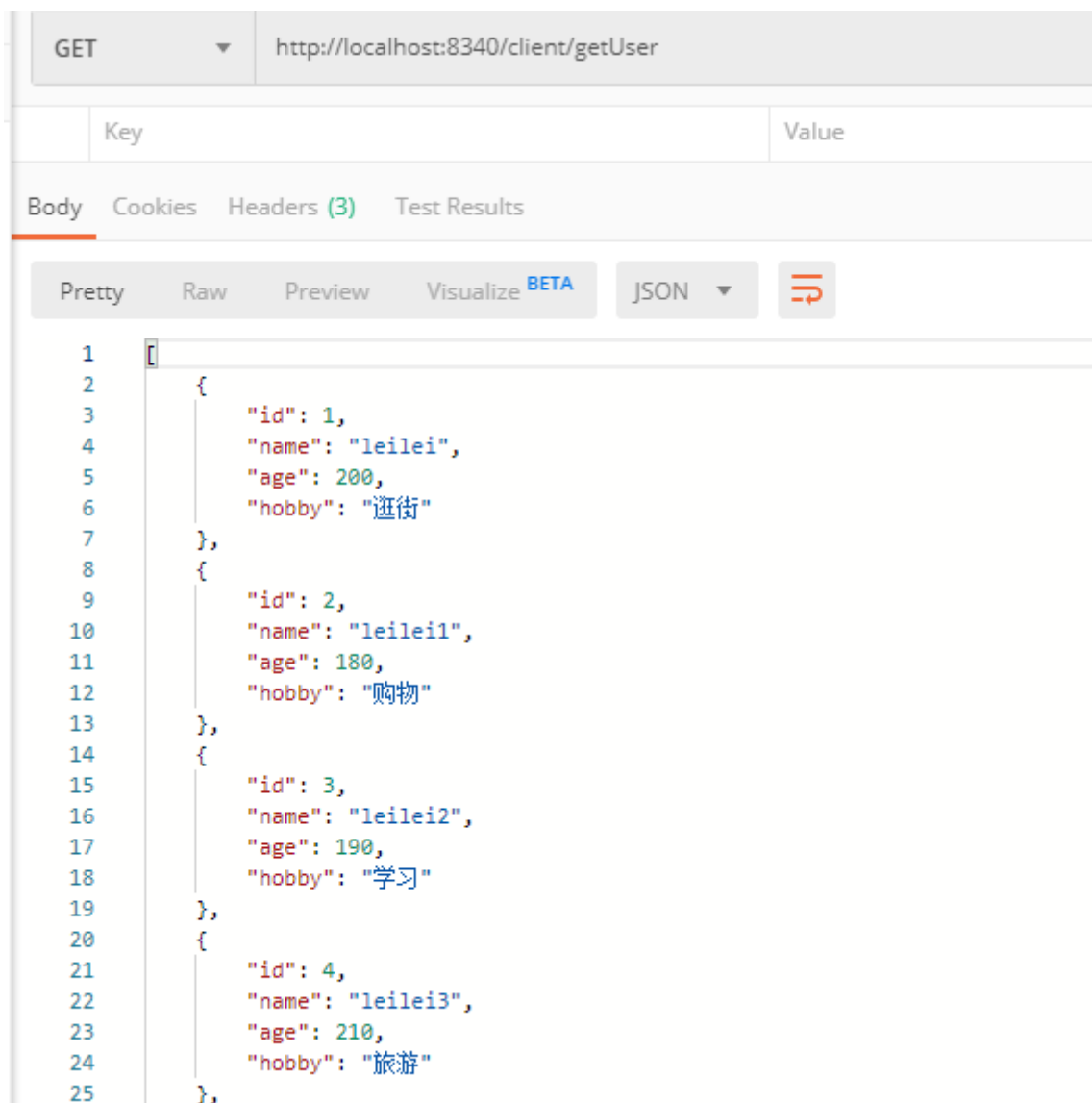
controller:

```
@Slf4j
@RestController
@RequestMapping("/services")
public class UserController {
    @Autowired
    private UserService userService;

    @RequestMapping("/getUser")
    public List<User> getUser() {
        return userService.getUser();
    }
}
```

。。。省略service与dao代码

用postman测试工具测试服务调用者接口: <http://localhost:8340/client/getUser>



通过服务调用者调用服务提供者接口方法成功,并返回数据。

1.6 eureka的自我保护

默认情况下，如果 Eureka Server 在一定的 90s 内没有接收到某个微服务实例的心跳，会注销该实例。但是在微服务架构下服务之间通常都是跨进程调用，网络通信往往会面临着各种问题，比如微服务状态正常，网络分区故障，导致此实例被注销。固定时间内大量实例被注销，可能会严重威胁整个微服务架构的可用性。为了解决这个问题，Eureka 开发了自我保护机制，**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 即会进入自我保护机制。**

触发自我保护机制时会有如下提示：

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

DS Replicas eureka的自我保护模式开启

localhost

Instances currently registered with Eureka

服务名为client-1的服务已注册

Application	AMIs	Availability Zones	Status
CLIENT-1	n/a (1)	(1)	UP (1) - localhost:client-1:8340

Eureka Server 进入自我保护机制，会出现以下几种情况：

- (1) Eureka 不再从注册列表中移除因为长时间没收到心跳而应该过期的服务
- (2) Eureka 仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)
- (3) 当网络稳定时，当前实例新的注册信息会被同步到其它节点中

如果想关闭自我保护模式则需配置：

```
#是否开启自我保护  
eureka.server.enableSelfPreservation=false
```

1.7 服务剔除与恢复

服务剔除：

DELETE http://localhost:8333/eureka/apps/client-1 localhost:client-1:8340

delete请求 eureka server地址 微服务名称 实例Id

KEY	VALUE	DESCRIPTION
Key	Value	Description

Status: 200 OK Time: 9ms

服务恢复：

PUT http://localhost:8333/eureka/apps/client-1/localhost:client-1:8340/status?value=UP

服务恢复时修改如上两个位置即可。

1.8 eureka的健康检查

1.8.1 查看服务信息

```
@GetMapping("/instance")  
public List<ServiceInstance> showInfo(){  
    return this.discoveryClient.getInstances("service");  
}
```


进入接口: <http://localhost:8340/client/instance>

```
{
  "host": "localhost",
  "port": 8336,
  "secure": false,
  "metadata": {
    "management.port": "8336",
    "jmx.port": "56483"
  },
  "uri": "http://localhost:8336",
  "instanceId": "localhost:service:8336",
  "serviceId": "SERVICE",
  "instanceInfo": {
    "instanceId": "localhost:service:8336",
    "app": "SERVICE",
    "appGroupName": null,
    "ipAddr": "192.168.186.1",
    "sid": "na",
    "homePageUrl": "http://localhost:8336/",
    "statusPageUrl": "http://localhost:8336/actuator/info",
    "healthCheckUrl": "http://localhost:8336/actuator/health",
    "secureHealthCheckUrl": null,
    "vipAddress": "service",
    "secureVipAddress": "service",
    "countryId": 1,
    "dataCenterInfo": {
      "@class": "com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo",
      "name": "MyOwn"
    },
    "hostName": "localhost",
    "status": "UP",
    "overriddenStatus": "UNKNOWN",
    "leaseInfo": {
```

可以查看服务的具体信息。

1.8.2 健康检查

配置属性:

```
management.endpoint.health.show-details=always
eureka.client.healthcheck.enabled=true
```

进入接口: <http://localhost:8340/actuator/health>

```

{
  "status": "UP",
  "components": {
    "discoveryComposite": {
      "status": "UP",
      "components": {
        "discoveryClient": {
          "status": "UP",
          "details": {
            "services": [
              "security",
              "service",
              "client-1"
            ]
          }
        },
        "eureka": {
          "description": "Remote status from Eureka server",
          "status": "UP",
          "details": {
            "applications": {
              "CLIENT-1": 1,
              "SECURITY": 1,
              "SERVICE": 1
            }
          }
        }
      }
    }
  }
}

```

可以查看服务的健康状态及注册表中的实例。

2 Ribbon

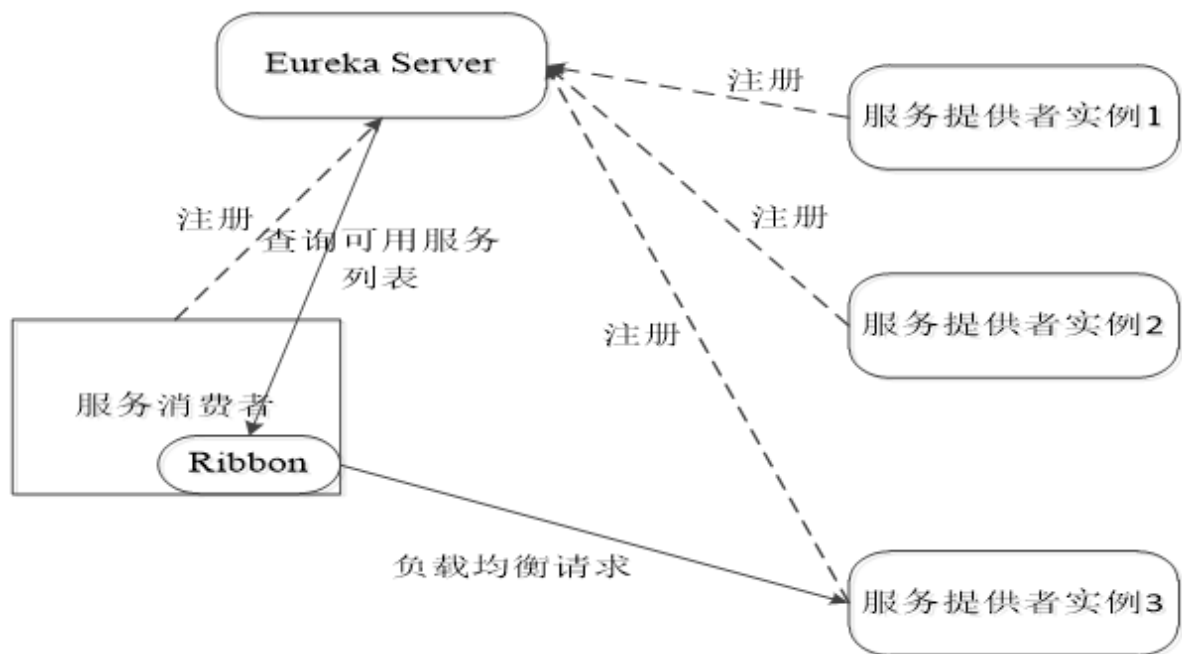
2.1 概念及框架

Spring Cloud Ribbon是一个基于HTTP和TCP的客户端负载均衡工具，它基于Netflix Ribbon实现。Spring Cloud Ribbon虽然只是一个工具类框架，它不像服务注册中心、配置中心、API网关那样需要独立部署，但是它几乎存在于每一个Spring Cloud构建的微服务和基础设施中。因为微服务间的调用，API网关的请求转发等内容，实际上都是通过Ribbon来实现的。

通过Spring Cloud Ribbon的封装，在微服务架构中使用客户端负载均衡调用非常简单，只需要如下两步：

- 服务提供者只需要启动多个服务实例并注册到一个注册中心或是多个相关联的服务注册中心。
- 服务消费者直接通过调用被@LoadBalanced注解修饰过的RestTemplate来实现面向服务的接口调用。

ribbon与eureka配合使用架构图：



2.2 服务调用者整合ribbon

(1)引入依赖

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

(2)修改配置类

为RestTemplate加上@LoadBalanced注解

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

(3) 修改controller代码

```
@GetMapping("/getUser")
public List<User> getUser(){
    List<User> userList =

    restTemplate.getForObject("http://service/services/getUser",List.class);
    return userList;
}
```

将微服务的地址改为:<http://service/services/getUser>,使用service代替原本的localhost: 8336 (原因后面讲)

(4) 测试

CLIENT-1	n/a (1)	(1)	UP (1) - localhost:client-1:8340
SECURITY	n/a (1)	(1)	UP (1) - localhost:security:8333
SERVICE	n/a (2)	(2)	UP (2) - localhost:service:8337 , localhost:service:8336

多个服务提供者实例

启动2个或多个服务提供者实例，启动此ribbon实例，进入接口可以发现每一次的服务调用都以轮询的方式调用服务提供者实例。说明负载均衡策略成功。

2.3 自定义ribbon配置

2.3.1 使用java代码自定义配置

(1) 定义负载均衡策略为随机分配

```
@Bean
public IRule ribbonRule(){
    return new RandomRule();
}
```

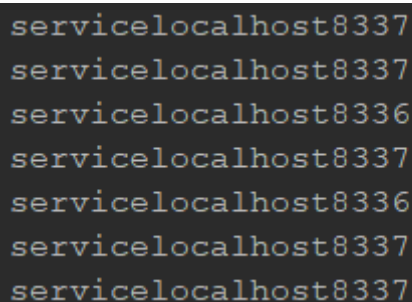
(2) 在启动类中加载此配置文件

```
@RibbonClient(name = "service",configuration = RestTemplateConfig.class)
public class EurekaClientApplicationclientApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaClientApplicationclientApplication.class,
args);
    }
}
```

(3) 定义接口，检查调用的是哪个接口

```
@GetMapping("/userInstance")
public void userInstance(){
    ServiceInstance serviceInstance = loadBalancerClient.choose("service");
    System.out.println(serviceInstance.getServiceId()+serviceInstance.getHost()+serviceInstance.getPort());
}
```

(3) 在postman中多次调用服务调用者接口<http://localhost:8340/client/getUser>，及<http://localhost:8340/client/userInstance>接口可以看出：



```
servicelocalhost8337
servicelocalhost8337
servicelocalhost8336
servicelocalhost8337
servicelocalhost8336
servicelocalhost8337
servicelocalhost8337
```

从图中可以看出调用服务提供者已变成随机调用。

2.3.2 使用属性自定义配置

上节使用代码自定义的配置在配置文件中使属性配置便可做到，这种方式也更方便。配置为：

```
service.ribbon.NFLoadBalancerRuleClassName=com.netflix.loadbalancer.RandomRule
```

2.4 脱离eureka使用ribbon

当许多微服务没有注册到eureka server中，也不是用springcloud开发的，怎样使用ribbon负载均衡？ 实战一下吧~

(1) 移除依赖

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

(2) 去除启动类上的@EnableEurekaClient 注解

(3) 修改配置文件

```
service.ribbon.listOfServers=localhost:8336,localhost:8337
```

2.5 负载均衡原理

本节只讨论为何RestTemplate加上@LoadBalanced注解便有了负载均衡的能力。

进入@LoadBalanced注解发现：

```
/**
 * Annotation to mark a RestTemplate bean to be configured to use a
 * LoadBalancerClient.
 * @author Spencer Gibb
 */
```

从@LoadBalanced注解源码的注释中，可以看出该注解用来给RestTemplate标记，以使用负载均衡的客户端（LoadBalancerClient）来配置它

然后搜索LoadBalancerClient可以看到：

```
package org.springframework.cloud.client.loadbalancer;

import org.springframework.cloud.client.ServiceInstance;

import java.io.IOException;
import java.net.URI;

/**
 * Represents a client side load balancer
 * @author Spencer Gibb
 */
public interface LoadBalancerClient extends ServiceInstanceChooser {

    /**
     * execute 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
     * @return the result of the LoadBalancerRequest callback on the selected
     * ServiceInstance
     */
    <T> T execute(String serviceId, LoadBalancerRequest<T> request) throws
    IOException;
```

```

/**
 * execute request using a ServiceInstance from the LoadBalancer for the
specified
 * service
 * @param serviceId the service id to look up the LoadBalancer
 * @param serviceInstance the service to execute the request to
 * @param request allows implementations to execute pre and post actions
such as
 * incrementing metrics
 * @return the result of the LoadBalancerRequest callback on the selected
 * ServiceInstance
 */
<T> T execute(String serviceId, ServiceInstance serviceInstance,
LoadBalancerRequest<T> request) throws IOException;

/**
 * Create a proper URI with a real host and port for systems to utilize.
 * Some systems use a URI with the logical service name as the host,
 * such as http://myservice/path/to/service. This will replace the
 * service name with the host:port from the ServiceInstance.
 * @param instance
 * @param original a URI with the host as a logical service name
 * @return a reconstructed URI
 */
URI reconstructURI(ServiceInstance instance, URI original);
}

```

此接口完成的功能：

ServiceInstance choose(String serviceId): 根据传入的服务名serviceId，从负载均衡器中挑选一个对应服务的实例。

T execute(String serviceId, LoadBalancerRequest request) throws IOException: 使用从负载均衡器中挑选出的服务实例来执行请求内容。

URI reconstructURI(ServiceInstance instance, URI original): 为系统构建一个合适的“host:port”形式的URI。在分布式系统中，我们使用逻辑上的服务名称作为host来构建URI（替代服务实例的“host:port”形式）进行请求，比如：<http://myservice/path/to/service>。在该操作的定义中，前者ServiceInstance对象是带有host和port的具体服务实例，而后者URI对象则是使用逻辑服务名定义为host的URI，而返回的URI内容则是通过ServiceInstance的服务实例详情拼接出的具体“host:port”形式的请求地址。

找到LoadBalancerAutoConfiguration为实现客户端负载均衡器的自动化配置类：

```

@Configuration
@ConditionalOnClass(RestTemplate.class)
@ConditionalOnBean(LoadBalancerClient.class)
@EnableConfigurationProperties(LoadBalancerRetryProperties.class)
public class LoadBalancerAutoConfiguration {

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

    @Bean
    public SmartInitializingSingleton loadBalancedRestTemplateInitializer(
        final List<RestTemplateCustomizer> customizers) {
        return new SmartInitializingSingleton() {

```

```

        @Override
        public void afterSingletonsInstantiated() {
            for (RestTemplate restTemplate :
LoadBalancerAutoConfiguration.this.restTemplates) {
                for (RestTemplateCustomizer customizer : customizers) {
                    customizer.customize(restTemplate);
                }
            }
        }
    };
}

@Autowired(required = false)
private List<LoadBalancerRequestTransformer> transformers =
Collections.emptyList();

@Bean
@ConditionalOnMissingBean
public LoadBalancerRequestFactory loadBalancerRequestFactory(
    LoadBalancerClient loadBalancerClient) {
    return new LoadBalancerRequestFactory(loadBalancerClient, transformers);
}

@Configuration

@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 new RestTemplateCustomizer() {
            @Override
            public void customize(RestTemplate restTemplate) {
                List<ClientHttpRequestInterceptor> list = new ArrayList<>(
                    restTemplate.getInterceptors());
                list.add(loadBalancerInterceptor);
                restTemplate.setInterceptors(list);
            }
        };
    }
}

@Configuration
@ConditionalOnClass(RetryTemplate.class)
public static class RetryAutoConfiguration {
    @Bean
    @ConditionalOnMissingBean
    public RetryTemplate retryTemplate() {
        RetryTemplate template = new RetryTemplate();
    }
}

```

```

        template.setThrowLastExceptionOnExhausted(true);
        return template;
    }

    @Bean
    @ConditionalOnMissingBean
    public LoadBalancedRetryPolicyFactory loadBalancedRetryPolicyFactory() {
        return new LoadBalancedRetryPolicyFactory.NeverRetryFactory();
    }

    @Bean
    @ConditionalOnMissingBean
    public LoadBalancedBackOffPolicyFactory
loadBalancedBackOffPolicyFactory() {
        return new
LoadBalancedBackOffPolicyFactory.NoBackOffPolicyFactory();
    }

    @Bean
    @ConditionalOnMissingBean
    public LoadBalancedRetryListenerFactory
loadBalancedRetryListenerFactory() {
        return new
LoadBalancedRetryListenerFactory.DefaultRetryListenerFactory();
    }
}

@Configuration
@ConditionalOnClass(RetryTemplate.class)
public static class RetryInterceptorAutoConfiguration {
    @Bean
    @ConditionalOnMissingBean
    public RetryLoadBalancerInterceptor ribbonInterceptor(
        LoadBalancerClient loadBalancerClient,
LoadBalancerRetryProperties properties,
        LoadBalancedRetryPolicyFactory lbRetryPolicyFactory,
        LoadBalancerRequestFactory requestFactory,
        LoadBalancedBackOffPolicyFactory backOffPolicyFactory,
        LoadBalancedRetryListenerFactory retryListenerFactory) {
        return new RetryLoadBalancerInterceptor(loadBalancerClient,
properties,
        lbRetryPolicyFactory, requestFactory, backOffPolicyFactory,
retryListenerFactory);
    }

    @Bean
    @ConditionalOnMissingBean
    public RestTemplateCustomizer restTemplateCustomizer(
        final RetryLoadBalancerInterceptor loadBalancerInterceptor) {
        return new RestTemplateCustomizer() {
            @Override
            public void customize(RestTemplate restTemplate) {
                List<ClientHttpRequestInterceptor> list = new ArrayList<>(
                    restTemplate.getInterceptors());
                list.add(loadBalancerInterceptor);
                restTemplate.setInterceptors(list);
            }
        };
    }
}

```



```
}  
}
```

在该自动化配置类中，主要做了下面三件事：

创建了一个LoadBalancerInterceptor的Bean，用于实现对客户端发起请求时进行拦截，以实现客户端负载均衡。

创建了一个RestTemplateCustomizer的Bean，用于给RestTemplate增加LoadBalancerInterceptor拦截器。

维护了一个被@LoadBalanced注解修饰的RestTemplate对象列表，并在这里进行初始化，通过调用RestTemplateCustomizer的实例来给需要客户端负载均衡的RestTemplate增加LoadBalancerInterceptor拦截器

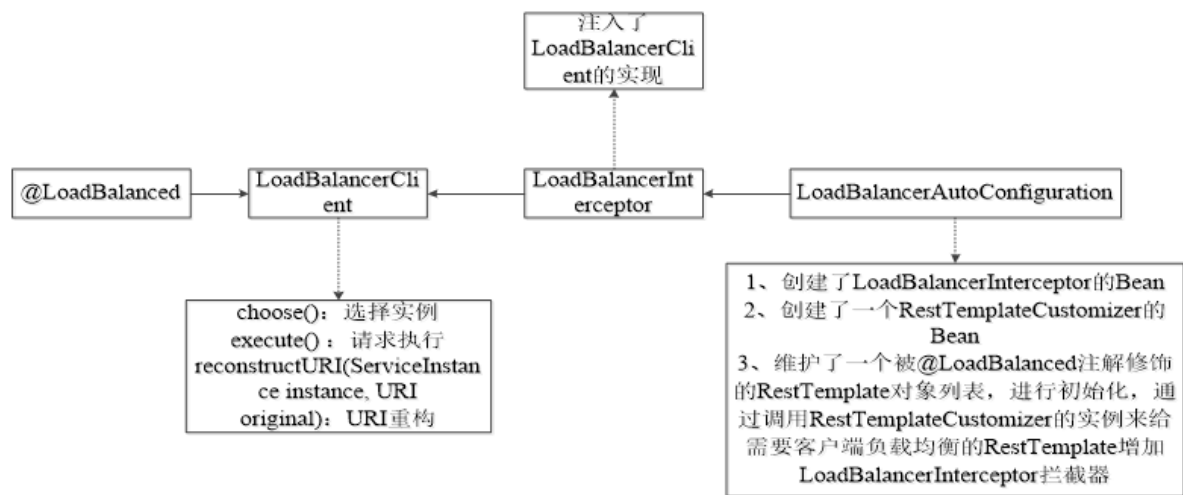
最后了解LoadBalancerInterceptor拦截器是如何将一个普通的RestTemplate变成客户端负载均衡的：

```
public class LoadBalancerInterceptor implements ClientHttpRequestInterceptor {  
  
    private LoadBalancerClient loadBalancer;  
    private LoadBalancerRequestFactory requestFactory;  
  
    public LoadBalancerInterceptor(LoadBalancerClient loadBalancer,  
    LoadBalancerRequestFactory requestFactory) {  
        this.loadBalancer = loadBalancer;  
        this.requestFactory = requestFactory;  
    }  
  
    public LoadBalancerInterceptor(LoadBalancerClient loadBalancer) {  
        // for backwards compatibility  
        this(loadBalancer, new LoadBalancerRequestFactory(loadBalancer));  
    }  
  
    @Override  
    public ClientHttpResponse intercept(final HttpRequest request, final byte[]  
    body,  
        final ClientHttpRequestExecution execution) throws IOException {  
        final URI originalUri = request.getURI();  
        String serviceName = originalUri.getHost();  
        Assert.state(serviceName != null, "Request URI does not contain a valid  
hostname: " + originalUri);  
        return this.loadBalancer.execute(serviceName,  
        requestFactory.createRequest(request, body, execution));  
    }  
}
```

我们看到在拦截器中注入了LoadBalancerClient的实现。当一个被@LoadBalanced注解修饰的RestTemplate对象向外发起HTTP请求时，会被LoadBalancerInterceptor类的intercept函数所拦截。由于我们在使用RestTemplate时候采用了服务名作为host，所以直接从HttpRequest的URI对象中通过getHost()就可以拿到服务名，然后调用execute函数去根据服务名来选择实例并发起实际的请求。

LoadBalancerClient接口的具体实现类完成具体负载均衡逻辑就是另一个故事啦~

此次总结如下图：



部分负载均衡原理总结图

3 feign

目的：解决url复杂的问题。

3.1 服务提供者整合feign

(1) 添加依赖

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
  <version>2.1.0.RELEASE</version>
</dependency>
```

(2) 创建一个接口

```
//添加@FeignClient注解指定服务提供者id，用于创建负载均衡器
@FeignClient(name = "service")
public interface UserFeignClient {
    @RequestMapping(value="/services/getUser",method=RequestMethod.GET)
    List<User> getUser();
}
```

(3) 编写controller代码，让其调用feign接口

```
@RequestMapping("/client")
public class UserController {
    @Autowired
    private UserFeignClient userFeignClient;

    @GetMapping("/getUser")
    public List<User> getUser(){
        return this.userFeignClient.getUser();
    }
}
```

(4) 为启动类添加注解,并扫描接口

```

@EnableFeignClients(basePackages =
    "com.springcloud.eurekaclientapplicationclientfeign.mapper")
public class EurekaClientApplicationClientFeignApplication {
    public static void main(String[] args) {

        SpringApplication.run(EurekaClientApplicationClientFeignApplication.class,
            args);
    }
}

```

(5) 测试

启动多个服务提供者实例，启动feign调用者实例，可以看出服务调用成功并实现了负载均衡。

3.2 自定义feign配置

(1) 创建feign的配置类

```

@Configuration
public class FeignConfiguration {
    @Bean
    public Contract feignContract(){
        return new Contract.Default();
    }
}

```

(2) 修改接口

```

//指定自定义配置类
@EnableFeignClients(name = "service",configuration=FeignConfiguration.class)
//使用feign自己的注解
public interface UserFeignClient {
    @RequestMapping(GET/services/getUser)
    List<User> getUser();
}

```

3.3 feign对压缩的支持

通过属性配置启用feign的压缩功能。

```

feign.compression.request.enabled=true
feign.compression.request.mime-types=text/xml,application/xml,application/json
feign.compression.request.min-request-size=2048

```

3.4 feign的日志

(1) 编写配置类

```

@Configuration
public class FeignLogConfiguration {
    @Bean
    public Logger.Level feignLogLevel(){
        return Logger.Level.FULL;
    }
}

```

(2) 修改feign的接口，指定配置类

```
@FeignClient(name = "service", configuration = FeignLogConfiguration.class)
public interface UserFeignClient {
    @GetMapping("/services/getUser")
    List<User> getUser();
}
```

(3) 增加配置信息

```
#将feign接口的日志级别设置为debug，因为feign的Logger.Level只对DEBUG做出响应
logging.level.com.springcloud.eurekaclientapplicationclientfeign.mapper.UserFeignClient=debug
```

(4) 日志信息如下

```

2019-11-07 15:11:37.197 DEBUG 2928 --- [strix-service-1]
c.s.e.mapper.UserFeignClient      : [UserFeignClient#getUser] <---
HTTP/1.1 200 (791ms)
2019-11-07 15:11:37.197 DEBUG 2928 --- [strix-service-1]
c.s.e.mapper.UserFeignClient      : [UserFeignClient#getUser] content-
type: application/json;charset=UTF-8
2019-11-07 15:11:37.197 DEBUG 2928 --- [strix-service-1]
c.s.e.mapper.UserFeignClient      : [UserFeignClient#getUser] date: Thu,
07 Nov 2019 07:11:37 GMT
2019-11-07 15:11:37.197 DEBUG 2928 --- [strix-service-1]
c.s.e.mapper.UserFeignClient      : [UserFeignClient#getUser] transfer-
encoding: chunked
2019-11-07 15:11:37.197 DEBUG 2928 --- [strix-service-1]
c.s.e.mapper.UserFeignClient      : [UserFeignClient#getUser]
2019-11-07 15:11:37.220 DEBUG 2928 --- [strix-service-1]
c.s.e.mapper.UserFeignClient      : [UserFeignClient#getUser]
[{"id":1,"name":"leilei","age":200,"hobby":"逛街"},
{"id":2,"name":"leilei1","age":180,"hobby":"购物"},
{"id":3,"name":"leilei2","age":190,"hobby":"学习"},
{"id":4,"name":"leilei3","age":210,"hobby":"旅游"},
{"id":5,"name":null,"age":null,"hobby":null},
{"id":6,"name":null,"age":null,"hobby":null},
{"id":7,"name":null,"age":null,"hobby":null},
{"id":8,"name":null,"age":null,"hobby":null},
{"id":9,"name":null,"age":null,"hobby":null},
{"id":10,"name":null,"age":null,"hobby":null},
{"id":11,"name":"yanzu","age":200,"hobby":"shuaishuai"},{"id":12,"name":"彭于
晏","age":200,"hobby":"健身"},{"id":13,"name":"彭于晏","age":200,"hobby":"健身"},
{"id":14,"name":"胡歌","age":190,"hobby":"拍戏"},
{"id":15,"name":null,"age":null,"hobby":null},
{"id":16,"name":null,"age":null,"hobby":null},
{"id":17,"name":null,"age":null,"hobby":null},
{"id":18,"name":null,"age":null,"hobby":null},{"id":19,"name":"鹿
晗","age":190,"hobby":"唱歌"},{"id":20,"name":null,"age":null,"hobby":null}]
2019-11-07 15:11:37.220 DEBUG 2928 --- [strix-service-1]
c.s.e.mapper.UserFeignClient      : [UserFeignClient#getUser] <--- END
HTTP (987-byte body)
2019-11-07 15:11:37.637 INFO 2928 --- [erListUpdater-0]
c.netflix.config.ChainedDynamicProperty : Flipping property:
service.ribbon.ActiveConnectionsLimit to use NEXT property:
niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647

```

3.5 使用feign构造多参数请求

(1) GET请求的多参数写法

```

@RequestMapping(value="/services/getUser",method=RequestMethod.GET)
List<User> getUser(@RequestParam("id") Long id,@RequestParam("userName") String
userName)

```

url中有几个参数，feign接口的方法中就有几个参数。

(2) POST请求的多参数

服务提供者的controller:

```
@PostMapping("/addUser")
public int addUser(@RequestBody User user) {
    log.info("运行到这里了没?????");
    log.info(user.getName(),user.getAge(),user.getHobby());
    return userService.addUser(user);
}
```

服务调用者的方法应该这样写：

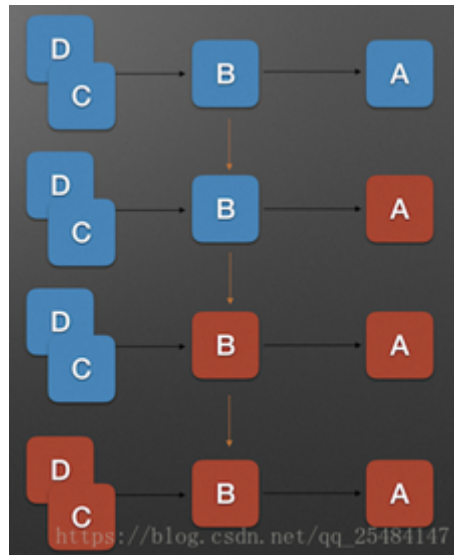
```
@RequestMapping(value = "/services/addUser", method = RequestMethod.POST)
int addUser(@RequestBody User user);
```

4 Hystrix

Hystrix作用是实现微服务的容错。

4.1 雪崩效应

服务雪崩效应是一种因 **服务提供者** 的不可用导致 **服务调用者** 的不可用,并将不可用 **逐渐放大** 的过程.如图示:



A为服务提供者, B为A的服务调用者, C和D是B的服务调用者. 当A的不可用,引起B的不可用,并将不可用逐渐放大C和D时, 服务雪崩就形成了。

服务雪崩效应形成的原因：

1. 服务提供者不可用

原因：硬件故障

程序Bug

缓存击穿

用户大量请求

硬件故障可能为硬件损坏造成的服务器主机宕机, 网络硬件故障造成的服务提供者的不可访问.

缓存击穿一般发生在缓存应用重启, 所有缓存被清空时,以及短时间内大量缓存失效时. 大量的缓存不命中, 使请求直击后端,造成服务提供者超负荷运行,引起服务不可用.

在秒杀和大促开始前,如果准备不充分,用户发起大量请求也会造成服务提供者的不可用.

2. 重试加大流量

原因：用户重试

代码逻辑重试

在服务提供者不可用后, 用户由于忍受不了界面上长时间的等待, 而不断刷新页面甚至提交表单. 服务调用端的会存在大量服务异常后的重试逻辑. 这些重试都会进一步加大请求流量.

3. 服务调用者不可用

原因: 同步等待造成的资源耗尽

当服务调用者使用 **同步调用** 时, 会产生大量的等待线程占用系统资源. 一旦线程资源被耗尽, 服务调用者提供的服务也将处于不可用状态, 于是服务雪崩效应产生了.

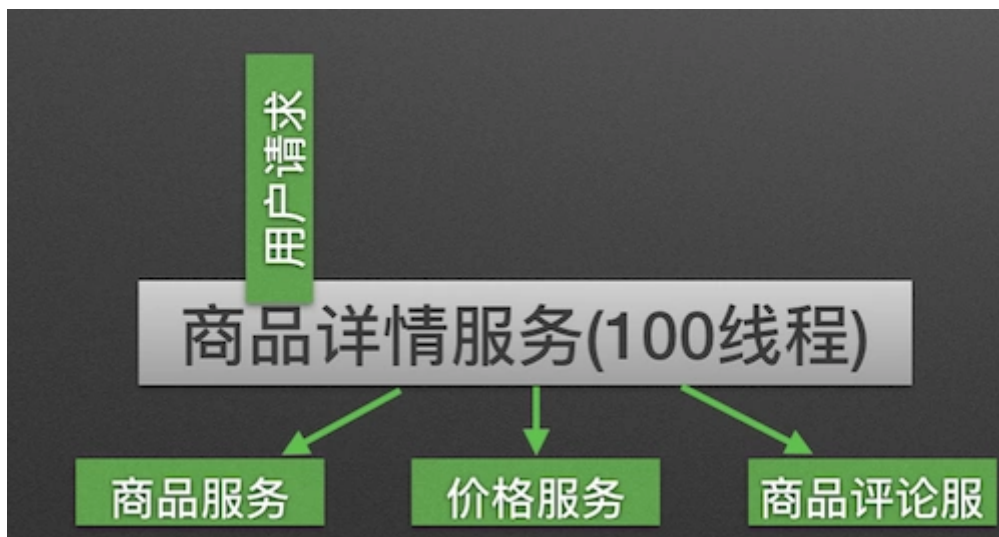
4.2 Hystrix实现容错

Hystrix设计原则:

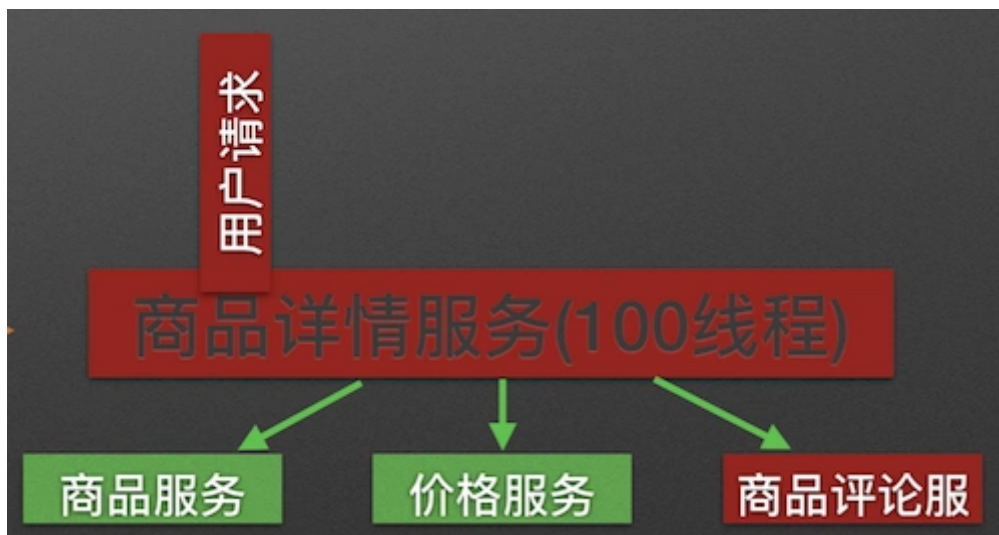
- 1、资源隔离
- 2、熔断器
- 3、命令模式

4.2.1 资源隔离

在一个高度服务化的系统中, 我们实现的一个业务逻辑通常会依赖多个服务, 比如: 商品详情展示服务会依赖商品服务, 价格服务, 商品评论服务. 如图所示:

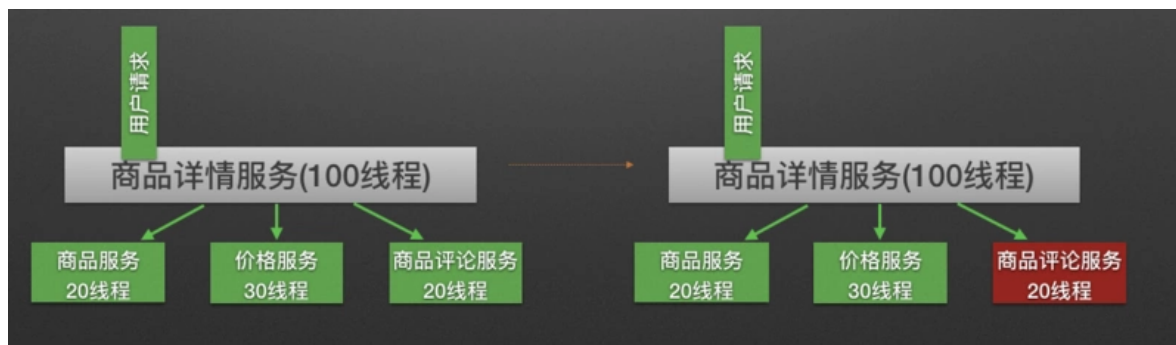


调用三个依赖服务会共享商品详情服务的线程池. 如果其中的商品评论服务不可用, 就会出现线程池里所有线程都因等待响应而被阻塞, 从而造成服务雪崩. 如图所示:



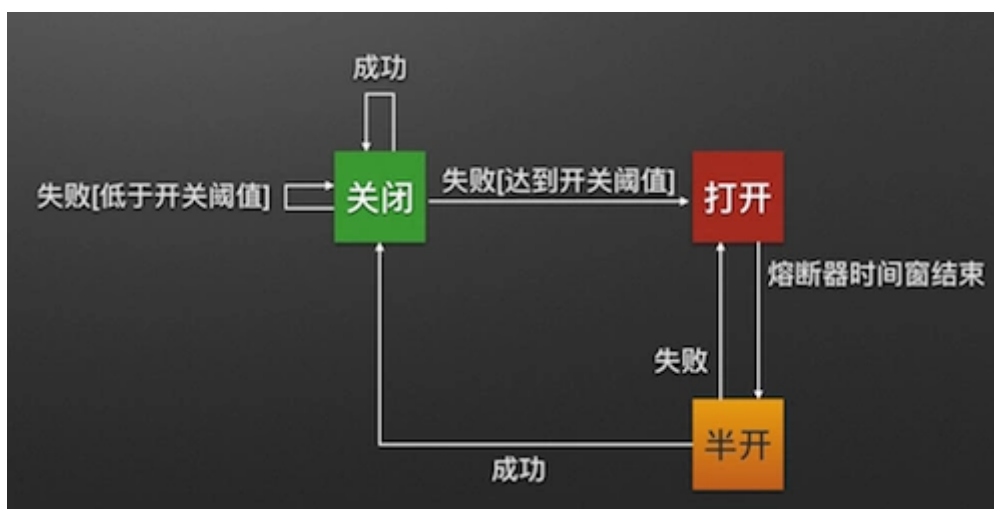
Hystrix通过将每个依赖服务分配独立的线程池进行资源隔离, 从而避免服务雪崩.

如下图所示, 当商品评论服务不可用时, 即使商品服务独立分配的20个线程全部处于同步等待状态, 也不会影响其他依赖服务的调用.



4.2.2 熔断器模式

熔断器模式定义了熔断器开关相互转换的逻辑



服务的健康状况 = 请求失败数 / 请求总数.

熔断器开关由关闭到打开的状态转换是通过当前服务健康状况和设定阈值比较决定的.

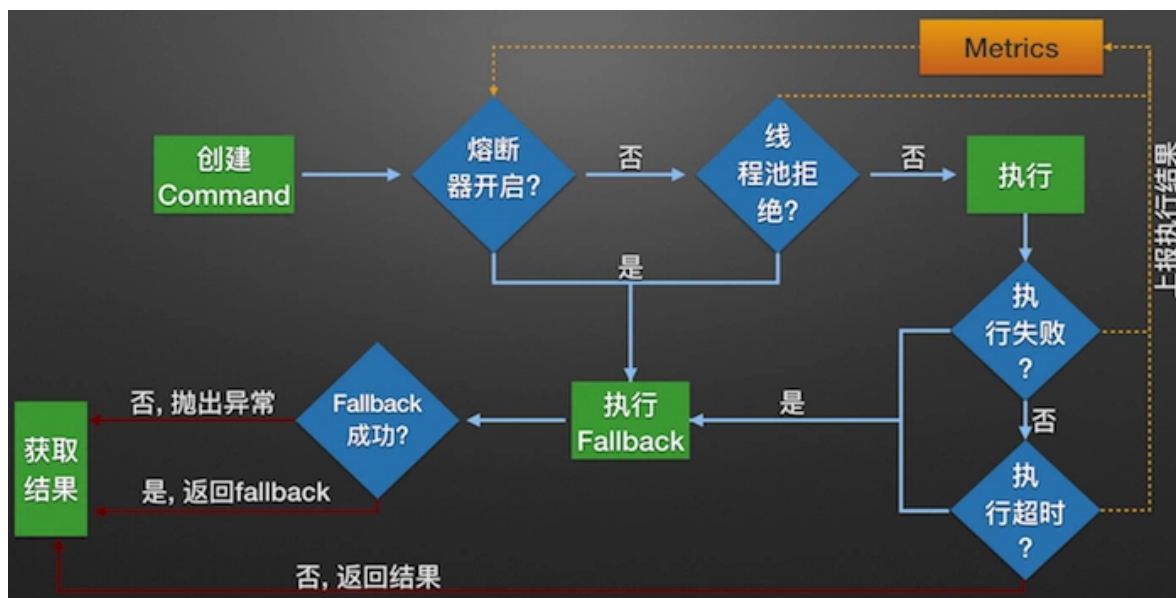
1. 当熔断器开关关闭时, 请求被允许通过熔断器. 如果当前健康状况高于设定阈值, 开关继续保持关闭. 如果当前健康状况低于设定阈值, 开关则切换为打开状态.
2. 当熔断器开关打开时, 请求被禁止通过.
3. 当熔断器开关处于打开状态, 经过一段时间后, 熔断器会自动进入半开状态, 这时熔断器只允许一个请求通过. 当该请求调用成功时, 熔断器恢复到关闭状态. 若该请求失败, 熔断器继续保持打开状态, 接下来的请求被禁止通过.

熔断器的开关能保证服务调用者在调用异常服务时, 快速返回结果, 避免大量的同步等待. 并且熔断器能在一段时间后继续侦测请求执行结果, 提供恢复服务调用的可能

4.2.3 命令模式

Hystrix使用命令模式(继承HystrixCommand类)来包裹具体的服务调用逻辑(run方法), 并在命令模式中添加了服务调用失败后的降级逻辑(getFallback).

Hystrix的内部处理逻辑:



1. 构建Hystrix的Command对象, 调用执行方法.
2. Hystrix检查当前服务的熔断器开关是否开启, 若开启, 则执行降级服务getFallback方法.
3. 若熔断器开关关闭, 则Hystrix检查当前服务的线程池是否能接收新的请求, 若超过线程池已满, 则执行降级服务getFallback方法.
4. 若线程池接受请求, 则Hystrix开始执行服务调用具体逻辑run方法.
5. 若服务执行失败, 则执行降级服务getFallback方法, 并将执行结果上报Metrics更新服务健康状况.
6. 若服务执行超时, 则执行降级服务getFallback方法, 并将执行结果上报Metrics更新服务健康状况.
7. 若服务执行成功, 返回正常结果.
8. 若服务降级方法getFallback执行成功, 则返回降级结果.
9. 若服务降级方法getFallback执行失败, 则抛出异常.

Hystrix的Metrics中保存了当前服务的健康状况, 包括服务调用总次数和服务调用失败次数等. 根据Metrics的计数, 熔断器从而能计算出当前服务的调用失败率, 用来和设定的阈值比较从而决定熔断器的状态切换逻辑

4.3 Hystrix具体实现

4.3.1 通用方式整合Hystrix

以Ribbon项目为例, 整合Hystrix

1、添加依赖

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

2、在启动类添加@EnableHystrix或@EnableCircuitBreaker, 为项目启用断路器支持

```
@EnableCircuitBreaker
public class HystrixRibbonClientApplication {
```

3、修改UserController, 为getUser()方法具备容错能力

```
// 使用@HystrixCommand指定回退方法为: getUserFallback
@HystrixCommand(fallbackMethod = "getUserFallback")
@GetMapping("/getUser")
public List<User> getUser(){
```

```

        List<User> userList =

restTemplate.getForObject("http://service/services/getUser",List.class);
        return userList;
    }

    public List<User> getUserFallback(){
        User user = new User();
        user.setName("默认用户");
        user.setId(-1);
        user.setAge(0);
        user.setHobby("无");
        List<User> list = new ArrayList<>();
        list.add(user);
        return list;
    }
}

```

4、测试

分别启动eureka服务者注册中心，服务提供者，以及此项目

访问：<http://localhost:8339/client/getUser> 可获得

```

{
  "id": 1,
  "name": "leilei",
  "age": 200,
  "hobby": "逛街"
},
{
  "id": 2,
  "name": "leilei1",
  "age": 180,
  "hobby": "购物"
},
{
  "id": 3,
  "name": "leilei2",
  "age": 190,
  "hobby": "学习"
},
{
  "id": 4,
  "name": "leilei3",

```

停止服务提供者服务，再次访问<http://localhost:8339/client/getUser>

```

1  [
2    {
3      "id": -1,
4      "name": "默认用户",
5      "age": 0,
6      "hobby": "无"
7    }
8  ]

```

说明当前服务提供者不可用时，进入了回退方法。

当请求失败、超时、或者断路器打开时都会进入回退方法，接下来探究观察断路器状态的方式。

4.3.2 Hystrix状态监控

1、引入依赖

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

2、在启动Eureka服务注册中心，服务提供者，以及此项目情况下，访问：<http://localhost:8339/actuator/health>查看断路器状态

```
{
  "hystrix": {
    "status": "UP"
  },
  "ping": {
    "status": "UP"
  },
  "rabbit": {
    "status": "UP",
    "details": {
      "version": "3.7.17"
    }
  }
}
```

可以看出此时断路器的状态是up，一切正常，断路器未打开。

3、在关闭服务调用者服务，此时进入回退方法，断路器的状态依然为：

```
{
  "hystrix": {
    "status": "UP"
  },
  "ping": {
    "status": "UP"
  },
  "rabbit": {
    "status": "UP",
    "details": {
      "version": "3.7.17"
    }
  }
}
```

可以发现，此时虽然已进入回退方法，但断路器依然未打开。原因是我们的请求失败率还未达到阈值（默认5秒20次失败，所以很显然看出执行回退逻辑并不代表断路器已打开。

4、持续快速访问<http://localhost:8339/client/getUser>，知道请求快速返回。

```
{
  "hystrix": {
    "status": "CIRCUIT_OPEN",
    "details": {
      "openCircuitBreakers": [
        "UserController::getUser"
      ]
    }
  }
}
```

从Hystrix的状态可以看出，断路器已处于打开模式。

断路器打开后，进入休眠模式（默认为5秒）在休眠状态时，是无法进行请求的。休眠结束时，断路器处于半开的状态，我们可以尝试调用一次服务，调用成功时，断路器关闭，状态变为up，调用失败时，断路器状态依然打开。

4.3.3 Feign使用Hystrix

1、将之前Feign项目中接口改变为：

```
//为此接口添加回退类
@FeignClient(name = "service", fallback = FeignClientFallback.class)
public interface UserFeignClient {
    @GetMapping("/services/getUser")
    List<User> getUser();
}
```

2、编写回退类:

```
@Component
public class FeignClientFallback implements UserFeignClient {
    @Override
    public List<User> getUser(){
        User user = new User();
        user.setName("默认用户");
        user.setId(-1);
        user.setAge(0);
        user.setHobby("无");
        List<User> list = new ArrayList<>();
        list.add(user);
        return list;
    }
}
```

3、通过fallbackFactory检查回退原因

```
@FeignClient(name = "service", configuration = FeignLogConfiguration.class)
public interface UserFeignClient {
    @GetMapping("/services/getUser")
    List<User> getUser();
}
```

```
@Component
@Slf4j
public class FeignClientFallbackFactory implements
    FallbackFactory<UserFeignClient> {

    @Override
    public UserFeignClient create(Throwable cause){
        return new UserFeignClient() {
            @Override
            public List<User> getUser() {
                log.info("回退原因是" + cause);
                User user = new User();
                user.setName("默认用户");
                user.setId(-1);
                user.setAge(0);
                user.setHobby("无");
                List<User> list = new ArrayList<>();
                list.add(user);
                return list;
            }
        };
    }
}
```

4.4 Hystrix的监控

1、在启动类中添加如下配置：

```
@Bean
public ServletRegistrationBean getServlet() {
    HystrixMetricsStreamServlet streamServlet = new
HystrixMetricsStreamServlet();
    ServletRegistrationBean registrationBean = new
ServletRegistrationBean(streamServlet);
    registrationBean.setLoadOnStartup(1);
    registrationBean.addUrlMappings("/actuator/hystrix.stream");
    registrationBean.setName("HystrixMetricsStreamServlet");
    return registrationBean;
}
```

2、访问路径：<http://localhost:8339/actuator/hystrix.stream>

当没有进行服务调用时，监控一直处于请求状态。当发起一次服务调用时，监控显示如下数据：

```
data:
{"type":"HystrixCommand","name":"getUser","group":"UserController","currentTime":1574220293976,"isCircuitBreakerOpen":false,"errorPercentage":0,"errorCount":0,"requestCount":0,"rollingCountBadRequests":0,"rollingCountCollapsedRequests":0,"rollingCountBait":0,"rollingCountExceptionsThrown":0,"rollingCountFailure":0,"rollingCountFallbackBait":0,"rollingCountFallbackFailure":0,"rollingCountFallbackMissing":0,"rollingCountFallbackRejection":0,"rollingCountFallbackSuccess":0,"rollingCountResponsesFromCache":0,"rollingCountSemaphoreRejected":0,"rollingCountShortCircuited":0,"rollingCountSuccess":0,"rollingCountThreadPoolRejected":0,"rollingCountTimeout":0,"currentConcurrentExecutionCount":0,"rollingMaxConcurrentExecutionCount":0,"latencyExecute_mean":184,"latencyExecute":
{"0":10,"25":10,"50":10,"75":362,"90":362,"95":362,"99":362,"99.5":362,"100":362},"latencyTotal_mean":186,"latencyTotal":
{"0":10,"25":10,"50":10,"75":362,"90":362,"95":362,"99":362,"99.5":362,"100":362},"propertyValue_circuitBreakerRequestVolumeThreshold":20,"propertyValue_circuitBreakerSleepWindowInMilliseconds":5000,"propertyValue_circuitBreakerErrorThresholdPercentage":50,"propertyValue_circuitBreakerForceOpen":false,"propertyValue_circuitBreakerForceClosed":false,"propertyValue_circuitBreakerEnabled":true,"propertyValue_executionIsolationStrategy":"THREAD","propertyValue_executionIsolationThreadTimeoutInMilliseconds":1000,"propertyValue_executionTimeoutInMilliseconds":1000,"propertyValue_executionIsolationThreadInterruptOnTimeout":true,"propertyValue_executionIsolationThreadPoolKeyOverride":null,"propertyValue_executionIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_fallbackIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"propertyValue_requestCacheEnabled":true,"propertyValue_requestLogEnabled":true,"reportingHosts":1,"threadPool":"UserController"}

data:
{"type":"HystrixThreadPool","name":"UserController","currentTime":1574220293976,"currentActiveCount":0,"currentCompletedTaskCount":2,"currentCorePoolSize":10,"currentLargestPoolSize":2,"currentMaximumPoolSize":10,"currentPoolSize":2,"currentQueueSize":0,"currentTaskCount":2,"rollingCountThreadsExecuted":0,"rollingMaxActiveThreads":0,"rollingCountCommandRejections":0,"propertyValue_queueSizeRejectionThreshold":5,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"reportingHosts":1}

ping:
```

内容非常全面，例如：HystrixCommand的名称、group名称、断路器状态、错误率、错误数等

4.5 使用Hystrix Dashboard可视化监控

1、创建maven项目，为项目添加依赖

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

2、编写启动类，在启动类上添加注解@EnableHystrixDashboard

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

3、在配置文件中，添加端口

```
server.port=8830
```

4、启动项目，查看可视化页面：



Hystrix Dashboard

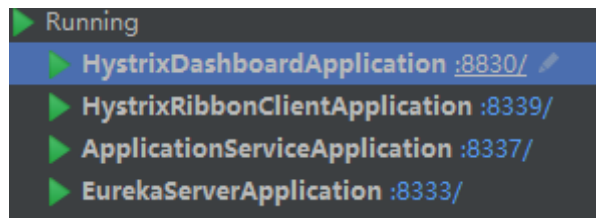
<https://hostname:port/turbine/turbine.stream>

Cluster via Turbine (default cluster): <https://turbine-hostname:port/turbine.stream>
Cluster via Turbine (custom cluster): [https://turbine-hostname:port/turbine.stream?cluster=\[clusterName\]](https://turbine-hostname:port/turbine.stream?cluster=[clusterName])
Single Hystrix App: <https://hystrix-app:port/actuator/hystrix.stream>

Delay: ms Title:

Monitor Stream

5、启动项目：



6、在上节Hystrix基础上监控，在URL初输入：<http://localhost:8339//actuator/hystrix.stream>

Hystrix Dashboard

<http://localhost:8339//actuator/hystrix.stream>

Cluster via Turbine (default cluster): <https://turbine-hostname:port/turbine.stream>
Cluster via Turbine (custom cluster): [https://turbine-hostname:port/turbine.stream?cluster=\[clusterName\]](https://turbine-hostname:port/turbine.stream?cluster=[clusterName])
Single Hystrix App: <https://hystrix-app:port/actuator/hystrix.stream>

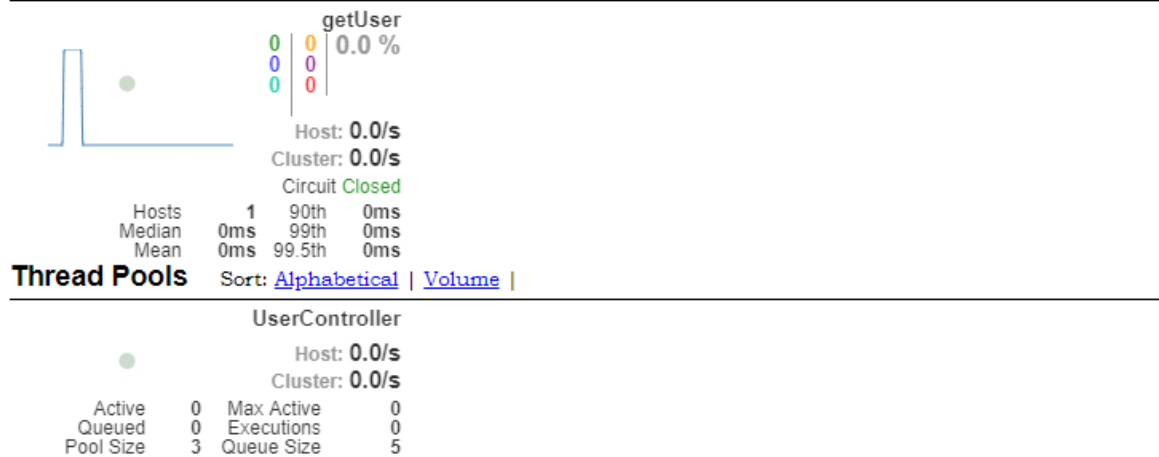
Delay: ms Title:

Monitor Stream

7、可以看到如下界面：

Hystrix Stream: hystrix

Circuit Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)



4.6 使用Turbine聚合监控数据

1、引入依赖

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

2、启动类添加注解

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

3、配置文件

```
spring.application.name=turbine
server.port=8831
eureka.client.service-url.defaultZone = http://localhost:8333/eureka/
#监控两个微服务
turbine.app-config=CLIENT,CLIENT-1
turbine.cluster-name-expression="default"
```

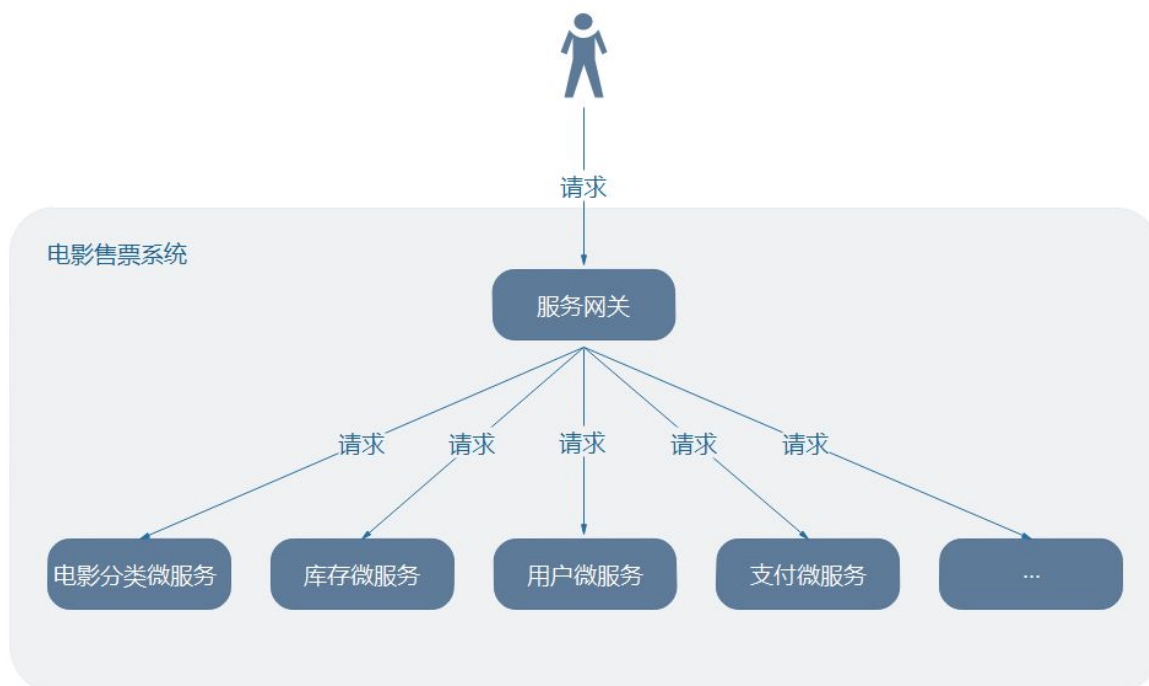
5 Zuul

5.1 为什么要使用微服务网关 (WHY)

在前面几节所学的微服务组件中，都是客户端直接与微服务通信，这种情况会产生一些问题：

- 1、客户端多次请求不同的微服务、增加客户端的复杂性
- 2、认证复杂，可能每个微服务都需要独立认证
- 3、难以重构，由于微服务直接与客户端进行通信，所以微服务在聚合或者拆分是复杂度较高

然而以上问题都可以借助微服务网关得到解决。来一张微服务网关架构图看看，一目了然：



从图中可以看出，网关是介于客户端与服务端的中间层，所有的客户端请求都会经过微服务网关，客户端只需跟网关进行交互，无需直接调用特定的微服务接口。

因此可以看出网关的直接优点：

- 1、易于监控
- 2、易于认证
- 3、减少客户端与各个微服务的交互次数

5.2 zuul简介 (WHAT)

zuul的核心是一系列过滤器（pre,router,post,error），这些过滤器完成的主要功能有：

- 1、动态路由
- 2、限流
- 3、身份认证与安全

5.3 zuul使用 (HOW)

5.3.1 编写一个微服务网关

- 1、添加依赖

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

- 2、启动类添加注解


```

@EnableZuulProxy
@EnableEurekaClient
public class ZuulApplication {
    public static void main(String[] args) {
        SpringApplication.run(ZuulApplication.class, args);
    }
}

```

3、配置文件

```

spring.application.name=zuul-fileupload
server.port=8888
eureka.client.service-url.defaultZone = http://localhost:8333/eureka/

```

4、测试路由规则



启动服务：



分别通过网关端口访问服务调用者及服务提供者两个微服务。

默认情况下，zuul会代理所有注册到eureka server上的微服务,并且路由规则如下：

<http://zuul> host : zuul port/微服务在eureka上的service-id/**

5.3.2 路由配置详解

推荐使用path与服务id的组合来进行配置：

```

#1、自定义路由
zuul.routes.service=/user/**

#2、忽略指定微服务
zuul.ignored-services=service

#3、忽略所有微服务，只路由指定微服务
zuul.ignored-services=*
zuul.routes.service=/user/**

#4、同时指定微服务的service_id和路径
zuul.routes.user-router.service-id=service
zuul.routes.user-router.path=/user/**

```

```
#5、同时指定path和url，并且不破坏zuul的Hystrix、ribbon特性
zuul.routes.user-router.path=/user/**
zuul.routes.user-router.service-id=service
ribbon.eureka.enabled=false
service.ribbon listOfServers=localhost:8337,localhost:8338
```

```
#6、忽略某些路径
zuul.ignored-patterns=/**/services/**
zuul.routes.service=/user/**
```

敏感header设置：

```
#敏感header设置、全局放行
zuul.sensitive-headers=Cookie,Set-Cookie,Authorization

#指定路由放行（局部放行）
zuul.routes.user-router.path=/user/**
zuul.routes.user-router.sensitive-headers=Cookie,Set-Cookie,Authorization
zuul.routes.user-router.url= http://localhost:8337
```

5.3.3 zuul的过滤器

过滤器类型：

pre：路由前调用，利用这种过滤器可以实现身份验证、在集群中选择请求的微服务，记录调试信息等。

routing：将请求路由到微服务，构建发送给服务的请求

post：路由之后执行，用来为响应添加标准的响应头等

error：在其他阶段发生错误时执行

编写一个zuul过滤器：

自定义过滤器：

```
package com.springcloud.zuulfliter.filter;

import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;
import lombok.extern.slf4j.Slf4j;
import org.apache.commons.lang.StringUtils;
import org.springframework.stereotype.Component;

import javax.servlet.http.HttpServletRequest;

/**
 * <pre>
 * 描述：自定义zuul过滤器
 * </pre>
 *
 * @author Nancy(Leilanjie)
 * @version 1.0.0
 * @date 2019/11/11 16:30
 */
@Slf4j
@Component
```

```

public class PreRequestLogFilter extends ZuulFilter {

    /**
     * 返回过滤器类型，对应于几种过滤器类型
     * @return
     */
    @Override
    public String filterType(){
        return "pre";
    }

    /**
     * 指定过滤器的执行顺序，不同类型的过滤器允许返回相同的数字,数字越小，执行优先级越高
     * @return
     */
    @Override
    public int filterOrder(){
        return 1;
    }

    /**
     * 判断该过滤器是否要执行
     * @return
     */
    @Override
    public boolean shouldFilter(){
        return true;
    }

    /**
     * 过滤器具体逻辑
     * @return
     */
    @Override
    public Object run(){
        RequestContext context = RequestContext.getCurrentContext();
        HttpServletRequest request = context.getRequest();
        String s = request.getHeader("hello");
        if(StringUtils.isBlank(s)){
            //header错误
            context.setResponseBody("缺少key值为hello的请求头");
            context.setResponseStatusCode(401);
            //拦截
            context.setSendZuulResponse(false);
        }
        return null;
    }
}

```

5.3.4 限流

1、添加限流依赖

```

<dependency>
  <groupId>com.marcosbarbero.cloud</groupId>
  <artifactId>spring-cloud-zuul-ratelimit</artifactId>
  <version>LATEST</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>

```

2、配置文件

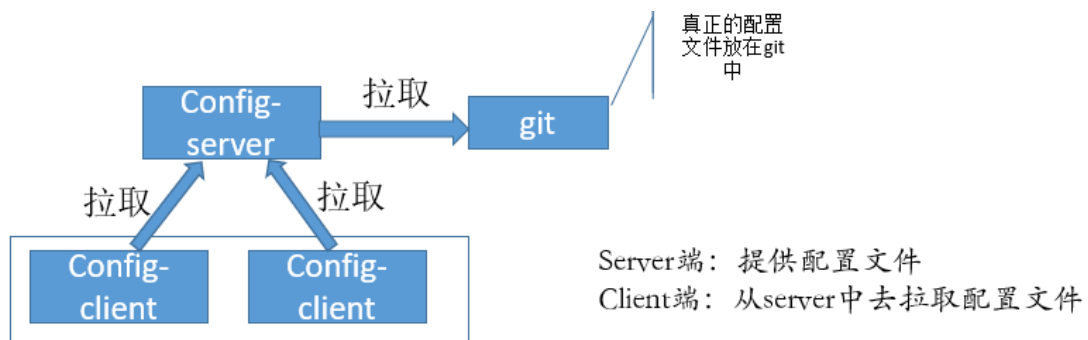
```

zuul.ratelimit.enabled=true
zuul.ratelimit.key-prefix=your-prefix
zuul.ratelimit.repository=REDIS
zuul.ratelimit.behind-proxy=true
zuul.ratelimit.add-response-headers=true
#针对全局的限制
#次数的限制
zuul.ratelimit.default-policy-list[0].limit=10
#时间的限制
zuul.ratelimit.default-policy-list[0].quota=1000
#60s
zuul.ratelimit.default-policy-list[0].refresh-interval=60
#针对某个ip地址
zuul.ratelimit.default-policy-list[0].type[0]=origin
#redis
spring.redis.host=127.0.0.1
spring.redis.port=6379

```

6 spring cloud config

6.1 spring cloud config 架构图



6.2 config实例

6.2.1 编写config server

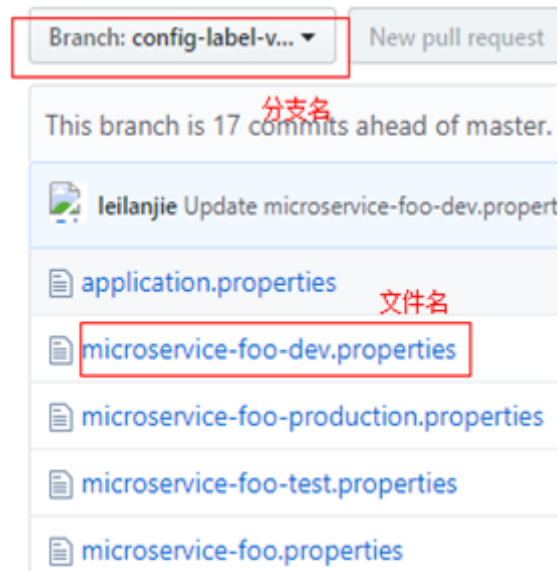
1、引入依赖

```

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

```

2、在github上新建几个文件，如图所示:确定分支



文件的内容为:

```
profile=dev-2.2
server.port=8082
eureka.client.service-url.defaultZone= http://localhost:8333/eureka/
```

3、配置server端配置文件

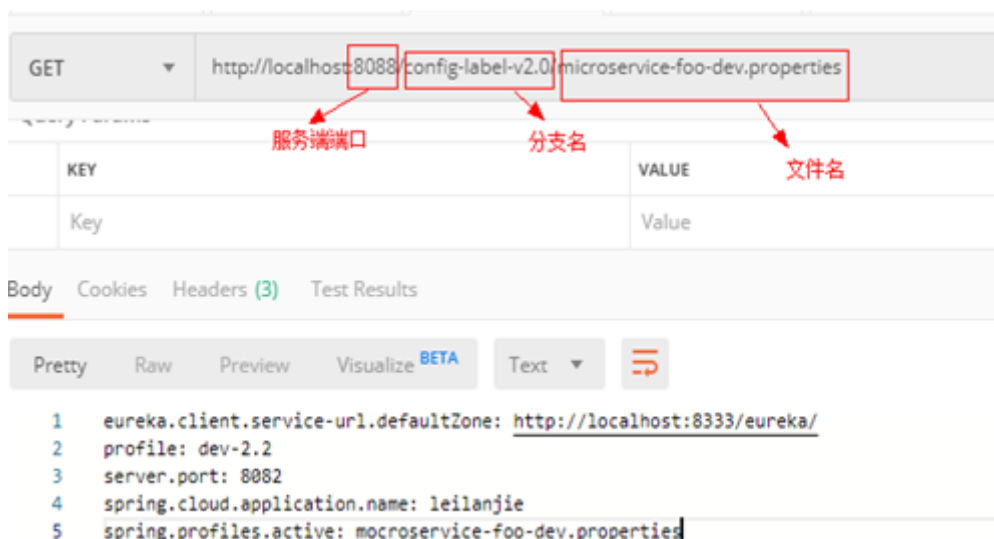
```
server.port=8088
spring.application.name=microservice-config-server
spring.cloud.config.server.git.uri=https://github.com/leilanjie/spring-cloud-config-repo
spring.cloud.config.server.git.username=leilanjie
spring.cloud.config.server.git.password=lj..0000
```

4、在启动类上添加注解

```
@SpringBootApplication
@EnableConfigServer
@EnableEurekaClient
public class ConfigApplication {

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

5、测试 (拉取配置)



6.2.2 编写config client

1、添加依赖

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
```

2、配置文件

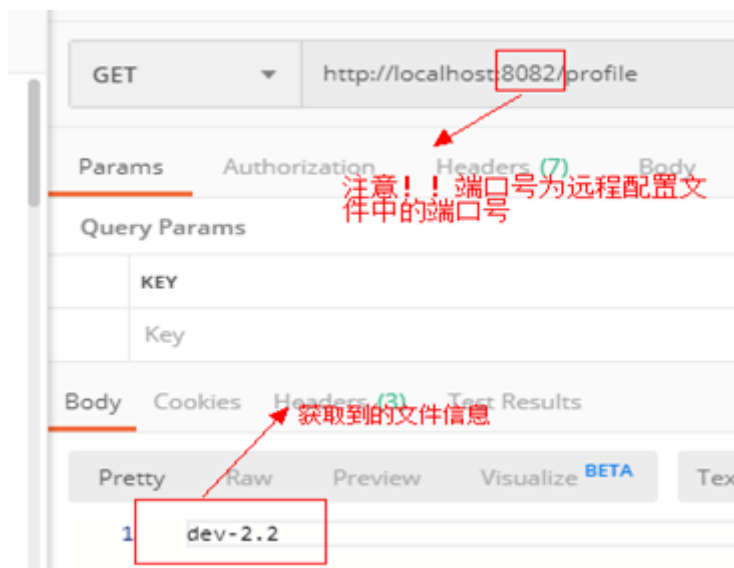
```
#对应远程配置文件中的{application}
spring.application.name=microservice-foo
#对应远程配置文件中的{profile}
spring.cloud.config.profile=dev
#分支
spring.cloud.config.label=config-label-v2.0
spring.cloud.config.uri=http://localhost:8088
```

3、通过访问接口获取配置文件

```
@Value("${profile}")
private String profile;

// 获取配置文件
@GetMapping("/profile")
public String Hello(){
    return this.profile;
}
```

4、postman测试配置文件拉取情况



6.3 手动刷新

远程配置文件发生改变时，config server可知道发生修改，但client却需要刷新才能知道。拉取配置文件保持一致性，步骤：

1、依赖中与要引入：

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

2、在controller添加注解

```
@RestController
@RefreshScope
public class ConfigClientController {

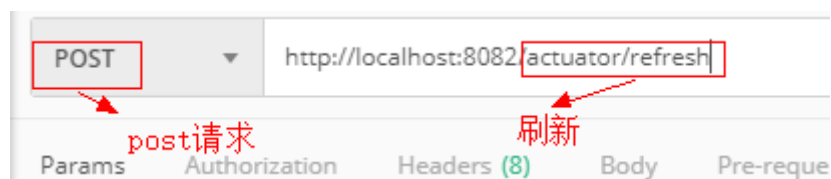
    @Value("${profile}")
    private String profile;
```

3、修改配置文件

在配置文件中添加：

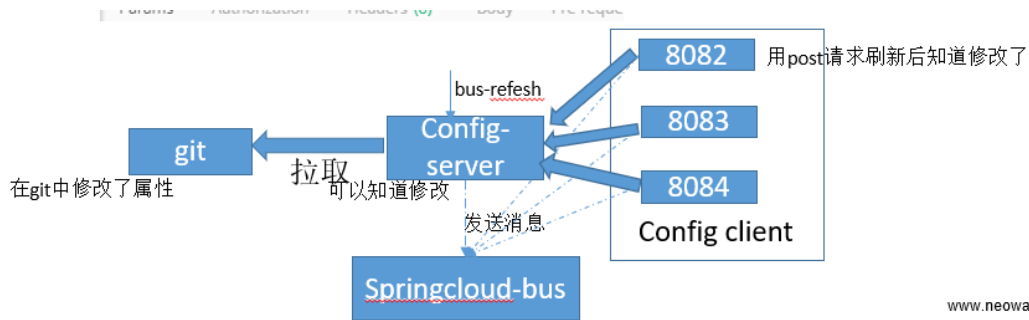
```
management.endpoint.health.show-details=always
management.endpoints.web.exposure.include=refresh,bus-refresh
```

4、手动刷新



6.4 Spring Cloud Bus

如图所示为Spring Cloud Bus的架构图：



实现自动刷新步骤:

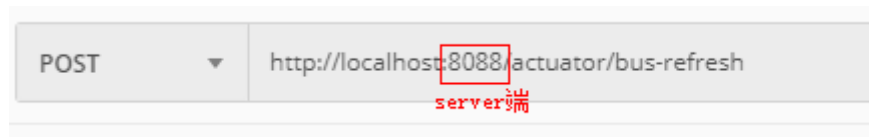
1、引入依赖

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

2、配置文件添加

```
#rabbitMQ
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

3、刷新配置



6.5 本地覆盖

spring.cloud.config.allowOverride=true

spring.cloud.config.overrideNone=true

Spring.cloud.config.overrideSystemProperties=true