# Spring Cloud

## Table of Contents

# 什么是微服务架构

Microservices 是系统架构上的一种设计风格，它的主旨是将一个原本独立的系统拆分成多个小型服务，这些小型服务都在各自独立的进程中运行，服务之间通过 HTTP 的 RESTful API 进行通信协作。被拆分成的每一个小型服务都围绕着系统中的某一项或一些耦合度较高的业务功能进行构建，并且每个服务都维护着自身的数据粗处、业务开发、自动化测试案例以及独立部署机制。由于又了轻量级的通信协作基础，所以这些微服务可以使用不同的语言来编写。

## 微服务架构的 9 大特性

服务组件化
按业务组织团队
做“产品”的态度
智能端点与哑管道
去中心化治理
去中心化管理数据
基础设施自动化
容错设计
演进式设计

# Spring Cloud 简介

Spring Cloud 是一个基于 Spring Boot 实现的微服务架构开发工具。它为微服务架构中涉及的配置管理、服务治理、断路器、智能路由、微代理、控制总线、全局锁、决策竞选、分布式会话和集群状态管理等操作提供了一种简单的开发方式。
Spring Cloud 包含了多个子项目：
Spring Cloud Config
Spring Cloud Netflix
Spring Cloud Bus
Spring Cloud Cluster
Spring Cloud Cloudfoundry
Spring Cloud Consul
Spring Cloud Stream
Spring Cloud AWS
Spring Cloud Security
Spring Cloud Sleuth
Spring Cloud ZooKeeper
Spring Cloud Starters
Spring Cloud CLI: 用于在 Groovy 中快速创建 Spring Cloud 应用的 Spring Boot CLI 插件

# Spring Cloud Eureka

## 什么是 Spring Cloud Eureka

Spring Cloud Eureka 是 Spring Cloud Netflix 微服务套件中的一部分，它基于 Netflix Eureka 做了二次封装，主要负责完成微服务架构中的服务治理功能。
服务治理是微服务架构中最为核心和基础的模块，它主要用来实现各个微服务实例的自动化注册与发现。

## Eureka 服务端

也称为服务注册中心。支持高可用配置。如果 Eureka 以集群模式部署，当集群中有分片出现故障时，那么 Eureka 就转入自我保护模式。

## Eureka 客户端

主要处理服务的注册与发现。客户端服务通过注解和参数配置的方式，嵌入在客户端应用程序的代码中，在应用程序运行时，Eureka 客户端向注册中心注册自身提供的服务并周期性的发送心跳来更新它的服务租约。同时它也能从服务端查询当前注册的服务信息并把它们缓存到本地并周期性的刷新服务状态。

# 构建服务注册中心

1. http://start.spring.io/ 使用 Gradle 或 Maven 建立 Spring Boot 2.0 版本工程，并勾选

   ☑ Eureka Server

   spring-cloud-netflix Eureka Server

   建议将 Artifact 设定为 eureka-server。之后将下载的项目导入 IDEA 或 Eclipse 等 IDE 中。

2. build.gradle 文件的内容如下：

```
buildscript {
	ext {
		springBootVersion = '2.0.0.RELEASE'
	}
	repositories {
		mavenCentral()
	}
	dependencies {
		classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
	}
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'
apply plugin: 'io.spring.dependency-management'

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8

repositories {
	mavenCentral()
	maven { url "https://repo.spring.io/milestone" }
}


ext {
	springCloudVersion = 'Finchley.M8'
}

dependencies {
	compile('org.springframework.cloud:spring-cloud-starter-netflix-eureka-server')
	testCompile('org.springframework.boot:spring-boot-starter-test')
}

dependencyManagement {
	imports {
		mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${springCloudVersion}"
	}
}
```

3. 在 EurekaServerApplication 上加上@EnableEurekaServer
4. 在 application.properties 中加入如下内容：

```
server.port=10000

eureka.instance.hostname=localhost
eureka.client.register-with-eureka=false
```

```
eureka.client.fetch-registry=false
eureka.client.service-url.defaultZone=http://${eureka.instance.hostname}:${server.port}/eureka/
```

5. 启动代码并访问 http://localhost:10000/

## 注册客户端到 eureka 服务器

1. 根据上面服务器的方式再次建立项目，但是需要在 Spring Boot 中多选择 Web 模块。建立项目名称叫做 hello-service
2. 在在 EurekaClientApplication 上加上@EnableEurekaClient
3. 建立 RESTful 风格的控制器

```
package com.example.eurekaclient1.controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HomeController {

    @RequestMapping
    public String home() {
        return "Hello, World";
    }
}
```

4. 在 application.properties 中加入如下内容：

```
spring.application.name=hello-service
eureka.client.service-url.defaultZone=http://localhost:10000/eureka
```

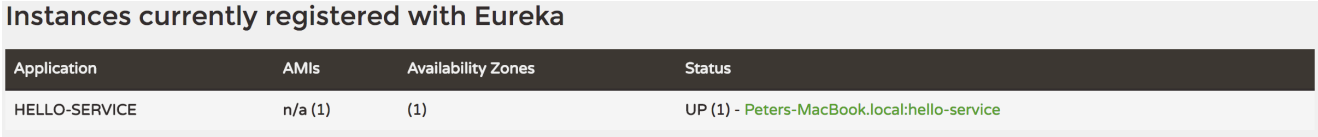
5. 启动该程序，在浏览器访问 http://localhost:8080/
6. 我们在该客户端输出中看到类似内容：

```
DiscoveryClient_HELLO-SERVICE/Peters-MacBook.local:hello-service: registering service...
DiscoveryClient_HELLO-SERVICE/Peters-MacBook.local:hello-service - registration status:
204
```

7. 在服务器的另一端看到类似输出：

```
Registered instance HELLO-SERVICE/Peters-MacBook.local:hello-service with status UP
(replication=false)
```

8. 通过访问 http://localhost:10000 看到类似内容




## 构建高可用性服务注册中心

1. 为了在单机上模拟多个 IP 地址，我们需要首先修改/etc/hosts 文件，加入

```
127.0.0.1       peer1
127.0.0.1       peer2
```

2. 以 eureka-server 为蓝本，建立新的 Spring Boot 程序 eureka-server1。或者利用 Spring Boot 的参数功能，在 eureka-server 中加入 application-peer1.properties 和 application-peer2.properties 文件。这两个文件或者这两个项目中的属性文件内容分别为
   - 修改 eureka-server 的 application.properties 为

```
server.port=10000

eureka.instance.hostname=peer1
eureka.client.service-url.defaultZone=http://peer2:10001/eureka/
```

     注意最后一行中的端口号是 10001 而不是 10000
   - 修改 eureka-server1 的 application.properties 为

```
server.port=10001

eureka.instance.hostname=peer2
eureka.client.service-url.defaultZone=http://peer1:10000/eureka/
```

     注意最后一行中的端口号是 10000 而不是 10001
3. 分别启动两个项目（或者使用 java –jar ****.jar –spring.profile.active=peer1, java –jar ****.jar –spring.profile.active=peer2 利用同一个项目但是使用不同的属性文件来达到启动两个项目的目的）
   访问 localhost:10000 以及 localhost:10001，会看到类似如下输出（等一小段时间后再刷新），双方都已经进行互相注册。

## Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
| --- | --- | --- | --- |
| HELLO-SERVICE | n/a (1) | (1) | UP (1) - peters-macbook:hello-service |
| UNKNOWN | n/a (2) | (2) | UP (2) - peters-macbook:10001 , peters-macbook:10000 |

## General Info

| Name | Value |
| --- | --- |
| total-avail-memory | 565mb |
| environment | test |
| num-of-cpus | 4 |
| current-memory-usage | 123mb (21%) |
| server-uptime | 00:01 |
| registered-replicas | http://peer1:10000/eureka/ |
| unavailable-replicas | |
| available-replicas | http://peer1:10000/eureka/, |

4. 在设置了多个节点的服务注册中心之后，服务提供方还需要进行修改才能够将服务注册到 Eureka Server 集群中：

```
spring.application.name=hello-service
eureka.client.service-url.defaultZone=http://peer1:10000/eureka,http://peer2:10001/eureka
```

# 服务发现与消费

1. 建立 Spring Boot 项目命名为 ribbon-consumer，加入如下依赖：

```
compile('org.springframework.boot:spring-boot-starter-web')
compile('org.springframework.cloud:spring-cloud-starter-netflix-eureka-server')
compile('org.springframework.cloud:spring-cloud-starter-netflix-ribbon')
```

2. 启动 2 个服务注册中心和 2 个 hello-service 服务并运行在不同端口
3. 在 RibbonConsumerApplication 加入如下内容：

```
package com.example.ribbonconsumer;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
@EnableDiscoveryClient
public class RibbonConsumerApplication {

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

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

4. 创建 ConsumerController

```
package com.example.ribbonconsumer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
public class ConsumerController {
```

```
    @Autowired
    private RestTemplate restTemplate;

    @RequestMapping("/ribbon-consumer")
    public String helloConsumer() {
        return restTemplate.getForEntity("http://HELLO-SERVICE/", String.class).getBody();
    }
}
```

注意 http://HELLO-SERVICE/

5. 修改 application.properties

   spring.application.name=ribbon-consumer

   server.port=10003

   eureka.client.service-url.defaultZone=http://localhost:10000/eureka/

6. 通过 localhost:10003/ribbon-consumer 发起请求，得到 hello，world

## 自我保护

当我们在本地调试 Eureka 程序时，在服务注册中心的信息面板上会出现类似如下的红色警告信息，

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.

可以使用 eureka.server.enable-self-preservation=false 来关闭自我保护机制。

# Spring Cloud Ribbon

## 什么是 Spring Cloud Ribbon

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。通过 Spring Cloud 的封装，可以让我们轻松地将面向服务的 REST 模版请求自动转换成客户端负责均衡的服务调用。

## 客户端负载均衡

客户端负载均衡和服务器端负载均衡最大的不同点在于服务清单所存储的位置。在客户端负载均衡中，所有客户端节点都护着自己要访问的服务端清单，而这种服务端的清单来自于服务注册中心（比如 Eureka 客户端）。同服务端负载均更的架构类似，在客户端负载均衡中也需要心跳去维护服务器端清单的健康性，只是这个步骤需要配合服务注册中心来完成。

在 Spring Cloud 实现的服务治理框架中，默认会创建针对各个服务治理框架的 Ribbon 自动化整合配置。

通过 Spring Cloud Ribbon 的封装，我们在微服务架构中使用客户端负载均衡调用非常简单：

1. 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
2. 服务消费者直接通过调用@LoadBalanced 注解修饰过的 RestTemplate 来实现面向服务的接口调用

```
@SpringBootApplication
@EnableDiscoveryClient
public class RibbonConsumerApplication {

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

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

```
package com.example.ribbonconsumer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
```

```
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
public class ConsumerController {

    @Autowired
    private RestTemplate restTemplate;

    @RequestMapping("/ribbon-consumer")
    public String helloConsumer() {
        return restTemplate
                .getForEntity("http://HELLO-SERVICE/", String.class)
                .getBody();

    }
}
```

3. 在 application.properties 中加入

```
spring.application.name=ribbon-consumer
server.port=1000
eureka.client.service-url.defaultZone=http://peer1:10000/eureka/
eureka.server.enable-self-preservation=false
```

4. 启动 ribbon-consumer，观察它的控制台信息。Robbin 输出了当前客户端维护的 HELLO-SERVICE 的服务列表情况。其中包含了各个实例的位置，Ribbon 就是按照此信息进行轮询访问，以实现基于客户端的负载均衡。另外还输出了一些非常有用的信息，如对各个实例的请求总数量、第一次连接信息，上一次连接信息，总的请求失败数量等。
5. 再次尝试发送几次请求，并观察 HELLO-SERVICE 控制台，可以看到两个控制台交替打印日志（日志中包含了我们代码中的输出），由此可以判断当前 ribbon-consumer 对 HELLO-SERVICE 的调用是负载均衡的。

# RestTemplate 详解

## GET 请求

在 RestTemplate 中，对 GET 请求可以通过如下两种方式进行调用实现：

1. getForEntity 方法。该方法的返回值时 ResponseEntity，该对象时 Spring 对 HTTP 请求响应的封装，其中主要存储了 HTTP 的几个重要元素，比如请求状态码的枚举对象 HttpStatus，在它的父类 HttpEntity 中还存储着 HTTP 请求的头信息对象 HttpHeaders 以及范型类型的请求体对象，比如：

```
RestTemplate restTemplate = new RestTemplate();
ResponseEntity<String> responseEntity = restTemplate.getForEntity(http://USER-
SERVICE/user?name={1}, String.class, "tom");
String body = responseEntity.getBody();
```

或者

```
ResponseEntity<User> responseEntity = restTemplate.getForEntity(http://USER-
SERVICE/user?name={1}, User.class, "tom");
String body = responseEntity.getBody();
```

2. getForObject 方法。该方法可以理解为 getFotEntity 的进一步封装，它通过 HttpMessageConverterExtractor 对 HTTP 的请求响应体 body 内容进行对象转换，实现请求直接返回包装好的对象内容，比如：

```
RestTemplate restTemplate = new RestTemplate();
String result = restTemplate.getForObject(uri, String.class);
```

或者

```
User result = restTemplate.getForObject(uri, User.class);
```

当不需要关注请求响应除了 body 外的其它内容时，该函数非常好用。

## POST 请求

## PUT 请求

## DELETE 请求

# Spring Cloud Hystrix

在微服务架构中，存在着许多服务单元，如果某些单元出现故障，那就很容易因为依赖关系而引发故障的蔓延，最终导致整个系统的瘫痪，这样的架构相较传统架构更加不稳定。为了解决这样的问题，产生了断路器等一系列的服务保护机制。
当某个服务单元发生故障之后，通过断路器的故障监控，想调用房返回一个错误相应，而不是长时间的等待。这样就不会使得线程因调用故障服务被长时间占用不释放，避免了故障在分布式系统中的蔓延。

## 构建断路器的实例

启动 eureka-server 集群，注册两个 hello-service 工程，注册 ribbon-consume 工程。此时关闭其中一个 hello-service 工程，发送请求到 ribbon-consume，可以获得 500 的错误信息。下面我们加入断路器功能：

1. 在 ribbon-consume 中引入断路器，即 Hystrix
2. 在 ribbon-consume 的主类中加入@EnableCircuitBreaker 注解，或者使用@SpringCloudApplication
3. 之后我们改造一下消费方式，增加 HelloService 类，注入 RestTemplate 实例

```
package com.example.ribbonconsumer;

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class HelloService {

    @Autowired
    private RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "helloFallback")
    public String helloservice() {
        return restTemplate
                .getForEntity("http://HELLO-SERVICE", String.class)
                .getBody();
    }

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

4. 修改 Controller

```
package com.example.ribbonconsumer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
public class ConsumerController {

    @Autowired
    private HelloService helloService;

    @RequestMapping("/ribbon-consumer")
    public String helloConsumer() {
        return helloService.helloservice();
    }
}
```

5. 下面， 我们来验证一 下 通过断路器实现的服务回调逻辑，重新启动之前关闭的 Hello-Service，确保此时服务注册中心、两 个 Hello-Service 以及 RIBBON-CONSUMER 均启动，访问 ribbon-consumer 可以轮询两个 HELLO-SERVICE 并返回一些信息。此 时我们断开某一个 HELLO-SERVIC，然后再次访问 ribbon-consumer，当轮询到被关闭的那个 HELLI-SERVICE 时，输出内容为 error， 而不是之前的 500 错误内容。此时 Hystrix 的服务回掉生效了。

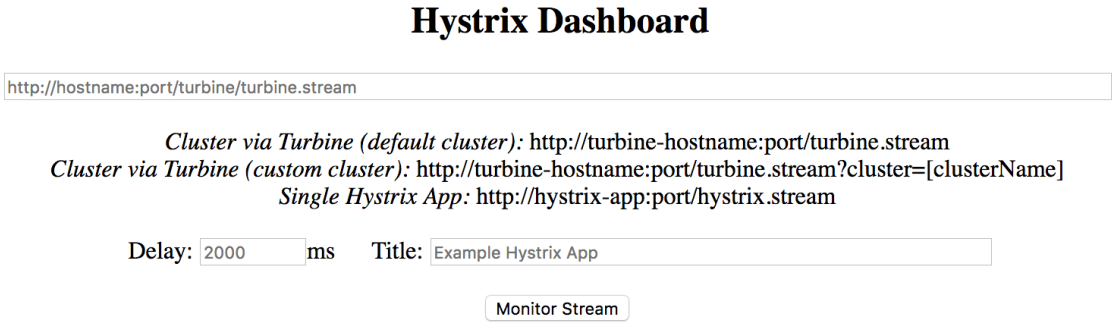
# Hystrix 仪表盘

## 搭建

1. 建立 hystrix-dashboard 工程，加入 hystrix-dashboard 依赖：spring-cloud-starter-hystrix
2. 在 Spring Boot 项目的主类上加入@EnableHystrixDashboard
3. 修改 application.propertis 配置文件，加入

```
spring.application.name=hystrix-dashboard
server.port=30000
```

4. 访问 http://localhost:30000/hystrix




## Hystrix Dashboard 的三种不同监控方式

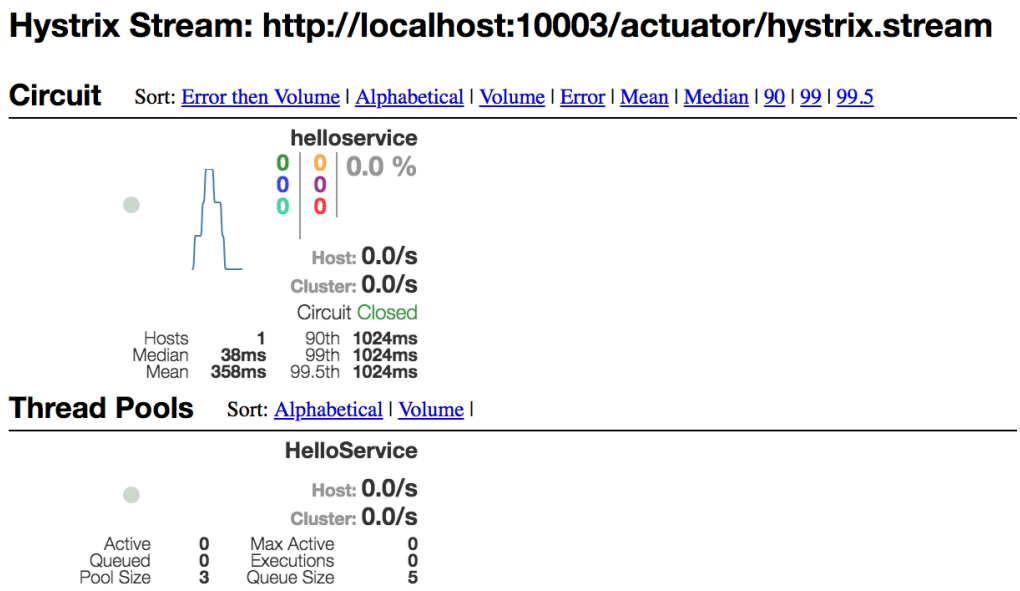
1. 单体应用的监控

   通过 http://turbine-hostname:port/hystrix.stream 开启

   - 在 RIBBON-CONSUMER 项目中引入 hystrix 和 actuator 依赖
   - 在服务实例的主类中使用@EnableCircuitHystrix 注解

   之后 actuator 会打印大量的监控点（注意在 application.properties 中加入：management.endpoints.web.exposure.include=*），其中 hystrix.stream 就是用于进行单体监控的。

   在 hystrix dashboard 的首页输入类似：http://localhost:10003/actuator/hystrix.stream 的内容，进入后可以看到如下结果：




2. 默认的集群监控
   通过 http://turbine-hostname:port/turbine.stream 开启
3. 指定的集群监控
   通过 http://turbine-hostname:port/turbine.stream?cluster=clusterName 开启

## Turbine 集群监控

上面的默认和指定集群监控是通过 Turbin 完成的。为此我们需要建立名为 turbine 的新 Spring Boot 工程，

1. 加入如下支持：

compile('org.springframework.boot:spring-boot-starter-actuator')
compile('org.springframework.cloud:spring-cloud-starter-netflix-turbine')

2. 在项目的主类中加入@EnableTurbine 和@EnableDiscoveryClient 注解
3. 在 application.properties 中加入 eureka 和 turbine 的相关配置

```
management.endpoints.web.exposure.include=*

spring.application.name=turbine

server.port=10004
management.prot=10005

eureka.client.service-url.defaultZone=http://peer1:10000/eureka/

turbine.app-config=RIBBON-CONSUMER
turbine.cluster-name-expression="default"
turbine.combine-host-port=true
```

4. 启动高可用注册中心，hello-service 的两个实例，ribbon-consumer，turbine 以及 hystrix-dashboard
5. 在 hystrix dashboard 的首页输入类似：http://localhost:10004/turbine.stream 的内容，进入后可以看到如下结果：
6. 如果我们启动两个名称不同(spring.application.name)的 RIBBON-CONSUMER，可以看到上图中有两个不同图示的输出。

# Spring Cloud Feign

在微服务的架构中，Ribbon 和 Hystrix 几乎是同时出现的。本部分介绍的 Feign 就是这样一个更高层次的封装，用于整合这两个基础的工具用以简化开发。除此之外，Feign 还提供了一种声明式的 Web 服务客户端定义方式。

1. 创建一个名为 feign-consumer 的 Spring Boot 工程，加入 eureka，feign，web 依赖
2. 在主类上启用@EnableFeignClients，@EnableDiscoveryClient
3. 定义 HelloService 接口

```
package com.example.feignconsumer;

import org.springframework.cloud.openfeign.FeignClient;
```

```
import org.springframework.web.bind.annotation.RequestMapping;

@FeignClient("hello-service")
public interface HelloService {

    @RequestMapping("/")
    String hello();
}
```

4. 创建 ConsumerContorller

```
package com.example.feignconsumer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ConsumerContorller {

    @Autowired
    private HelloService helloService;

    @RequestMapping("/feign-consumer")
    public String helloConsumer() {
        return helloService.hello();
    }
}
```

5. 修改 application.properties

```
spring.application.name=feign-consumer
server.port=30000
eureka.client.service-url.defaultZone=http://peer1:10000/eureka
```

6. 测试验证

   发送几次请求到 feign-consumer，可以得到和之前 ribbon 实现时的相同效果即可。

# Spring Cloud Zuul

## 环境搭建

1. 创建名为 api-gateway 的 Spring Boot 工程，加入 zuul 的依赖。
2. 在主类中加入@EnableZuulProxy，@SpringCloudApplication
3. 在 application.properties 中加入

```
spring.application.name=gateway
server.port=11000
```

## 请求路由

### 传统路由

启动注册中心，hello-service 和 feign-consumer

对 api-gateway 服务增加关于路由规则的配置，就能实现传统的路由转发，如：

```
zuul.routes.apihello.path=/apihello/**
zuul.routes.apihello.url=http://localhost:20000/
```

注意以上两行需要配对使用。当我们访问 http://localhost:11000/apihello/hello 时会自动定位到 http://localhost:20000/ 。

### 面向服务的路由

由于可以让 Spring Cloud Zuul 和 Cloud Eureka 无缝整合，所有路由器的 path 可以不去具体映射某一个 url 地址，而是让它映射到某一个具体的服务，而具体的 url 则交给 Eureka 服务发现机制去自动维护，我们称这类路由为面向服务的路由。此时，我们只需要引入 eureka 并注册 api-gateway 即可。配置文件：

```
spring.application.name=gateway
server.port=11000

zuul.routes.apihello.path=/apihello/**
zuul.routes.apihello.serviceId=hello-service

zuul.routes.feignconsumer.path=/feignconsumer/**
zuul.routes.feignconsumer.serviceId=feign-consumer
```

```
eureka.client.service-url.defaultZone=http://peer1:10000/eureka
```

### 请求过滤

每个客户端用户请求微服务应用提供的接口时，它们的访问权限往往都会有一定的限制，系统并不会将所有的微服务接口都对它们开放。然而，目前的服务路由并没有限制权限这样的功能。对于这样的问题，好的做法是通过前置的网关服务来完成这些非业务性质的校验。为了实现该功能，我们只需要继承 ZuulFilter 抽象类即可：

```
package com.example.apigateway;

import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;
import com.netflix.zuul.exception.ZuulException;

import javax.servlet.http.HttpServletRequest;

public class AccessFilter extends ZuulFilter {
    @Override
    public String filterType() {
        return "pre";
    }

    @Override
    public int filterOrder() {
        return 0;
    }

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

    @Override
    public Object run() throws ZuulException {
        RequestContext requestContent = RequestContext.getCurrentContext();
        HttpServletRequest httpServletRequest = requestContent.getRequest();

        Object accessToken = httpServletRequest.getParameter("accessToken");
        if(accessToken == null) {
            requestContent.setSendZuulResponse(false);
            requestContent.setResponseStatusCode(401);
        }
        return null;
    }
}
```

在主类中将上面的代码配置为一个 Bean。之后我们发起请求：

| | |
|---|---|
| http://localhost:11000/apihello/ | 会返回 401 错误 |
| http://localhost:11000/apihello&accessToken=token | 正确路由得到 hello world |

## Spring Cloud Config

## Spring Cloud Bus

## Spring Cloud Stream

## Spring Cloud Sleuth