



Spring Cloud

Edgware.RELEASE

凯盛软件

- <https://projects.spring.io/spring-cloud/>
- Spring Cloud 是一个基于SpringBoot实现的微服务架构开发工具。为微服务架构中涉及的服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等操作提供了一种简单的开发方式
- Spring Cloud 中包含大量的子项目，这些子项目协同工作一起完成微服务中的各个方面
 - Spring Cloud Netflix 集成众多 Netflix 的开源软件
 - Netflix Eureka服务中心，云端服务发现，一个基于 REST 的服务，用于定位服务，以实现云端中间层服务发现和故障转移
 - Netflix Hystrix 熔断器，容错管理工具，旨在通过熔断机制控制服务和第三方库的节点,从而对延迟和故障提供更强大的容错能力
 - Netflix Zuul 云平台上提供动态路由,监控,弹性,安全等边缘服务的框架
 - Netflix Archaius 配置管理API，包含一系列配置管理API，提供动态类型化属性、线程安全配置操作、轮询框架、回调机制等功能
 - Spring Cloud Config 配置中心，配置管理工具包，让你可以把配置放到远程服务器，集中化管理集群配置，目前支持本地存储、Git以及Subversion
 - ...

Spring Cloud VS Dubbo

凯盛软件

核心要素	Dubbo	Spring Cloud
服务注册中心	Zookeeper、Redis	Spring Cloud Netflix Eureka
服务调用方式	RPC	REST API
服务网关	无	Spring Cloud Netflix Zuul
断路器	不完善	Spring Cloud Netflix Hystrix
分布式配置	无	Spring Cloud Config
分布式追踪系统	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream 基于Redis,Rabbit,Kafka实现的消息微服务
批量任务	无	Spring Cloud Task

版本号

凯盛软件

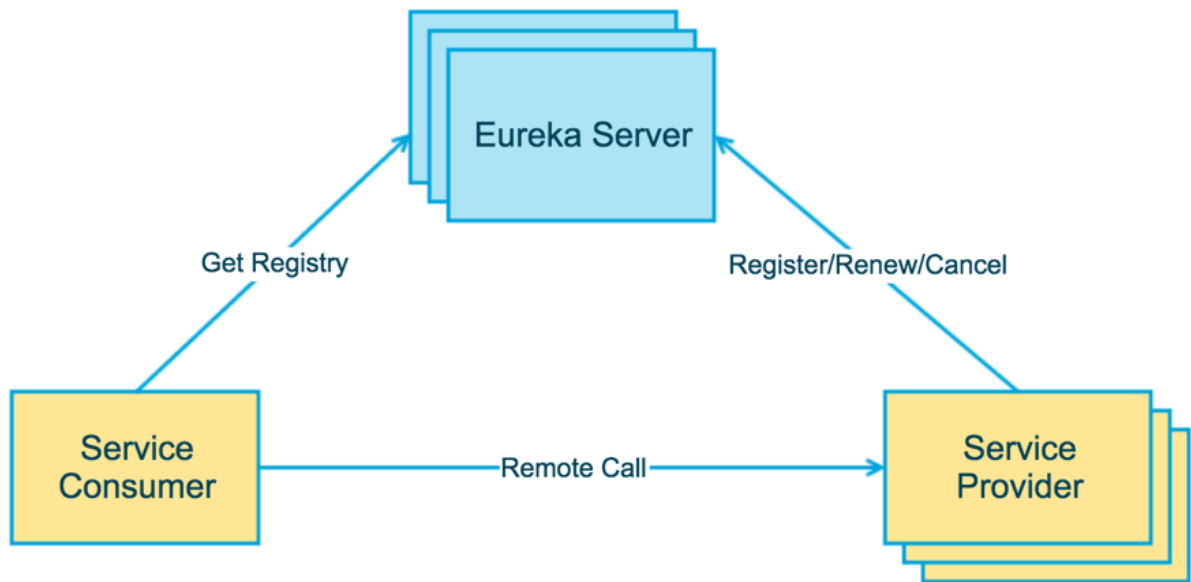
Spring Cloud是一个拥有诸多子项目的大型综合项目，原则上其子项目也都维护着自己的发布版本号。那么每一个Spring Cloud的版本都会包含不同的子项目版本，为了要管理每个版本的子项目清单，避免版本名与子项目的发布号混淆，所以没有采用版本号的方式，而是通过命名的方式

版本名字采用了伦敦地铁站的名字，根据字母表的顺序来对应版本时间顺序，比如：最早的Release版本：Angel，第二个Release版本：Brixton，以此类推.....

当一个版本的Spring Cloud项目的发布内容积累到临界点或者一个严重bug解决可用后，就会发布一个“service releases”版本，简称SRX版本，其中X是一个递增数字

在发布SR版本前先发布Release版本





- Spring Cloud 封装了 Netflix 公司开发的 Eureka 模块来实现服务注册和发现。Eureka 采用了 C-S 的设计架构。Eureka Server 作为服务注册功能的服务器，它是服务注册中心。而系统中的其他微服务，使用 Eureka 的客户端连接到 Eureka Server，并维持心跳连接。
- Eureka由两个组件组成：Eureka服务器和Eureka客户端。

- 创建Spring Boot 项目
- 添加pom依赖

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Edgware.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

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

- 在App类上添加注解@EnableEurekaServer

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {

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

- 在application.properties中添加配置

#应用的名称

spring.application.name=EurekaServer-1

#项目端口号

server.port=7788

#是否将自己作为服务注册到eureka中

eureka.client.register-with-eureka=false


#是否从Eureka Server获取注册信息

eureka.client.fetch-registry=false

#设置与Eureka Server交互的地址，查询服务和注册服务都需要依赖这个地址，多个地址可使用，分隔

eureka.client.service-url.defaultZone=http://127.0.0.1:7788/eureka

- 打开浏览器访问 <http://ip:7788>



[HOME](#) [LAST 1000 SINCE STARTUP](#)

System Status

Environment	test	Current time	2017-12-21T10:19:37 +0800
Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

DS Replicas

127.0.0.1

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

Name	Value
total-avail-memory	452mb

- 创建Spring boot 项目
- 添加 pom 依赖

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Edgware.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

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

- 在App类上添加注解@EnableDiscoveryClient开启服务注册与发现

```
@SpringBootApplication
@EnableDiscoveryClient
public class MovieServiceProviderApplication {

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

- 在application.properties中添加配置

```
#服务名称
spring.application.name=movie-service-provider
#eureka server的地址
eureka.client.service-url.defaultZone=http://127.0.0.1:7788/eureka
```


- 声明Controller用来提供服务

```
@RestController
public class MovieController {

    @GetMapping("/movie/{id:\\d+}")
    public Movie findById(@PathVariable Integer id) {
        return new Movie(1001, "无间道II", "Andy Liu");
    }

}
```

- 在Eureka管理界面查看注册的服务

HOME LAST 1000 SINCE STARTUP

System Status

Environment	test	Current time	2017-12-21T11:05:27 +0800
Data center	default	Uptime	00:07
		Lease expiration enabled	false
		Renews threshold	3
		Renews (last min)	0

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

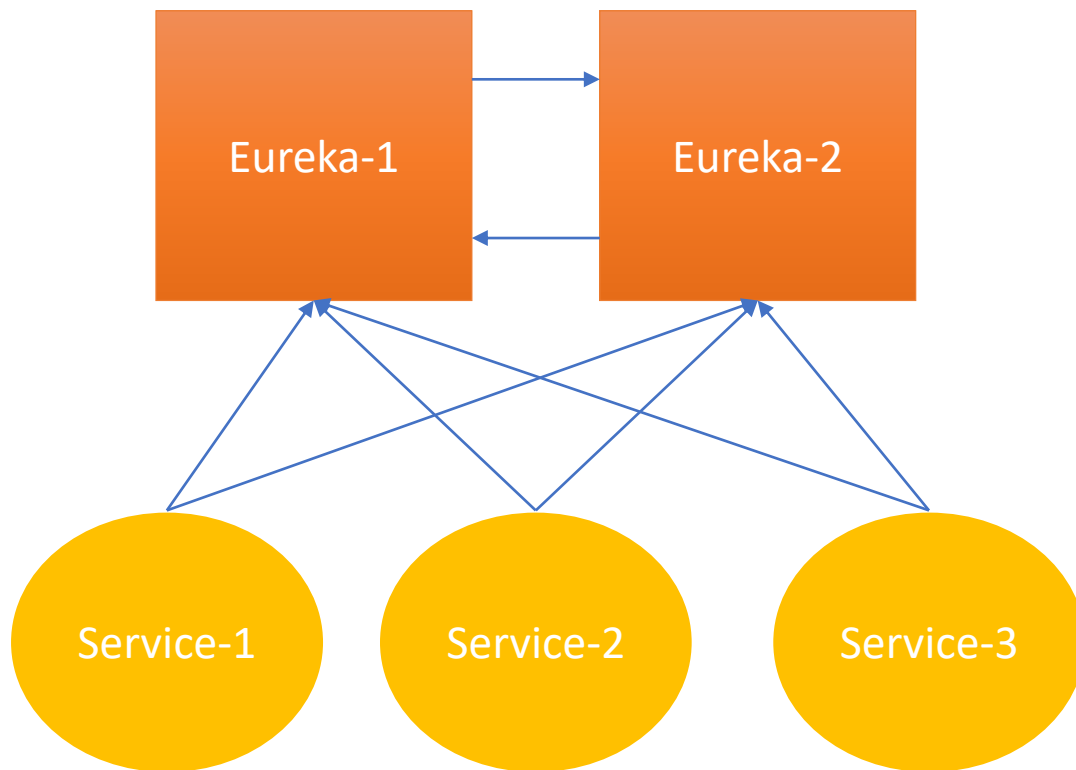
127.0.0.1

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MOVIE-SERVICE-PROVIDER	n/a (1)	(1)	UP (1) - localhost:movie-service-provider

General Info

Name	Value
total-avail-memory	547mb



- application-**node1**.properties

`spring.application.name=eureka-server-1`

`server.port=7788`

#自己的hostname

`eureka.instance.hostname=eureka-server-1`

#将自己注册到对方

`eureka.client.service-url.defaultZone=http://127.0.0.1:7799/eureka`

- application-**node2**.properties

`spring.application.name=eureka-server-2`

`server.port=7799`

`eureka.instance.hostname=eureka-server-2`

`eureka.client.service-url.defaultZone=http://127.0.0.1:7788/eureka`

- 修改操作系统的host文件
127.0.0.1 eureka-server-1
127.0.0.1 eureka-server-2
- 访问http://ip:7788和http://ip:7799

DS Replicas

127.0.0.1

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EUREKA-SERVER-1	n/a (1)	(1)	UP (1) - localhost:eureka-server-1:7788
EUREKA-SERVER-2	n/a (1)	(1)	UP (1) - localhost:eureka-server-2:7799

General Info

Name	Value
total-avail-memory	508mb
environment	test
num-of-cpus	8
current-memory-usage	215mb (42%)
server-uptime	00:01
registered-replicas	http://127.0.0.1:7788/eureka/
unavailable-replicas	http://127.0.0.1:7788/eureka/

将服务发布到Eureka集群中

凯盛软件

#服务名称

`spring.application.name=movie-service-provider`

#eureka server的地址

`eureka.client.service-url.defaultZone=http://127.0.0.1:7788/eureka,http://127.0.0.1:7799/eureka`

为Eureka添加用户认证

- 在Eureka服务项目中添加pom依赖并配置properties文件

`<dependency>`

`<groupId>org.springframework.boot</groupId>`

`<artifactId>spring-boot-starter-security</artifactId>`

`</dependency>`

#开启HTTP基本认证

`security.basic.enabled=true`

#账号

`security.user.name=ksit`

#密码

`security.user.password=root`

- 修改注册服务的配置

```
spring.application.name=eureka-server-1
```

```
server.port=7788
```

```
#自己的hostname
```

```
eureka.instance.hostname=eureka-server-1
```

```
#将自己注册到对方
```

```
eureka.client.service-url.defaultZone=http://ksit:root@127.0.0.1:7799/eureka
```

```
#服务名称
```

```
spring.application.name=movie-service-provider
```

```
#eureka server的地址
```

```
eureka.client.service-url.defaultZone=http://ksit:root@127.0.0.1:7788/eureka,http://ksit:root@127.0.0.1:7799/eureka
```

- 创建服务消费者的SpringBoot项目，添加web-starter依赖

`<dependency>`

`<groupId>org.springframework.boot</groupId>`

`<artifactId>spring-boot-starter-web</artifactId>`

`</dependency>`

- 创建RestTemplate实例

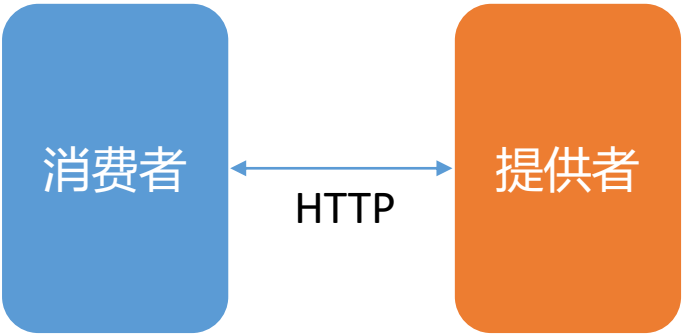
`@Bean`

```
public RestTemplate restTemplate() {  
    return new RestTemplate();  
}
```

- 在控制器中使用RestTemplate对象调用服务提供者提供的地址

```
@Autowired
private RestTemplate restTemplate;

@GetMapping("/movie/{id:\\d+}")
public Movie showMovieInfo(@PathVariable Integer id) {
    String url = "http://127.0.0.1:8080/movie/"+id;
    return restTemplate.getForObject(url,Movie.class);
}
```



- 创建服务消费者的SpringBoot项目，添加eureka依赖

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

- 配置application.properties

server.port=9090

#应用名称

spring.application.name=movie-service-consumer

#eureka服务地址

eureka.client.service-url.defaultZone=http://ksit:root@127.0.0.1:7788/eureka,http://ksit:root@127.0.0.1:7799/eureka

- 启用Eureka客户端

```
@SpringBootApplication
```

```
@EnableDiscoveryClient
```

```
public class MovieServiceConsumerApplication {
```

```
    @Bean
```

```
    public RestTemplate restTemplate() {
```

```
        return new RestTemplate();
```

```
    }
```

- 使用LoadBalancerClient接口获取ServiceInstance对象，该对象中保存了服务提供者的主机地址和端口号等信息，这样可以避免将服务提供者的信息硬编码到代码中
- 如果同一个服务有多个提供者， LoadBalancerClient接口还提供了负载均衡的功能

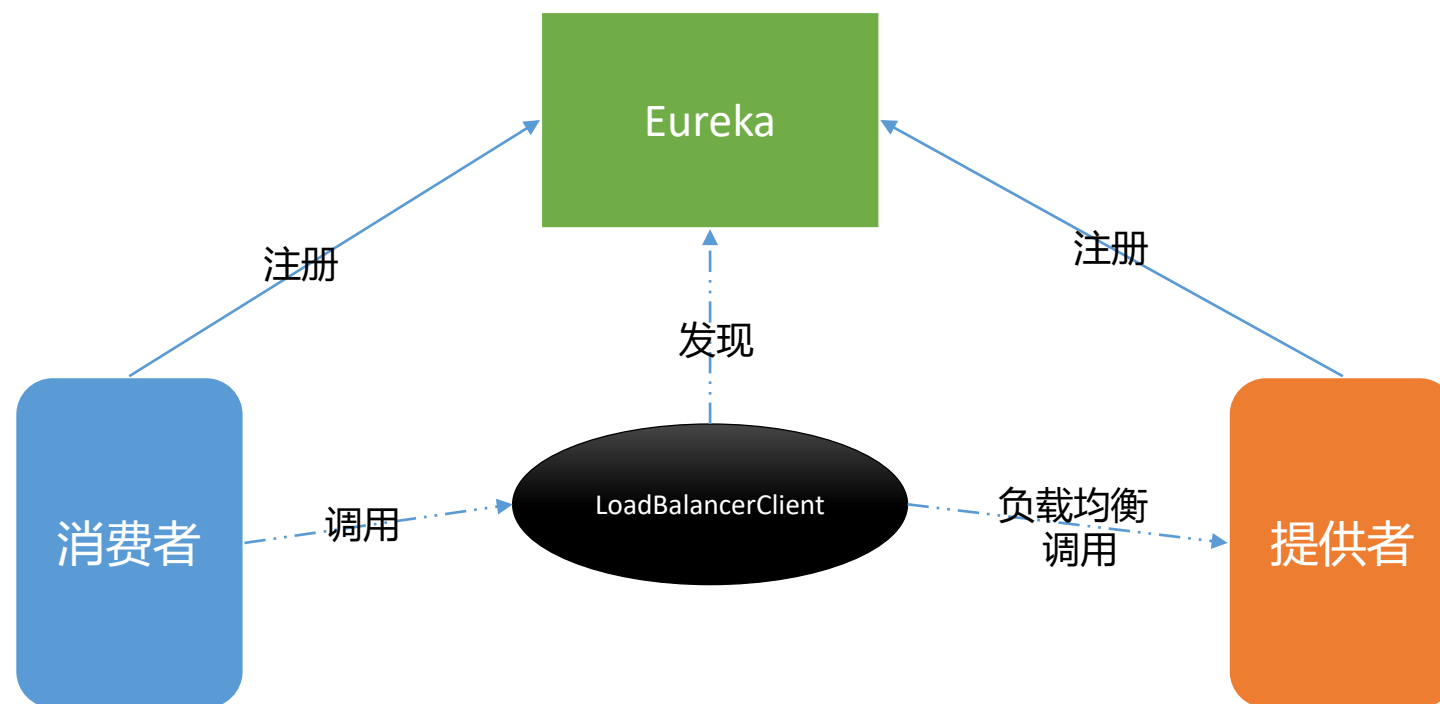
```
@Autowired
private RestTemplate template;

@Autowired
private LoadBalancerClient loadBalancerClient;

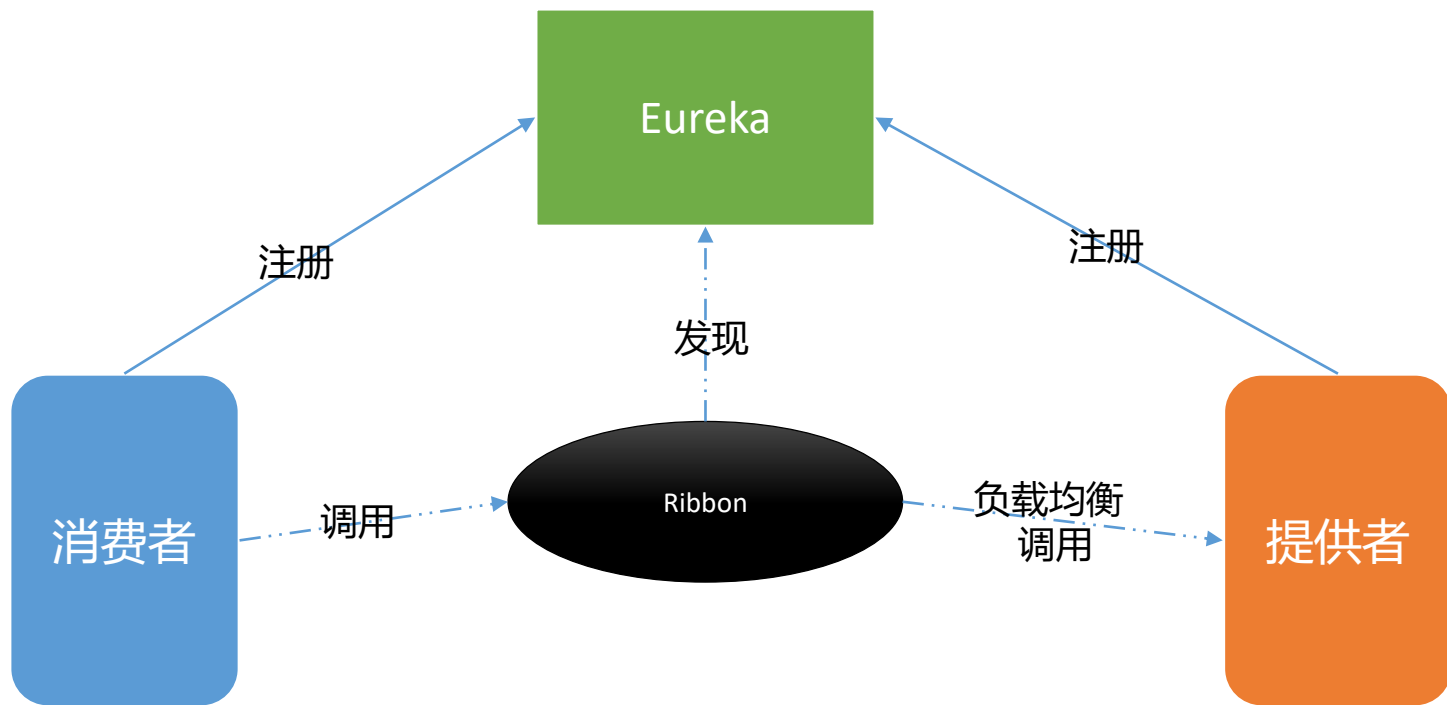
@GetMapping("/movie/{id:\\d+}")
public Movie showMovieInfo(@PathVariable Integer id) {
    ServiceInstance serviceInstance = loadBalancerClient.choose("MOVIE-SERVICE-PROVIDER");
    String url = "http://" + serviceInstance.getHost() + ":" + serviceInstance.getPort() + "/movie/" + id;
    return template.getForObject(url, Movie.class);
}
```

Instances currently registered with Eureka		
Application	AMIs	Availability Zones
EUREKA-SERVER-1	n/a (1)	(1)
EUREKA-SERVER-2	n/a (1)	(1)
MOVIE-SERVICE-CONSUMER	n/a (1)	(1)
MOVIE-SERVICE-PROVIDER	n/a (1)	(1)

- 当前架构



- Spring Cloud Ribbon是基于Netflix Ribbon实现的一套客户端负载均衡的工具。它是一个基于HTTP和TCP的客户端负载均衡器
- 当Ribbon与Eureka联合使用时，可以自动从Eureka注册中心中获取服务实例列表



- 添加Ribbon的依赖

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

- 在RestTemplate上添加注解

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

- 在控制器中调用

```
@GetMapping("/movie/{id:\\d+}")
public Movie showMovieInfo(@PathVariable Integer id) {
    String url = "http://MOVIE-SERVICE-PROVIDER/movie/"+id;
    return template.getForObject(url,Movie.class);
}
```

Instances currently registered with Eureka		
Application	AMIs	Availability Zones
EUREKA-SERVER-1	n/a (1)	(1)
EUREKA-SERVER-2	n/a (1)	(1)
MOVIE-SERVICE-CONSUMER	n/a (1)	(1)
MOVIE-SERVICE-PROVIDER	n/a (1)	(1)

- Spring Cloud Feign是一套基于Netflix Feign实现的声明式服务调用客户端。它使得编写Web服务客户端变得更加简单。只需要通过创建接口并用注解来配置它既可完成对Web服务接口的绑定。它具备可插拔的注解支持，包括Feign注解、JAX-RS注解。它也支持可插拔的编码器和解码器。Spring Cloud Feign还扩展了对Spring MVC注解的支持，同时还整合了Ribbon和Eureka来提供均衡负载的HTTP客户端实现。

- 添加 pom 依赖

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

- 启用FeignClient注解的支持

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

- 创建访问接口

```
@FeignClient(name = "MOVIE-SERVICE-PROVIDER")
public interface MovieServiceClient {

    @GetMapping("/movie/{id}")
    Movie findById(@PathVariable(name = "id") Integer id);
}
```

- 在控制器中使用

```
@Autowired
private MovieServiceClient movieServiceClient;

@GetMapping("/movie/{id:\\d+}")
public Movie showMovieInfo(@PathVariable Integer id) {
    return movieServiceClient.findById(id);
}
```

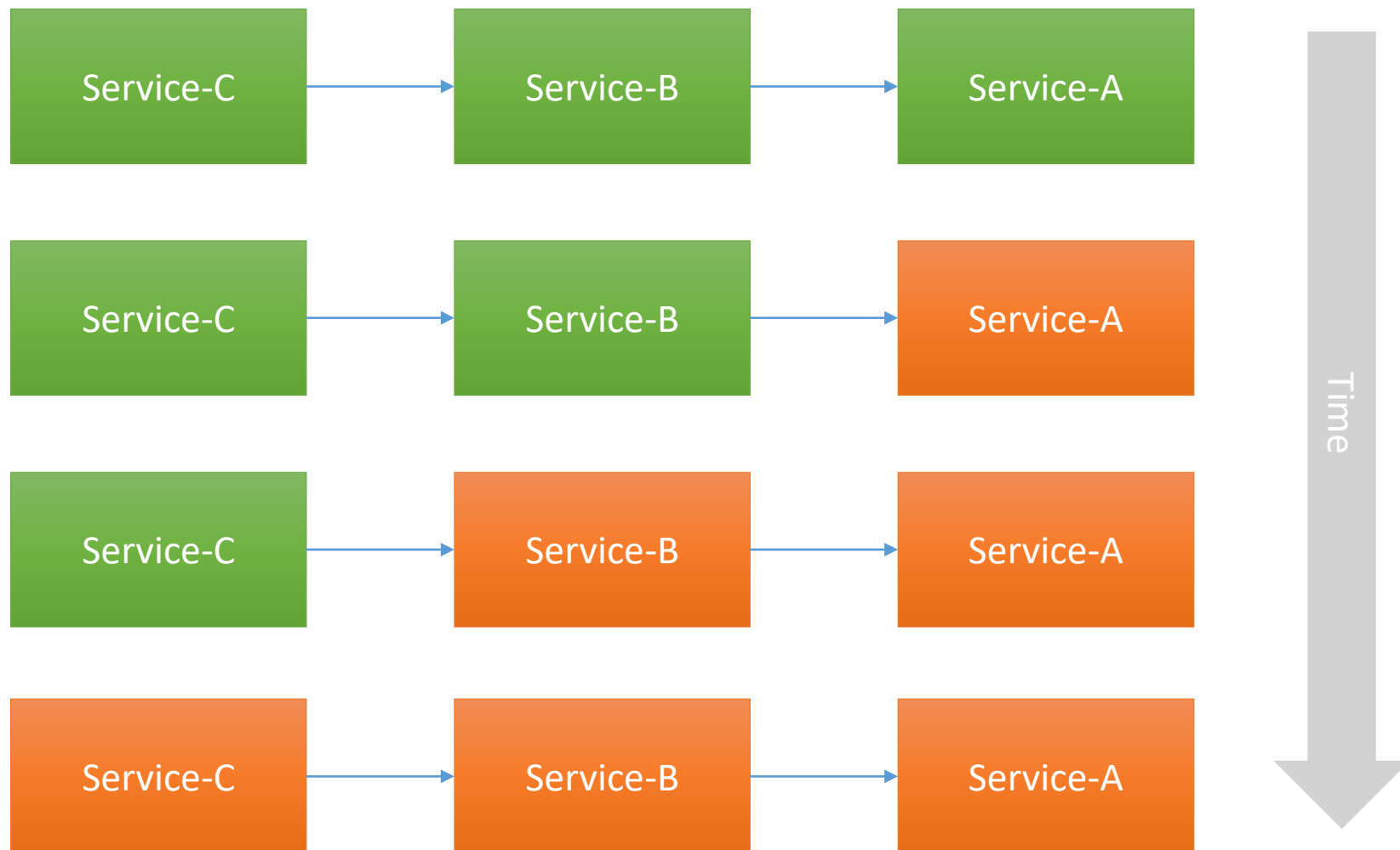
- post请求

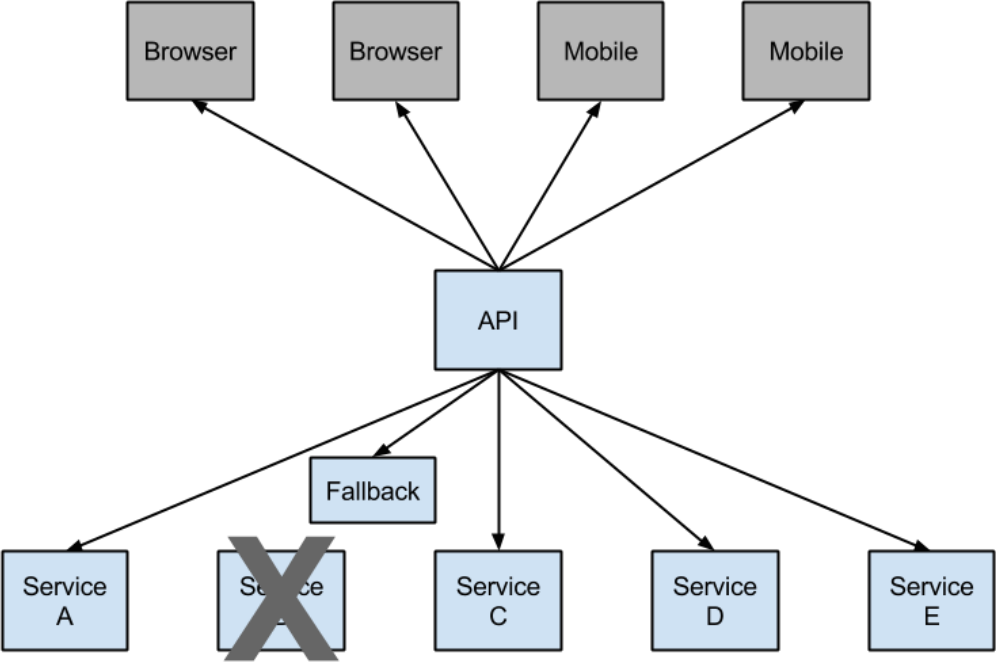
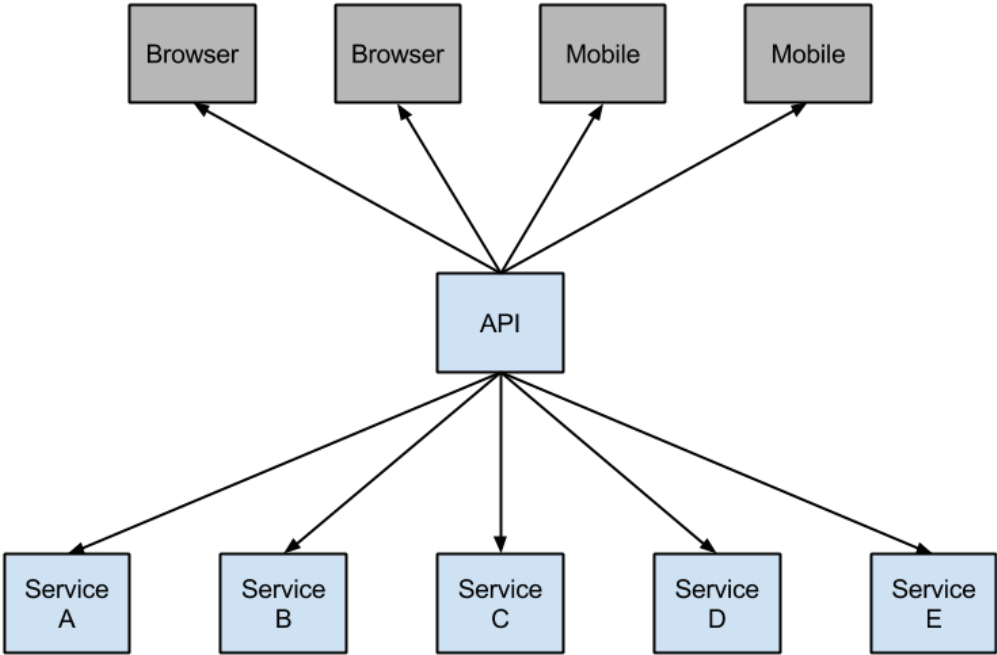
```
@PostMapping("/new")
```

```
String saveNewMovie(@RequestParam(name = "name") String name,  
                    @RequestParam(name = "actor") String actor);
```



- <https://github.com/Netflix/Hystrix>
- Netflix开源的一个延迟和容错库，用于隔离访问远程系统、服务或第三方库，防止级联失败，从而提升系统的可用性和容错性





- 在 pom 中添加依赖

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

- 启用Hystrix

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
public class MovieServiceConsumerApplication {

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

- 创建service类，在类中添加业务代码。使用@HystrixCommand注解标记业务方法，该注解的fallbackMethod属性用于配置容错的方法名称，该方法的返回值和参数列表需要和业务方法一致

@Service

```
public class MovieService {

    @Autowired
    private RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "getMovieFallback")
    public Movie getMovie(Integer id) {
        return restTemplate.getForObject("http://MOVIE-SERVICE-PROVIDER/movie/"+id,Movie.class);
    }

    public Movie getMovieFallback(Integer id) {
        return new Movie(-1,null,null);
    }
}
```

- 控制器中调用service类的业务方法

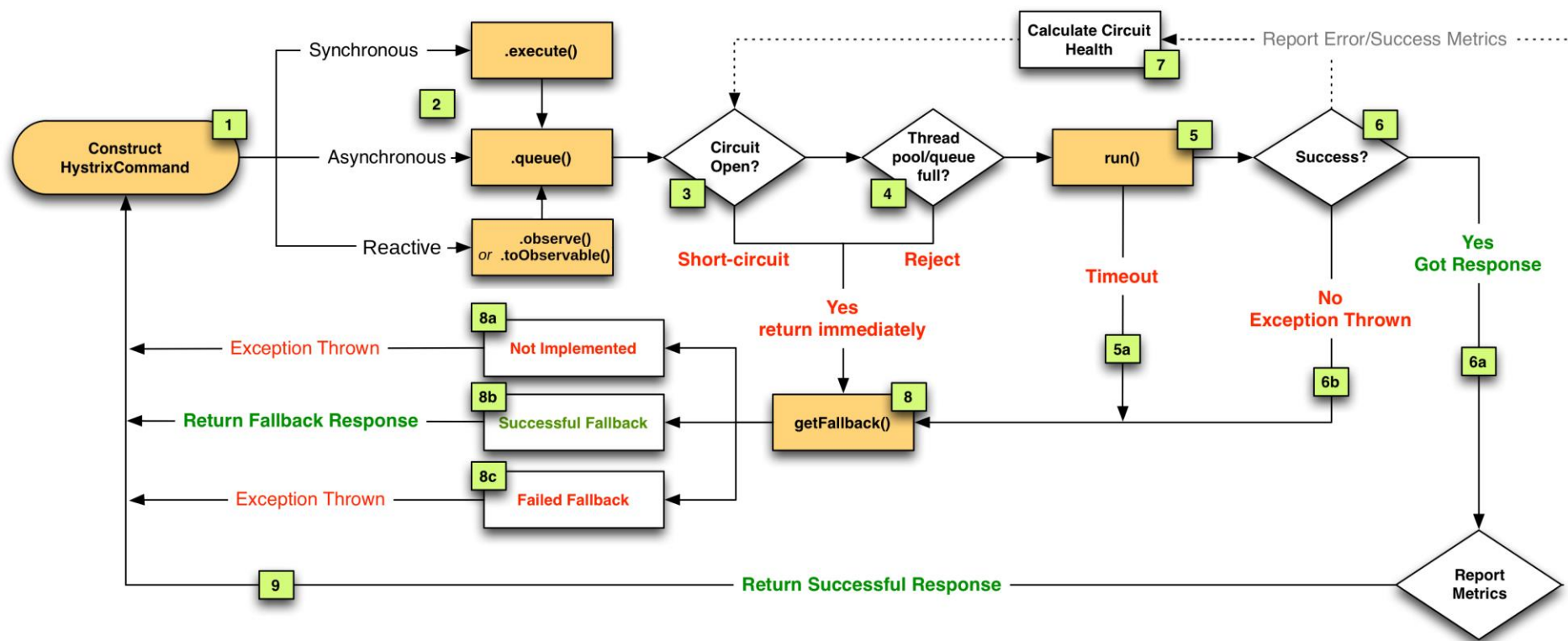
```
@Autowired
private MovieService movieService;

@GetMapping("/movie/{id:\\d+}")
public Movie showMovieInfo(@PathVariable Integer id) {
    return movieService.getMovie(id);
}
```

- 当服务提供者宕机或网络隔离等问题造成服务不可访问时，Hystrix会自动调用容错方法（服务降级）。服务恢复后，会调用正常的业务方法

Hystrix请求流程

凯盛软件



- 在application.properties配置文件中打开Feign对Hystrix的支持

```
#启用Feign Hystrix
```

```
feign.hystrix.enabled=true
```

- 创建Feign接口的实现类，并放入Spring容器

```
@FeignClient(name = "MOVIE-SERVICE-PROVIDER", fallback = MovieServiceClientFallback.class)
```

```
public interface MovieServiceClient {
```

```
    @GetMapping("/movie/{id}")
```

```
    Movie findById(@PathVariable(name = "id") Integer id);
```

```
@Component
```

```
public class MovieServiceClientFallback implements MovieServiceClient {
```

```
    @Override
```

```
    public Movie findById(Integer id) {
```

```
        return new Movie(0, null, null);
```

```
    }
```

- 在服务消费者项目中添加actuator依赖

```
<dependency>
```

```
  <groupId>org.springframework.boot</groupId>
```

```
  <artifactId>spring-boot-starter-actuator</artifactId>
```

```
</dependency>
```

- 启动项目后访问 <http://ip:port/hystrix.stream> 就可以看到监控信息

- Hystrix Dashboard 是一个可以将Hystrix监控数据进行可视化的项目
- 创建新的SpringBoot项目，并添加pom依赖

```
<dependency>
```

```
    <groupId>org.springframework.cloud</groupId>
```

```
    <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
```

```
</dependency>
```

- 在App类上添加@EnableHystrixDashboard注解，用来启用dashboard

```
@SpringBootApplication
```

```
@EnableHystrixDashboard
```

```
public class HystrixDashboardApplication {
```


- 访问 `http://ip:port/hystrix` 可以看到如下页面



Hystrix Dashboard

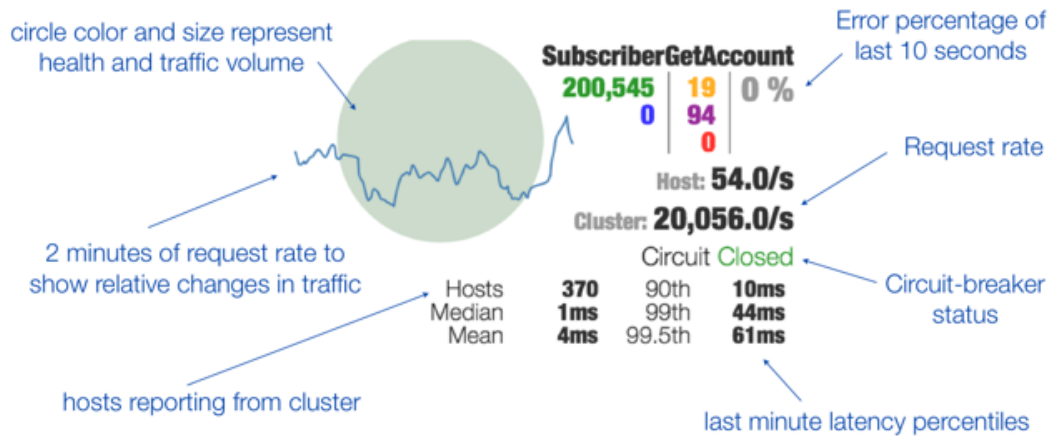
监控目标地址

Cluster via Turbine (default cluster): `http://turbine-hostname:port/turbine.stream`
Cluster via Turbine (custom cluster): `http://turbine-hostname:port/turbine.stream?cluster=[clusterName]`
Single Hystrix App: `http://hystrix-app:port/hystrix.stream`

Delay: ms Title: 名称: 自定义

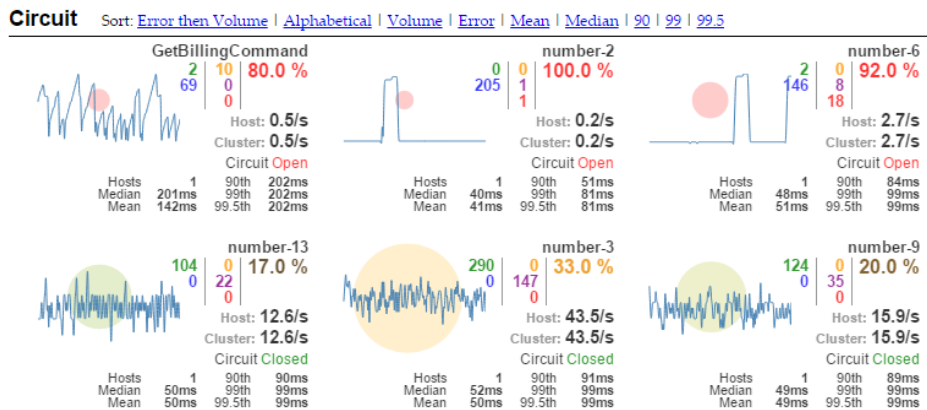
监控延迟

Monitor Stream



Rolling 10 second counters with 1 second granularity

Successes	200,545	19	Thread timeouts
Short-circuited (rejected)	0	94	Thread-pool Rejections
		0	Failures/Exceptions

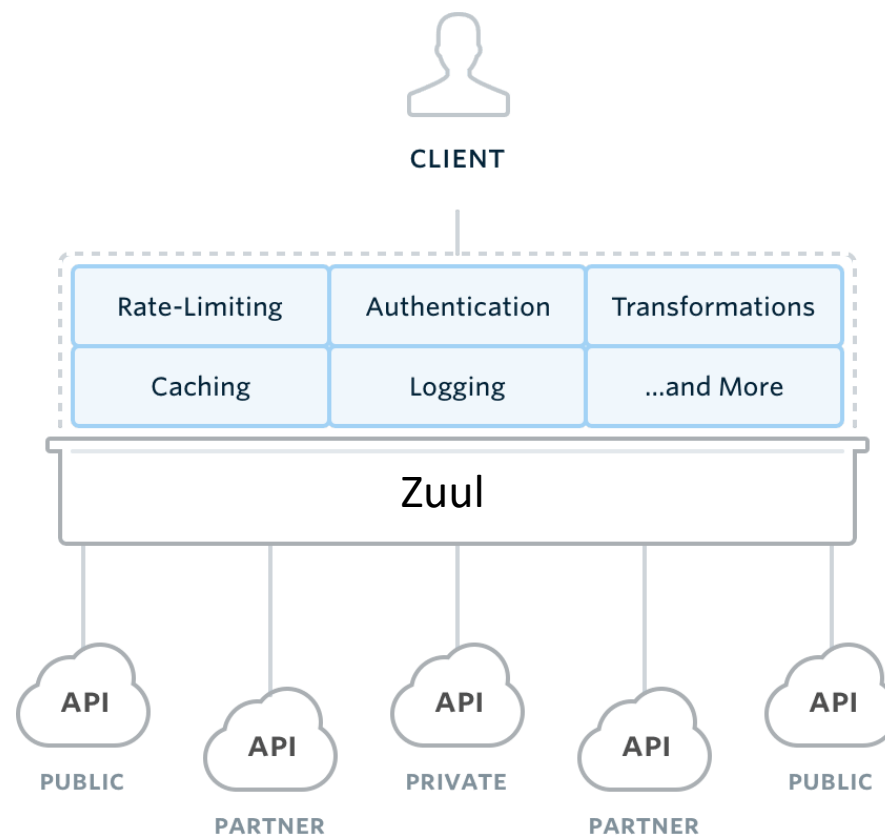
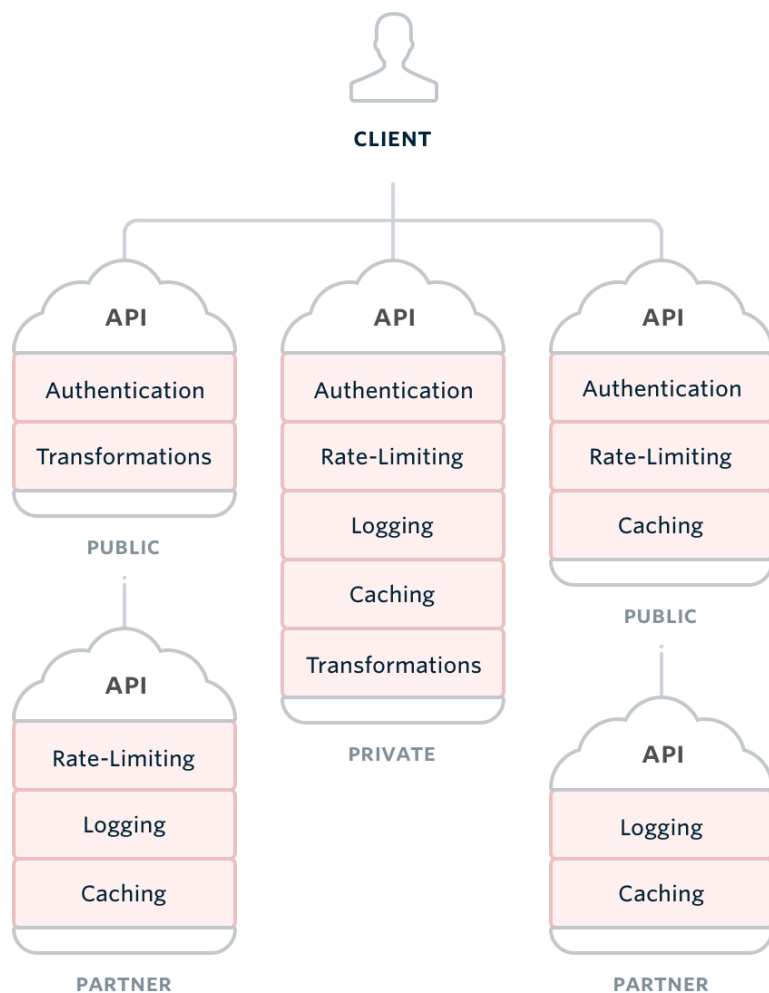


实心圆的颜色从绿黄橙红依次递减，表示服务的健康程度依次降低，圆越大，表示服务的流量越大

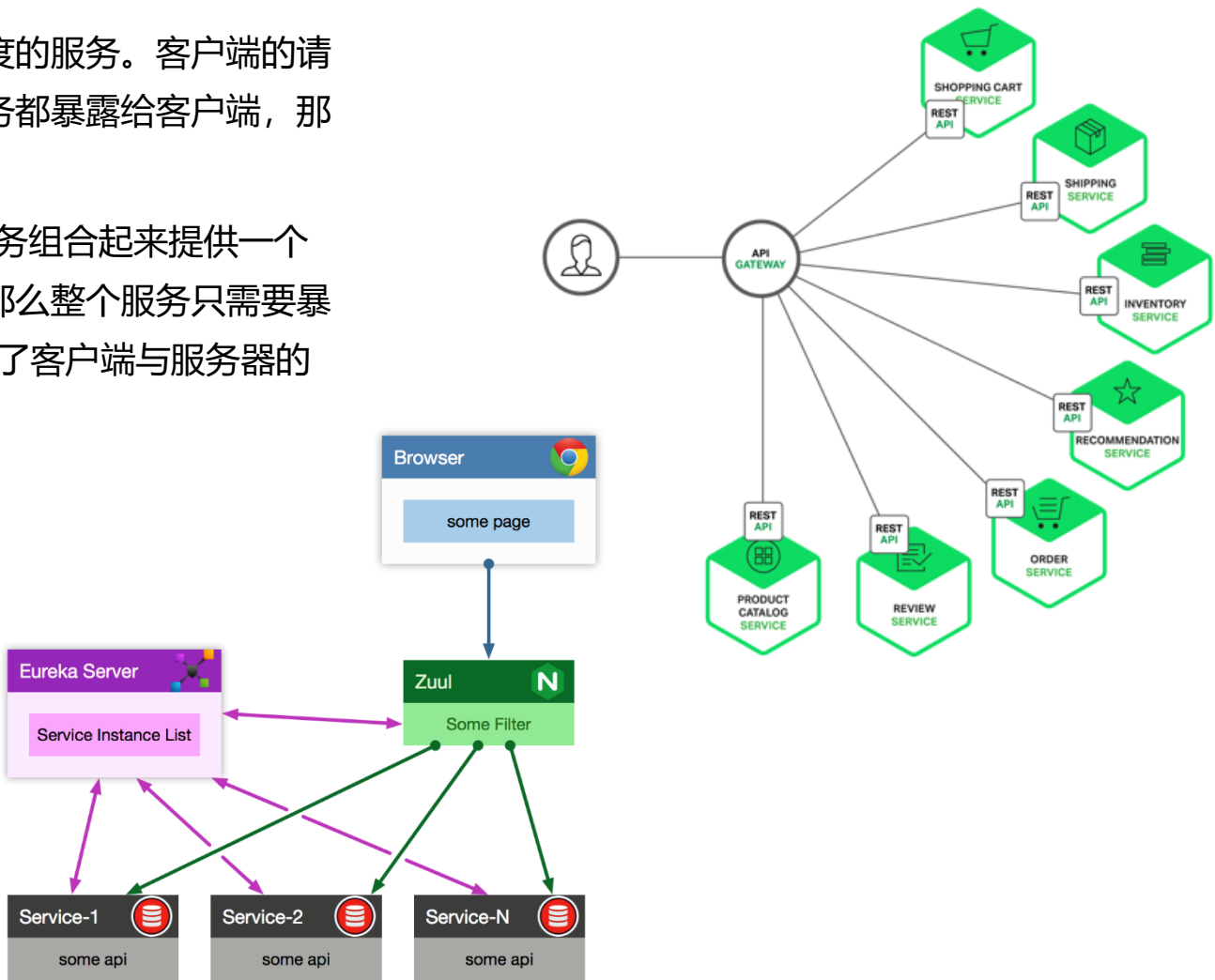
Hystrix Stream: myconsumer



<https://github.com/Netflix/Hystrix/wiki/Dashboard>



- 微服务场景下，每一个微服务对外暴露了一组细粒度的服务。客户端的请求可能会涉及到一串的服务调用，如果将这些微服务都暴露给客户端，那么会增加客户端代码的复杂度
- 参考GOF设计模式中的Facade模式，将细粒度的服务组合起来提供一个粗粒度的服务，所有请求都导入一个统一的入口，那么整个服务只需要暴露一个api，对外屏蔽了服务端的实现细节，也减少了客户端与服务器的网络调用次数，这称为api gateway
- Zuul的作用
 - 动态路由
 - 监控
 - 安全
 - 认证鉴权
 - 压力测试
 - 审查
 - 服务迁移
 - 负载剪裁
 - 静态应答处理



- 创建SpringBoot项目，添加pom依赖

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

- 配置application.properties

```
server.port=10000
spring.application.name=api-gateway
eureka.client.service-url.defaultZone=http://ksit:root@127.0.0.1:7788/eureka,http://ksit:root@127.0.0.1:7799/eureka
```

- 在App类中添加@EnableZuulProxy注解，启用zuul

@SpringBootApplication

@EnableZuulProxy

public class ApiGatewayApplication {

- 启动程序后，zuul会连接到Eureka上获取到注册的服务信息，并提供网关服务，例如现在Eureka上注册有user-service和order-service，那么可以通过http://zuulip:port/<service-name>/<url>来访问微服务

Instances currently registered with Eureka			
Application	AMIs	Avallability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - localhost:api-gateway:10000
EUREKA-SERVER-1	n/a (1)	(1)	UP (1) - localhost:eureka-server-1:7788
EUREKA-SERVER-2	n/a (1)	(1)	UP (1) - localhost:eureka-server-2:7799
MOVIE-SERVICE-CONSUMER	n/a (1)	(1)	UP (1) - localhost:movie-service-consumer:9090
MOVIE-SERVICE-PROVIDER	n/a (1)	(1)	UP (1) - localhost:movie-service-provider

- 如上图，可以访问 http://localhost:10000/movie-service-consumer/movie/1，来获取微服务提供的数据

- Zuul 默认提供了/routes路径可以方便的看到zuul当前映射的路由
- 访问/routes会返回401错误，应配置application.properties

#禁用路由安全设置

endpoints.routes.sensitive=false



```
{  
  /eureka-server-1/**: "eureka-server-1",  
  /eureka-server-2/**: "eureka-server-2",  
  /movie-service-consumer/**: "movie-service-consumer",  
  /movie-service-provider/**: "movie-service-provider"  
}
```

- 可以通过配置endpoints.routes.enabled来关掉/routes端点

#禁用/routes路由断点

endpoints.routes.enabled=false

- Zuul默认会自动从Eureka中读取所有服务的name，并作为路由名称
- 通过zuul.routes.<server-name>=path来自定义微服务的访问路径

#自定义路由路径（movie-service-consumer服务就会映射到/m/**）

```
zuul.routes.movie-service-consumer=/m/**
```

或

#自定义路由路径（movie-service-consumer服务就会映射到/m/**,其中movieService为自定义名称）

```
zuul.routes.movieService.serviceId=movie-service-consumer
```

```
zuul.routes.movieService.path=/m/**
```

- 忽略服务

#忽略微服务,服务名之间使用逗号分割，设置为*表示忽略所有微服务

```
zuul.ignored-services=eureka-server-1,eureka-server-2,api-gateway
```


- 前缀

```
zuul.routes.movie-service-consumer=/movie/**
```

```
zuul.routes.movie-service-consumer.stripPrefix=false
```

当访问zuul的/movie/100路径，请求将会映射到movie-service-consumer的/movie/100

- 使用zuul.prefix定义公共前缀

```
zuul.prefix=/api
```

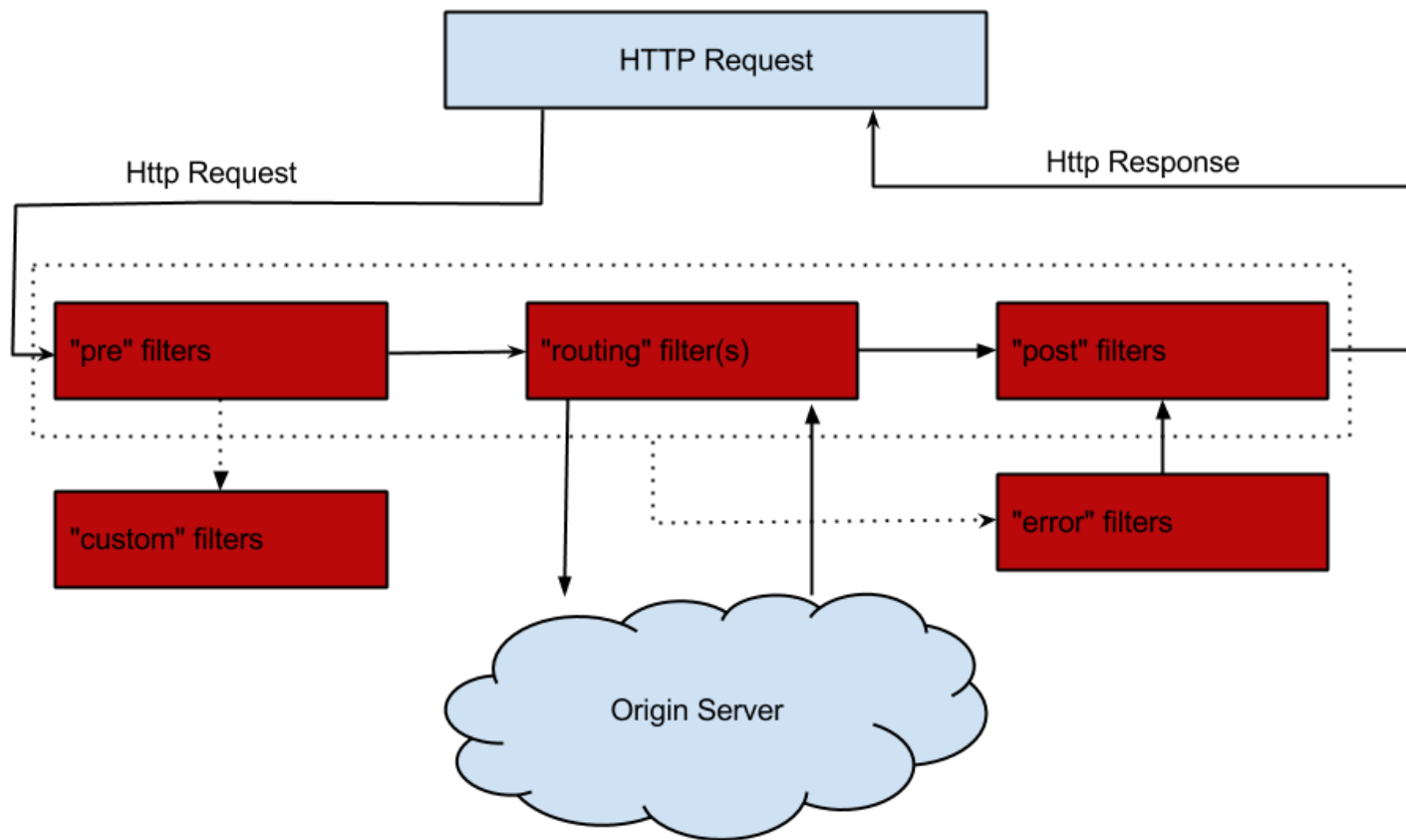
```
zuul.routes.movie-service-consumer=/movie/**
```

```
zuul.routes.movie-service-consumer.stripPrefix=false
```

- 当访问zuul的/api/movie/100路径，请求将会映射到movie-service-consumer的/movie/100

Zuul请求生命周期

凯盛软件



- 第一个阶段pre，在这里它会被pre类型的过滤器进行处理，该类型的过滤器主要目的是在进行请求路由之前做一些前置加工，比如请求的校验等
- 第二个阶段routing，也就是之前说的路由请求转发阶段，请求将会被routing类型过滤器处理，这里的具体处理内容就是将外部请求转发到具体服务实例上去的过程，当服务实例将请求结果都返回之后，routing阶段完成
- 第三个阶段post，此时请求将会被post类型的过滤器进行处理，这些过滤器在处理的时候不仅可以获取到请求信息，还能获取到服务实例的返回信息，所以在post类型的过滤器中，我们可以对处理结果进行一些加工或转换等内容
- 还有一个特殊的阶段error，该阶段只有在上述三个阶段中发生异常的时候才会触发，但是它的最后流向还是post类型的过滤器，因为它需要通过post过滤器将最终结果返回给请求客户端

- Zuul中定义了四种过滤器类型
 - pre : 可以在请求被路由之前调用
 - routing : 在路由请求时候被调用
 - post : 在routing和error过滤器之后被调用
 - error : 处理请求时发生错误时被调用
- 自定义Zuul过滤器，只需要声明一个类并继承自com.netflix.zuul.ZuulFilter类即可。并实现如下抽象方法
 - public String filterType() : 返回一个字符串代表过滤器的类型，例如 pre
 - public int filterOrder() : 通过int值来定义过滤器的执行顺序
 - public boolean shouldFilter() : 返回一个boolean类型来判断该过滤器是否要执行，所以通过此函数可实现过滤器的开关，一般返回为true
 - public Object run() : 过滤器的具体业务逻辑

```
public class AccessFilter extends ZuulFilter {  
    @Override  
    public String filterType() {  
        return FilterConstants.PRE_TYPE;  
    }  
  
    @Override  
    public int filterOrder() {  
        return 0;  
    }  
  
    @Override  
    public boolean shouldFilter() {  
        return true;  
    }  
}
```

@Override

```
public Object run() {  
    //RequestContext封装了HttpServletRequest和HttpServletResponse等对象  
    RequestContext requestContext = RequestContext.getCurrentContext();  
  
    HttpServletRequest request = requestContext.getRequest();  
  
    String accessToken = request.getParameter("accessToken");  
    if(accessToken == null) {  
        //拒绝路由  
        requestContext.setSendZuulResponse(false);  
        //设置相应码为401  
        requestContext.setResponseStatusCode(401);  
    }  
    return null;  
}
```

- 将过滤器放入Spring容器，就会自动加入Zuul的过滤器中

```
@SpringBootApplication
```

```
@EnableZuulProxy
```

```
public class ApiGatewayApplication {
```

```
    @Bean
```

```
    public AccessFilter accessFilter() {
```

```
        return new AccessFilter();
```

```
    }
```

Zuul自带的过滤器

凯盛软件

顺序	过滤器	功能	
-3	ServletDetectionFilter	标记处理Servlet的类型	pre
-2	Servlet30WrapperFilter	包装HttpServletRequest请求	
-1	FormBodyWrapperFilter	包装请求体	
1	DebugFilter	标记调试标志	
5	PreDecorationFilter	处理请求上下文供后续使用	
10	RibbonRoutingFilter	serviceld请求转发	route
100	SimpleHostRoutingFilter	url请求转发	
500	SendForwardFilter	forward请求转发	
0	SendErrorFilter	处理有错误的请求响应	post
1000	SendResponseFilter	处理正常处理的请求响应	

- Zuul中默认已经默认整合了Hystrix，访问Zuul中/hystrix.stream可以看到Hystrix监控数据
- 如果此时关掉服务提供者，那么系统会出现500错误，因为默认没有提供回退机制
- 提供Zuul的回退机制需要声明一个类，实现org.springframework.cloud.netflix.zuul.filters.route.ZuulFallbackProvider接口并放入Spring容器
 - public String getRoute(): 容错的服务名称，*表示所有服务
 - public ClientHttpResponse fallbackResponse(): 容错的结果

```
@Component
public class MovieServiceFallback implements ZuulFallbackProvider {

    /**
     * 容错的服务名称，*表示所有服务。
     */
    @Override
    public String getRoute() {
        return "movie-service-consumer";
    }
}
```

```
@Override
public ClientHttpResponse fallbackResponse() {
    return new ClientHttpResponse() {
        @Override
        public HttpStatus getStatusCode() throws IOException {
            //HTTP 状态码
            return HttpStatus.OK;
        }

        @Override
        public int getRawStatusCode() throws IOException {
            //数字类型的状态码, 当前为200
            return getStatusCode().value();
        }

        @Override
        public String getStatusText() throws IOException {
            //状态文本, 当前为OK
            return this.getStatusCode().getReasonPhrase();
        }
    }
}
```

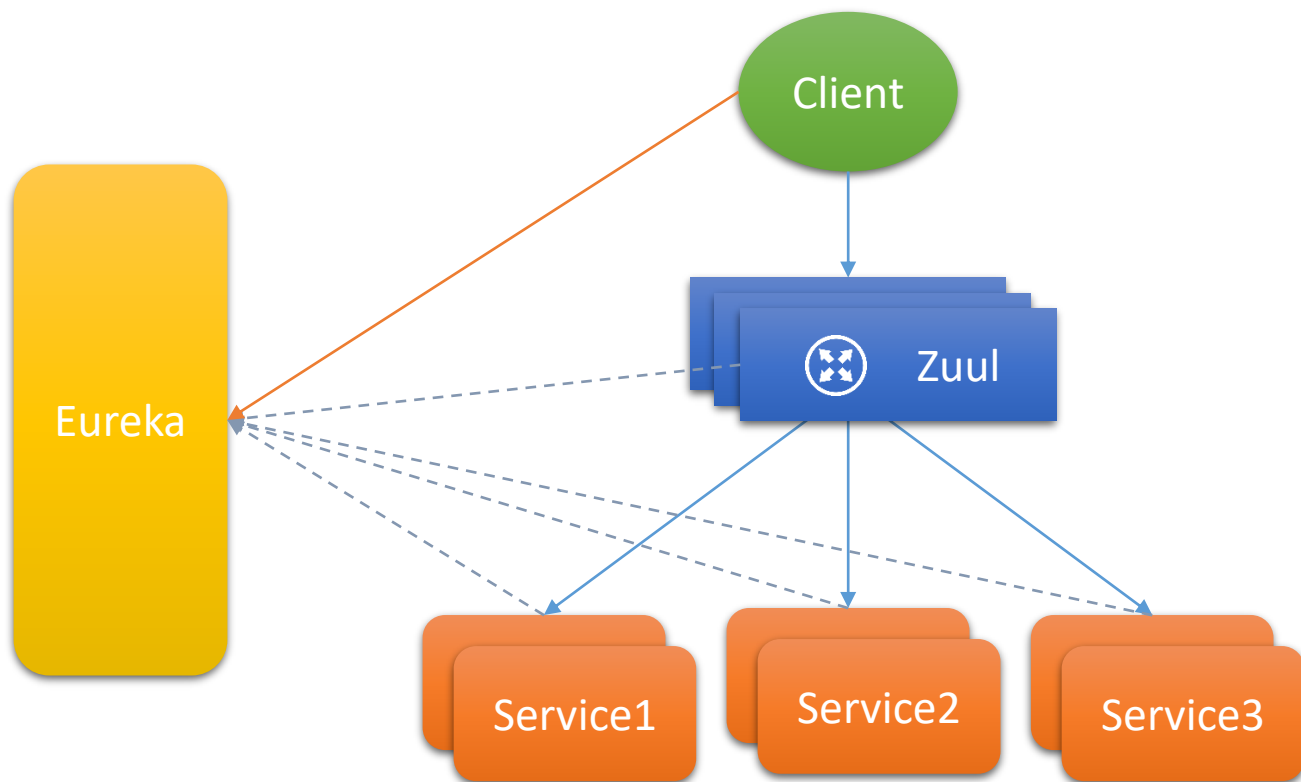
```
@Override
public void close() {

}

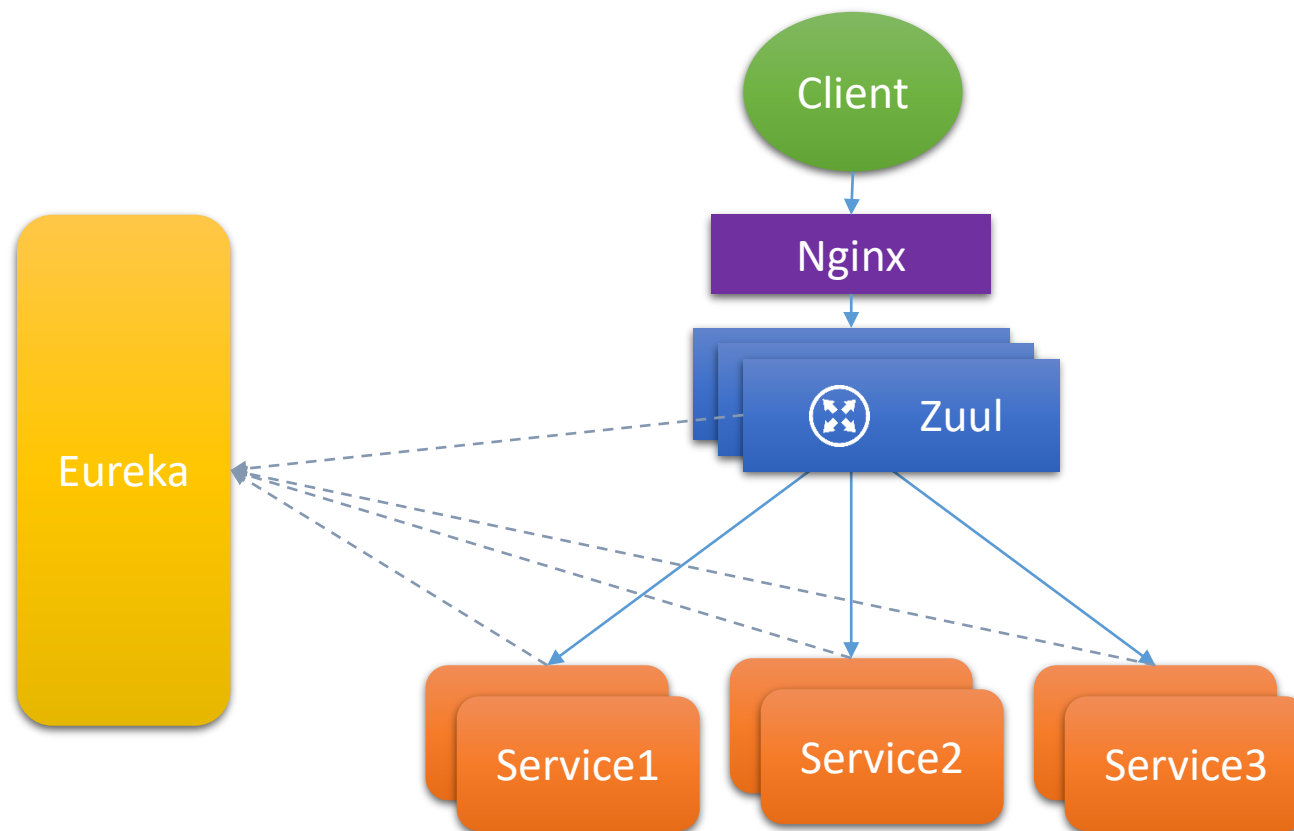
@Override
public InputStream getBody() throws IOException {
    // 响应内容
    return new ByteArrayInputStream("Movie API 暂时不可用".getBytes());
}

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

- 外部请求到后端的微服务的流量都会经过Zuul，所以Zuul的高可用可以避免单点故障

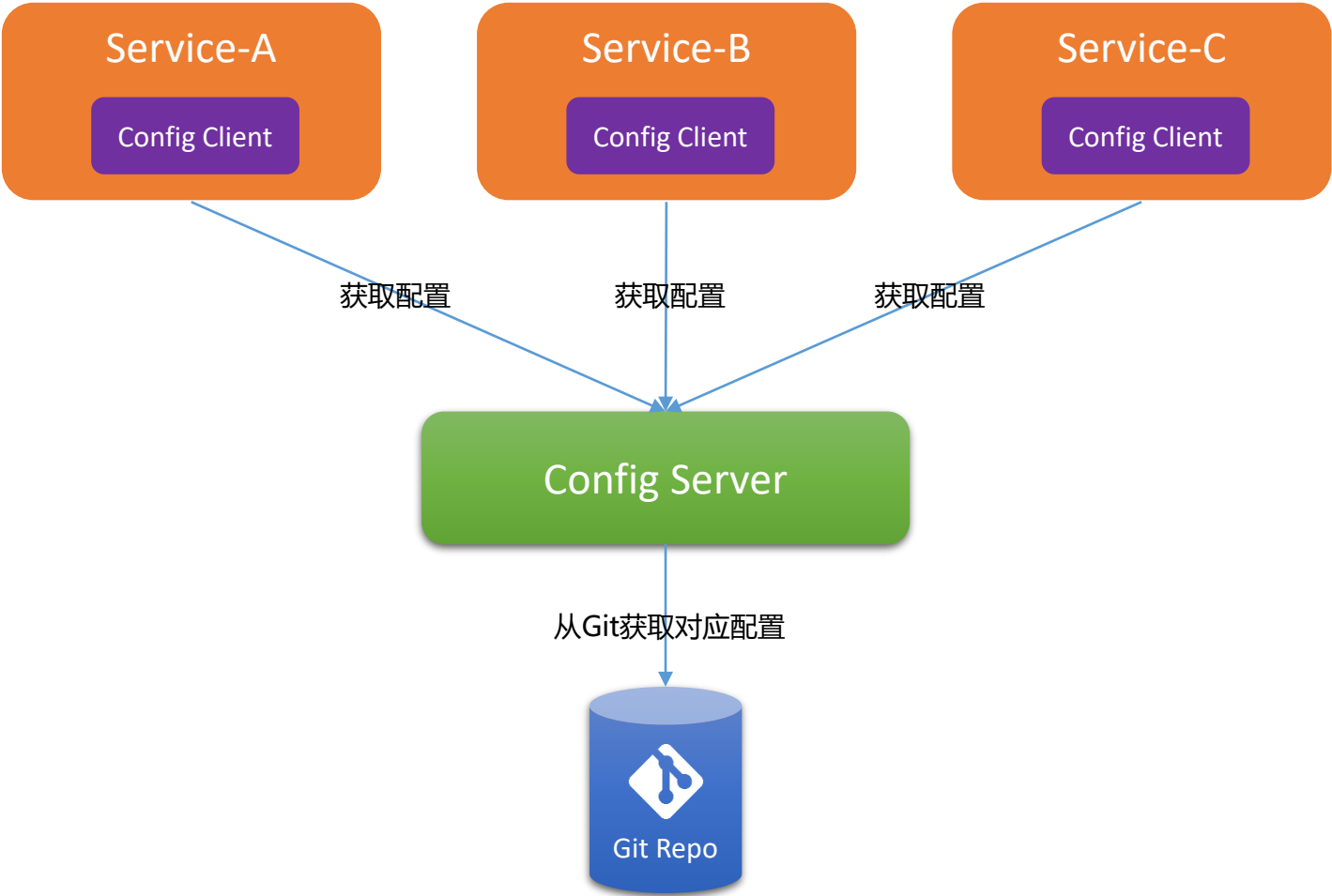


情况A: zuul客户端也注册在Eureka中，客户端自动从Eureka中获取Zuul服务列表，通过Ribbon进行负载均衡



情况B：zuul客户端未注册在Eureka中，例如移动端。该情况下就需要借助其他的负载均衡设备来实现Zuul高可用，例如Nginx

- Spring Cloud Config 为分布式系统外部化配置提供了服务器端和客户端的支持，它包括config server端和config client端两部分
- Config server端是一个可以横向扩展、集中式的配置服务器，它用于集中管理应用程序各个环境下的配置，默认使用Git存储配置内容
- Config client 是config server的客户端，用于操作存储在server中的配置属性
- 优势
 - 集中管理配置
 - 不同环境不同配置
 - 运行期间可以动态调整
 - 配置修改后可以自动更新



- 创建SpringBoot项目，并添加pom依赖

```
<dependency>
```

```
    <groupId>org.springframework.cloud</groupId>
```

```
    <artifactId>spring-cloud-config-server</artifactId>
```

```
</dependency>
```

- 在App上添加注解@EnableConfigServer，开启ConfigServer

```
@SpringBootApplication
```

```
@EnableConfigServer
```

```
public class ConfigServerApplication {
```


- 配置application.properties文件

#git仓库地址

spring.cloud.config.server.git.uri=http://192.168.135.111/root/cloudconfig.git

#git仓库账号

spring.cloud.config.server.git.username=root

#git仓库密码

spring.cloud.config.server.git.password=rootroot


- 在git仓库中推送配置文件，配置文件内容为myName=Rose

master

cloudconfig /

History



Find file



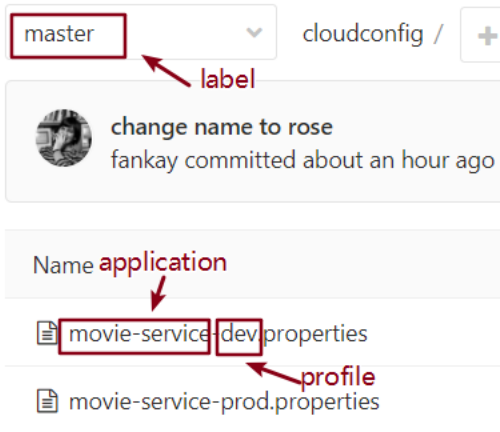
change name to rose

fankay committed about an hour ago

997bdc80

Name	Last commit	Last update
 movie-service-dev.properties	change name to rose	about an hour ago
 movie-service-prod.properties	修改内容	about 2 hours ago

- 启动服务后，会自带访问端点
 - `/{{application}}/{{profile}}/{{label}}`
 - `/{{application}}-{{profile}}.yaml`
 - `/{{label}}/{{application}}-{{profile}}.yaml`
 - `/{{application}}-{{profile}}.properties`
 - `/{{label}}/{{application}}-{{profile}}.properties`
 - `{{application}}`表示微服务的名称，`{{label}}`表示Git对应的分支，默认为master，`{{profile}}`表示配置文件的profile
 - 例如 `http://localhost:20000/movie-service-dev.properties` 或 `http://localhost:20000/movie-service/dev`



```

< > ↻ ⓘ localhost:20000/movie-service/dev
{
  name: "movie-service",
  profiles: [
    "dev"
  ],
  label: null,
  version: "997bdc80b952fe0d5c68a8fb18562b055f13a33f",
  state: null,
  propertySources: [
    {
      name: "https://github.com/fankay/springcloudconfig.git/movie-service-dev.properties",
      source: {
        myName: "Rose"
      }
    }
  ]
}

```

- 在微服务的项目中添加pom依赖

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

- 创建配置文件bootstrap.properties，当存在bootstrap.properties.yml文件时，应用程序启动时会先加载该文件，然后在加载application.properties文件

#应用的名称，需要和{application}一致

```
spring.application.name=movie-service
```

#ConfigServer端的地址

```
spring.cloud.config.uri=http://localhost:20000
```

#{profile}的名称

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

#{label}分支的名称

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

- 在需要的地方注入git repo中配置文件的key值

```
@Value("${myName}")
```

```
private String myName;
```

```
@GetMapping("/hello")
```

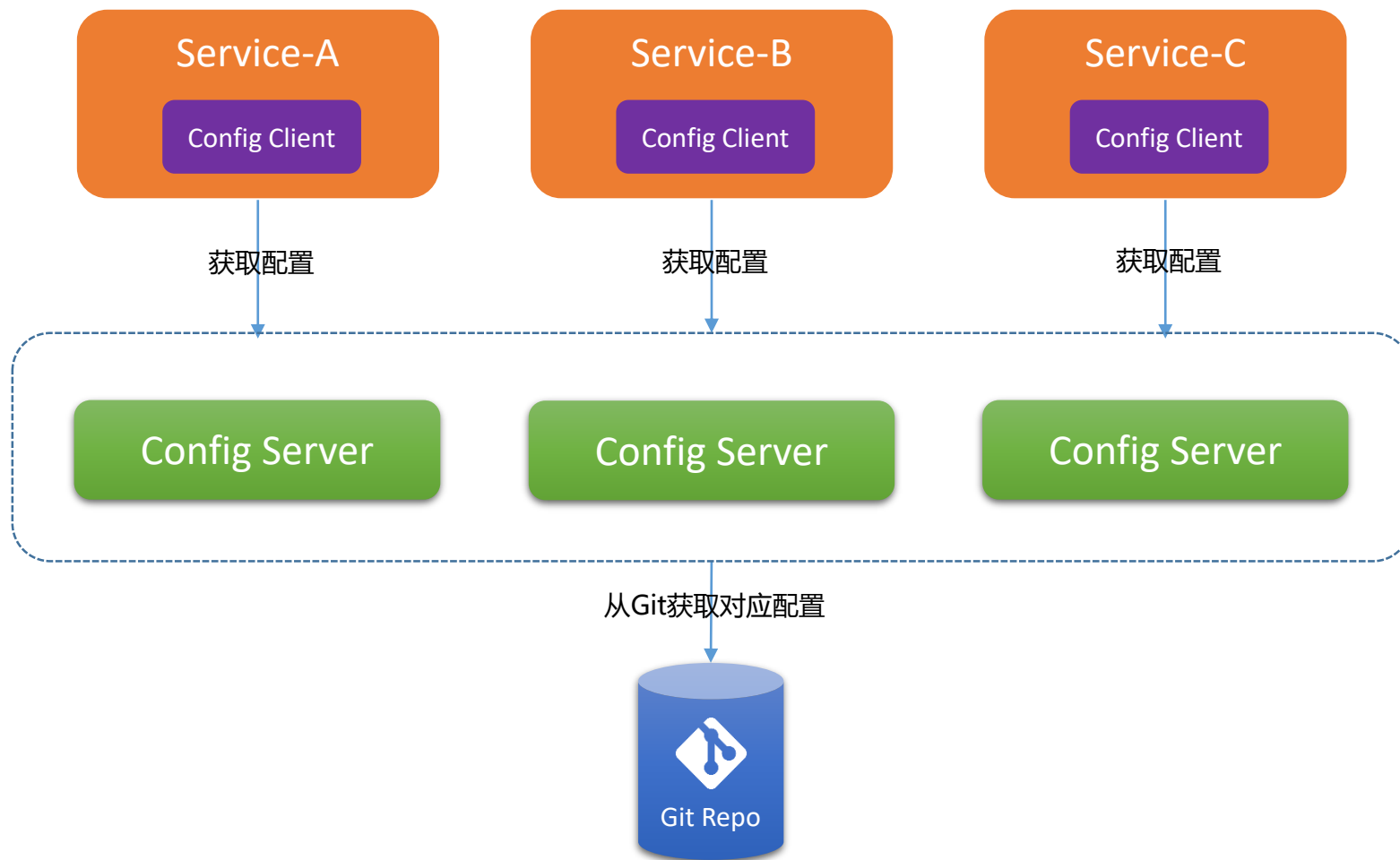
```
public String sayHello() {
```

```
    return "Hi, " + myName;
```

```
}
```

高可用的Config Server

凯盛软件



- 实现高可用的Config Server的做法就是将Config Server配置到Eureka中
- 在Config Server中添加pom依赖

```
<dependency>
```

```
    <groupId>org.springframework.cloud</groupId>
```

```
    <artifactId>spring-cloud-starter-eureka</artifactId>
```

```
</dependency>
```

- 在App中添加@EnableDiscoveryClient注解，开启服务发现

```
@SpringBootApplication
```

```
@EnableConfigServer
```

```
@EnableDiscoveryClient
```

```
public class ConfigServerApplication {
```

- 在application.properties中配置Eureka的地址

```
eureka.client.service-url.defaultZone=http://ksit:root@127.0.0.1:7788/eureka,http://ksit:root@127.0.0.1:7799/eureka
```

- 修改bootstrap.properties配置文件

#应用的名称, 需要和{application}一致

spring.application.name=movie-service

#Eureka地址

eureka.client.service-url.defaultZone=http://ksit:root@127.0.0.1:7788/eureka,http://ksit:root@127.0.0.1:7799/eureka

#开始服务发现

spring.cloud.config.discovery.enabled=true

#ConfigServer端的服务名称

spring.cloud.config.discovery.service-id=server-config

#{profile}的名称

spring.cloud.config.profile=dev

#{label}分支的名称

spring.cloud.config.label=master

- 如果修改了git repo中的配置文件内容，微服务是不会自动获取最新配置的
- 可以通过post方式访问/refresh的形式，让微服务来刷新配置信息
- 在微服务端添加pom依赖

`<dependency>`

`<groupId>org.springframework.boot</groupId>`

`<artifactId>spring-boot-starter-actuator</artifactId>`

`</dependency>`

- 修改bootstrap.properties文件

`#禁用/refresh端点的权限`

`endpoints.refresh.sensitive=false`

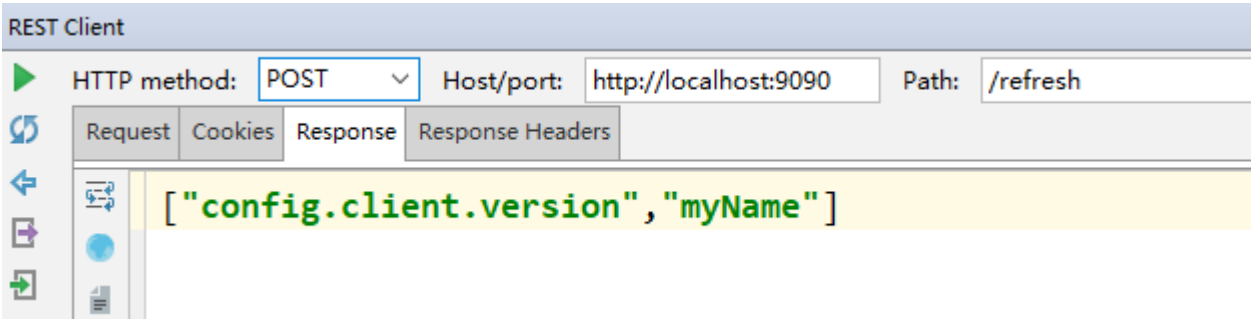
- 在控制器上添加@RefreshScope注解

```
@RestController
@RefreshScope
public class ApiController {

    @Value("${myName}")
    private String myName;

    @GetMapping("/hello")
    public String sayHello() {
        return "Hi, " + myName;
    }
}
```

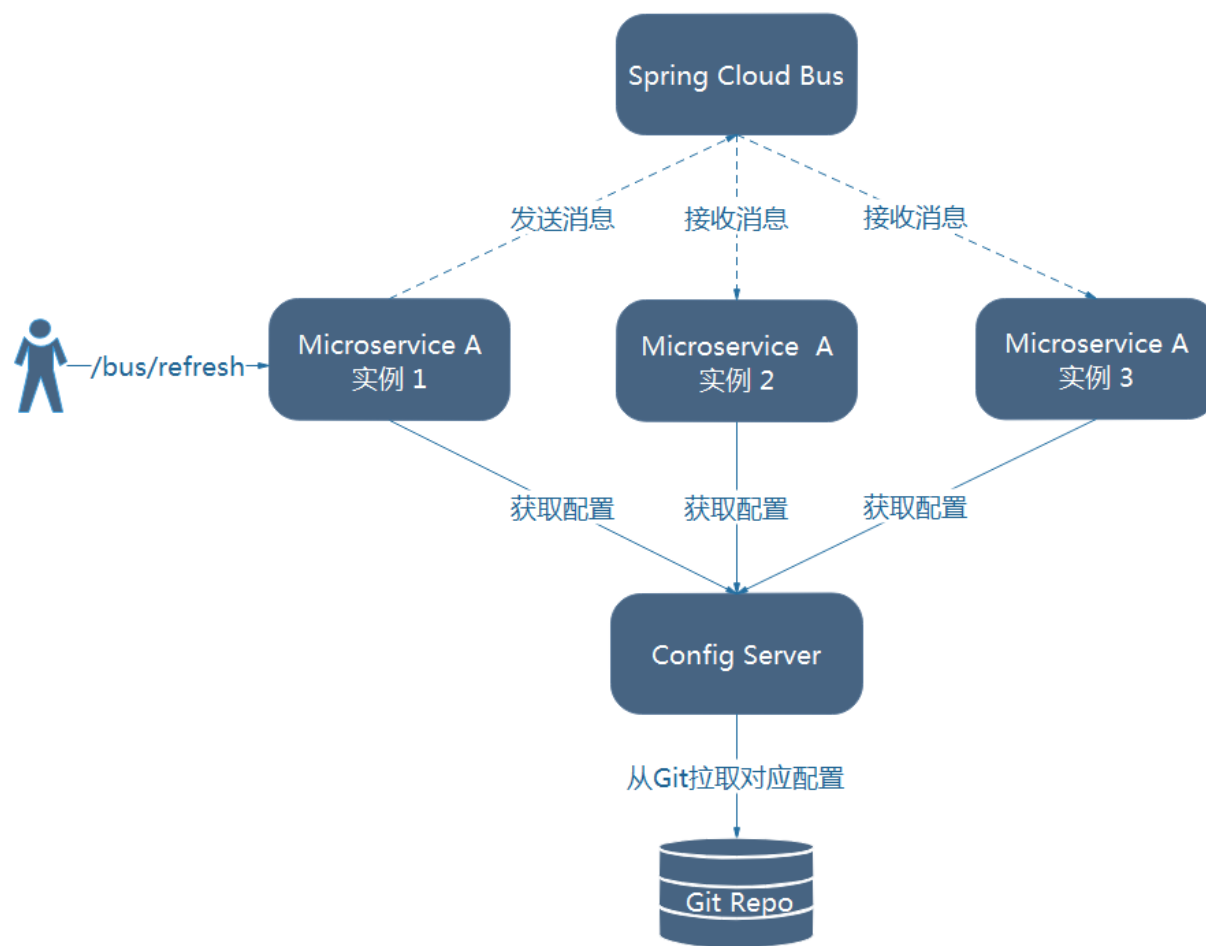
- 修改git repo中的内容
- 通过post方式请求http://ip:port/refresh



使用Spring Cloud Bus自动刷新配置

凯盛软件

Spring Cloud Bus 使用轻量级的消息代理（例如RabbitMQ、Kafuka等）连接分布式系统的节点，当发出类似配置更改的系统指令时可以以广播的形式传播状态的更改。



1. 下载RabbitMQ 的安装文件
 - https://github.com/rabbitmq/rabbitmq-server/releases/download/v3.7.1/rabbitmq-server_3.7.1-1_all.deb
2. 下载Erlang的安装文件
 - http://packages.erlang-solutions.com/site/esl/esl-erlang/FLAVOUR_1_general/esl-erlang_20.1-1~ubuntu~xenial_amd64.deb
3. 安装Erlang
 - `dpkg -i esl-erlang_20.1.7-1~ubuntu~xenial_amd64.deb`
4. 安装RabbitMQ
 - `dpkg -i rabbitmq-server_3.7.1-1_all.deb`
5. 启动RabbitMQ
 - `service rabbitmq-server start`
6. 安装RabbitMQ Management
 - `rabbitmq-plugins enable rabbitmq_management`
7. 添加管理账号
 - `rabbitmqctl add_user root root`
 - `rabbitmqctl set_user_tags root administrator`
 - `rabbitmqctl set_permissions -p / root ".*" ".*" ".*"`
8. 登录 <http://192.168.135.116:15672/>

- 添加依赖Spring Cloud Bus的依赖

`<dependency>`

`<groupId>org.springframework.cloud</groupId>`

`<artifactId>spring-cloud-starter-bus-amqp</artifactId>`

`</dependency>`

- 在application.properties中配置RabbitMQ

`spring.rabbitmq.host=192.168.135.116`

`spring.rabbitmq.port=5672`

`spring.rabbitmq.username=root`

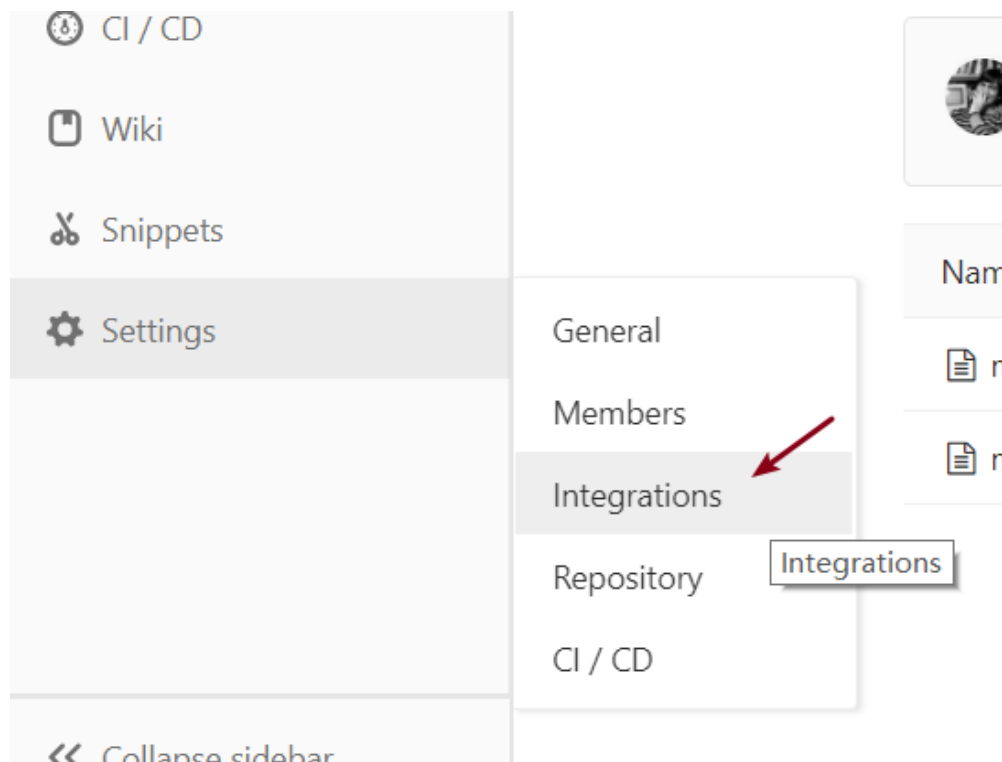
`spring.rabbitmq.password=root`

- 禁用权限认证 `endpoints.bus.sensitive=false`
- 启动服务后，以post请求访问/bus/refresh，就可以刷新配置了

配置Git仓库的Web Hooks

凯盛软件

- 在Gitlab仓库中选择Settings/Integrations



- 填写WebHooks的URL为http://ip:port/bus/refresh，Trigger的方式为Push event，这样当发生push事件时，Gitlab就是请求/bus/refresh路径，进行自动刷新

Integrations

Webhooks can be used for binding events when something is happening within the project.

URL

http://192.168.135.1:9090/bus/refresh

Secret Token

Use this token to validate received payloads. It will be sent with the request in the X-Gitlab-Token HTTP header.

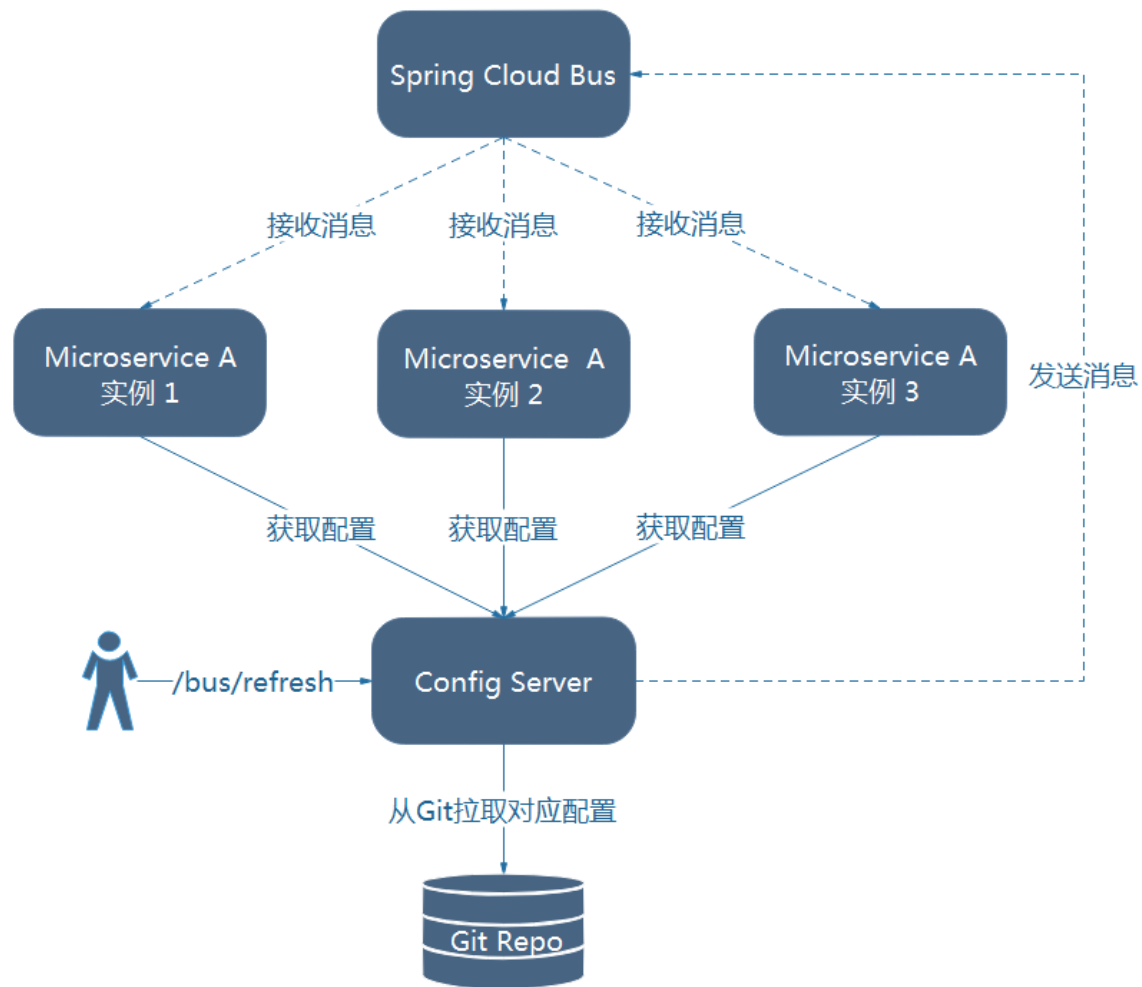
Trigger

☒ **Push events**

This URL will be triggered by a push to the repository

将刷新定位到配置服务器上

凯盛软件



- 在配置服务器的项目中添加依赖

```
<dependency>
```

```
  <groupId>org.springframework.cloud</groupId>
```

```
  <artifactId>spring-cloud-starter-bus-amqp</artifactId>
```

```
</dependency>
```

```
<dependency>
```

```
  <groupId>org.springframework.boot</groupId>
```

```
  <artifactId>spring-boot-starter-actuator</artifactId>
```

```
</dependency>
```


- 修改application.properties

```
spring.rabbitmq.host=192.168.135.116
```

```
spring.rabbitmq.port=5672
```

```
spring.rabbitmq.username=root
```

```
spring.rabbitmq.password=root
```

```
endpoints.refresh.sensitive=false
```

```
endpoints.bus.sensitive=false
```

- 修改webhooks的地址

Administrator > cloudconfig > Integrations

Integrations

Webhooks can be used for binding events when something is happening within the project.

URL

http://192.168.135.1:20001/bus/refresh

Secret Token

Use this token to validate received payloads. It will be sent with the request in the X-Gitlab-Token HTTP header.

Trigger

☒ Push events

This URL will be triggered by a push to the repository