HAND_TRAINING
SPRING CLOUD

文档作者:	HAND
创建日期:	November 26, 2020
更新日期:	November 26, 2020
文档编码:	
当前版本:	1.0

文档控制

变更记录

日期	作者	版本	变更参考

审核

姓名	职位

分发

拷贝编号	姓名	位置/岗位
1		
2		
3		
4		

备注:

出于文档管理的目的，如果您收到了本文档的电子版本，请打印出来并在封面的相应位置写上您的名字。

出于文档管理的目的，如果您收到了本文档的纸介质版本，请在封面写上您的名字。

目录

1. 概述

1.1. 开发环境要求

1.1.1. 环境要求

1、 Java7 及以上版本

2、 Spring Boot 1.3.7

3、 Spring Cloud Brixton.SR5

1.1.2. 开发工具

1、 IDEA

2、 Maven

2. 服务治理Eureka

2.1. 基础架构

2.1.1. 服务注册中心

1、 创建基于 Maven 的 Spring Boot 工程

包含 Spring Boot、Spring Cloud 依赖

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.3.7.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <java.version>1.8</java.version>
</properties>

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

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Brixton.SR5</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
```

```
</dependencyManagement>
```

2、 配置文件

```
server.port=1111


eureka.instance.hostname=localhost
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${server.port}/eureka/
```

3、 启动主类

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaApplication {


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


}
```

4、 配置详解

服务注册中心不注册自己

```
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

5、 服务注册中心集群

创建 application-peer1.properties，作为 peer1 服务中心配置文件

```
spring.application.name=eureka-server
server.port=1111


eureka.instance.hostname=peer1
eureka.client.serviceUrl.defaultZone=http://peer2:1112/eureka/
```

创建 application-peer2.properties，作为 peer2 服务中心配置文件

```
spring.application.name=eureka-server
server.port=1112
```

```
eureka.instance.hostname=peer2
eureka.client.serviceUrl.defaultZone=http://peer1:1111/eureka/
```

6、 注解启动

在启动主类上添加@EnableEurekaServer 注解即可

2.1.2. 服务提供者

1、 技术平台无关，只要遵循 Eureka 通信机制即可。

2、 基于第一个 Spring Boot 的 hello 工程，做如下修改：

a) 修改配置文件

```
server.port=8000
spring.application.name=hello-service

eureka.client.serviceUrl.defaultZone=http://peer1:1111/eureka/,http://peer2:11
12/eureka/
```

b) 主类添加注解

主类只需添加**@EnableDiscoveryClient** 注解即可。

2.1.3. 服务消费者

1、 服务消费的实现方式有两种：Ribbon、Feign，这里使用 Ribbon。

a) 创建基本的 Spring Boot 应用，添加 Spring Boot 和 Cloud 依赖

b) 修改配置文件

```
server.port=9000
spring.application.name=ribbon-consumer
eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

c) 修改启动主类

```
@EnableDiscoveryClient
@SpringBootApplication
public class ConsumerApplication {
   @Bean
   @LoadBalanced
   RestTemplate restTemplate(){
      return new RestTemplate();
   }
   public static void main(String[] args) {
      SpringApplication.run(ConsumerApplication.class, args);
   }
}
```

2、 添加 Controller 类

```
@RestController
public class ConsumerController {

    @Autowired
    RestTemplate restTemplate;

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

2.1.4. 测试

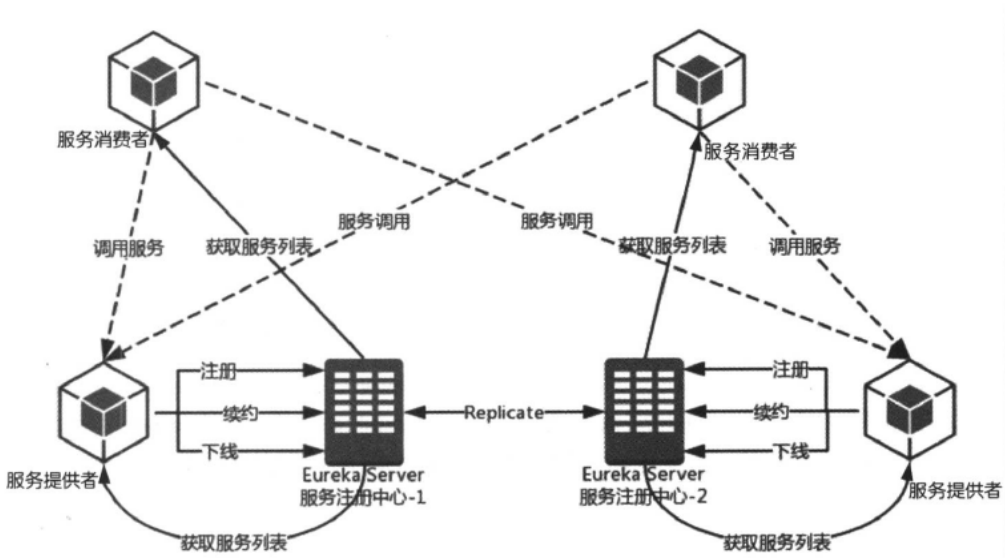
1、 启动注册中心、服务消费者、服务提供者

2、 访问 http://localhost:1111/eureka/可以看到注册的服务

3、 访问 http://localhost:9000/ribbon-consumer/可以看到返回 Hello World

2.2. 服务治理机制

2.2.1. 系统架构图



其中主要的四个通信行为：

1、 服务续约

关于服务续约有两个重要属性，我们可以关注并根据需要来进行调整：

```
eureka.instance.lease-renewal-interval-in-seconds=30
eureka.instance.lease-expiration-duration-in-seconds=90
```

eureka.instance.lease-renewal-interval-in-seconds 参数用于定义服务续约任务的调用间隔时间，默认为 30 秒。eureka.instance.lease-expiration-duration-in-seconds 参数用于定义服务失效的时间，默认为 90 秒。

2、 服务获取

获取服务是服务消费者的基础，所以必须确保 eureka.client.fetch-registry=true 参数没有被修改成 false，该值默认为 true。若希望修改缓存清单的更新时间，可以通过 eureka.client.registry-fetch-interval-seconds=30 参数进行修改，该参数默认值为 30，单位为秒。

3、 服务调用

体调用哪个实例，在 Ribbon 中会默认采用轮询的方式进行调用，从而实现客户端的负载均衡。

对于访问实例的选择，Eureka 中有 Region 和 Zone 的概念，一个 Region 中可以包含多个 Zone，每个服务客户端需要被注册到一个 Zone 中，所以每个客户端对应一个 Region 和一个 Zone。在进行服务调用的时候，优先访问同处一个 Zone 中的服务提供方，若访问不到，就访问其他的 Zone，更多关于 Region 和 Zone 的知识，我们会在后续的源码解读中介绍。

4、 自我保护

由于本地调试很容易触发注册中心的保护机制，这会使得注册中心维护的服务实例不那么准确。所以，我们在本地进行开发的时候，可以使用 eureka.server.enable-self-preservation=false 参数来关闭保护机制，以确保注册中心可以将不可用的实例正确剔除。

3. 负载均衡Ribbon

3.1. 负载均衡实现

3.1.1. 工程详解

1、 工程创建

这里使用的是上一节建立的服务消费者工程。

2、 依赖引入

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

这里引入ribbon的依赖以后，euerka会配合ribbon进行一系列的自动化配置。其中默认负载均衡的方式为轮询。

3、 主类添加 RestTemplate

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

RestTemplate是Spring框架中原有的用来发送rest请求的类。这里通过@LoadBalanced注解的修饰以后，它的请求会自动进行负载均衡。

3.1.2. 测试

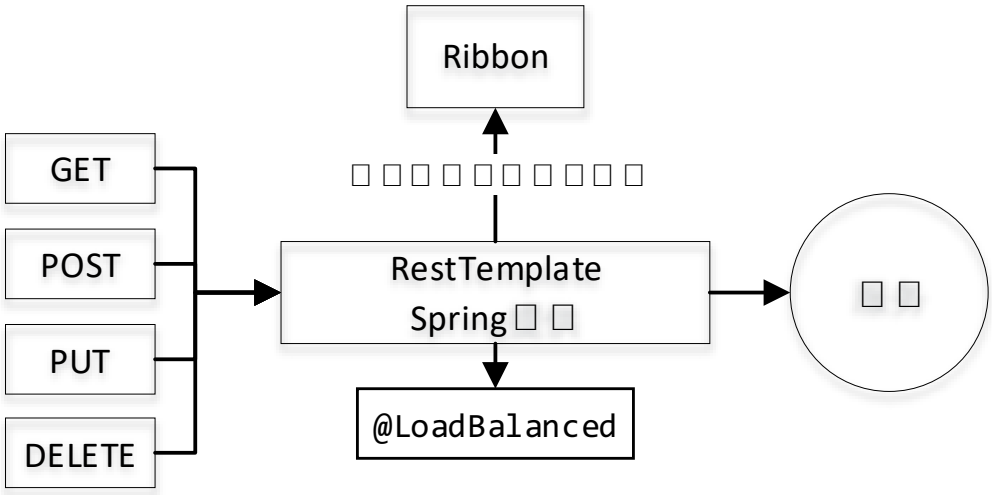
1、 启动两个 hello-service 实例、euerka 服务注册中心、ribbon-consumer 服务消费者

2、 访问 http://localhost:9000/ribbon-consumer ，进行多次请求。

3、 观察 hello-service 的两个实例的控制台输出，可以发现两个请求是被分别依次请求的

3.2. 实现方式

3.2.1. 请求方式



如图，是Ribbon实现客户端负载均衡的方式，可以看到请求方式有四种：

1、 GET 请求

a) 调用实现

i. 方法一：

getForEntity 函数
RestTemplate restTemplate = new RestTemplate(); ResponseEntity<String> responseEntity = restTemplate.getForEntity("http://USERSERVICE/user?name= { 1}" , String.c lass, "didi"); String body = responseEntity. getBody () ;

ii. 方法二：

getForObject 函数
RestTemplate restTemplate = new Rest Template() ; String result= restTemplate.getForObject(uri, String.class);

b) 方法重载

getForEntity(String url, Class responseType, Object ... urlVariables)
getForEntity(String url, Class responseType, Map urlVariables)
getForEntity(URI url, Class responseType)

注：

a. 其中 url 为请求的地址

b. responseType 为请求响应体 body 的包装类型

c. urlVariables 为 url 中的参数绑定

方法二的重载方法与方法一相同。

2、 POST 请求

a) 调用实现

除与GET相似的两种调用实现方法外，还有postForLocation。该方法实现了以POST请求提交资源， 并返回新资源的URI。

```
User user = new User("didi", 40);
URI responseURI = rest Template. postForLocation ("http:/ /USER-SERVICE/user", user);
```

b) 方法重载

postForLocation函数也实现了三种不同的重载方法：

postForLocation (String url, Object request, Object ... url Variables)
postForLocation(String url, Object request, Map urlVariables)
postForLocation(URI url, Object request)

3、 PUT 请求

a) 调用实现

```
RestTemplate restTemplate = new RestTemplate ();
Long id = 10001L;
User user = new User("didi", 40);
restTemplate.put("http://USER-SERVICE/user/{l}", user, id);
```

b) 方法重载

put(String url, Object request, Object ... urlVariables)
put(String url, Object request, Map urlVariables)
put(URI url, Object request)

4、 DELETE 请求

a) 调用实现

```
RestTemplate restTemplate = new RestTemplate();
Long id= 10001L;
restTemplate.delete("http://USER-SERVICE/user/{1}", id);
```

b) 方法重载

```
delete(String url, Object ... urlVariables)
```

delete(String url, Map urlVariables)
delete(URI url)

3.3. 自定义配置

3.3.1. 配置方法

1、 只需在 Spring Boot 应用中创建对应的实现实例就能覆盖这些默认的配置实现。

```
@Configuration
public class MyRibbonConfiguration {
    @Bean
    public IPing ribbonPing(IClientConfig config) {
        return new PingUrl();
    }
}
```

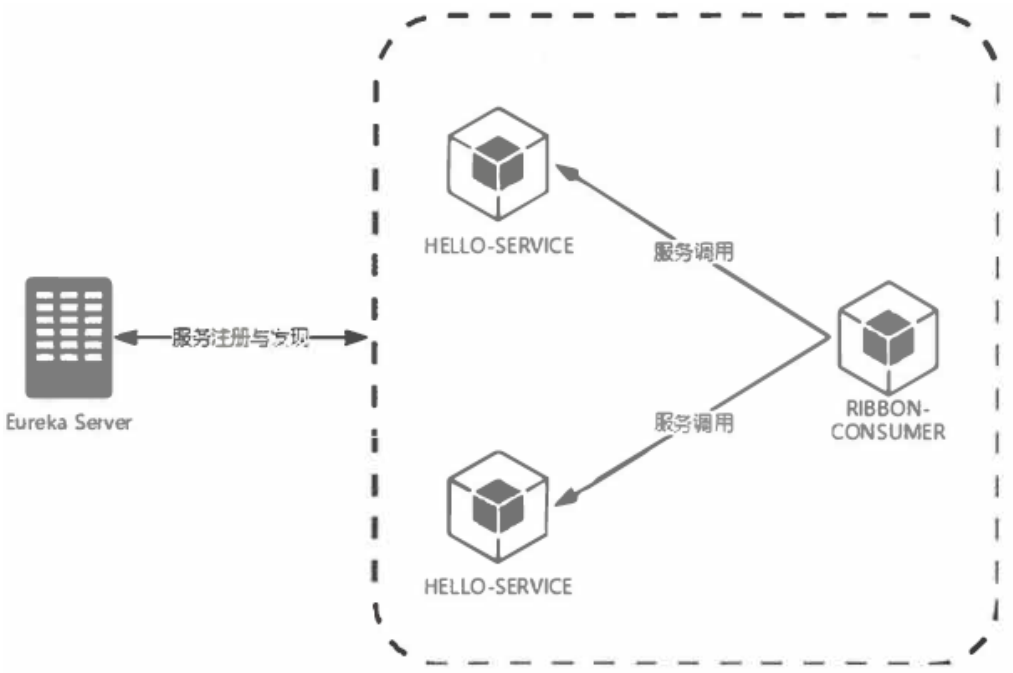
2、 指定服务使用配置

```
@Configuration
@RibbonClient(name = "hello-service",
configuration=HelloServiceConfiguration.class)
public class RibbonConfiguration {
}
```

4. 容错保护Hystrix

4.1. 工程创建

4.1.1. 工程准备



1、 经过前面的学习，现在的应有如图的服务调用关系：

a) eureka-server 工程： 服务注册中心， 端口为 1111。

b) hello-service 工程： HELLO-SERVICE 的服务单元， 两个实例启动端口分别为 8081 和 8082。

c) ribbon-consume 工程：使用 ribbon 实现的服务消费者，端口为 9000。

4.1.2. Hystrix整合

由于服务容错是基于服务创建的，所以我们这里针对ribbon-consumer工程进行整合和使用。

2、 在 ribbon-consumer 工程的 pom.xml 中添加依赖

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

3、 在主类 ConsumerApplication 中添加注解开启断路器

```
@EnableCircuitBreaker
@EnableDiscoveryClient
```

```
@SpringBootApplication
public class ConsumerApplication {

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

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

注：如果使用@SpringCloudApplication修饰主类，则默认是带有@EnableCircuitBreaker、@EnableDiscoveryClient注解的

4、 改造服务消费方式，新增 HelloService

```
@Service
public class HelloService {

    @Autowired
    RestTemplate restTemplate;

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

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

5、 修改 ConsumerController 类

```
@RestController
public class ConsumerController {

    @Autowired
```

```
    private HelloService helloService;


    @RequestMapping(value = "/ribbon-consumer", method =
RequestMethod.GET)
    public String helloConsumer(){
        return helloService.helloService();
    }


}
```

4.1.3. 测试

1、 重启 8081 端口的 Hello-Service，确保服务注册中心、两个 hello-service 以及 ribbon-consumer 均已启动。

2、 访问 http://localhost:9000/ribbon-consumer ，此时两个服务都运行正常。

3、 停止 8001 端口的 hello-service，访问 http://localhost:9000/ribbon-consumer ，可以看到，返回的结果是 helloFallback()方法定义的返回值 error，即 Hystrix 的服务回调生效。

4.1.4. 附：模拟服务阻塞

1、 前面我们是通过停止服务的方式模拟服务无响应，现在介绍另外一种方法。

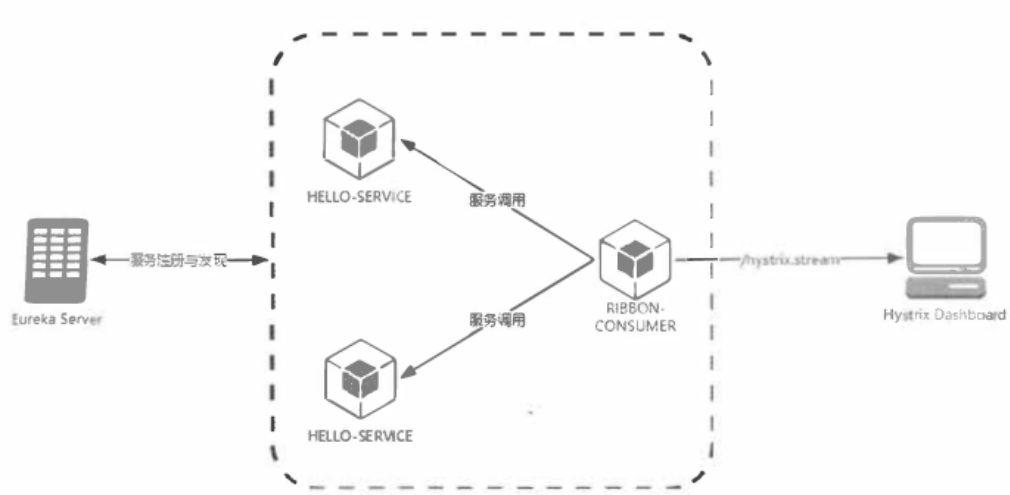
2、 对 HELLO-SERVICE 的/hello 接口做一些修改，具体如下：

```
@RequestMapping(value = "/hello", method= RequestMethod.GET)
public String hello() throws Exception {
      Serviceinstance instance= client.getLocalServiceinstance();
      //让处理线程等待几秒钟
      int sleepTime = new Random() . nextInt (3000);
      logger.info("sleepTime:" + sleepTime);
      Thread.sleep(sleepTime);
      logger.info("/hello, host:" + instance.getHost() + ", service id:"
+instance.getServiceid());
      return "Hello World";
}
```

3、 通过 Thread. sleep ()函数可让/hello 接口的处理线程不是马上返回内容，而是在阻塞几秒之后才返回内容。由于 Hystrix 默认超时时间为 2000 毫秒， 所以这里采用了 0 至 3000 的随机数以让处理过程有一定概率发生超时来触发断路器。

4.2. Hystrix仪表盘

4.2.1. 实现结构



现在构建一个Hystrix Dashboard 来对RIBBON-CONSUMER实现监控，完成后的架构如图所示。

4.2.2. 构建过程

1、 创建一个标准的 Spring Boot 工程，命名为 hystrix-dashboard 。

2、 编辑 pom. xml。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.hand</groupId>
  <artifactId>hystrix-dashboard</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
  </properties>
  <parent>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-hystrix</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-actuator</artifactId>
    </dependency>
  </dependencies>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>Brixton.SR5</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```

3、 添加主类

```
package com.hand.cloud;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;
```

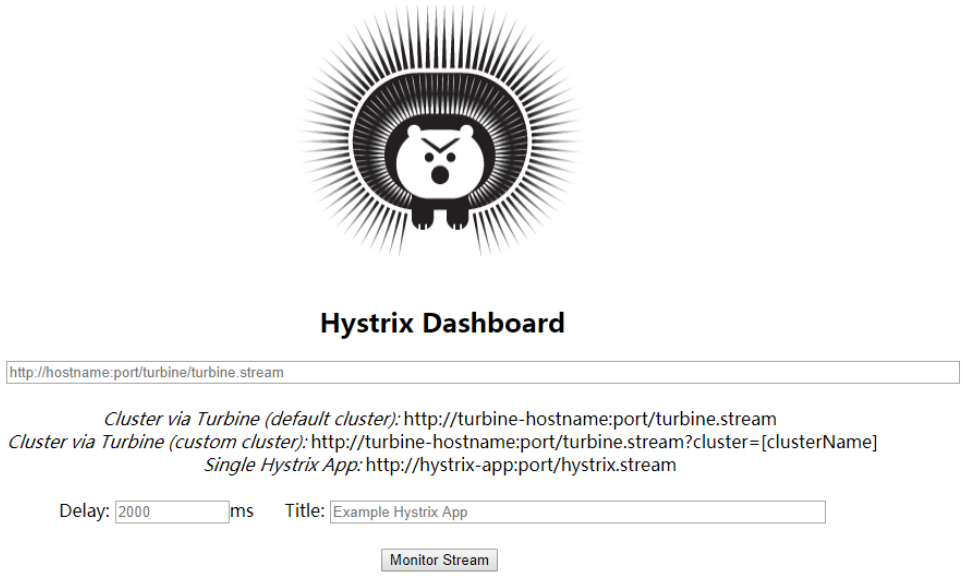
```
@EnableHystrixDashboard
@SpringBootApplication
public class HystrixApplication {

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

4、 添加配置文件

```
spring.application.name=hystrix-dashboard
server.port=2001
```

4.2.3. 测试

1、 启动应用，访问 http://localhost:2001/hystrix ，可以看到如下界面：



4.2.4. 实现监控

1、 在服务 hello-service 中添加依赖

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

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

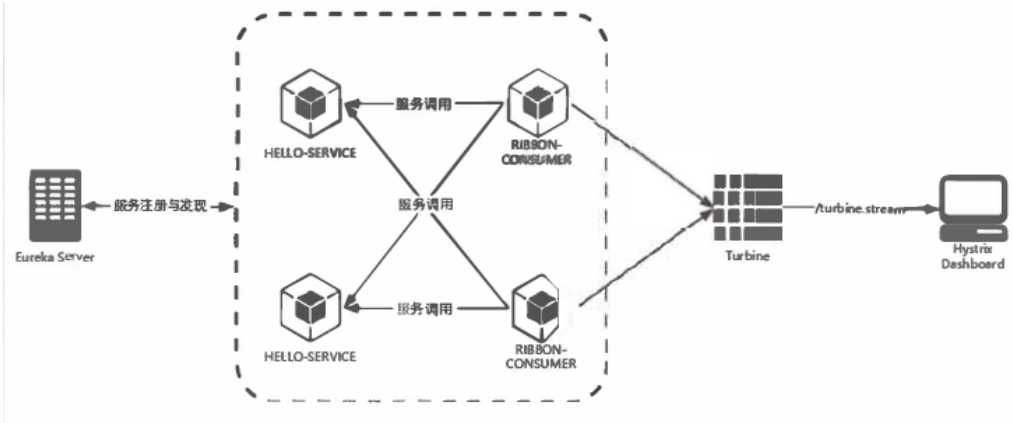
2、 确保在 hello-service 服务实例的主类中已经使用@EnableCircuitBreaker 注解， 开启了断路器功能。

3、 在 Hystrix Dashboard 的首页输入 http://localhost:9000/hystrix.strearn, 可以看到已启动对 RIBBON-CONSUMER 的监控，单击 MonitorStream 按钮， 可以看到如下页面。

4.3. Turbine集群监控

4.3.1. 监控聚合服务

1、 引入 Turbine 来聚合 RIBBON-CONSUMER 服务的监控信息， 并输出给 Hystrix Dashboard 来进行展示， 最后完成如下图所示的结构。



4.3.2. 工程创建

1、 创建一个标准的 Spring Boot 工程， 命名为 turbine。编辑 porn. xml

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
    <java.version>l.8</java.version>
</properties>
<parent>
    <groupid>org.springframework.cloud</groupid>
    <artifactid>spring -cloud -starter -parent</artifactid>
    <version>Brixton.SR5</version>
    <relativePath />
</parent>
<dependencies>
    <dependency>
        <groupid>org.springframework.cloud</groupid>
        <artifactid>spring -cloud - starter -turbine</artifactid>
    </dependency>
    <dependency>
        <groupid>org.springframework.boot</groupid>
        <artifactid>spring - boot -starter - actuator</artifactId>
    </dependency>
```

```
</dependencies>
```

2、 创建应用主类 TurbineApplication , 并使用@EnableTurbine 注解开启 Turbine。

```
@EnableTurbine
@EnableDiscoveryClient
@SpringBootApplication
public class TurbineApplication {
    public static void main(String[] args) {
        SpringApplication.run(TurbineApplica七ion.class, args);
    }
}
```

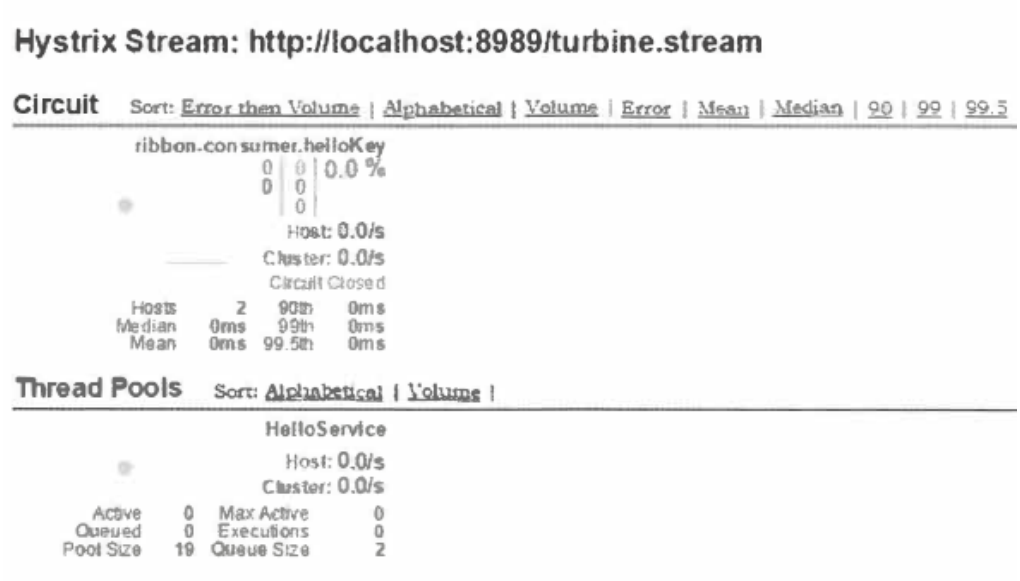
3、 在 application.properties 中加入 Eureka 和 Turbine 的相关配置

```
spring.application.name=turbine
server.port=8989
management.port=8990
eureka.client.serviceUrl.defaultZone=http://localhost:llll/eureka/
turbine. app-config=RIBBON - CONSUMER
turbine. cluster-name-expression= "default"
turbine. combine-host-port=true
```

4.3.3. 测试

1、 分别启动 eureka-server、HELLO-SERVICE、RIBBON-CONSUMER、Turbine 以及 HystrixDashboard。

2、 Hystrix Dashboard, 并开启对 http://localhost:8989/turbine.stream 的监控， 我们可以看到如下页面：



5. 声明式服务调用：Spring Cloud Feign

5.1. Spring Cloud Feign服务客户端

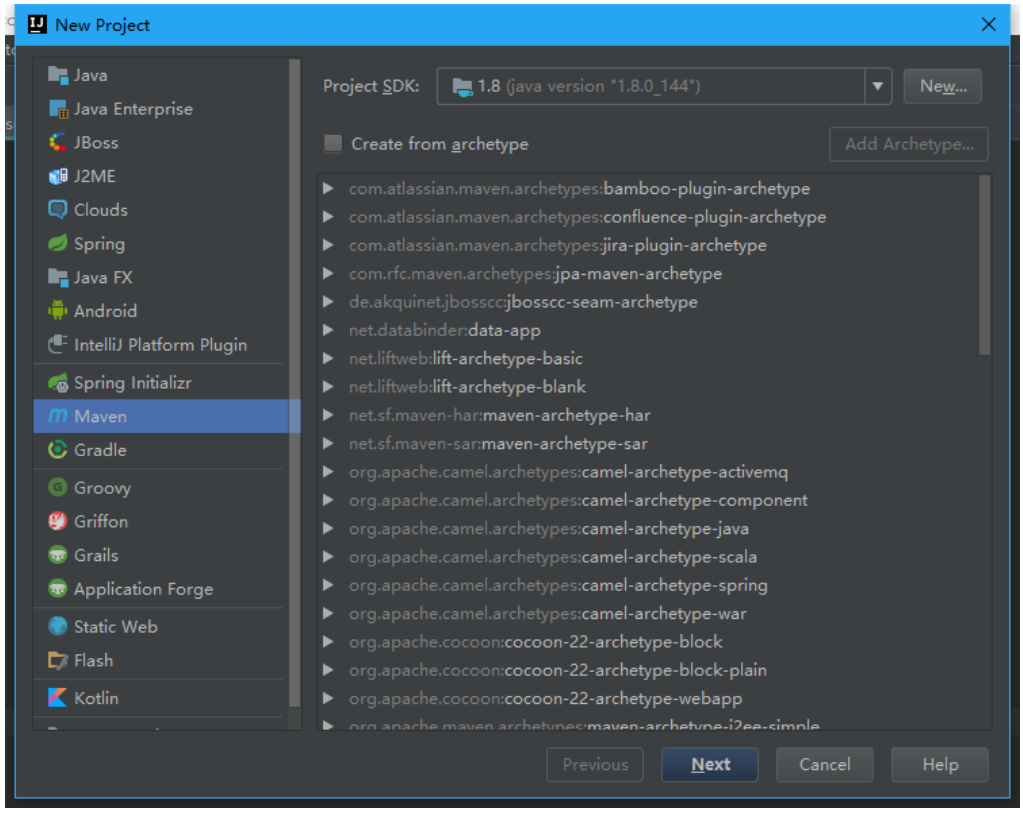
5.1.1. 概述

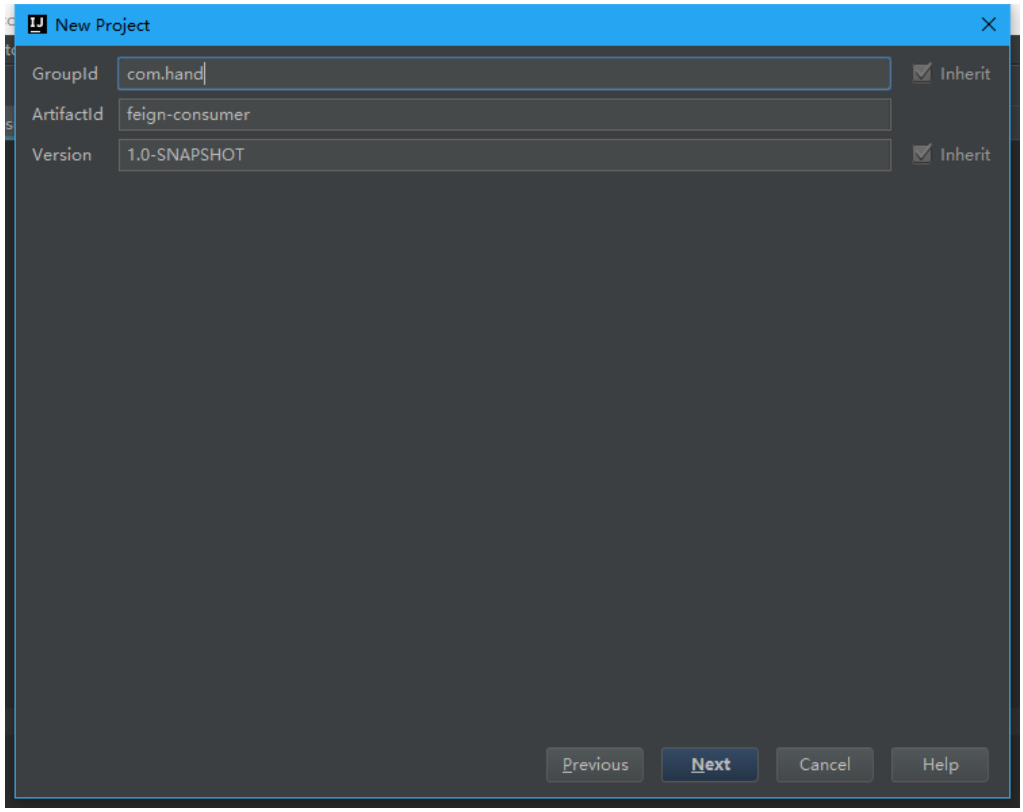
我们在使用 Spring Cloud ribbon 时，通常都会利用它对 RestTemplate 的请求拦截实现对依赖服务的接口调用， 而 RestTemplate 已经实现了对 HTTP 请求的封装处理，形成了一 套模板化的调用方法。在之前的例子中，我们只是简单介绍了 RestTemplate 调用的实现，但是在实际开发中，由于对服务依赖的调用可能不止于一处，往往 一个接口会被多处调用，所以我们通常都会针对各个微服务自行封装 一 些客户端类来包装这些依赖服务的调用。 这个时候我们会发现，由于 RestTemplate 的封装，几乎每 一个调用都是简单的模板化内容。综合上述这些情况， Spring Cloud Feign 在此基础上做了进 一 步封装，由它来帮助我们定义和实现依赖服务接口的定义。在 Spring Cloud Feign 的实现下， 我们只需创建 一个接口并用注解的方式来配置它， 即可完成对服务提供方的接口绑定，简化了在使用 Spring Cloud ribbon 时自行封装服务调用客户端的开发量。 Spring Cloud Feign 具备可插拔的注解支持，包括 Feign 注解和 JAX-RS 注解。 同时，为了适应 Spring 的广大用户，它在 Netflix Feign 的基础上扩展了对 Spring MVC 的注解支待。 这对习惯于 Spring MVC 的开发者来说， 无疑是 一 个好消息， 因为这样可以大大减少学习使用它的成本。 另外， 对于 Feign 自身的 一些主要组件， 比如编码器和解码器等， 它也以可插拔的方式提供， 在有需求的时候我们可以方便地扩展和替换它们。

注意：一些配置修改：由于6，7，8 章作者与之前作者不是同一人，一些应用端口使用可能不同，注册中心端口为：1110

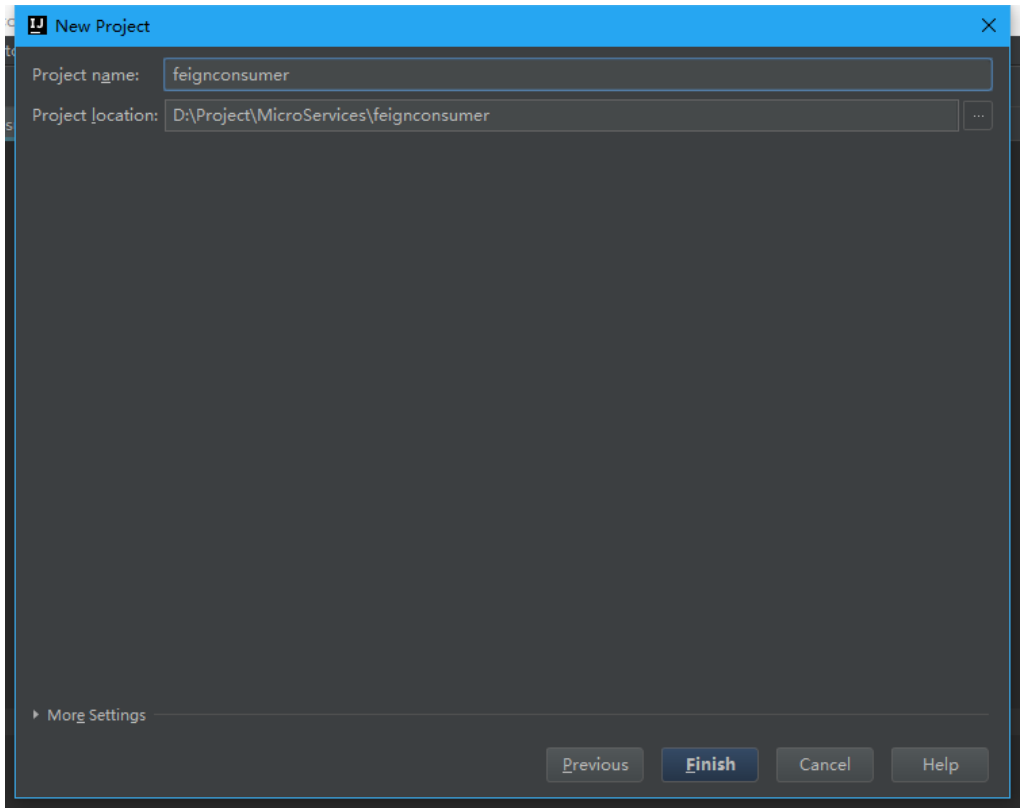
5.1.2. 创建Spring Cloud Feign 在服务客户端

1、 新建 Spring Boot 项目,下一步

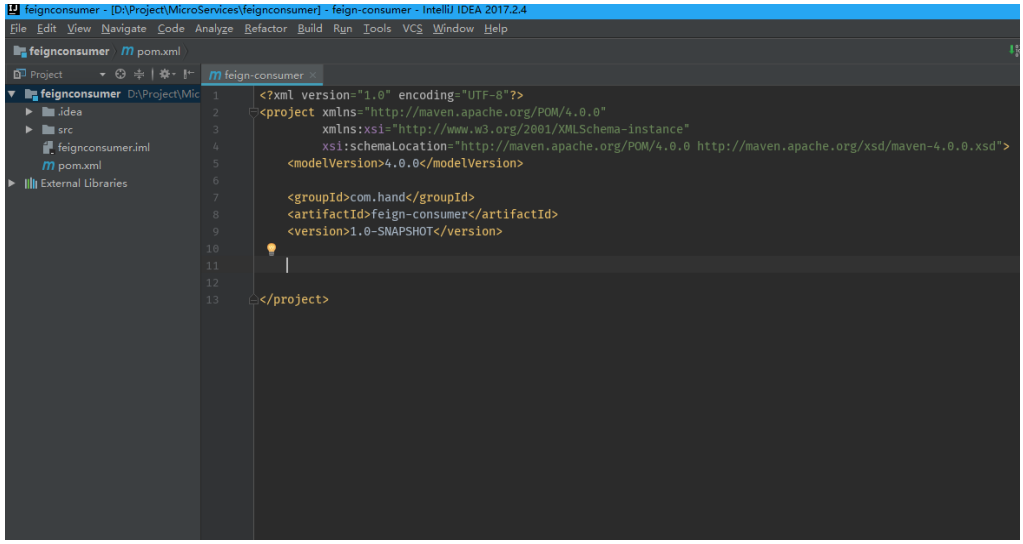




2、 输入 GroupId 和 ArtifactId,完成



3、 完成，会自动打开新建项目

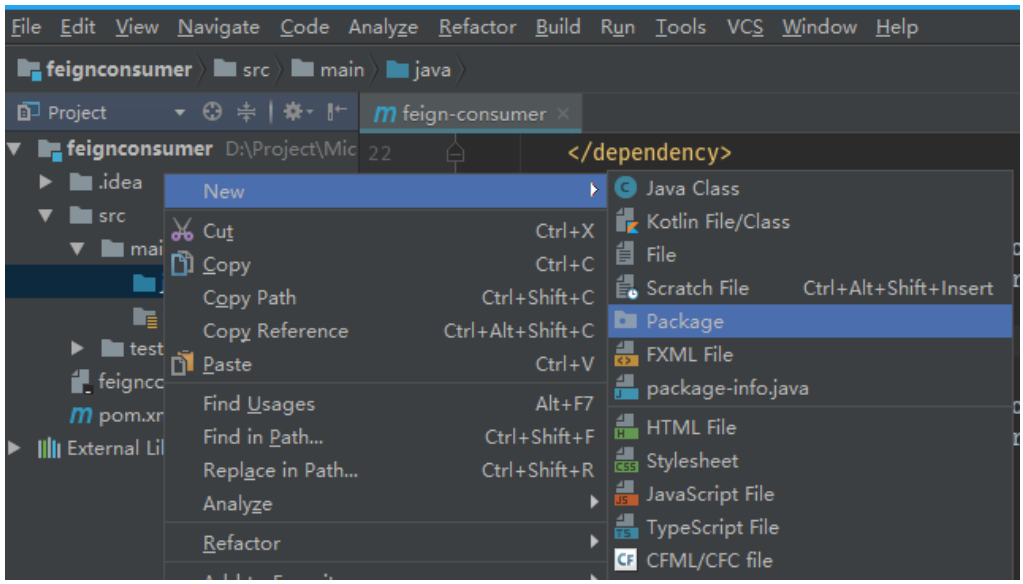


4、 在 Pom.xml 加入以下依赖

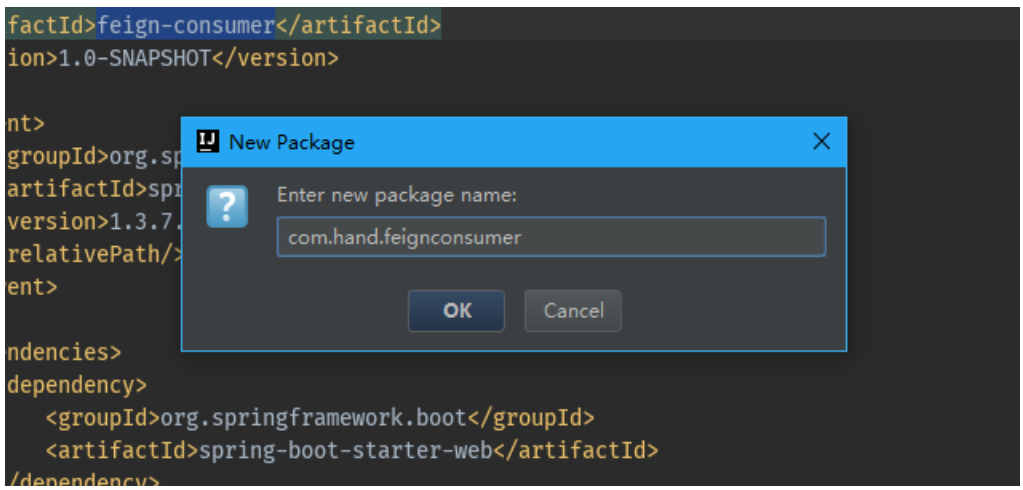
```
<parent>

<groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>1.3.7.RELEASE</version>
   <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencies>
   <dependency>

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

   <dependency>

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

   <dependency>

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

<dependencyManagement>
```

```
                                    <dependencies>
                                      <dependency>

                                    <groupId>org.springframework.cloud</group
                                    Id>
                                          <artifactId>spring-cloud-
                                    dependencies</artifactId>
        <version>Brixton.SR5</version>
                                     <type>pom</type>
                                          <scope>import</scope>
                                       </dependency>
                                     </dependencies>
</dependencyManagement>
```

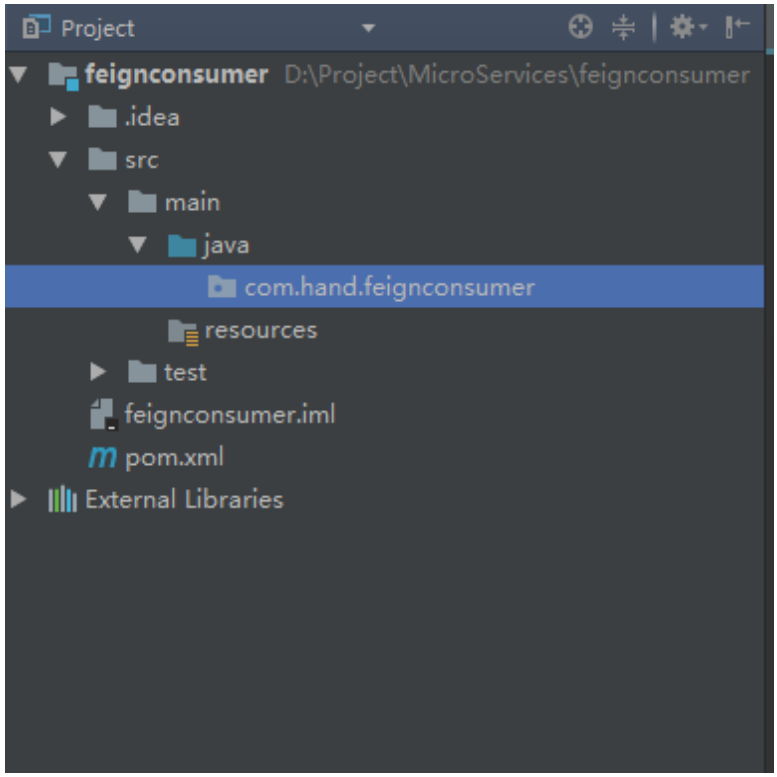
5、 右击 main 文件夹下的 java,new->package



6、 输入包名：com.hand.feignconsumer

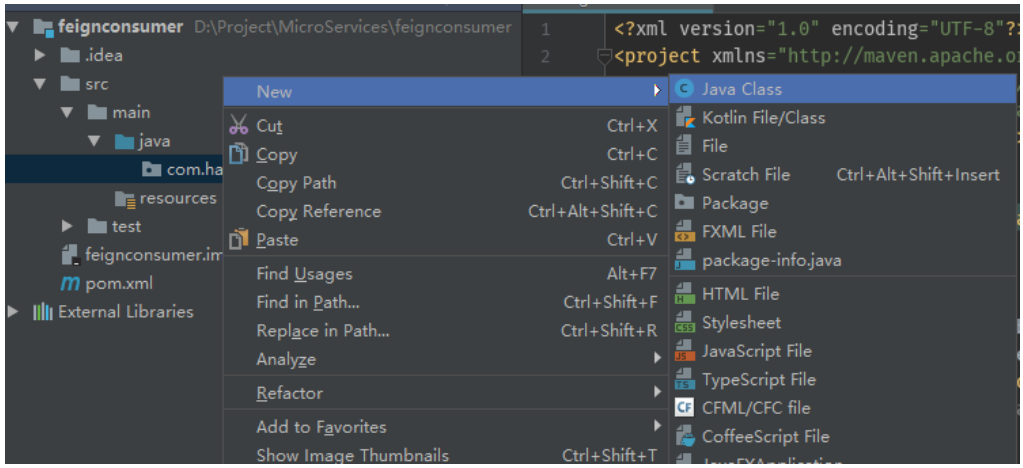


7、 生成 Java 包

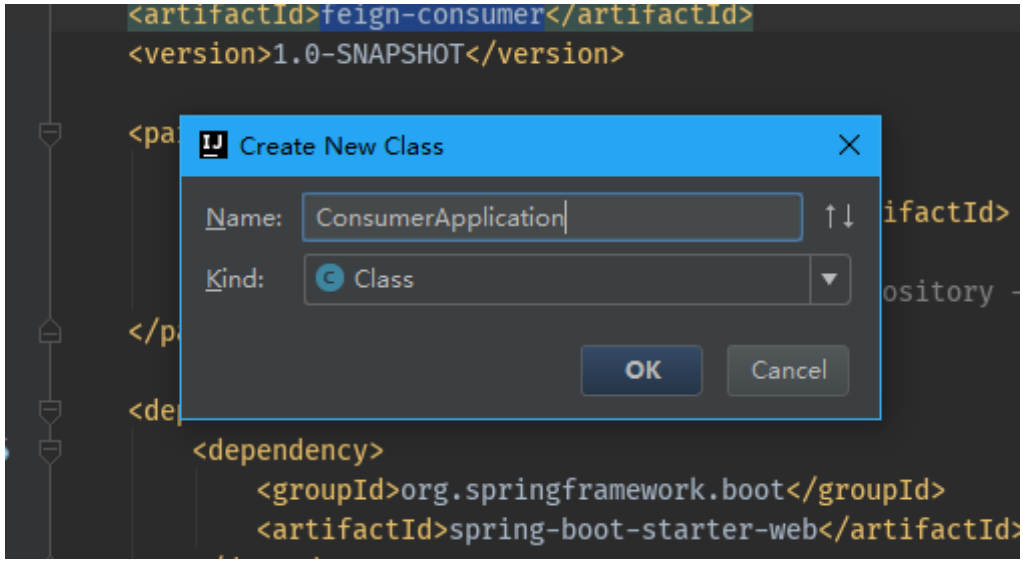


8、 建立应用主类：ConsumerApplication

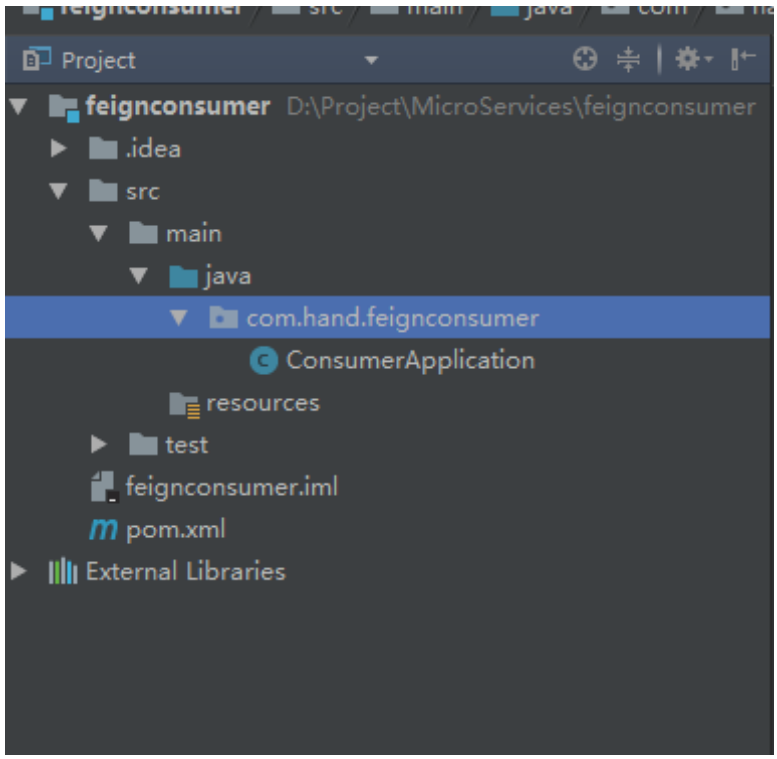
右击包 com.hand.feignconsumer->New->Java Class



9、 输入类名



10、 此时的目录结构如下图



11、 编辑主类

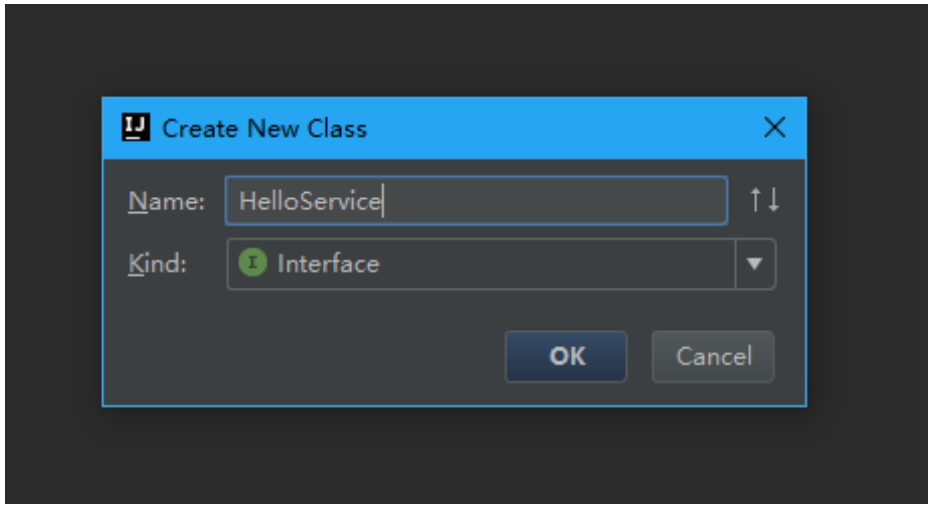
加入注解和主函数

```
@EnableFeignClients
@EnableDiscoveryClient
@SpringBootApplication
public class ConsumerApplication {
    public static void main(String[] args){

SpringApplication.run(ConsumerApplication.class,args);
```

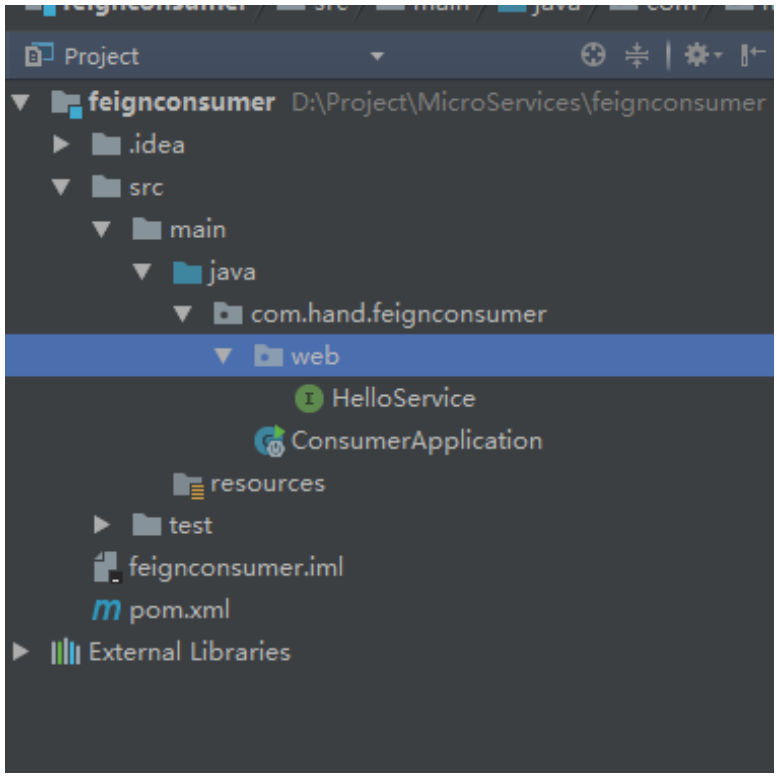
```
    }
}
```

12、 在 com.hand.feignconsumer 包下新建一个包

在该包下新建一个接口 HelloService



13、 此时项目的目录结构如下图：

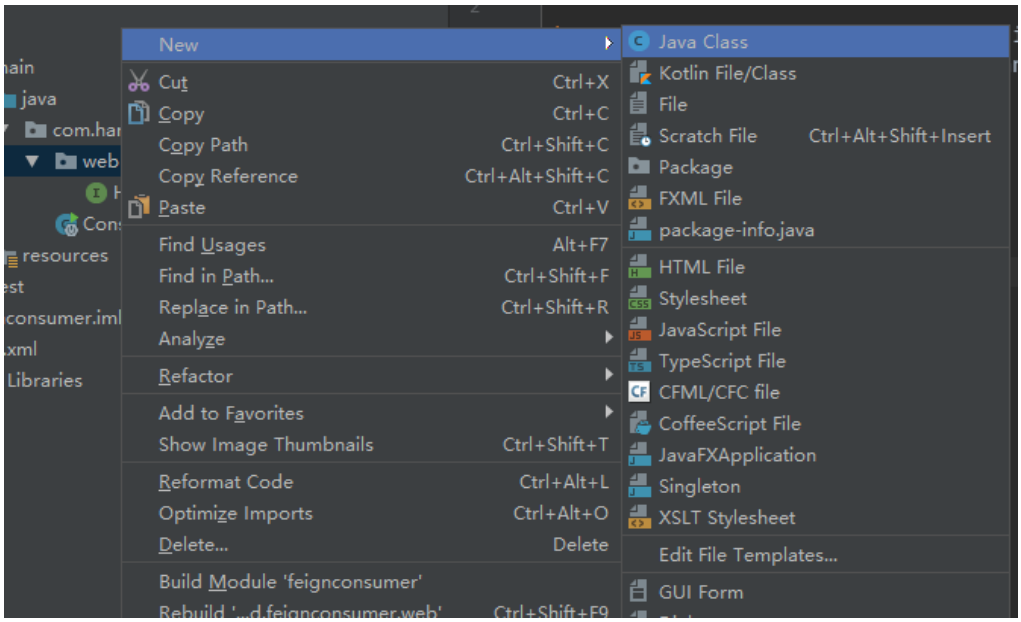


14、 定义服务接口，通过注解@FeignClient 指定服务名来绑定服务，然后再使用 Spring MVC 的注解来绑定具体该服务提供的 REST 接口

```
@FeignClient("hello-service")

public interface HelloService {

  @RequestMapping("/hello")
```

```
String hello();

}
```

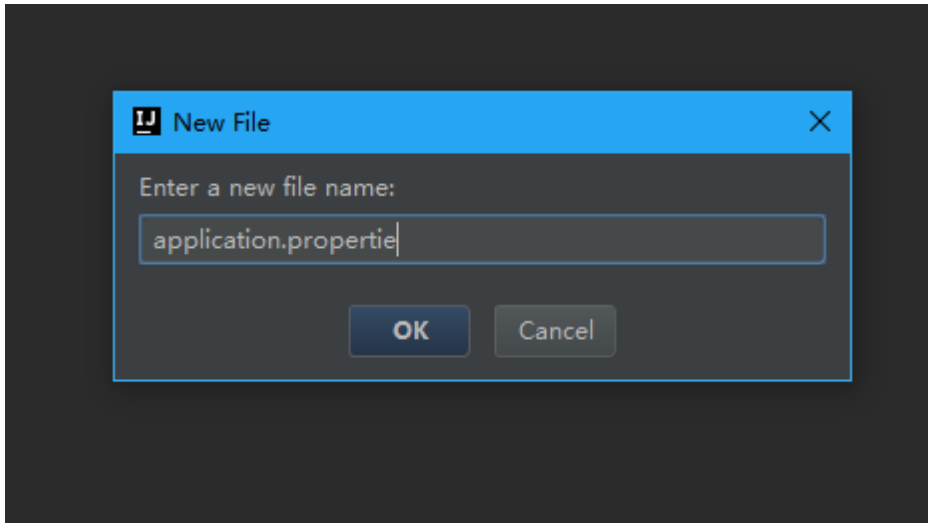
15、 创建一个 ConsumerController 来实现对 Feign 客户端的调用



16、 使用 @Autowired 直接注入上面定义的 HelloService 实例，并在 helloConsumer 函数中调用这个绑定了 hello-service 服务接口的客户端来向该服务发起 /hello 接口的调用。

```
@RestController

public class ConsumerController {

   @Autowired

   HelloService helloService;

   @RequestMapping(value = "/feign-consumer",method = RequestMethod.GET)

   public String helloConsumer(){

      return helloService.hello();

   }

}
```

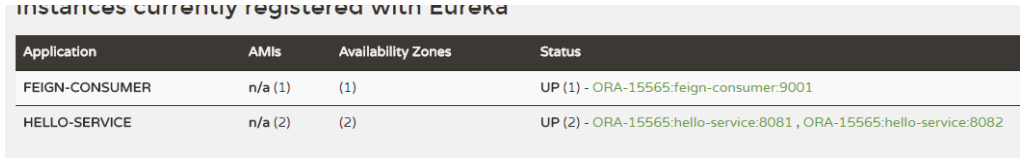
17、 创建应用配置文件



18、 指定服务注册中心

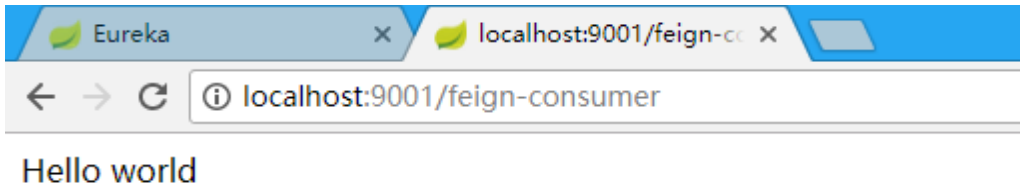
```
spring.application.name=feign-consumer

server.port=9001

eureka.client.serviceUrl.defaultZone =
http://localhost:1110/eureka/
```

19、 运行服务注册中心和 HELLO-SERVICE，进入服务注册中心可以看到服务已经注册成功



Application	AMIs	Availability Zones	Status
FEIGN-CONSUMER	n/a (1)	(1)	UP (1) - ORA-15565:feign-consumer:9001
HELLO-SERVICE	n/a (2)	(2)	UP (2) - ORA-15565:hello-service:8081 , ORA-15565:hello-service:8082

20、 测试



5.2. 参数绑定

5.2.1. 概述

在上一节的示例中，我们使用 Spring Cloud Feign 实现的是一个不带参数的 REST 服务绑定。然而现实系统中的各种业务接口要比它复杂得多，我们会在 HTTP 的各个位置传入各种不同类型的参数，并且在返回请求响应的时候也可能是一个复杂的对象结构。在本节中，我们将详细介绍 Feign 中对几种不同形式参数的绑定方法。

5.2.2. 扩展服务提供方接口

1、 扩展 Hello-Service 的 RESTful API

```
@RequestMapping(value = "/hello1",method = RequestMethod.GET)
public String hello(@RequestParam String name){
return "Hello world"+name;
}

@RequestMapping(value = "/hello2",method = RequestMethod.GET)
public User hello(@RequestHeader String name,@RequestHeader Integer age){
   return new User(name,age);
}

@RequestMapping(value = "/hello3",method = RequestMethod.POST)
public String hello(@RequestBody User user){
   return "Hello"+user.getName()+","+user.getAge();
}
```

2、 在 HelloServie 中构建 User 类，在参数绑定时将会用到 User 实例，需要实现 getName 和 getAge,同时注意和 Java 类中的 user 类的区分

```
public class User{
   private String name;
   private Integer age;

   public User(){

   }

   public User(String name,Integer age){
      this.name = name;
      this.age = age;
   }

   public String getName(){
      return name;
   }

   public Integer getAge(){
      return age;
   }

   @Override
   public String toString(){
      return "name = "+name+",age = "+age;
   }
}
```

3、 接着对 feign-coonsumer 进行改造

a) 首先， 在 feign-consumer 中创建与上面 一 样的 User 类。

b) 然后，在 HelloService 接口中增加对上述三个新增接口的绑定声明

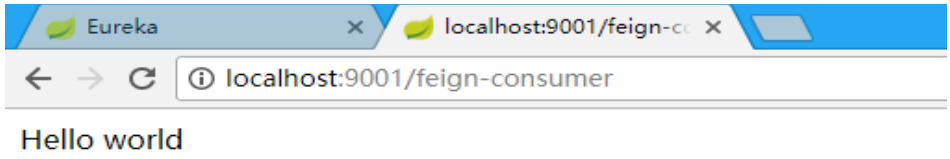
```
@RequestMapping(value = "/hello1",method = RequestMethod.GET)
String hello(@RequestParam("name") String name);

@RequestMapping(value = "/hello2",method = RequestMethod.GET)
User hello(@RequestHeader("name") String name,@RequestHeader("age")
Integer age);

@RequestMapping(value = "/hello3",method = RequestMethod.POST)
String hello(@RequestBody User user);
```

这里 一 定要注意，在定义各参数绑定时， @RequestParam、@RequestHeader 等可以指定参数名称的注解， 它们的 value 千万不能少。 在 SpringMVC 程序中，这些注解会根据参数名来作为默认值，但是在 Feign 中绑定参数必须通过 value 属性来指明具体的参数名，不然会抛出口 legalStateException 异常， value 属性不能为空。

4、 在 ConsumerController 中新增 一 个 /feign-consumer2 接口， 来对本节新增的声明接口进行调用

```
                                @RequestMapping(value = "/feign-
                                consumer2",method = RequestMethod.GET)
                                public String helloConsumer2(){
                                   StringBuilder sb = new StringBuilder();

                                sb.append(helloService.hello()).append("<br>");

                                sb.append(helloService.hello("DIDI")).append("
                                <br>");

                                sb.append(helloService.hello("DIDI",30)).appen
                                d("<br>");
                                   sb.append(helloService.hello(new
                                User("DIDI",30))).append("<br>");
                                   return sb.toString();
}
```

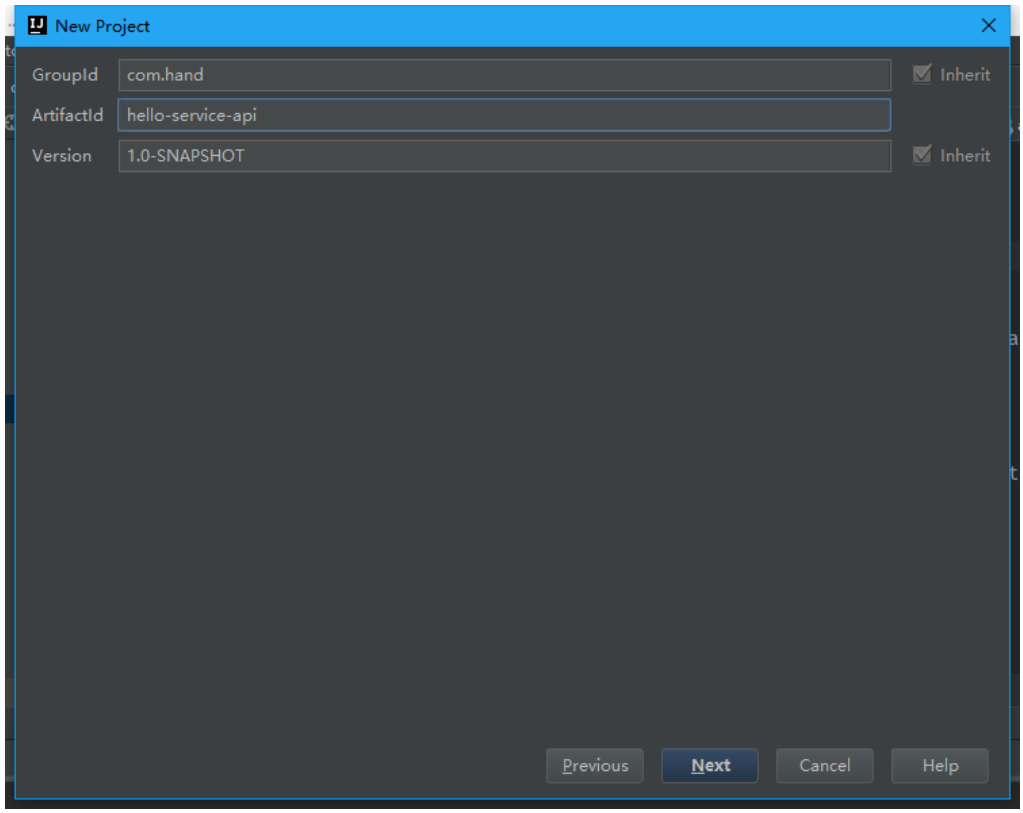
5、 验证测试



5.3. 继承特性

5.3.1. 概述

1、 创建一个基础的 Maven 工程，命名为 hello-service-api



2、 打开 pom.xml，添加依赖

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>1.3.7.RELEASE</version>
   <relativePath/> <!-- lookup parent from repository -->
</parent>

<properties>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
   <java.version>1.8</java.version>
</properties>

<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
</dependencies>

<build>
   <plugins>
      <plugin>
```

```
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
  </plugins>
</build>
```

3、 创建 HelloService 接口

```
@RequestMapping("/refactor")
public interface HelloService {
   @RequestMapping(value = "hello4",method = RequestMethod.GET)
   String hello(@RequestParam("name") String name);

   @RequestMapping(value = "hello5",method = RequestMethod.GET)
   User hello(@RequestHeader("name") String name, @RequestHeader("age")
Integer age);

   @RequestMapping(value = "hello6",method = RequestMethod.POST)
   String hello(@RequestBody User user);
}
```

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>1.3.7.RELEASE</version>
   <relativePath/> <!-- lookup parent from repository -->
</parent>

<properties>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
   <java.version>1.8</java.version>
</properties>

<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
</dependencies>

<build>
   <plugins>
      <plugin>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
   </plugins>
</build>
```

4、 创建 User 类，和之前 HelloService 创建的 User 类一样，这里可以直接复制过来

```
public class User {
   private String name;
   private Integer age;

   public User(){
```

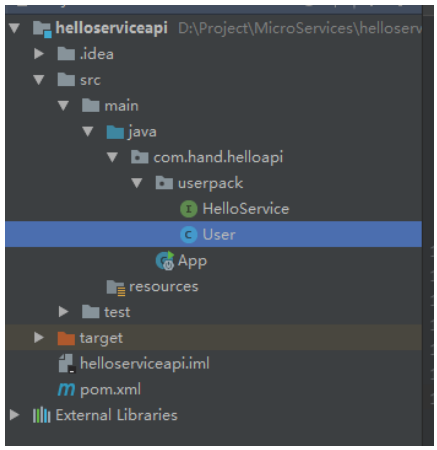
```
    }

    public User(String name,Integer age){
        this.name = name;
        this.age = age;
    }

    public String getName(){
        return name;
    }

    public Integer getAge(){
        return age;
    }

    @Override
    public String toString(){
        return "name = "+name+",age = "+age;
    }
}
```

5、 创建 HelloService 接口，该接口中使用的 User 类为刚刚新建的 User 类

```
@RequestMapping("/refactor")
public interface HelloService {
    @RequestMapping(value = "hello4",method = RequestMethod.GET)
    String hello(@RequestParam("name") String name);

    @RequestMapping(value = "hello5",method = RequestMethod.GET)
    User hello(@RequestHeader("name") String name, @RequestHeader("age")
Integer age);

    @RequestMapping(value = "hello6",method = RequestMethod.POST)
    String hello(@RequestBody User user);
}
```

为了和之前的参数绑定的 RESTful API 区别这里使用 value = hello4~6 进行区别

6、 在重构 Hello-Service 之前需要对 hello-service-api 进行打包，打包需要给该 api 增加主类和主函数，增加后项目结构如图



7、 使用 maven 进行打包 jar 包，使用 cmd 进入 hello-service-api 目录，使用命令

```
maven install
```

进行性打包，打包之后会在本地仓库找到对应的jar包

之后继承就是对该jar包中的HelloService进行继承

8、 对 Hello-Service 进行重构，在 porn.xrnl 的 dependency 节点中， 新增对 hello-service-api 的依赖。

```
<dependency>
  <groupId>com.hand</groupId>
  <artifactId>hello-service-api</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>
```

注意版本号和本地中的要对上

9、 创建 RefactorHelloController 类继承 hello-service-api 中定义的

HelloService 接口， 并参考之前的 HelloController 来实现这三个接口
在 Controller 中不再包含以往会定义的请求映射注解 @RequestMapping, 而参数的注解定义在重写的时候会自动带过来。
这就是继承的优势之一。除了要实现接口逻辑之外， 只需再增加 @RestController 注解使该类成为 一 个 REST 接口类
就大功告成了.

10、 完成了服务提供者的重构， 接下来在服务消费者 feign-consumer 的 pom.xml 文件中， 如在服务提供者中 一 样， 新增对 hello-service-api 的依赖。

```
@RestController
public class RefactorHelloController implements HelloService{
   @Override
   public String hello(@RequestParam("name") String name){
      return "hello "+name;
   }

   @Override
   public User hello(@RequestHeader("name") String name,
@RequestHeader("age") Integer age){
      return new User(name,age);
   }

   @Override
   public String hello(@RequestBody User user){
      return "Hello "+user.getName()+", "+user.getAge();
   }
}
```

```
<dependency>
   <groupId>com.hand</groupId>
   <artifactId>hello-service-api</artifactId>
   <version>1.0-SNAPSHOT</version>
</dependency>
```

11、 创建 RefactorHelloService 接口， 并继承 hello-service-api 包中的

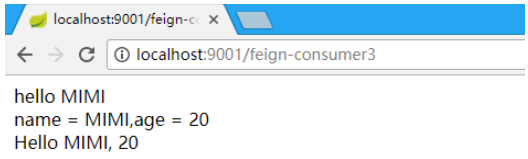
HelloService 接口， 然后添加 @FeignClient 注解来绑定服务。

```
@FeignClient(value = "hello-service")
public interface RefactorHelloService extends
com.hand.helloapi.userpack.HelloService {

}
```

12、 最后， 在 Consume 沁 ontroller 中， 注入 RefactorHelloService 的实例， 并新增 一 个请求 /feign-consumer3 来触发对 RefactorHelloService 的实例 的调用。

```
@Autowired
RefactorHelloService refactorHelloService;
@RequestMapping(value = "/feign-consumer3",method =
RequestMethod.GET)
public String helloConsumer3(){
   StringBuilder sb = new StringBuilder();
   sb.append(refactorHelloService.hello("MIMI")).append("<br>");
   sb.append(refactorHelloService.hello("MIMI",20)).append("<br>");
   sb.append(refactorHelloService.hello(new
com.hand.helloapi.userpack.User("MIMI",
         20))).append("<br>");
   return sb.toString();
}
```

13、 测试验证,验证成功



5.4. 服务降级配置

5.4.1. 概述

Hystrix提供的服务降级是服务容错的重要功能，由于Spring Cloud Feign在定义服务客户端的时候与Spring Cloud和Ribbon有很大差别，HystrixCommand定义被封装了起来， 我们无法像之前介绍Spring Cloud Hystrix时， 通过@ HystrixCommand注解的fallback参数那样来指定具体的服务降级处理方法。

服务降级逻辑的实现只需要为 Feign 客户端的定义接口编写 一 个具体的接口实现类。

5.4.2. 服务降级的实现

1、 为 HelloService 接口实现 一 个服务降级类 HelloServiceFallback, 其中每个重写方法的实现逻辑都可以用来定义相应的服务降级逻辑， 具体如下：

```
@Component
public class HelloServiceFallback implements HelloService {

    @Override
    public String hello(){
        return "error";
    }

    @Override
    public String hello(@RequestParam("name") String name){
        return "error";
    }

    @Override
    public User hello(@RequestHeader("name") String
name,@RequestHeader("age") Integer age){
        return new User("UNKNOW",0);
    }

    @Override
    public String hello(@RequestBody User user){
        return "error";
    }
}
```

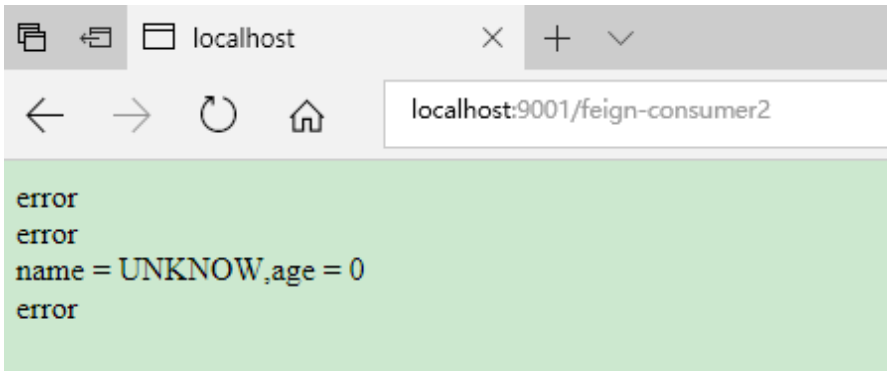
2、 在服务绑定接口 HelloService 中， 通过 @FeignClient 注解的 fallback 属性来指定对应的服务降级实现类

```
@FeignClient(name = "hello-service",fallback = HelloServiceFallback.class)
public interface HelloService {
    @RequestMapping("/hello")
    String hello();

    @RequestMapping(value = "/hello1",method = RequestMethod.GET)
    String hello(@RequestParam("name") String name);

    @RequestMapping(value = "/hello2",method = RequestMethod.GET)
    User hello(@RequestHeader("name") String name,@RequestHeader("age")
Integer age);

    @RequestMapping(value = "/hello3",method = RequestMethod.POST)
```

```
    String hello(@RequestBody User user);
}
```

3、 测试验证

启动服务注册中心和feign- consumer,但是不启 动hello-service 服 务。 发送GET请 求到http://localhost:9001/feign-consumer2, 该接口会分别调用HelloService中的 4 个绑定接口， 但因为hello-service服务没有启动， 会直接触发服务降级， 并获得下面的输出内容



6. API网关服务：Spring Cloud Zuul

6.1. 构建网关

6.1.1. 概述

微服务架构虽然可以将我们的开发单元拆分得更为细致， 有效降低了开发难度， 但是它所引出的各种问题如果处理不当会成为实施过程中的不稳定因素， 甚至掩盖掉原本实施微服务带来的优势。 所以， 在微服务架构的实施方案中， API网关服务的使用几乎成为了必然的选择。

1、 创建 一 个基础的 Spring Boot 工程， 命名为 api-gateway, 并在 pom.xml 中引入 spring-cloud-starter-zuul 依赖， 具体如下：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

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

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

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>Brixton.SR5</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
```

2、 创建应用主类， 使用@EnableZuulProxy 注解开启 Zuul 的 API 网关服务功能

```
@EnableZuulProxy
@SpringCloudApplication
public class Application {
  public static void main(String[] args) {
    new SpringApplicationBuilder(Application.class).web(true).run(args);
  }

  @Bean
  public AccessFilter accessFilter() {
```

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

3、 在 application.properties 中配置 Zuul 应用的基础信息， 如应用名、 服务端口号等， 具体内容如下：

```
spring.application.name=api-gateway
server.port=5555
```

4、 完成上面的工作后， 通过 Zuul 实现的 API 网关服务就构建完毕了

6.1.2. 传统路由方式

使用Spring Cloud Zuul实现路由功能非常简单， 只需要对 ap丁gateway服务增加 一些关于路由 规则的配置， 就能实现传统的路由转发功能，比如：

```
zuul.routes.api-a-url.path=/api-a-url/**
zuul.routes.api-a-url.url=http://localhost:8080/
```

该配置定义了发往API网关服务的请求中， 所有符合/api-a-url/**规则的访问都将被路由 转发到http://localhost:8080/地址上， 也就是说，当我 们 访问 http://localhost:5555/api-a-url/hello的时候，API网关服务会将该请求路由到 http://localhost:8080/hello 提供的 微 服 务接口上。 其中，配 置属性zuul.routes.api-a-url.path 中的api-a-url部分为路由的名字， 可以任意定义，但是一 组path和url 映射关系的路由 名要相同，下面将要介绍的面向服务的映射方式也是如此。

6.1.3. 面向服务的路由

很显然， 传统 路由的配置方式对于我们来说并不友好， 它同样需要运维人员花费大量的时间来维护各个路由 path与url的关系 。 为了解决这个问题， SpringCloud Zuul实现了与Spring Cloud Eureka的无缝整合， 我们可以让路由的path不是映射具体的url, 而是让它映射到某个具体的服务 ， 而具体的url则交给Eure ka的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。 在Zuul中使用服务 路由也同样简单， 只需做下面这些配置。

1、 在 api-gateway 的 pom.xml 中引入spring- cloud-starter-eureka依赖，具体如下：

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

2、 在api-gateway的application.properties 配置文件中指定Eureka注册中心的位置，并且配置服务 路由。

```
zuul.routes.api-a.path=/api-a/**
zuul.routes.api-a.serviceId=hello-service
zuul.routes.api-b.path=/api-b/**
zuul.routes.api-b.serviceId=feign-consumer
eureka.client.serviceUrl.defaultZone=http://localhost:1110/eureka/
```

3、在完成了上面的服务路由配置之后，我们可以将eureka-server、hello-service、feign-consumer以及这里用Spring Clo ud Zuul构建的api-gateway都启动起来

3、 通过上面的搭建工作， 我们已经可以通过服务网关来访问 hello-service 和feign-consumer这两个服务 了。 根据配置的映射关系， 分别向网关发 起下面这些请求。

4、 http://localhost:5555/api-a/hello: 该 url符合/api-a/**规则， 由api-a路由负责转发，该路由映射的serviceid为hello-service, 所以最终/hello请求会被发送到hello-service服务的 某个 实例上去。

5、 http://localhost:5555/api-b/feign-consumer: 该url符合/api-b/**规则，由 api-b路由负责转发，该路由映射的serviceId为feign-consumer,所以最终/feign-consumer请求会被发送到feign-consumer服务的某个 实例上去。

6、 通过面向服务的路由配置 方式， 我们不需要再为各个路由维护微服务应用的具体实例的位置，而是 通过简单的path与serviceId的映射组合，使得维护工作变得非常简单。 这完全归功于Spring Cloud Eureka的服务发现机制，它使得API网关服务可以自动化完成服务实例清单的维护，完美地解决了对路由映射 实例的维护问题.

7、 上述两个链接请读者自行验证。

8、 End

6.2. 请求过滤

6.2.1. 概述

为了在API网关中实现对客户端请求的校验，我们将继续介绍 Spring Cloud Zuul 的另外 一 个核心功能：请求过滤。 Zuul 允许开发者在API网关上通过定义过滤器来实现对请求的拦截与过滤，实现的方法非常简单，我们只需要继承 ZuulFi li er 抽象类并实现它定义的4个抽象函数就可以完成对请求的拦截和过滤了。

6.2.2. Zuul过滤器

下面的代码定义了一 个简单的Zuul 过滤器， 它实现了在请求被路由之前检查 HttpServletReque 江中是否有 accessToken 参数， 若有就进行路由， 若没有就拒绝访问， 返回 401 Unauthorized 错误。

1、 实现过滤器ZuulFilter接口

```
public class AccessFilter extends ZuulFilter{
   private static Logger log = LoggerFactory.getLogger(AccessFilter.class);

   @Override
   public String filterType(){
      return "pre";
   }

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

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

   @Override
   public Object run(){
      RequestContext ctx = RequestContext.getCurrentContext();
      HttpServletRequest request = ctx.getRequest();

      log.info("send {} request to
{}",request.getMethod(),request.getRequestURL().toString());

      Object accessToken = request.getParameter("accessToken");
      if(accessToken == null){
         log.warn("access token is empty");
         ctx.setSendZuulResponse(false);
         ctx.setResponseStatusCode(401);
         return null;
      }
      log.info("access token is ok");
      return null;
   }
}
```

2、 在实现了自定义过滤器之后， 它并不会直接生效， 我们还需要为其创建具体的Bean才能启动该过滤器， 比如， 在应用主类中增加如下内容：

```
@EnableZuulProxy
@SpringCloudApplication
public class Application {
    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

    @Bean
    public AccessFilter accessFilter() {
        return new AccessFilter();
}
    }
```

3、 在对api-gateway服务完成了上面的改造之后， 我们可以重新启动它， 并发起下面的请求， 对上面定义的过滤器做 一 个验证。

4、 访问http://localhost:5555/api-a/hello: 返回401错误

5、 访问http://localhost:5555/api-a/hello?accessToken=token正确路由到hello-service的/hello接口，并返回Hello world

6、 End

6.3. 动态路由

6.3.1. 概述

通过之前对请求路由的详细介绍， 我们可以发现对于路由规则的控制几乎都可以在配置文件application.properties或application.yaml中完成。 既然这样， 对于如何实现Zuul的动态路由，我们很自然地会将它与SpringCloud Config的动态刷新机制联系到一 起。 只需将API网关服务的配置文件通过Spring Cloud Config连接的Git仓库存储和管理， 我们就能轻松实现动态刷新路由规则的功能。

在介绍如何 具体实现API网关服务的动态路由之前， 我们首先需要 一 个连接到Git仓库的分布式配置中心config-server应用。 如果还没有搭建过分布式配置中心的话， *建议先阅读第8章的内容， 对分布式配置中心的运作机制有 一 个基础的了解，* 并构建一 个config-server应用， 以配合完成下面的内容。

在具备了分布式配置中心之后， 为了方便理解， 我们重新构建 一 个API网关服务，该服务的配置中心不再配置 于本地工程中， 而是从config-server中获取。

6.3.2. 构建动态路由

1、 创建 一 个基础的SpringBoot工程， 命名为api -gateway-dynamic-route

2、 在pom.xml中引入对zuul、 eureka 和config 的依赖， 具体内容如下：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

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

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

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

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>Brixton.SR5</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
```

```
</dependencyManagement>
```

3、 在/resource目录下创建配置文件bootstrap.properties, 并在该文件中指定config-server和eureka-server的具体地址， 以获取应用的配置文件和实现服务注册与发现

```
spring.application.name=api-gateway
server.port=5556

spring.cloud.config.uri=http://localhost:7001/

eureka.client.serviceUrl.defaultZone = http://localhost:1110/eureka/
```

4、 创建用来启动API网关的应用主类。 这里我们需要使用@RefreshScope注解来将Zuul的配置内容动态化， 具体实现如下

```
@EnableZuulProxy
@SpringCloudApplication
public class Application {
   public static void main(String[] args) {
      new SpringApplicationBuilder(Application.class).web(true).run(args);
   }

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

5、 在完成了所有程序相关的编写之后， 我们还需要在 Git 仓库中增加网关的配置文件，取名为 api-gateway.properties 。 在配置文件中， 我们为 API 网关服务预定义以下路由规则

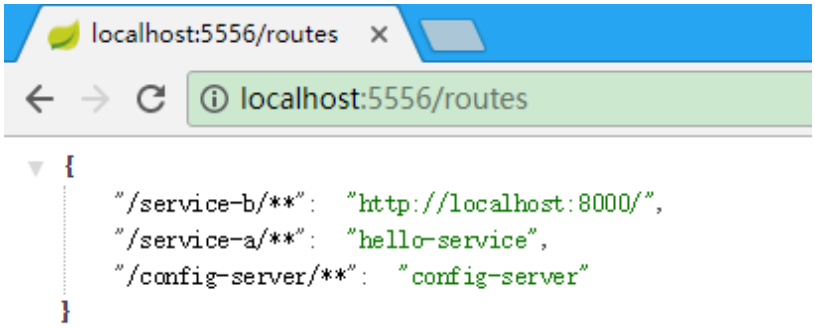
```
zuul. routes. service - a. pa th=/ service - a/**
zuul. routes. service - a. serviceid = hello - service
zuul. routes. service - b .path=/service - b/**
zuul. routes. service - b. url=http: / /localhost：8001/
```

6、 测试与验证：在完成了上述内容之后， 我们可以将 config-server、 eureka-server、api-gateway-dynamic-route 以及配置文件中路由规则指向的具体服务，比如hello-service 启动起来，在启动api -gateway-dynamic-routes时，若启动正确的情况下，控制台会输出一下配置信息

```
: Updating port to 5556
: Started Application in 11.216 seconds (JVM running for 12.737)
: Fetching config from server at: http://localhost:7001/
: Located environment: name=application, profiles=[default], label=master, version=79a5c78e16ca43e01bf9e77e4b7c129de0d83008
```

7、 在 api-gateway-dynamic-route 启动完成之后， 可以通过对 API 网关服务调用

/routes 接口来获取当前网关上的路由规则， 根据上述配置我们可以得到如下返回信息



8、 关于config-server的内容可以先阅读下章，然后回过头来阅读本节内容

9、 End

6.4. 动态过滤器

6.4.1. 概述

通过之前介绍的请求路由和请求过滤的示例， 我们可以看到请求路由通过配置文件就能实现，而请求过滤则都是通过编码实现。 所以，对于实现请求过滤器的动态加载，我们需要借助基于NM实现的动态语言的帮助， 比如Groovy

6.4.2. 构建动态过滤器

1、 创建 一 个基础的 Spring Boot 工程，命名为 api-gateway-dynarnic-filter

2、 在 pom.xml 中引入对zuul 、 eureka 和 groovy 的依赖，具体内容如下

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

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

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

    <dependency>
      <groupId>org.codehaus.groovy</groupId>
      <artifactId>groovy-all</artifactId>
    </dependency>
  </dependencies>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>Brixton.SR5</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
```

3、 在 /resource 目录下创建配置文件application.properties, 并在该文件中设置API网关服务的应用名和服务端口号，以及指定 eureka-server 的具体地址。同时， 再配置 一 个用于测试的路由规则， 我们可以用之前章节实现的 hello- service 为例。具体配置内容如下

```
spring.application.name=api-gateway
server.port=5555
```

```
eureka.client.serviceUrl.defaultZone=http://localhost:1110/eureka/

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

zuul.filter.root = filter
zuul.filter.interval = 5
```

4、 创建用来加载自定义属性的配置类， 命名为 FilterConfiguration, 具体内容如下

```
@ConfigurationProperties("zuul.filter")
public class FilterConfiguration {

   private String boot;
   private Integer interval;

   public String getRoot() {
      return boot;
   }

   public void setRoot(String boot) {
      this.boot = boot;
   }

   public Integer getInterval() {
      return interval;
   }

   public void setInterval(Integer interval){
      this.interval = interval;
   }
}
```

5、 创建应用启动主类， 并在该类中引入上面定义的 FilterConfiguration 配置，并创建动态加载过滤器的实例。

```
@EnableZuulProxy
@EnableConfigurationProperties(FilterConfiguration.class)
@SpringCloudApplication
public class Application {
   public static void main(String[] args) {
      new SpringApplicationBuilder(Application.class).web(true).run(args);
   }

   @Bean
   public FilterLoader filterLoader(FilterConfiguration filterConfiguration){
      FilterLoader filterLoader = FilterLoader.getInstance();
      filterLoader.setCompiler(new GroovyCompiler());
      try {
         FilterFileManager.setFilenameFilter(new GroovyFileFilter());
         FilterFileManager.init(
               filterConfiguration.getInterval(),
               filterConfiguration.getRoot()+ "/pre",
               filterConfiguration.getRoot()+ "/post"
         );

      }catch (Exception e){
         throw new RuntimeException(e);
      }
```

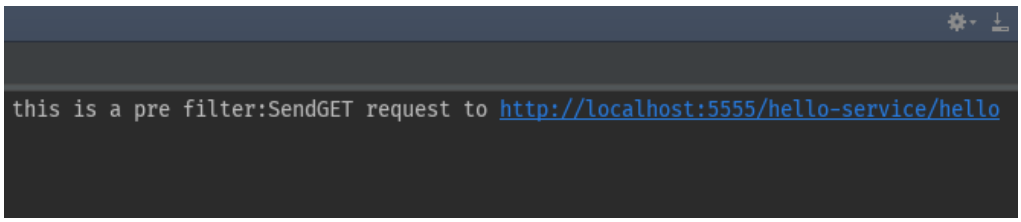
```
        return filterLoader;
    }
}
```

6、 在 filter/pre 目录下创建 一 个 pre 类型的过滤器，命名为 PreFilter.groovy 。由于 pre 类型的过滤器在请求路由前执行，通常用来做 一 些签名校验的功能， 所以我们可以在过滤器中输出一 些请求相关的信息， 比如下面的实现

```
class PreFilter extends  ZuulFilter{
    Logger log = LoggerFactory.getLogger(PreFilter.class);

    @Override
    String filterType(){
        return  "pre"
    }

    @Override
    int filterOrder(){
        return 1000
    }

    @Override
    boolean shouldFilter(){
        return true
    }

    @Override
    Object run(){
        HttpServletRequest request =
RequestContext.getCurrentContext().getRequest()
        log.info("this is a pre filter:Send{} request to
{}",request.getMethod(),request.getRequestURL().toString())
        return null
    }
}
```

7、 启动eureka-server ， hello-service, api-gateway-dynarnic-filter 向api网关服务请求http:/ /localhost: 5555/hello-service/hello

8、 若配置正确的情况下api-gateway-dynarnic-filter控制台会有消息输出：



9、 在此基础上可以在filter/post文件夹中创建Post类型过滤器命名为 PostFilter.groovy 。由于 post 类型的过滤器在请求路由返回后执行， 我们可以进一 步对这个结果做 一些处理， 对微服务返回的信息做 一 些加工

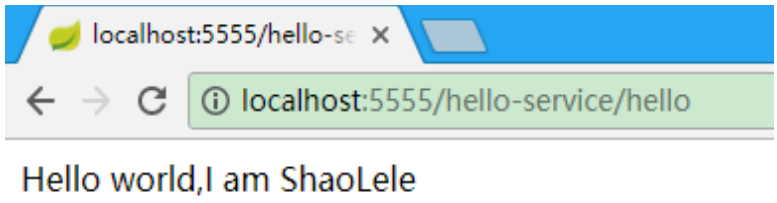
```
class PostFilter extends  ZuulFilter{
    Logger log = LoggerFactory.getLogger(PostFilter.class);

    @Override
    String filterType(){
        return  "post"
    }
```

```
    @Override
    int filterOrder(){
        return 2000
    }

    @Override
    boolean shouldFilter(){
        return true
    }

    @Override
    Object run(){
        log.info("this is a post filter: Receive response")
        HttpServletResponse response =
RequestContext.getCurrentContext().getResponse()
        response.getOutputStream().print(",I am ShaoLele")
        response.flushBuffer()
    }
}
```

10、 无需重启服务，只需5分钟后，重新向网关服务请求，http:/ /localhost:5555/hello-service/hello在配置正确的情况下可以看到控制台有消息输出：

```
: Resolving eureka endpoints via configuration
: this is a pre filter:SendGET request to http://localhost:5555/hello-service/hello
: this is a post filter: Receive response
```

11、 接口返回我们修改后的字符串而不再是Hello word.



12、 End

7. 分布式配置中心： Spring Cloud Config

Spring Cloud Config 是 Spring Cloud 团队创建的 一 个全新项目，用来为分布式系统中的基础设施和微服务应用提供集中化的外部配置支持， 它分为服务端与客户端两个部分。 其中服务端也称为分布式配置中心， 它是 一 个独立的微服务应用， 用来连接配置仓库并为客户端提供获取配置信息、 加密 / 解密信息等访问接口；而客户端则是微服务架构中的各个微服务应用或基础设施， 它们通过指定的配置中心来管理应用资源与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息。 Spring Cloud Config 实现了对服务端和客户端中环境变量和属性配置的抽象映射，所以它除了适用于 Spring 构建的应用程序之外，也可以在任何其他语言运行的应用程序中使用。 由于 Spring Cloud Config 实现的配置中心默认采用 Git 来存储配置信息， 所以使用 Spring Cloud Config 构建的配置服务器， 天然就支持对微服务应用配置信息的版本管理， 并且可以通过 Git 客户端工具来方便地管理和访问配置内容。 当然它也提供了对其他存储方式的支持， 比如 SYN 仓库、 本地化文件系统。接下来，我们从 一 个简单的入门示例开始学习 Spring Cloud Config 服务端以及客户端的详细构建与使用方法。

7.1. SpringCloudConfig初识

7.1.1. 概述

在上章最后一节中我们以及提到并且使用了分布式配置中心，通过本节内容我们可以构建一个配置中心，实现远程配置。

在本节中， 我们将演示如何构建 一 个基于 Git 存储的分布式配置中心， 同时对配置的详细规则进行讲解， 并在客户端中演示如何通过配置指定微服务应用的所属配置中心， 并让其能够从配置中心获取配置信息并绑定到代码中的整个过程。

7.1.2. 构建配置中心

通过 Spring Cloud Config 构建 一 个分布式配置中心非常简单， 只需要以下三步。

1、 创建 一 个基础的 Spring Boot 工程， 命名为 config-server, 并在 pom.xml 中引入下面的依赖：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

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

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
```

```
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

2、 创建 Spring Boot 的程序主类， 并添加 @EnableConfigServer 注解， 开启 SpringCloud Config 的服务端功能。

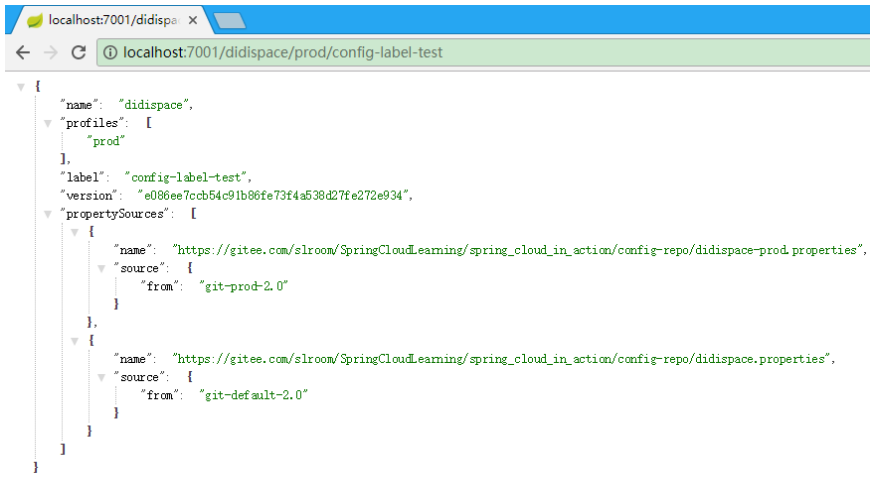
```
@EnableDiscoveryClient
@EnableConfigServer
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }
}
```

3、 在application.properties 中添加配置服务的基本信息以及 Git 仓库的相关信息，如下所示：

```
spring.application.name=config-server
server.port=7001
spring.cloud.config.server.git.uri=https://gitee.com/slroom/SpringCloudLearning/
spring.cloud.config.server.git.searchPaths=spring_cloud_in_action/config-repo
spring.cloud.config.server.git.username=username
spring.cloud.config.server.git.password=password
```

在这里的配置地址是已经配置好的Git仓库，读者可以自己在中国的github—“码云” https://gitee.com/ 上注册账号，然后创建自己的git仓库，具体配置步骤见下节。这里读者可以使用作者的git仓库进行验证。

4、 启动分布式配置中心，在一切配置正确的情况下



7.2. 配置详解

7.2.1. Git配置仓库

1、 在SpringCloud Config的服务端， 对于配置仓库的默认实现采用了Git。Git非常适用于存储配置内容，它可以非常方便地使用各种第三方工具来对其内容进行管理更新和版本化，同时Git仓库的Hook功能还可以帮助我们实时地监控配置内容的修改。 其中，Git自身的版本控制功能正是其他 一 些配置中心所欠缺的， 通过Git 进行存储意味着，一 个应用的不同部署实例可以从SpringCloud Config的服务端获取不同的版本配置， 从而支持 一 些特殊的应用场景。

2、 由于SpringCloud Config中默认使用Git, 所以对于Git的配置也非常简单， 只需在Config Server的apptication.properties 中设置spring.cloud.config.server.git.uri 属性，为其指定Git仓库的网络地址和账户信息即可比如在上节中的Demo:

```
spring.cloud.config.server.git.uri=https://gitee.com/slroom/SpringCloudLearning/
spring.cloud.config.server.git.searchPaths=spring_cloud_in_action/config-repo
spring.cloud.config.server.git.username=username
spring.cloud.config.server.git.password=password
```

3、 如果我们将该值通过file://前缀来设置为 一 个文件地址（在Windows系统中， 需要使用file:///来定位文件内容），那么它将以本地仓库的方式运行，这样我们就可以脱离Git服务端来快速进行调试与开发， 比如：spring.cloud.config.server.git.uri=file://${user.home}/config-repo其中${user.home}代表当前用户的所属目录 。file://配置的本地文件系统方式虽然对千本地开发调试时使用非常方便， 但是该方式也仅用于开发与测试， 在生产环境中请务必搭建自己的Git仓库来存储配置资源。

7.2.2. 占位符配置URI

1、 {apptication}、{profile}、{label}这些占位符除了用于标识配置文件的规则之外，还可以用于ConfigServer 中对 Git 仓库地址的URI配置。 比如， 我们可以通过{apptication}占位符来实现 一 个应用对应 一 个Git仓库目录的配置效果，具体配置实现如下：

```
spring.cloud.config.server.git.uri=https://gitee.com/slroom/{application}
spring.cloud.config.server.git.username=username
spring.cloud.config.server.git.password=password
```

2、 其中，{application}代表了应用名， 所以当客户端应用向ConfigServer发起获取配置的请求时，ConfigServer会根据客户端的spring.application.name 信息来填充{application}占位符以定位配置资源的存储位置， 从而实现根据微服务应用的属性动态获取不同位置的配置。 另外， 在这些占位符中， {label}参数较为特别 ， 如果Git的分支和标签名包含"!", 那么{label}参数在HTTP的URL中应该使用 "U" 来替代， 以避免改变了URI含义， 指向到其他的URI资源。

3、 当我们使用Git作为配置中心来存储各个微服务应用配置文件的时候， 该功能会变得非常有用， 通过在URI中使用占位符可以帮助我们规划和实现通用的仓库配置。 例如， 我们可以对微服务应用做如下规划 。

a) 代码库

b) 配置库

7.2.3. 练习

请读者学习上节内容后，自己搭建一个仓库，码云地址：

https://gitee.com/

注册账号后新建应用仓库，在之后的练习中会用到。

7.3. 安全保护

7.3.1. 概述

1、 由于配置中心存储的内容比较敏感，做 一 定的安全处理是必需的。 为配置中心实现安全保护的方式有很多，比如物理网络限制、 0Auth2 授权等。 不过，由于我们的微服务应用和配置中心都构建于 Spring Boot 基础上，所以与 Spring Security 结合使用会更加方便。

2、 我们只需要在配置中心的 pom.xml 中加入 spring-boot-starter-securit y 依赖，不需要做任何其他改动就能实现对配置中心访问的安全保护。

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

3、 默认情况下，我们可以获得 一 个名为 user 的用户，并且在配置中心启动的时候，在日志中打印出该用户的随机密码

4、 大多数情况下，我们并不会使用随机生成密码的机制。 我们可以在配置文件中指定用户和密码， 比如：

```
security.user.name=user
security.user.password=37cc5635-559b-4e6f-b633-7e932b813f73
```

5、 由于我们已经为 config-server 设置了安全保护，如果这时候连接到配置中心的客户端中没有设置对应的安全信息，在获取配置信息时会返回401错误。 所以，需要通过配置的方式在客户端中加入安全信息来通过校验， 比如：

```
spring.cloud.config.username=user
spring.cloud.config.password=37cc5635-559b-4e6f-b633-7e932b813f73
```

7.4. 服务化配置中心

7.4.1. 服务端配置

在第3章中， 我们已经学会了如何构建服务注册中心、 如何发现与注册服务。 那么Config Server是否也能以服务的方式注册到服务中心， 并被其他应用所发现来实现配置信息的获取呢？答案是肯定的。 在SpringCloud中， 我们也可以把ConfigServer视为微服务架构中与其他业务服务 一 样的 一 个基本单元。下面， 我们就来详细介绍如何将ConfigServer注册到服务中心， 并通过服务发现来访问ConfigServer并获取Git仓库中的配置信息。 下面的内容将基于快速入门中实现的config-server和config-client工程来进行改造实现。

1、 在config-server 的pom.xml 中增加spring-cloud-starter-eureka依赖， 以实现将分布式配置中心加入Eureka的服务治理体系

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

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

2、 在 application.properties 中配置参数eureka.client.serviceUrl.defaultZone以指定服务注册中心的位置， 详细内容如下

```
spring.application.name=config-server
server.port=7001

#配置服务注册中心
eureka.client.serviceUrl.defaultZone=http://localhost:1110/eureka/
#Git管理
spring.cloud.config.server.git.uri=https://gitee.com/slroom/SpringCloudLearning/
spring.cloud.config.server.git.searchPaths=spring_cloud_in_action/config-repo
spring.cloud.config.server.git.username=username
spring.cloud.config.server.git.password=password
```

3、 在应用主类中，新增@EnableDiscoveryClient注解，用来将config-server注册到上面配置的服务注册中心上去。

```
@EnableDiscoveryClient
@EnableConfigServer
@SpringBootApplication
public class Application {
   public static void main(String[] args) {
      new SpringApplicationBuilder(Application.class).web(true).run(args);
   }
}
```

4、 启动该应用， 并访问http://localhost: 1110/, 可以在Eureka Server的信息面板中看到config-server已经被注册了。

7.4.2. 客户端配置

1、 在config-client的porn.xrnl中新增spring-cloud-starter-eureka依赖， 以实现客户端发现 config-server 服务， 具体配置如下：

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

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

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

</dependencies>
```

2、 在 bootstrap.properties 中， 按如下配置：

```
pring.application.name=didispace
#spring.cloud.config.label=master
#spring.cloud.config.uri=http://localhost:7001/

server.port=7002

eureka.client.serviceUrl.defaultZone=http://localhost:1110/eureka/
spring.cloud.config.discovery.enabled=true
spring.cloud.config.discovery.serviceId=config-server
spring.cloud.config.profile=dev
```

其中， 通过 eureka.client.serviceUrl.defaultZone 参数指定服务注册中心， 用于服务的注册与发现； 再将 spring.cloud.config.discovery.enabled 参数设置为 true, 开启通过服务来访问 Config Server 的功能；最后利用 spring.cloud.config.discovery.serviceid 参数来指定 Config Server 注册的服务名。这里的 spring.application.name 和 spring.cloud.config.profile 如之前通过 URI 的方式访问的时候 一 样， 用来定位 Git 中的资源。

3、 在应用主类中，增加 @EnableDiscoveryClient 注解，用来发现 config-server服务， 利用其来加载应用配置

```
@EnableDiscoveryClient
@SpringBootApplication
public class Application {
   public static void main(String[] args) {
      new SpringApplicationBuilder(Application.class).web(true).run(args);
   }
}
```

4、 沿用之前我们创建的Controlle r来加载Git中的配置信息：

```
@RefreshScope
@RestController
public class TestController {
   @Value("${from}")
   private String from;
    @RequestMapping("/from")
    public String from(){
        return this.from;
}
```

5、 完成上述配置之后， 我们启动该客户端应用。 若启动成功， 访问http:// localhost: 1111/, 可以在EurekaServer的信息面板中看到该应用已经被注册成功。

Application	AMIs	Availability Zones	Status
CONFIG-SERVER	n/a (1)	(1)	UP (1) - localhost:config-server:7001
DIDISPACE	n/a (1)	(1)	UP (1) - ORA-15565.mshome.net:didispace:7002

6、 访问客户端应用提供的服务http://localhost:7002/from, 此时， 我们会返回在Git仓库中di中space-dev.properties文件中配置的from属性内容：

" git -dev-1.0"

7.5. 动态刷新配置

7.5.1. 概述

1、 有时候， 我们需要对配置 内容做 一 些实时更新， 那么Spring Cloud Config是否可以实现呢？答案显然是可以的。 下面， 我们以之前的示例作为基础， 看看如何进行改造来实现配置内容的实时更新。

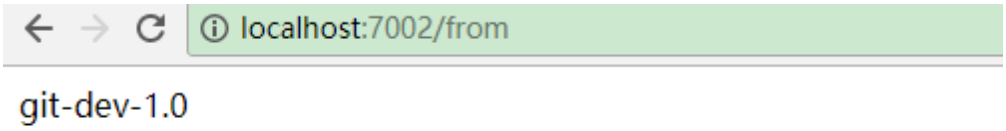
2、 首先，回顾 一 下，当前我们已经实现了哪些内容。

a) config-repo: 定义在Git仓库中的一 个目录，其中存储了应用名为主主space的多环境配置文件， 配置文件中有一 个from参数。

b) config-server: 配置了Git仓库的服务端。

c) config-client: 指定了 config-server 为配置中心的客户端， 应用名为didispace, 用来访问配置 服务器以获取配置信息。 该 应用中提供了一 个/from接口， 它会获取config-repo/didispace-dev.proper巨es 中的from属性返回。

3、 在改造程序之前，我们先将config-server和config-clien七都启动起来，并访问客户端提供的REST接口 http://localhost:7002/from 来获取配置信息，获得的返回内容为git-dev-1.0。接着，我们可以尝试使用Git工具修改当前配置的内容，比如，将 config-repo/didispace-dev.properties 中的 from的值从from = git-dev-1.0修改为from = git-dev-2.0, 再访问http://localhost:7002/from, 可以看到其返回内容还是git-dev-1.0。



接下来， 我们将在config-client端做 一 些改造以实现配置信息的动态刷新。

4、 在config-client的pom.xml中新增spring-boot-starter-actuator监控模块。 其中包含了/refresh 端点的 实现，该端点将用于实现客户端应用配置信息的重新获取与刷新。

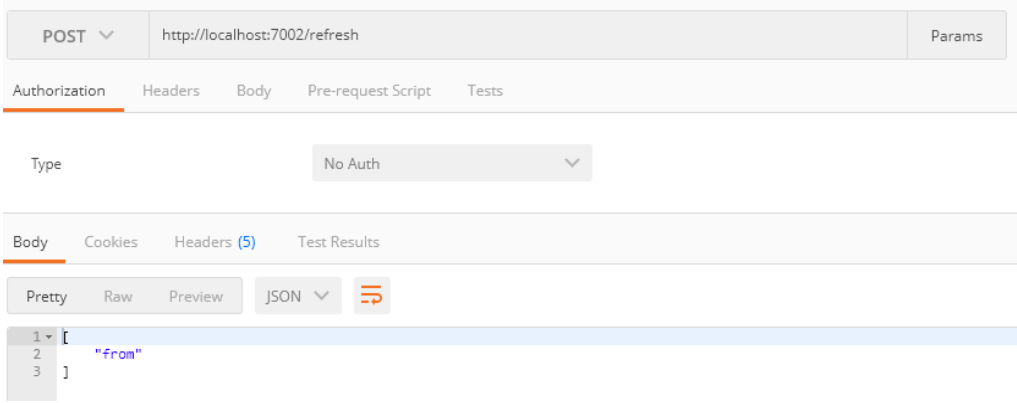
```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

5、 重新启动config-client ， 访问 一 次http://localhost:7002/frorn, 可以看到当前的配置值。

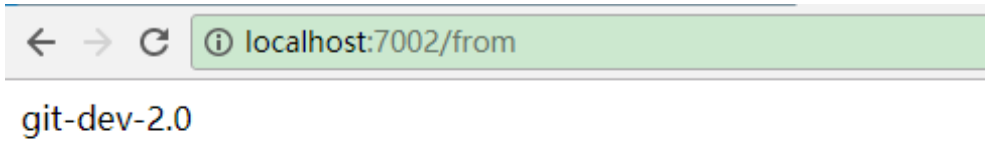
a) 修改Git仓库config-repo/didispace-dev.properties文件中from的值。

b) 再访间一 次http://localhost：7002/frorn, 可以看到配置值没有改变。

c) 通过POST请求发送到http://localhost:7002/refresh, 我们可以看到返回内容如下， 代表from参数的配置内容被更新了：



6、 再访问一 次http://localhost：7002/from, 可以看到配置值已经是更新后的值了。



7、 End

8. 未结事项与已结事项

8.1. 未结事项

序号	事项	解决方案	责任人	计划完成日期	实际完成日期

8.2. 已结事项

序号	事项	解决方案	责任人	计划完成日期	实际完成日期