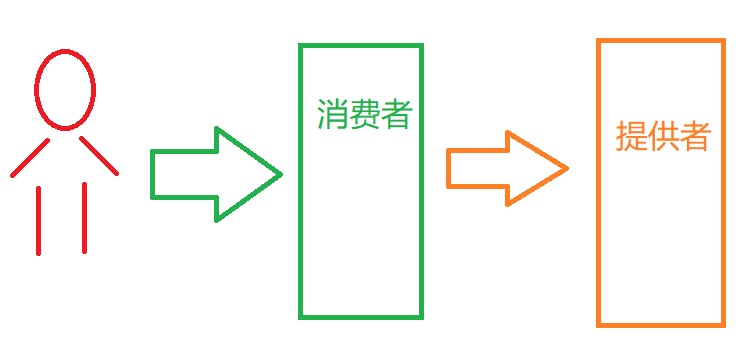
# springCloud高级

## 服务注册中心

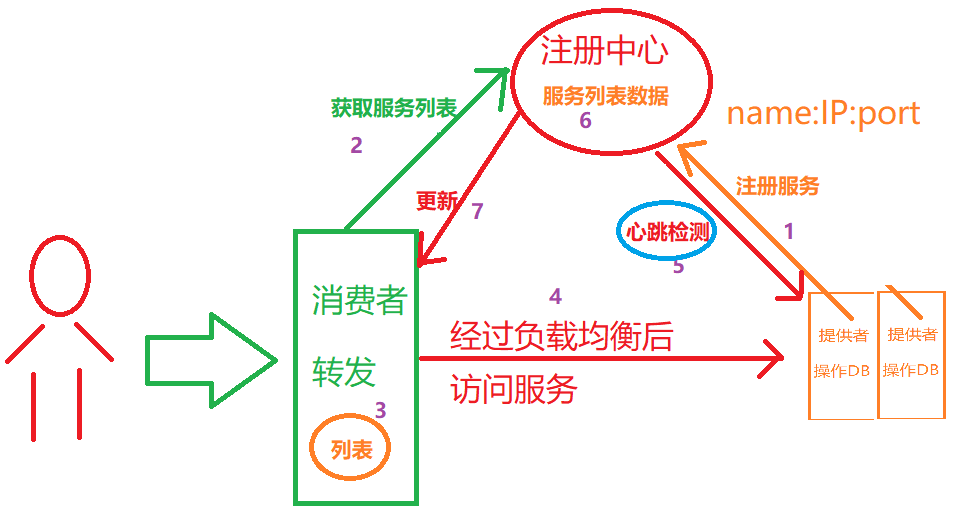
### 单台引用存在的问题

说明:如果遇到了高并发..则需要准备多台服务的提供者.如果有特殊需要才会准备多台的消费者.因为服务的消费者只负责请求的转发和响应.

由于服务的消费者是通过IP:端口的形式进行提供者的调用.如果有多个提供者.这时通过IP和端口调用则会出现问题.无法让IP和端口同时链接多台服务提供者.



### 注册中心原理



1. 当服务提供者启动时,将服务信息(服务名称/服务IP/端口)都写入注册中心中.
2. 当服务消费者启动时,链接注册中心.获取服务列表数据.
3. 将获取到的服务列表数据,保存到消费者本地.
4. 消费者通过服务名称获取全部提供者数据(多个),之后经过负载均衡策略.选出一个服务提供者进行访问.
5. 当服务提供者有一台宕机时,由注册中心,利用心跳检测机制,将服务改为down.
6. 当心跳检测生效时,动态的更新服务列表数据.
7. 当服务列表更新时,同步消费者的服务列表数据.

优点:用户不担心后台服务宕机.由注册中心动态的进行服务的切换.

## Eureka

### Eureka介绍

Eureka Server 提供服务发现的能力，各个微服务启动时，会向Eureka Server注册自己的信息（例如IP、端口、微服务名称等），Eureka Server会存储这些信息。

Eureka Client是一个Java客户端，用于简化与Eureka Server的交互

微服务启动后会周期性（默认30秒）地向Eureka Server发送心跳以续约自己的“租期”。

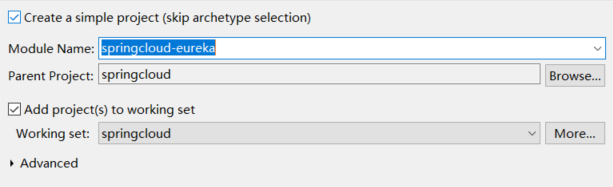
如果Eureka Server在一定时间内没有接收到某个微服务实例的心跳，Eureka Server将会注销该实例（默认90秒）

默认情况下，Eureka Server同时也是Eureka Client。多个Eureka Server实例，相互之间通过复制的方式，来实现服务注册表中的数据的同步。

Eureka Client 会缓存服务注册表中的信息。这种方式有一定的优势——首先，微服务无须每次请求都查询Eureka Server 的压力；

其次即使Eureka Server所有节点都宕掉，服务消费者依然可以使用缓存中过的信息找到服务提供者并完成调用

### 创建Eureka服务



### 编辑pom.xml文件

<dependencies>

<!--添加 eureka 注册中心 -->

<dependency>

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

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

</dependency>

</dependencies>

### 编辑yml文件

server:

port: 7000 #定义注册中心端口

eureka:

server:

enable-self-preservation: **true** #设定自我保护模式 默认值为true 不建议关闭

instance:

hostname: localhost #eureka服务的实例名称

client:

register-with-eureka: **false** #表示注册中心 不会注册自己本身

fetch-registry: **false** #表示自己就是注册中心,不需要检索服务

service-url:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

### 编辑主启动类

@SpringBootApplication(exclude=DataSourceAutoConfiguration.**class**)

@EnableEurekaServer //启动Eureka服务.

**public** **class** SpringBoot\_Run {

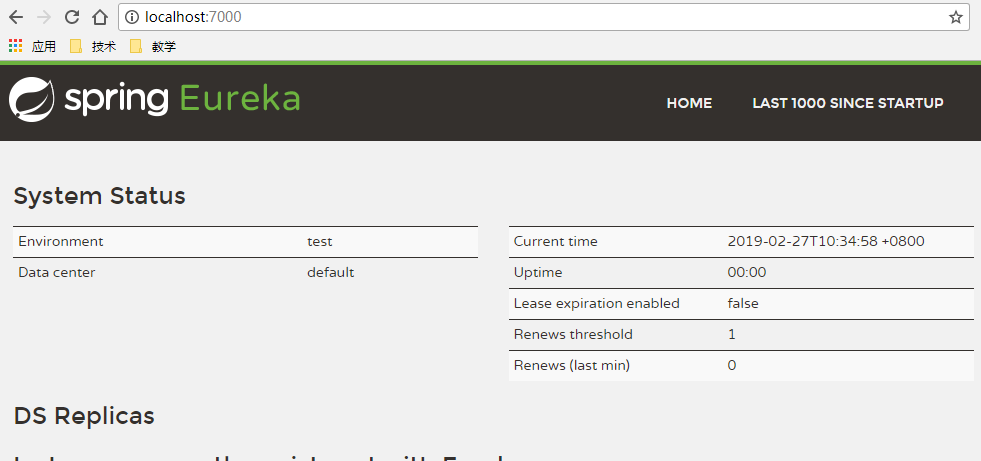
**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBoot\_Run.**class**, args);

}

}

### 启动效果



## 添加服务到注册中心

### 编辑客户端

<!--添加eureka 客户端地址 -->

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

### 编辑提供者配置文件

server:

port: 8000

servlet:

context-path: /

spring:

datasource:

#引入druid数据源

type: com.alibaba.druid.pool.DruidDataSource

driver-class-name: com.mysql.jdbc.Driver

url: jdbc:mysql://127.0.0.1:3306/jtdb?serverTimezone=GMT%2B8&useUnicode=true&characterEncoding=utf8&autoReconnect=true&allowMultiQueries=true

username: root

password: root

application: #定义服务名称

name: provider-user

eureka:

client:

service-url:

defaultZone: http://localhost:7000/eureka

instance:

instance-id: provider-user-8000 #定义微服务的名称

prefer-ip-address: **true** #是否显示IP和端口

#mybatis-plush配置

mybatis-plus:

type-aliases-package: com.jt.springcloud.pojo

mapper-locations: classpath:/mybatis/mappers/\*.xml

configuration:

map-underscore-to-camel-case: **true** #自动开启驼峰规则映射

### 编辑主启动类

@SpringBootApplication

@MapperScan("com.jt.springcloud.mapper")

@EnableEurekaClient //启动服务客户端

**public** **class** SpringBoot\_provider\_8000 {

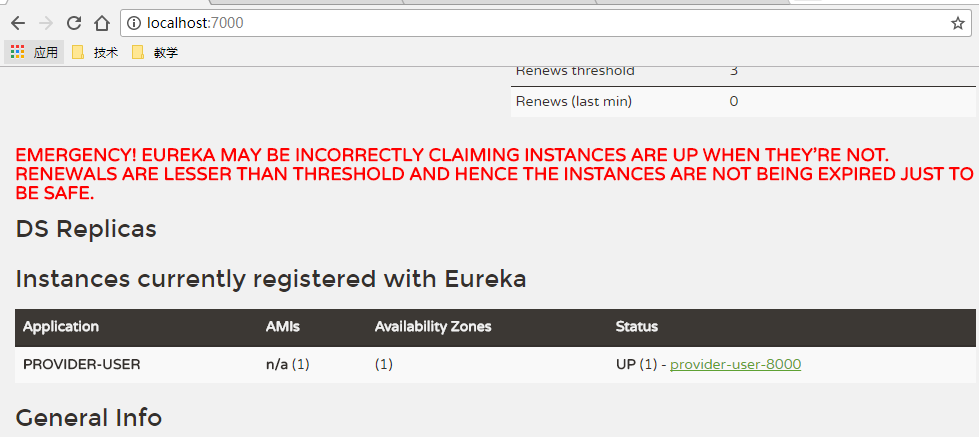
**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBoot\_provider\_8000.**class**, args);

}

}

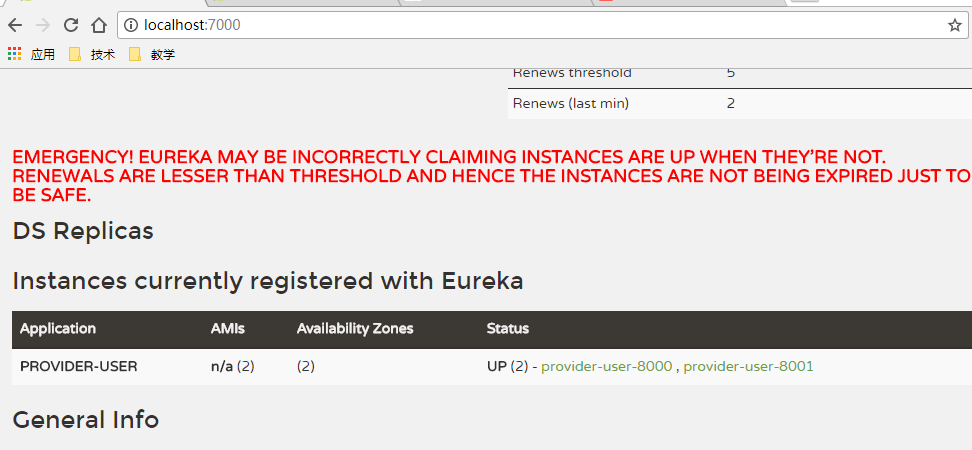
### 页面效果



如图所示.如果能够正常展现服务数据,则表示Eureka正确.

### 构建第二个提供者

根据第一个提供者的配置,拷贝配置文件.形成2个服务提供者



## 消费者注册中心配置

### 添加jar文件

<dependency>

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

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

</dependency>

<!--Eureka客户端配置 -->

<dependency>

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

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

</dependency>

### 编辑yml文件

server:

port: 8020

servlet:

context-path: /

eureka:

client:

register-with-eureka: **false** #不会将自己的信息注册到eureka中

service-url:

#链接注册中心集群

defaultZone: http://localhost:7000/eureka

### 编辑负载均衡策略

//一般使用springCloud时,如果是get请求采用Rest形式.

//如果请求是post提交时,后台直接接收JSON数据即可.

@Configuration

**public** **class** RestConfig {

@Bean

@LoadBalanced //使用负载均衡 轮询

**public** RestTemplate getBean() {

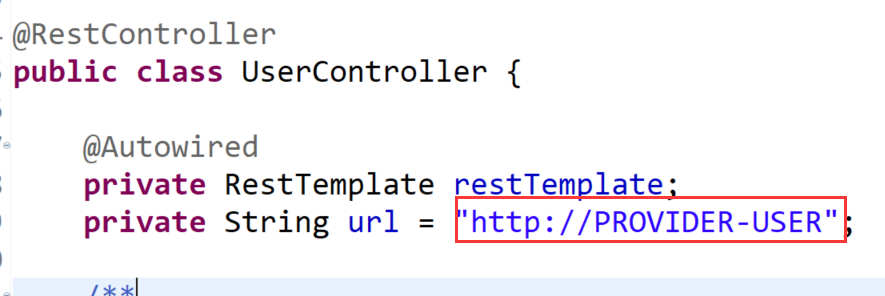
**return** **new** RestTemplate();

}

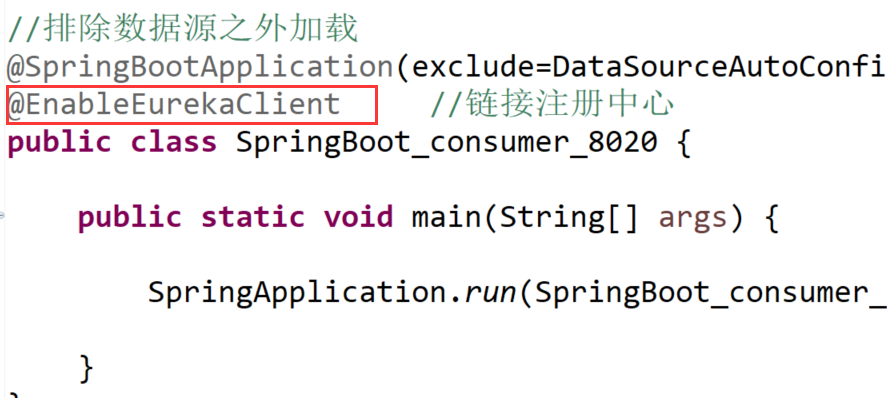
}

### 编辑消费者配置

1. 编辑Controller



1. 编辑主启动类



之后测试,检查是否有负载均衡.

### Ribbo介绍

核心:**ribbon是客户端负载均衡器**.用户发起请求后,直接获取真实的访问路径.而不需要像nginx那样等待集中式的分配.并且如果消费者宕机,则也不影响用户的使用.因为通过注册中心,可以动态的实现故障迁移.

### 负载均衡策略

@Bean

public IRule myRule() {

return new RoundRobinRule(); //轮询策略

//return new RandomRule(); //随机策略

//return new AvailabilityFilteringRule(); //首先会过滤掉故障机或者并发链接数超过阈值的服务器.剩余的机器轮询配置

//new WeightedResponseTimeRule(); //服务器影响时间越快,则权重越高

//new BestAvailableRule(); //最大可用策略，即先过滤出故障服务器后，选择一个当前并发请求数最小的;

}

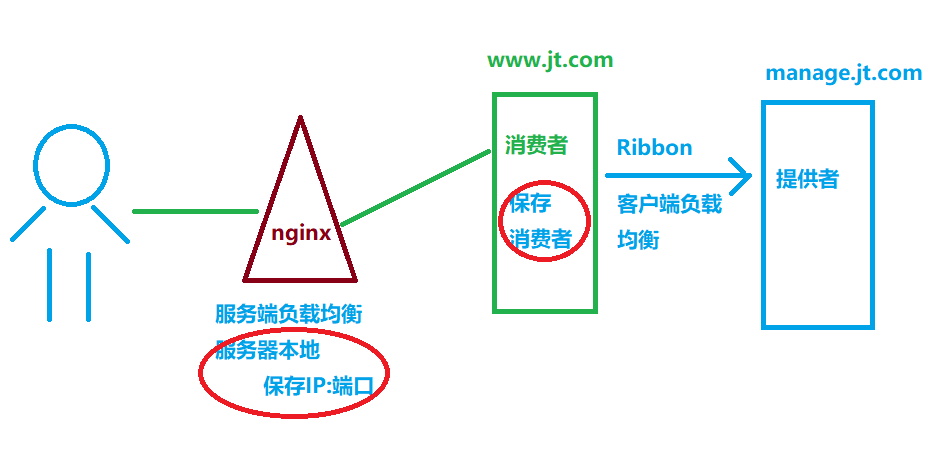
### Ribbon和Nginx区别

1. Ribbon是客户端负载均衡器.
2. Nginx主要作用实现反向代理.

使用场景不同:

Ribbon再能在SpringCloud中当消费者访问提供者时,实现负载均衡.

Nginx:是业务系统中,用户访问的统一入口.



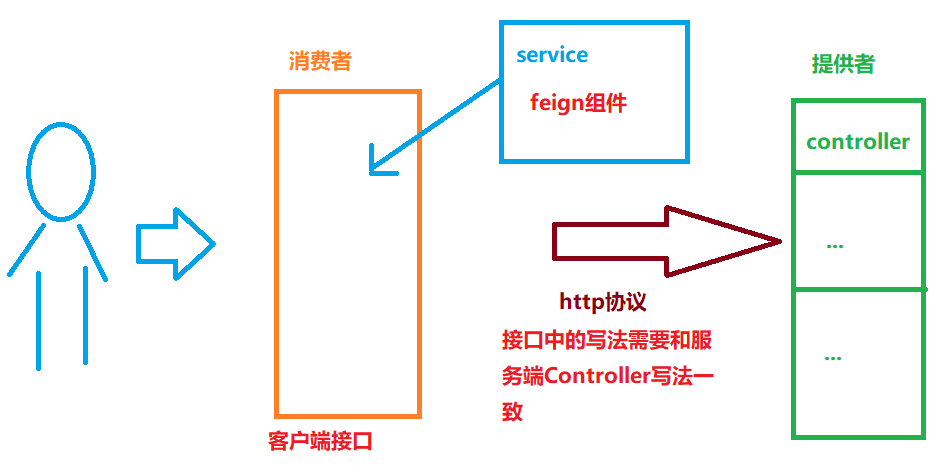
## Feign

### 介绍

Feign是一种声明式、模板化的HTTP**客户端**。简化客户端编码,可以使用户直接**通过接口的方式声明式的调用**,内部集成了ribbon.内部实现负载均衡

网址: <https://springcloud.cc/>

### Feign的工作原理



### 导入jar包

说明:在接口项目中添加jar包依赖

<!--引入Feign支持 -->

<dependency>

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

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

</dependency>

### 定义接口

在接口项目中,.添加业务接口UserService.指定接口访问的服务端是什么.

@FeignClient(value="provider-user")

//定义当前接口访问的提供者

**public** **interface** UserService {

@RequestMapping("/findAll") //提供者路径

**public** List<User> findAll();

@RequestMapping("/saveUser")

**public** String saveUser(@RequestBody User user);

}

### 消费者Controller编辑

@RestController

**public** **class** UserController {

@Autowired

**private** UserService userService;

@RequestMapping("/consumer/findAll")

**public** List<User> findUser(){

**return** userService.findAll();

}

//完成客户端新增

@RequestMapping("/consumer/saveUser/{name}/{age}/{sex}")

**public** String saveUser(User user) {

**return** userService.saveUser(user);

}

}

### 定义主启动类

//排除数据源之外加载

@SpringBootApplication(exclude=DataSourceAutoConfiguration.**class**)

@EnableEurekaClient //链接注册中心

@EnableFeignClients //添加feign的支持

**public** **class** SpringBoot\_consumer\_8020 {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBoot\_consumer\_8020.**class**, args);

}

}

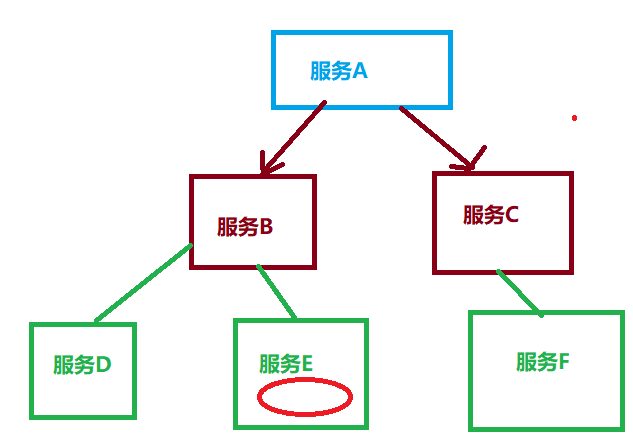
## Hytrix(断路器)

### 服务扇出

概念:多个微服务之间调用的时候,假设A服务调用B服务和C服务.B调用D或者其他更多的服务.称之为扇出

存在的问题:如果这时D服务因为执行较慢或者是服务不可用,导致A和B占用大量的系统资源,最终导致服务崩溃.

核心:用户请求**长时间无法获取响应结果**,造成服务一直等待,造成资源浪费.



### Hystrix

Hystrix是断路器.是一个用于处理分布式系统的延时和容错的开源库.在分布式系统里许多依赖可能会调用失败.hystrix能够保证当服务单元发生故障后,通过断路器机制.返回一个满足预期处理条件的数据.而不是长时间的等待或者抛出异常.这样就能避免服务长时间没有响应或者报错等影响,提升了软件的可靠性.

熔断机制:

是微服务中防止因为扇出,而导致服务雪崩的一种**保护机制**.

### 引入jar包

<!--添加断路器配置 -->

<dependency>

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

<artifactId>spring-cloud-starter-netflix-hystrix</artifactId>

</dependency>

### 编辑业务处理

//查询用户列表数据

@RequestMapping("/findAll")

@HystrixCommand(fallbackMethod="hystrix\_findAll")

**public** List<User> findAll(){

System.***out***.println("我是第一个服务");

**int** a = 1/0;

**return** userService.findAll();

}

**public** List<User> hystrix\_findAll(){

List<User> userList =

**new** ArrayList<User>();

User user = **new** User();

user.setId(0).setName("服务器异常").setAge(0)

.setSex("");

userList.add(user);

**return** userList;

}

### 编辑主启动类

@SpringBootApplication

@MapperScan("com.jt.springcloud.mapper")

@EnableEurekaClient //启动服务客户端

@EnableHystrix //启动熔断机制

**public** **class** SpringBoot\_provider\_8000 {

**public** **static** **void** main(String[] args) {

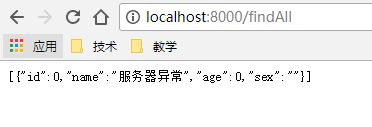
SpringApplication.*run*(SpringBoot\_provider\_8000.**class**, args);

}

}

### 熔断机制测试

暂时访问服务端程序.当程序执行有误时,通过断路器执行熔断操作,返回一个满足预期结果的数据.减少资源的浪费.



## 断路器配置优化

### 问题说明

如果断路器机制只配置到后台,如果后台的服务器宕机.则用户无法获取熔断数据.直接报错返回.

优化:在消费者接口中指定断路器机制,如果服务的提供者宕机了,则采用服务消费端的断路器机制.

### 编辑工厂方法

@Component //交给Spring容器管理

**public** **class** UserFallbackFactory

**implements** FallbackFactory<UserService>{

@Override

**public** UserService create(Throwable cause) {

**return** **new** UserService() {

//该接口方法,完成当提供者出错时,返回的有效数据

@Override

**public** String saveUser(User user) {

**return** "后台服务器异常";

}

@Override

**public** List<User> findAll() {

List<User> userList =

**new** ArrayList<User>();

User user = **new** User();

user.setId(0).setName("后台服务器异常").setAge(0)

.setSex("");

userList.add(user);

**return** userList;

}

};

}

}

### 编辑接口文件

@FeignClient(value="provider-user",

fallbackFactory=UserFallbackFactory.**class**)

//定义当前接口访问的提供者

**public** **interface** UserService {

@RequestMapping("/findAll") //提供者路径

**public** List<User> findAll();

@RequestMapping("/saveUser")

**public** String saveUser(@RequestBody User user);

}

### 编辑客户端yml文件

server:

port: 8020

servlet:

context-path: /

#添加服务的名称

spring:

application: #定义服务名称

name: consumer-user

eureka:

client:

service-url:

defaultZone: http://localhost:7000/eureka

instance:

instance-id: consumer-user-8020 #定义微服务的名称

prefer-ip-address: **true** #是否显示IP和端口

feign:

hystrix:

enabled: **true** #启动熔断器机制 !!!!!!!!!!!!!

hystrix:

command:

default:

execution:

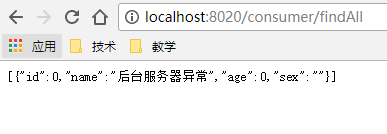
isolation:

thread:

timeoutInMilliseconds: 3000 #设定断路器超时时间

### 断路器测试

当提供者宕机时,进行熔断.



### 服务降级

说明:当当前服务器的资源不能满足用户的需求时.一般会将不重要的业务服务器暂时关闭/乃至整个服务停止.因为有熔断机制.返回友好的数据信息.当用户的访问趋于平稳,则再次调集资源启动服务.这种方式称之为服务降级.

## 服务监控

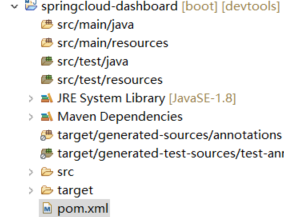
### 说明

Hystrix会记录用户的所有访问请求,并且将统计数据以报表的形式展现给用户.包括执行时间/成功和失败次数.功能强大.

通过记录服务器响应的数据.为管理者提供数据的支持.

### 创建项目

说明:仪表盘监控是一个客户端-服务器系统.首先搭建服务器.



### 添加jar包

<dependencies>

<!--添加断路器配置 -->

<dependency>

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

<artifactId>spring-cloud-starter-netflix-hystrix</artifactId>

</dependency>

<!--配置hystrix监控 -->

<dependency>

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

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

</dependency>

</dependencies>

### 编辑yml配置

server:

port: 9000

### 编辑主启动类

@SpringBootApplication(exclude=DataSourceAutoConfiguration.**class**)

@EnableHystrixDashboard //启动仪表盘监控程序

**public** **class** SpringCloud\_Dashboard {

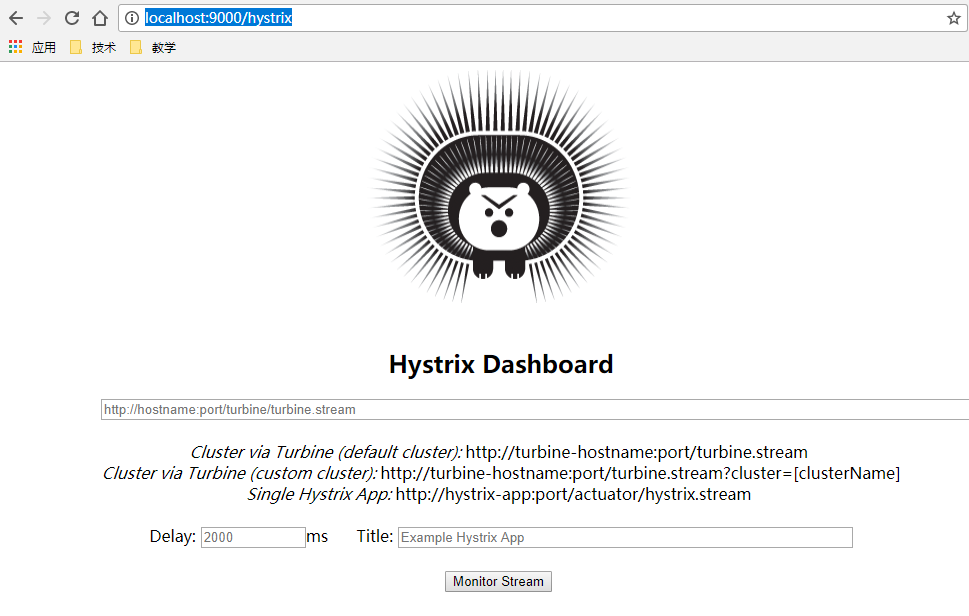
**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringCloud\_Dashboard.**class**, args);

}

}

### 访问效果



## 编辑客户端

### 添加jar包

在服务提供者的pom.xml文件中添加监控jar包

<!--添加监控 -->

<dependency>

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

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

</dependency>

### 编辑配置类

@Configuration

**public** **class** DashboardConfig {

//添加Servlet配置

@Bean

**public** ServletRegistrationBean<Servlet> getServlet(){

HystrixMetricsStreamServlet streamServlet = **new** HystrixMetricsStreamServlet();

ServletRegistrationBean<Servlet> registrationBean = **new** ServletRegistrationBean<>(streamServlet);

registrationBean.setLoadOnStartup(1);

registrationBean.addUrlMappings("/actuator/hystrix.stream");

registrationBean.setName("HystrixMetricsStreamServlet");

**return** registrationBean;

}

}

### 项目测试

url地址: <http://localhost:8000/actuator/hystrix.stream>

注意事项:添加断路器注解

//查询用户列表数据

@RequestMapping("/findAll")

**@HystrixCommand(fallbackMethod="hystrix\_findAll")**

**public** List<User> findAll(){

System.***out***.println("我是第一个服务");

//int a = 1/0;

**return** userService.findAll();

}

