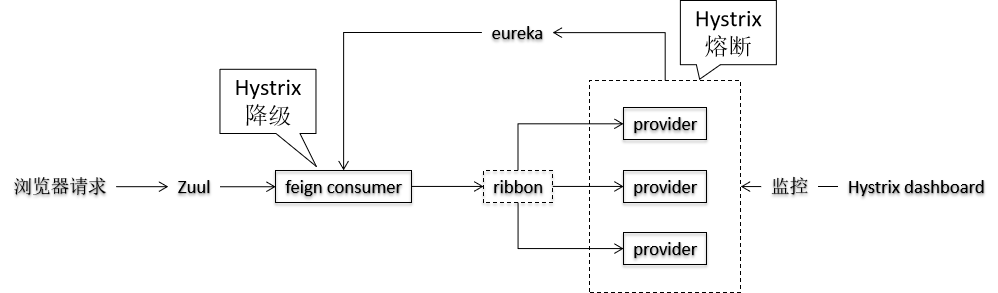
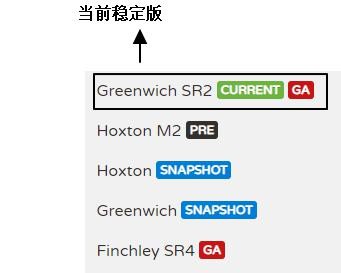
SpringCloud

# 基本组件

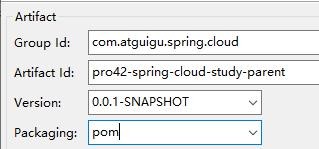
* + 注册中心：Eureka
  + 负载均衡：Ribbon
  + 声明式调用远程方法：Feign
  + 熔断、降级、监控：Hystrix
  + 网关：Zuul

# 使用版本



1. **目标 1：准备基础测试环境**

## 结构

* 1. **创建父工程**

配置依赖管理

<dependencyManagement>

<dependencies>

<!-- 导入 SpringCloud 需要使用的依赖信息 -->

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>Greenwich.SR2</version>

<type>pom</type>

<!-- import 依赖范围表示将spring-cloud-dependencies 包中的依赖信息导入 -->

<scope>import</scope>

</dependency>

<!-- 导入 SpringBoot 需要使用的依赖信息 -->

<dependency>

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

<artifactId>spring-boot-dependencies</artifactId>

<version>2.1.6.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

## 创建通用工程

<parent>

<groupId>com.atguigu.spring.cloud</groupId>

<artifactId>pro42-spring-cloud-study-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

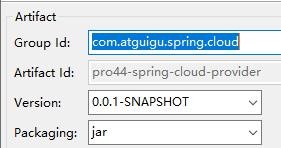
</parent>

<artifactId>pro43-spring-cloud-common</artifactId>

public class Employee {

* 1. **创建提供者工程**

private Integer empId; private String empName; private Double empSalary;



加入如下依赖信息：

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

<groupId>com.atguigu.spring.cloud</groupId>

<artifactId>pro43-spring-cloud-common</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

</dependencies>

创建主启动类：

@SpringBootApplication public class AtguiguMainType {

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

}

}

创建 application.yml 配置文件：

server:

port: 1000

创建 handler 类和方法：

@RestController

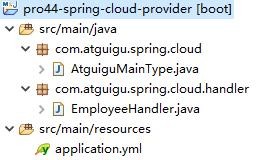
public class EmployeeHandler {

@RequestMapping("/provider/get/employee/remote") public Employee getEmployeeRemote() {

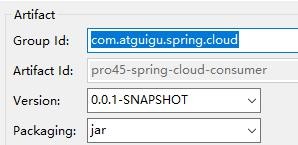
return new Employee(555, "tom555", 555.55);

}

}



## 创建消费者工程



加入下面依赖：

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

<groupId>com.atguigu.spring.cloud</groupId>

<artifactId>pro43-spring-cloud-common</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

</dependencies>

创建主启动类：

@SpringBootApplication

public class AtguiguMainType {

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

}

}

创建配置类提供 RestTemplate：

@Configuration

public class AtguiguCloudConfig {

@Bean

public RestTemplate getRestTemplate() { return new RestTemplate();

}

}

创建 handler 类：

@RestController

public class HumanResourceHandler {

@Autowired

private RestTemplate restTemplate;

@RequestMapping("/consumer/get/employee") public Employee getEmployeeRemote() {

// 远程调用方法的主机地址

String host = "http://localhost:1000";

// 远程调用方法的具体 URL 地址

String url = "/provider/get/employee/remote";

return restTemplate.getForObject(host + url, Employee.class);

}

}

创建 application.yml：

server:

port: 4000

# 目标 2：创建 Eureka 注册中心

## 子目标 1：创建 Eureka 注册中心工程

加入如下依赖信息：

<dependency>

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

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

</dependency>

创建主启动类：

// 启用 Eureka 服务器功能**@EnableEurekaServer** @SpringBootApplication public class AtguiguMainType {

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

}

}

创建 application.yml：

server:

port: 5000

eureka:

instance:

hostname: localhost client:

registerWithEureka: false # 自己就是注册中心，所以自己不注册自己fetchRegistry: false # 自己就是注册中心，所以不需要“从注册中心取回信息” serviceUrl: # 客户端访问 Eureka 时使用的地址

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

## 子目标 2：将 provider 注册到Eureka

加入如下依赖：

<dependency>

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

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

</dependency>

配置 application.yml：

eureka:

client:

serviceUrl:

defaultZone: http://localhost:5000/eureka/

※关于相关注解

较低版本需要使用@EnableEurekaClient 注解。

稍高版本也可以使用@EnableDiscoveryClient 注解。当前版本可以省略。

注册的效果：



这里显示 UNKOWN 是因为 provider 工程没有指定应用名称，指定应用名称配置方式如下：

重启 provider 后，再查看注册情况：

spring:

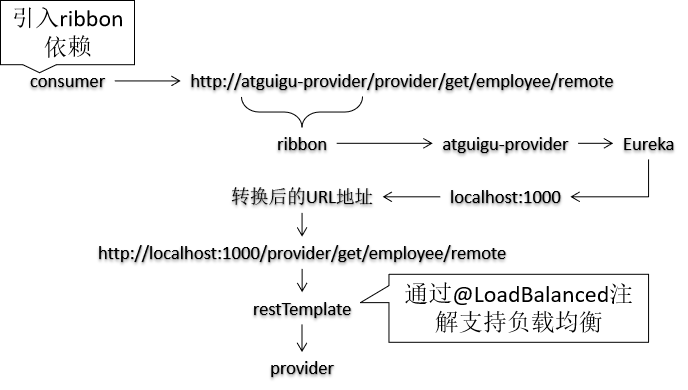
application:

name: atguigu-provider

以后在 SpringCloud 环境下开发，每一个微服务工程都要设置一个应用名称。

# 目标 3：consumer 访问 provider 时使用微服务名称代替 localhost:1000

## 分析



* 1. **操作**

在 consumer 工程加入如下依赖：

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

在 application.yml 中加入如下配置：

spring:

application:

name: atguigu-consumer eureka:

client:

serviceUrl:

defaultZone: http://localhost:5000/eureka/

在 RestTemplate 的配置方法处使用@LoadBalanced 注解：

@Configuration

public class AtguiguCloudConfig {

@Bean

**@LoadBalanced**

public RestTemplate getRestTemplate() { return new RestTemplate();

}

}

修改 consumer 工程的 handler 方法：

@RequestMapping("/consumer/ribbon/get/employee") public Employee getEmployeeRemote() {

// 远程调用方法的主机地址

// String host = "http://**localhost:1000**";

// 引入 Eureka 和 Ribbon 后，就可以使用微服务名称替代 IP 地址+端口号

String host = ["http://**atguigu-provider**](http://atguigu-provider/)";

// 远程调用方法的具体 URL 地址

String url = "/provider/get/employee/remote";

return restTemplate.getForObject(host + url, Employee.class);

}

# 目标 4：provider 以集群方式启动

## 修改provider 的handler 方法

@RequestMapping("/provider/get/employee/remote")

public Employee getEmployeeRemote(HttpServletRequest request) {

// 获取当前 Web 应用的端口号

int serverPort = request.getServerPort();

return new Employee(555, "tom555-"+serverPort, 555.55);

}

* 1. **provider 以集群方式启动**

按照端口号 1000 启动第一个实例

按照端口号 2000 启动第二个实例

按照端口号 3000 启动第三个实例

## consumer 正常访问

* 1. **注意**

provider 的微服务名称必须使用**同一个名称**才能构成一个集群，否则将不会认定为是属于同一个集群。

# 目标 5：使用 Feign 实现远程方法声明式调用

## 分析

* 1. **操作**
     1. **common 工程**引入如下依赖：

<dependencies>

<dependency>

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

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

</dependency>

</dependencies>

创建远程调用方法的接口：

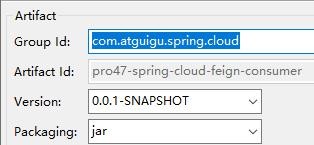
@FeignClient("atguigu-provider")

public interface EmployeeRemoteService {

@RequestMapping("/provider/get/employee/remote") public Employee getEmployeeRemote();

}

### 新建 Feign-consumer 工程



加入如下依赖：

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

<groupId>com.atguigu.spring.cloud</groupId>

<artifactId>pro43-spring-cloud-common</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

主启动类：

// 启用 Feign 客户端功能**@EnableFeignClients** @SpringBootApplication public class AtguiguMainType {

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

}

}

application.yml

server:

port: 7000 spring:

application:

name: atguigu-feign-consumer eureka:

client:

serviceUrl:

defaultZone: http://localhost:5000/eureka/ handler 类：

@RestController

public class EmployeeFeignHandler {

@Autowired

private EmployeeRemoteService employeeRemoteService;

@RequestMapping("/feign/consumer/get/emp") public Employee getEmployeeRemote() {

return employeeRemoteService.getEmployeeRemote();

}

}

## 传参

* + 1. **简单类型**接口写法：

@RequestMapping("/provider/get/employee/by/id")

public Employee getEmployeeById(**@RequestParam("empId")** Integer empId);

别忘了写@RequestParam 注解

provider 的 handler 方法：

@RequestMapping("/provider/get/employee/by/id")

public Employee getEmployeeById(@RequestParam("empId") Integer empId) {

return new Employee(empId, "tom999-", 999.99);

}

方法声明部分和接口中一致

* + 1. **复杂类型**接口写法：

@RequestMapping("/provider/save/emp")

public Employee saveEmp(**@RequestBody** Employee employee);

别忘了写@RequestBody 注解

provider 的 handler 方法：

@RequestMapping("/provider/save/emp")

public Employee saveEmp(@RequestBody Employee employee) { return employee;

}

方法声明部分和接口中一致

# Hystrix

## 分布式系统面临的问题

在微服务架构体系下，服务间的调用错综复杂，交织成一张大网。如果其中某个节 点突然无法正常工作，则访问它的众多服务都会被卡住，进而有更多服务被卡住，系统 中的线程、CPU、内存等资源有可能被迅速耗尽，最终整个服务体系崩溃。

我们管这样的现象叫服务雪崩。



## Hytrix 介绍

Hystrix 是一个用于处理分布式系统的延迟和容错的开源库，在分布式系统里，许多依赖不可避免的会调用失败，比如超时、异常等，Hystrix 能够保证在一个依赖出问题的情况下，不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性。

“断路器”本身是一种开关装置，当某个服务单元发生故障之后，通过断路器的故 障监控（类似熔断保险丝），向调用方返回一个符合预期的、可处理的备选响应

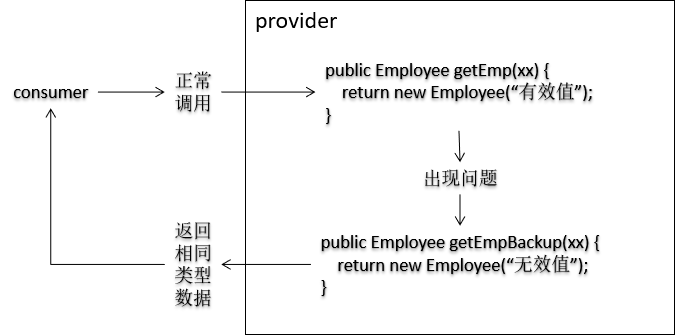
（FallBack），而不是长时间的等待或者抛出调用方无法处理的异常，这样就保证了服务调用方的线程不会被长时间、不必要地占用，从而避免了故障在分布式系统中的蔓延， 乃至雪崩。

Hytrix 能够提供服务降级、服务熔断、服务限流、接近实时的监控等方面的功能。

## 服务熔断机制

熔断机制是应对雪崩效应的一种微服务链路保护机制。

当扇出链路的某个微服务不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速响应错误信息。当检测到该节点微服务调用响应正常后恢复调用链路。在 SpringCloud 框架里熔断机制通过 Hystrix 实现。Hystrix 会监控微服务间调用的状况， 当失败的调用到一定阈值，缺省是 5 秒内 20 次调用失败就会启动熔断机制。熔断机制的注解是@HystrixCommand。



### 依赖信息

<dependency>

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

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

</dependency>

* + 1. **主启动类注解**

// 启用断路器功能**@EnableCircuitBreaker** @SpringBootApplication public class AtguiguMainType {

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

}

}

* + 1. **ResultEntity**

/\*\*

* 整个项目统一使用这个类型作为 Ajax 请求或远程方法调用返回响应的数据格式
* @author Lenovo

\*

* @param <T>

\*/

public class ResultEntity<T> {

public static final String SUCCESS = "SUCCESS"; public static final String FAILED = "FAILED";

public static final String NO\_MESSAGE = "NO\_MESSAGE"; public static final String NO\_DATA = "NO\_DATA";

/\*\*

* 操作成功，不需要返回数据
* @return

\*/

public static ResultEntity<String> successWithoutData() {

return new ResultEntity<String>(SUCCESS, NO\_MESSAGE, NO\_DATA);

}

/\*\*

* 操作成功，需要返回数据
* @param data
* @return

\*/

public static <E> ResultEntity<E> successWithData(E data) { return new ResultEntity<>(SUCCESS, NO\_MESSAGE, data);

}

/\*\*

* 操作失败，返回错误消息
* @param message
* @return

\*/

public static <E> ResultEntity<E> failed(String message) { return new ResultEntity<>(FAILED, message, null);

}

private String result; private String message; private T data;

public ResultEntity() {

// TODO Auto-generated constructor stub

}

public ResultEntity(String result, String message, T data) {

## —————————————————————————————

super();

this.result = result; this.message = message; this.data = data;

}

@Override

public String toString() {

return "ResultEntity [result=" + result + ", message=" + message + ", data=" + data + "]";

}

public String getResult() { return result;

}

public void setResult(String result) { this.result = result;

}

public String getMessage() { return message;

}

public void setMessage(String message) { this.message = message;

}

public T getData() { return data;

}

public void setData(T data) { this.data = data;

}

}

### handler 方法

// @HystrixCommand 注解通过 fallbackMethod 属性指定断路情况下要调用的备份方法**@HystrixCommand**(**fallbackMethod** = "**getEmpBackup**") @RequestMapping("/provider/circuit/breaker/get/emp")

public ResultEntity<Employee> getEmp(@RequestParam("signal") String signal) {

if("bang".equals(signal)) {

throw new RuntimeException();

}

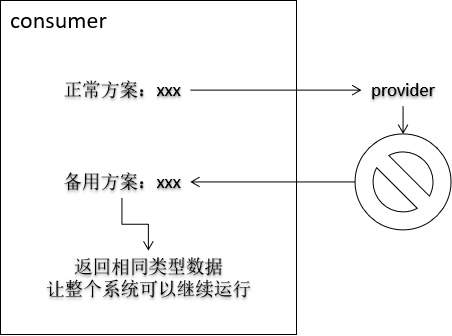
return ResultEntity.successWithData(new Employee(666, "sam666", 666.66));

}

public ResultEntity<Employee> **getEmpBackup**(@RequestParam("signal") String signal) { return ResultEntity.failed("circuit break workded,with signal="+signal);

}

* 1. **服务降级机制**

服务降级处理是在客户端(Consumer 端)实现完成的，与服务端(Provider 端)没有关系。当某个 Consumer 访问一个 Provider 却迟迟得不到响应时执行预先设定好的一个解决方案， 而不是一直等待。

### common 工程：依赖

<dependency>

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

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

</dependency>

* + 1. **common 工程：FallbackFactory**

// 请注意自动扫描包的规则

// 比如：feign-consumer 工程需要使用 MyFallBackFactory，那么 MyFallBackFactory 应该在

feign-consumer 工程的主启动类所在包或它的子包下

// 简单来说：哪个工程用这个类，哪个工程必须想办法扫描到这个类

@Component

public class MyFallBackFactory implements FallbackFactory<EmployeeRemoteService> {

// cause 对象是失败原因对应的异常对象

@Override

public EmployeeRemoteService create(Throwable cause) { return new EmployeeRemoteService() {

@RequestMapping("/provider/save/emp")

public Employee saveEmp(@RequestBody Employee employee) { return null;

}

@RequestMapping("/provider/get/employee/by/id")

public Employee getEmployeeById(@RequestParam("empId") Integer empId) { return null;

}

@RequestMapping("/provider/get/employee/remote") public Employee getEmployeeRemote() {

return new Employee(444, "call provider failed,fall back here, reason is "+cause.getClass().getName()+" "+cause.getMessage(), 444.444);

}

};

}

}

* + 1. **common 工程：Feign 接口**

// 在@FeignClient 注解中增加 fallbackFactory 属性

// 指定 consumer 调用 provider 时如果失败所采取的备用方案

// fallbackFactory 指定 FallbackFactory 类型的类，保证备用方案返回相同类型的数据

@FeignClient(value="atguigu-provider", fallbackFactory=MyFallBackFactory.class) public interface EmployeeRemoteService {

……

* + 1. **consumer 工程：application.yml**

feign:

hystrix:

enabled: true

* + 1. **测试**

正常访问：



人为把 provider 停掉



## 监控

* + 1. **provider 工程**加入如下依赖：

<dependency>

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

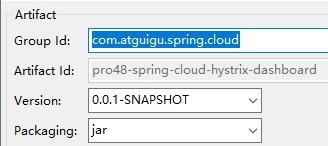
<artifactId>spring-**boot-**starter-actuator</artifactId>

</dependency>

配置 application.yml

management.endpoints.web.exposure.include: hystrix.stream

### 监控工程



<dependency>

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

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

</dependency>

// 启用 Hystrix 仪表盘功能**@EnableHystrixDashboard** @SpringBootApplication public class AtguiguMainType {

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

}

}

server:

port: 8000 spring:

application:

name: atguigu-dashboard

* + 1. **查看监控数据**
       - 直接查看监控数据本身

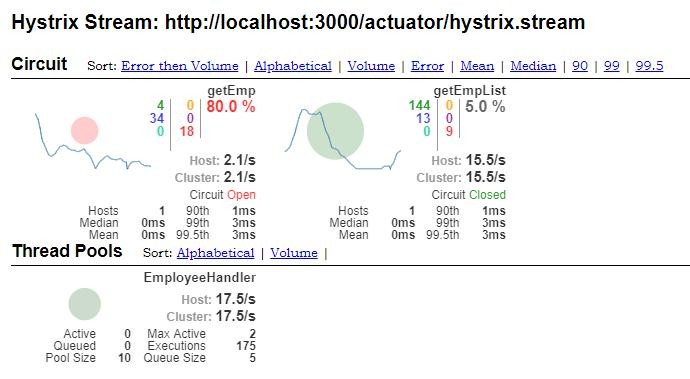
http://localhost:3000/actuator/hystrix.stream

说明 1：http://localhost:3000 访问的是被监控的 provider 工程说明 2：/actuator/hystrix.stream 是固定格式

说明 3：如果从 provider 启动开始它的方法没有被访问过，那么显示的数据只有“ping:”，要实际访问一个带熔断功能的方法才会有实际数据。

* + - * 通过仪表盘工程访问监控数据
        + 第一步：打开仪表盘工程的首页

http://localhost:8000/hystrix

* + - * + 第二步：填入获取监控数据的地址（上面直接查看时使用的地址）

# Zuul 网关

不同的微服务一般有不同的网络地址，而外部的客户端可能需要调用多个服务的接口才 能完成一个业务需求。比如一个电影购票的手机 APP，可能会调用电影分类微服务，用户微服务，支付微服务等。如果客户端直接和微服务进行通信，会存在以下问题：

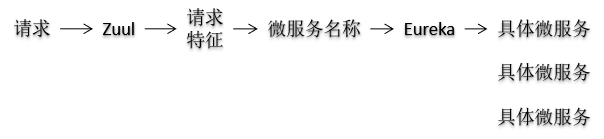
* + 客户端会多次请求不同微服务，增加客户端的复杂性
  + 存在跨域请求，在一定场景下处理相对复杂
  + 认证复杂，每一个服务都需要独立认证
  + 难以重构，随着项目的迭代，可能需要重新划分微服务，如果客户端直接和微服务 通信，那么重构会难以实施
  + 某些微服务可能使用了其他协议，直接访问有一定困难

Zuul 包含了对请求的路由和过滤两个最主要的功能：

其中路由功能负责将外部请求转发到具体的微服务实例上，是实现外部访问统一入口的 基础而过滤器功能则负责对请求的处理过程进行干预，是实现请求校验、服务聚合等功能的

基础。

Zuul 和 Eureka 进行整合，将 Zuul 自身注册为 Eureka 服务治理下的应用，同时从 Eureka

中获得其他微服务的信息，也即以后的访问微服务都是通过 Zuul 跳转后获得。总体来说，Zuul 提供了**代理**、**路由**和**过滤**的功能。

## 创建Zuul 工程

加入如下依赖：

<dependencies>

<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-netflix-zuul</artifactId>

</dependency>

</dependencies>

配置 applicaton.yml

server:

port: 9000 spring:

application:

name: zuul-gateway eureka:

client:

serviceUrl:

defaultZone: http://localhost:5000/eureka/

主启动类

// 启用 Zuul 代理功能**@EnableZuulProxy** @SpringBootApplication public class AtguiguMainType {

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

}

}

## 访问测试

### 初步访问

此时：通过 Zuul 可以访问，也可以不经过 Zuul 直接访问目标微服务。

### 使用指定地址代替微服务名称

zuul:

routes:

**employee**: # 自定义路由规则的名称，在底层的数据结构中是 Map 的键serviceId: atguigu-feign-consumer # 目标微服务名称，ZuulRoute 类型的一个属性path: /zuul-emp/\*\* # 用来代替目标微服务名称的路径，

ZuulRoute 类型的一个属性

# /\*\*表示匹配多层路径，如果没有加/\*\*则不能匹配后续的多层路径了效果：使用微服务名称和新配置的地址都可以访问

http://localhost:9000/atguigu-feign-consumer/feign/consumer/get/emp http://localhost:9000/zuul-emp/feign/consumer/get/emp

### 让用户不能通过微服务名称访问

zuul:

**ignored-services**: # 忽略指定微服务名称，让用户不能通过微服务名称访问

- **atguigu-feign-consumer**

routes:

employee: # 自定义路由规则的名称，在底层的数据结构中是 Map 的键serviceId: atguigu-feign-consumer # 目标微服务名称，ZuulRoute 类型的一个属性path: /zuul-emp/\*\* # 用来代替目标微服务名称的路径，

ZuulRoute 类型的一个属性

# /\*\*表示匹配多层路径，如果没有加/\*\*则不能匹配后续的多层路径了效果：微服务名称不能访问，只有新配置的地址可以访问

~~http://localhost:9000/atguigu-feign-consumer/feign/consumer/get/emp~~ http://localhost:9000/zuul-emp/feign/consumer/get/emp

### 忽略所有微服务名称

serviceId: atguigu-feign-consumer # 目标微服务名称，ZuulRoute 类型的一个属性

path: /zuul-emp/\*\* # 用来代替目标微服务名称的路径，

ZuulRoute 类型的一个属性

# /\*\*表示匹配多层路径，如果没有加/\*\*则不能匹配后续的多层路径了

# 自定义路由规则的名称，在底层的数据结构中是 Map 的键

# 忽略所有微服务名称

**ignored-services: '\*'**

routes:

employee:

zuul:

# ignored-services: 忽略指定微服务名称，让用户不能通过微服务名称访问

# - atguigu-feign-consumer

* + 1. **给访问路径添加统一前缀**

zuul:

# ignored-services: 忽略指定微服务名称，让用户不能通过微服务名称访问

# - atguigu-feign-consumer

ignored-services: '\*' # 忽略所有微服务名称**prefix: /maomi** # 给访问路径添加统一前缀routes:

employee: # 自定义路由规则的名称，在底层的数据结构中是 Map 的键serviceId: atguigu-feign-consumer # 目标微服务名称，ZuulRoute 类型的一个属性path: /zuul-emp/\*\* # 用来代替目标微服务名称的路径，

ZuulRoute 类型的一个属性

# /\*\*表示匹配多层路径，如果没有加/\*\*则不能匹配后续的多层路径了

http://localhost:9000**/maomi**/zuul-emp/feign/consumer/get/emp

## ZuulFilter

@Component

public class MyZuulFilter extends ZuulFilter {

**—————————————————————————————**

@Override

public boolean shouldFilter() {

// 1.获取当前 RequestContext 对象

RequestContext context = RequestContext.getCurrentContext();

// 2.获取当前请求对象

HttpServletRequest request = context.getRequest();

// 3.获取当前请求要访问的目标地址String servletPath = request.getServletPath();

// 4.打印System.err.println("servletPath="+servletPath);

// 5.当前方法返回值

// true：表示应该过滤，下面继续执行 run()方法

// false：表示不过滤，直接放行return true;

}

@Override

public Object run() throws ZuulException {

// 执行具体过滤逻辑

System.err.println("run() ...");

// 官方文档说：当前实现会忽略这个返回值，所以返回 null 即可

return null;

}

@Override

public String filterType() {

// 返回当前过滤器类型

// 可选类型包括：pre、route、post、static

// 如果需要在目标微服务前面执行过滤操作，选用 pre 类型

return "pre";

}

@Override

## —————————————————————————————

public int filterOrder() {

// 过滤器执行顺序

return 0;

}

}

* 1. **使用Zuul 进行用户是否登录的检查大致方案**

