

Spring Cloud

2021年3月31日 13:25

为什么要学习Spring Cloud

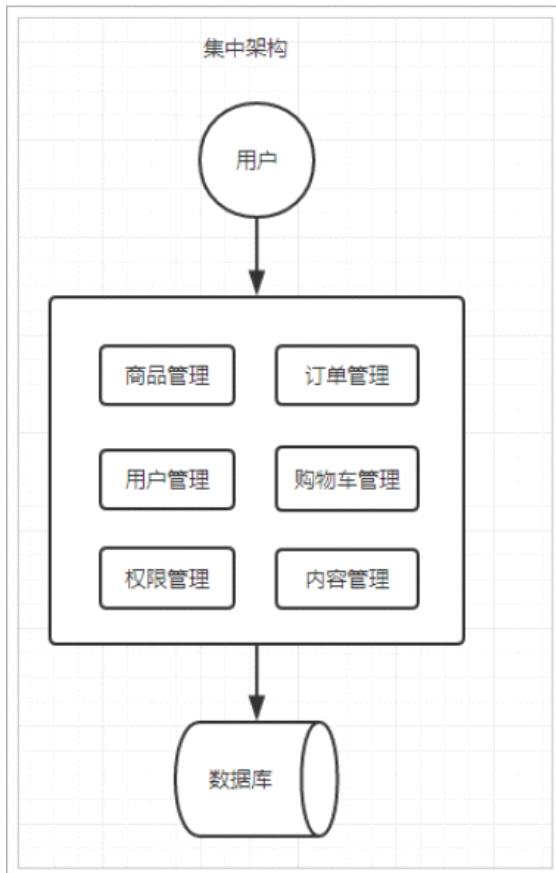
在项目开发中随着业务越来越多，导致功能之间耦合性高、开发效率低、系统运行缓慢难以维护、不稳定。微服务架构可以解决这些问题，而Spring Cloud是微服务架构最流行的实现。

1、系统架构演变

随着互联网的发展，网站应用的规模不断扩大，需求激增，随之而来的是技术上的压力。系统架构也因此不断的演进、升级、迭代。从单一应用，到垂直拆分，到分布式服务，再到SOA，以及现在火热的微服务架构。

1.1 集中式架构

当网站流量很小时，只需要一个应用，将所有的功能都部署在一起，以减少部署节点和成本。



优点：

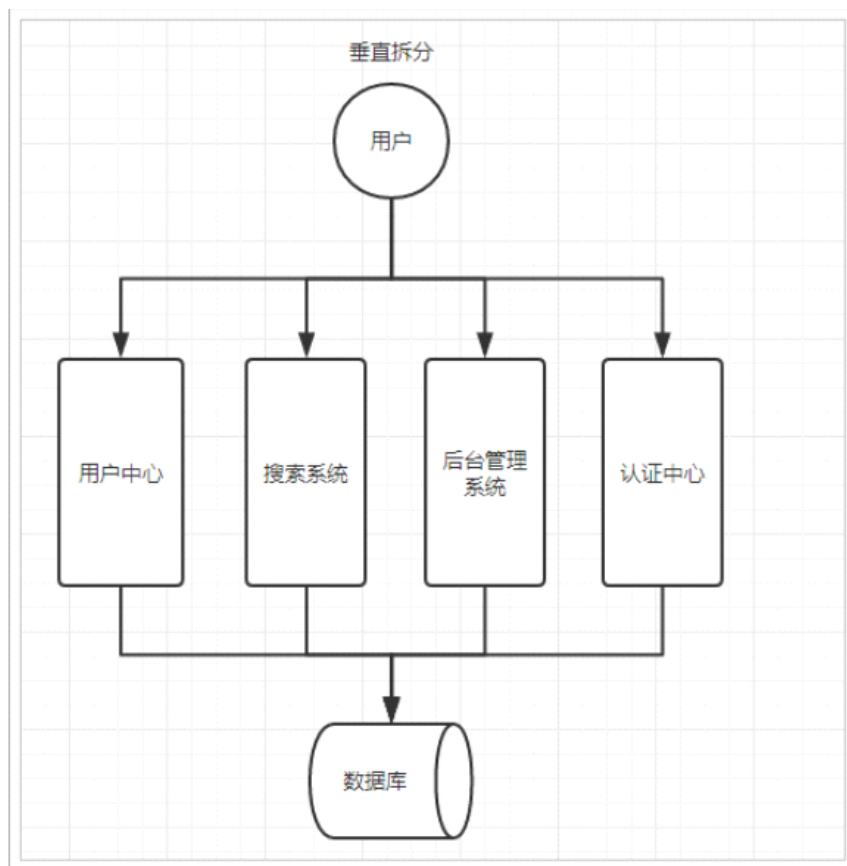
- 系统开发速度快
- 维护成本低
- 适用于并发要求较低的系统

缺点：

- 代码耦合度高，后期维护困难
- 无法针对不同模块进行优化
- 无法水平扩展
- 单点容错率低，并发能力差

1.2 垂直拆分

当访问量逐渐增大，单一应用无法满足需求，此时为了应对更高的并发和业务需求，我们根据业务功能对系统进行拆分：



优点:

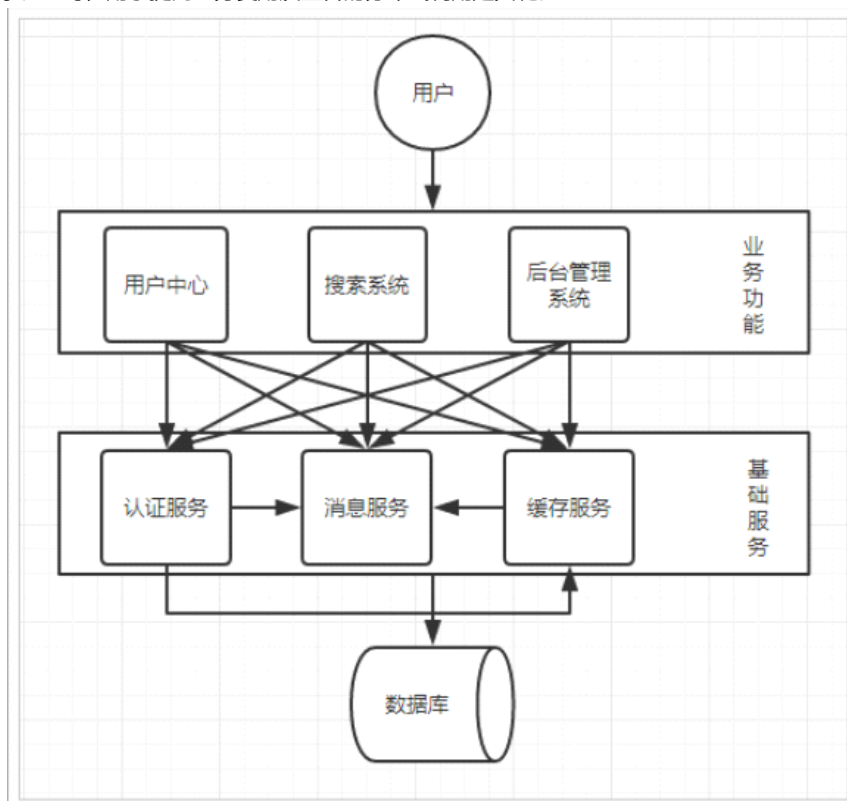
- 系统拆分实现了流量分担，解决了并发问题
- 可以针对不同模块进行优化
- 方便水平扩展，负载均衡，容错率提高

缺点:

- 系统之间相互独立，会有很多重复开发工作，影响开发效率

1.3 分布式服务

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。此时，用于提高业务复用及整合的分布式调用是关键。



优点：

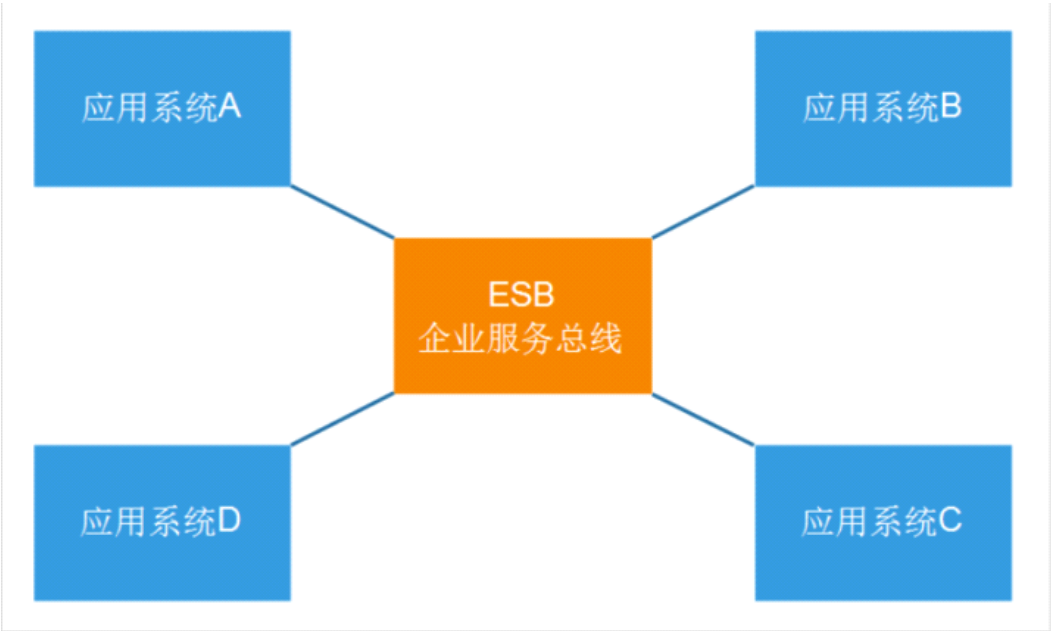
- 将基础服务进行了抽取，系统间相互调用，提高了代码复用和开发效率

缺点：

- 系统间耦合度变高，调用关系错综复杂，难以维护

1.4 服务治理(SOA)

SOA (Service Oriented Architecture) 面向服务的架构：它是一种设计方法，其中包含多个服务，服务之间通过相互依赖最终提供一系列的功能。一个服务通常以独立的形式存在于操作系统进程中。各个服务之间通过网络调用。



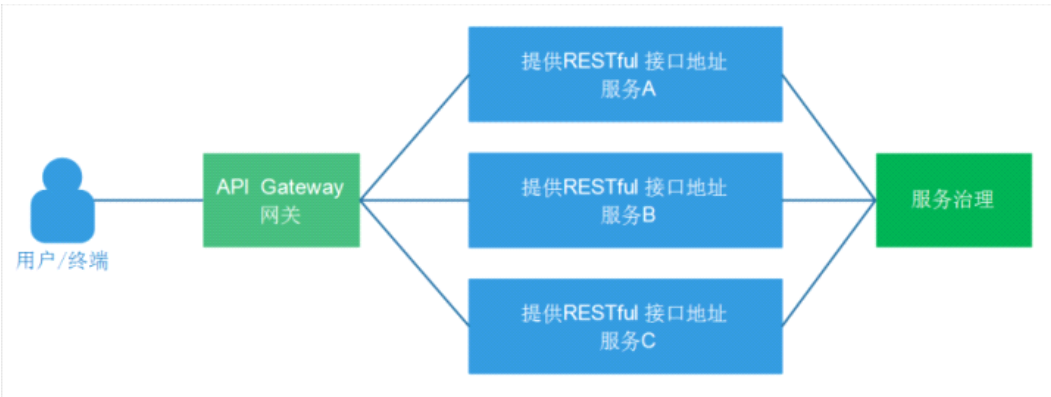
ESB(企业服务总线)，简单来说ESB就是一根管道，用来连接各个服务节点。为了集成不同系统，不同协议的服务，ESB做了消息的转解释和路由工作，让不同的服务互联互通。

SOA缺点：每个供应商提供的ESB产品有偏差，自身实现较为复杂；应用服务力度较大，ESB集成整合所有服务和协议、数据转换使得运维、测试部署困难。所有服务都通过一个通路通信，直接降低了通信速度。

1.5 微服务

微服务架构是使用一套小服务来开发单个应用的方法或途径，每个服务基于单一业务能力构建，运行在自己的进程中，并使用轻量级机制通信，通常是 HTTP API，并能通过自动化部署机制来独立部署。这些服务可以使用不同的编程语言实现，以及不同数据存储技术，并保持最低限度的集中式管理。

微服务结构图：



API Gateway网关是一个服务器，是系统的唯一入口。网关提供RESTful/HTTP的方式访问服务。而服务端通过服务注册中心进行服务注册和管理。

微服务的特点：

- 单一职责：微服务中每一个服务都对应唯一的业务能力，做到单一职责
- 面向服务：面向服务是说每个服务都要对外暴露服务接口API。并不关心服务的技术实现，做到与平台和语言无关，也不限定用什么技术实现，只要提供REST的接口即可。
- 自治：自治是说服务间相互独立，互不干扰
 - 团队独立：每个服务都是一个独立的开发团队。
 - 技术独立：因为是面向服务，提供REST接口，使用什么技术没有别人干涉
 - 前后端分离：采用前后端分离开发，提供统一REST接口，后端不用再为PC、移动端开发不同接口
 - 数据库分离：每个服务都使用自己的数据源

微服务和SOA比较：

功能	SOA	微服务
组件大小	大块业务逻辑	单独任务或小块业务逻辑
耦合	通常松耦合	总是松耦合
管理	着重中央管理	着重分散管理
目标	确保应用能够交互操作	易维护、易扩展、更轻量级的交互

2、远程调用方式

无论是微服务还是SOA，都面临着服务间的远程调用。那么服务间的远程调用方式有哪些呢？

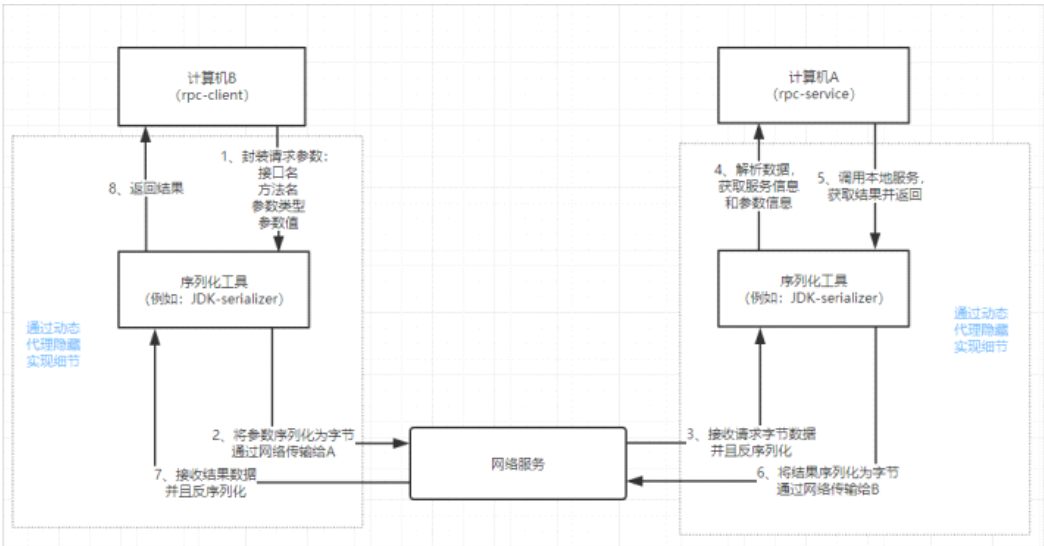
常见的远程调用方式有以下几种：

- RPC：Remote Procedure Call远程过程调用，类似的还有RMI。自定义数据格式，基于原生TCP通信，速度快、效率高。早期的Web Service，现在热门的Dubbo，都是RPC的典型。
 - HTTP：HTTP其实是一种网络传输协议，基于TCP，规定了数据传输的格式。现在客户端浏览器与服务端通信基本都是采用HTTP协议。也可以用来进行远程服务调用。缺点就是消息封装臃肿。
- 现在热门的REST风格，就可以通过HTTP协议来实现。

2.1 认识RPC

RPC，即Remote Procedure Call(远程过程调用)，是一个计算机通信协议。该协议允许运行于一台计算机的程序调用另一台计算机的子程序，而程序员无需额外的为这个交互作用编程。说的通俗一点就是：A计算机提供一个服务，B计算机可以像调用本地服务那样调用A计算机的服务。

RPC调用流程图：



2.2 认识HTTP

HTTP其实是一种网络传输协议，基于TCP，工作在应用层，规定了数据传输的格式。现在客户端浏览器与服务端通信基本都是采用HTTP协议，也可以用来进行远程服务调用。缺点是消息封装臃肿，优势是对服务的提供和调用方没有任何技术限定，自由灵活，更符合微服务理念。现在热门的REST风格，就可以通过HTTP协议来实现。



2.3 如何选择

RPC的机制是根据语言的API(language API)来定义的，而不是根据基于网络的应用来定义的。如果公司全部采用java技术栈，那么使用Dubbo作为微服务架构是一个不错的选择。

相反，如果公司的技术栈多样化，而且你更青睐Spring家族，那么Spring Cloud搭建微服务是不二之选。会选择Spring Cloud套件，因此会使用HTTP方式来实现服务间调用。

3、Spring Cloud简介

3.1 简介

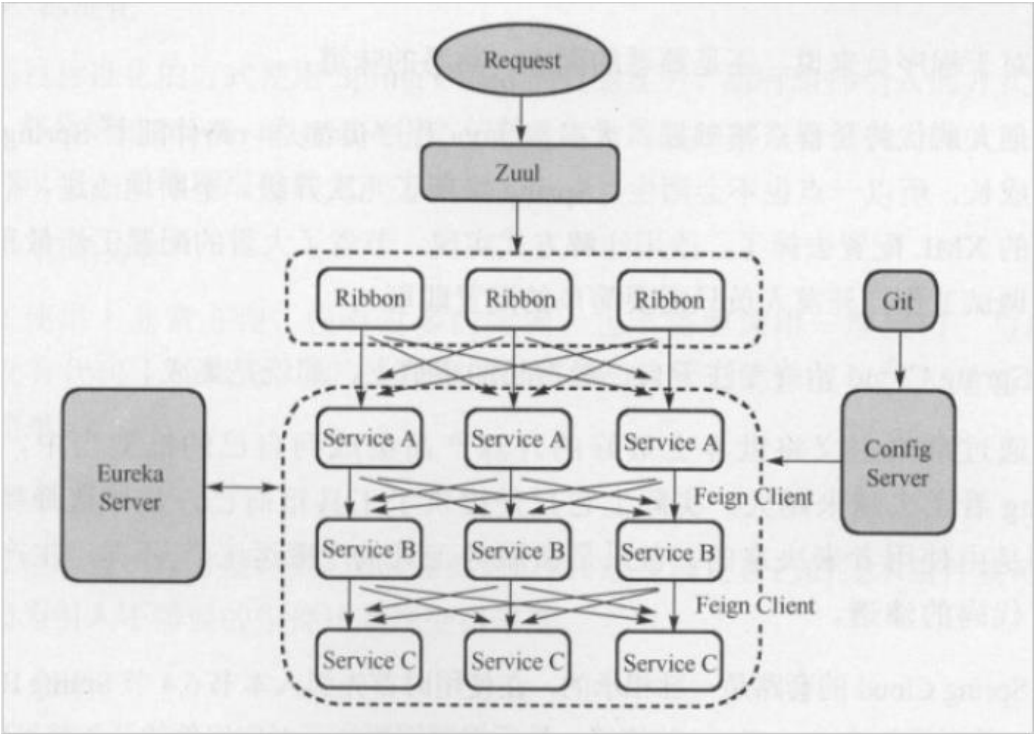
Spring Cloud是Spring旗下的项目之一，官网地址：<https://spring.io/projects/spring-cloud>

Spring最擅长的就是集成，把世界上最好的框架拿过来，集成到自己的项目中。

Spring Cloud也是一样，它将现在非常流行的一些技术整合到一起，实现了诸如：配置管理、服务发现、智能路由、负载均衡、熔断器、控制总线、集群状态等等功能。其主要涉及的组件包括：

- Eureka：注册中心
- Zuul：服务网关
- Ribbon：负载均衡
- Feign：服务调用
- Hystrix：熔断器

以上只是其中的一部分，Spring Cloud架构图：



3.2 版本

Spring Cloud的版本命名比较特殊，因为它不是一个组件，而是许多组件的集合，它的命名是以A到Z为首字母的一些单词组成(其实是伦敦地铁站的名字)：

2020.0.2	CURRENT	GA	Reference Doc.
2020.0.3-SNAPSHOT	SNAPSHOT		Reference Doc.
Hoxton.SR10	GA		Reference Doc.
Hoxton.BUILD-SNAPSHOT	SNAPSHOT		Reference Doc.

Spring Cloud和Spring Boot版本对应关系

Release Train	Boot Version
Hoxton	2.2.x
Greenwich	2.1.x
Finchley	2.0.x
Edgware	1.5.x
Dalston	1.5.x

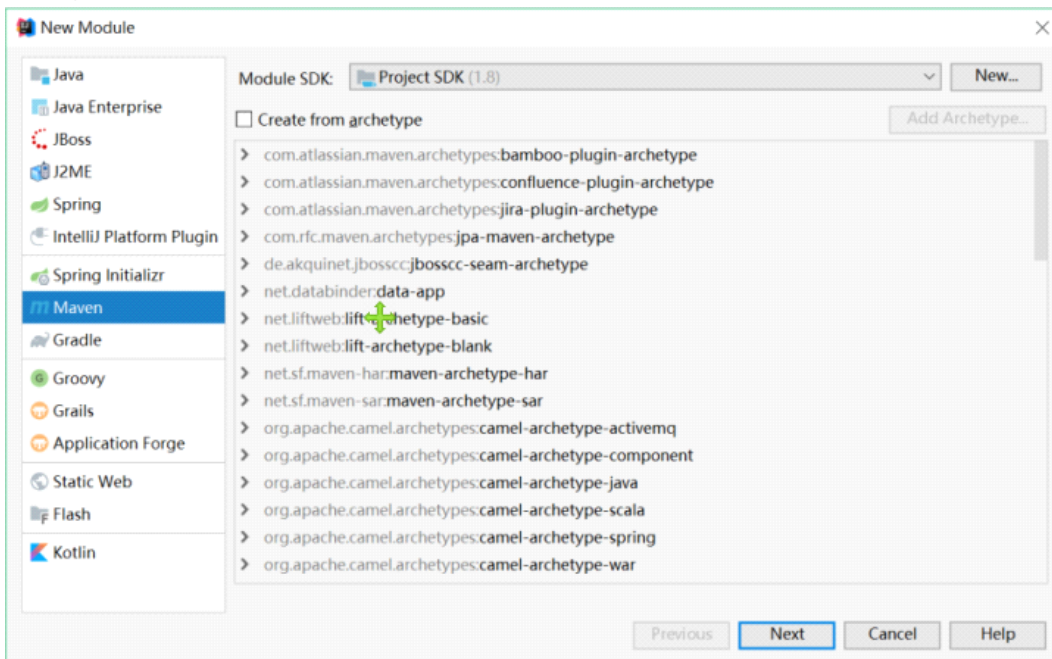
课程采用Greenwich版本

4、微服务场景模拟

模拟一个服务调用的场景。方便学习后面的课程

4.1 创建父工程

微服务中需要同时创建多个项目，为了方便课堂演示，先创建一个父工程，后续的工程都以这个工程为父，使用maven的聚合和继承，统一管理子工程的版本和配置



pom.xml文件:

```
<?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.colin</groupId>
    <artifactId>colin-springcloud</artifactId>
    <packaging>pom</packaging>
    <version>1.0-SNAPSHOT</version>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.5.RELEASE</version>
        <relativePath/>
    </parent>
    <properties>
        <java.version>1.8</java.version>
        <spring-cloud.version>Greenwich.SR1</spring-cloud.version>
        <mapper.starter.version>2.1.5</mapper.starter.version>
        <mysql.version>5.1.46</mysql.version>
    </properties>
    <dependencyManagement>
        <dependencies>
            <!-- springCloud -->
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```



```

        <type>pom</type>
        <scope>import</scope>
    </dependency>
    <!-- 通用Mapper启动器 -->
    <dependency>
        <groupId>tk.mybatis</groupId>
        <artifactId>mapper-spring-boot-starter</artifactId>
        <version>${mapper.starter.version}</version>
    </dependency>
    <!-- mysql驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>${mysql.version}</version>
    </dependency>
</dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
    </dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>

```

注意：Spring Cloud和Spring Boot的版本对应greenwich版本Cloud对应Spring Boot 2.1.x

注意：注意聚合父工程<packaging>pom</packaging>

这里已经对大部分要用到的依赖的版本进行管理，方便后续使用

4.2 服务提供者

新建一个项目，对外提供查询用户的服务。

4.2.1 创建Module

选中colin-springcloud，创建子工程：

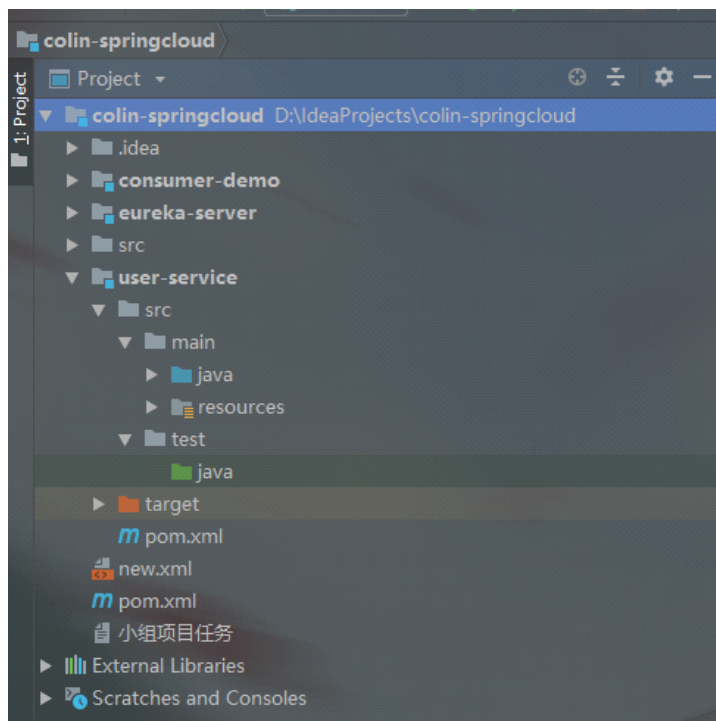
pom.xml文件

```

<?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">
    <parent>
        <artifactId>colin-springcloud</artifactId>
        <groupId>com.colin</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>user-service</artifactId>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!-- 通用Mapper启动器 -->
        <dependency>
            <groupId>tk.mybatis</groupId>
            <artifactId>mapper-spring-boot-starter</artifactId>
        </dependency>
        <!-- mysql驱动 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
    </dependencies>
</project>

```

项目结构



4.2.2 编写配置文件

创建user-service\src\main\resources\application.yml配置文件，这里我们采用yaml语法，而不是properties:

```
server:
  port: 9091
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/jdbc202003
    username: root
    password: abc123
mybatis:
  type-aliases-package: com.colin.user.pojo
```

4.2.3 编写代码

启动类

```
package com.colin.user;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import tk.mybatis.spring.annotation.MapperScan;
```

```
/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-01-24
 * Time: 16:27
 */
@SpringBootApplication
@MapperScan("com.colin.user.mapper")
public class UserApplication {
    public static void main(String[] args) {
        SpringApplication.run(UserApplication.class, args);
    }
}
```

实体类

```
package com.colin.user.pojo;
```

```
import lombok.Data;
```

```
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
```

```
/**
```



```

* Created with IntelliJ IDEA.
* Description:
* User: DengLong
* Date: 2021-03-25
* Time: 10:57
*/
@Data
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String name;
    private String password;
}

UserMapper
package com.colin.user.mapper;

import com.colin.user.pojo.User;
import tk.mybatis.mapper.common.Mapper;

/**
* Created with IntelliJ IDEA.
* Description:
* User: DengLong
* Date: 2021-03-25
* Time: 11:01
*/
public interface UserMapper extends Mapper<User> {
}

Controller
package com.colin.user.controller;

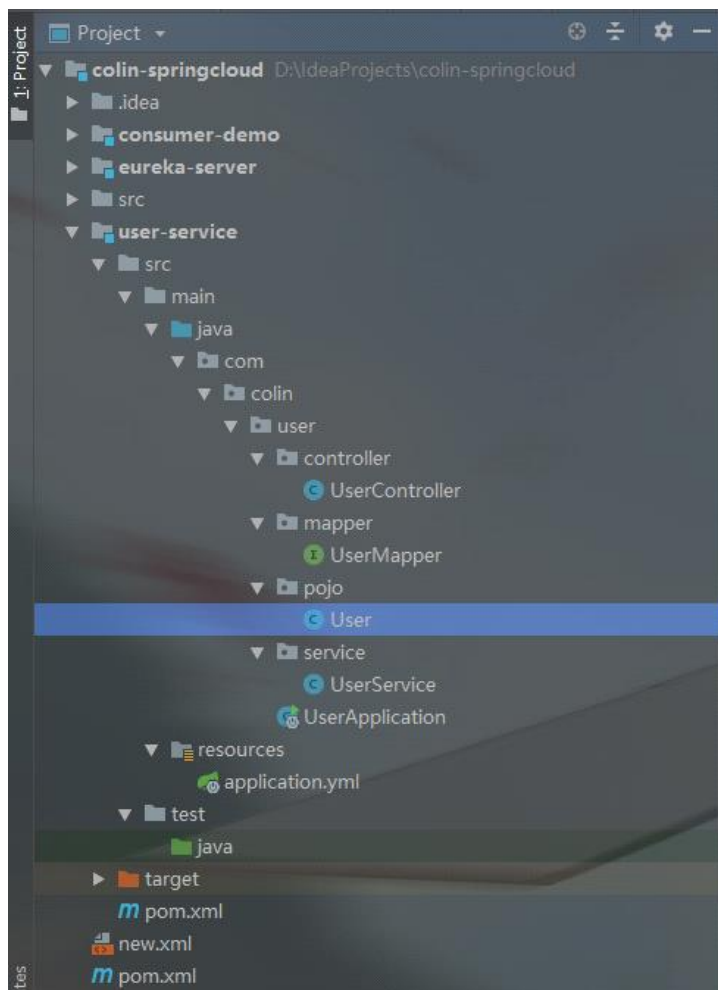
import com.colin.user.pojo.User;
import com.colin.user.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

/**
* Created with IntelliJ IDEA.
* Description:
* User: DengLong
* Date: 2021-03-25
* Time: 11:03
*/
@RestController
@RequestMapping("/user")
public class UserController {
    @Autowired
    private UserService userService;

    @GetMapping("/{id}")
    public User queryById(@PathVariable Integer id) {
        return userService.queryById(id);
    }
}

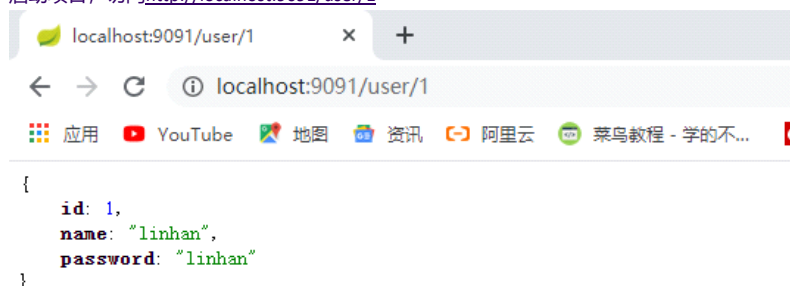
```

完成上述代码后的项目结构



4.2.4 启动并测试

启动项目，访问<http://localhost:9091/user/1>



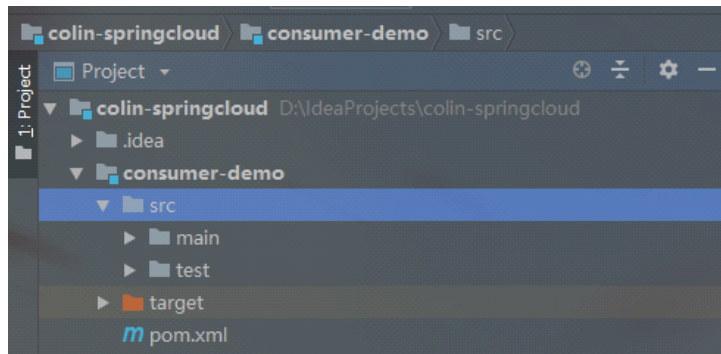
4.3 服务调用者

4.3.1 创建工程

与上面类似，这里不再赘述，需要注意的是，我们调用user-service的功能，因此不需要Mybatis相关依赖了。拷贝之前的user-service模块，更改相应的坐标

```
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">
    <parent>
        <artifactId>colin-springcloud</artifactId>
        <groupId>com.colin</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>consumer-demo</artifactId>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>
</project>
```

项目结构如下:



4.3.2 编写代码

启动器:

```
package com.colin.consumer;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-03-25
 * Time: 13:19
 */
@SpringBootApplication
public class ConsumerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class, args);
    }

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

Spring提供了一个RestTemplate模板工具类, 对基于HTTP的客户端进行了封装, 并且实现了对象与json的序列化和反序列化, 非常方便。RestTemplate并没有限定HTTP的客户端类型, 而是进行了抽象, 目前常用的3种都有支持:

- HttpClient
- OkHTTP
- JDK远程的URLConnection (默认的)

实体类

```
package com.colin.consumer.pojo;
```

```
import lombok.Data;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-03-25
 * Time: 13:20
 */
@Data
public class User {
    private Integer id;
    private String name;
    private String password;
}
```

Controller

```
package com.colin.consumer.controller;

import com.colin.consumer.pojo.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
```

```

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

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-03-25
 * Time: 13:21
 */
@RestController
@RequestMapping("/consumer")
public class ConsumerController {
    @Autowired
    private RestTemplate restTemplate;

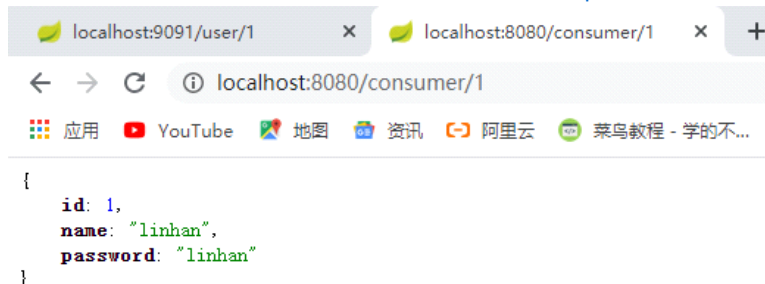
    @GetMapping("/{id}")
    public User queryById(@PathVariable Integer id) {
        String url = "http://localhost:9091/user/" + id;

        return restTemplate.getForObject(url, User.class);
    }
}

```

4.3.3 启动测试：

因为我们没有配置端口，那么默认就是8080，我们访问：<http://localhost:8080/consumer/1>



一个简单的远程服务调用案例就实现了。

4.4 思考问题

简单回顾一下，刚才我们写了什么：

- user-service：对外提供了查询用户的接口
- consumer-demo：通过RestTemplate访问<http://localhost:9091/user/{id}>接口，查询用户数据

存在什么问题？

- 在consumer中，我们把user地址硬编码到了代码中，不方便后期维护
- consumer需要记忆user-service的地址，如果出现变更，可能得不到通知，地址将失效
- consumer不清楚user-service的状态，服务宕机也不知道
- user-service只有1台服务，不具备高可用性
- 即便user-service形成集群，consumer还需自己实现负载均衡

其实上面说的问题，概括一下就是分布式服务必然要面临的问题：

- 服务管理
 - 如何自动注册和发现
 - 如何实现状态监管
 - 如何实现动态路由
- 服务如何实现负载均衡
- 服务如何解决容灾问题
- 服务如何实现统一配置

以上的问题，我们都将在Spring Cloud中得到答案。

5、Eureka注册中心

5.1 Eureka简介

问题分析

在刚才的案例中，user-service对外提供服务，需要对外暴露自己的地址。而consumer需要记录服务提供者的地址。将来地址出现变更，还需要及时更新。这在服务较少的时候并不觉得有什么，但是在现在日益复杂的互联网环境，一个项目肯定会拆分成十几，甚至数十个微服务。此时如果还人为管理地址，不仅开发困难，将来测试、发布上线都会非常麻烦，这与DevOps的思想背道而驰。

网约车

这就好比是网约车出现之前，人们出门叫车只能叫出租车。一些私家车想做出租却没有资格，被称为黑车。而很多人想要约车，但是无奈出租车太少，不方便。私家车很多却不敢拦，而且满大街的车，谁知道哪个才是愿意载人的。一个想要，一个愿意给，就是缺少引子，缺乏管理。

此时滴滴这样的网约车平台出现了，所有想载客的私家车全部到滴滴注册，记录你的车型(服务类型)，身份信息(联系方式)。这样提供服务的私家车，在滴滴那里都能找到，一目了然。

此时要叫车的人，只需要打开APP，输入你的目的地，选择车型(服务类型)，滴滴自动安排一个符合需求的车到你面前，为你服务。

Eureka做什么？

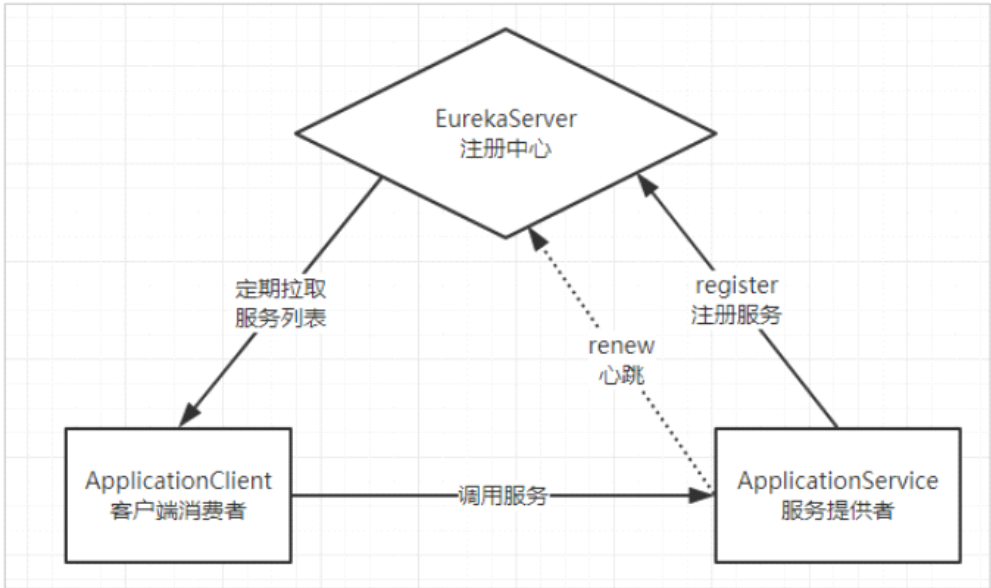
Eureka就好比是滴滴，负责管理、记录服务提供者的信息。服务调用者无需自己寻找服务，而是把自己的需求告诉Eureka，然后Eureka会把符合你需要的服务告诉你。

同时，服务提供方与Eureka之间通过“心跳”机制进行监控，当某个服务提供方出现问题，Eureka自然会把它从服务列表中剔除。

这就实现了服务的自动注册、发现、状态监控。

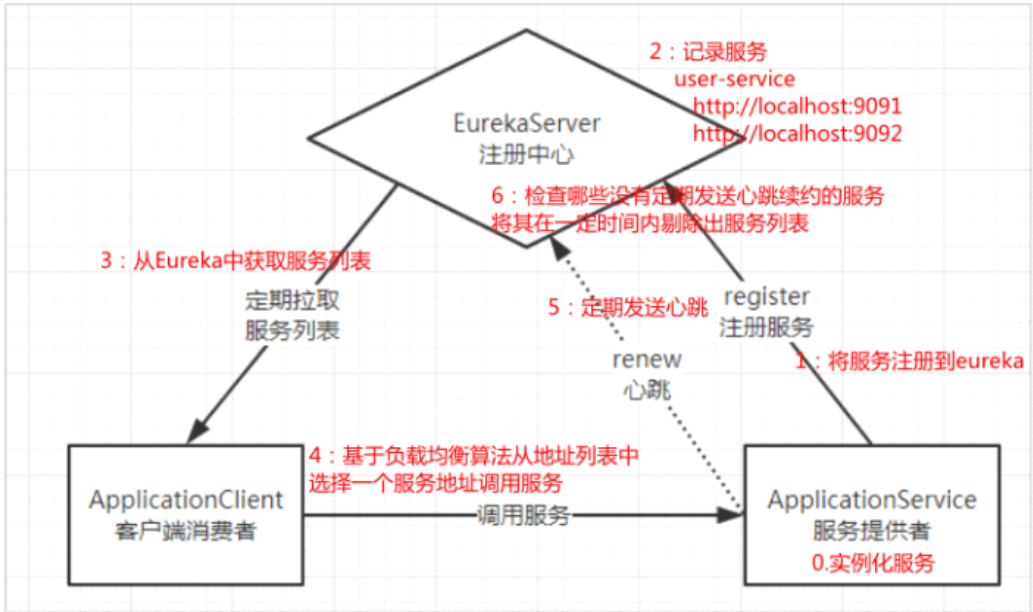
5.2 原理图

基本架构：



- Eureka：就是服务注册中心(可以是一个集群)，对外暴露自己的地址
- 提供者：启动后向Eureka注册自己信息(地址，提供什么服务)
- 消费者：向Eureka订阅服务，Eureka会将对应服务的所有提供者地址列表发送给消费者，并且定期更新
- 心跳(续约)：提供者定期通过HTTP方式向Eureka刷新自己的状态

工作原理解析



5.3入门案例

5.3.1 编写EurekaServer

Eureka是服务注册中心，只做服务注册；自身并不提供服务也不消费服务。可以搭建Web工程使用Eureka，可以使用Spring Boot方式搭建。

1、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>
    <parent>
        <artifactId>colin-springcloud</artifactId>
        <groupId>com.colin</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>eureka-server</artifactId>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
        </dependency>
    </dependencies>
</project>
```

2、编写启动类

```
package com.colin;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-01-24
 * Time: 17:24
 */
@EnableEurekaServer
@SpringBootApplication
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

3、编写配置

```
server:
  port: 10086
spring:
  application:
    name: eureka-server
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:10086/eureka
    #不注册自己
    register-with-eureka: false
    #不拉取服务
    fetch-registry: false
```

4、启动服务，并访问：<http://127.0.0.1:10086/>

The screenshot shows the Spring Eureka web interface in a browser. The address bar indicates the URL is 127.0.0.1:10086. The interface has a dark header with the Spring Eureka logo and navigation links for HOME and LAST 1000 SINCE STARTUP.

System Status

Environment	test	Current time	2021-03-31T19:41:26 +0800
Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

DS Replicas

127.0.0.1

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

Name	Value
total-avail-memory	319mb
environment	test
num-of-cpus	4
current-memory-usage	203mb (63%)
server-uptime	00:00
registered-replicas	http://127.0.0.1:10086/eureka/
unavailable-replicas	http://127.0.0.1:10086/eureka/
available-replicas	

Instance Info

Name	Value
ipAddr	192.168.160.1
status	UP

5.3.2 服务注册

在服务提供工程user-service上添加Eureka客户端依赖；自动将服务注册到EurekaServer服务地址列表。

1、添加依赖

```
<!--Eureka客户端-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

2、在启动类上开启Eureka客户端功能

通过添加@EnableDiscoveryClient来开启Eureka客户端功能

```
package com.colin.user;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import tk.mybatis.spring.annotation.MapperScan;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-03-25
 * Time: 10:55
 */
@SpringBootApplication
@MapperScan("com.colin.user.mapper")
@EnableDiscoveryClient // 开启Eureka客户端发现功能
public class UserApplication {
    public static void main(String[] args) {
        SpringApplication.run(UserApplication.class, args);
    }
}

```

3、编写配置

```

server:
  port: 9091
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/jdbc202003
    username: root
    password: abc123
  application:
    name: user-service
mybatis:
  type-aliases-package: com.colin.user.pojo
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:10086/eureka

```

注意:

- 这里我们添加了spring.application.name属性来指定应用名称，将来会作为应用的id使用。
- 不用指定register-with-eureka和fetch-registry，因为默认是true

重启项目，访问Eureka监控页面查看

The screenshot shows the Spring Eureka web interface in a browser. The address bar indicates the URL is 127.0.0.1:10086. The interface has a dark header with the 'spring Eureka' logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'.

System Status

Environment	test	Current time	2021-03-31T19:50:42 +0800
Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	5
		Renews (last min)	0

DS Replicas

127.0.0.1

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
USER-SERVICE	n/a (1)	(1)	UP (1) - localhost:user-service:9091

General Info

Name	Value
total-avail-memory	302mb
environment	test
num-of-cpus	4
current-memory-usage	205mb (67%)
server-uptime	00:00
registered-replicas	http://127.0.0.1:10086/eureka/
unavailable-replicas	http://127.0.0.1:10086/eureka/
available-replicas	

Instance Info

Name	Value
ipAddr	192.168.160.1
status	UP

我们发现user-service服务已经注册成功了。

5.3.3 服务发现

在服务消费工程consumer-demo上添加Eureka客户端依赖；可以使用工具类DiscoveryClient根据服务名称获取对应的服务地址列表。

1、添加依赖

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

2、在启动类添加开启Eureka客户端发现的注解

```
package com.colin.consumer;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;
```

```
/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-03-25
 * Time: 13:19
 */
@SpringBootApplication
@EnableDiscoveryClient // 开启Eureka客户端
public class ConsumerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class, args);
    }

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

3. 修改配置

```
spring:
  application:
    name: consumer-demo
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:10086/eureka
```

4. 修改代码，用DiscoveryClient类的方法，根据服务名称，获取服务实例

```
package com.colin.consumer.controller;

import com.colin.consumer.pojo.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import java.util.List;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-03-25
 * Time: 13:21
 */
@RestController
@RequestMapping("/consumer")
public class ConsumerController {
    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private DiscoveryClient discoveryClient;

    @GetMapping("/{id}")
    public User queryById(@PathVariable Integer id) {
        String url = "http://localhost:9091/user/" + id;

        List<ServiceInstance> serviceInstances = discoveryClient.getInstances("user-service");
        ServiceInstance serviceInstance = serviceInstances.get(0);

        url = "http://" + serviceInstance.getHost() + ":" + serviceInstance.getPort() + "/user/" + id;
        return restTemplate.getForObject(url, User.class);
    }
}
```

5.4 Eureka详解

接下来我们详细讲解Eureka的原理及配置。

5.4.1 基础架构

Eureka架构中的三个核心角色：

- 服务注册中心
Eureka的服务端应用，提供服务注册和发现功能，就是上面我们建立的eureka-server
- 服务提供者
提供服务的应用，可以是Spring Boot应用，也可以是其它任意技术实现，只要对外提供的是REST风格服务即可。本例中就是我们实现的user-service
- 服务消费者
消费应用从注册中心获取服务列表，从而得知每个服务方的信息，知道去哪里调用服务方。本例中就是我们实现的consumer-demo

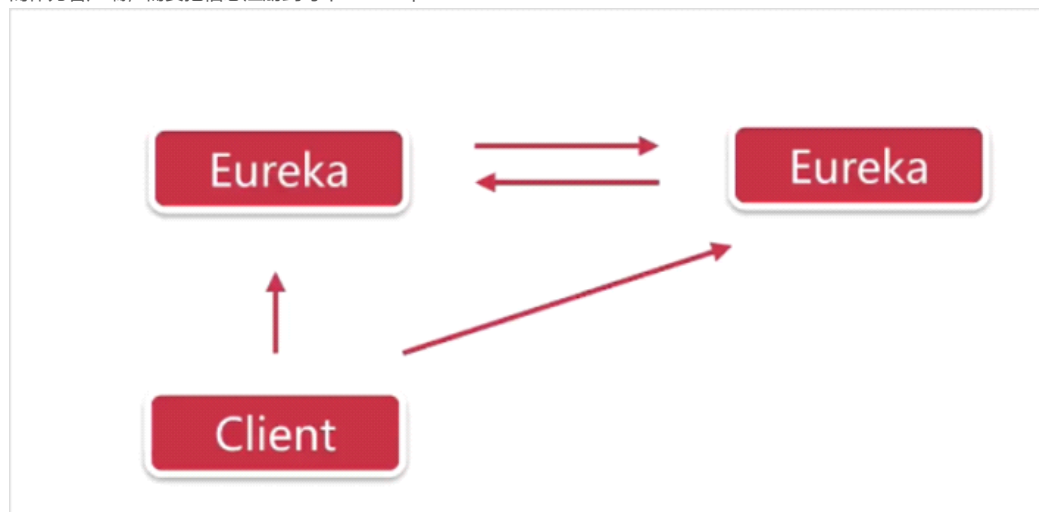
5.4.2 高可用的Eureka Server

Eureka Server即服务的注册中心，在上面的案例中，我们只有一个EurekaServer，实际上EurekaServer也可以是一个集群，形成高可用的Eureka中心。
Eureka Server是一个web应用，可以启动多个实例(配置不同端口)保证Eureka Server的高可用。

服务同步

多个Eureka Server之间也会互相注册为服务，当服务者注册到Eureka Server集群中的某个节点时，该节点会把服务的信息同步给集群中的每个节点，从而实现数据同步。因此，无论客户端访问到Eureka Server集群中的任意一个节点，都可以获取到完整的服务列表信息。

而作为客户端，需要把信息注册到每个Eureka中



如果有三个Eureka，则每个EurekaServer都需要注册到其它几个Eureka服务中。

例如：有三个分别为10086/10087/10088，则：

10086要注册到10087和10088上

10087要注册到10086和10088上

10088要注册到10086和10087上

动手搭建高可用的EurekaServer

我们假设要搭建两条EurekaServer的集群，端口分别为：10086和10087

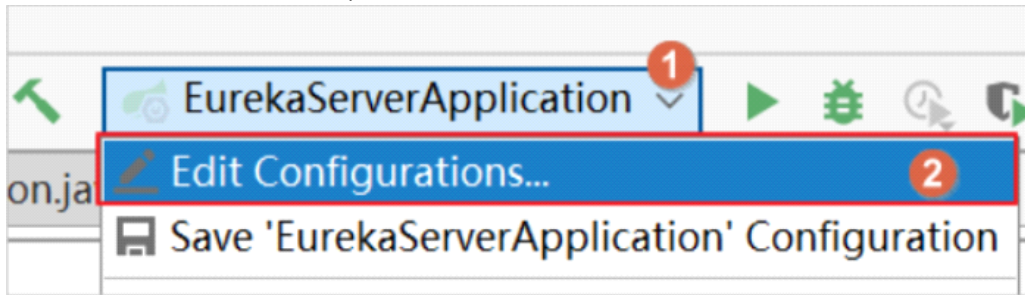
1、我们修改原来的EurekaServer配置：

```
server:
  port: ${port:10086}
spring:
  application:
    name: eureka-server
eureka:
  client:
    #eureka的服务地址，如果是集群，需要指定其他集群eureka地址
    service-url:
      defaultZone: ${defaultZone:http://127.0.0.1:10086/eureka}
    #不注册自己
    #register-with-eureka: false
    #不拉取服务
    #fetch-registry: false
```

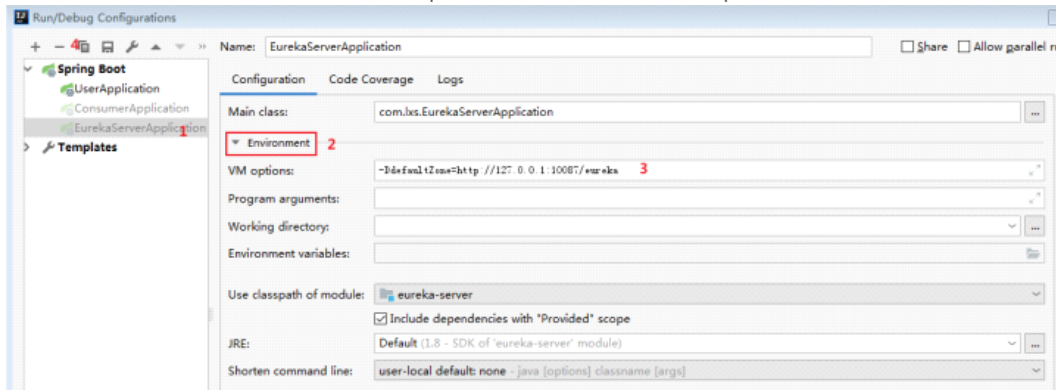
所谓的高可用注册中心，其实就是把EurekaServer自己也作为一个服务进行注册，这样多个EurekaServer之间就能互相发现对方，从而形成集群。因为我们做了以下修改：

- 删除了register-with-eureka=false和fetch-registry=false两个配置。因为默认值是true，这样就会把自己注册到注册中心了。
- 把service-url的值改成了另外一台EurekaServer的地址，而不是自己

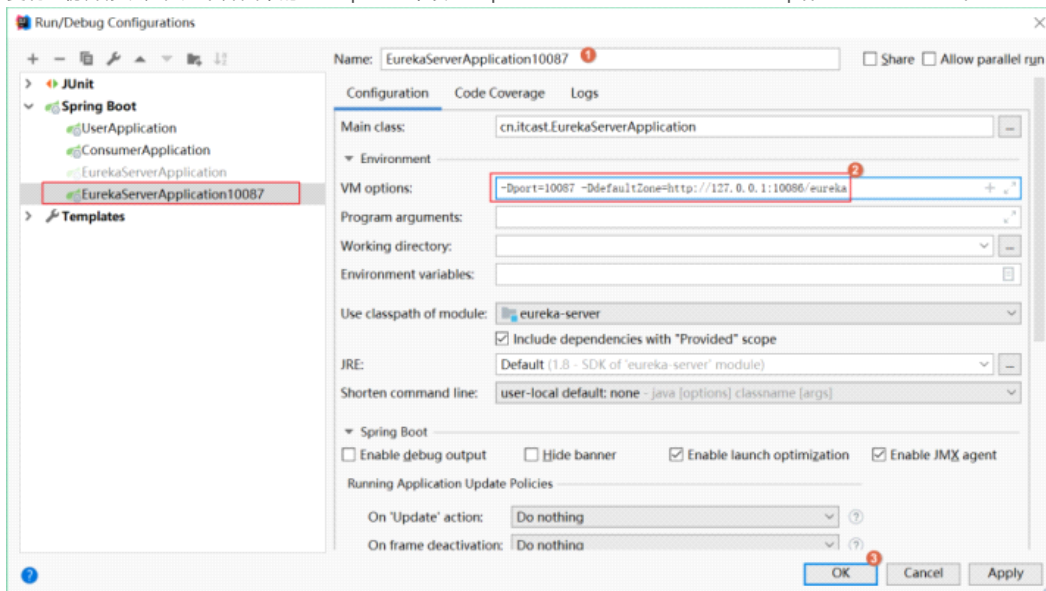
2、另外一台在启动的时候可以指定端口和defaultZone配置：



修改原来的启动配置组件；在如下界面中的VM options中设置-DdefaultZone=http://127.0.0.1:10087/eureka



复制一份并修改；在如下界面中的VM options中设置-Dport=10087 -DdefaultZone=http://127.0.0.1:10086/eureka



3、启动测试；同时启动两台eureka server



4. 客户端注册服务到集群

因为EurekaServer不止一个，因此注册服务的时候，service-url参数需要变化：

```
eureka:
  client:
    service-url: # EurekaServer地址,多个地址以','隔开
    defaultZone: http://127.0.0.1:10086/eureka,http://127.0.0.1:10087/eureka
```

为了方便上课和后面内容的修改，在测试完上述配置后可以再次改回单个eureka server的方式。

5.4.3 Eureka客户端和服务端配置

这个小节我们进行一系列的配置：

- Eureka客户端工程
 - user-service 服务提供
 - 服务地址使用ip方式
 - 续约
 - consumer-demo 服务消费
 - 获取服务地址的频率
- Eureka服务端工程eureka-server
 - 失效剔除
 - 自我保护

服务提供者要向EurekaServer注册服务，并且完成服务续约等工作。

服务注册

服务提供者在启动时，会检测配置属性中的eureka.client.register-with-eureka=true参数是否为true，事实上默认就是true。如果值确实为true，则会向EurekaServer发起一个REST请求，并携带自己的元数据信息，EurekaServer会把这些信息保存到一个双层Map结构中。

- 第一层Map的key就是服务id，一般是配置中的spring.application.name属性，user-service
- 第二层Map的key是服务的实例id，一般host+serviceld+port，例如：localhost:user-service:8081
- 值则是服务的实例对象，也就是说一个服务，这样可以同时启动多个不同实例，形成集群。

默认注册时使用的是主机名或者localhost，如果想用ip进行注册，可以在user-service中添加配置如下：

```
eureka:
  instance:
    ip-address: 127.0.0.1 # ip地址
    prefer-ip-address: true # 更倾向于使用ip，而不host名
```

修改完后先后重启user-service和consumer-demo；在调用服务的时候已经变成ip地址；需要注意的是：不是在eureka中的控制台服务实例状态显示。

服务续约

在注册服务完成以后，服务提供者会维持一个心跳(定时向EurekaServer发起REST请求)，告诉EurekaServer：“我还活着”。这个我们称为服务的需求(renew)；有两个重要参数可以修改服务续约的行为；可以在user-service中添加如下配置项：

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

- lease-renewal-interval-in-seconds：服务续约(renew)的间隔，默认为30秒
- lease-expiration-duration-in-seconds：服务失效时间，默认值90秒

也就是说，默认情况下每隔30秒服务会向注册中心发送一次心跳，证明自己活着。如果超过90秒没有发送心跳，EurekaServer就会认为该服务宕机，会从服务列表中移除，这两个值在生产环境不要修改，默认即可。

获取服务列表

当服务消费者启动时，会检测eureka.client.fetch-registry=true参数的值，如果为true，则会从EurekaServer服务的列表中拉取只读备份，然后缓存在本地。并且每隔30秒会重新拉取并更新数据。可以在consumer-demo项目中通过下面的参数来修改：

```
eureka:
  client:
    registry-fetch-interval-seconds: 30
```

生产环境中，我们不需要修改这个值。

但是为了开发环境下，能够更快速得到服务的最新状态，我们可以将其设置小一点。

5.4.5 失效剔除和自我保护

如下的配置都是在EurekaServer服务端进行：

服务下线

当服务进行正常关闭操作时，它会触发一个服务下线的REST请求给EurekaServer，告诉服务注册中心：“我要下线了”。服务中心接受到请求之后，将该服务置为下线状态。

失效剔除

有时我们的服务可能由于内存溢出或网络故障等原因使得服务不能正常的工作，而服务注册中心并未收到“服务下线”的请求。相对于服务提供者的“服务续约”操作，服务注册中心在启动时会创建一个定时任务，默认每个一段时间(默认为60秒)将当前清单中超时(默认为90秒)没有续约的服务剔除，这个操作被称为失效剔除。可以通过eureka.server.eviction-interval-timer-in-ms参数对其进行修改，单位是毫秒。

自我保护

我们关停一个服务，就会在Eureka面板看到一条警告：



这是触发了Eureka的自我保护机制。当一个服务未按时进行心跳续约时，Eureka会统计最近15分钟心跳失败的服务实例的比例是否超过了85%，当EurekaServer节点在短时间内丢失过多客户端(可能发生了网络分区故障)。在生产环境下，因为网络延迟等原因，心跳失败实例的比例很有可能超标，但是此时就把服务剔除列表并不妥当，因为服务可能没有宕机。Eureka就会把当前实例的注册信息保护起来，不予剔除。生产环境下这很有效，保证了大多数服务依然可用。

但这给我们的开发带来了麻烦，因此开发阶段我们都会关闭自我保护模式：

```
eureka:
  server:
    enable-self-preservation: false # 关闭自我保护模式（缺省为打开）
    eviction-interval-timer-in-ms: 1000 # 扫描失效服务的间隔时间（缺省为60*1000ms）
```

小结：

- user-service


```
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:10086/eureka
    instance:
      # 更倾向使用ip地址，而不是host名
      prefer-ip-address: true
      # ip地址
      ip-address: 127.0.0.1
      # 续约间隔，默认30秒
      lease-renewal-interval-in-seconds: 5
      # 服务失效时间，默认90秒
      lease-expiration-duration-in-seconds: 5
```
- consumer-demo


```
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:10086/eureka
      # 获取服务地址列表间隔时间，默认30秒
      registry-fetch-interval-seconds: 10
```

- eureka-server
eureka:
server:
服务失效剔除时间间隔，默认60秒
eviction-interval-timer-in-ms: 60000
关闭自我保护模式（默认是打开的）
enable-self-preservation: false

6、负载均衡Ribbon

在刚才的案例中，我们启动了一个user-service，通过DiscoveryClient来获取服务实例信息，然后获取ip和端口来访问。

但是实际环境中，我们往往会开启很多个user-service的集群。此时我们获取的服务列表中就会有多个，到底访问哪一个呢？

一般这种情况下我们就需要编写负载均衡算法，在多个实例列表中进行选择。

不过Eureka中已经帮我们集成了负载均衡组件：Ribbon，简单修改代码即可使用。

什么是Ribbon：

Ribbon 是 Netflix 发布的负载均衡器，它有助于控制 HTTP 和 TCP 客户端的行为。为 Ribbon 配置服务提供者地址列表后，Ribbon 就可基于某种负载均衡算法，自动地帮助服务消费者去请求。Ribbon 默认为我们提供了很多的负载均衡算法，例如轮询、随机等。当然，我们也可为 Ribbon 实现自定义的负载均衡算法。

接下来，我们就使用Ribbon实现负载均衡。

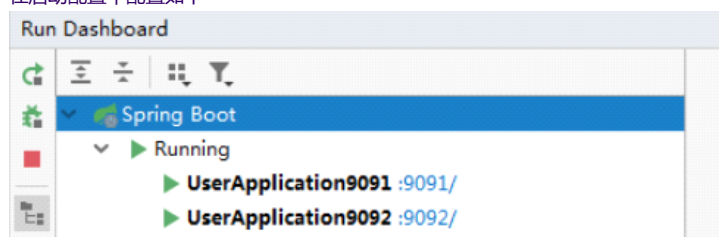
6.1 启动两个服务实例

首先我们启动两个user-service实例，一个端口号9091、另一个端口号9092。

在user-service的application.yml中配置如下端口：

```
server:
  port: ${port:9091}
```

在启动配置中配置如下



Eureka监控面板

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONSUMER-DEMO	n/a (1)	(1)	UP (1) - localhost:consumer-demo
USER-SERVICE	n/a (2)	(2)	UP (2) - localhost:user-service:9092 , localhost:user-service:9091

6.2 开启负载均衡

因为Eureka中已经集成了Ribbon，所以我们无需引入新的依赖。直接修改代码：

在ConsumerApplication的RestTemplate的配置方法上添加@LoadBalanced注解：

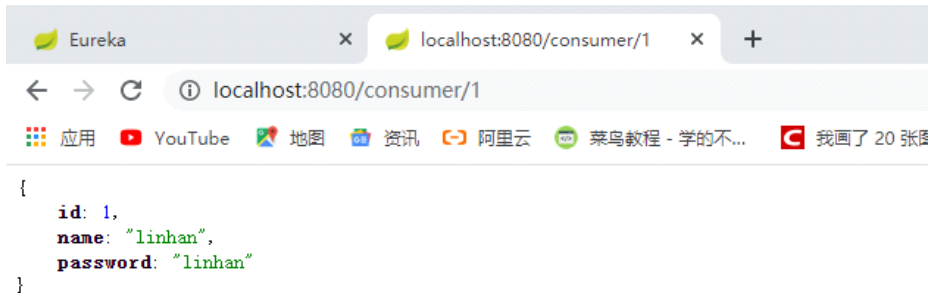
```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

修改调用方式，不再手动获取ip和端口，而是直接通过服务名称调用，在ConsumerController中修改请求地址：

```
@GetMapping("/{id}")
public User queryById(@PathVariable("id") Long id) {
    String url = "http://user-service/user/" + id;
    return restTemplate.getForObject(url, User.class);
}
```

```
}
```

访问页面，查看结果：



访问页面，查看结果；并可以在9091和9092的控制台查看执行情况。

6.3 源码跟踪

为什么只输入service名称就可以访问了呢？之前还要获取ip和端口号。

显然是有组件根据service名称，获取到了服务实例的ip和端口。因为consumer-demo使用的是RestTemplate，spring使用LoadBalancerInterceptor拦截器，这个类会在对RestTemplate的请求进行拦截，然后从Eureka根据服务id获取服务列表，随后利用负载均衡算法得到真实的服务地址信息，替换服务id。

我们进行源码跟踪：

```
public class LoadBalancerInterceptor implements ClientHttpRequestInterceptor {  
  
    private LoadBalancerClient loadBalancer; loadBalancer: RibbonLoadBalancerClient@8374  
    private LoadBalancerRequestFactory requestFactory; requestFactory: LoadBalancerRequestFactory@8373  
  
    public LoadBalancerInterceptor(LoadBalancerClient loadBalancer, LoadBalancerRequestFactory requestFactory) {  
        this.loadBalancer = loadBalancer;  
        this.requestFactory = requestFactory;  
    }  
  
    public LoadBalancerInterceptor(LoadBalancerClient loadBalancer) {  
        // for backwards compatibility  
        this(loadBalancer, new LoadBalancerRequestFactory(loadBalancer));  
    }  
  
    @Override  
    public ClientHttpResponse intercept(final HttpRequest request, final byte[] body, request: InterceptingClientHttpRequestExecution execution) throws IOException { execution: InterceptingClientHttpRequestExecution@8375  
        final URI originalUri = request.getURI(); originalUri: "http://user-service/user/8" request: InterceptingClientHttpRequestExecution@8375  
        String serviceName = originalUri.getHost(); originalUri: "http://user-service/user/8"  
        Assert.state(expression: serviceName != null, message: "Request URI does not contain a valid hostname: " + originalUri);  
        return this.loadBalancer.execute(serviceName, requestFactory.createRequest(request, body, execution));  
    }  
}
```

继续跟入execute方法：发现获取了9092端口的服务

```
public <T> T execute(String serviceId, LoadBalancerRequest<T> request, Object hint) throws IOException {  
    LoadBalancerClient loadBalancer = getLoadBalancer(serviceId); loadBalancer: "DynamicServerListLoadBalancer"  
    Server server = getServer(loadBalancer, hint); loadBalancer: "DynamicServerListLoadBalancer"  
    if (server == null) { 使用负载均衡器从服务列表获取服务  
        throw new IllegalStateException("No instances available for " + serviceId);  
    }  
    RibbonServer ribbonServer = new RibbonServer(serviceId, server, isSecure(server, serviceId), serverIntrospector(serviceId).getMetadata(server));  
  
    return execute(serviceId, ribbonServer, request);  
}
```

再跟下一次，发现获取的是9091：

```
@Override  
public <T> T execute(String serviceId, ServiceInstance serviceInstance, LoadBalancerRequest<T> request) throws IOException {  
    Server server = null; server: "127.0.0.1:9092"  
    if (serviceInstance instanceof RibbonServer) {  
        server = ((RibbonServer) serviceInstance).getServer(); serviceInstance: "RibbonServer{serverId: '127.0.0.1:9092', status: 'UP'}"  
    }  
    if (server == null) { server: "127.0.0.1:9092"  
        throw new IllegalStateException("No instances available for " + serviceId);  
    }  
  
    RibbonLoadBalancerContext context = this.clientFactory.getLoadBalancerContext(serviceId);  
    RibbonStatsRecorder statsRecorder = new RibbonStatsRecorder(context, server);  
  
    try {  
        T returnVal = request.apply(serviceInstance);  
        statsRecorder.recordStats(returnVal);  
        return returnVal;  
    }  
}
```

6.4 负载均衡策略

Ribbon默认的负载均衡策略是简单的轮询，我们可以测试一下：

编写测试类，在刚才的源码中我们看到拦截中是使用RibbonLoadBalancerClient来进行负载均衡的，其中有一个choose方法，是这样介绍的：

```
/**
 * Choose a ServiceInstance from the LoadBalancer for the specified service
 * @param serviceId the service id to look up the LoadBalancer
 * @return a ServiceInstance that matches the serviceId
 */
ServiceInstance choose(String serviceId);
```

现在这个就是负载均衡获取实例的方法。

在consumer-demo的pom.xml中增加依赖

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

我们注入这个类的对象，然后对其进行测试，选中User类，通过快捷键创建单元测试类：

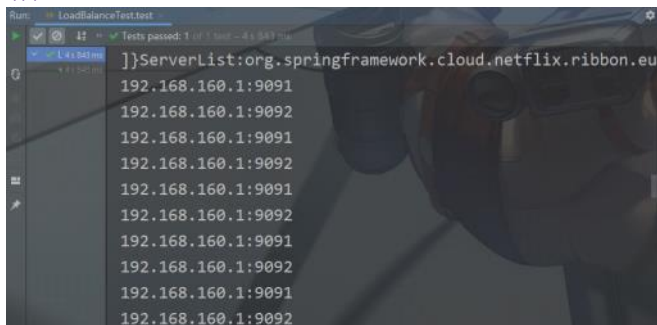
```
package com.colin.consumer.pojo;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.cloud.netflix.ribbon.RibbonLoadBalancerClient;
import org.springframework.test.context.junit4.SpringRunner;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-04-12
 * Time: 9:51
 */
@RunWith(SpringRunner.class)
@SpringBootTest
public class LoadBalanceTest {
    @Autowired
    RibbonLoadBalancerClient client;

    @Test
    public void test() {
        for (int i = 0; i < 100; i++) {
            ServiceInstance instance = this.client.choose("user-service");
            System.out.println(instance.getHost() + ":" + instance.getPort());
        }
    }
}
```

结果：



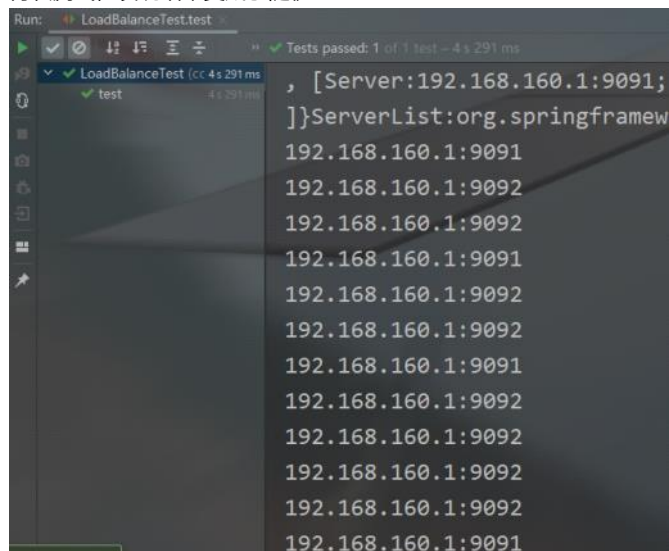
符合了我们的预期，确实是轮询方式。

SpringBoot也帮我们提供了修改负载均衡规则的配置入口，在consumer-demo的application.yml中增加配置：

```
user-service:
  ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
```

格式是: {服务名称}.ribbon.NFLoadBalancerRuleClassName , 值就是IRule的实现类。

再次测试, 发现结果变成了随机:



7、Hystrix

7.1 简介

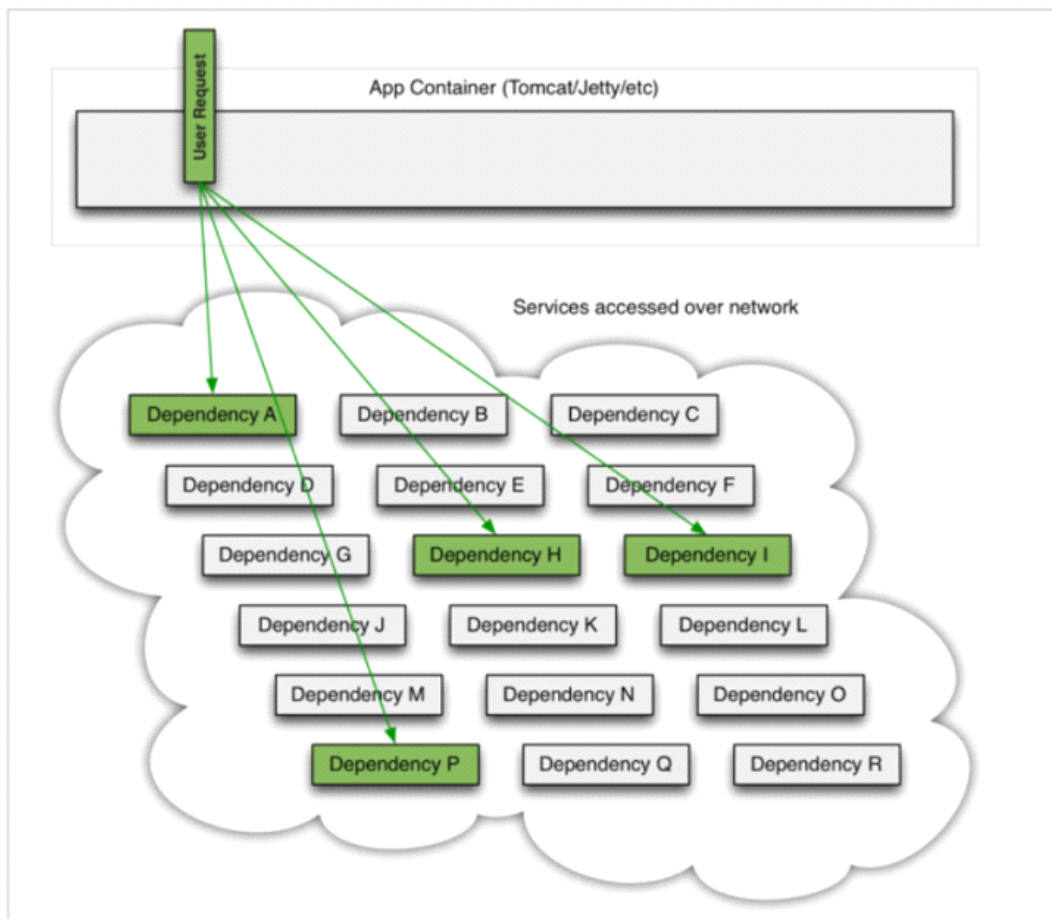
主页: <https://github.com/Netflix/Hystrix/>



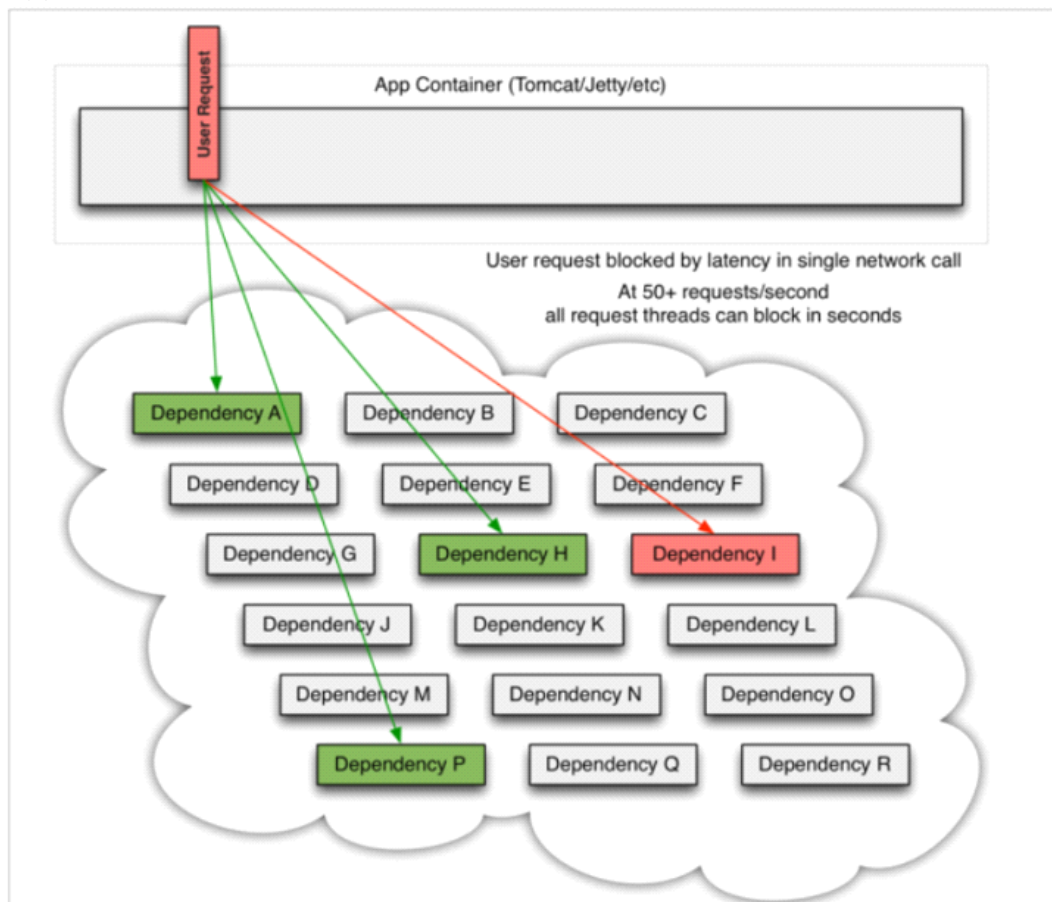
Hystrix是Netflix开源的一个延迟和容错库, 用于隔离访问远程服务, 防止出现级联失败。

7.2 雪崩问题

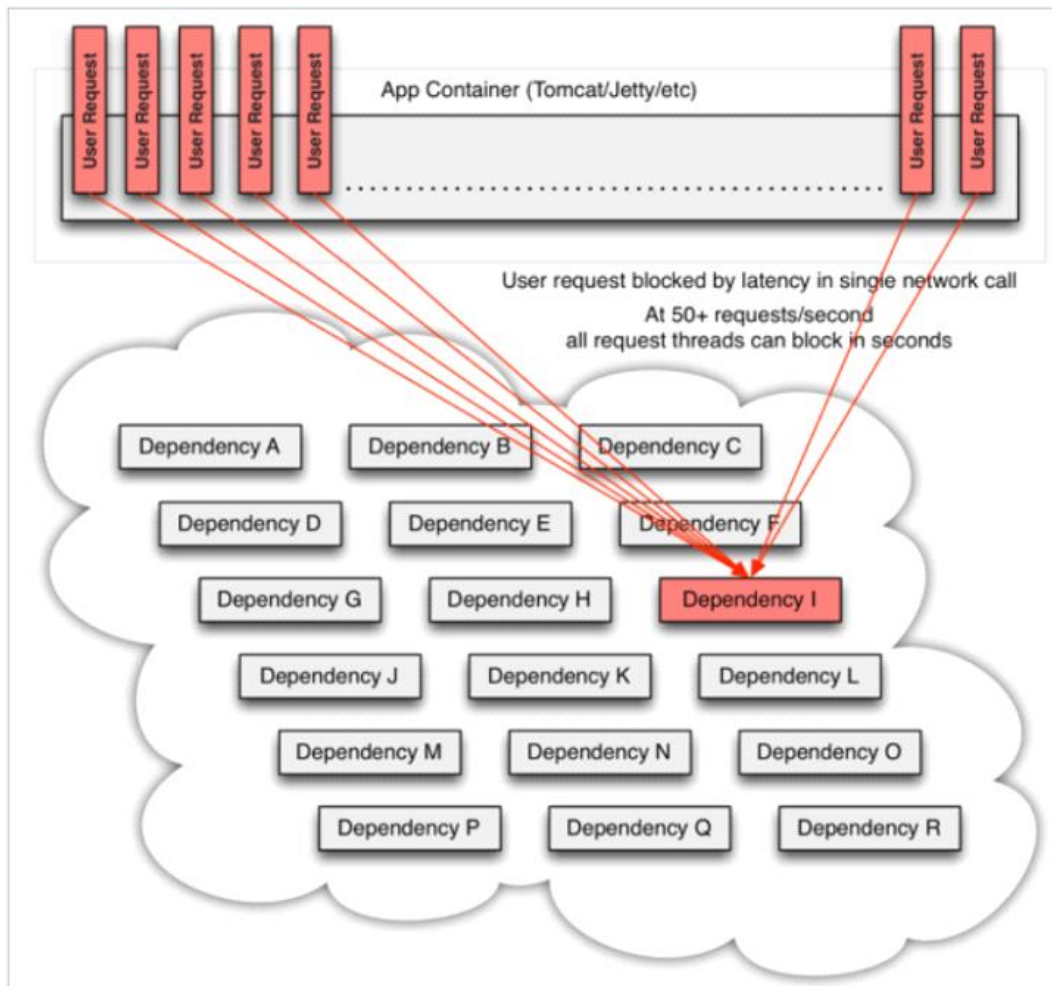
微服务中, 服务间调用关系错综复杂, 一个请求, 可能需要调用多个微服务接口才能实现, 会形成非常复杂的调用链路:



如图，一次业务请求，需要调用A、P、H、I四个服务，这四个服务又可能调用其它服务。如果此时，某个服务出现异常：



例如：微服务I发生异常，请求阻塞，用户请求就不会得到响应，则tomcat的这个线程不会释放，于是越来越多的用户请求到来，越来越多的线程会阻塞。



服务器支持的线程和并发数有限，请求一直阻塞，会导致服务器资源耗尽，从而导致所有其它服务都不可用，形成雪崩效应。

这就好比，一个汽车生产线，生产不同的汽车，需要使用不同的零件，如果某个零件因为种种原因无法使用，那么就会造成整台车无法装配，陷入等待零件的状态，直到零件到位，才能继续组装。此时如果有很多个车型都需要这个零件，那么整个工厂都将陷入等待的状态，导致所有生产都陷入瘫痪。一个零件的波及范围不断扩大。

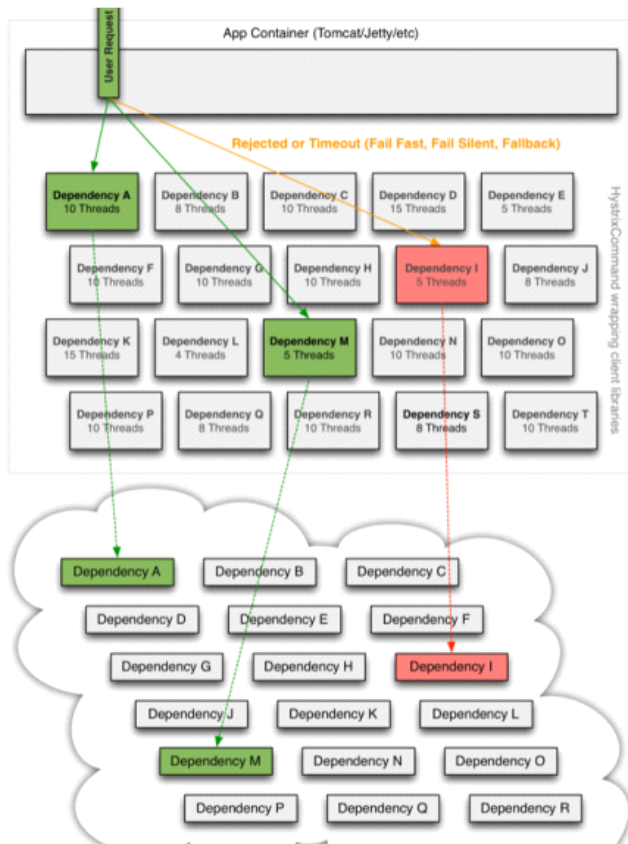
Hystrix解决雪崩问题的手段，主要包括：

- 线程隔离
- 服务降级

7.3 线程隔离&服务降级

7.3.1 原理

线程隔离示意图



解读：

Hystrix为每个依赖服务调用分配一个小的线程池，如果线程池已满调用将被立即拒绝，默认不采用排队，加速失败判定时间。

用户的请求将不再直接访问服务，而是通过线程池中的空闲线程来访问服务，如果线程池已满，或者请求超时，则会进行降级处理，什么是服务降级？

服务降级：可以优先保证核心服务。

用户的请求故障时，不会被阻塞，更不会无休止的等待或者看到系统崩溃，至少可以看到一个执行结果(例如返回友好的提示信息)。

服务降级虽然会导致请求失败，但是不会导致阻塞，而且最多会影响这个依赖服务对应的线程池中的资源，对其它服务没有响应。触发Hystrix服务降级的情况：

- 线程池已满
- 请求超时

7.3.2 动手实践

服务降级：及时返回服务调用失败的结果，让线程不因为等待服务而阻塞

1、引入依赖

在consumer-demo消费端系统的pom.xml文件添加如下依赖：

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

2、开启熔断

在启动类ConsumerApplication上添加注解：@EnableCircuitBreaker

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
public class ConsumerApplication {
  // ...
}
```

可以看到，我们类上的注解越来越多，在微服务中，经常会引入上面的三个注解，于是spring就提供了一个组合注解：@SpringCloudApplication

```

/**
 * @author Spencer Gibb
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
public @interface SpringCloudApplication {
}

```

因此，我们可以使用这个组合注解来代替之前的3个注解：

```

@SpringCloudApplication
public class ConsumerApplication {
    // ...
}

```

3、编写降级逻辑

当目标服务的调用出现故障，我们希望快速失败，给用户一个友好提示。因此需要提前编写好失败时的降级处理逻辑，要使用HystrixCommand来完成。

改造ConsumerController处理类，如下：

```

package com.colin.consumer.controller;

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-04-09
 * Time: 18:01
 */
@RestController
@RequestMapping("/consumer")
@Slf4j
public class ConsumerController {
    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private DiscoveryClient discoveryClient;

    @GetMapping("/{id}")
    @HystrixCommand(fallbackMethod = "queryByIdFallback")
    public String queryById(@PathVariable Integer id) {
        String url = "http://user-service/user/" + id;
        return restTemplate.getForObject(url, String.class);
    }

    public String queryByIdFallback(Integer id) {
        log.error("查询用户信息失败。id : " + id);
        return "对不起，网络太拥挤了";
    }
}

```

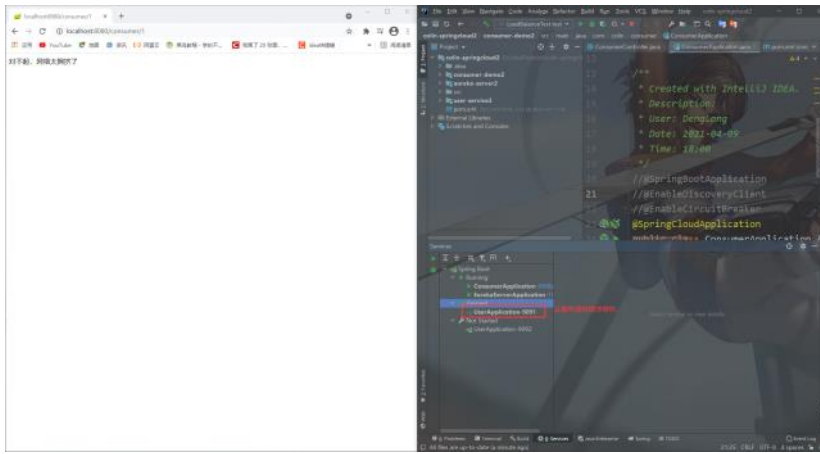
注意：熔断的降级逻辑方法必须跟正常逻辑方法保证拥有相同的参数列表和返回值声明。

失败逻辑中返回User对象没有太大意义，一般会返回友好提示。所以把queryById的方法改造为返回String，反正也是json数据。这样失败逻辑中返回一个错误说明，会比较方便。

说明：@HystrixCommand(fallbackMethod="queryByIdFallback")：用来声明一个降级逻辑的方法

测试：

当user-service正常提供服务时，访问与以前一致。但是当将user-service停机时，会发现页面返回了降级处理信息：



4. 默认的fallback

刚才把fallback写在了某个业务方法上，如果这样的方法很多，那岂不是要写很多。所以可以把Fallback配置加在类上，实现默认fallback；再次改造ConsumerController

```
package com.colin.consumer.controller;

import com.colin.consumer.pojo.User;
import com.netflix.hystrix.contrib.javanica.annotation.DefaultProperties;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import java.util.List;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-04-09
 * Time: 18:01
 */
@RestController
@RequestMapping("/consumer")
@Slf4j
@DefaultProperties(defaultFallback = "defaultFallback")
public class ConsumerController {
    @Autowired
    private RestTemplate restTemplate;

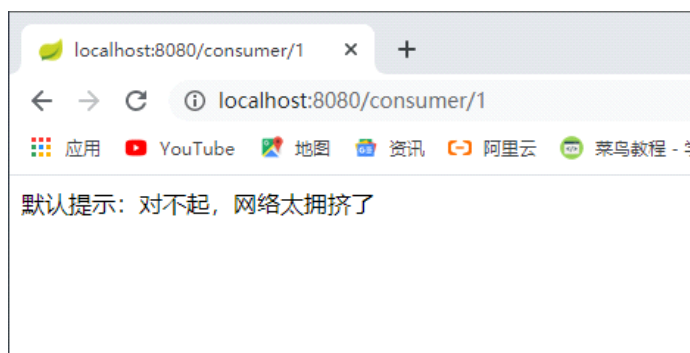
    @Autowired
    private DiscoveryClient discoveryClient;

    @GetMapping("/{id}")
    // @HystrixCommand(fallbackMethod = "queryByIdFallback")
    @HystrixCommand
    public String queryById(@PathVariable Integer id) {
        String url = "http://user-service/user/" + id;
        return restTemplate.getForObject(url, String.class);
    }

    public String queryByIdFallback(Integer id) {
        log.error("查询用户信息失败。id : " + id);
        return "对不起，网络太拥挤了";
    }

    public String defaultFallback() {
        return "默认提示：对不起，网络太拥挤了";
    }
}
```

@DefaultProperties(defaultFallback="defaultFallback"):在类上指明统一的失败降级方法；该类中所有方法返回类型要与处理失败的方法的返回类型一致。



5、超时设置

在之前的案例中，请求在超时1秒后都会返回错误信息，这是因为Hystrix的默认超时时长为1，我们可以通过配置修改这个值；修改consumer-demo的application.yml添加如下配置：

```
hystrix:
  command:
    default:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 2000
```

这个配置会作用于全局所有方法。为了方便复制到yml配置文件中，可以复制

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=2000 到yml文件中会自动格式化后再进行修改。

为了触发超时，可以在user-service的UserController类的queryById方法中增加休眠2秒：

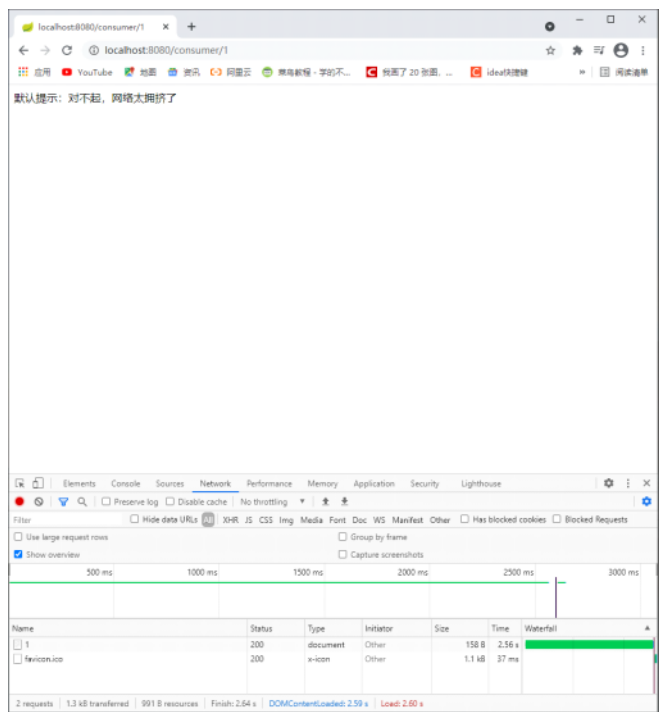
```
package com.colin.user.controller;

import com.colin.user.pojo.User;
import com.colin.user.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

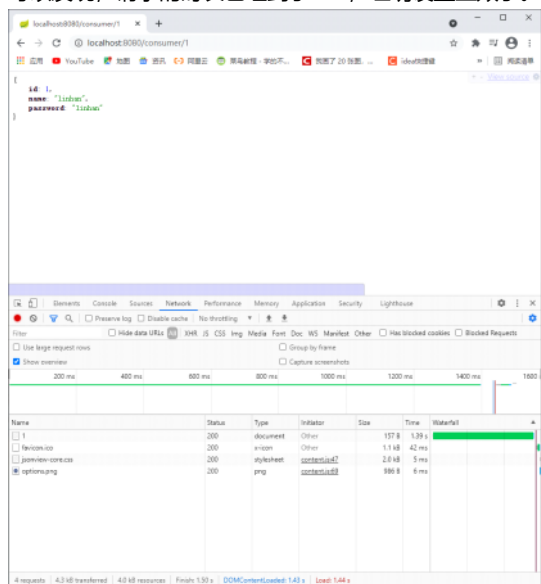
/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-04-09
 * Time: 17:52
 */
@RestController
@RequestMapping("/user")
public class UserController {
    @Autowired
    private UserService userService;

    @GetMapping("/{id}")
    public User queryById(@PathVariable Integer id) {
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return userService.queryById(id);
    }
}
```

测试：



可以发现, 请求的时长已经到了2s+, 证明设置生效了。如果把睡眠时间修改到2秒以下(下图示例为睡眠时间1s的情况), 又可以正常访问。



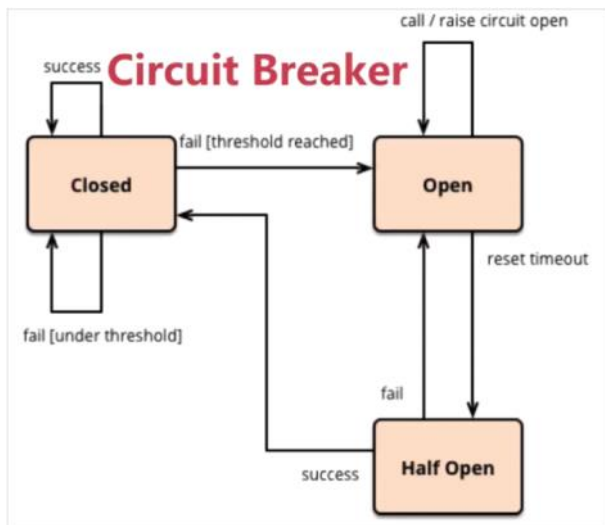
7.4 服务熔断

7.4.1 熔断原理

在服务熔断中, 使用的熔断器, 也叫断路器, 其英文单词为: Circuit Breaker熔断机制与家里使用的电路熔断原理类似; 如果发生短路的时候能立刻熔断电路, 避免发生灾难。在分布式系统中应用服务熔断后, 服务调用方可以自己进行判断哪些服务反应慢或存在大量超时, 可以针对这些服务进行主动熔断, 防止整个系统被拖垮。

Hystrix的服务熔断机制, 可以实现弹性容错, 当服务请求情况好转之后, 可以自动重连。通过断路的方式, 将后续请求直接拒绝, 一段时间(默认5秒)之后允许部分请求通过, 如果调用成功则回到断路器关闭状态, 否则继续打开, 拒绝请求的服务。

Hystrix的熔断状态机模型:



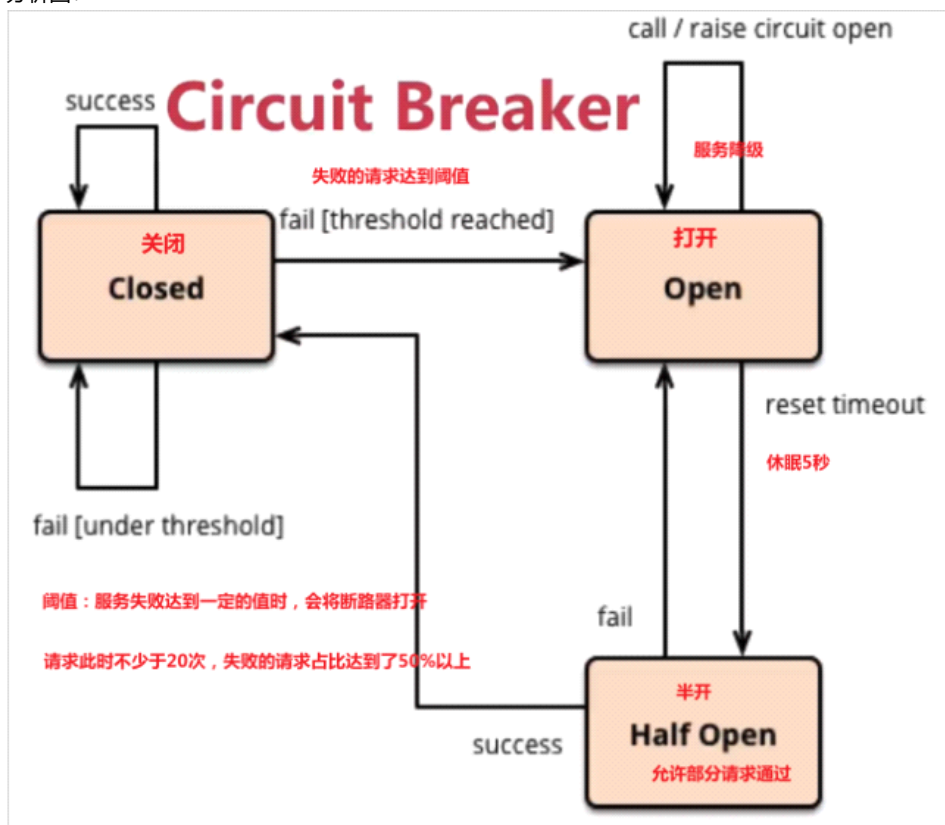
状态机有3个状态：

closed：关闭状态(断路器关闭)，所有请求都正常访问。

Open：打开状态(断路器打开)，所有请求都会被降级。Hystrix会对请求情况计数，当一定时间内失败请求百分比打到阈值，则触发熔断，断路器完全打开。默认失败比例的阈值是50%，请求次数最少不低于20次。

Half Open：半开状态，不是永久的，断路器打开后会进入休眠时间(默认是5s)。随后断路器会自动进入半开状态。此时会释放部分请求通过，若这些请求都是健康的，则会关闭断路器，否则继续保持打开，再次进行休眠计时。

分析图：



7.4.2 动手实践

为了能够准确控制请求的成功或失败，在consumer-demo的处理器业务方法中加入一段逻辑；修改ConsumerController

```
@GetMapping("/{id}")
@HystrixCommand
public String queryById(@PathVariable Integer id) {
    if (id == 1) {
        throw new RuntimeException("太忙了");
    }
    String url = "http://user-service/user/" + id;
    return restTemplate.getForObject(url, String.class);
}
```

这样如果参数是id为1，一定失败，其他情况都成功。(不要忘记冲正user-service中的休眠逻辑) 我们准备两个请求窗口：

- 一个请求 <http://localhost:8080/consumer/1> 注定失败
- 一个请求 <http://localhost:8080/consumer/2> 肯定成功

当我们疯狂访问id为1的请求时(超过20次), 就会触发熔断。断路器会打开, 一切请求都会被降级处理。

此时再次访问id为2的请求, 会发现返回的也是失败, 而且失败时间很短, 只有20毫秒左右。进入半开状态之后2是可以的。



不过, 默认的熔断触发要求较高, 休眠时间窗较短, 为了测试方便, 我们可以通过配置修改熔断策略:

```
hystrix:
  command:
    default:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 2000 # 服务降级超时时间
      circuitBreaker:
        errorThresholdPercentage: 50 # 触发熔断错误比例阈值, 默认值50%
        sleepWindowInMilliseconds: 10000 # 熔断后休眠时长, 默认值5秒
        requestVolumeThreshold: 10
```

为了方便复制上述配置, 可以使用如下格式复制到yml文件中会自动格式化:

```
hystrix.command.default.circuitBreaker.requestVolumeThreshold=10
hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds=10000
hystrix.command.default.circuitBreaker.errorThresholdPercentage=50
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=2000
```

上述的配置项可以参考HystrixCommandProperties类。

8、Feign

在前面的学习中, 我们使用了Ribbon的负载均衡功能, 大大简化了远程调用时的代码:

```
String baseUrl = "http://user-service/user/";
User user = this.restTemplate.getForObject(baseUrl + id, User.class)
```

如果就学到这里, 你可能以后需要编写类似的大量重复代码, 格式基本相同, 无非参数不一样。有没有更优雅的方式, 来对这些代码再次优化呢?

这就是我们接下来要学的Feign的功能了。

8.1 简介

为什么叫伪装?

Feign可以把Rest的请求进行隐藏, 伪装成类似SpringMVC的Controller一样。你不用再自己拼接url, 拼接参数等等操作, 一切都交给Feign去做。

项目主页: <https://github.com/OpenFeign/feign>

8.2 快速入门

8.2.1 导入依赖

在consumer-demo的pom文件中引入依赖

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

8.2.2 Feign的客户端

```
package com.colin.consumer.client;

import com.colin.consumer.pojo.User;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
```

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

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-04-13
 * Time: 9:41
 */
@FeignClient("user-service")
public interface UserClient {
    // http://user-service/user/1
    @GetMapping("/user/{id}")
    User queryById(@PathVariable Integer id);
}
```

- 首先这是一个接口，Feign会通过动态代理，帮我们生成实现类。这点跟Mybatis的mapper很像
- @FeignClient，声明这是一个Feign的客户端，同时通过value属性指定服务名称
- 接口中的定义方法，完全采用SpringMVC的注解，Feign会根据注解帮我们生成URL，并访问获取结果
- @GetMapping中的/user，请不要忘记，因为Feign需要拼接可访问的地址

编写新的控制器类 ConsumerFeignController，使用UserClient访问：

```
package com.colin.consumer.controller;

import com.colin.consumer.client.UserClient;
import com.colin.consumer.pojo.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-04-13
 * Time: 9:49
 */
@RestController
@RequestMapping("/cf")
public class ConsumerFeignController {
    @Autowired
    private UserClient userClient;

    @GetMapping("/{id}")
    public User queryById(@PathVariable Integer id) {
        return userClient.queryById(id);
    }
}
```

8.2.3 开启Feign功能

在ConsumerApplication启动类上，添加注解，开启Feign功能

```
package com.colin.consumer;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.SpringCloudApplication;
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.loadbalancer.LoadBalanced;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.cloud.openfeign.EnableFeignClients;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-04-09
 * Time: 18:00
 */
//@SpringBootApplication
//@EnableDiscoveryClient
//@EnableCircuitBreaker
@SpringCloudApplication
@EnableFeignClients // 开启Feign功能
public class ConsumerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class, args);
    }
}
```

```

    }

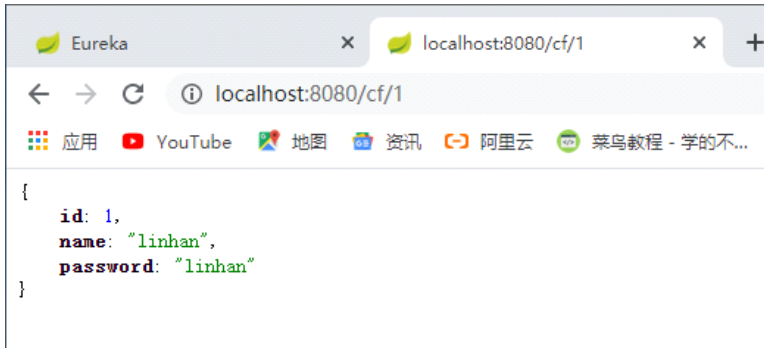
    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

```

Feign中已经自动集成了Ribbon负载均衡，因此不需要自己定义RestTemplate进行负载均衡的配置。

8.2.4 启动测试：

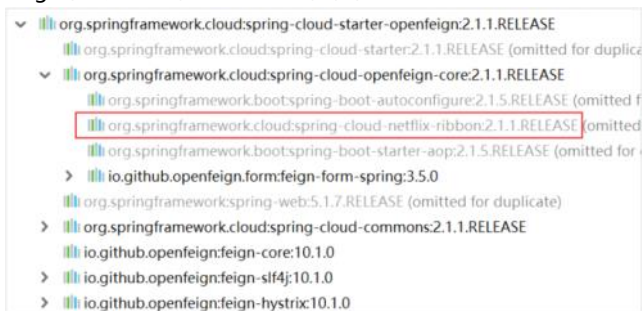
访问接口：<http://localhost:8080/cf/1>



正常获取到了结果。

8.3 负载均衡

Feign中本身已经集成了Ribbon依赖和自动配置：



因此不需要额外引入依赖，也不需要再注册RestTemplate对象。

Feign内置的ribbon默认设置了请求超时时长，默认是1000，我们可以通过手动配置来修改这个超时时长：

```

ribbon:
    ReadTimeout: 2000 # 读取超时时长
    ConnectTimeout: 1000 # 建立链接超时时长

```

或者为某一个具体service指定

```

user-service
ribbon:
    ReadTimeout: 2000 # 读取超时时长
    ConnectTimeout: 1000 # 建立链接超时时长

```

在user-service中增加睡眠时间2s进行测试

因为ribbon内部有重试机制，一旦超时，会自动重新发起请求。如果不希望重试，可以添加配置：

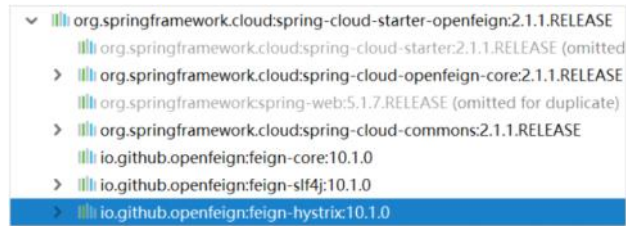
```

ribbon:
    ConnectTimeout: 1000 # 连接超时时长
    ReadTimeout: 2000 # 数据通信超时时长
    MaxAutoRetries: 0 # 当前服务器的重试次数
    MaxAutoRetriesNextServer: 0 # 重试多少次服务

```

8.4 Hystrix支持

Feign默认也有对Hystrix的集成:



只不过，默认情况下是关闭的。需要通过下面的参数来开启:

1、修改consumer-demo的application.yml 添加如下配置:

```
feign:
  hystrix:
    enabled: true # 开启Feign的熔断功能
```

但是，Feign中的Fallback配置不像Ribbon中的那样简单了。

2、首先，要定义一个类，实现刚才编写的UserFeignClient，作为fallback的处理类

```
package com.colin.consumer.client;

import com.colin.consumer.pojo.User;
import org.springframework.stereotype.Component;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-04-13
 * Time: 10:17
 */
@Component
public class UserClientFallback implements UserClient {
    @Override
    public User queryById(Integer id) {
        User user = new User();
        user.setId(id);
        user.setName("用户异常");
        return user;
    }
}
```

3、然后在UserClient中，指定刚才编写的实现类

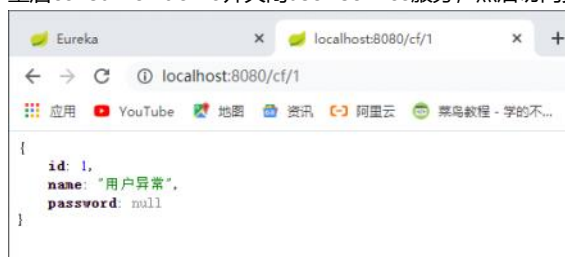
```
package com.colin.consumer.client;

import com.colin.consumer.pojo.User;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-04-13
 * Time: 9:41
 */
@FeignClient(value = "user-service", fallback = UserClientFallback.class)
public interface UserClient {
    // http://user-service/user/1
    @GetMapping("/user/{id}")
    User queryById(@PathVariable Integer id);
}
```

4、重启测试:

重启consumer-demo并关闭user-service服务，然后访问页面: <http://localhost:8080/cf/1>



8.5 请求压缩(了解)

Spring Cloud Feign支持对请求和响应进行GZIP压缩，以减少通信过程中的性能损耗。通过下面的参数即可开启请求与响应的压缩功能：

在consumer-demo的application.yml中增加配置

```
feign:
  compression:
    request:
      enabled: true # 开启请求压缩
    response:
      enabled: true # 开启响应压缩
```

同时，我们也可以对请求的数据模型，以及触发压缩的大小下限进行设置：

```
feign:
  compression:
    request:
      enabled: true # 开启请求压缩
      mime-types: text/html,application/xml,application/json # 设置压缩的数据类型
      min-request-size: 2048 # 设置触发压缩的大小下限
```

注意：上面的数据类型、压缩大小下限均为默认值。

8.6 日志级别(了解)

前面讲过，通过logging.level.com.colin=debug来设置日志级别。然后这个对Feign客户端而言不会产生效果。因为@FeignClient注解修改的客户端在被代理时，都会创建一个新的Feign.Logger实例。我们需要额外指定这个日志的级别才可以。

1、在consumer-demo的配置文件中设置com.colin包下的日志级别都为debug，修改consumer-demo的application.yml 添加如下配置：

```
logging:
  level:
    com.colin: debug
```

2、编写配置类，定义日志级别

```
package com.colin.consumer.config;

import feign.Logger;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-04-13
 * Time: 10:29
 */
@Configuration
public class FeignConfig {
    @Bean
    Logger.Level feignLoggerLevel() {
        // 记录所有请求和响应的明细，包括头信息、请求体、元数据
        return Logger.Level.FULL;
    }
}
```

这里指定的Level级别是FULL，Feign支持4中级别：

- NONE：不记录任何日志信息，这是默认值。
- BASIC：仅记录请求的方法，URL以及响应状态码和执行时间
- HEADERS：在BASIC的基础上，额外记录了请求和响应的头信息
- FULL：记录所有请求和响应的明细，包括头信息、请求体、元数据。

3、在consumer-demo的UserClient接口类上的@FeignClient注解中指定配置类：

别忘了去掉user-service的休眠时间。

```
package com.colin.consumer.client;

import com.colin.consumer.config.FeignConfig;
import com.colin.consumer.pojo.User;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
```



```

* Date: 2021-04-13
* Time: 9:41
*/
@FeignClient(value = "user-service", fallback = UserClientFallback.class, configuration = FeignConfig.class)
public interface UserClient {
    // http://user-service/user/1
    @GetMapping("/user/{id}")
    User queryById(@PathVariable Integer id);
}

```

4、重启项目，即可看到每次访问的日志。

9、Spring Cloud Gateway网关

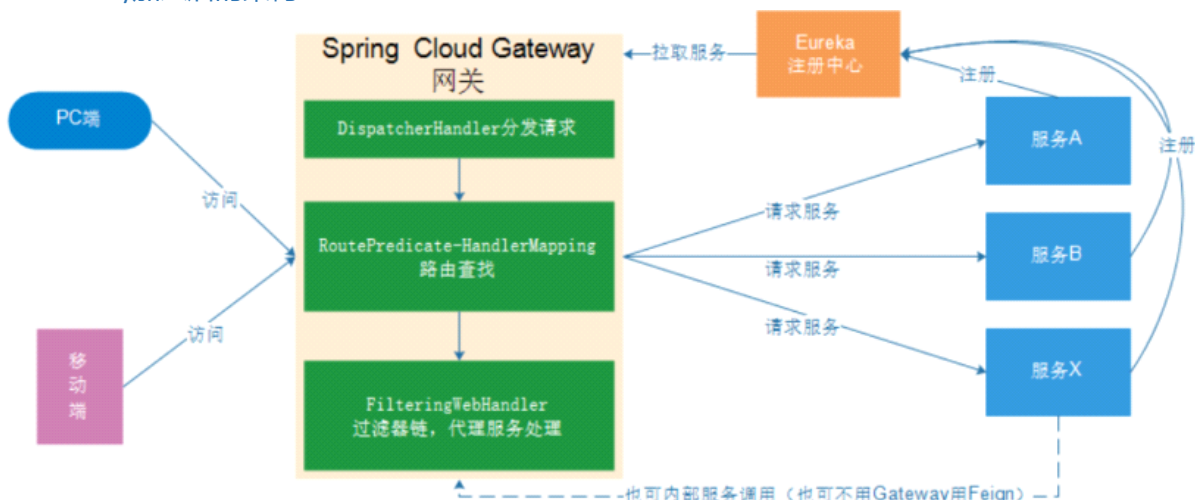
9.1 简介

- Spring Cloud Gateway是Spring官网基于Spring5.0、Spring Boot2.0、Project Reactor等技术开发的网关服务。
- Spring Cloud Gateway基于Filter链提供网关基本功能：安全、监控/埋点、限流等。
- Spring Cloud Gateway为微服务架构提供简单、有效且统一的API路由管理方式。
- Spring Cloud Gateway是替代Netflix Zuul的一套解决方案。

Spring Cloud Gateway组件的核心是一系列的过滤器，通过这些过滤器可以将客户端发送的请求转发(路由)到对应的微服务。Spring Cloud Gateway是加在整个微服务最前沿的防火墙和代理器，隐藏微服务节点IP端口信息，从而加强安全保护。Spring Cloud Gateway本身也是一个微服务，需要注册到Eureka服务注册中心。

网关的核心功能是：过滤和路由。

9.2 Gateway加入后的架构



不管是来自于客户端(PC或移动端)的请求，还是服务内部调用。一切对服务的请求都可经过网关，然后再由网关来实现鉴权、动态路由等操作。Gateway就是我们服务的统一入口。

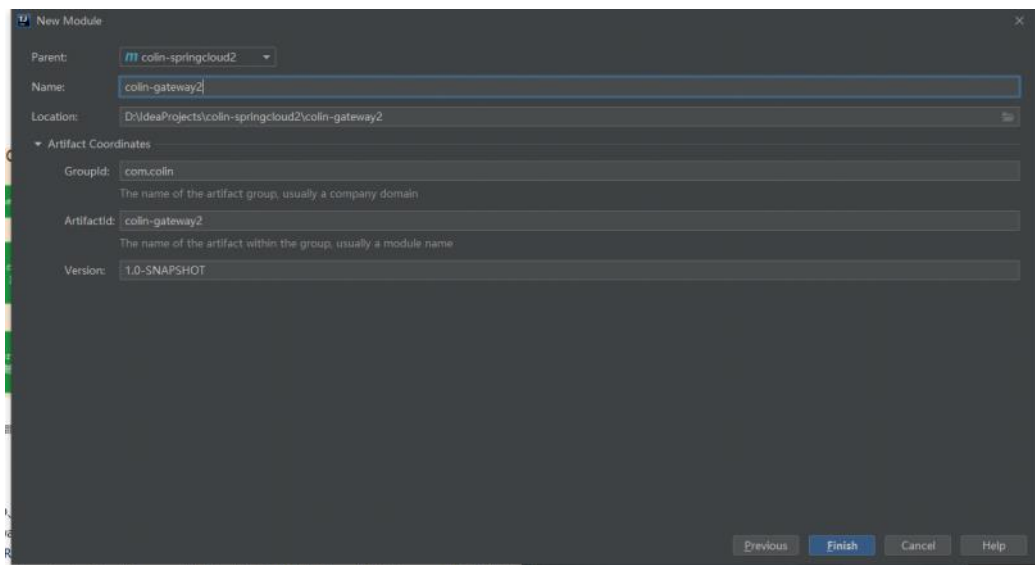
9.3 核心概念

- **路由(route)** 路由信息的组成：由一个ID、一个目的URL、一组断言工厂、一组Filter组成。如果路由断言为真，说明请求URL和配置路由匹配。
- **断言(Predicate)** Spring Cloud Gateway中的断言函数输入类型是Spring 5.0框架中的ServerWebExchange。Spring Cloud Gateway的断言函数允许开发者去定义匹配来自于HTTP Request中的任何信息比如请求头和参数。
- **过滤器(Filter)** 一个标准的Spring WebFilter。Spring Cloud Gateway中的Filter分为两种类型的Filter，分别是Gateway Filter和Global Filter。过滤器Filter将会对请求和响应进行修改处理。

9.4 快速入门

通过网关系统colin-gateway将包含有/user的请求路由到http://127.0.0.1:9091/user/用户id

9.4.1 新建工程



增加依赖:

```
<?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">

    <parent>
        <artifactId>colin-springcloud2</artifactId>
        <groupId>com.colin</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>colin-gateway2</artifactId>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-gateway</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
        </dependency>
    </dependencies>
</project>
```

9.4.2 编写启动类

在colin-gateway中创建GatewayApplication启动类

```
package com.colin.gateway;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-04-14
 * Time: 18:47
 */
@SpringBootApplication
@EnableDiscoveryClient
public class GatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }
}
```

9.4.3 编写配置

创建colin-gateway下的application.yml文件:

```
server:
  port: 10010
spring:
  application:
    name: api-gateway
eureka:
  client:
    service-url:
```

```
defaultZone: http://127.0.0.1:10086/eureka
instance:
  prefer-ip-address: true
```

需要用网关代理user-service服务，先看一下控制面板中的服务状态：

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
USER-SERVICE	n/a (1)	(1)	UP (1) - localhost:user-service:9091

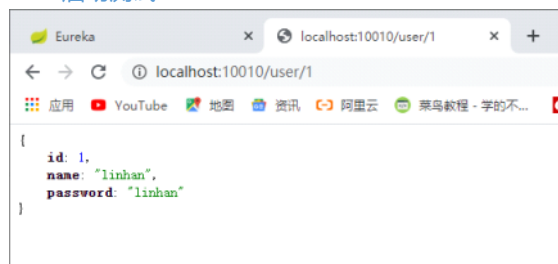
- ip为：127.0.0.1
- 端口为：9091

修改colin-gateway下的application.yml文件为：

```
server:
  port: 10010
spring:
  application:
    name: api-gateway
  cloud:
    gateway:
      routes:
        # 路由id, 可以随意写
        - id: user-service-route
          # 代理的服务地址
          uri: http://127.0.0.1:9091
          # 路由断言, 可以配置映射路径
          predicates:
            - Path=/user/**
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:10086/eureka
  instance:
    prefer-ip-address: true
```

将符合Path规则的一切请求，都代理到uri参数指定的地址。本例中，我们将路径中包含有/user/**开头的请求，代理到<http://127.0.0.1:9091>

9.4.5 启动测试



<http://localhost:10010/user/1> -> <http://localhost:9091/user/1>

9.5 面向服务的路由

在刚才的路由规则中，把路径对应的服务地址写死了。如果同一服务有多个实例的话，这样做显然不合理。应该根据服务的名称，去Eureka注册中心查找对应的所有实例列表，然后进行动态路由。

9.5.1 修改映射配置，通过服务名称获取

修改colin-gateway下的application.yml文件如下：

```
server:
  port: 10010
spring:
  application:
    name: api-gateway
  cloud:
    gateway:
      routes:
        # 路由id, 可以随意写
        - id: user-service-route
          # 代理的服务地址
          uri: http://127.0.0.1:9091
          # 路由断言, 可以配置映射路径
          predicates:
            - Path=/user/**
```

```

        - Path=/user/**

eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:10086/eureka
  instance:
    prefer-ip-address: true

```

路由配置中uri所用的协议为lb时(以uri:lb://user-service为例), gateway将使用LoadBalancerClient把user-service通过eureka解析为实际的主机和端口, 并进行ribbon负载均衡。

9.5.2 启动测试

再次启动colin-gateway, 这次gateway进行代理时, 会利用Ribbon进行负载均衡访问: <http://localhost:10010/user/1> 日志中可以看到使用了负载均衡器:



9.6 路由前缀

客户端的请求地址与微服务的服务地址如果不一致的时候, 可以通过配置路径过滤器实现路径前缀的添加和去除。

提供服务的地址: <http://127.0.0.1:9091/user/1>

- 添加前缀: 对请求地址添加前缀路径之后再作为代理的服务地址:
<http://127.0.0.1:10010/1> -> <http://127.0.0.1:9091/user/1> 添加前缀路径/user
- 去除前缀: 将请求地址中路径去除一些前缀路径之后再作为代理的服务地址:
<http://127.0.0.1:10010/api/user/1> -> <http://127.0.0.1:9091/user/1> 去除前缀路径/api

9.6.1 添加前缀

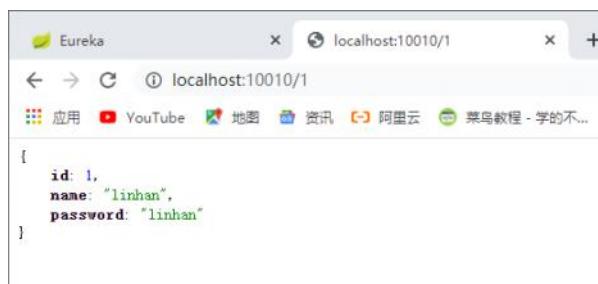
在gateway中可以通过配置路由的过滤器PrefixPath, 实现映射路径中地址的添加。

修改colin-gateway的application.yml文件:

```

server:
  port: 10010
spring:
  application:
    name: api-gateway
  cloud:
    gateway:
      routes:
        # 路由id, 可以随意写
        - id: user-service-route
          # 代理的服务地址
          uri: http://127.0.0.1:9091
          # 路由断言, 可以配置映射路径
          predicates:
            - Path=/**
          filters:
            # 添加请求路径的前缀
            - PrefixPath=/user
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:10086/eureka
  instance:
    prefer-ip-address: true

```



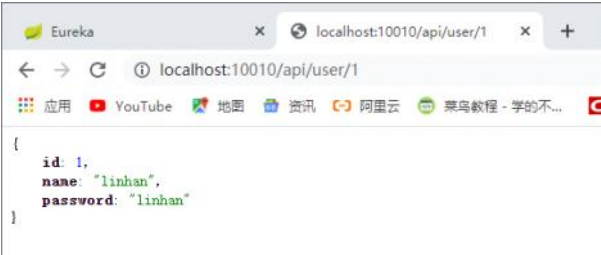
通过PrefixPath=/xxx来指定路由要添加的前缀。也就是：
PrefixPath=/user <http://localhost:10010/1> -> <http://localhost:9091/user/1>
PrefixPath=/user/colin <http://localhost:10010/1> -> <http://localhost:9091/user/colin/1>

9.6.2 去除前缀

在gateway中可以通过配置路由的过滤器StripPrefix，实现映射路径中地址的去除。

修改colin-gateway的application.yml文件：

```
server:
  port: 10010
spring:
  application:
    name: api-gateway
  cloud:
    gateway:
      routes:
        # 路由id, 可以随意写
        - id: user-service-route
          # 代理的服务地址
          uri: http://127.0.0.1:9091
          uri: lb://user-service
          # 路由断言, 可以配置映射路径
          predicates:
            - Path=/api/user/**
          filters:
            # 添加请求路径的前缀
            - PrefixPath=/user
            # 1表示过滤一个路径, 2表示两个路径, 以此类推
            - StripPrefix=1
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:10086/eureka
  instance:
    prefer-ip-address: true
```



通过StripPrefix=1来指定路由要去掉的前缀个数。如：路径/api/user/1 将会被代理到/user/1。也就是：

StripPrefix=1 <http://localhost:10010/api/user/1> -> <http://localhost:9091/user/1>
StripPrefix=2 <http://localhost:10010/api/user/1> -> <http://localhost:9091/1>

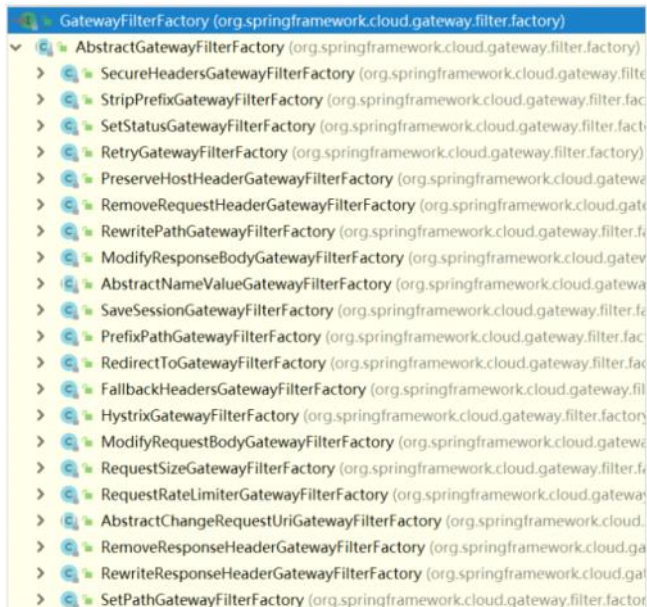
9.7 过滤器

9.7.1 简介

Gateway作为网关的其中一个重要功能，就是实现请求的鉴权。而这个动作往往是通过网关提供的过滤器来实现的。前面的路由前缀章节中的功能也是使用过滤器实现的。

Gateway自带过滤器有几十个，常见自带过滤器有：

过滤器名称	说明
AddRequestHeader	对匹配上的请求加上Header
AddRequestParameters	对匹配上的请求路由添加参数
AddResponseHeader	对从网关返回的响应添加Header
StripPrefix	对匹配上的请求路径去除前缀

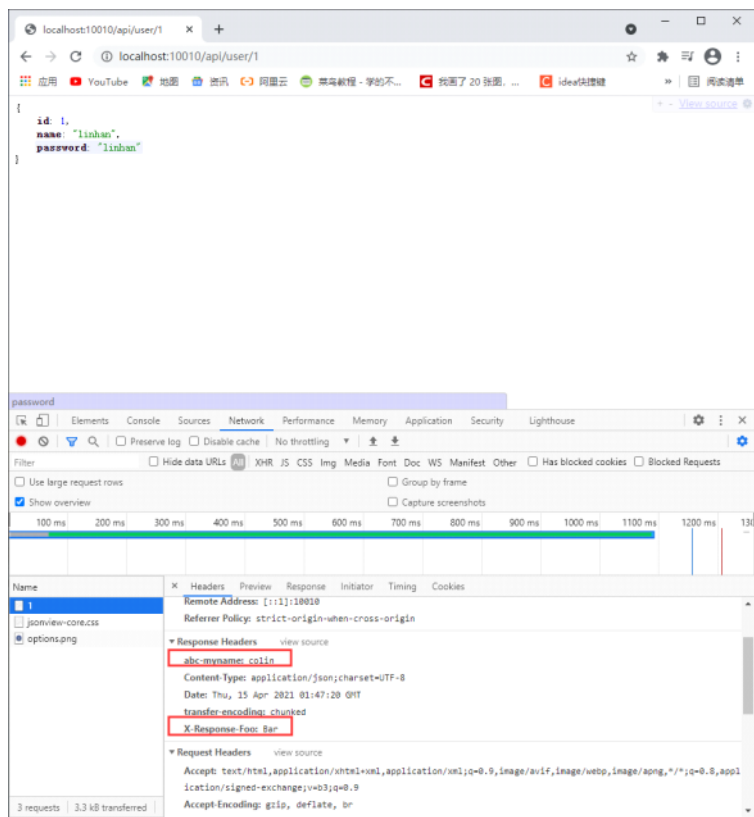


配置全局默认过滤器

这些自带的过滤器可以和使用路由前缀章节中的用法类似，也可以将这些过滤器配置成不只是针对某个路由，而是可以针对所有路由生效，也就是配置默认过滤器：

```
server:
  port: 10010
spring:
  application:
    name: api-gateway
  cloud:
    gateway:
      routes:
        # 路由id, 可以随意写
        - id: user-service-route
          # 代理的服务地址
          uri: http://127.0.0.1:9091
          uri: lb://user-service
          # 路由断言, 可以配置映射路径
          predicates:
            - Path=/api/user/**
          filters:
            # 添加请求路径的前缀
            - PrefixPath=/user
            # 1表示过滤一个路径, 2表示两个路径, 以此类推
            - StripPrefix=1
            # 默认过滤器, 对所有路由都生效
            default-filters:
              - AddResponseHeader=X-Response-Foo, Bar
              - AddResponseHeader=abc-myname, colin
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:10086/eureka
  instance:
    prefer-ip-address: true
```

上述配置后，再访问<http://localhost:10010/api/user/1>的话，那么可以从其响应中查看到如下信息：

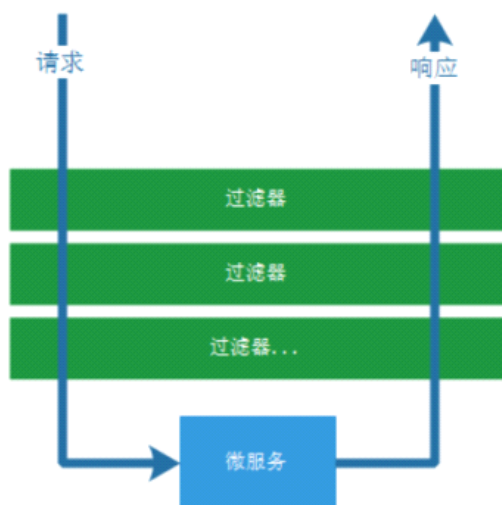


过滤器类型：Gateway实现方式上，有两种过滤器。

- 局部过滤器：通过Spring.cloud.gateway.route.fileters配置在具体路由下，只作用在当前路由上。如果配置spring.cloud.gateway.default-filters上会对所有路由失效也算是全局的过滤器，但是这些过滤器的实现上都要实现GatewayFilterFactory接口。
- 全局过滤器：不需要在配置文件中配置，作用在所有的路由上，实现GlobalFilter接口即可。

9.7.2 执行生命周期

Spring Cloud Gateway的Filter的生命周期也类似Spring MVC的拦截器有两个："pre"和"post"。"pre"和"post"分别会在请求被执行前调用和被执行后调用



这里的pre和post可以通过过滤器的GatewayFilterChain执行filter方法前后来实现。

9.7.3 使用场景

常见的应用场景如下：

- 请求鉴权：一般GatewayFilterChain执行filter方法前，如果发现没有访问权限，直接就返回空。
- 异常处理：一般GatewayFilterChain执行filter方法后，记录异常并返回。
- 服务调用时长统计：GatewayFilterChain执行filter方法前后根据时间统计。

9.8 自定义过滤器

9.8.1 自定义局部过滤器

需求：在过滤器(MyParamGatewayFilterFactory)中将<http://localhost:10010/api/user/8?name=colin>中的参数name的值获取到并输出到控制台，并且参数名是可变的，也就是不一定每次都是name，需要可以通过配置过滤器的时候用到配置参数名。

在application.yml中对某个路由配置过滤器，该过滤器可以在控制台输出配置文件中指定名称的请求参数的值。

1、编写过滤器

```
package com.colin.gateway.filter;

import org.springframework.cloud.gateway.filter.GatewayFilter;
import org.springframework.cloud.gateway.filter.factory.AbstractGatewayFilterFactory;
import org.springframework.http.server.reactive.ServerHttpRequest;
import org.springframework.stereotype.Component;

import java.util.Arrays;
import java.util.List;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-04-15
 * Time: 13:52
 */
@Component
public class MyParamGatewayFilterFactory extends AbstractGatewayFilterFactory<MyParamGatewayFilterFactory.Config> {
    private static final String PARAM_NAME = "param";

    public MyParamGatewayFilterFactory() {
        super(Config.class);
    }

    public List<String> shortcutFieldOrder() {
        return Arrays.asList(PARAM_NAME);
    }

    @Override
    public GatewayFilter apply(Config config) {
        return (exchange, chain) -> {
            //http://localhost:10010/api/user/1?name=colin config.param = name
            ServerHttpRequest request = exchange.getRequest();
            if (request.getQueryParams().containsKey(config.param)) {
                request.getQueryParams().get(config.param).forEach((v) -> {
                    System.out.printf("--局部过滤器--获得参数 %s = %s ----", config.param, v);
                });
            }
            return chain.filter(exchange);
        };
    }

    // 读取过滤器配置的参数
    public static class Config {
        // 对应配置在application.yml配置文件中的过滤器参数
        private String param;

        public String getParam() {
            return param;
        }

        public void setParam(String param) {
            this.param = param;
        }
    }
}
```

2、修改配置文件

```
server:
  port: 10010
spring:
  application:
    name: api-gateway
  cloud:
    gateway:
      routes:
        # 路由id, 可以随意写
        - id: user-service-route
          # 代理的服务地址
          uri: http://127.0.0.1:9091
          uri: lb://user-service
          # 路由断言, 可以配置映射路径
          predicates:
            - Path=/api/user/**
```

```

        filters:
            # 添加请求路径的前缀
            - PrefixPath=/user
            # 1表示过滤一个路径, 2表示两个路径, 以此类推
            - StripPrefix=1
            # 自定义过滤器
            - MyParam=name
        # 默认过滤器, 对所有路由都生效
        default-filters:
            - AddResponseHeader=X-Response-Foo, Bar
            - AddResponseHeader=abc-myname, colin
eureka:
    client:
        service-url:
            defaultZone: http://127.0.0.1:10086/eureka
        instance:
            prefer-ip-address:true

```

注意: 自定义过滤器的命名应为: `***GatewayFilterFactory`

测试访问: <http://localhost:10010/api/user/1?name=colin> 检查后台是否输出name和colin, 但是若访问<http://localhost:10010/api/user/1?name2=hehehe>则不会输出。

9.8.2 自定义全局过滤器

需求: 编写全局过滤器, 在过滤器中检查请求中是否携带token请求头。如果token请求头存在则放行; 如果token为空或者不存在则设置返回的状态码为: 未授权也不再执行下去。

在colin-gateway工程编写全局过滤器类MyGlobalFilter

```

package com.colin.gateway.filter;

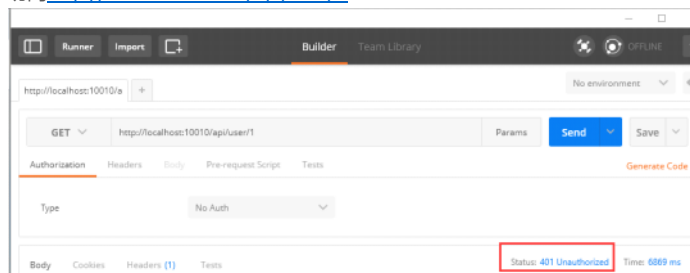
import org.apache.commons.lang.StringUtils;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.core.Ordered;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

/**
 * Created with IntelliJ IDEA.
 * Description:
 * User: DengLong
 * Date: 2021-04-15
 * Time: 14:10
 */
@Component
public class MyGlobalFilter implements GlobalFilter, Ordered {
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        System.out.println("全局过滤器");
        String token = exchange.getRequest().getHeaders().getFirst("token");
        if (StringUtils.isBlank(token)) {
            exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();
        }
        return null;
    }

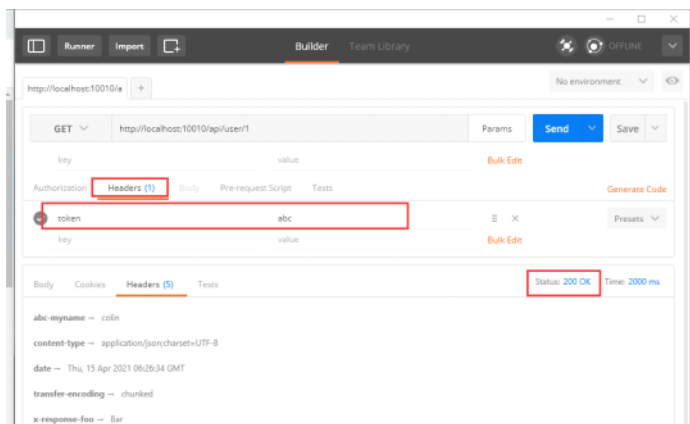
    @Override
    public int getOrder() {
        // 值越小, 越先执行
        return 1;
    }
}

```

访问<http://localhost:10010/api/user/1>



访问<http://localhost:10010/api/user/1> 带上header参数token



9.9 负载均衡和熔断(了解)

Gateway中默认已经集成了Ribbon负载均衡和Hystrix熔断机制。但是所有的超时策略都是走的默认值，比如熔断超时时间只有1s，很容易就触发了。因此建议手动进行配置：

```
server:
  port: 10010
spring:
  application:
    name: api-gateway
  cloud:
    gateway:
      routes:
        # 路由id, 可以随意写
        - id: user-service-route
          # 代理的服务地址
          uri: http://127.0.0.1:9091
          # 路由断言, 可以配置映射路径
          predicates:
            - Path=/api/user/**
          filters:
            # 添加请求路径的前缀
            - PrefixPath=/user
            # 1表示过滤一个路径, 2表示两个路径, 以此类推
            - StripPrefix=1
            # 自定义过滤器
            - MyParam=name
          # 默认过滤器, 对所有路由都生效
          default-filters:
            - AddResponseHeader=X-Response-Foo, Bar
            - AddResponseHeader=abc-myname, colin
      eureka:
        client:
          service-url:
            defaultZone: http://127.0.0.1:10086/eureka
        instance:
          prefer-ip-address: true

      hystrix:
        command:
          default:
            execution:
              isolation:
                thread:
                  timeoutInMilliseconds: 6000 #服务降级超时时间, 默认1s

      ribbon:
        ConnectTimeout: 1000 # 连接超时时长
        ReadTimeout: 2000 # 数据通信超时时长
        MaxAutoRetries: 0 #当前服务器的重试次数
        MaxAutoRetriesNextServer: 0 #重试多少次服务
```

9.10 Gateway跨域配置

一般网关都是所有微服务的统一入口，必然在被调用的时候会出现跨域问题。

跨域：在js请求访问中，如果访问的地址与当前服务器的域名、ip或者端口号不一致则称为跨域请求。若不解决则不能获取到对应地址的返回结果。

如：从在http://localhost:9090中的js访问http://localhost:9000的数据，因为端口不同，所以也是跨域请求。

在访问Spring Cloud Gateway网关服务器的时候，出现跨域问题的话，可以在网关服务器中通过配置解决，允许哪些服务是可以跨域请求的。具体配置如下：

```
server:
  port: 10010
```

```

spring:
  application:
    name: api-gateway
  cloud:
    gateway:
      routes:
        # 路由id, 可以随意写
        - id: user-service-route
          # 代理的服务地址
          uri: http://127.0.0.1:9091
          uri: lb://user-service
          # 路由断言, 可以配置映射路径
          predicates:
            - Path=/api/user/**
          filters:
            # 添加请求路径的前缀
            - PrefixPath=/user
            # 1表示过滤一个路径, 2表示两个路径, 以此类推
            - StripPrefix=1
            # 自定义过滤器
            - MyParam=name
          # 默认过滤器, 对所有路由都生效
          default-filters:
            - AddResponseHeader=X-Response-Foo, Bar
            - AddResponseHeader=abc-myname, colin
      globalcors:
        corsConfigurations:
          '[/**]':
            #allowedOrigins: * # 这种写法或者下面的都可以, *表示全部
            allowedOrigins:
              - "http://docs.spring.io"
            allowedMethods:
              - GET

eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:10086/eureka
    instance:
      prefer-ip-address: true

hystrix:
  command:
    default:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 6000 #服务降级超时时间, 默认1S

ribbon:
  ConnectTimeout: 1000 # 连接超时时长
  ReadTimeout: 2000 # 数据通信超时时长
  MaxAutoRetries: 0 #当前服务器的重试次数
  MaxAutoRetriesNextServer: 0 #重试多少次服务

```

上述配置表示: 可以允许来自<http://docs.spring.io>的get请求方式获取服务数据。

allowedOrigins指定运行访问的服务器地址, 如: <http://localhost:10000>也是可以的。

'[/**]'表示对所有访问到网关服务器的请求地址

官网具体说明: https://cloud.spring.io/spring-cloud-static/spring-cloudgateway/2.1.1.RELEASE/multi/multi__cors_configuration.html

9.11 Gateway的高可用(了解)

启动多个Gateway服务, 自动注册到Eureka, 形成集群。如果是服务内部访问, 访问Gateway, 自动负载均衡, 没问题。

但是, Gateway更多的外部访问, PC端、移动端等。它们无法通过Eureka进行负载均衡, 那么该怎么办? 此时, 可以使用其它的服务网关, 来对Gateway进行代理。比如: Nginx。

9.12 Gateway和Feign的区别

Gateway作为整个应用的流量入口, 接收所有的请求, 如PC、移动端等, 并且将不同的请求转发至不同的处理微服务模块, 其作用可视为nginx, 大部分情况下用作权限鉴定、服务端流量控制。

Feign则是将当前微服务的部分服务接口暴露出来, 并且主要用于各个微服务之间的服务调用。