尚硅谷2020最新版周阳SpringCloud(H版&alibaba)框架开发教程 学习笔记

原创甲如\_0314 最后发布于2020-03-05 21:04:14 阅读数 3651 收藏

展开

前言：今天看到周阳老师出了新课，十分欣喜，很喜欢周阳老师的讲课风格，内容也充实，我也算是周阳老师忠实粉丝啦。

新出的springcloud第二版很符合我现阶段的学习需求。但美中不足的是，目前只有视频资料，并没有周阳老师上课的脑图，一点一点跟着老师构建项目，稍有吃力。将目前学习的笔记整理，供自己日后复习，供大家参考。若有不足，还请指正。

周阳微服务架构与SpringCloud

源码已分享在个人github： https://github.com/Jiaru0314/cloud2020 不断更新，供参考，觉得有用的化给个star吧哈哈哈。

SpringCloud=分布式微服务架构的一站式解决方案，是多种微服务架构落地技术的集合体，俗称微服务全家桶

SpringBoot是一种服务开发技术

服务注册与发现：EUREKA

服务负载均衡与调用：NETFLIX OSS RIBBON

服务负载与调用：NETTFLIX

服务熔断降级：HYSTRIX

服务网关：Zuul

服务分布式配置：SpringCloud Config

服务开发：SpingBoot

SpringBoot 2.0版和SpringCloud H版 强烈建议使用SpringBoot 2.0以上

SpringBoot和SpringCloud之间版本有约束 H版对应2.2 G版对应2.1

课程版本约束

cloud:Hoxton.SR1

boot:2.2.2.RELEASE

cloud alibaba:2.1.0.RELEASE

java：java8

Maven 3.5以上

Mysql:5.7以上

1、新建maven项目：cloud2020

项目结构目录如下：

pom.xml如下

注意：周阳老师使用的是mysql5,我用的是mysql8

根据自己的需要改变 <mysql.version></mysql.version>

<?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>org.xzq.springcloud</groupId>

<artifactId>cloud2020</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>pom</packaging>

<modules>

<module>cloud-provider-payment8001</module>

<module>cloud-consumer-order80</module>

<module>cloud-api-commons</module>

<module>cloud-eureka-server7001</module>

<module>cloud-eureka-server7002</module>

<module>cloud-provider-payment8002</module>

<module>cloud-provider-payment8004</module>

<module>cloud-comsumerzk-order80</module>

<module>cloud-providerconsul-payment8006</module>

<module>cloud-consumerconsul-order80</module>

<module>cloud-consumer-feign-order80</module>

</modules>

<!--统一管理jar包和版本-->

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

<junit.version>4.12</junit.version>

<log4j.version>1.2.17</log4j.version>

<lombok.version>1.16.18</lombok.version>

<mysql.version>8.0.18</mysql.version>

<druid.verison>1.1.16</druid.verison>

<mybatis.spring.boot.verison>1.3.0</mybatis.spring.boot.verison>

</properties>

<dependencyManagement>

<dependencies>

<!--spring boot 2.2.2-->

<dependency>

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

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

<version>2.2.2.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<!--spring cloud Hoxton.SR1-->

<dependency>

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

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

<version>Hoxton.SR1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<!--spring cloud alibaba 2.1.0.RELEASE-->

<dependency>

<groupId>com.alibaba.cloud</groupId>

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

<version>2.2.0.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<!-- MySql -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>${mysql.version}</version>

</dependency>

<!-- Druid -->

<dependency>

<groupId>com.alibaba</groupId>

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

<version>${druid.verison}</version>

</dependency>

<!-- mybatis-springboot整合 -->

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

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

<version>${mybatis.spring.boot.verison}</version>

</dependency>

<!--lombok-->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>${lombok.version}</version>

</dependency>

<!--junit-->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>${junit.version}</version>

</dependency>

<!-- log4j -->

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>${log4j.version}</version>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<fork>true</fork>

<addResources>true</addResources>

</configuration>

</plugin>

</plugins>

</build>

</project>

2、创建数据库

CREATE DATABASE /\*!32312 IF NOT EXISTS\*/`cloud` /\*!40100 DEFAULT CHARACTER SET utf8 COLLATE utf8\_unicode\_ci \*/;

​

USE `cloud`;

​

/\*Table structure for table `payment` \*/

​

DROP TABLE IF EXISTS `payment`;

​

CREATE TABLE `payment` (

`id` bigint(20) NOT NULL AUTO\_INCREMENT COMMENT 'ID',

`serial` varchar(200) COLLATE utf8\_unicode\_ci DEFAULT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=5 DEFAULT CHARSET=utf8 COLLATE=utf8\_unicode\_ci;

​

/\*Data for the table `payment` \*/

​

insert  into `payment`(`id`,`serial`) values (1,'尚硅谷'),(2,'alibaba'),(3,'京东'),(4,'头条');

3、resource目录下新建 application.yml

注意： mysql8的数据库驱动包是 com.mysql.cj.jdbc.Driver

mysql5是 com.mysql.jdbc.Driver

server:

port: 8001 #服务端口

​

spring:

application:

  name: cloud-payment-service #服务名

datasource:

  type: com.alibaba.druid.pool.DruidDataSource #当前数据源操作类型

  driver-class-name: com.mysql.cj.jdbc.Driver #数据库驱动包

  url: jdbc:mysql://localhost:3306/cloud?characterEncoding=utf8&useSSL=false&serverTimezone=UTC&rewriteBatchedStatements=true

  username: root

  password: 1234

​

devtools:

  restart:

    enabled: true #是否支持热部署

​

mybatis:

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

type-aliases-package: com.xzq.springcloud.entities #所有entity别名所在包

​

4、新建启动类PaymentMain8001

/\*\*

\* @ClassName: PaymentMain8001

\* @description: PaymentMain8001主启动类

\* @author: XZQ

\* @create: 2020/3/5 17:30

\*\*/

@SpringBootApplication

public class PaymentMain8001 {

   public static void main(String[] args) {

       SpringApplication.run(PaymentMain8001.class, args);

  }

}

​

5、dao层开发

新建PaymentDao接口

/\*\*

\* @ClassName: PaymentDao

\* @description: 支付持久层

\* @author: XZQ

\* @create: 2020/3/5 17:58

\*\*/

@Mapper

public interface PaymentDao {

   int create(Payment payment);

​

   Payment getPaymentById(@Param("id") Long id);

}

mapper.xml

resource下创建mapper文件夹，新建PaymentMapper.xml

<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >

<mapper namespace="com.xzq.springcloud.dao.PaymentDao">

   <resultMap id="BaseResultMap" type="com.xzq.springcloud.entities.Payment">

       <id column="id" property="id" jdbcType="BIGINT"/>

       <id column="serial" property="serial" jdbcType="VARCHAR"/>

   </resultMap>

​

   <insert id="create" parameterType="payment" useGeneratedKeys="true" keyProperty="id">

          insert into payment(serial) values (#{serial})

   </insert>

​

   <select id="getPaymentById" parameterType="Long" resultMap="BaseResultMap">

      select \* from payment where id = #{id}

   </select>

</mapper>

6、service层

service接口

package com.xzq.springcloud.service;

​

import com.xzq.springcloud.entities.Payment;

import org.apache.ibatis.annotations.Param;

​

/\*\*

\* @InterfaceName: PaymentService

\* @description:

\* @author: XZQ

\* @create: 2020/3/5 18:18

\*\*/

public interface PaymentService {

   int create(Payment payment);

​

   Payment getPaymentById(@Param("id") Long id);

}

​

service实现类

package com.xzq.springcloud.service.impl;

​

import com.xzq.springcloud.dao.PaymentDao;

import com.xzq.springcloud.entities.Payment;

import com.xzq.springcloud.service.PaymentService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

​

/\*\*

\* @ClassName: PaymentServiceImpl

\* @description:

\* @author: XZQ

\* @create: 2020/3/5 18:19

\*\*/

@Service

public class PaymentServiceImpl implements PaymentService {

​

   @Autowired

   private PaymentDao paymentDao;

​

   @Override

   public int create(Payment payment) {

       return paymentDao.create(payment);

  }

​

   @Override

   public Payment getPaymentById(Long id) {

       return paymentDao.getPaymentById(id);

  }

}

​

7、controller层

package com.xzq.springcloud.controller;

​

import com.xzq.springcloud.entities.CommonResult;

import com.xzq.springcloud.entities.Payment;

import com.xzq.springcloud.service.PaymentService;

import lombok.extern.slf4j.Slf4j;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.\*;

​

/\*\*

\* @ClassName: PaymentController

\* @description:

\* @author: XZQ

\* @create: 2020/3/5 19:03

\*\*/

@RestController

@Slf4j

@RequestMapping("/payment")

public class PaymentController {

   @Autowired

   private PaymentService paymentService;

​

   @PostMapping("/create")

   public CommonResult create(Payment payment) {

       int result = paymentService.create(payment);

       log.info("插入数据的ID:\t" + payment.getId());

       log.info("插入结果：" + result);

       if (result > 0) {

           return new CommonResult(200, "插入数据成功", result);

      } else {

           return new CommonResult(444, "插入数据失败", null);

      }

  }

​

   @GetMapping("/get/{id}")

   public CommonResult getPaymentById(@PathVariable("id") Long id) {

       Payment payment = paymentService.getPaymentById(id);

       log.info("\*\*\*查询结果：" + payment);

       if (payment != null) {

           return new CommonResult(200, "查询数据成功", payment);

      } else {

           return new CommonResult(444, "没有对应记录", null);

      }

  }

​

​

}

​

8、测试

1、get测试：浏览器输入：http://localhost:8001/payment/get/2

结果：{"code":200,"message":"查询数据成功","data":{"id":2,"serial":"alibaba"}}

2、post测试：

二、cloud-consumer-order80

1、pom文件中添加依赖

<dependencies>

   <dependency>

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

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

   </dependency>

   <dependency>

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

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

   </dependency>

   <dependency>

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

       <artifactId>spring-boot-devtools</artifactId>

       <scope>runtime</scope>

       <optional>true</optional>

   </dependency>

   <dependency>

       <groupId>org.projectlombok</groupId>

       <artifactId>lombok</artifactId>

       <optional>true</optional>

   </dependency>

   <dependency>

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

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

   </dependency>

</dependencies>

2、application.yml

server:

port: 80

3、启动类（同上个模块省略）

4、拷贝实体类、

5、RestTemplate

RestTemplate提供了多种便捷访问远程Http服务的方法，

是一种简单便捷的访问restful服务的模板类，是spring提供的用于访问Rest服务的客户端模板工具集。

配置类

package com.xzq.springcloud.config;

​

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.client.RestTemplate;

​

/\*\*

\* @ClassName: ApplicationContextConfig

\* @description: 配置类

\* @author: XZQ

\* @create: 2020/3/5 21:25

\*\*/

@Configuration

public class ApplicationContextConfig {

​

   @Bean

   public RestTemplate getRestTemplate() {

       return new RestTemplate();

  }

}

​

6、controller层

package com.xzq.springcloud.controller;

​

import com.xzq.springcloud.entities.CommonResult;

import com.xzq.springcloud.entities.Payment;

import lombok.extern.slf4j.Slf4j;

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.RestController;

import org.springframework.web.client.RestTemplate;

​

/\*\*

\* @ClassName: OrderController

\* @description:

\* @author: XZQ

\* @create: 2020/3/5 21:18

\*\*/

@RestController

@Slf4j

public class OrderController {

​

   private final static String PAYMENT\_URL = "http://localhost:8001";

​

   @Autowired

   private RestTemplate restTemplate;

​

   @GetMapping("/consumer/payment/get/{id}")

   public CommonResult<Payment> getPaymentById(@PathVariable("id") Long id) {

       return restTemplate.getForObject(PAYMENT\_URL + "/payment/get/" + id, CommonResult.class, id);

  }

​

   @GetMapping("/consumer/payment/create")

   public CommonResult<Payment> create(Payment payment) {

       return restTemplate.postForObject(PAYMENT\_URL + "/payment/create", payment, CommonResult.class);

  }

}

​

7、rundashbroad

运用spring cloud框架基于spring boot构建微服务，一般需要启动多个应用程序，在idea开发工具中，多个同时启动的应用

需要在RunDashboard运行仪表盘中可以更好的管理，但有时候idea中的RunDashboard窗口没有显示出来，也找不到直接的开启按钮

idea中打开Run Dashboard的方法如下

　　　　view > Tool Windows > Run Dashboard

如果上述列表找不到Run Dashboard,则可以在工程目录下找到.idea文件夹下的workspace.xml，在其中相应位置加入以下代码（替换）即可：

<component name="RunDashboard">

<option name="configurationTypes">

 <set>

   <option value="SpringBootApplicationConfigurationType"/>

 </set>

</option>

<option name="ruleStates">

 <list>

   <RuleState>

     <option name="name" value="ConfigurationTypeDashboardGroupingRule"/>

   </RuleState>

   <RuleState>

     <option name="name" value="StatusDashboardGroupingRule"/>

   </RuleState>

 </list>

</option>

</component>

关闭重启后出现。

三、工程重构

1、新建模块 cloud-api-commons

依赖：

<dependencies>

   <dependency>

       <groupId>org.projectlombok</groupId>

       <artifactId>lombok</artifactId>

       <optional>true</optional>

   </dependency>

   <dependency>

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

       <artifactId>spring-boot-devtools</artifactId>

       <scope>runtime</scope>

       <optional>true</optional>

   </dependency>

   <dependency>

       <groupId>cn.hutool</groupId>

       <artifactId>hutool-all</artifactId>

       <version>5.1.0</version>

   </dependency>

</dependencies>

2、将消费者和服者种的entities拷贝至新模块中，删除原来的entities包，clean、install cloud-api-commons 模块，在消费则服务者pom.xml中分别引入依赖，测试运行。

四、Eureka

1、Eureka是什么

Eureka 是 Netflix 开发的，一个基于 REST 服务的，服务注册与发现的组件，以实现中间层服务器的负载平衡和故障转移。

它主要包括两个组件：Eureka Server 和 Eureka Client

Eureka Client：一个Java客户端，用于简化与 Eureka Server 的交互（通常就是微服务中的客户端和服务端）

Eureka Server：提供服务注册和发现的能力（通常就是微服务中的注册中心）

服务在Eureka上注册，然后每隔30秒发送心跳来更新它们的租约。如果客户端不能多次续订租约，那么它将在大约90秒内从服务器注册表中剔除。注册信息和更新被复制到集群中的所有eureka节点。来自任何区域的客户端都可以查找注册表信息（每30秒发生一次）来定位它们的服务（可能在任何区域）并进行远程调用

2、 Eureka 客户端与服务器之间的通信

服务发现有两种模式：一种是客户端发现模式，一种是服务端发现模式。Eureka采用的是客户端发现模式。

2.1. Register（注册）

Eureka客户端将关于运行实例的信息注册到Eureka服务器。注册发生在第一次心跳。

2.2. Renew（更新 / 续借）

Eureka客户端需要更新最新注册信息（续借），通过每30秒发送一次心跳。更新通知是为了告诉Eureka服务器实例仍然存活。如果服务器在90秒内没有看到更新，它会将实例从注册表中删除。建议不要更改更新间隔，因为服务器使用该信息来确定客户机与服务器之间的通信是否存在广泛传播的问题。

2.3. Fetch Registry（抓取注册信息）

Eureka客户端从服务器获取注册表信息并在本地缓存。之后，客户端使用这些信息来查找其他服务。通过在上一个获取周期和当前获取周期之间获取增量更新，这些信息会定期更新(每30秒更新一次)。获取的时候可能返回相同的实例。Eureka客户端自动处理重复信息。

2.4. Cancel（取消）

Eureka客户端在关机时向Eureka服务器发送一个取消请求。这将从服务器的实例注册表中删除实例，从而有效地将实例从流量中取出。

3、Eureka自我保护模式

如果 Eureka 服务器检测到超过预期数量的注册客户端以一种不优雅的方式终止了连接，并且同时正在等待被驱逐，那么它们将进入自我保护模式。这样做是为了确保灾难性网络事件不会擦除eureka注册表数据，并将其向下传播到所有客户端。

任何客户端，如果连续3次心跳更新失败，那么它将被视为非正常终止，病句将被剔除。当超过当前注册实例15%的客户端都处于这种状态，那么自我保护将被开启。

当自我保护开启以后，eureka服务器将停止剔除所有实例，直到：

它看到的心跳续借的数量回到了预期的阈值之上，或者

自我保护被禁用

默认情况下，自我保护是启用的，并且，默认的阈值是要大于当前注册数量的15%

4、Eureka VS Zookeeper

4.1. Eureka保证AP

Eureka服务器节点之间是对等的，只要有一个节点在，就可以正常提供服务。

Eureka客户端的所有操作可能需要一段时间才能在Eureka服务器中反映出来，随后在其他Eureka客户端中反映出来。也就是说，客户端获取到的注册信息可能不是最新的，它并不保证强一致性

4.2. Zookeeper保证CP

Zookeeper集群中有一个Leader，多个Follower。Leader负责写，Follower负责读，ZK客户端连接到任何一个节点都是一样的，写操作完成以后要同步给所有Follower以后才会返回。如果Leader挂了，那么重新选出新的Leader，在此期间服务不可用。

4.3. 为什么用Eureka

分布式系统大都可以归结为两个问题：数据一致性和防止单点故障。而作为注册中心的话，即使在一段时间内不一致，也不会有太大影响，所以在A和C之间选择A是比较适合该场景的。

工程实现步骤

1、新建cloud-eureka-server7001模块

pom.xml中加入依赖：

<dependencies>

   <!--eureka-server-->

   <dependency>

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

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

   </dependency>

   <!--自定义api通用包-->

   <dependency>

       <groupId>org.xzq.springcloud</groupId>

       <artifactId>cloud-api-commons</artifactId>

       <version>${project.version}</version>

   </dependency>

   <!--boot web acctuator-->

   <dependency>

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

       <artifactId>spring-boot</artifactId>

   </dependency>

   <dependency>

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

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

   </dependency>

   <dependency>

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

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

   </dependency>

   <dependency>

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

       <artifactId>spring-boot-devtools</artifactId>

       <scope>runtime</scope>

       <optional>true</optional>

   </dependency>

</dependencies>

2、resources目录下新建application.yml

server:

port: 7001

​

eureka:

instance:

  hostname: localhsot #eureka服务端实例名称

client:

  register-with-eureka: false #表示不像注册中心注册自己

  fetch-registry: false #false表示自己就是注册中心，我的职责就是维护服务实例,并不区检索服务

  service-url:

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

3、编写启动类EurekaApplication7001

package com.xzq.springcloud;

​

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

​

/\*\*

\* @ClassName: EurekaApplicatin7001

\* @description:

\* @author: XZQ

\* @create: 2020/3/5 23:19

\*\*/

@SpringBootApplication

@EnableEurekaServer//开启EurekaServer

public class EurekaApplicatin7001 {

   public static void main(String[] args) {

       SpringApplication.run(EurekaApplicatin7001.class, args);

  }

}

​

2、支付微服务8001入驻7001

​ 1、pom.xml添加

<!--eureka-server-->

<dependency>

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

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

</dependency>

​ 2、application.yml

eureka:

client:

  register-with-eureka: true #表示向注册中心注册自己 默认为true

  fetch-registry: true #是否从EurekaServer抓取已有的注册信息，默认为true,单节点无所谓,集群必须设置为true才能配合ribbon使用负载均衡

  service-url:

    defaultZone: http://localhost:7001/eureka/ # 入驻地址

​ 3、启动类加上注解@EnableEurekaClient

3、订单微服务80入驻7001

​ 同上

五、Eureka集群

原理说明：

服务注册：将服务信息注册到注册中心

服务发现：从注册中心获取服务信息

实质：存key服务名，取value调用地址

步骤：

先启动eureka注册中心

启动服务提供者payment支付服务

支付服务启动后，会把自身信息注册到eureka

消费者order服务在需要调用接口时，使用服务别名去注册中心获取实际的远程调用地址

消费者获得调用地址后，底层实际是调用httpclient技术实现远程调用

消费者获得服务地址后会缓存在本地jvm中，默认每30秒更新异常服务调用地址

问题：微服务RPC远程调用最核心的是说明？

高可用，如果注册中心只有一个，出现故障就麻烦了。会导致整个服务环境不可用。

解决办法：搭建eureka注册中心集群，实现负载均衡+故障容错

互相注册，相互守望

集群搭建步骤

1、依照7001新建7002，除了主启动类和yml配置文件按，其他都一样

2、修改C:\Windows\System32\drivers\etc下的hosts

末尾加上

# springcloud2020

127.0.0.1 eureka7001.com

127.0.0.1 eureka7002.com

127.0.0.1 eureka7003.com

3、修改7001项目 applicaton.yml

server:

port: 7001

​

eureka:

instance:

  hostname: eureka7001.com #eureka服务端实例名称

client:

  register-with-eureka: false #表示不向注册中心注册自己

  fetch-registry: false #false表示自己就是注册中心，我的职责就是维护服务实例,并不区检索服务

  service-url:

    defaultZone: http://eureka7002.com:7002/eureka/

4、修改7002

server:

port: 7002

​

eureka:

instance:

  hostname: eureka7002.com #eureka服务端实例名称

client:

  register-with-eureka: false #表示不向注册中心注册自己

  fetch-registry: false #false表示自己就是注册中心，我的职责就是维护服务实例,并不区检索服务

  service-url:

    defaultZone: http://eureka7001.com:7001/eureka/

5、eurekaserver集群效果

6、支付和订单两个微服务注册到eureka集群

修改80项目yml配置文件

  service-url:

    #defaultZone: http://localhost:7001/eureka/ # 入驻地址

    defaultZone: http://eureka7001.com:7001/eureka/, http://eureka7001.com:7001/eureka/

8001同理

启动7001、7002再启动8001、80项目查看效果

六、搭建支付服务集群

参照8001搭建8002服务

访问localhost:8001/payment/get/1

结果：{"code":200,"message":"查询数据成功,serverport:8001","data":{"id":1,"serial":"尚硅谷"}}

访问localhost:8002/payment/get/1

结果：{"code":200,"message":"查询数据成功,serverport:8002","data":{"id":1,"serial":"尚硅谷"}}

成功

访问localhost/consumer/payment/get/1

结果：{"code":200,"message":"查询数据成功,serverport:8001","data":{"id":1,"serial":"尚硅谷"}}

但是会发现 ，每次范文都是8001端口

原因是再80项目的controller层中，我们将请求路径写死了。

修改如下：

public class OrderController {

​

   //   private final static String PAYMENT\_URL = "http://localhost:8001";

   private final static String PAYMENT\_URL = "http://CLOUD-PAYMENT-SERVICE";

但是访问localhost/consumer/payment/get/1会出现500错误

报错信息如下：

There was an unexpected error (type=Internal Server Error, status=500).

I/O error on GET request for "http://CLOUD-PAYMENT-SERVICE/payment/get/1": CLOUD-PAYMENT-SERVICE; nested exception is java.net.UnknownHostException: CLOUD-PAYMENT-SERVICE

org.springframework.web.client.ResourceAccessException: I/O error on GET request for "http://CLOUD-PAYMENT-SERVICE/payment/get/1": CLOUD-PAYMENT-SERVICE; nested exception is java.net.UnknownHostException: CLOUD-PAYMENT-SERVICE

原因是，我们配置了以服务名的方式访问，但不能确定是哪一个服务。

我们需要给restTemplate开启负载均衡，默认是轮循。

/\*\*

\* @ClassName: ApplicationContextConfig

\* @description: 配置类

\* @author: XZQ

\* @create: 2020/3/5 21:25

\*\*/

@Configuration

public class ApplicationContextConfig {

​

   @Bean

   @LoadBalanced//开启负载均衡

   public RestTemplate getRestTemplate() {

       return new RestTemplate();

  }

}

​

我们看一下@LoadBalanced源码

/\*\*

\* Annotation to mark a RestTemplate bean to be configured to use a LoadBalancerClient

\* @author Spencer Gibb

\*/

@Target({ ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD })

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@Qualifier

public @interface LoadBalanced {

}

​

可以看到 @LoadBalanced是一个注解，用于标记要配置为使用LoadBalancerClient的RestTemplate bean的。

那么我们再来看看 LoadBalancerClient

public interface ServiceInstanceChooser {

   ServiceInstance choose(String serviceId);

}

public interface LoadBalancerClient extends ServiceInstanceChooser {

​

   <T> T execute(String serviceId, LoadBalancerRequest<T> request) throws IOException;

​

   <T> T execute(String serviceId, ServiceInstance serviceInstance, LoadBalancerRequest<T> request) throws IOException;

​

   URI reconstructURI(ServiceInstance instance, URI original);

}

LoadBalancerClient是一个接口，该接口中有四个方法，我们来大概看一下这几个方法的作用：

ServiceInstance choose(String serviceId)根据传入的服务名serviceId从客户端负载均衡器中挑选一个对应服务的实例。

T execute() ,使用从负载均衡器中挑选出来的服务实例来执行请求。

URI reconstructURI(ServiceInstance instance, URI original)表示为系统构建一个合适的URI，我们在Spring Cloud中服务的发现与消费一文中发送请求时使用了服务的逻辑名称(http://HELLO-SERVICE/hello)而不是具体的服务地址，在reconstructURI方法中，第一个参数ServiceInstance实例是一个带有host和port的具体服务实例，第二个参数URI则是使用逻辑服务名定义为host和port的URI，而返回的URI则是通过ServiceInstance的服务实例详情拼接出的具体的host:port形式的请求地址。一言以蔽之，就是把类似于http://HELLO-SERVICE/hello这种地址转为类似于http://195.124.207.128/hello地址（IP地址也可能是域名）。

六、搭建支付服务集群

参照8001搭建8002服务

访问localhost:8001/payment/get/1

结果：{"code":200,"message":"查询数据成功,serverport:8001","data":{"id":1,"serial":"尚硅谷"}}

访问localhost:8002/payment/get/1

结果：{"code":200,"message":"查询数据成功,serverport:8002","data":{"id":1,"serial":"尚硅谷"}}

成功

访问localhost/consumer/payment/get/1

结果：{"code":200,"message":"查询数据成功,serverport:8001","data":{"id":1,"serial":"尚硅谷"}}

但是会发现 ，每次范文都是8001端口

原因是再80项目的controller层中，我们将请求路径写死了。

修改如下：

public class OrderController {

​

   //   private final static String PAYMENT\_URL = "http://localhost:8001";

   private final static String PAYMENT\_URL = "http://CLOUD-PAYMENT-SERVICE";

但是访问localhost/consumer/payment/get/1会出现500错误

报错信息如下：

There was an unexpected error (type=Internal Server Error, status=500).

I/O error on GET request for "http://CLOUD-PAYMENT-SERVICE/payment/get/1": CLOUD-PAYMENT-SERVICE; nested exception is java.net.UnknownHostException: CLOUD-PAYMENT-SERVICE

org.springframework.web.client.ResourceAccessException: I/O error on GET request for "http://CLOUD-PAYMENT-SERVICE/payment/get/1": CLOUD-PAYMENT-SERVICE; nested exception is java.net.UnknownHostException: CLOUD-PAYMENT-SERVICE

原因是，我们配置了以服务名的方式访问，但不能确定是哪一个服务。

我们需要给restTemplate开启负载均衡，默认是轮循。

/\*\*

\* @ClassName: ApplicationContextConfig

\* @description: 配置类

\* @author: XZQ

\* @create: 2020/3/5 21:25

\*\*/

@Configuration

public class ApplicationContextConfig {

​

   @Bean

   @LoadBalanced//开启负载均衡

   public RestTemplate getRestTemplate() {

       return new RestTemplate();

  }

}

​

我们看一下@LoadBalanced源码

/\*\*

\* Annotation to mark a RestTemplate bean to be configured to use a LoadBalancerClient

\* @author Spencer Gibb

\*/

@Target({ ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD })

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@Qualifier

public @interface LoadBalanced {

}

​

可以看到 @LoadBalanced是一个注解，用于标记要配置为使用LoadBalancerClient的RestTemplate bean的。

那么我们再来看看 LoadBalancerClient

public interface ServiceInstanceChooser {

   ServiceInstance choose(String serviceId);

}

public interface LoadBalancerClient extends ServiceInstanceChooser {

​

   <T> T execute(String serviceId, LoadBalancerRequest<T> request) throws IOException;

​

   <T> T execute(String serviceId, ServiceInstance serviceInstance, LoadBalancerRequest<T> request) throws IOException;

​

   URI reconstructURI(ServiceInstance instance, URI original);

}

LoadBalancerClient是一个接口，该接口中有四个方法，我们来大概看一下这几个方法的作用：

ServiceInstance choose(String serviceId)根据传入的服务名serviceId从客户端负载均衡器中挑选一个对应服务的实例。

T execute() ,使用从负载均衡器中挑选出来的服务实例来执行请求。

URI reconstructURI(ServiceInstance instance, URI original)表示为系统构建一个合适的URI，我们在Spring Cloud中服务的发现与消费一文中发送请求时使用了服务的逻辑名称(http://HELLO-SERVICE/hello)而不是具体的服务地址，在reconstructURI方法中，第一个参数ServiceInstance实例是一个带有host和port的具体服务实例，第二个参数URI则是使用逻辑服务名定义为host和port的URI，而返回的URI则是通过ServiceInstance的服务实例详情拼接出的具体的host:port形式的请求地址。一言以蔽之，就是把类似于http://HELLO-SERVICE/hello这种地址转为类似于http://195.124.207.128/hello地址（IP地址也可能是域名）。

七、Eureka的自我保护机制

为什么会产生自我保护机制？

为防止EurekaClient可以正常运行，但是与EurekaServer网络不同的情况下，EurekaServer不会立刻将EurekaClient服务剔除。

什么是自我保护机制？

默认情况下，当Eureka server在一定时间内没有收到实例的心跳，便会把该实例从注册表中删除（默认是90秒），但是，如果短时间内丢失大量的实例心跳，便会触发eureka server的自我保护机制。

比如在开发测试时，需要频繁地重启微服务实例，但是我们很少会把eureka server一起重启（因为在开发过程中不会修改eureka注册中心），当一分钟内收到的心跳数大量减少时，会触发该保护机制。可以在eureka管理界面看到Renews threshold和Renews(last min)，当后者（最后一分钟收到的心跳数）小于前者（心跳阈值）的时候，触发保护机制，会出现红色的警告：

EMERGENCY!EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT.RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEGING EXPIRED JUST TO BE SAFE.

从警告中可以看到，eureka认为虽然收不到实例的心跳，但它认为实例还是健康的，eureka会保护这些实例，不会把它们从注册表中删掉。

在自我保护模式中，EurekaServer会保护服务注册表中的信息，不再注销任何服务实例。

综上，自我保护模式是一种应对网络异常的安全保护措施它的架构哲学是宁可同时保留所有微服务，也不忙保姆注销如何健康的微服务，使用自我保护模式，可以让Eureka集群更加健壮，稳定。

署于CAP 的AP分支。

如何禁止自我保护机制

服务提供者：

lease-renewal-interval-in-seconds: 1 # eureka客户端向服务端发送心跳的时间间隔 单位秒 默认30

lease-expiration-duration-in-seconds: 2 # eureka

注册中心配置：

server:

enable-self-preservation: false # 关闭自我保护机制 保证不可用服务及时清除

eviction-interval-timer-in-ms: 2000

springboot 整合zookeeper

1、pom.xml中添加依赖

注意 添加的zookeeper版本要与自己安装在服务器上的一致。

<dependencies>

   <dependency>

       <groupId>org.xzq.springcloud</groupId>

       <artifactId>cloud-api-commons</artifactId>

       <version>${project.version}</version>

   </dependency>

   <dependency>

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

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

   </dependency>

   <!--SpringBoot整合Zookeeper客户端-->

   <dependency>

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

       <artifactId>spring-cloud-starter-zookeeper-discovery</artifactId>

       <exclusions>

           <!--先排除自带的zookeeper3.5.3-->

           <exclusion>

               <groupId>org.apache.zookeeper</groupId>

               <artifactId>zookeeper</artifactId>

           </exclusion>

       </exclusions>

   </dependency>

   <!--添加zookeeper3.4.6版本 -->

   <dependency>

       <groupId>org.apache.zookeeper</groupId>

       <artifactId>zookeeper</artifactId>

       <version>3.4.6</version>

   </dependency>

   <dependency>

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

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

   </dependency>

   <dependency>

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

       <artifactId>spring-boot-devtools</artifactId>

       <scope>runtime</scope>

       <optional>true</optional>

   </dependency>

   <dependency>

       <groupId>org.projectlombok</groupId>

       <artifactId>lombok</artifactId>

       <optional>true</optional>

   </dependency>

   <dependency>

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

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

       <scope>test</scope>

   </dependency>

</dependencies>

2、配置yml文件

server:

port: 8004

spring:

application:

name: cloud-provider-payment

cloud:

zookeeper:

connect-string: 118.178.91.123:2181

3、编写控制层代码

/\*\*

\* @ClassName: PaymentController

\* @description:

\* @author: XZQ

\* @create: 2020/3/6 15:45

\*\*/

@RestController

@RequestMapping("/payment")

public class PaymentController {

​

   @Value("${server.port}")

   private String SERVER\_PORT;

​

   @RequestMapping("/zk")

   public String paymentZK() {

       return "springcloud with zookeeper :" + SERVER\_PORT + "\t" + UUID.randomUUID().toString();

  }

}

​

问题:zookeeper中的节点是持久还是临时的？

答：临时的。

九、consul

Consul是什么

Consul是一个服务网格（微服务间的 TCP/IP，负责服务之间的网络调用、限流、熔断和监控）解决方案，它是一个一个分布式的，高度可用的系统，而且开发使用都很简便。它提供了一个功能齐全的控制平面，主要特点是：服务发现、健康检查、键值存储、安全服务通信、多数据中心。

与其它分布式服务注册与发现的方案相比，Consul 的方案更“一站式”——内置了服务注册与发现框架、分布一致性协议实现、健康检查、Key/Value 存储、多数据中心方案，不再需要依赖其它工具。Consul 本身使用 go 语言开发，具有跨平台、运行高效等特点，也非常方便和 Docker 配合使用。

十、Eureka、Zookeeper、Consul三个注册中心的异同点

组件名 语言 健康检查 对外暴露接口 CAP Spring Cloud 集成

Eureka Java 可配支持 HTTP AP 集成

Consul Go 支持 HTTP/DFS CP 集成

Zookeeper java 支持 客户端 CP 集成

十一、Ribbon

Spring Cloud Ribbon是一个基于HTTP和TCP的客户端负载均衡工具，它基于Netflix Ribbon实现。通过Spring Cloud的封装，可以让我们轻松地将面向服务的REST模版请求自动转换成客户端负载均衡的服务调用。Spring Cloud Ribbon虽然只是一个工具类框架，它不像服务注册中心、配置中心、API网关那样需要独立部署，但是它几乎存在于每一个Spring Cloud构建的微服务和基础设施中。因为微服务间的调用，API网关的请求转发等内容，实际上都是通过Ribbon来实现的，包括后续我们将要介绍的Feign，它也是基于Ribbon实现的工具。所以，对Spring Cloud Ribbon的理解和使用，对于我们使用Spring Cloud来构建微服务非常重要。

实现负载均衡的算法。

负载规则替换，注意，不能与主启动类在同一个包下！

package com.xzq.myrule;

​

import com.netflix.loadbalancer.IRule;

import com.netflix.loadbalancer.RandomRule;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

​

/\*\*

\* @ClassName: MySelfRule

\*\*/

@Configuration

public class MySelfRule {

​

   @Bean

   public IRule myRule() {

       return new RandomRule();

  }

}

​

主启动类添加注释：

@RibbonClient(name = "CLOUD-PAYMENT-SERVICE", configuration = MySelfRule.class)

作用是替换ribbon负载均衡规则

负载均衡轮询算法 :rest接口第几次请求次数 % 服务器集群总数量 = 实际调用服务器位置下标，每次服务器重启后，rest接口计数从1开始。

ribbon源码

private int incrementAndGetModulo(int modulo) {

   int current;

   int next;

   do {

       current = this.nextServerCyclicCounter.get();

       next = (current + 1) % modulo;

  } while(!this.nextServerCyclicCounter.compareAndSet(current, next));

​

   return next;

}

手写一个负载的算法CAS+自旋锁

首先8001、8002服务controller层加上

@GetMapping("/payment/lb")

public String getPaymentLB() {

return SERVER\_PORT;

}

LoadBalancer接口：

package com.xzq.springcloud.lb;

​

import org.springframework.cloud.client.ServiceInstance;

​

import java.util.List;

​

/\*\*

\* @InterfaceName: LoadBalancer

\* @description:

\* @author: XZQ

\* @create: 2020/3/7 15:55

\*\*/

public interface LoadBalancer {

   ServiceInstance instances(List<ServiceInstance> serviceInstances);

}

​

实现类：

package com.xzq.springcloud.lb;

​

import org.springframework.cloud.client.ServiceInstance;

import org.springframework.stereotype.Component;

​

import java.sql.SQLOutput;

import java.util.List;

import java.util.concurrent.atomic.AtomicInteger;

​

/\*\*

\* @ClassName: MyLB

\* @description:

\* @author: XZQ

\* @create: 2020/3/7 15:55

\*\*/

@Component

public class MyLB implements LoadBalancer {

​

   private AtomicInteger atomicInteger = new AtomicInteger(0);

​

   private final int getAndIncrement() {

       int current;

       int next;

​

       do {

           current = this.atomicInteger.get();

           next = current >= Integer.MAX\_VALUE ? 0 : current + 1;

      } while (!atomicInteger.compareAndSet(current, next));

       System.out.println("第几次访问,次数next:" + next);

       return next;

  }

​

   @Override

   public ServiceInstance instances(List<ServiceInstance> serviceInstances) {

       int index = getAndIncrement() % serviceInstances.size();

       return serviceInstances.get(index);

  }

}

​

controller类中添加：

@GetMapping("/consumer/payment/lb")

public String getPaymentLB() {

   List<ServiceInstance> instances = discoveryClient.getInstances("CLOUD-PAYMENT-SERVICE");

   if (instances == null || instances.size() <= 0) {

       return null;

  }

​

   ServiceInstance serviceInstance = loadBalancer.instances(instances);

   URI uri = serviceInstance.getUri();

​

   return restTemplate.getForObject(uri + "/payment/lb", String.class);

}

十二、OpenFeign

OpenFeign是什么？

Feign是一个声明式的Web Service客户端。它的出现使开发Web Service客户端变得很简单。使用Feign只需要创建一个接口加上对应的注解，比如：FeignClient注解。Feign有可插拔的注解，包括Feign注解和JAX-RS注解。Feign也支持编码器和解码器，Spring Cloud Open Feign对Feign进行增强支持Spring MVC注解，可以像Spring Web一样使用HttpMessageConverters等。

Feign是一种声明式、模板化的HTTP客户端。在Spring Cloud中使用Feign，可以做到使用HTTP请求访问远程服务，就像调用本地方法一样的，开发者完全感知不到这是在调用远程方法，更感知不到在访问HTTP请求。

功能可插拔的注解支持，包括Feign注解和JAX-RS注解。支持可插拔的HTTP编码器和解码器（Gson，Jackson，Sax，JAXB，JAX-RS，SOAP）。支持Hystrix和它的Fallback。支持Ribbon的负载均衡。支持HTTP请求和响应的压缩。灵活的配置：基于 name 粒度进行配置支持多种客户端：JDK URLConnection、apache httpclient、okhttp，ribbon）支持日志支持错误重试url支持占位符可以不依赖注册中心独立运行

代码实现：

1、cloud-provider-payment8001模块controller层添加代码：

@GetMapping(value = "/feign/timeout")

public String paymentFeignTimeout() {

   try {

       // 暂停3秒钟 模拟超时任务

       TimeUnit.SECONDS.sleep(3);

  } catch (InterruptedException e) {

       e.printStackTrace();

  }

   return SERVER\_PORT;

}

2、新建cloud-consumer-feign-order80模块

3、pom文件（注意将实体类依赖换成自己的！）

<dependencies>

   <!--openfeign-->

   <dependency>

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

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

   </dependency>

   <!--eureka client-->

   <dependency>

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

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

   </dependency>

   <dependency>

       <groupId>org.xzq.springcloud</groupId>

       <artifactId>cloud-api-commons</artifactId>

       <version>${project.version}</version>

   </dependency>

   <dependency>

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

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

   </dependency>

   <!--监控-->

   <dependency>

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

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

   </dependency>

   <!--热部署-->

   <dependency>

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

       <artifactId>spring-boot-devtools</artifactId>

       <scope>runtime</scope>

       <optional>true</optional>

   </dependency>

   <dependency>

       <groupId>org.projectlombok</groupId>

       <artifactId>lombok</artifactId>

       <optional>true</optional>

   </dependency>

   <dependency>

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

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

       <scope>test</scope>

   </dependency>

</dependencies>

4、yml配置

server:

port: 80

​

eureka:

client:

  register-with-eureka: true

  fetch-registry: true

  service-url:

    defaultZone: http://eureka7001.com:7001/eureka/, http://eureka7002.com:7002/eureka/ # 入驻地址

5、主启动类

/\*\*

\* @ClassName: OrderFeignMain80

\* @description:

\* @author: XZQ

\* @create: 2020/3/7 17:21

\*\*/

@SpringBootApplication

@EnableEurekaClient

@EnableFeignClients//启用feign客户端

public class OrderFeignMain80 {

   public static void main(String[] args) {

       SpringApplication.run(OrderFeignMain80.class, args);

  }

}

6、service包下创建PaymentFeignService接口

/\*\*

\* @ClassName: PaymentFeignService

\* @description:

\* @author: XZQ

\* @create: 2020/3/8 15:30

\*\*/

@Component

@FeignClient(value = "CLOUD-PAYMENT-SERVICE")

public interface PaymentFeignService {

   /\*\*

    \* 根据id查询

    \*

    \* @param id

    \* @return

    \*/

   @GetMapping(value = "/payment/get/{id}")

   CommonResult getPaymentById(@PathVariable("id") Long id);

​

   /\*\*

    \* 模拟feign超时

    \*

    \* @return

    \*/

   @GetMapping(value = "/payment/feign/timeout")

   String paymentFeignTimeout();

}

​

7、controller层代码

/\*\*

\* @ClassName: OrderFeignClientController

\* @description:

\* @author: XZQ

\* @create: 2020/3/8 15:28

\*\*/

@RestController

@RequestMapping("/consumer")

public class OrderFeignClientController {

   @Autowired

   private PaymentFeignService paymentFeignService;

​

   /\*\*

    \* 根据id查询

    \*

    \* @param id

    \* @return

    \*/

   @GetMapping(value = "/payment/get/{id}")

   public CommonResult getPaymentById(@PathVariable("id") Long id) {

       return paymentFeignService.getPaymentById(id);

  }

​

   /\*\*

    \* 模拟feign超时

    \*

    \* @return

    \*/

   @GetMapping(value = "/payment/feign/timeout")

   public String paymentFeignTimeout() {

       // openfeign-ribbon, 客户端一般默认等待1秒钟

       return paymentFeignService.paymentFeignTimeout();

  }

}

8、测试出现访问超时错误，原因是，feign客户端默认超时时间是1秒，超时就出现异常。

解决办法：yml配置中添加

# 设置feign客户端超时时间(OpenFeign默认支持ribbon)

ribbon:

# 指的是建立连接所用的时间,适用于网络状态正常的情况下,两端连接所用的时间

ReadTimeout: 5000

# 指的是建立连接后从服务器读取到可用资源所用的时间

ConnectTimeout: 5000

9、访问成功！

OpenFeign日志增强

openfeign提供了日志打印功能。

Logger有四种类型：NONE(默认)、BASIC、HEADERS、FULL，通过注册Bean来设置日志记录级别

1、配置类

@Configuration

public class FeignConfig {

​

   /\*\*

    \* feignClient日志级别配置

    \*

    \* @return

    \*/

   @Bean

   public Logger.Level feignLoggerLevel() {

       // 请求和响应的头信息,请求和响应的正文及元数据

       return Logger.Level.FULL;

  }

}

2、yml配置文件添加

logging:

level:

  # feign日志以什么级别监控哪个接口

  com.atguigu.springcloud.service.PaymentFeignService: debug

十三、Hystrix

官网：https://github.com/Netflix/Hystrix/wiki

参考博客：https://www.cnblogs.com/cjsblog/p/9391819.html

 十三、Hystrix

Hystrix介绍

在微服务场景中，通常会有很多层的服务调用。如果一个底层服务出现问题，故障会被向上传播给用户。我们需要一种机制，当底层服务不可用时，可以阻断故障的传播。这就是断路器的作用。他是系统服务稳定性的最后一重保障。

在springcloud中断路器组件就是Hystrix。Hystrix也是Netflix套件的一部分。他的功能是，当对某个服务的调用在一定的时间内（默认10s），有超过一定次数（默认20次）并且失败率超过一定值（默认50%），该服务的断路器会打开。返回一个由开发者设定的fallback。

fallback可以是另一个由Hystrix保护的服务调用，也可以是固定的值。fallback也可以设计成链式调用，先执行某些逻辑，再返回fallback。

Hystrix的作用

对通过第三方客户端库访问的依赖项（通常是通过网络）的延迟和故障进行保护和控制。

在复杂的分布式系统中阻止级联故障。

快速失败，快速恢复。

回退，尽可能优雅地降级。

启用近实时监控、警报和操作控制。

官网：https://github.com/Netflix/Hystrix/wiki

参考博客：https://www.cnblogs.com/cjsblog/p/9391819.html

https://blog.csdn.net/tongtong\_use/article/details/78611225

服务降级

三种情况：

访问超时

运行错误

宕机

实例代码：

/\*\*

\* 访问超时

\*

\* @param id

\* @return

\*/

@HystrixCommand(fallbackMethod = "paymentInfo\_TimeOutHandler", commandProperties = {

@HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "3000")

})

public String paymentInfo\_TimeOut(Integer id) {

int timeNumber = 5;

try {

TimeUnit.SECONDS.sleep(timeNumber);

} catch (Exception e) {

e.printStackTrace();

}

return "线程池：" + Thread.currentThread().getName() + "paymentinfo\_Timeout,id:" + id + "\t" + "耗时(秒)" + timeNumber;

}

​

private String paymentInfo\_TimeOutHandler(Integer id) {

return "线程池：" + Thread.currentThread().getName() + "paymentInfo\_TimeOutHandler,id:" + id + "\t";

}

主启动类

@SpringBootApplication

@EnableEurekaClient

@EnableCircuitBreaker

public class PaymentHystrixMain8001 {

   public static void main(String[] args) {

       SpringApplication.run(PaymentHystrixMain8001.class, args);

  }

}

​

问题：

1、每个方法有一个对应的处理方法，代码膨胀。

2、处理方法和主业务逻辑混合在一起

解决方案：

@DefaultProperties(defaultFallback="")

1、类上加上注解：

@RestController

@RequestMapping("/consumer")

@DefaultProperties(defaultFallback = "payment\_Global\_FallbackMethod")

public class OrderHystrixController {

2、默认全局处理方法

/\*\*

\* 全局fallback方法

\*

\* @return

\*/

private String payment\_Global\_FallbackMethod() {

   return "Global异常处理信息,请稍后再试。";

}

3、@HystrixCommand//不加属性代表使用默认的全局处理方法。

服务熔断

//====服务熔断

​

/\*\*

\* 在10秒窗口期中10次请求有6次是请求失败的,断路器将起作用

\*

\* @param id

\* @return

\*/

@HystrixCommand(

       fallbackMethod = "paymentCircuitBreaker\_fallback", commandProperties = {

       @HystrixProperty(name = "circuitBreaker.enabled", value = "true"),// 是否开启断路器

       @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "10"),// 请求次数

       @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "10000"),// 时间窗口期/时间范文

       @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "60")// 失败率达到多少后跳闸

}

)

public String paymentCircuitBreaker(@PathVariable("id") Integer id) {

   if (id < 0) {

       throw new RuntimeException("\*\*\*\*\*id不能是负数");

  }

   String serialNumber = IdUtil.simpleUUID();

   return Thread.currentThread().getName() + "\t" + "调用成功,流水号:" + serialNumber;

}

​

public String paymentCircuitBreaker\_fallback(@PathVariable("id") Integer id) {

   return "id 不能负数,请稍后重试,o(╥﹏╥)o id:" + id;

}

Hystrix断路器使用时最常用的三个重要指标参数

​

在微服务中使用Hystrix 作为断路器时，通常涉及到以下三个重要的指标参数（这里是写在@HystrixProperties注解中，当然实际项目中可以全局配置在yml或properties中）

1、circuitBreaker.sleepWindowInMilliseconds

断路器的快照时间窗，也叫做窗口期。可以理解为一个触发断路器的周期时间值，默认为10秒（10000）。

2、circuitBreaker.requestVolumeThreshold

断路器的窗口期内触发断路的请求阈值，默认为20。换句话说，假如某个窗口期内的请求总数都不到该配置值，那么断路器连发生的资格都没有。断路器在该窗口期内将不会被打开。

3、circuitBreaker.errorThresholdPercentage

断路器的窗口期内能够容忍的错误百分比阈值，默认为50（也就是说默认容忍50%的错误率）。打个比方，假如一个窗口期内，发生了100次服务请求，其中50次出现了错误。在这样的情况下，断路器将会被打开。在该窗口期结束之前，即使第51次请求没有发生异常，也将被执行fallback逻辑。

综上所述，在以上三个参数缺省的情况下，Hystrix断路器触发的默认策略为：

在10秒内，发生20次以上的请求时，假如错误率达到50%以上，则断路器将被打开。（当一个窗口期过去的时候，断路器将变成半开（HALF-OPEN）状态，如果这时候发生的请求正常，则关闭，否则又打开）

十四、GateWay

暂时省略

十五、Spring Config

简介：在分布式系统中，由于服务数量巨多，为了方便服务配置文件统一管理，实时更新，所以需要分布式配置中心组件。Spring Cloud Config项目是就是这样一个解决分布式系统的配置管理方案。它包含了Client和Server两个部分，server提供配置文件的存储、以接口的形式将配置文件的内容提供出去，client通过接口获取数据、并依据此数据初始化自己的应用。

入门示例：

1、在github上新建一个仓库，如图所示

2、复制新建的git地址

3、在本地新建git仓库并clone

4、新建开发、生产、测试配置文件，并提交到github

参考下图：

命令：git add .

git commit

git push origin master

5、新建配置中心模块3344 cloud-config-center3344

添加依赖：

<dependencies>

   <dependency>

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

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

   </dependency>

   <dependency>

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

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

   </dependency>

   <dependency>

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

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

   </dependency>

   <dependency>

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

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

   </dependency>

   <dependency>

       <groupId>org.projectlombok</groupId>

       <artifactId>lombok</artifactId>

       <optional>true</optional>

   </dependency>

   <dependency>

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

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

       <scope>test</scope>

   </dependency>

</dependencies>

yml文件配置

server:

port: 3344

​

spring:

application:

  name: cloud-config-center

cloud:

  config:

    server:

      git:

        skipSslValidation: true # 跳过ssl认证

        uri: https://github.com/Jiaru0314/springcloud-config.git

        search-paths:

          - springcloud-config

    label: master

​

eureka:

client:

  service-url:

    defaultZone: http://eureka7001.com:7001/eureka

主启动类

@SpringBootApplication

@EnableConfigServer

public class MainConfigCenter3344 {

   public static void main(String[] args) {

       SpringApplication.run(MainConfigCenter3344.class, args);

  }

}

Spring Cloud Config 有它的一套访问规则，我们通过这套规则在浏览器上直接访问就可以。

/{application}/{profile}[/{label}]

/{application}-{profile}.yml

/{label}/{application}-{profile}.yml

/{application}-{profile}.properties

/{label}/{application}-{profile}.properties

{application} 就是应用名称，对应到配置文件上来，就是配置文件的名称部分，例如我上面创建的配置文件。

{profile} 就是配置文件的版本，我们的项目有开发版本、测试环境版本、生产环境版本，对应到配置文件上来就是以 application-{profile}.yml 加以区分，例如application-dev.yml、application-sit.yml、application-prod.yml。

{label} 表示 git 分支，默认是 master 分支，如果项目是以分支做区分也是可以的，那就可以通过不同的 label 来控制访问不同的配置文件了。

访问：http://localhost:3344/config-dev.yml

结果：

config:

info: master branch,springcloud-config/config-dev.yml version=7

配置成功，但是这样还存在一个问题，就是当我们github上更新配置后，我们的服务要重新启动才能使用新的配置，否则配置无法生效。解决方案就是动态的修改配置。

使用标签:@RefreshScope

步骤：

pom文件引入actuator依赖

修改yml,暴露监控端口

#暴露监控端点

management:

endpoints:

web:

exposure:

include: "\*"

@RefreshScope修饰控制类

cmd下 执行

curl -X POST "http://localhost:3355/actuator/refresh"

3355模块重新获取配置信息

十六、Bus消息中心

1、首先配置RabbitMQ

下载erlang 地址：http://erlang.org/download/otp\_win64\_21.3.exe 傻瓜式安装即可

下载RabbitMQ 地址：https://github.com/rabbitmq/rabbitmq-server/releases/download/v3.8.2/rabbitmq-server-3.8.2.exe 傻瓜式安装即可

进入rabbitmq安装目录下的sbin目录：输入

rabbitmq-plugins enable rabbitmq\_management

双击RabbitMQ service start

浏览器输入：localhost:15672 如果出现如下页面，则安装成功

输入账号和密码：guest、guest 可以看到RabbitMQ的主界面

2、参照3355模块新建3366模块

3344模块添加新依赖：

<dependency>

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

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

</dependency>

3344新添yml:

rabbitmq:

host: localhost

port: 5672

username: guest

password: guest

# 暴露bus刷新配置的端点

management:

endpoints:

web:

exposure:

include: "bus-refresh"

3355、3366yml新增：

rabbitmq: #rabbitmq相关配置，15672是web管理端口，5672是mq访问端口

port: 5672

host: localhost

username: guest

password: guest

#暴露监控端点

management:

endpoints:

web:

exposure:

include: "\*"

1、修改github中版本号

2、cmd 刷新3344

curl -X POST "http://localhost:3344/actuator/bus-refresh"

结果：3355、3366 都已经改变，一次修改、广播通知、到处运行。

上面是全局通知，但如果我们想定点通知该如何做呢？

定点通知：只通知3355，不通知3366

实现方法：cmd 刷新3344

curl -X POST "http://localhost:3344/actuator/bus-refresh/config-client:3355"

第十五章 SpringCloud Stream

问题：为什么要引入SpringCloud Stream

举例：对于我们Java程序员来说，可能有时要使用ActiveMQ,有时要使用RabbitMQ,甚至还有RocketMQ以及Kafka，这之间的切换似乎很麻烦，我们很难，也没有太多时间去精通每一门技术，那有没有一种新技术的诞生，让我们不再关注具体MQ的细节，自动的给我们在各种MQ内切换。

简介：Spring Cloud Stream 是一个用来为微服务应用构建消息驱动能力的框架。它可以基于 Spring Boot 来创建独立的、可用于生产的 Spring 应用程序。Spring Cloud Stream 为一些供应商的消息中间件产品提供了个性化的自动化配置实现，并引入了发布-订阅、消费组、分区这三个核心概念。通过使用 Spring Cloud Stream，可以有效简化开发人员对消息中间件的使用复杂度，让系统开发人员可以有更多的精力关注于核心业务逻辑的处理。但是目前 Spring Cloud Stream 只支持 RabbitMQ 和 Kafka 的自动化配置。

一句话：屏蔽底层消息中间件的差异，降低切换成本，统一消息的编程模型。

Binder Implementations 绑定器

通过绑定器Binder作为中间件，实现了应用程序与消息中间件细节的解耦。

Input对应消息生产者

Output对应消息消费者

入门案例

新建子模块

cloud-stream-rabbitmq-provider8801 消息生产模块

pom

<dependencies>

   <dependency>

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

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

   </dependency>

   <dependency>

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

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

   </dependency>

   <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-stream-rabbit</artifactId>

   </dependency>

   <dependency>

       <groupId>org.projectlombok</groupId>

       <artifactId>lombok</artifactId>

       <optional>true</optional>

   </dependency>

   <dependency>

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

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

       <scope>test</scope>

   </dependency>

</dependencies>

yml

server:

port: 8801

spring:

application:

name: cloud-stream-provider

cloud:

stream:

binders: # 在此处配置要绑定的rabbitMQ的服务信息

defaultRabbit: # 表示定义的名称，用于binding的整合

type: rabbit # 消息中间件类型

environment: # 设置rabbitMQ的相关环境配置

spring:

rabbitmq:

host: localhost

port: 5672

username: guest

password: guest

bindings: # 服务的整合处理

output: # 这个名字是一个通道的名称

destination: studyExchange # 表示要使用的exchange名称定义

content-type: application/json # 设置消息类型，本次为json，文本则设为text/plain

binder: defaultRabbit # 设置要绑定的消息服务的具体设置

eureka:

client:

service-url:

defaultZone: http://eureka7001.com:7001/eureka

instance:

lease-renewal-interval-in-seconds: 2 # 设置心跳的间隔时间，默认30

lease-expiration-duration-in-seconds: 5 # 超过5秒间隔，默认90

instance-id: send-8801.com # 主机名

prefer-ip-address: true # 显示ip

业务类

主启动类

IMessageProvider接口

public interface IMessageProvider {

   String send();

}

MessageProviderImpl实现类

public class MessageProviderImpl implements IMessageProvider {

@Override

public String send() {

   return null;

}

}

控制层

@RestController

public class SendMessageController {

​

   @Autowired

   private IMessageProvider messageProvider;

​

   @GetMapping("/sendMessage")

   public String send() {

       return messageProvider.send();

  }

}

启动rabbitMQ,7001,8801 访问localhost:8801/sendMessage

返回结果：62b3cb3e-ca40-410e-a601-dc72dfd004b5

新建子模块

cloud-stream-rabbitmq-consumer8802 消息消费模块

pom文件同8801

yml文件

server:

port: 8802

​

spring:

application:

  name: cloud-stream-consumer

cloud:

  stream:

    binders: # 在此处配置要绑定的rabbitMQ的服务信息

      defaultRabbit: # 表示定义的名称，用于binding的整合

        type: rabbit # 消息中间件类型

        environment: # 设置rabbitMQ的相关环境配置

          spring:

            rabbitmq:

              host: localhost

              port: 5672

              username: guest

              password: guest

    bindings: # 服务的整合处理

      input: # 这个名字是一个通道的名称

        destination: studyExchange # 表示要使用的exchange名称定义

        content-type: application/json # 设置消息类型，本次为json，文本则设为text/plain

        binder: defaultRabbit # 设置要绑定的消息服务的具体设置

        # group: spectrumrpcA

​

eureka:

client:

  service-url:

    defaultZone: http://eureka7001.com:7001/eureka

instance:

  lease-renewal-interval-in-seconds: 2 # 设置心跳的间隔时间，默认30

  lease-expiration-duration-in-seconds: 5 # 超过5秒间隔，默认90

  instance-id: receive-8802.com #主机名

  prefer-ip-address: true # 显示ip

业务类

主启动类

服务类

/\*\*

\* @ClassName: ReceiveMessageListenerController

\* @description:

\* @author: XZQ

\* @create: 2020/3/10 15:34

\*\*/

@Component

@EnableBinding(Sink.class)

public class ReceiveMessageListenerController {

​

   @Value("${server.port}")

   private String serverPort;

​

   @StreamListener(Sink.INPUT)

   public void input(Message<String> message) {

       System.out.println("消费者1，-------" + message.getPayload() + "\t port:" + serverPort);

  }

}

启动8802，8801发送消息，可以看到8802可以接受到消息，案例完成。

消息重复消费

不同的组存在重复消费，相同的组之间竞争消费。

参照8802建立8803模块

8802、8803yml加入组名 spectrumrpcA

bindings: # 服务的整合处理

input: # 这个名字是一个通道的名称

destination: studyExchange # 表示要使用的exchange名称定义

content-type: application/json # 设置消息类型

binder: defaultRabbit # 设置要绑定的消息服务的具体设置

# group: spectrumrpcA

分别启动8802、8803

8801发送消息、发现只有一个模块收到消息