

# Spring Cloud

## Spring Boot和Spring Cloud的版本选型

spring boot版本最少需要2.0以上，Spring Cloud使用Hoxton版本

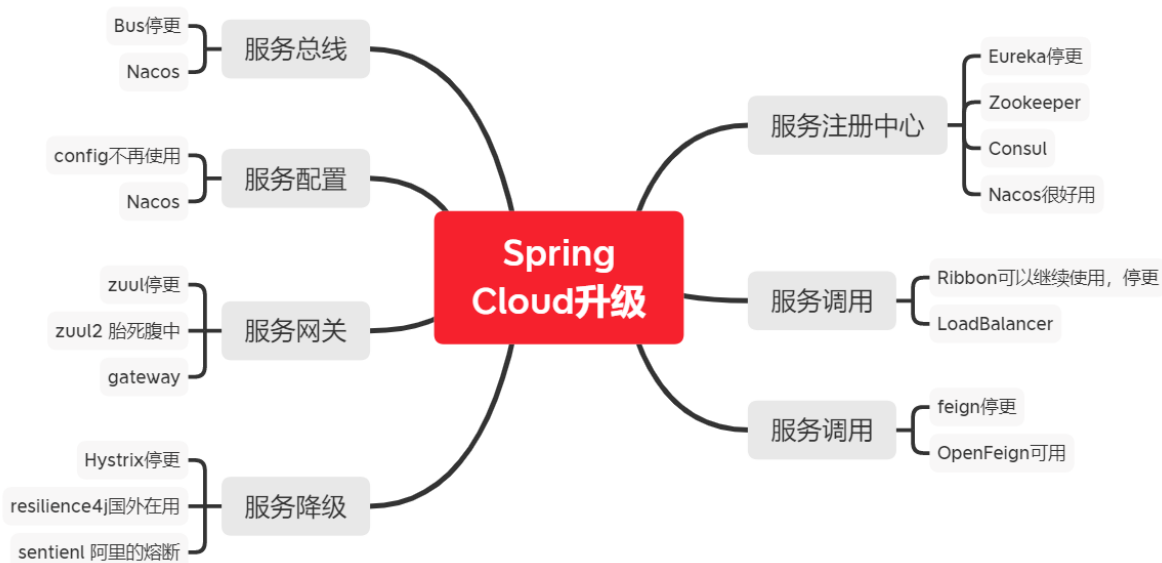
<https://spring.io/projects/spring-cloud#overview> 该网站可以看spring boot与spring cloud版本之间的对应关系

<https://start.spring.io/actuator/info> 这个API可以查看更详细的信息

```
"spring-cloud": {  
  "Hoxton.SR10": "Spring Boot >=2.2.0.RELEASE and <2.3.10.BUILD-SNAPSHOT",  
  "Hoxton.BUILD-SNAPSHOT": "Spring Boot >=2.3.10.BUILD-SNAPSHOT and <2.4.0.M1",  
  "2020.0.0-M3": "Spring Boot >=2.4.0.M1 and <=2.4.0.M1",  
  "2020.0.0-M4": "Spring Boot >=2.4.0.M2 and <=2.4.0-M3",  
  "2020.0.0": "Spring Boot >=2.4.0.M4 and <=2.4.0",  
  "2020.0.2": "Spring Boot >=2.4.1 and <2.5.0-M1",  
  "2020.0.3-SNAPSHOT": "Spring Boot >=2.4.5-SNAPSHOT"  
},
```

## Spring Cloud各种组件的停更/升级和替换

不再推进使用，不再修复小bug，不再接受合并请求，不再发布新版本。。。



最好的资料，还是官网

<https://docs.spring.io/spring-cloud/docs/Hoxton.SR10/reference/htmlsingle/>

## 父工程POM

```
<?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.echo</groupId>  
  <artifactId>SpringCloudStudyNew</artifactId>
```

```

<version>1.0-SNAPSHOT</version>
<packaging>pom</packaging>
<!-- 统一jar包和版本号的管理 -->
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>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>5.1.47</mysql.version>
    <druid.version>1.1.16</druid.version>
    <mybatis.spring.boot.version>1.3.0</mybatis.spring.boot.version>
</properties>
<!--在子模块继承之后，可以锁定版本，子module不用写groupId和version-->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-project-info-reports-plugin</artifactId>
            <version>3.0.0</version>
        </dependency>
        <!--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>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-alibaba-dependencies</artifactId>
            <version>2.1.0.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <!--mysql-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>${mysql.version}</version>
            <scope>runtime</scope>
        </dependency>
        <!-- druid-->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>${druid.version}</version>
        </dependency>
    </dependencies>
</dependencyManagement>

```

```

        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
            <version>${mybatis.spring.boot.version}</version>
        </dependency>
        <!--junit-->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>${junit.version}</version>
        </dependency>
        <!--log4j-->
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>${log4j.version}</version>
        </dependency>
    </dependencies>
</dependencyManagement>
</project>

```

Maven中的dependencies和dependencyManagement的区别：

**dependencyManagement:**提供了一种管理依赖版本号的方式。通常会在一个组织或者项目的最顶层的父POM中看到该元素。使用pom.xml的dependencyMangment元素能让所有在子项目中沿用一個依赖而不用显示的列出版本号。Maven会沿着父子层次向上走，直到找到一个拥有dependencyMangment元素的项目，然后它就会使用这个dependencyMangment元素中指定的版本号，说直白一点：

如果子项目中没有指定版本号，而父项目的dependencyMangment元素指定了，就用父项目的，如果子项目中指定了，就用子项目的。而且这个元素中只是声明依赖，并没有引入，就像一个接口一样。我只是说了我会用这些版本，真正的引入还是在子项目中，子项目需要显示的声明依赖。如果不在子项目中声明依赖，是不会从父项目中继承下来的，只有在子项目中写了该依赖项，并且没有指定具体版本，才会从父项目中继承该项，并且version和scope都读取自父pom

## 支付模块构建

- 建module
- 改POM

```

<?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>SpringCloudStudyNew</artifactId>
        <groupId>com.echo</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>cloud-provider-payment-8001</artifactId>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>

```

```

        <maven.compiler.target>8</maven.compiler.target>
    </properties>
    <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.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid-spring-boot-starter</artifactId>
            <version>1.1.10</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-jdbc</artifactId>
        </dependency>
    <!--      springboot热部署-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
    <!--      runtime表示被依赖项目无需参与项目的编译，不过后期的测试和运行周期需要其
    参与。与compile相比，跳过编译而已，-->
        <scope>runtime</scope>
    <!--      <optional>true</optional>表示两个项目之间依赖不传递；不设置
    optional或者optional是false，表示传递依赖。-->
        <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>
</project>

```

- 写YML

```

server:
  port: 8001

```

```

spring:
  application:
    name: cloud-payment-service
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource           #当前数据源操作类型
    driver-class-name: org.gjt.mm.mysql.Driver           #mysql驱动包，用的是
spring的jdbc
    url: jdbc:mysql://localhost:3306/db2019?
useUnicode=true&characterEncoding=utf-8&useSSL=false
    username: root
    password: s814466057

mybatis:
  mapper-locations: classpath:mapper/*.xml
  type-aliases-package: com.echo.pojo

```

- 主启动

```

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

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

```

- 业务类

代码不粘了

## 热部署配置

代码修改，自动重启

1.添加devtools热部署工具到项目中,子模块的pom

```

<!--      springboot热部署-->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
<!--      runtime表示被依赖项目无需参与项目的编译，不过后期的测试和运行周期需要其参与。与compile相比，跳过编译而已，-->
  <scope>runtime</scope>
<!--      <optional>true</optional>表示两个项目之间依赖不传递；不设置optional或者optional是false，表示传递依赖。-->
  <optional>true</optional>
</dependency>

```

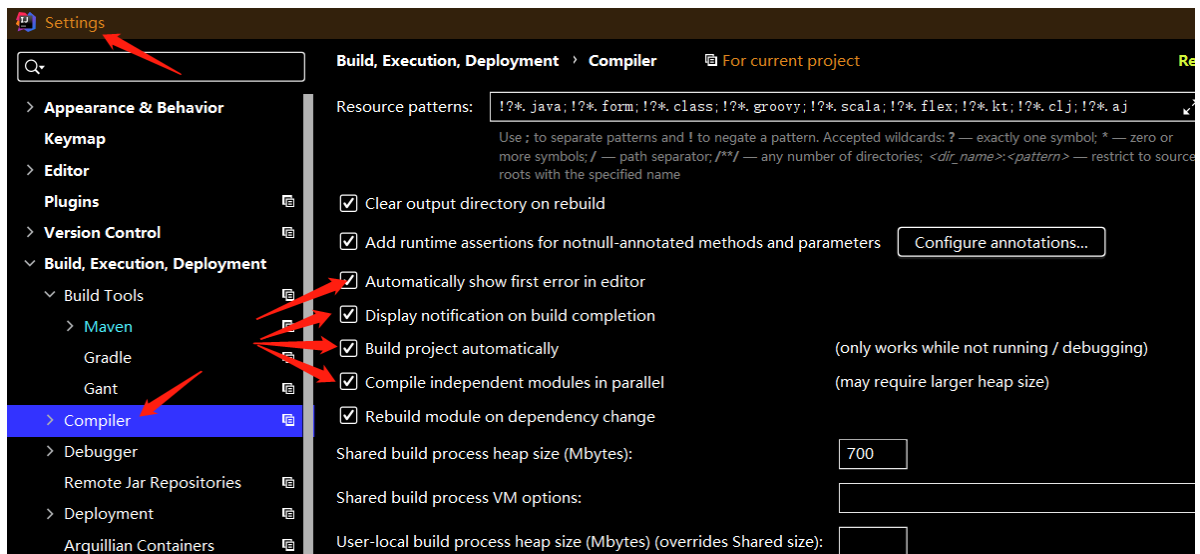
2.在聚合父类总工程的pom.xml中添加插件

```

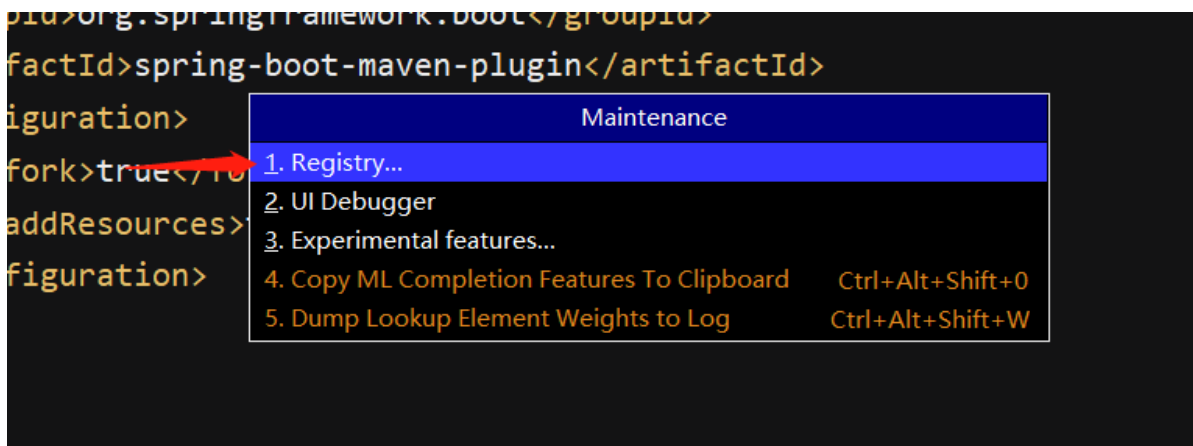
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <fork>true</fork>
        <addResources>true</addResources>
      </configuration>
    </plugin>
  </plugins>
</build>

```

### 3.开启自动编译选项



### 4.更新值，在IDEA中按快捷键Ctrl + Shift + Alt + 左/ 调出一个菜单



选择 Registry 勾选 compiler.automake.allow.when.app.running

Key	Value
batch.inspections.startup.activities.timeout	180
batch.jvm.inspections	<input type="checkbox"/>
bigger.font.in.project.view	<input type="checkbox"/>
✓ caches.indexerThreadsCount	-1
✓ cidr.xcode.derived.data.override	<input checked="" type="checkbox"/>
clipboard.history.max.items	100
clipboard.history.max.memory	10000000
com.intellij.sh.run.with.wsl	<input type="checkbox"/>
command.line.execution.timeout	30
comment.by.line.bulk.lines.trigger	100
compiler.automake.allow.parallel	<input checked="" type="checkbox"/>
compiler.automake.allow.when.app.running	<input checked="" type="checkbox"/>
compiler.automake.postpone.when.idle.less.than	3000
compiler.automake.trigger.delay	300
compiler.build.data.unused.threshold	30
compiler.build.report.statistics	<input type="checkbox"/>
compiler.document.save.enabled	<input type="checkbox"/>

勾选actionSystem.assertFocusAccessFromEdt

action.aware.typeahead.actions.list	GotoClass,GotoFile,GotoSymbol,FindInPath,ReplaceInPath,FileStr
action.aware.typeaheadTimeout	10000
actionSystem.always.update.toolbar.actions	<input type="checkbox"/>
actionSystem.assertFocusAccessFromEdt	<input checked="" type="checkbox"/>
actionSystem.cache.data	<input checked="" type="checkbox"/>
actionSystem.commandProcessingTimeout	3000
actionSystem.enableAbbreviations	<input checked="" type="checkbox"/>
actionSystem.fix.alt.gr	<input checked="" type="checkbox"/>
actionSystem.fixLostTyping	<input checked="" type="checkbox"/>

5.重启IDEA即可

重启后发现已经可以热部署，不过有个短暂的缓冲期，并不是一修改代码就会立即重启，而是会稍等一会

但是，项目上线之后一定要关闭该功能！！！！只允许在开发阶段使用！！

## 消费者模块的构建 cloud-consumer-order-80

1.同样的套路

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>
<!--     .springboot热部署-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <!--      runtime表示被依赖项目无需参与项目的编译，不过后期的测试和运行
周期需要其参与。与compile相比，跳过编译而已，-->
    <scope>runtime</scope>
    <!--      <optional>true</optional>表示两个项目之间依赖不传递；不设置optional或者optional是false，表示传递依赖。-->
    <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>

```

需要注意的是，这部分只是服务的消费方，因此不需要操作数据库，也就不需要引入数据库操作的包

## 2. 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服务模板类，是Spring提供的用于访问Rest服务的客户端模板工具集。具体什么是RestTemplate自行百度。

说白了，该工程就是通过RestTemplate远程调用服务提供方来实现的。具体需要记录的就是如何使用RestTemplate进行远程调用。方法如下：

- 配置一个config类，用来注入RestTemplate

```

package com.echo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class ApplicationContextConfig {

    @Bean
    public RestTemplate getRestTemplate(){
        return new RestTemplate();
    }
}

```

- 实现一个Controller，进行远程调用

```

package com.echo.controller;

import com.echo.pojo.CommonResult;
import com.echo.pojo.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;

@RestController
@Slf4j
public class OrderController {
    public static final String PAYMENT_URL = "http://localhost:8001";

    @Autowired
    private RestTemplate restTemplate;

    @GetMapping("/consumer/payment/add")
    public CommonResult<Payment> create(Payment payment){
        //参数说明: url,post请求体, 请求返回的结果映射
        return restTemplate.postForObject(PAYMENT_URL +
"/payment/add",payment,CommonResult.class);
    }

    @GetMapping("/consumer/payment/get/{id}")
    public CommonResult<Payment> getPayment(@PathVariable("id")Long id){
        return restTemplate.getForObject(PAYMENT_URL + "/payment/get/" +
id,CommonResult.class);
    }
}

```

## • 结果

GET
http://localhost/consumer/payment/get/1
Send

Params
Authorization
Headers (7)
Body
Pre-request Script
Tests
Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body
Cookies
Headers (5)
Test Results

Status: 200 OK
Time: 181 ms
Size: 230 B
Save

Pretty
Raw
Preview
Visualize
JSON

```

1 {
2   "code": 200,
3   "message": "success",
4   "data": {
5     "id": 1,
6     "serial": "aabbbaa"
7   }
8 }

```

## 3.测试插入

GET http://localhost/consumer/payment/add?serial=111

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	serial	111	
	Key	Value	Description

Body Cookies Headers (5) Test Results

Status: 200 OK Time: 265 ms Size: 205 B Sav

Pretty Raw Preview Visualize JSON

```
1 {
2   "code": 200,
3   "message": "success",
4   "data": 1
5 }
```

出现问题!

id	serial
1	aabbbaa
2	xxwwda
3	(Null)

出现问题的原因是，并没有在服务提供方的Controller中接收参数的地方加@RequestBody注解。

@RequestBody注解将Post请求体中的参数，映射到bean上

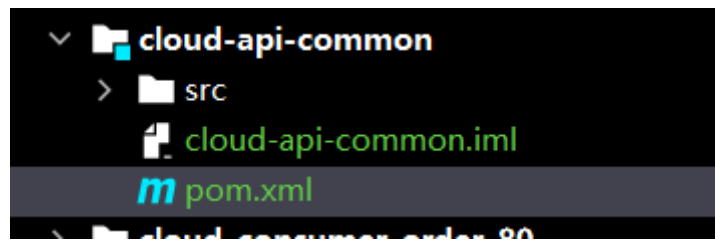
要在服务提供方添加该注解

```
@PostMapping(value = "/payment/add")
public CommonResult add(@RequestBody Payment payment){ //要加RequestBody注解
    int result = paymentService.add(payment);
    log.info("insert result : " + result);
    if (result > 0){
        return new CommonResult(200,"success",result);
    }
    else{
        return new CommonResult(500,"error",result)
    }
}
```

## 工程重构

将系统中的冗余部分进行重构，比如POJO，在消费方和提供方都会存在。

新建一个moudle进行公共部分的封装。



1.pom

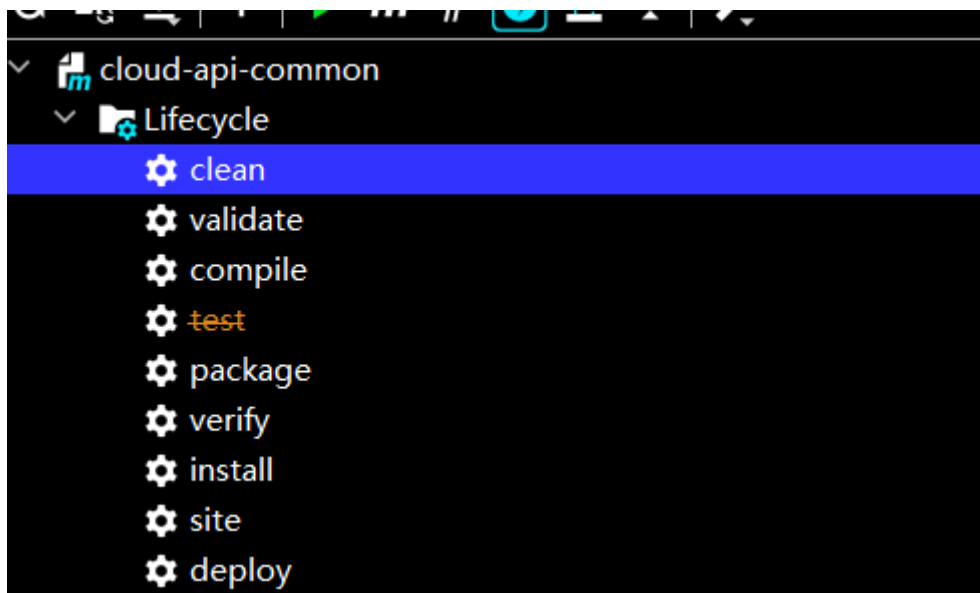
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <parent>
    <artifactId>SpringCloudStudyNew</artifactId>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

  <artifactId>cloud-api-common</artifactId>

  <properties>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>8</maven.compiler.target>
  </properties>

  <dependencies>
    <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>cn.hutool</groupId>
      <artifactId>hutool-all</artifactId>
      <version>5.1.0</version>
    </dependency>
  </dependencies>
</project>
```

2.对cloud-api-common进行清理，在maven中进行clean和install，将其作为组件进行安装，方便其余模块进行调用



### 3.重新整理consumer-80和provider-8001

将两个子工程中的pojo包进行删除，然后将我们安装的cloud-api-common在80和8001的pom文件中进行引入，从而使两个子工程可以对pojo进行调用。

```
<!--      引入自定义的公共模块-->
<dependency>
  <groupId>com.echo</groupId>
  <artifactId>cloud-api-common</artifactId>
  <version>${project.version}</version>
</dependency>
```

-----第一阶段结束，项目架构-----



### EurekaServer服务端安装 cloud-eureka-server-7001

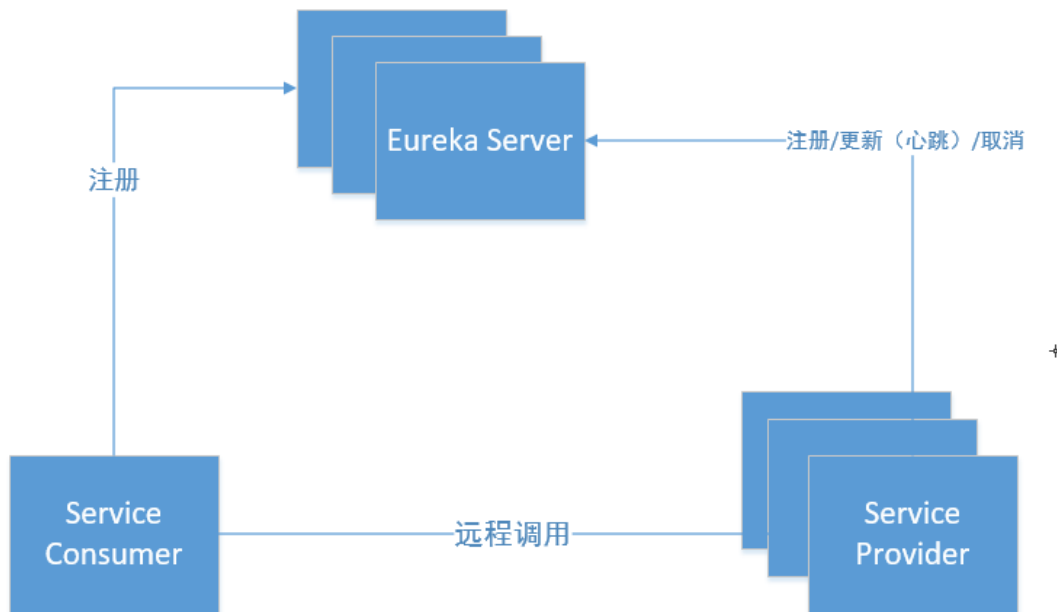
eureka包含两个组件：Eureka Server和Eureka Client

#### Eureka Server提供服务注册服务

各个微服务节点通过配置启动后，会在EurekaServer中进行注册，这样EurekaServer中的服务注册表中将会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观看到。

#### Eureka Client通过注册中心进行访问

Eureka Client是一个java客户端，用于简化Eureka Server的交互，客户端同时也具备一个内置的，使用轮询负载均衡算法的负载均衡器。在启动应用后，将会向Eureka Server发送心跳（默认周期为30秒）。如果Eureka Server在多个心跳周期内没有接收到某个节点的心跳，Eureka Server将会从服务注册表中将这个节点移除（默认90秒）



1.pom

```
<?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>SpringCloudStudyNew</artifactId>
    <groupId>com.echo</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <artifactId>cloud-eureka-server-7001</artifactId>

  <properties>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>8</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
    </dependency>
    <dependency>
      <groupId>com.echo</groupId>
      <artifactId>cloud-api-common</artifactId>
```

```

        <version>1.0-SNAPSHOT</version>
    </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>
    <!--      springboot热部署-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <!--      runtime表示被依赖项目无需参与项目的编译，不过后期的测试和运行
周期需要其参与。与compile相比，跳过编译而已，-->
        <scope>runtime</scope>
        <!--      <optional>true</optional>表示两个项目之间依赖不传递；不设置optional或者optional是false，表示传递依赖。-->
        <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>
</project>

```

## 2.application.yml

```

server:
  port: 7001

eureka:
  instance:
    hostname: localhost #eureka服务端的实例名称
  client:
    register-with-eureka: false #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，职责是维护服务实例，并不需要去检索服务
  service-url:
    # 设置与Eureka Server交互的地址查询服务和注册服务都需要依赖这个地址
    defaultZone: http://${eureka.instance.hostname}:${server.port}

```

## 3.启动类

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

@SpringBootApplication
@EnableEurekaServer //代表这是服务的注册中心
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

## 服务提供方8001入驻Eureka

相当于公司注册进物业公司

### 1.改pom

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

### 2.改yml

```
eureka:
  client:
    #表示将自己注册进EurekaServer,默认为true
    register-with-eureka: 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
    fetch-registry: true
    service-url:
      defaultZone: http://localhost:7001/eureka
```

### 3.修改主启动类

```
@SpringBootApplication
@EnableEurekaClient //标识是EurekaClient端
public class PaymentApplication {
    public static void main(String[] args) {
        SpringApplication.run(PaymentApplication.class, args);
    }
}
```

## 服务消费方80入驻Eureka

### 1.改pom

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

## 2.改yml

```
spring:
  application:
    name: cloud-order-service

eureka:
  client:
    #表示将自己注册进EurekaServer,默认为true
    register-with-eureka: 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
    fetch-registry: true
    service-url:
      defaultZone: http://localhost:7001/eureka
```

### eureka.client.registry-fetch-interval-seconds

表示eureka client间隔多久去拉取服务注册信息,默认为30秒,对于api-gateway,如果要迅速获取服务注册状态,可以缩小该值,比如5秒

## 3.改主启动类

```
@SpringBootApplication
@EnableEurekaClient      //标识为Eureka客户端
public class OrderApplication {
    public static void main(String[] args) {
        SpringApplication.run(OrderApplication.class,args);
    }
}
```

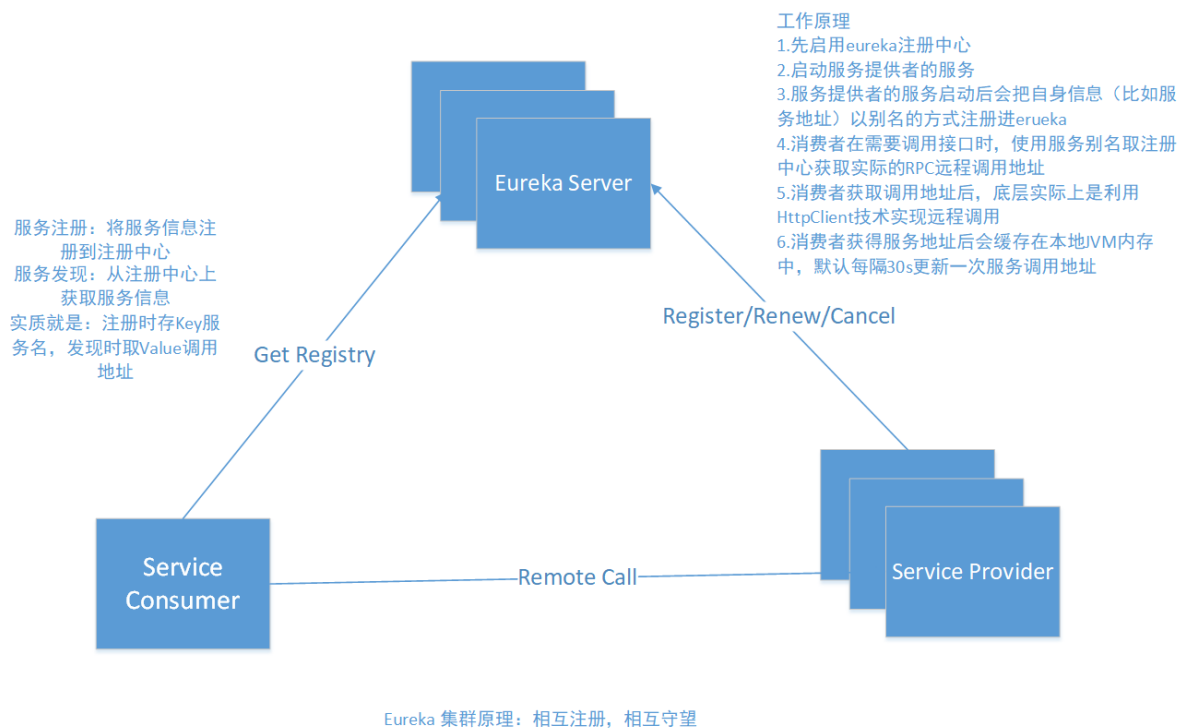
消费者服务启动时,会发送一个Rest请求给服务注册中心,来获取上面注册的服务清单。为了性能考虑,Eureka Server会维护一份只读的服务注册清单来返回给客户端,同时该缓存清单默认会每隔30秒更新一次。

服务消费者在获取服务清单后,通过服务名可以获取具体提供服务的实例名和该实例的元数据信息。因为有这些服务实例的详细信息,所以客户端可以根据自己的需要决定具体调用哪个实例,在Ribbon中会默认采用轮询的方式进行调用,从而实现客户端的负载均衡。

在系统运行过程中必然会面临关闭或重启服务的某个实例的情况,在服务关闭操作时,会触发一个服务下线的Rest服务请求给Eureka Server,告诉服务注册中心:“我要下线了。”服务端在接收到该请求后,将该服务状态置位下线(DOWN),并把该下线事件传播出去。



## Eureka集群原理说明



1.搭建, 新建一个cloud-eureka-server-7002

2.修改本机host映射文件

```
#####SpringCloud#####
127.0.0.1 eureka7001.com
127.0.0.1 eureka7002.com
```

3.修改7001的yml

```
server:
  port: 7001

eureka:
  instance:
    hostname: eureka7001.com #配置集群时要使用自己的名字
    # hostname: localhost #eureka服务端的实例名称
  client:
    register-with-eureka: false #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, 职责是维护服务实例, 并不需要去检索服务
  service-url:
    # 设置与Eureka Server交互的地址查询服务和注册服务都需要依赖这个地址
    #defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
    defaultZone: http://eureka7002.com:7002/eureka/
```

4.修改7002的yml

```
server:
  port: 7002

eureka:
  instance:
    hostname: eureka7002.com #eureka服务端的实例名称
  client:
    register-with-eureka: false #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，职责是维护服务实例，并不需要去检索服务
  service-url:
    # 设置与Eureka Server交互的地址查询服务和注册服务都需要依赖这个地址
    defaultZone: http://eureka7001.com:7001/eureka/
```

5.主启动类不变，然后分别启动7001与7002即可

← → ↻ ⚠ 不安全 | eureka7001.com:7001

## spring Eureka

### System Status

Environment	test	Current time
Data center	default	Uptime
		Lease expiration enabled
		Renews threshold
		Renews (last min)

### DS Replicas

eureka7002.com

← → ↻ ⚠ 不安全 | eureka7002.com:7002

## spring Eureka

### System Status

Environment	test
Data center	default

### DS Replicas

eureka7001.com

## 服务消费方和服务提供方入驻Eureka集群

1.服务消费方和服务提供方都只需修改配置文件即可

只需要修改eureka的defaultZone就行

```
defaultZone: http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka
#集群版
```

## 服务提供方集群配置

1.再创建一个服务提供方的模块cloud-provider-payment-8002，将8001中的代码和配置复制进去

2.修改两个服务提供方的controller，声明端口号

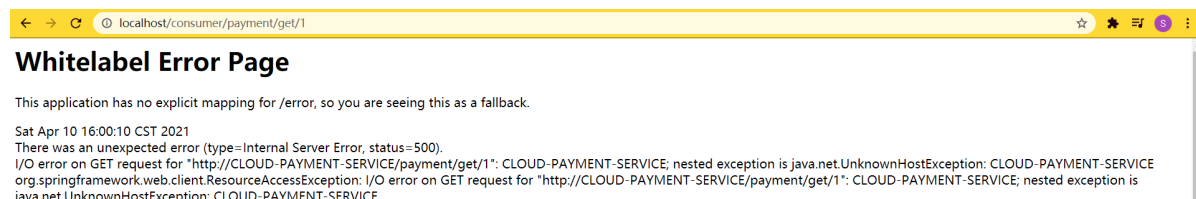
```
@Value("${server.port}")    //读取配置文件中的端口号
private String serverPort;
```

3.修改服务消费方中对于服务提供方的硬编码

找到OrderController，将原本的单机版服务提供方的硬编码进行修改，改为以服务名称进行调用的方式

```
public class OrderController {
    //public static final String PAYMENT_URL = "http://localhost:8001";    //单机版
    public static final String PAYMENT_URL = "http://CLOUD-PAYMENT-SERVICE";
```

4.出现问题



原因：虽然是通过服务名称进行的访问，但是该服务名称下有多台主机来提供服务，服务消费方无法确定是哪一台主机进行服务的提供，所以会产生错误

5.问题解决

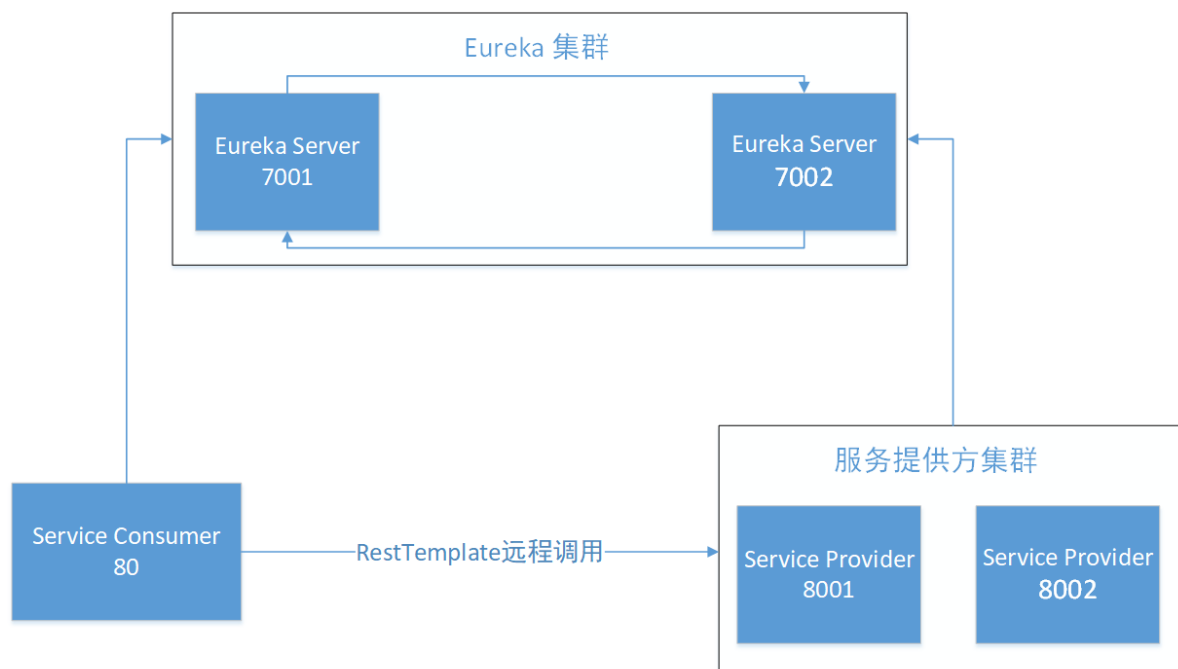
开启RestTemplate的负载均衡，在ApplicationContextConfig.java配置类对RestTemplate的配置上，添加负载均衡的注解

```
@Configuration
public class ApplicationContextConfig {

    @Bean
    @LoadBalanced    //负载均衡，根据服务名，找到不同的主机名,是由loadbalancer提供的
    public RestTemplate getRestTemplate(){
        return new RestTemplate();
    }
}
```

负载均衡和网关路由有什么区别？

-----第二阶段结束，项目架构-----



## Actuator微服务信息完善

### 1.主机名称：服务名称修改

修改8001和8002的yml文件，添加一个配置instance instance-id

```
eureka:
  client:
    #表示将自己注册进EurekaServer,默认为true
    register-with-eureka: 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
    fetch-registry: true
    service-url:
      #defaultZone: http://localhost:7001/eureka #单机版
      defaultZone:
        http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka #集群版
    instance: # 在eureka界面展示时的名称
    instance-id: payment8002
```

8001也一样

最后修改之后，可以在eureka的展示页面上看到如下

DS Replicas			
eureka7001.com			
Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - localhost:cloud-order-service:80
CLOUD-PAYMENT-SERVICE	n/a (2)	(2)	UP (2) - payment8002, payment8001

### 2.访问路径提示IP地址

在instance配置下再添加一条配置即可

```
instance: # 在eureka界面展示时的名称
  instance-id: payment8001
  prefer-ip-address: true
```

CLOUD-PAYMENT-SERVICE	n/a (2)	(2)	UP (2) - <a href="#">payment8002</a>
General Info			
Name	Value		
total-avail-memory	445mb		
environment	test		
num-of-cpus	8		
current-memory-usage	96mb (21%)		
server-uptime	01:04		
192.168.45.1:8002/actuator/info	http://eureka7001.com:7001/eureka/		

## Eureka服务发现Discovery

功能：对于注册进eureka中的微服务，可以通过服务发现来获得该服务的信息，说白了就是注册进来的服务将自己微服务的信息写好暴露给外面,暴露给注册中心和服务提供方

1.修改服务提供方8001的Controller

添加discoveryClient,注意一定是spring包下的。然后定义一个方法获得服务信息

```
package com.echo.controller;

import com.echo.pojo.CommonResult;
import com.echo.pojo.Payment;
import com.echo.service.PaymentService;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@Slf4j
public class PaymentController {
    @Autowired
    private PaymentService paymentService;

    @Value("${server.port}")
    private String serverPort;

    @Autowired
    private DiscoveryClient discoveryClient;    //添加discoveryClient

    @PostMapping(value = "/payment/add")
    public CommonResult add(@RequestBody Payment payment){
        int result = paymentService.add(payment);
        log.info("insert result : " + result);
        if (result > 0){
            return new CommonResult(200,"success,the port is :" +
serverPort,result);
        }
        else{
            return new CommonResult(500,"error",result);
        }
    }
}
```

```

    }

    @GetMapping(value = "/payment/get/{id}")
    public CommonResult add(@PathVariable("id")Long id){
        Payment paymentById = paymentService.getPaymentById(id);
        log.info("query result : " + paymentById);
        System.out.println("Hhhh");
        System.out.println("HHHHH");
        if (paymentById != null){
            return new CommonResult(200,"success,the port is :" +
serverPort,paymentById);
        }
        else{
            return new CommonResult(500,"error",id);
        }
    }
}

@GetMapping(value = "/payment/discovery")
public Object discovery(){    //获得服务信息
    //得到服务清单列表
    List<String> services = discoveryClient.getServices();
    services.forEach(System.out::println);
    //得到CLOUD-PAYMENT-SERVICE服务下的所有服务实例
    List<ServiceInstance> instances = discoveryClient.getInstances("CLOUD-
PAYMENT-SERVICE");
    for (ServiceInstance instance : instances){
        System.out.println(instance.getServiceId() + "\t" +
instance.getHost() + "\t" + instance.getPort()
        + instance.getUri());
    }
    return this.discoveryClient;
}
}

```

8002一样

## 2.主启动类添加注解

```

@SpringBootApplication
@EnableEurekaClient    //标识是EurekaClient端
@EnableDiscoveryClient //添加的注解
public class PaymentApplication {
    public static void main(String[] args) {
        SpringApplication.run(PaymentApplication.class,args);
    }
}

```

## 3.结果

```
localhost:8002/payment/discovery
{
  - discoveryClients: [
    - {
      - services: [
        "cloud-payment-service",
        "cloud-order-service"
      ],
      order: 0
    },
    - {
      services: [ ],
      order: 0
    }
  ],
  - services: [
    "cloud-payment-service",
    "cloud-order-service"
  ],
  order: 0
}
```

cloud-payment-service		
cloud-order-service		
CLOUD-PAYMENT-SERVICE	192.168.45.1	8001http://192.168.45.1:8001
CLOUD-PAYMENT-SERVICE	192.168.45.1	8002http://192.168.45.1:8002

#### 4.结论

由上述可以得出，如果服务提供方给服务消费方暴露这样一个接口，那么服务调用方就可以通过该接口获得服务提供方的一些信息，从而进行访问

### Eureka自我保护机制

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

#### 1.概述

保护模式主要用于一组客户端和Eureka Server之间存在网络分区场景下的保护，**一旦进入保护模式，Eureka Server将会尝试保护其服务注册表中的信息，不再删除服务注册表中的数据，也就是不会注销任何微服务。**简单来说就是，某一时刻微服务不可用了，Eureka不会立刻清理，依旧会对该微服务信息进行保存。属于CAP里面的AP分支

如果在Eureka Server的首页可以看到上述的那段提示信息，则说明Eureka进入了保护模式。

#### 2.为什么会产生Eureka自我保护机制？

为了防止，EurekaClient可以正常运行，但是与EurekaServer网络不通畅的情况下，EurekaServer不会立刻将EurekaClient服务剔除。

#### 3.什么是自我保护模式？

默认情况下，EurekaClient定时向EurekaServer发送心跳包，如果EurekaServer在一定时间内没有收到某个微服务实例的心跳，EurekaServer将会注销该实例（默认90秒）。但是当网络分区故障发生(延迟、卡顿、拥挤)时，微服务与EurekaServer之间无法正常通信，以上行为可能变得非常危险---因为微服务本身其实是健康的，**此时本不应该注销该微服务。**Eureka通过“自我保护模式”来解决这个问题---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），那么这个节点会进入自我保护模式。

**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。**它的设计哲学就是宁可保留错误的服务注册信息，也不盲目注销任何可能健康的服务实例。一句话：好死不如赖活着。使用自我保护模式，可以让Eureka集群更加的健壮、稳定。

#### 4.关闭自我保护

在7001和7002的application.yml的Eureka中添加配置

```
server:
  enable-self-preservation: false #关闭自我保护机制，保证不可用服务被及时剔除
  eviction-interval-timer-in-ms: 2000
```

修改8001和8002的eureka配置

```
eureka:
  client:
    #表示将自己注册进EurekaServer, 默认为true
    register-with-eureka: 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
    fetch-registry: true
    service-url:
      #defaultZone: http://localhost:7001/eureka #单机版
      defaultZone: http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka #集群版
  instance: # 在eureka界面展示时的名称
    instance-id: payment8001
    prefer-ip-address: true
    #Eureka客户端向服务端发送心跳的间隔时间。默认是30s
    lease-renewal-interval-in-seconds: 10 #改为10秒
    #Eureka服务端在收到最后一次心跳后等待的时间上线, 单位为秒(默认是90s), 超时将被剔除
    lease-expiration-duration-in-seconds: 20
```

```
#Eureka客户端向服务端发送心跳的间隔时间。默认是30s
lease-renewal-interval-in-seconds: 10 #改为10秒
#Eureka服务端在收到最后一次心跳后等待的时间上线, 单位为秒(默认是90s), 超时将被剔除
lease-expiration-duration-in-seconds: 20
```

#### eureka.instance.lease-expiration-duration-in-seconds

leaseExpirationDurationInSeconds, 表示eureka server至上一次收到client的心跳之后，等待下一次心跳的超时时间，在这个时间内若没收到下一次心跳，则将移除该instance。

默认为90秒

如果该值太大，则很可能将流量转发过去的时候，该instance已经不存活了。

如果该值设置太小了，则instance则很可能因为临时的网络抖动而被摘除掉。

该值至少应该大于leaseRenewalIntervalInSeconds

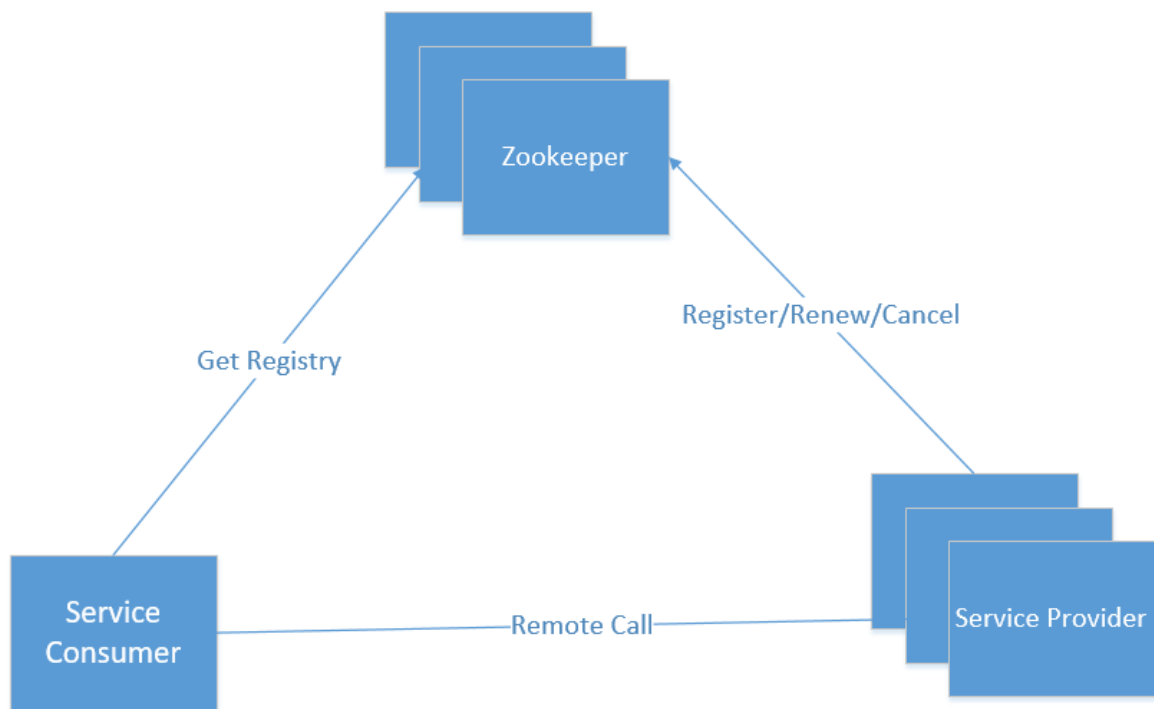
#### eureka.instance.lease-renewal-interval-in-seconds

leaseRenewalIntervalInSeconds, 表示eureka client发送心跳给server端的频率。如果在leaseExpirationDurationInSeconds后，server端没有收到client的心跳，则将摘除该instance。除此之外，如果该instance实现了HealthCheckCallback，并决定让自己unavailable的话，则该instance也不会接收到流量。

默认30秒



## 新架构, zookeeper注册中心



1. 下载zookeeper并解压。

2. 关闭防火墙

```
[echo@localhost jdk1.8.0_221]$ sudo systemctl stop firewalld
```

3. 可以直接尝试打开zookeeper的bin目录然后执行sudo ./zkServer.sh start 但是会报两个错

第一个: java\_home问题。

解决:

修改zkServer.sh, 在文件首部添加自己jdk环境中的java\_home

```
ZOOBIN="${BASH_SOURCE-$0}"
ZOOBIN="$(dirname "${ZOOBIN}")"
ZOOBINDIR="$(cd "${ZOOBIN}"; pwd)"
export JAVA_HOME=/software/jdk1.8.0_221
if [ -e "$ZOOBIN/../../libexec/zkEnv.sh" ]; then
    . "$ZOOBINDIR/../../libexec/zkEnv.sh"
else
```

第二个: 无法创建目录

```
ZooKeeper JMX enabled by default
Using config: /software/apache-zookeeper-3.5.8-bin/bin/../../conf/zoo.cfg
grep: /software/apache-zookeeper-3.5.8-bin/bin/../../conf/zoo.cfg: 没有那个文件或目录
grep: /software/apache-zookeeper-3.5.8-bin/bin/../../conf/zoo.cfg: 没有那个文件或目录
mkdir: 无法创建目录"": 没有那个文件或目录
Starting zookeeper ... FAILED TO START
```

解决:

将zookeeper的conf目录下的zoo\_sample.cfg 文件改名。

mv zoo\_sample.cfg 文件zoo.cfg

#### 4.启动之后可以通过zkCli.sh连接查看zookeeper的信息

```
[echo@localhost bin]$ ./zkCli.sh
Connecting to localhost:2181
2021-04-11 15:38:09,308 [myid:] - INFO [main:Environment@109] - Client environment: zookeeper.version=3.5.8- f439ca583e70862c3068a1f2a7d4d068eec33315, built on 05/04/2020 15:07 GMT
2021-04-11 15:38:09,310 [myid:] - INFO [main:Environment@109] - Client environment: host.name=localhost
2021-04-11 15:38:09,311 [myid:] - INFO [main:Environment@109] - Client environment: java.version=1.8.0_221
2021-04-11 15:38:09,313 [myid:] - INFO [main:Environment@109] - Client environment: java.vendor=Oracle Corporation
2021-04-11 15:38:09,313 [myid:] - INFO [main:Environment@109] - Client environment: java.home=/software/jdk1.8.0_221/jre
2021-04-11 15:38:09,313 [myid:] - INFO [main:Environment@109] - Client environment: java.class.path=/software/apache-zookeeper-3.5.8-bin/bin/../zookeeper-server/target/classes:/software/apache-zookeeper-3.5.8-bin/bin/../build/classes:/software/apache-zookeeper-3.5.8-bin/bin/../zookeeper-server/target/lib/*.jar:/software/apache-zookeeper-3.5.8-bin/bin/../build/lib/*.jar:/software/apache-zookeeper-3.5.8-bin/bin/../lib/zookeeper-jute-3.5.8.jar:/software/apache-zookeeper-3.5.8-bin/bin/../lib/zookeeper-3.5.8.jar:/software/apache-zookeeper-3.5.8-bin/bin/../lib/slf4j-log4j12-1.7.25.jar:/software/apache-zookeeper-3.5.8-bin/bin/../lib/slf4j-
```

#### 5.创建模块cloud-provider-payment-8004

6.引入pom文件，其实和其他的服务提供方的pom文件类似，只是将eureka依赖更改成了zookeeper的依赖

```
<dependencies>
<!--      springboot整合zookeeper客户端，作为注册中心-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zookeeper-discovery</artifactId>
    </dependency>
<!--      引入自定义的公共模块-->
    <dependency>
        <groupId>com.echo</groupId>
        <artifactId>cloud-api-common</artifactId>
        <version>${project.version}</version>
    </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.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid-spring-boot-starter</artifactId>
        <version>1.1.10</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
</dependencies>
```

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<!--      springboot热部署-->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <!--      runtime表示被依赖项目无需参与项目的编译，不过后期的测试和运行
周期需要其参与。与compile相比，跳过编译而已，-->
  <scope>runtime</scope>
  <!--      <optional>true</optional>表示两个项目之间依赖不传递；不设置optional或者optional是false，表示传递依赖。-->
  <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>

```

7.修改yml，同样的也是将eureka的配置去掉，然后更改为zookeeper的配置

```

server:
  port: 8004

spring:
  application:
    name: cloud-payment-service
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource          #当前数据源操作类型
    driver-class-name: org.gjt.mm.mysql.Driver           #mysql驱动包，用的是spring的
    jdbc
    url: jdbc:mysql://localhost:3306/db2019?
    useUnicode=true&characterEncoding=utf-8&useSSL=false
    username: root
    password: s814466057
    #zookeeper注册中心
  cloud:
    zookeeper:
      connect-string: 192.168.45.133:2181    #安装zookeeper的服务器的地址

mybatis:
  mapper-locations: classpath:mapper/*.xml
  type-aliases-package: com.echo.pojo        #配置所有别名的包的位置，意思就是
  com.echo.pojo.Payment可以直接写

```

8.主启动类

```
@SpringBootApplication
@EnableDiscoveryClient //该注解用于向使用consul或者zookeeper作为注册中心时注册服务
public class PaymentApplication {
    public static void main(String[] args) {
        SpringApplication.run(PaymentApplication.class,args);
    }
}
```

9.启动该服务。

启动该服务之后将会使其注册进zookeeper中，但是这里要注意，如果zookeeper的版本和pom.xml中的版本不一致，会出现错误。可以采用在pom中排除，重新引入。

启动成功后可以在zookeeper的zkCli.sh中查看信息

```
zk: localhost:2181(CONNECTING) 0] ls
ls [-s] [-w] [-R] path
zk: localhost:2181(CONNECTED) 1] ls /
[zookeeper]
zk: localhost:2181(CONNECTED) 2] get /zookeeper
Path must start with / character
zk: localhost:2181(CONNECTED) 3] ls /
[services-zookeeper]
zk: localhost:2181(CONNECTED) 4] ls /services
[cloud-payment-service]
zk: localhost:2181(CONNECTED) 5] ls /services/cloud-payment-service
93871ac9-2bf1-48c7-ae06-4aaa77ce69fc
zk: localhost:2181(CONNECTED) 6] ls /services/cloud-payment-service/93871ac9-2bf1-48c7-ae06-4aaa77ce69fc
[]
zk: localhost:2181(CONNECTED) 7] get /services/cloud-payment-service/93871ac9-2bf1-48c7-ae06-4aaa77ce69fc
{"name": "cloud-payment-service", "id": "93871ac9-2bf1-48c7-ae06-4aaa77ce69fc", "address": "localhost", "port": 8004, "sslPort": null, "payload": {"@class": "org.springframework.cloud.zookeeper.discovery.ZookeeperInstance", "id": "application-1", "name": "cloud-payment-service", "metadata": {}}, "registrationTimeUTC": 1618126814952, "serviceType": "DYNAMIC", "uriSpec": {"parts": [{"value": "scheme", "variable": true}, {"value": "://", "variable": false}, {"value": "address", "variable": true}, {"value": ":", "variable": false}, {"value": "port", "variable": true}]}}
```

出现了服务

该服务提供方的名字

该服务提供方的一个主机实例，这是实例的ID

该实例的所有信息

10.zookeeper的服务节点是临时的还是永久的

经过测试zookeeper的服务节点是临时的，当关掉服务之后，在一定的时间内没有心跳包了，zookeeper就会直接杀掉服务，服务重启后，zookeeper就会重新拉取该服务。

## 服务消费方入驻zookeeper

1.同理新建一个cloud-consumerzk-order-80,

其余操作与服务提供方类似

2.注意点

controller中的url，即调用服务提供方的url，要以zookeeper中注册的服务名为准，而且大小写敏感，服务提供方注册进zookeeper的服务名字是什么，那么url中就应该写什么

```
[zk: localhost:2181(CONNECTED) 10] ls /services
[cloud-order-service, cloud-payment-service]
[zk: localhost:2181(CONNECTED) 11] ls /services/cloud-order-service
cloud-order-service cloud-payment-service
[zk: localhost:2181(CONNECTED) 11] ls /services/cloud-order-service
8fd4f648-a988-4008-96ee-9482c5cdc68e
[zk: localhost:2181(CONNECTED) 12] ls /services/cloud-order-service/8fd4f648-a988-4008-96ee-9482c5cdc68e
[]
[zk: localhost:2181(CONNECTED) 13] get /services/cloud-order-service/8fd4f648-a988-4008-96ee-9482c5cdc68e
{"name": "cloud-order-service", "id": "8fd4f648-a988-4008-96ee-9482c5cdc68e", "address": "localhost", "port": 80, "sslPort": null, "payload": {"@class": "org.springframework.cloud.zookeeper.discovery.ZookeeperInstance", "id": "application-1", "name": "cloud-order-service", "metadata": {}}, "registrationTimeUTC": 1618128926605, "serviceType": "DYNAMIC", "uriSpec": {"parts": [{"value": "scheme", "variable": true}, {"value": "://", "variable": false}, {"value": "address", "variable": true}, {"value": ":", "variable": false}, {"value": "port", "variable": true}]}}
```

@RestController

```
@Slf4j
public class OrderController {
    //public static final String PAYMENT_URL = "http://localhost:8001";    //单机版
    //注意这里和
    public static final String PAYMENT_URL = "http://cloud-payment-service";

    @Autowired
    private RestTemplate restTemplate;

    @GetMapping("/consumer/payment/add")
    public CommonResult<Payment> create(Payment payment){
        //参数说明: url,post请求体, 请求返回的结果映射
        return restTemplate.postForObject(PAYMENT_URL +
"/payment/add",payment,CommonResult.class);
    }

    @GetMapping("/consumer/payment/get/{id}")
    public CommonResult<Payment> getPayment(@PathVariable("id")Long id){
        return restTemplate.getForObject(PAYMENT_URL + "/payment/get/" +
id,CommonResult.class);
    }
}
```

## Consul

### 1.什么是Consul

consul是一套开源的分布式服务发现和配置管理系统，由HashiCorp公司用Go语言开发。

Consul提供了微服务系统中的服务治理、配置中心、控制总线等功能。这些功能中的每一个都可以根据需要单独使用，也可以一起使用以构建全方位的服务网格，总之，Consul提供了一套完整的服务网格解决方案。

### 2.Consul的功能

服务发现：提供HTTP和DNS两种发现方式

健康检测：支持多种方式


KV存储：Key Value的存储方式

多数据中心：Consul支持多数据中心

可视化Web界面

### 3.使用

下载完成之后只有一个exe

此电脑 > 软件 (D:) > consul		
名称	修改日期	
 consul.exe	2019/9/13 3:46	

在命令行使用

## Windows PowerShell

```
PS D:\consul> .\consul.exe version
Consul v1.6.1
Protocol 2 spoken by default, understands 2 to 3 (agents)
PS D:\consul>
```

命令行启动

```
PS D:\consul> .\consul.exe agent -dev
```

```
PS D:\consul> .\consul.exe agent -dev
==> Starting Consul agent...
  Version: 'v1.6.1'
  Node ID: 'ladfc3fd-3d31-da18-d523-230cbdcdeb85'
  Node name: 'DESKTOP-QJG9A26'
  Datacenter: 'dc1' (Segment: '<all>')
  Server: true (Bootstrap: false)
  Client Addr: [127.0.0.1] (HTTP: 8500, HTTPS: -1, gRPC: 8502, DNS: 8600)
  Cluster Addr: 127.0.0.1 (LAN: 8301, WAN: 8302)
  Encrypt: Gossip: false, TLS-Outgoing: false, TLS-Incoming: false, Auto-Encrypt-TLS: false

==> Log data will now stream in as it occurs:

2021/04/11 17:01:06 [DEBUG] agent: Using random ID "ladfc3fd-3d31-da18-d523-230cbdcdeb85" as node ID
2021/04/11 17:01:06 [DEBUG] tlsutil: Update with version 1
2021/04/11 17:01:06 [DEBUG] tlsutil: OutgoingRPCWrapper with version 1
2021/04/11 17:01:06 [INFO] raft: Initial configuration (index=1): [{Suffrage:Voter ID:ladfc3fd-3d31-da18-d523-230cbdcdeb85 Address:127.0.0.1:8300}]
2021/04/11 17:01:06 [INFO] raft: Node at 127.0.0.1:8300 [Follower] entering Follower state (Leader: "")
2021/04/11 17:01:06 [INFO] serf: EventMemberJoin: DESKTOP-QJG9A26.dc1 127.0.0.1
2021/04/11 17:01:06 [INFO] serf: EventMemberJoin: DESKTOP-QJG9A26 127.0.0.1
2021/04/11 17:01:06 [INFO] consul: Adding LAN server DESKTOP-QJG9A26 (Addr: tcp/127.0.0.1:8300) (DC: dc1)
2021/04/11 17:01:06 [INFO] consul: Handled member-join event for server "DESKTOP-QJG9A26.dc1" in area "wan"
2021/04/11 17:01:06 [INFO] agent: Started DNS server 127.0.0.1:8600 (udp)
2021/04/11 17:01:06 [INFO] agent: Started DNS server 127.0.0.1:8600 (tcp)
2021/04/11 17:01:06 [INFO] agent: Started HTTP server on 127.0.0.1:8500 (tcp)
2021/04/11 17:01:06 [INFO] agent: Started gRPC server on 127.0.0.1:8502 (tcp)
2021/04/11 17:01:06 [INFO] agent: started state syncer
==> Consul agent running!
2021/04/11 17:01:06 [WARN] raft: Heartbeat timeout from "" reached, starting election
2021/04/11 17:01:06 [INFO] raft: Node at 127.0.0.1:8300 [Candidate] entering Candidate state in term 2
2021/04/11 17:01:06 [DEBUG] raft: Votes needed: 1
2021/04/11 17:01:06 [DEBUG] raft: Vote granted from ladfc3fd-3d31-da18-d523-230cbdcdeb85 in term 2. Tally: 1
```

访问可视化界面

Service	Health Checks	Tags
consul	✓ 1	

如果访问出来没有加载，记得在power shell那里按一下ctrl + c，可能是阻塞了

## 服务提供方入驻Consul

- 1.新建一个cloud-provider-payment-8006
- 2.pom中修改一个consul注册中心的依赖

```
<!-- consul服务注册中心-->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
```

### 3.application.yml中配置一下consul

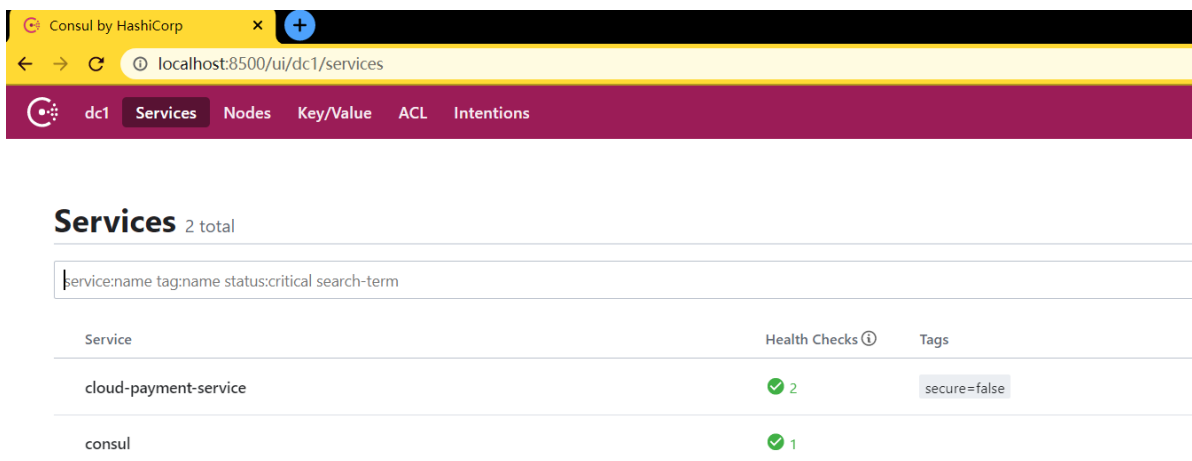
```
server:
  port: 8006

spring:
  application:
    name: cloud-payment-service
    ###consul注册中心地址###
  cloud:
    consul:
      host: localhost
      port: 8500
      discovery:
        service-name: ${spring.application.name}
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource #当前数据源操作类型
    driver-class-name: org.gjt.mm.mysql.Driver #mysql驱动包，用的是spring的
    jdbc
      url: jdbc:mysql://localhost:3306/db2019?
      useUnicode=true&characterEncoding=utf-8&useSSL=false
      username: root
      password: s814466057
```

### 4.主启动类

```
@SpringBootApplication
@EnableDiscoveryClient //该注解用于向使用consul或者zookeeper作为注册中心时注册服务
public class PaymentApplication {
    public static void main(String[] args) {
        SpringApplication.run(PaymentApplication.class,args);
    }
}
```

### 5.启动完成后可以在界面中看到



The screenshot shows the Consul UI interface. The top navigation bar includes tabs for Services, Nodes, Key/Value, ACL, and Intentions. The main content area is titled 'Services 2 total' and contains a search bar and a table of registered services.

Service	Health Checks ⓘ	Tags
cloud-payment-service	✓ 2	secure=false
consul	✓ 1	

## 服务消费方入驻Consul

1.新建一个cloud-consumerconsul-order-80

2.其余配置之类的和服务提供方类似。

## 三个注册中心的异同

1.CAP

C:Consistency 强一致性

A:Availability 可用性

P:Patition tolerance 分区容错性

CAP理论可百度

组件名	语言	CAP	服务健康检测	对外暴露接口	SpringCloud集成
Eureka	Java	AP	可配支持	HTTP	已集成
Consul	Go	CP	支持	HTTP/DNS	已集成
Zookeeper	Java	CP	支持	客户端	已集成





