Eureka项目布置的流程

1、环境准备

1.1本地环境准备

（1）确保本地已经安装好mvn仓库，如果没有安装，到官网进行下载https://maven.apache.org/download.cgi

下载完成之后配置C:\Program Files\apache-maven-3.5.4\conf路径下的settings.xml文件。

settings.xml文件需要配置的地方有三个，本地仓库地址、镜像下载地址、代理（仅内网需要）。

本地仓库地址一般情况下默认就好，如果有特殊需要可以按照下面的方式进行配置：

```
<localRepository>${user.home}/.m2/repository</localRepository>
```

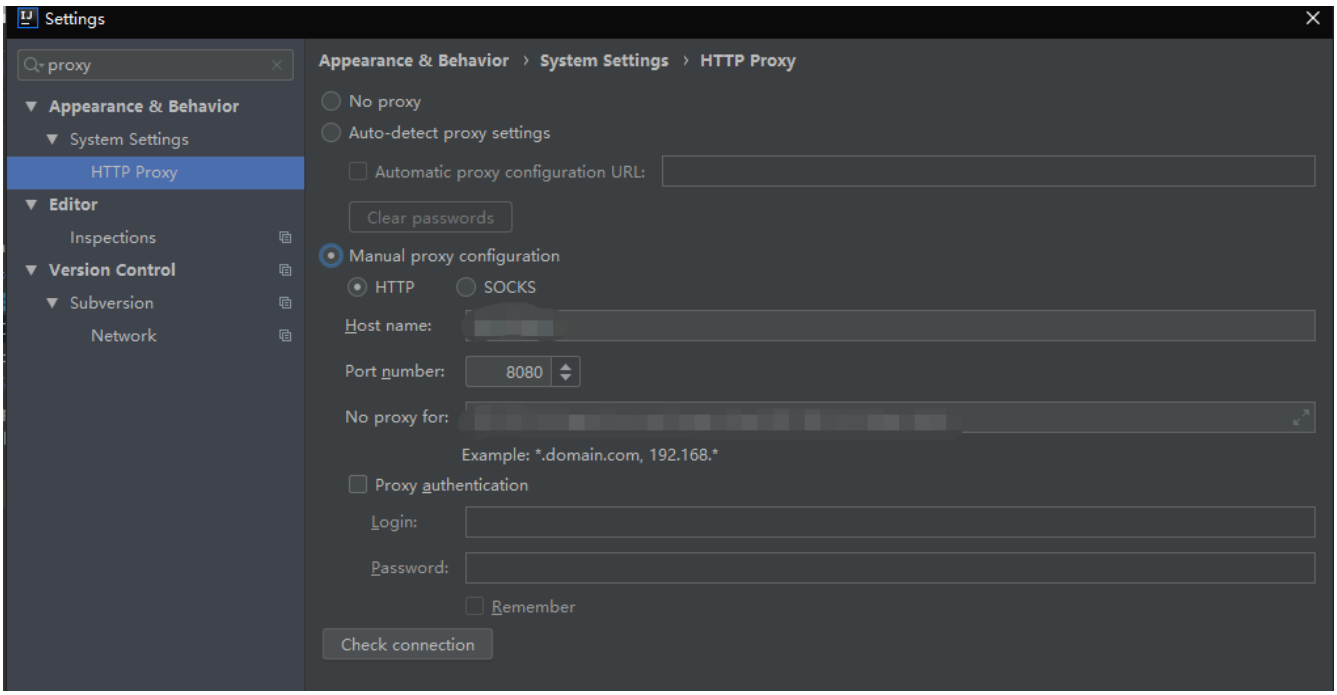
镜像的下载地址可以填写阿里云的镜像仓库：

```
<mirror>
 <id>alimaven</id>
 <name>aliyun maven</name>
 <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
 <mirrorOf>central</mirrorOf>
</mirror>
```

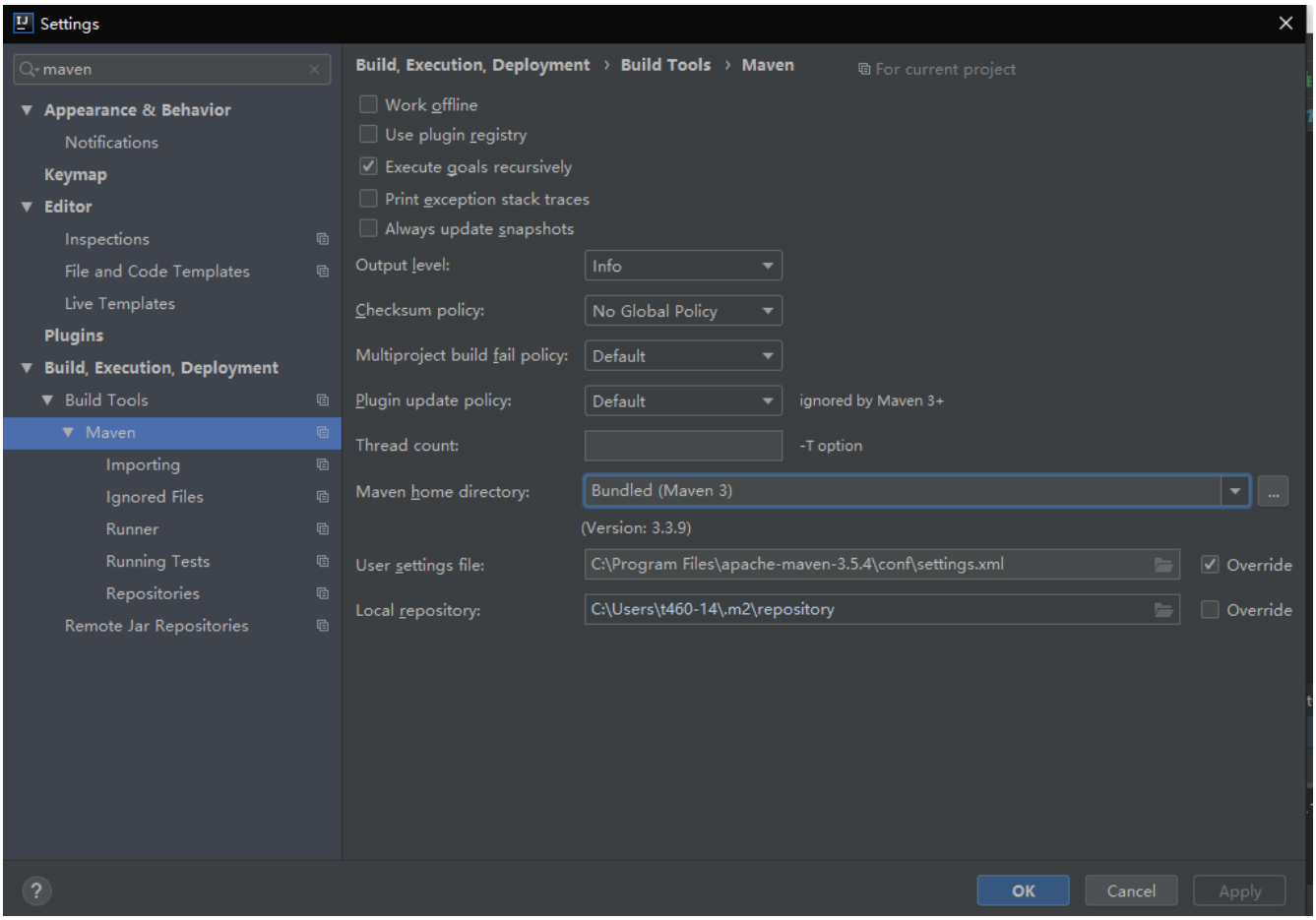
1.2idea环境准备

由于笔者使用的是idea工具进行的开发设计，所以这里只讲一下idea的环境准备。

idea的代理开启（仅内网需要）：



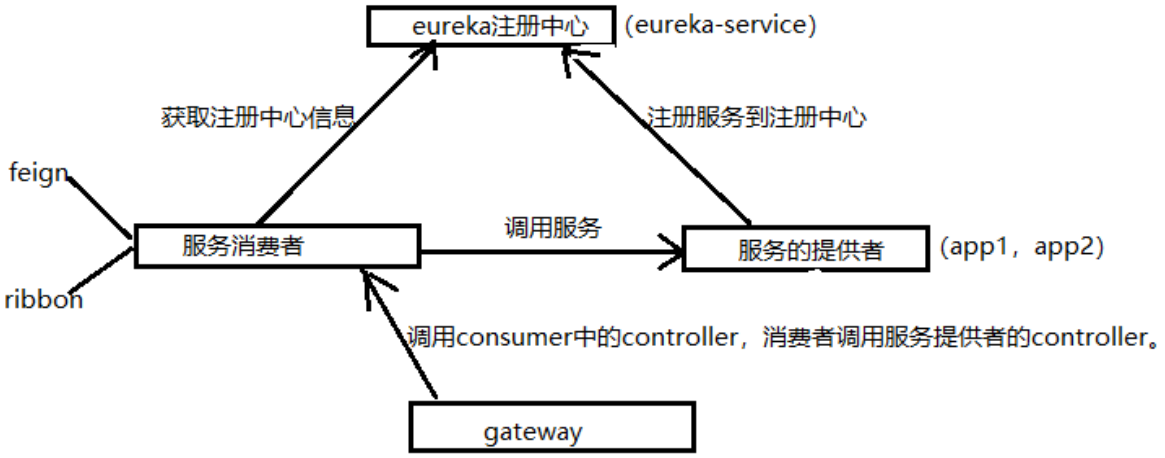
idea的maven仓库的配置：



user settings file里面填的是本地的settings.xml地址，local repository填的是本地仓库地址。

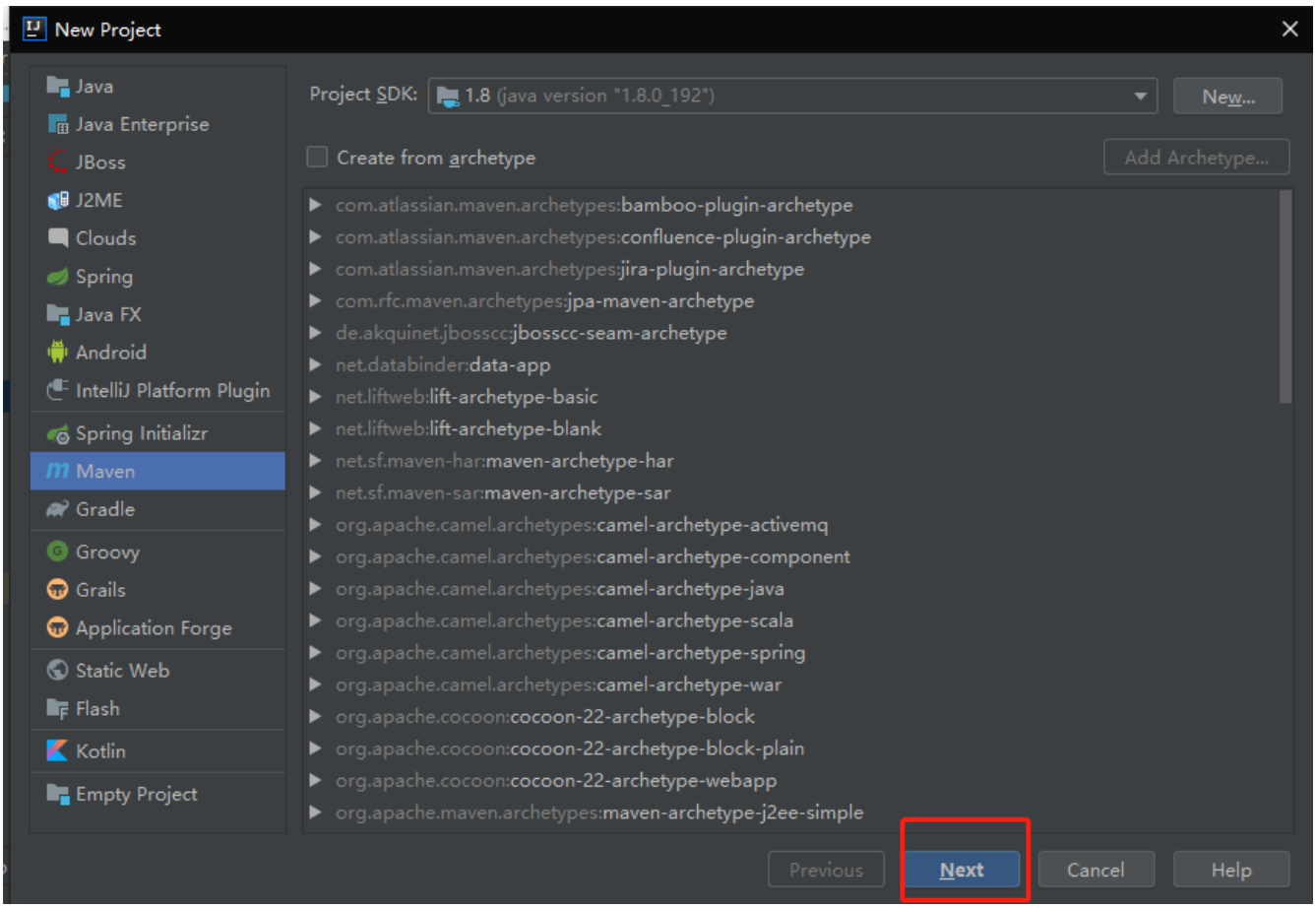
2.搭建eureka的三个基本组件

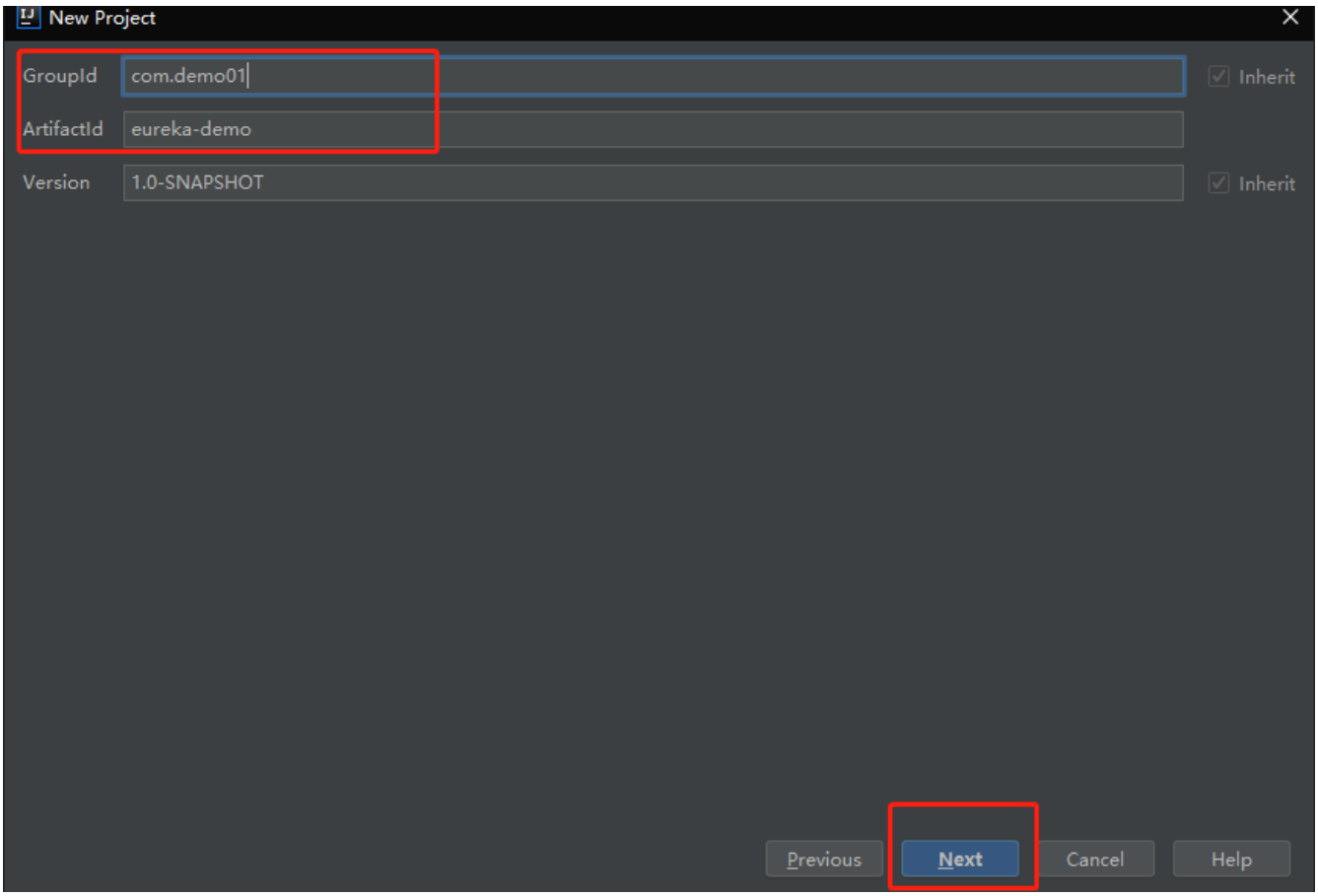
eureka的整体框架图如下：

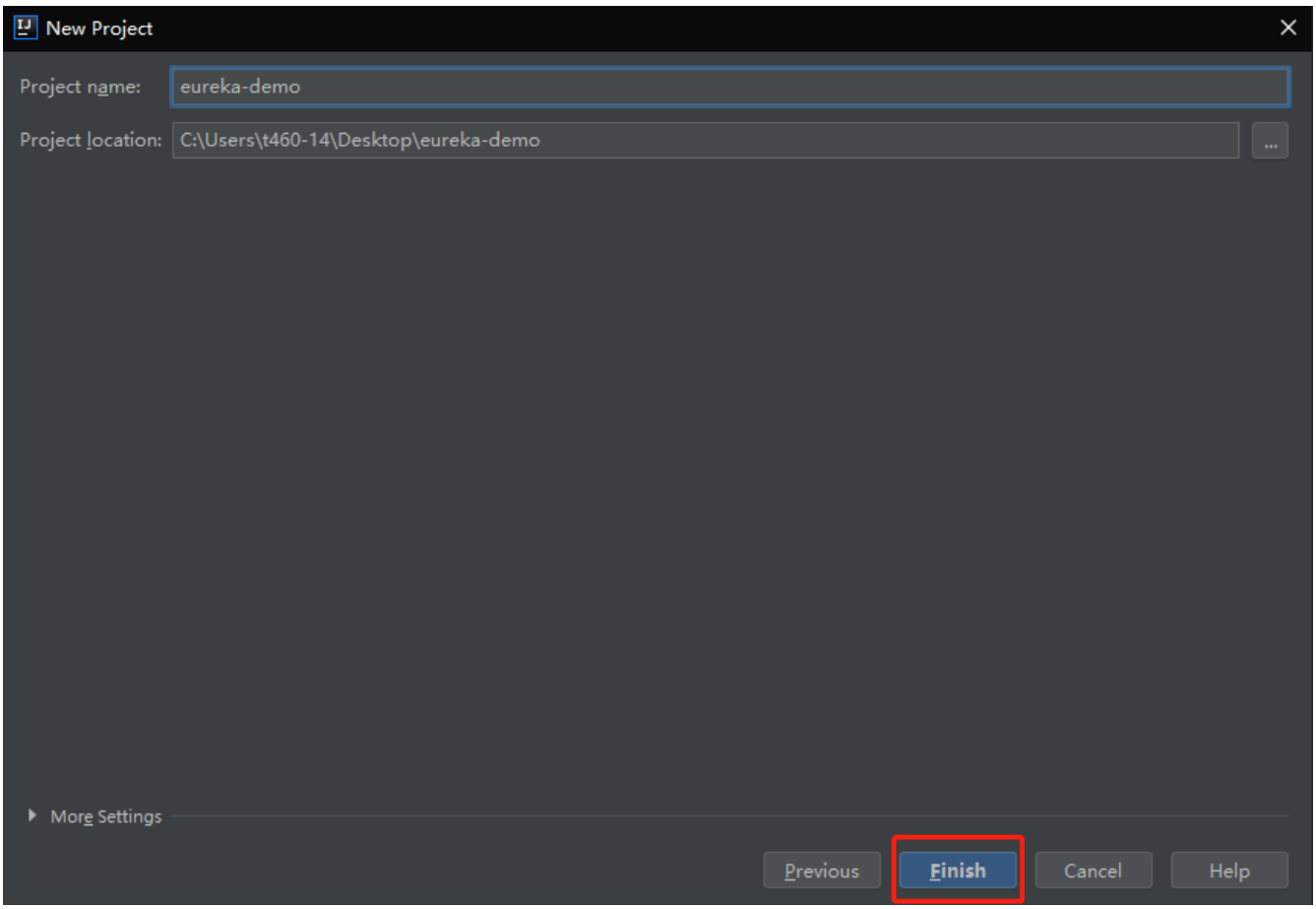


2.1搭建主框架

首先建立一个maven项目，流程如下图：



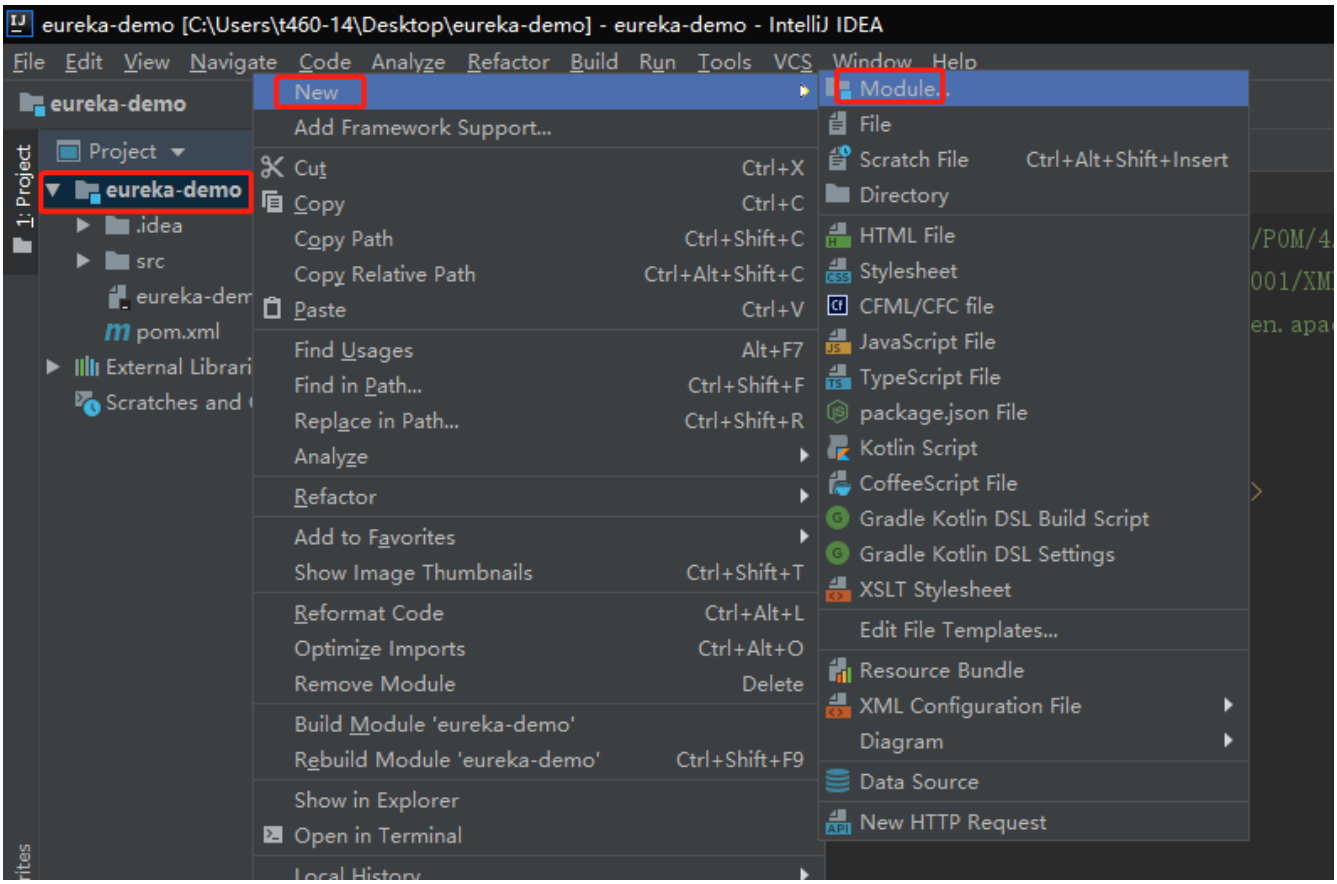




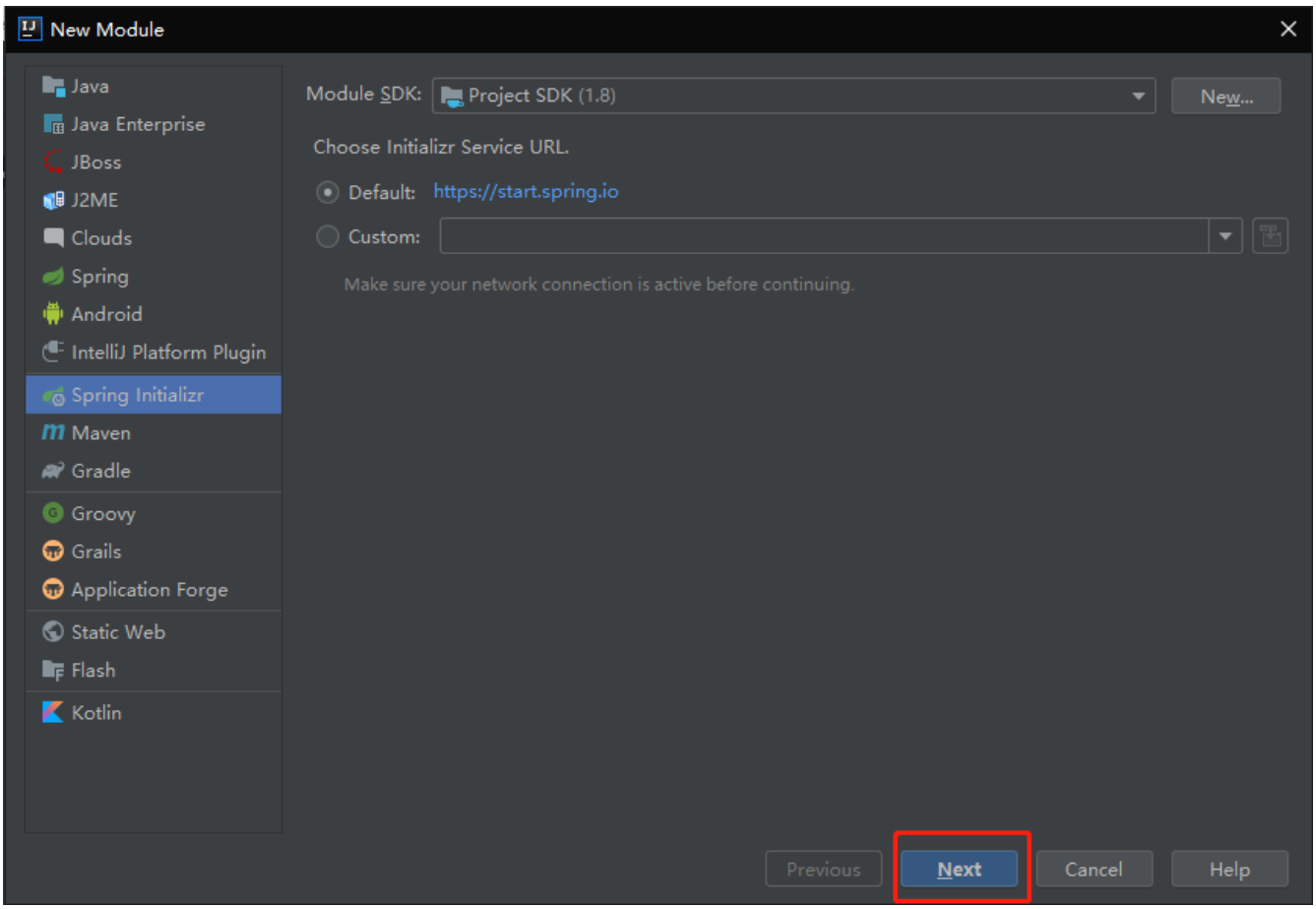
至此eureka的主框架就建立好了，剩下的所有的子module都是建立在这个里面的。

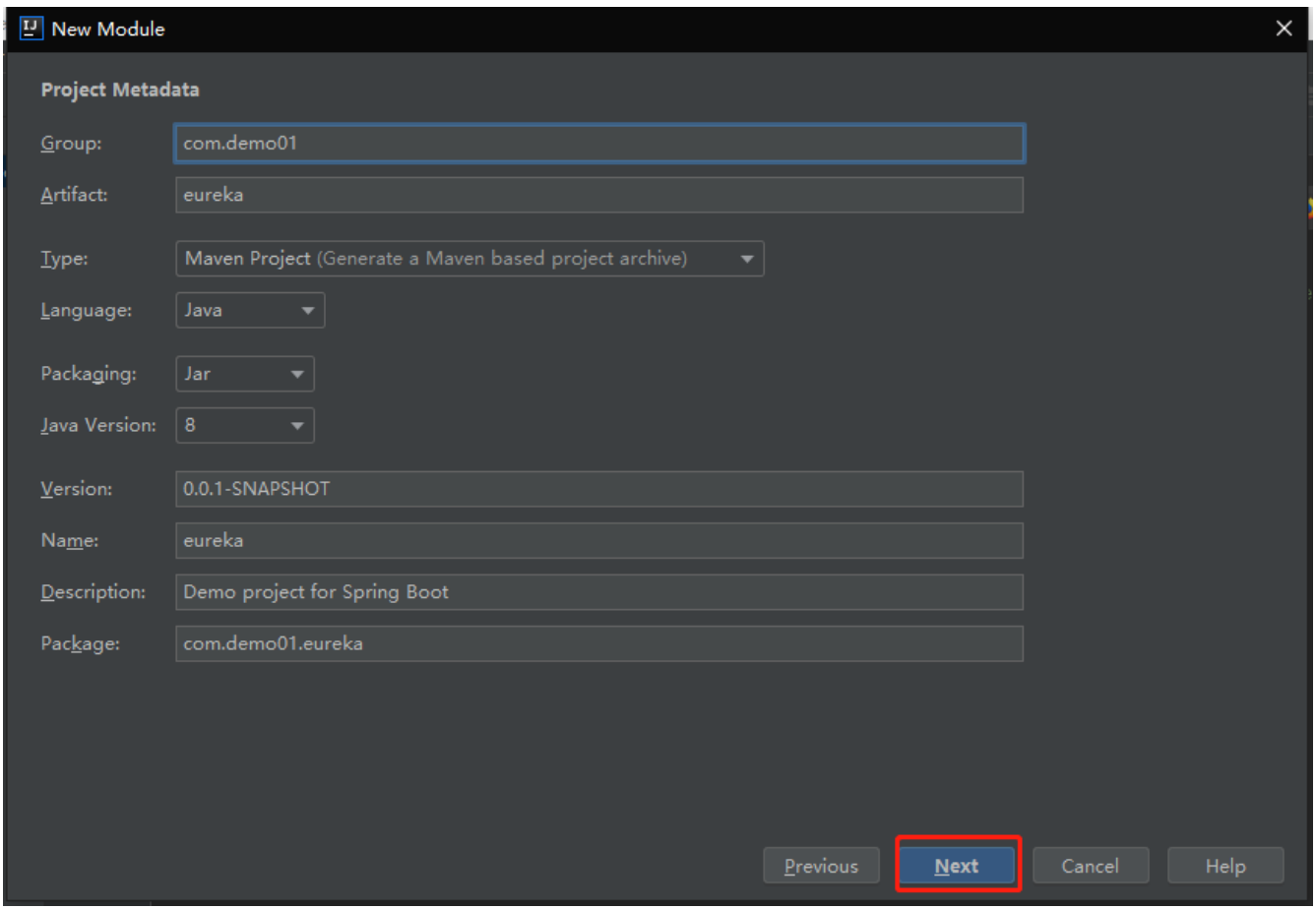
2.2搭建eureka注册中心

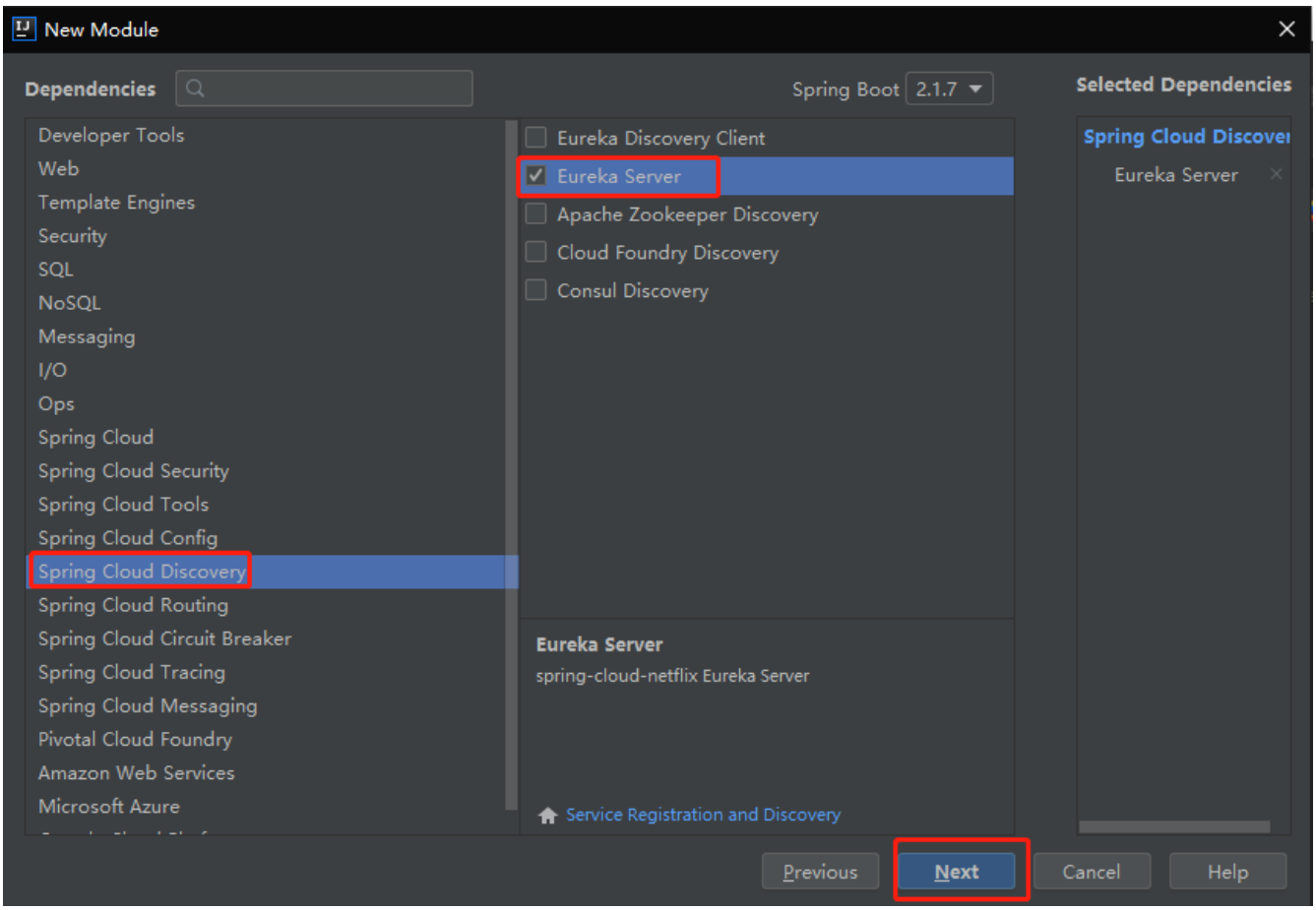
建好的maven项目上右键-----》new-------》Module

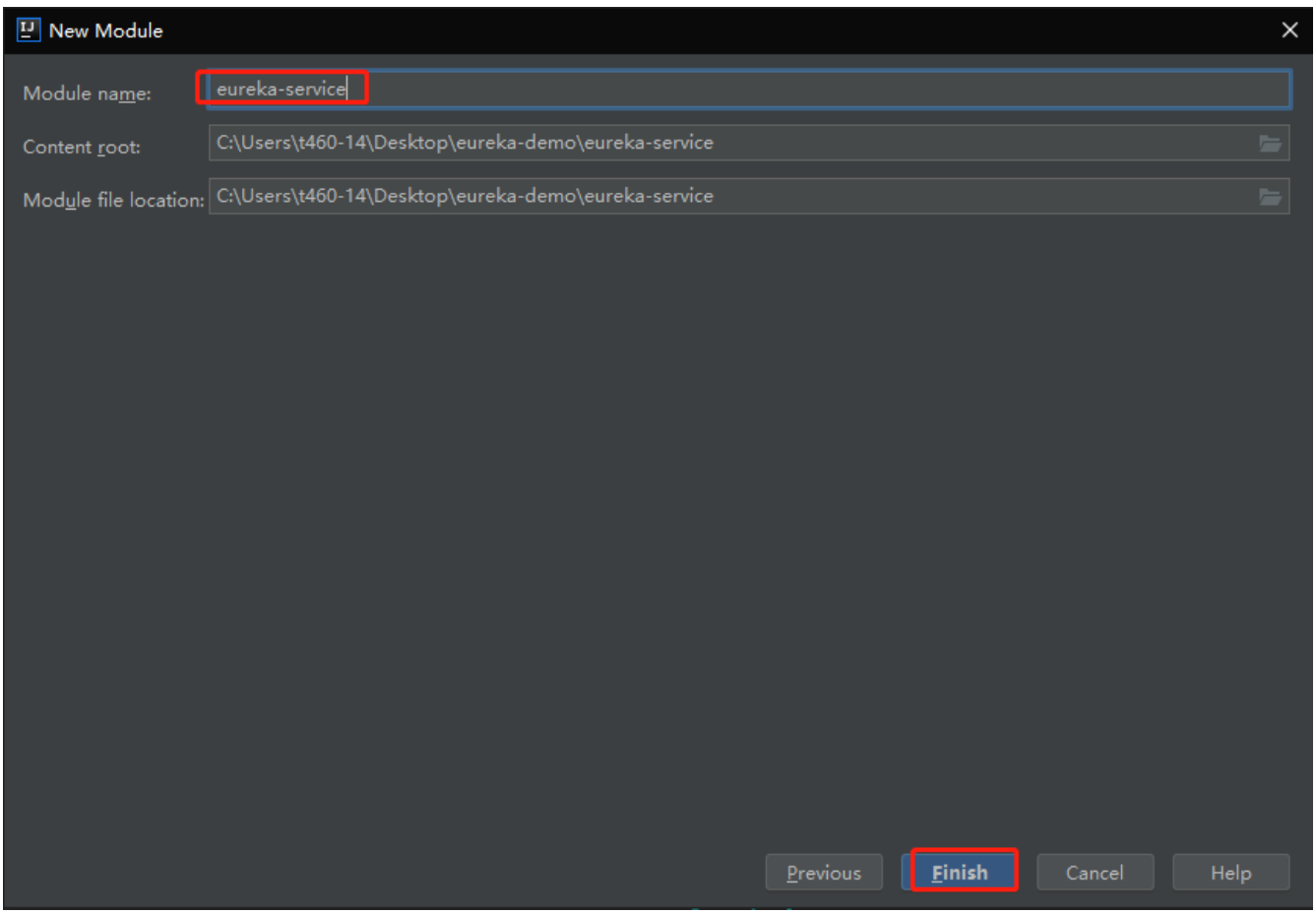


选择Spring Initializr


New Module
Java
Java Enterprise
JBoss
J2ME
Clouds
Spring
Android
IntelliJ Platform Plugin
Spring Initializr
Maven
Gradle
Groovy
Grails
Application Forge
Static Web
Flash
Kotlin
Module SDK:
Project SDK (1.8)
New...
Choose Initializr Service URL.
Default:
https://start.spring.io
Custom:
Make sure your network connection is active before continuing.
Previous
Next
Cancel
Help


New Module
Project Metadata
Group:
com.demo01
Artifact:
eureka
Type:
Maven Project (Generate a Maven based project archive)
Language:
Java
Packaging:
Jar
Java Version:
8
Version:
0.0.1-SNAPSHOT
Name:
eureka
Description:
Demo project for Spring Boot
Package:
com.demo01.eureka
Previous
Next
Cancel
Help



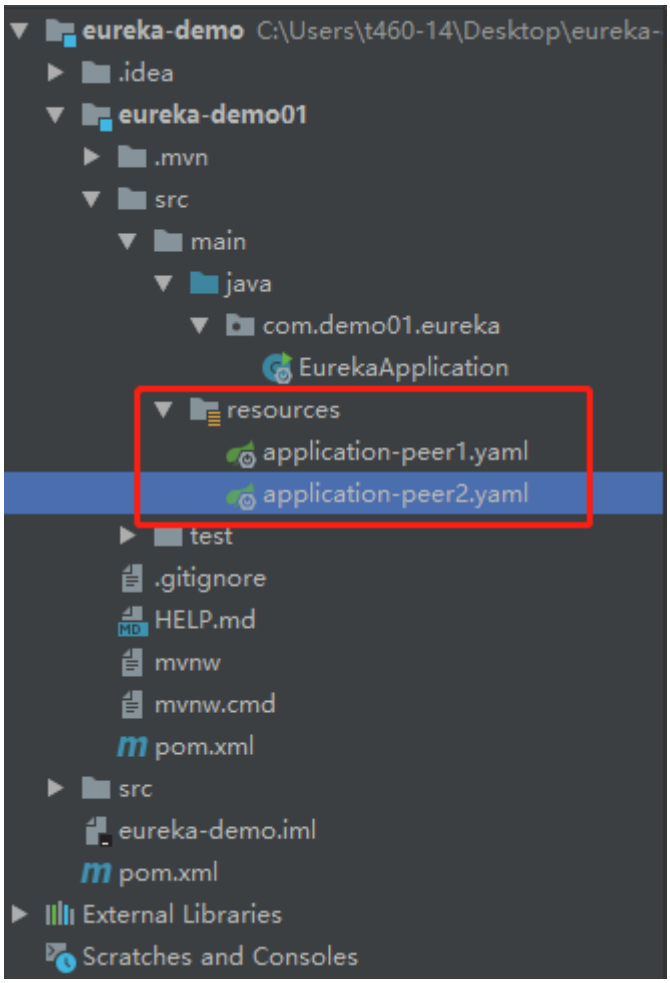


由于一般情况下都是高可用的注册中心，所以这里笔者采用的是双节点的注册中心。

需要到hosts文件末尾加入如下部分：(hosts文件的位置在：C:\Windows\System32\drivers\etc\hosts)

```
127.0.0.1  peer1
127.0.0.1  peer2
```

首先将原先resources中的application.properties文件改为application-peer1.yaml文件，然后复制改之后的文件，重命名为application-peer2.yaml。



对这两个配置文件分别进行设置：

application-peer1.yaml

```
server:
  port: 1111  #服务端口号
eureka:
  client:
    service-url:
      #相互注册，组成一个集群，实现高可用
      defaultZone: http://peer2:2222/eureka/
  instance:
    #主机名
    hostname: peer1
spring:
  application:
    #服务名称
```

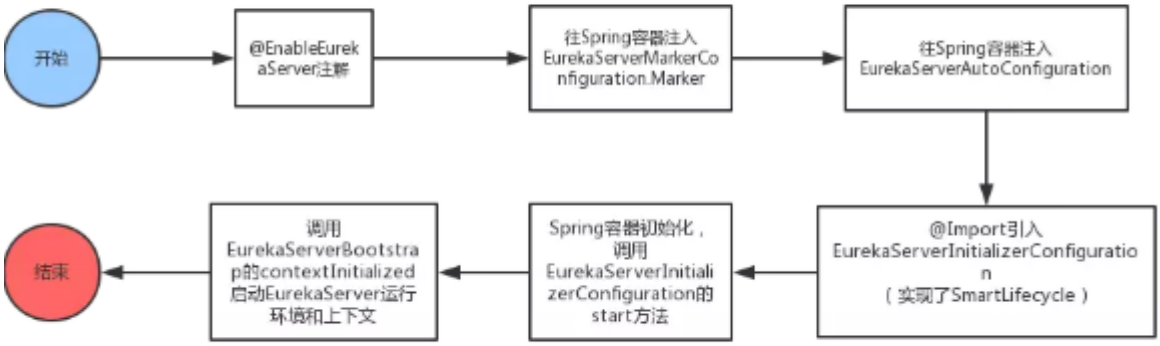
```
    name: eureka-peer-server
```

application-peer2.yaml

```
server:
  port: 2222  #服务端口号
eureka:
  client:
    service-url:
      #相互注册，组成一个集群，实现高可用
      defaultZone: http://peer1:1111/eureka/
  instance:
    #主机名
    hostname: peer2
spring:
  application:
    #服务名称
    name: eureka-peer-server
```

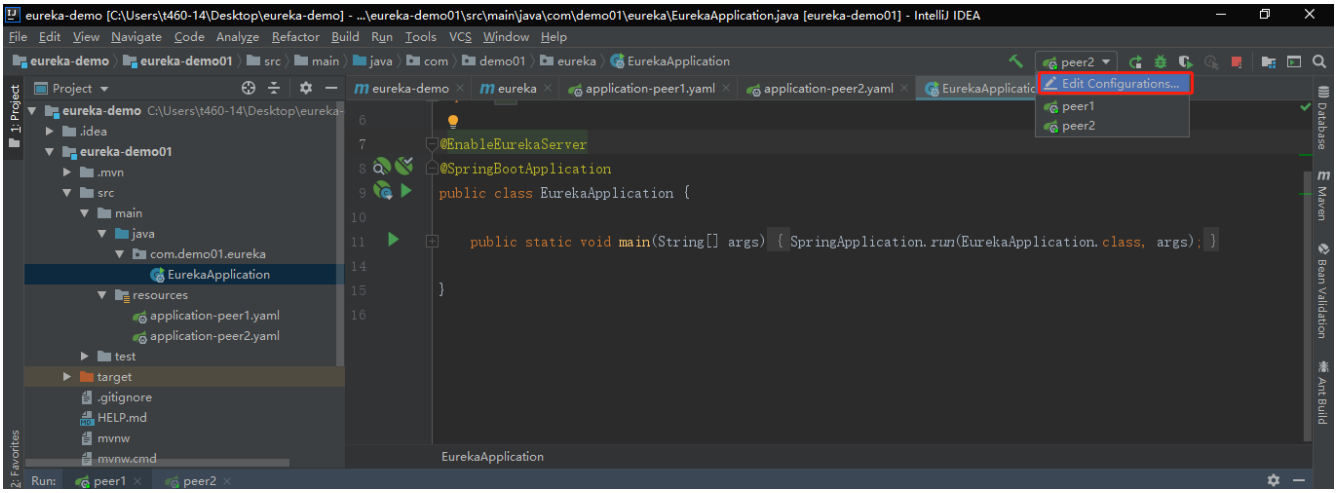
配置完成需要到EurekaApplication.java里面添加注解，@EnableEurekaServer，所用是为了：

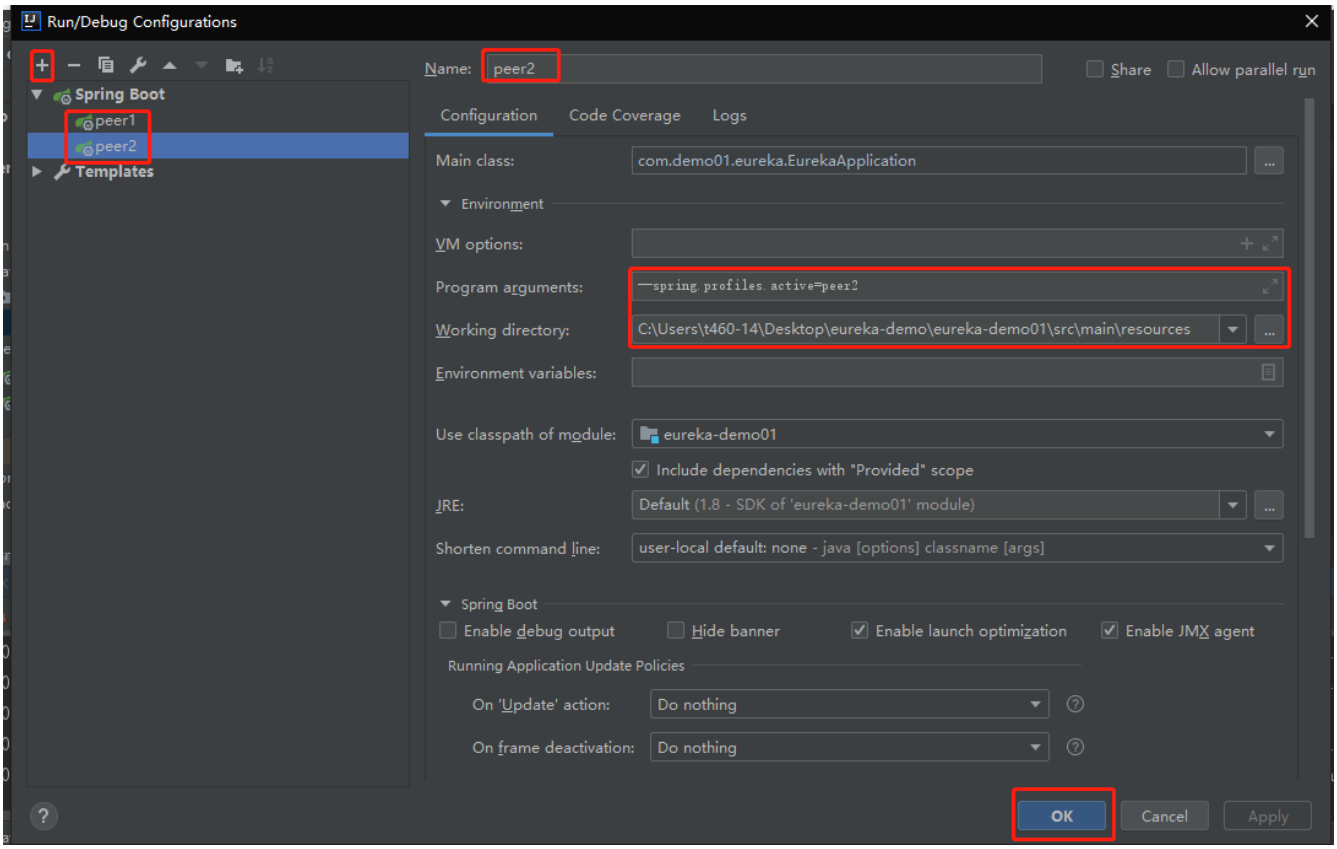
相关流程图如下：



Eureka源码剖析（一）注解方式启动Eureka Server.png

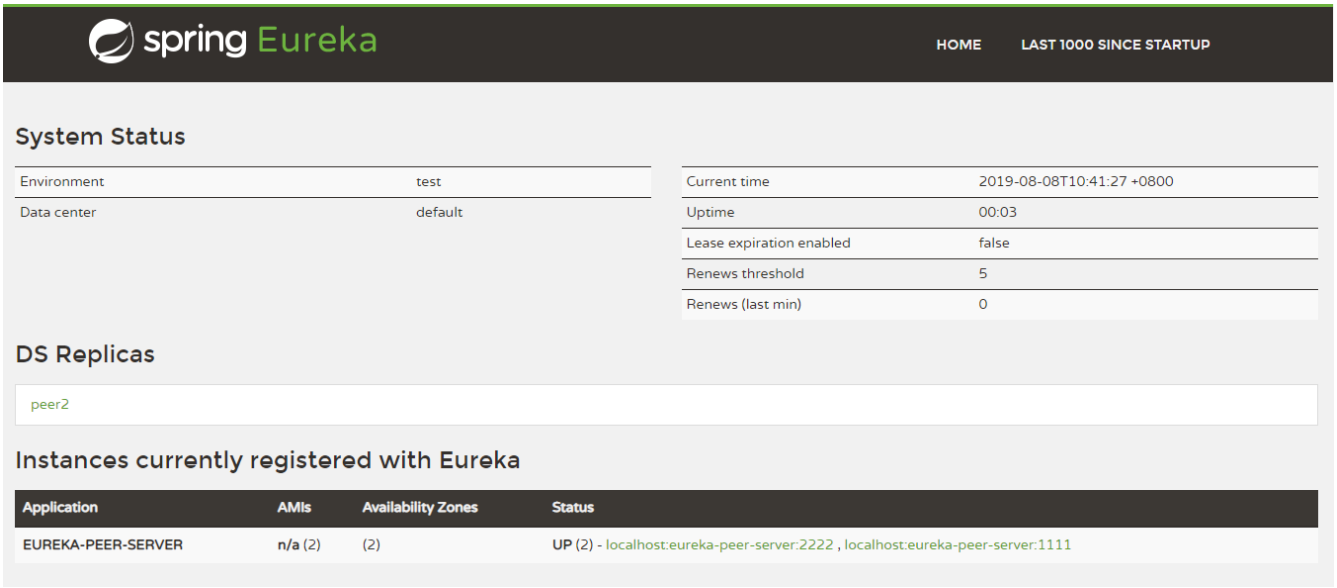
接下来启动，启动的流程如下：





做完上面的配置，然后分别启动peer1和peer2，peer1启动的时候会报错，不用理会，继续启动peer2就不会报错了。

启动完成之后访问localhost:1111 会出现下面的界面：

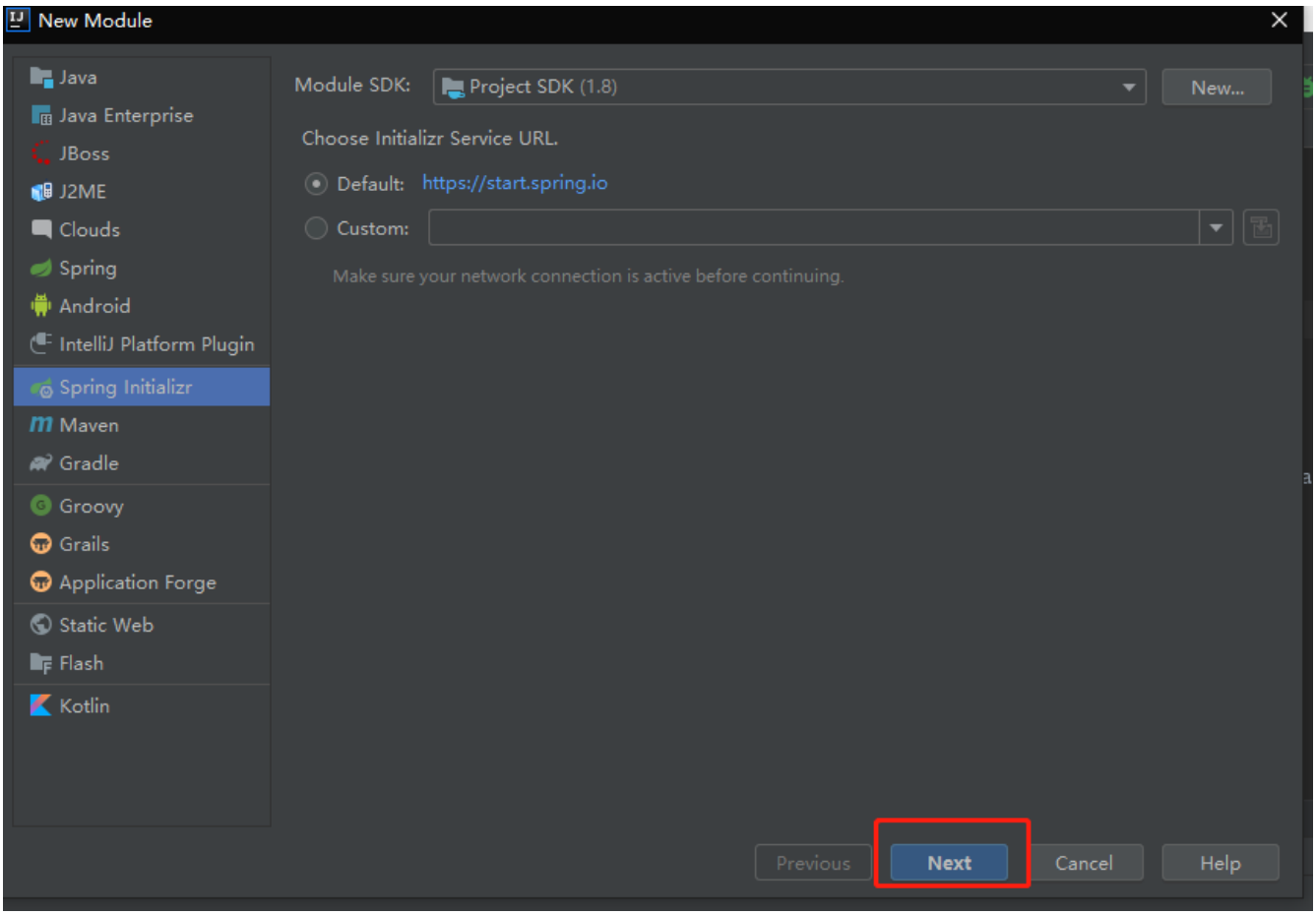


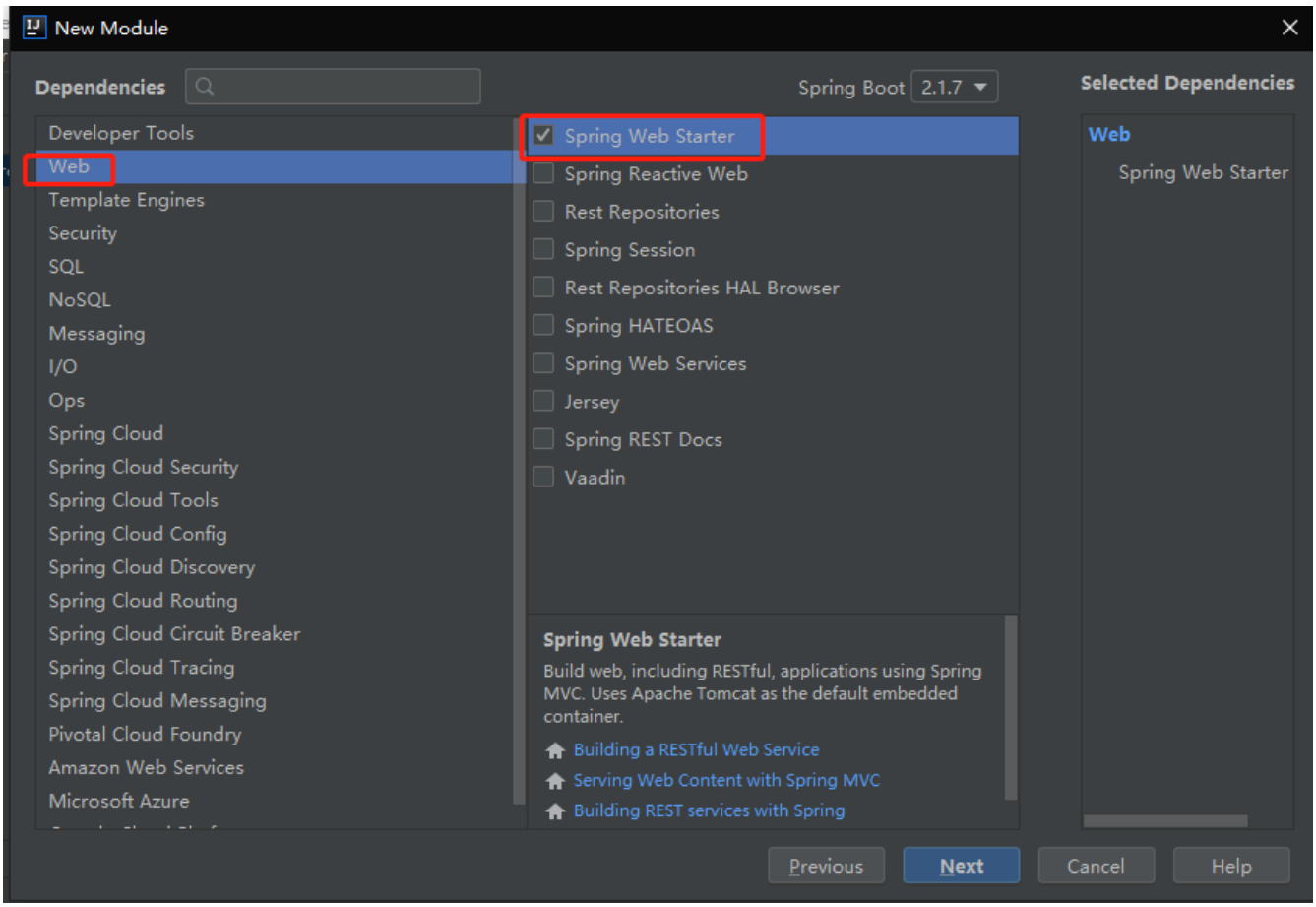
注册中心就搭建成功了。接下来搭建服务的提供者。

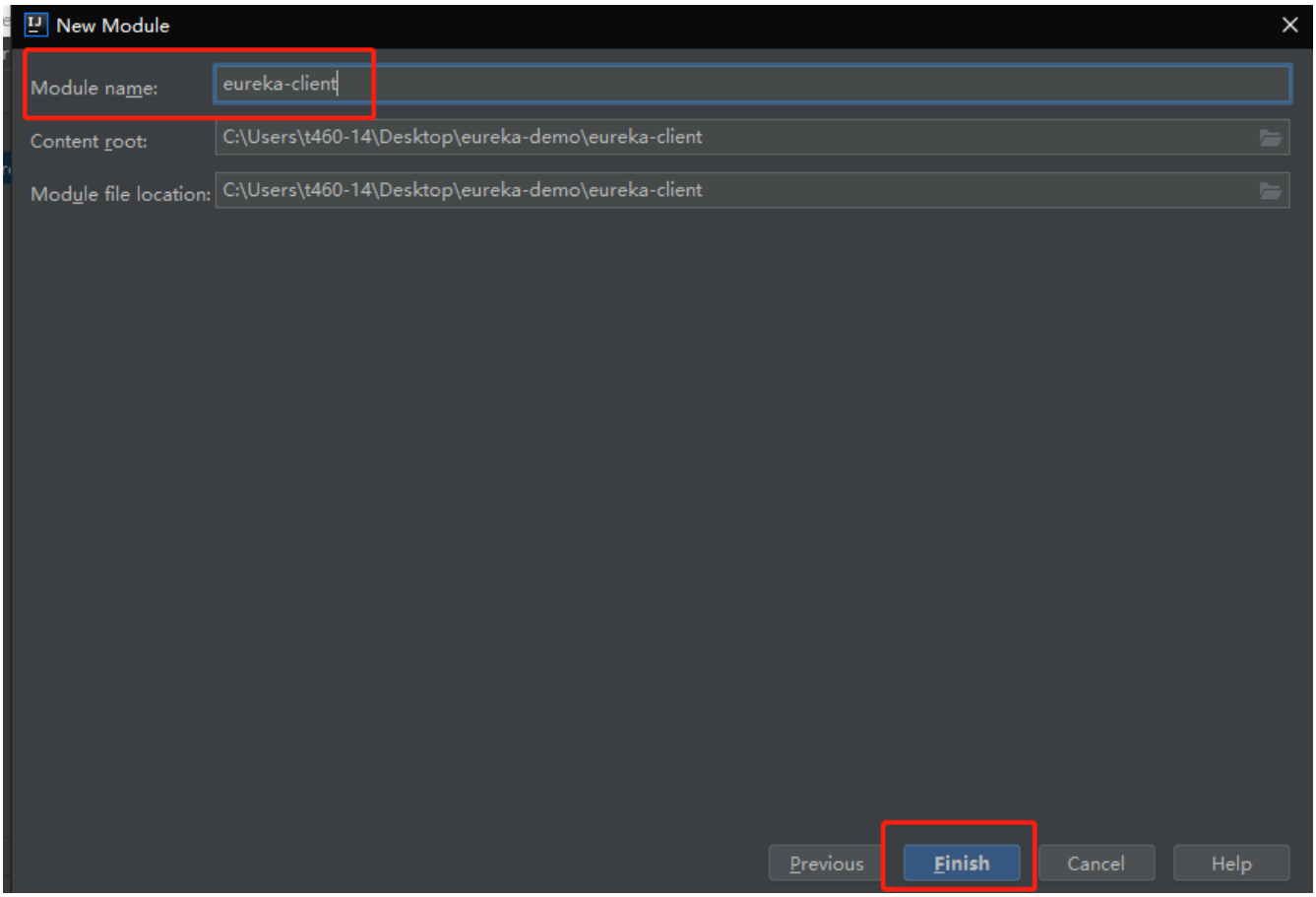
2.3搭建服务提供者

服务的提供者其实就是向eureka注册中心进行注册的服务，可以看作是一个个的app。

建好的maven项目-----》右键----------》new----------》Module-------》Spring Initializr







将pom文件相应的部分改为：

```
<?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>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.7.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.demo01</groupId>
    <artifactId>eureka</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>eureka</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>1.8</java.version>
        <spring-cloud.version>Greenwich.SR2</spring-cloud.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>5.1.9.RELEASE</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>5.1.9.RELEASE</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>5.1.9.RELEASE</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
```

```
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot</artifactId>
            <version>2.1.7.RELEASE</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-autoconfigure</artifactId>
            <version>2.1.7.RELEASE</version>
            <scope>compile</scope>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

把application.properites文件改为application.yaml，并添加下面的内容

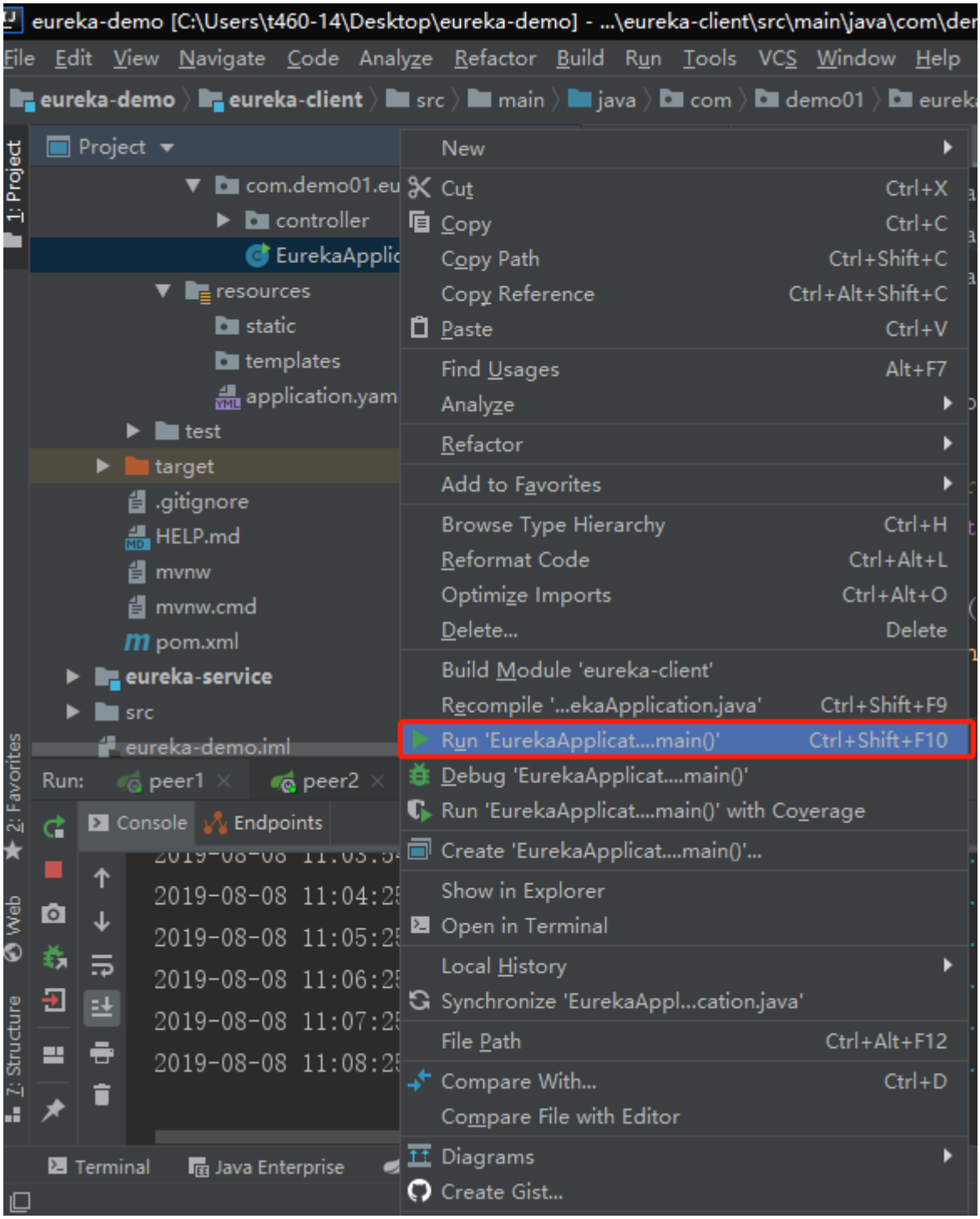
```
server:
  port: 6666
eureka:
  client:
    service-url:
      defaultZone: http://peer1:1111/eureka/
  instance:
    hostname: localhost
spring:
  application:
    name: eureka-peer-client
```

同样的，需要在启动文件中添加注解，参考上面。

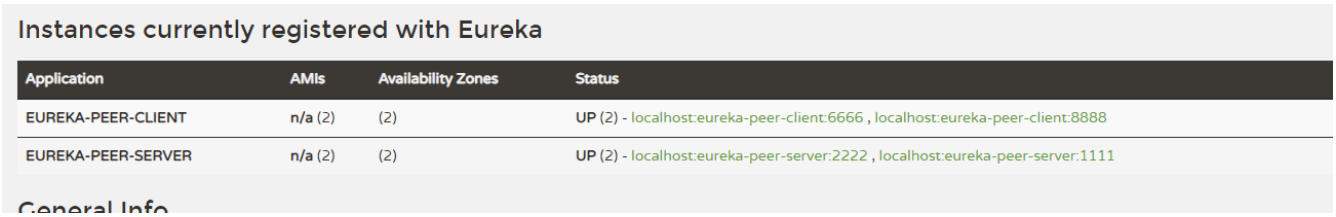
在com.demo01.eureka目录下面建一个package，名为controller，里面建HelloController

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

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

    @RequestMapping(value = "/test", method = RequestMethod.GET)
    public String index(){
        return "Hello World!,端口: "+port;
    }
}
```

之后，在EurekaAapplication右键，点击run



启动之后，可以看到eureka注册界面已经有6666端口的服务注册进来了，修改配置文件中的端口号为8888，再次启动，会发现eureka里面就有两个服务了。



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EUREKA-PEER-CLIENT	n/a (2)	(2)	UP (2) - localhost:eureka-peer-client:6666 , localhost:eureka-peer-client:8888
EUREKA-PEER-SERVER	n/a (2)	(2)	UP (2) - localhost:eureka-peer-server:2222 , localhost:eureka-peer-server:1111

General Info

如果想注册到eureka中的是ip，那么在配置文件中需要：

```
eureka:
  instance:
    prefer-ip-address: true
    instance-id: ${spring.cloud.client.ip-address}:${server.port}
```

还需要添加相关依赖：

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

访问地址可以看到controller里面的内容。

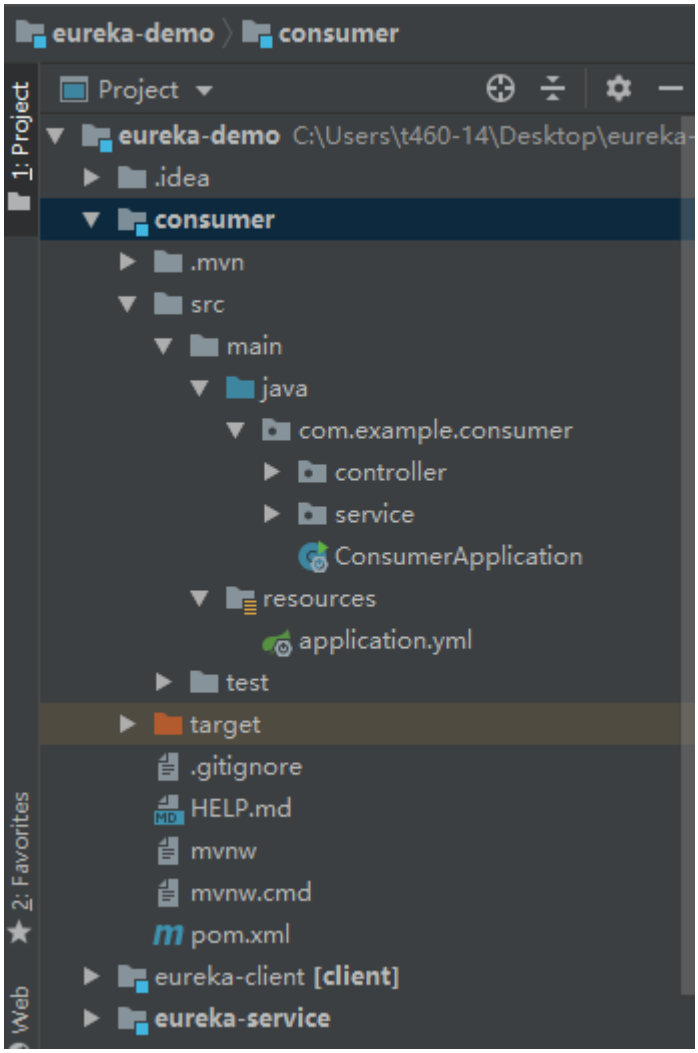
Hello World!,端口：8888

至此，服务的提供者就搭建好了。

2.4搭建服务消费者

服务的消费者有两种实现方式，一种是ribbon，一种是feign。由于feign已经集成了ribbon，所以这里就使用feign的方式来建立消费者。

首先像上面建立服务提供者的方式一样建立，建立完之后，再建立几个package，目录如下：



pom文件中的内容为：

```
<?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>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.6.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>consumer</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>consumer</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>1.8</java.version>
        <spring-cloud.version>Greenwich.SR2</spring-cloud.version>
```

```
    </properties>

    <dependencies>
        <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-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!--open feign-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-openfeign</artifactId>
            <version>2.0.2.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-openfeign-core</artifactId>
            <version>2.0.2.RELEASE</version>
        </dependency>

    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

yaml文件中的内容为：

```
server:
  port: 7777
spring:
  application:
    name: eureka-consumer
eureka:
  client:
    service-url:
      defaultZone: http://peer1:1111/eureka/
    fetch-registry: true
feign:
  hystrix:
    enabled: true
```

启动类中如下：

```
package com.example.consumer;

import ...

@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients(basePackages = "com.example")
@ComponentScan("com.example")
public class ConsumerApplication {

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

}
```

编写consumer的service

```
package com.example.consumer.service;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@FeignClient(name = "eureka-peer-client")
public interface TestFeginServiceClient {

    @RequestMapping(value = "/test", method = RequestMethod.GET)
    public String test();
}
```

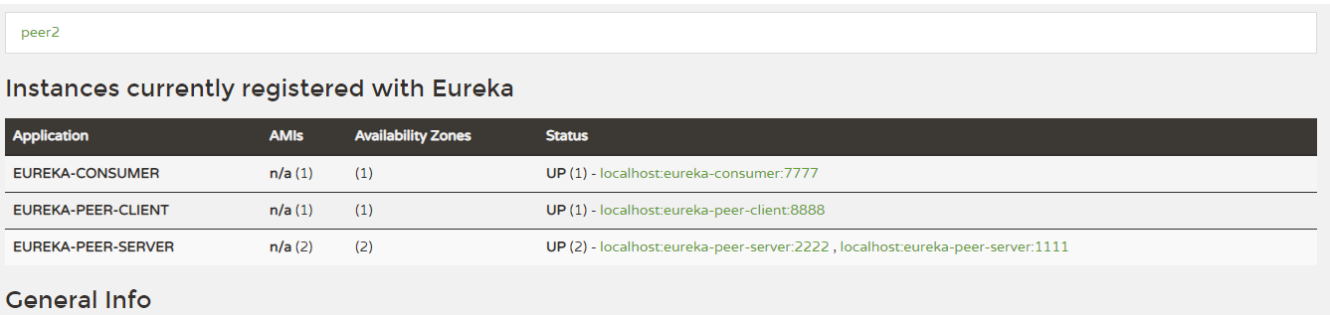
编写consumer的controller

```
package com.example.consumer.controller;
```

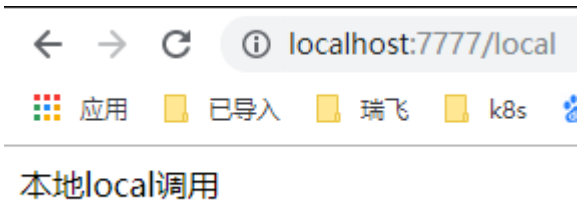
```
import com.example.consumer.service.TestFeginServiceClient;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class TestController {

    @Autowired
    TestFeginServiceClient testService;

    /**
     * 普通Restful
     * @return
     */
    @RequestMapping(value = "/local", method = RequestMethod.GET)
    public String local() {
        return "本地local调用";

    }

    /**
     * 利用Fegin客户端实现RPC调用服务
     * @return
     */
    @RequestMapping(value = "/order/test", method = RequestMethod.GET)
    public String test(){
        return testService.test();
    }

}
```

当访问localhost:7777/local的时候，会调用consumer自身的controller，但是访问localhost:7777/order/test的时候，会调用接口的方法。

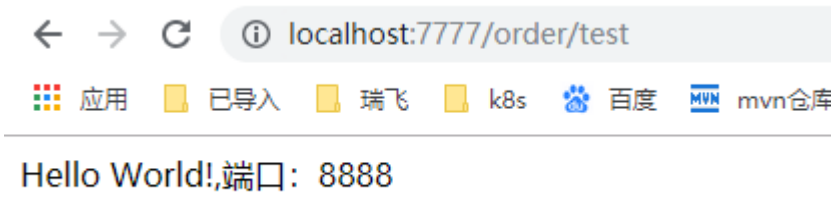
启动consumer会发现，eureka中consumer已经注册了进来。



访问consumer自身的local：



访问生产者的controller：



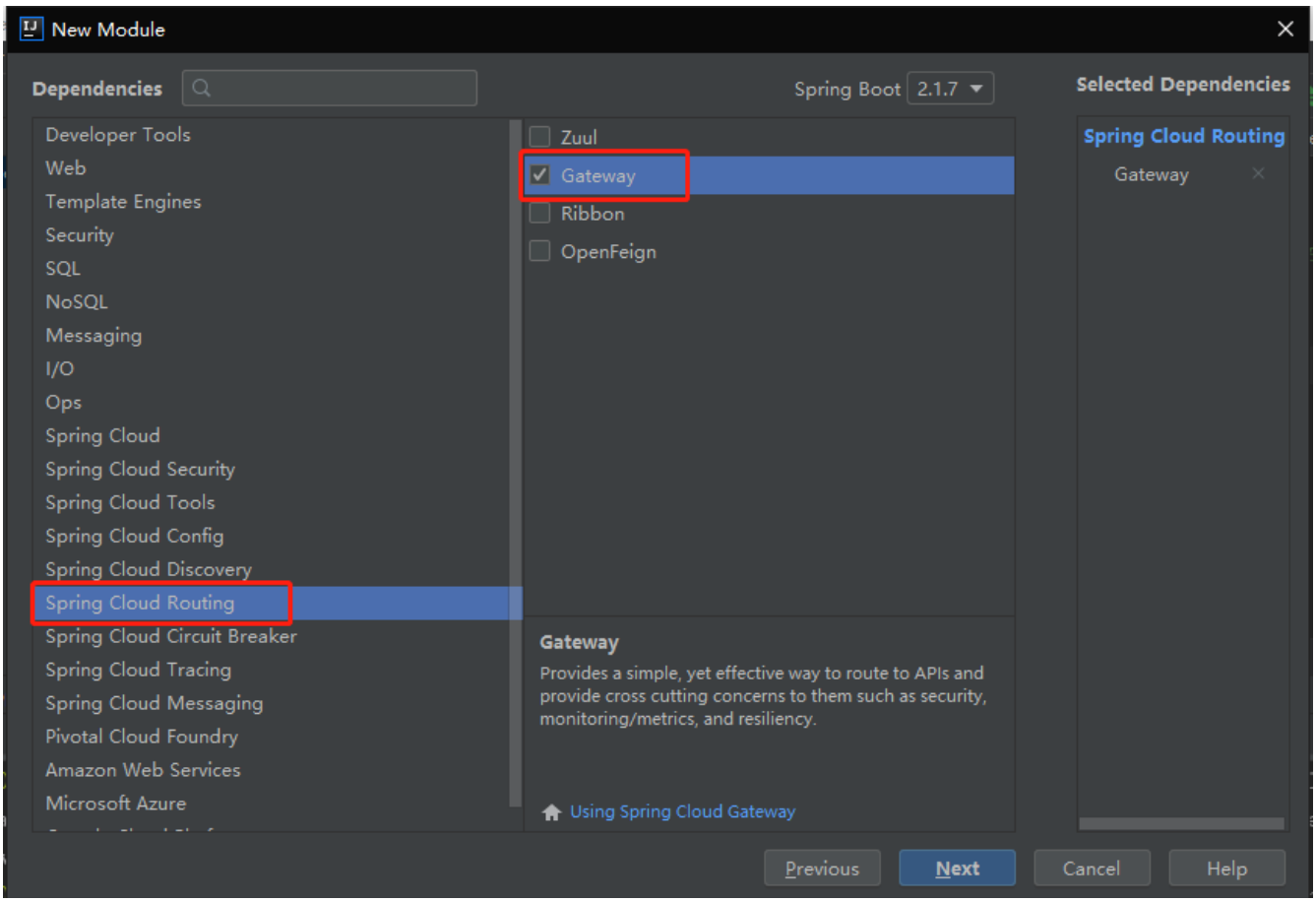
同时，如果多次刷新会发现一个规律，那就是8888和6666交替出现，这是由于feign自带的负载均衡决定的。

3.搭建gateway

gateway的作用是路由转发。外部访问eureka的时候，可以访问consumer，由consumer实现内部的访问，从而达到gateway访问生产者的目的。

这样做的原因是，可以在gateway处对用户的访问进行限制，实现限流的功能，同时，将微服务的各个组件进行解耦，实现低耦合的目的。

重复之前的创建过程，选择Spring Cloud Routing ----》gateway



pom文件:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.6.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>gateway</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>gateway</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>1.8</java.version>
        <spring-cloud.version>Greenwich.SR2</spring-cloud.version>
    </properties>

    <dependencies>

        <!--新增-->
        <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-gateway</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-commons</artifactId>
        </dependency>
    </dependencies>
```

```
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

application.yaml文件内容：

```
server:
  port: 3333
  #端口
spring:
  application:
    name: gateway-service #服务名

  cloud:
    gateway:
      routes:
        - id: customer
          uri: lb://EUREKA-CONSUMER #eureka注册中心存在的服务名称
          predicates:
            - Path=/api/** #路径配置
          filters:
           - StripPrefix=1
eureka:
  client:
    service-url:
      defaultZone: http://localhost:1111/eureka/ #注册中心地址
```

关于routes下的uri的地址，填的是注册中心的服务名称，如果填EUREKA-CONSUMER，那么最后访问的地址是：localhost:8002/api/test/order

但是如果填的是GATEWAY-SERVICE，那么访问的时候，地址为：localhost:8002/api/EUREKA-CONSUMER/test/order

同样，在gateway的启动类中添加@EnableEurekaClient注解。

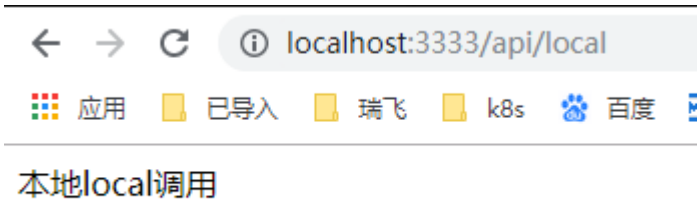
配置完成，启动gateway，可以看到eureka注册中心新增了gateway的 服务。

peer2

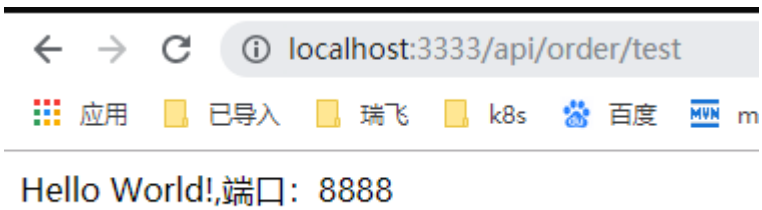
Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EUREKA-CONSUMER	n/a (1)	(1)	UP (1) - localhost:eureka-consumer:7777
EUREKA-PEER-CLIENT	n/a (1)	(1)	UP (1) - localhost:eureka-peer-client:8888
EUREKA-PEER-SERVER	n/a (2)	(2)	UP (2) - localhost:eureka-peer-server:2222 , localhost:eureka-peer-server:1111
GATEWAY-SERVICE	n/a (1)	(1)	UP (1) - 192.168.145.1:3333

同时，访问gateway的地址，访问consumer的local调用：



通过consumer访问service的controller：



并且，不断刷新可以看出，端口在8888和6666之间切换，gateway处也存在负载均衡。

4.实现熔断、限流

熔断的部分有两处，一处是gateway访问consumer的时候，一处是consumer访问service的时候。

4.1 consumer访问service时熔断

用到feign的熔断方法。

在consumer处，当consumer的controller调用service的时候，如果错误就让它返回一个自定义的错误页面。

将service里面的内容改为：

```
package com.example.consumer.service;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

/**
```

```
 * 当调用eureka-peer-client2中的test接口时，如果发生了熔断，会调用FeignServiceFallBack中指定的降级逻辑
 * 将会触发降级
 */
@FeignClient(name = "eureka-peer-client",fallback = FeignServiceFallBack.class)
public interface TestFeginServiceClient {

    @RequestMapping(value = "/test", method = RequestMethod.GET)
    public String test();
}
```

编写fallback的页面：

```
package com.example.consumer.service;
import org.springframework.stereotype.Component;

//定义降级逻辑
@Component
public class FeignServiceFallBack implements TestFeginServiceClient {
    @Override
    public String test() {
        return "调用eureka-peer-client2中的test方法时，发生熔断，熔断位置位于服务的消费者访问服务的提供者处，由此触发了服务降级";
    }
}
```

同时修改yaml配置文件：

```
server:
  port: 7777
spring:
  application:
    name: eureka-consumer
eureka:
  client:
    service-url:
      defaultZone: http://peer1:1111/eureka/
    fetch-registry: true
feign:
  hystrix:
    enabled: true
hystrix:
  command:
    default:
      circuitBreaker:
        requestVolumeThreshold: 3  #5秒之内至少请求3次，熔断器才发生作用。默认20
      execution:
        timeout:
          enabled: true #开启超时
        isolation:
          thread:
```

```
            interruptOnTimeOut: true     #超时中断线程
            timeoutInMilliseconds: 2000 #设定超时时间，默认是1000毫秒
```

现在重启consumer

访问生产者的时候一切正常，但是，如果停掉生产者的服务，页面就会显示：

调用eureka-peer-client2中的test方法时，发生熔断，熔断位置位于服务的消费者访问服务的提供者处，由此触发了服务降级

4.2 gateway访问consumer熔断

gateway到consumer的熔断，需要在gateway的配置文件中加入以下的配置：

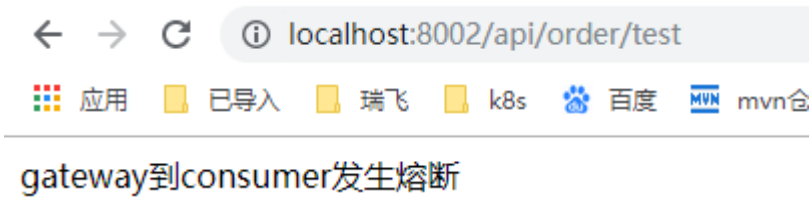
```
server:
  port: 8002
spring:
  application:
    name: gateway-service
  cloud:
    gateway:
      default-filters:
      discovery:
        locator:
          enabled: true
      routes:
        - id: customer_route
          uri: lb://GATEWAY-SERVICE
          predicates:
            - Path= /api/**
          filters:
            - StripPrefix=1
            - name: Hystrix
              args:
                name: fallbackcmd
                fallbackUri: forward:/fallback
eureka:
  instance:
    prefer-ip-address: true
    instance-id: ${spring.cloud.client.ip-address}:${server.port}
  client:
    service-url:
      defaultZone: http://peer1:1111/eureka/

logging:
  level:
    org.springframework.cloud.gateway: trace
    org.springframework.http.server.reactive: debug
    org.springframework.web.reactive: debug
    reactor.ipc.netty: debug
```

```
feign:
  hystrix:
    enabled: true
hystrix:
 command:
    fallbackcmd:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 5000
```

同时指定返回时的页面，fallback

```
package com.example.gateway.controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class FallBackController{

    @RequestMapping("fallback")
    public String fallback(){
        return "gateway到consumer发生熔断";
    }
}
```

当停掉consumer的服务的时，会出现下面的情况：



4.3 限流

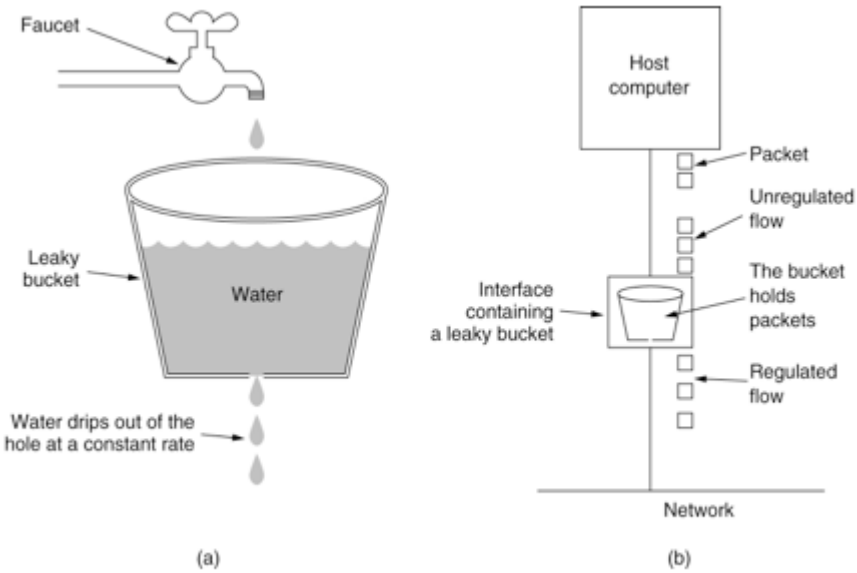
计数器算法

计数器算法采用计数器实现限流有点简单粗暴，一般我们会限制一秒钟的能够通过的请求数，比如限流qps为100，算法的实现思路就是从第一个请求进来开始计时，在接下去的1s内，每来一个请求，就把计数加1，如果累加的数字达到了100，那么后续的请求就会被全部拒绝。等到1s结束后，把计数恢复成0，重新开始计数。具体的实现可以是这样的：对于每次服务调用，可以通过AtomicLong#incrementAndGet()方法来给计数器加1并返回最新值，通过这个最新值和阈值进行比较。这种实现方式，相信大家都知道有一个弊端：如果我在单位时间1s内的前10ms，已经通过了100个请求，那后面的990ms，只能眼巴巴的把请求拒绝，我们把这种现象称为“突刺现象”

漏桶算法

漏桶算法为了消除"突刺现象"，可以采用漏桶算法实现限流，漏桶算法这个名字就很形象，算法内部有一个容器，类似生活用到的漏斗，当请求进来时，相当于水倒入漏斗，然后从下端小口慢慢匀速的流出。不管上面流量多大，下面流出的速度始终保持不变。不管服务调用方多么不稳定，通过漏桶算法进行限流，每10毫秒处理一次请求。因为处理的速度是固定的，请求进来的速度是未知的，可能突然进来很多请求，没来得及处理的请求就先放在桶里，既然是个

桶，肯定是有容量上限，如果桶满了，那么新进来的请求就丢弃。

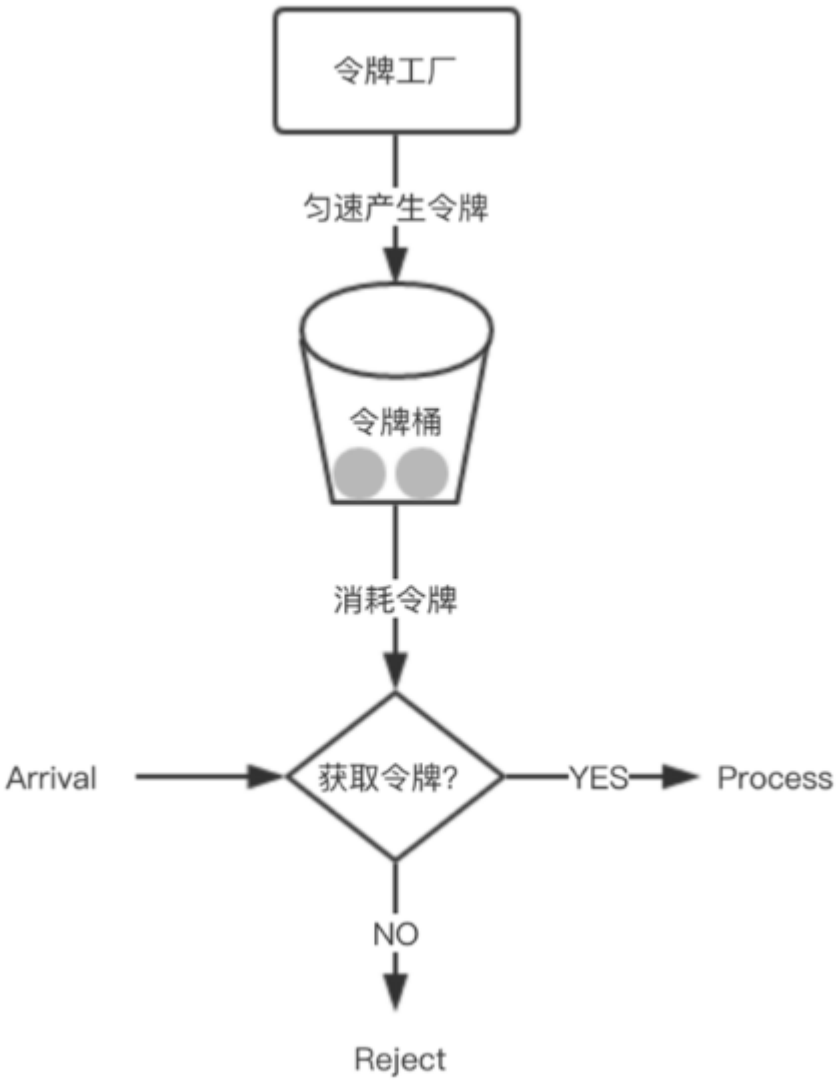


在算法实现方面，可以准备一个队列，用来保存请求，另外通过一个线程池（ScheduledExecutorService）来定期从队列中获取请求并执行，可以一次性获取多个并发执行。

这种算法，在使用过后也存在弊端：无法应对短时间的突发流量。

令牌桶算法

从某种意义上讲，令牌桶算法是对漏桶算法的一种改进，桶算法能够限制请求调用的速率，而令牌桶算法能够在限制调用的平均速率的同时还允许一定程度的突发调用。在令牌桶算法中，存在一个桶，用来存放固定数量的令牌。算法中存在一种机制，以一定的速率往桶中放令牌。每次请求调用需要先获取令牌，只有拿到令牌，才有机会继续执行，否则选择选择等待可用的令牌、或者直接拒绝。放令牌这个动作是持续不断的进行，如果桶中令牌数达到上限，就丢弃令牌，所以就存在这种情况，桶中一直有大量的可用令牌，这时进来的请求就可以直接拿到令牌执行，比如设置qps为100，那么限流器初始化完成一秒后，桶中就已经有100个令牌了，这时服务还没完全启动好，等启动完成对外提供服务时，该限流器可以抵挡瞬时的100个请求。所以，只有桶中没有令牌时，请求才会进行等待，最后相当于以一定的速率执行。



实现思路：可以准备一个队列，用来保存令牌，另外通过一个线程池定期生成令牌放到队列中，每来一个请求，就从队列中获取一个令牌，并继续执行。

采用的是redis限流，所以在本地需要启动redis。

为了让效果更明显，在consumer的controller中设置一个线程：

```
@RequestMapping(value = "/order/test", method = RequestMethod.GET)
   public String test(){
       try {
           Thread.sleep(3000);
       } catch (InterruptedException e) {
           e.printStackTrace();
       }
       return testService.test();
   }
```

限流是在gateway处进行设置，限流的结果是等待限流结束，但是等待的过程中会出现超时的现象，超时会导致gateway访问consumer出现熔断，所以进行限流的测试时，把gateway超时时间调的大一点，防止出现熔断。

```
hystrix:
 command:
    fallbackcmd:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 5000
```

限流有三种实现的方式，通过ip限流，通过user限流，通过uri限流。

在gateway的java目录下建立resolver包，包里面放的就是限流的方式：

```
package com.example.gateway.resolver;


import org.springframework.cloud.gateway.filter.ratelimit.KeyResolver;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

public class HostAddKeyResolver implements KeyResolver {
    @Override
    public Mono<String> resolve(ServerWebExchange exchange) {
        return Mono.just(exchange.getRequest().getQueryParams().getFirst("user"));
    }

}
```

```
package com.example.gateway.resolver;

import org.springframework.cloud.gateway.filter.ratelimit.KeyResolver;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

public class IpKeyResolver implements KeyResolver {
    @Override
    public Mono<String> resolve(ServerWebExchange exchange) {
        return
Mono.just(exchange.getRequest().getRemoteAddress().getAddress().getHostAddress());
    }
}
```

```
package com.example.gateway.resolver;

import org.springframework.cloud.gateway.filter.ratelimit.KeyResolver;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

/**
```

```
 * 做测试用的uri，所以将user写死
 */
public class UriKeyResolver implements KeyResolver {

    @Override
    public Mono<String> resolve(ServerWebExchange exchange) {
//        return Mono.just(exchange.getRequest().getURI().getPath());
        String user = "1";
        return Mono.just(user);
    }

}
```

限流同时需要在gateway的启动类中添加bean标签，

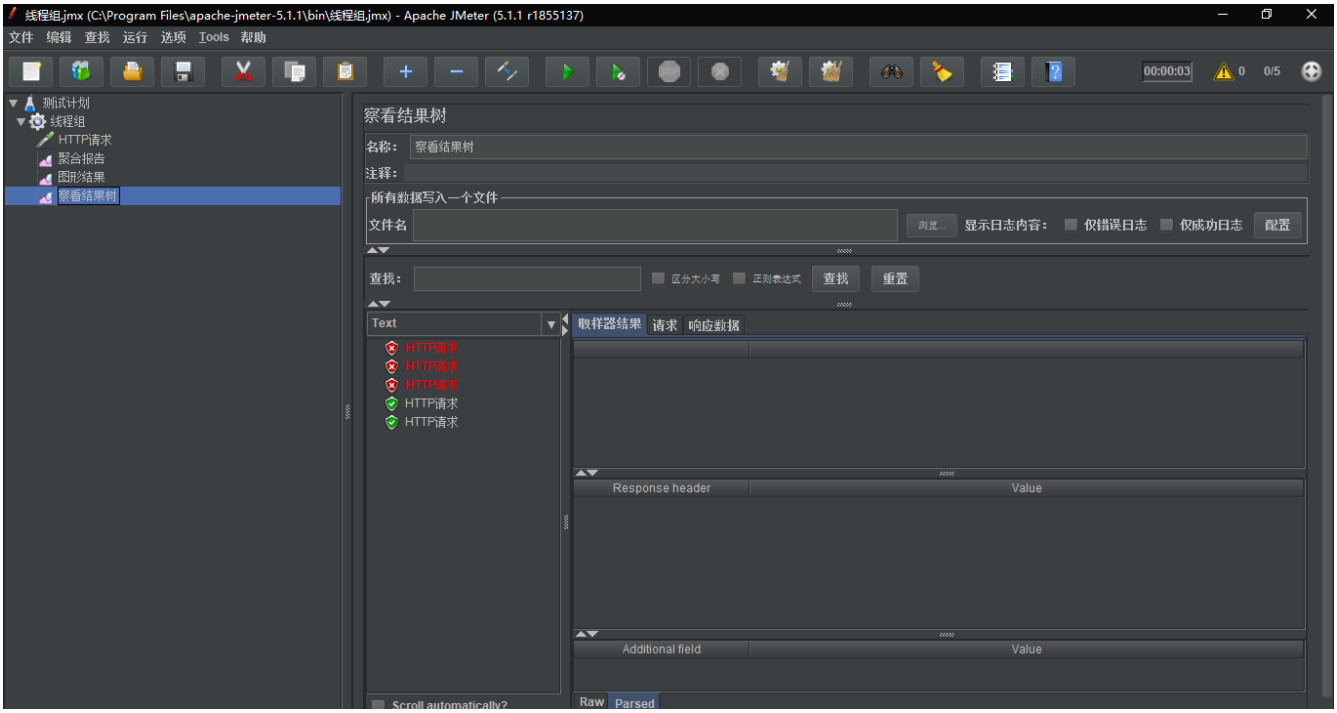
```
package com.example.gateway;

import com.example.gateway.resolver.UriKeyResolver;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.context.annotation.Bean;

@EnableEurekaClient
@SpringBootApplication
public class GatewayApplication {

    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }
    @Bean
/*    public HostAddKeyResolver hostAddKeyResolver(){
        return new HostAddKeyResolver();
    }*/
/*    public IpKeyResolver ipKeyResolver(){
        return new IpKeyResolver();
    }*/
    public UriKeyResolver uriKeyResolver() {
        return new UriKeyResolver();
    }
}
```

修改gateway的配置文件为：

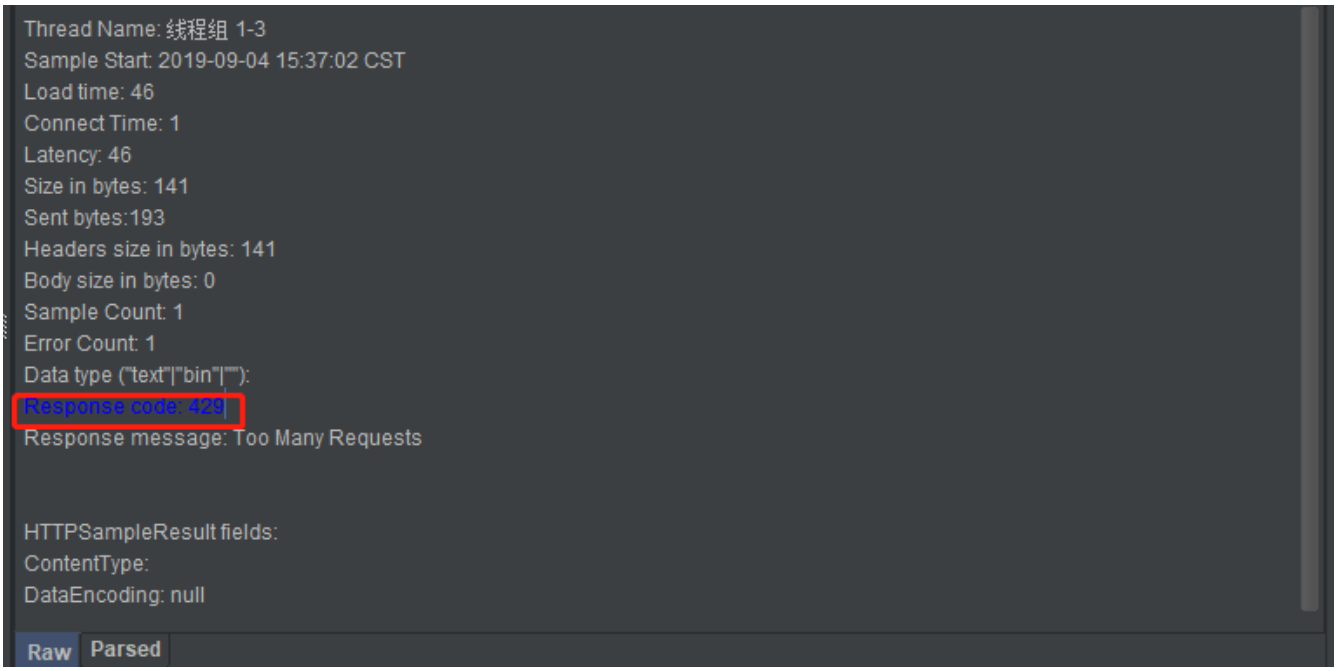
```
server:
  port: 8002
spring:
  application:
    name: gateway-service
  cloud:
    gateway:
```

```
      default-filters:
      discovery:
        locator:
          enabled: true
      routes:
        - id: customer_route
          uri: lb://GATEWAY-SERVICE
          predicates:
            - Path= /api/**
            - Method=GET
          filters:
            - StripPrefix=1
            - name: Hystrix
              args:
                name: fallbackcmd
                fallbackUri: forward:/fallback
            - name: RequestRateLimiter
              args:
                redis-rate-limiter.replenishRate: 1 #令牌桶每秒填充速率
                redis-rate-limiter.burstCapacity: 2 #令牌桶总容量
                key-resolver: "#{@remoteAddrKeyResolver}" #bean对象的名字，从spring容器中获取
 redis:
    host: 127.0.0.1
    port: 6379
eureka:
  instance:
    prefer-ip-address: true
    instance-id: ${spring.cloud.client.ip-address}:${server.port}
  client:
    service-url:
      defaultZone: http://peer1:1111/eureka/

logging:
  level:
    org.springframework.cloud.gateway: trace
    org.springframework.http.server.reactive: debug
    org.springframework.web.reactive: debug
    reactor.ipc.netty: debug
feign:
  hystrix:
    enabled: true
hystrix:
 command:
    fallbackcmd:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 5000
```

通过JMeter的压力测试可以清楚的看到

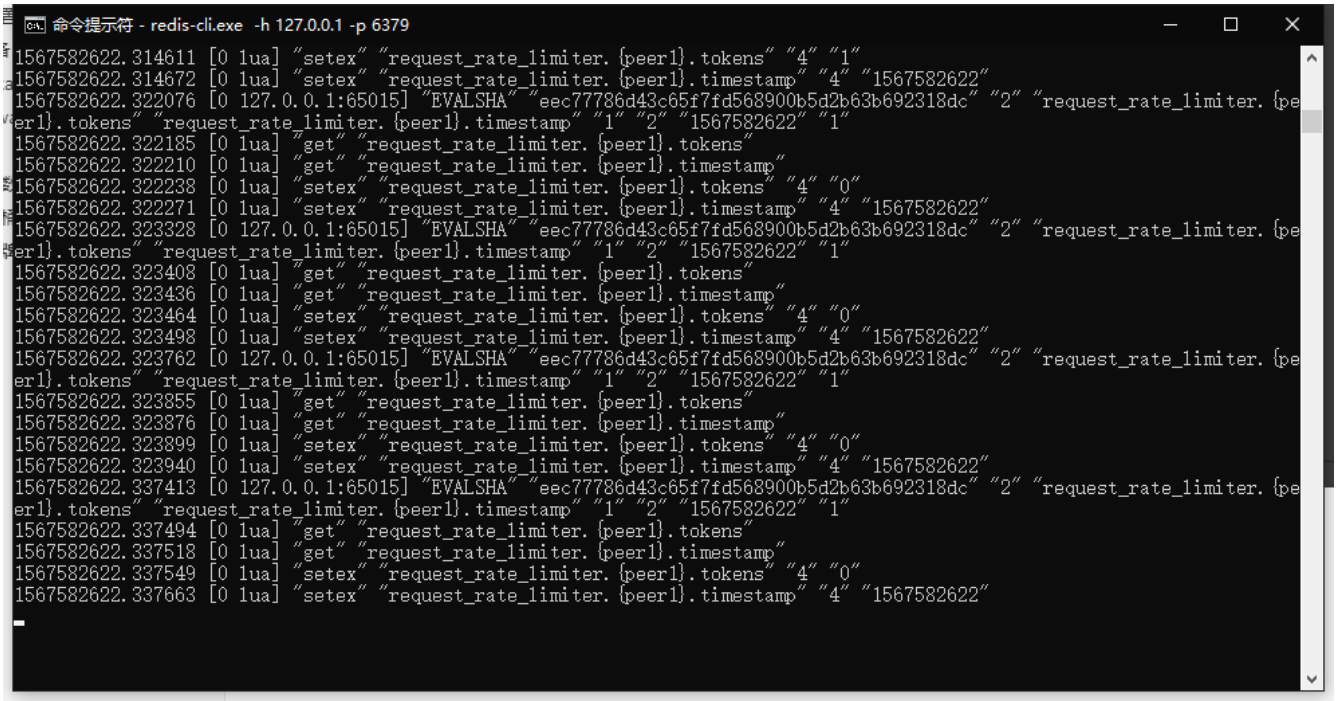


五个线程同时进行访问，会有3个请求被拒绝。



显示429，说明限流成功。

在redis的界面上面，可以看到：



说明，gateway的限流是基于Redis的限流。

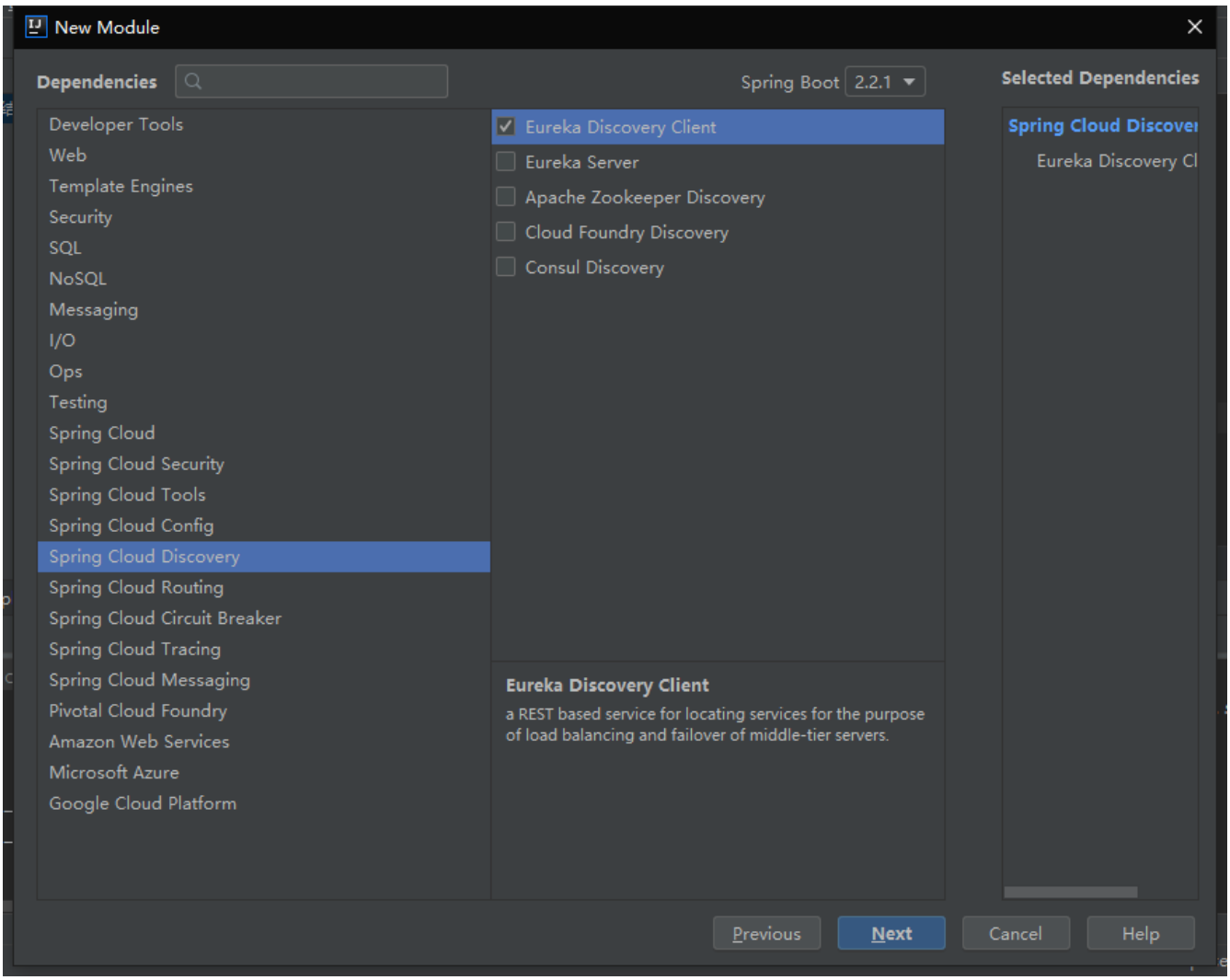
5、部署spring cloud config

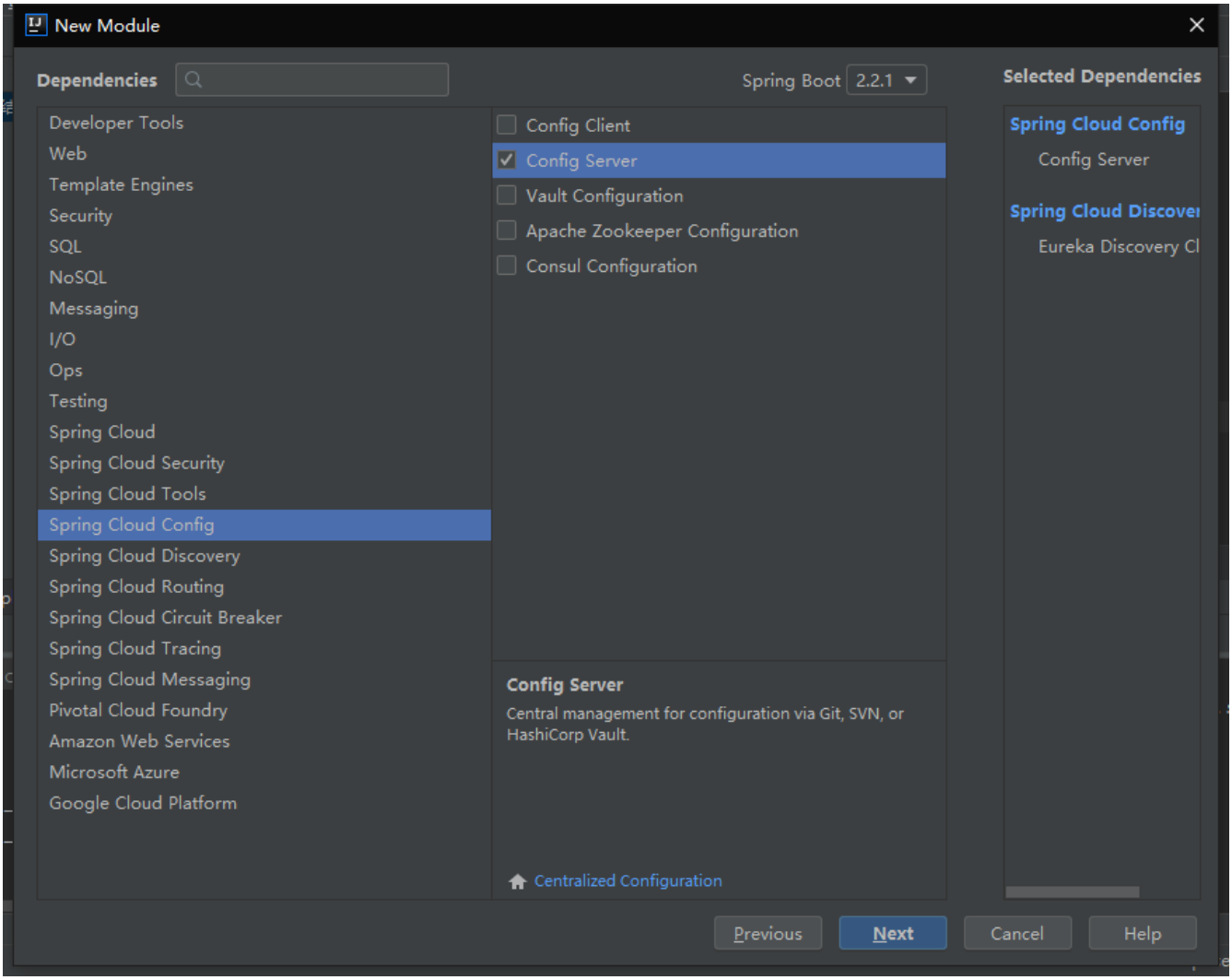
config的类型有：spring cloud config 、apollo、alibaba nacos

5.1部署及说明

5.1.1spring cloud config部署及说明

新建一个module


New Module
Dependencies
Spring Boot 2.2.1
Selected Dependencies
Developer Tools
Web
Template Engines
Security
SQL
NoSQL
Messaging
I/O
Ops
Testing
Spring Cloud
Spring Cloud Security
Spring Cloud Tools
Spring Cloud Config
Spring Cloud Discovery
Spring Cloud Routing
Spring Cloud Circuit Breaker
Spring Cloud Tracing
Spring Cloud Messaging
Pivotal Cloud Foundry
Amazon Web Services
Microsoft Azure
Google Cloud Platform
Eureka Discovery Client
Eureka Server
Apache Zookeeper Discovery
Cloud Foundry Discovery
Consul Discovery
Spring Cloud Discover
Eureka Discovery Cl
Eureka Discovery Client
a REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.
Previous
Next
Cancel
Help



然后开始编辑config server的配置文件。

这就提到了了两种配置文件，一个是bootstrap.yml，一个是application.yml

当使用spring cloud config server的时候，可以在bootstrap.yml中指定spring.application.name 和 spring.cloud.config.server.git.uri以及一些加密的信息

bootstrap.yml比application.yml更早加载，配置的更多的是不变的或者是不经常变化的属性，application属性文件可以配置更灵活的属性

在普通服务中从配置中心服务加载配置文件，在bootstrap.yml配置如下：

```
spring:
  application:
    name: demo    #当前服务名称(这个可以放在application.yml里配置)
  cloud:
    config: #配置文件获取
      uri: http://localhost:8040    #不使用服务发现（eureka等），则直接通过uri指定配置中心的地址
      label: master          #github仓库的分支名(默认应该就是master，其他分支就在这指定)
      name: config-file-name   #name指定想要从从配置中心加载的配置文件名，不用加后缀，获取多个则以逗号
隔开
```

5.1.2apollo部署及说明

5.2测试