

1. 课程介绍
  - 1.1 课程内容
  - 1.2 课程收获
2. 微服务介绍
  - 2.1 什么是微服务
  - 2.2 微服务优势
  - 2.3 使用 Spring Cloud 的优势
3. Spring Cloud 介绍
  - 3.1 什么是 Spring Cloud
  - 3.2 核心特性
  - 3.3 版本名称
4. Spring Cloud 体系
  - 4.1 Spring Cloud 包含的组件
  - 4.2 Spring Cloud 和 Spring Boot 版本关系
5. Eureka
  - 注册中心
  - Eureka
  - 5.1 Eureka 搭建
  - 5.2 Eureka 集群
  - 5.3 Eureka 工作细节
    - 5.3.1 Eureka Server
    - 5.3.2 Eureka Client
      - 5.3.2.1 服务注册
      - 5.3.2.2 服务续约
      - 5.3.2.3 服务下线
      - 5.3.2.4 获取注册表信息
  - 5.4 Eureka 集群原理
6. 服务注册与消费
  - 6.1 服务注册
  - 6.2 服务消费
    - 6.2.1 基本用法
    - 6.2.2 升级改造
  - 6.3 RestTemplate
    - 6.3.1 GET
    - 6.3.2 POST
    - 6.3.3 PUT
    - 6.3.4 DELETE
  - 6.4 客户端负载均衡
  - 6.5 负载均衡原理
7. Consul
  - 7.1 安装
  - 7.2 Consul 使用
  - 7.3 Consul 集群注册
  - 7.4 消费
8. Hystrix
  - 8.1 基本介绍
  - 8.2 基本用法
  - 8.3 请求命令
  - 8.4 异常处理
  - 8.5 请求缓存
  - 8.6 请求合并
9. OpenFeign
  - 9.1 OpenFeign
    - 9.1.1 HelloWorld
  - 9.2 参数传递

- 9.3 继承特性
- 9.4 日志
- 9.5 数据压缩
- 9.6 +Hystrix
- 10. Resilience4j
  - 10.1 Resilience4j 简介
  - 10.2 基本用法
    - 10.2.1 断路器
    - 10.2.2 限流
    - 10.2.3 请求重试
  - 10.3 结合微服务
    - 10.3.1 Retry
    - 10.3.2 CircuitBreaker
    - 10.3.3 RateLimiter
  - 10.4 服务监控
    - 10.4.1 Prometheus
- 11. Zuul
  - 11.1 服务网关
  - 11.2 Zuul
    - 11.2.1 HelloWorld
    - 11.2.2 请求过滤
    - 11.2.3 Zuul 中的其他配置
- 12. Gateway
  - 12.1 简介
  - 12.2 基本用法
    - 12.2.1 服务化
  - 12.3 Predicate
  - 12.4 Filter
- 13. 分布式配置中心
  - 13.1 基本用法
    - 13.1.1 简介
    - 13.1.2 准备工作
    - 13.1.2 ConfigServer
    - 13.1.3 ConfigClient
    - 13.1.4 配置
  - 13.2 配置文件加解密
    - 13.2.1 常见加密方案
    - 13.2.2 对称加密
    - 13.2.3 非对称加密
  - 13.3 安全管理
  - 13.4 服务化
  - 13.5 动态刷新
  - 13.6 请求失败重试
- 14.Spring Cloud Bus
  - 逐个刷新

## 1. 课程介绍

---

### 1.1 课程内容

1. 介绍微服务的由来，以及微服务和 Spring Cloud 之间的关系
2. 介绍 Spring Cloud 核心组件的使用，使小伙伴们通过核心组件可以快速搭建一个微服务架构
3. 介绍 Spring Cloud 中的辅助类组件，例如微服务监控、链路追踪等等
4. 介绍 Spring Cloud Alibaba，以及相关核心组件的具体用法

## 1.2 课程收获

1. 了解微服务的由来以及基本原理
2. 学会 Spring Cloud 中各个组件的使用
3. 了解 Spring Cloud 中核心组件的运行原理
4. 掌握通过 Spring Cloud 搭建微服务架构
5. 掌握辅助组件的用法

## 2. 微服务介绍

微服务架构越来越流行，这个没有异议。

2009 年，Netflix 重新定义了它的应用程序员的开发模型，这个算是微服务的首次探索。

20014 年，[《Microservices》](#)，这篇文章以一个更加通俗易懂的方式，为大家定义了微服务。

为什么要用微服务？

互联网应用产品的两大特点：

1. 需求变化快
2. 用户群体庞大

在这样的情况下，我们需要构建一个能够灵活扩展，同时能够快速应对外部环境变化的一个应用，使用传统的开发方式，显然无法满足需求。这个时候，微服务就登场了。

### 2.1 什么是微服务

简单来说，微服务就是一种将一个单一应用程序拆分为一组小型服务的方法，拆分完成后，每一个服务都运行在独立的进程中，服务于服务之间采用轻量级的通信机制来进行沟通（Spring Cloud 中采用基于 HTTP 的 RESTful API）。

每一个服务，都是围绕具体的业务进行构建，例如一个电商系统，订单服务、支付服务、物流服务、会员服务等等，这些拆分后的应用都是独立的应用，都可以独立的部署到生产环境中。就是在采用微服务之后，我们的项目不再拘泥于一种语言，可以 Java、Go、Python、PHP 等等，混合使用，这在传统的应用开发中，是无法想象的。而使用了微服务之后，我们可以根据业务上下文来选择合适的语言和构建工具进行构建。

微服务可以理解为是 SOA 的一个传承，一个本质的区别是微服务是一个真正分布式、去中心化的，微服务的拆分比 SOA 更加彻底。

### 2.2 微服务优势

1. 复杂度可控
2. 独立部署
3. 技术选型灵活
4. 较好的容错性
5. 较强的可扩展性

### 2.3 使用 Spring Cloud 的优势

Spring Cloud 可以理解为微服务这种思想在 Java 领域的一个具体落地。Spring Cloud 在发展之初，就借鉴了微服务的思想，同时结合 Spring Boot，Spring Cloud 提供了组件的一键式启动和部署的能力，极大的简化了微服务架构的落地。

Spring Cloud 这种框架，从设计之初，就充分考虑了分布式架构演化所需要的功能，例如服务注册、配置中心、消息总线以及负载均衡等。这些功能都是以可插拔的形式提供出来的，这样，在分布式系统不断演化的过程中，我们的 Spring Cloud 也可以非常方便的进化。

## 3. Spring Cloud 介绍

### 3.1 什么是 Spring Cloud

Spring Cloud 是一系列框架的集合，Spring Cloud 内部包含了许多框架，这些框架互相协作，共同来构建分布式系统。利用这些组件，可以非常方便的构建一个分布式系统。

### 3.2 核心特性

1. 服务注册与发现
2. 负载均衡
3. 服务之间调用
4. 容错、服务降级、断路器
5. 消息总线
6. 分布式配置中心
7. 链路器

### 3.3 版本名称

不同于其他的框架，Spring Cloud 版本名称是通过 A (Angel)、B (Brixton)、C (Camden)、D (Dalston)、E (Edgware)、F (Finchley)。。。这样来命名的，这些名字使用了伦敦地铁站的名字，目前最新版是 H (Hoxton) 版。

Spring Cloud 中，除了大的版本之外，还有一些小版本，小版本命名方式如下：

- M，M 版是 milestone 的缩写，所以我们会看到一些版本叫 M1、M2
- RC，RC 是 Release Candidate，表示该项目处于候选状态，这是正式发版之前的一个状态，所以我们会看到 RC1、RC2
- SR，SR 是 Service Release，表示项目正式发布的稳定版，其实相当于 GA (Generally Available) 版。所以，我们会看到 SR1、SR2
- SNAPSHOT，这个表示快照版

## 4. Spring Cloud 体系

### 4.1 Spring Cloud 包含的组件

- Spring Cloud Netflix，这个组件，在 Spring Cloud 成立之初，立下了汗马功劳。但是，2018 年的断更，也是 Netflix 掉链子了。
- Spring Cloud Config，分布式配置中心，利用 **Git/Svn** 来集中管理项目的配置文件
- Spring Cloud Bus，消息总线，可以构建消息驱动的微服务，也可以用来做一些状态管理等
- Spring Cloud Consul，服务注册发现
- Spring Cloud Stream，基于 Redis、RabbitMQ、Kafka 实现的消息微服务
- Spring Cloud OpenFeign，提供 OpenFeign 集成到 Spring Boot 应用中的方式，主要解决微服务之间的调用问题
- Spring Cloud Gateway，Spring Cloud 官方推出的网关服务
- Spring Cloud Cloudfoundry，利用 Cloudfoundry 集成我们的应用程序
- Spring Cloud Security，在 Zuul 代理中，为 OAuth2 客户端认证提供支持
- Spring Cloud AWS，快速集成亚马逊云服务
- Spring Cloud Contract，一个消费者驱动的、面向 Java 的契约框架

- Spring Cloud Zookeeper, 基于 Apache Zookeeper 的服务注册和发现
- Spring Cloud Data Flow, 在一个结构化的平台上, 组成数据微服务
- Spring Cloud Kubernetes, Spring Cloud 提供的针对 Kubernetes 的支持
- Spring Cloud Function
- Spring Cloud Task, 短生命周期的微服务

## 4.2 Spring Cloud 和 Spring Boot 版本关系

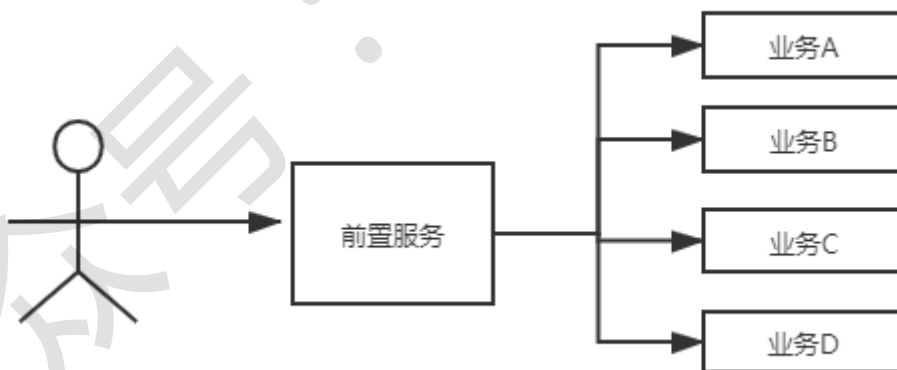
Spring Cloud	Spring Boot
Hoxton	2.2.x
Greenwich	2.1.x
Finchley	2.0.x
Edgware	1.5.x
Dalston	1.5.x

## 5. Eureka

### 注册中心

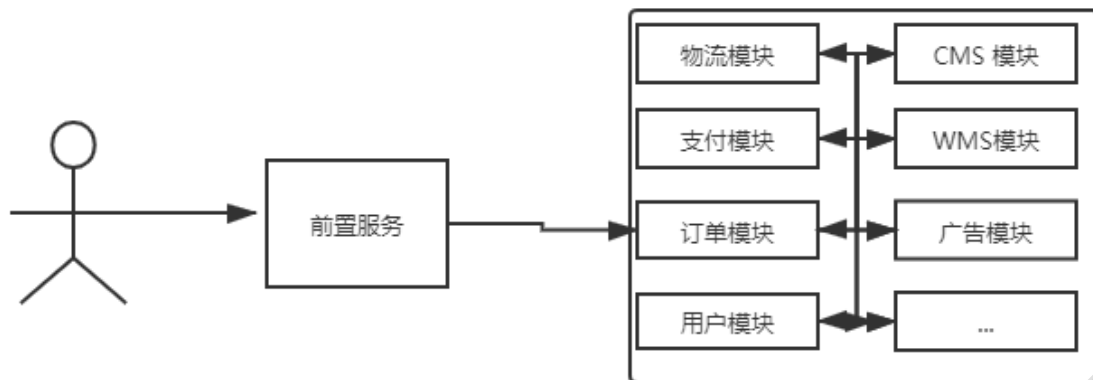
Eureka 是 Spring Cloud 中的注册中心, 类似于 Dubbo 中的 Zookeeper。那么到底什么是注册中心, 我们为什么需要注册中心?

我们首先来看一个传统的单体应用:



在单体应用中, 所有的业务都集中在一个项目中, 当用户从浏览器发起请求时, 直接由前端发起请求给后端, 后端调用业务逻辑, 给前端请求做出响应, 完成一次调用。整个调用过程是一条直线, 不需要服务之间的中转, 所以没有必要引入注册中心。

随着公司项目越来越大, 我们会将系统进行拆分, 例如一个电商项目, 可以拆分为订单模块、物流模块、支付模块、CMS 模块等等。这样, 当用户发起请求时, 就需要各个模块之间进行协作, 这样不可避免的要进行模块之间的调用。此时, 我们的系统架构就会发生变化:



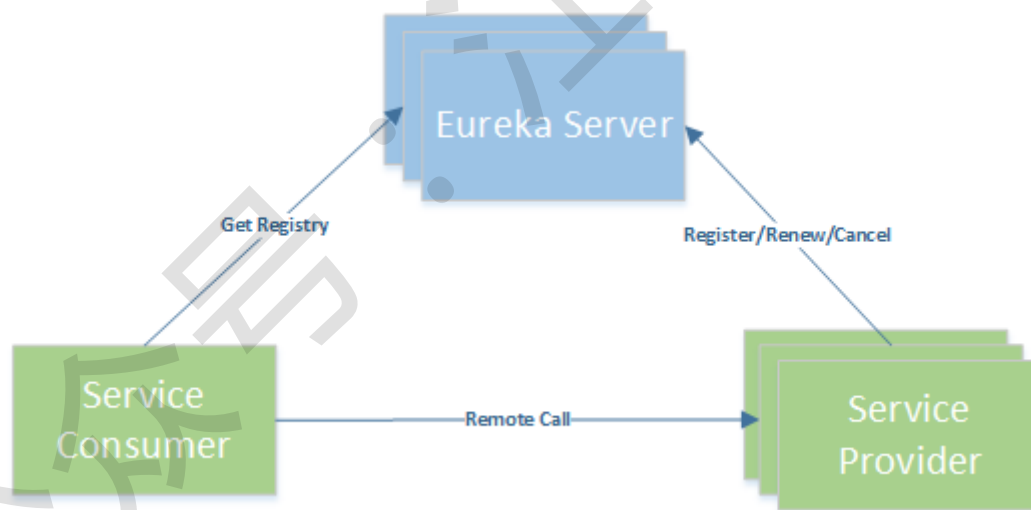
在这里，大家可以看到，模块之间的调用，变得越来越复杂，而且模块之间还存在强耦合。例如 A 调用 B，那么就要在 A 中写上 B 的地址，也意味着 B 的部署位置要固定，同时，如果以后 B 要进行集群化部署，A 也需要修改。

为了解决服务之间的耦合，注册中心闪亮登场。

## Eureka

Eureka 是 Netflix 公司提供的一款服务注册中心，Eureka 基于 REST 来实现服务的注册与发现，曾经，Eureka 是 Spring Cloud 中最重要的核心组件之一。Spring Cloud 中封装了 Eureka，在 Eureka 的基础上，优化了一些配置，然后提供了可视化的页面，可以方便的查看服务的注册情况以及服务注册中心集群的运行情况。

Eureka 由两部分：服务端和客户端，服务端就是注册中心，用来接收其他服务的注册，客户端则是一个 Java 客户端，用来注册，并可以实现负载均衡等功能。



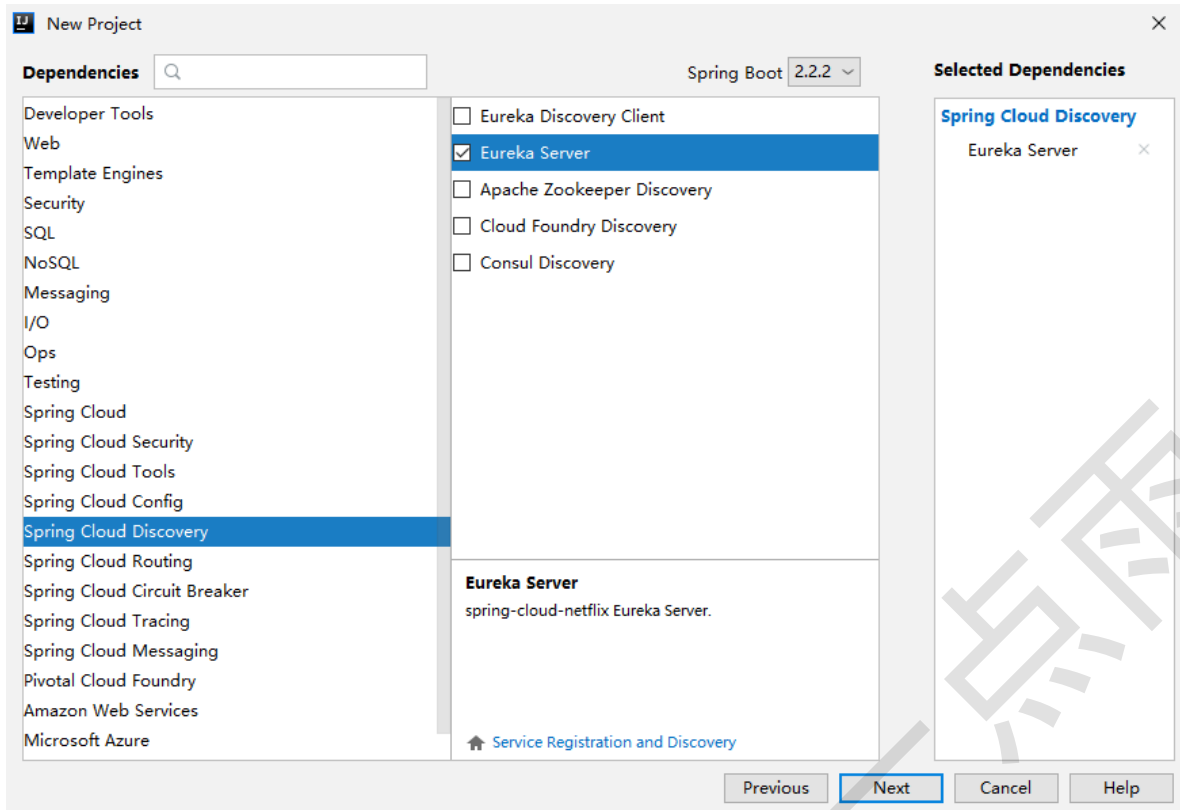
从图中，我们可以看出，Eureka 中，有三个角色：

- Eureka Server：注册中心
- Eureka Provider：服务提供者
- Eureka Consumer：服务消费者

## 5.1 Eureka 搭建

Eureka 本身是使用 Java 来开发的，Spring Cloud 使用 Spring Boot 技术对 Eureka 进行了封装，所以，在 Spring Cloud 中使用 Eureka 非常方便，只需要引入 `spring-cloud-starter-netflix-eureka-server` 这个依赖即可，然后就像启动一个普通的 Spring Boot 项目一样启动 Eureka 即可。

创建一个普通的 Spring Boot 项目，创建时，添加 Eureka 依赖：



项目创建成功后，在项目启动类上添加注解，标记该项目是一个 Eureka Server：

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaApplication {

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

}
```

`@EnableEurekaServer` 注解表示开启 Eureka 的功能。

接下来，在 `application.properties` 中添加基本配置信息：

```
# 给当前服务取一个名字
spring.application.name=eureka
# 设置端口号
server.port=1111
# 默认情况下，Eureka Server 也是一个普通的微服务，所以当它还是一个注册中心的时候，他会有两层
身份：1.注册中心；2.普通服务，即当前服务会自己把自己注册到自己上面来
# register-with-eureka 设置为 false，表示当前项目不要注册到注册中心上
eureka.client.register-with-eureka=false
# 表示是否从 Eureka Server 上获取注册信息
eureka.client.fetch-registry=false
```

配置完成后，就可以启动项目了。

如果在项目启动时，遇到 `java.lang.TypeNotPresentException: Type javax.xml.bind.JAXBContext not present` 异常，这是因为 JDK9 以上，移除了 JAXB，这个时候，只需要我们手动引入 JAXB 即可。

```

<dependency>
  <groupId>javax.xml.bind</groupId>
  <artifactId>jaxb-api</artifactId>
  <version>2.3.0</version>
</dependency>
<dependency>
  <groupId>com.sun.xml.bind</groupId>
  <artifactId>jaxb-impl</artifactId>
  <version>2.3.0</version>
</dependency>
<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-runtime</artifactId>
  <version>2.3.0</version>
</dependency>
<dependency>
  <groupId>javax.activation</groupId>
  <artifactId>activation</artifactId>
  <version>1.1.1</version>
</dependency>

```

项目启动成功后，浏览器输入 <http://localhost:1111> 就可以查看 Eureka 后台管理页面了：

The screenshot shows the Spring Eureka Admin interface. At the top, there's a navigation bar with 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content area is divided into several sections:

- System Status** (系统状态，例如系统启动时间等): A table showing environment (test), data center (default), current time (2020-01-08T18:09:25 +0800), uptime (00:06), lease expiration enabled (false), renew threshold (1), and renew last min (0).
- DS Replicas** (集群环境下的副本，也就是当前服务从哪里同步数据): A text input field containing 'localhost'.
- Instances currently registered with Eureka** (当前注册上来的服务): A table with columns Application, AMIs, Availability Zones, and Status. It shows 'No instances available'.
- General Info** (系统运行环境，如内存、CPU): A table with columns Name and Value. It shows total-avail-memory (72mb), environment (test), and num-of-cpus (4).
- Instance Info** (当前服务的基本信息，例如 ip 地址，状态等等): A table with columns Name and Value. It shows 'im&ndr' and '192.168.0.1 1'.

## 5.2 Eureka 集群

使用了注册中心之后，所有的服务都要通过服务注册中心来进行信息交换。服务注册中心的稳定性就非常重要了，一旦服务注册中心掉线，会影响到整个系统的稳定性。所以，在实际开发中，Eureka 一般都是以集群的形式出现的。

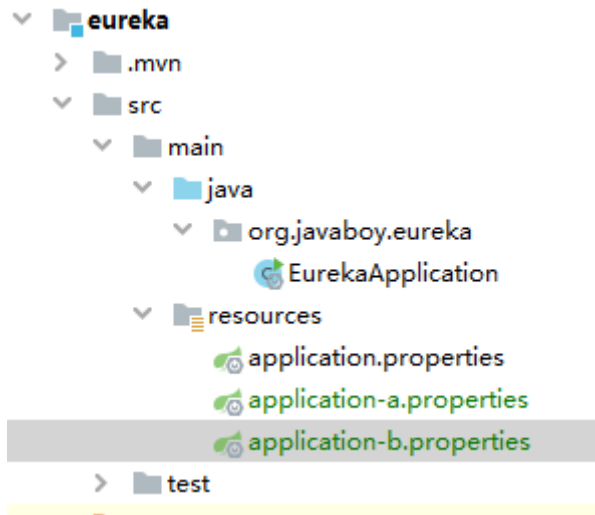
Eureka 集群，实际上就是启动多个 Eureka 实例，多个 Eureka 实例之间，互相注册，互相同步数据，共同组成一个 Eureka 集群。

搭建 Eureka 集群，首先我们需要一点准备工作，修改电脑的 hosts 文件  
( C:\Windows\System32\drivers\etc\hosts )：

```
127.0.0.1 eureka-a eureka-b
```

在 5.1 小节 demo 的基础上，我们在 resources 目录下，再添加两个配置文件，分别为 application-a.properties 以及 application-b.properties:





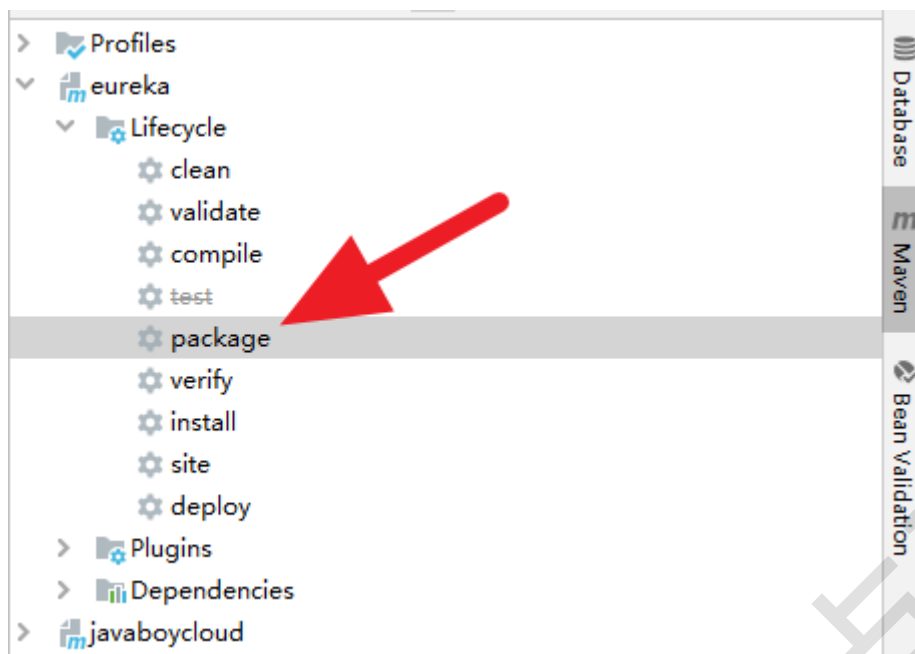
application-a.properties 内如如下：

```
# 给当前服务取一个名字
spring.application.name=eureka
# 设置端口号
server.port=1111
eureka.instance.hostname=eurekaA
# 默认情况下，Eureka Server 也是一个普通的微服务，所以当它还是一个注册中心的时候，他会有两层
身份：1.注册中心；2.普通服务，即当前服务会自己把自己注册到自己上面来
# register-with-eureka 设置为 false，表示当前项目不要注册到注册中心上
eureka.client.register-with-eureka=true
# 表示是否从 Eureka Server 上获取注册信息
eureka.client.fetch-registry=true
# A 服务要注册到 B 上面
eureka.client.service-url.defaultZone=http://eurekaB:1112/eureka
```

application-b.properties 内如如下：

```
# 给当前服务取一个名字
spring.application.name=eureka
# 设置端口号
server.port=1112
eureka.instance.hostname=eurekaB
# 默认情况下，Eureka Server 也是一个普通的微服务，所以当它还是一个注册中心的时候，他会有两层
身份：1.注册中心；2.普通服务，即当前服务会自己把自己注册到自己上面来
# register-with-eureka 设置为 false，表示当前项目不要注册到注册中心上
eureka.client.register-with-eureka=true
# 表示是否从 Eureka Server 上获取注册信息
eureka.client.fetch-registry=true
eureka.client.service-url.defaultZone=http://eurekaA:1111/eureka
```

配置完成后，对当前项目打包，打成 jar 包：



打包完成后，在命令行启动两个 Eureka 实例。两个启动命令分别如下：

```
java -jar eureka-0.0.1-SNAPSHOT.jar --spring.profiles.active=a
java -jar eureka-0.0.1-SNAPSHOT.jar --spring.profiles.active=b
```

启动成功后，就可以看到，两个服务之间互相注册，共同给组成一个集群。

## 5.3 Eureka 工作细节

Eureka 本身可以分为两大部分，Eureka Server 和 Eureka Client

### 5.3.1 Eureka Server

Eureka Server 主要对外提供了三个功能：

1. 服务注册，所有的服务都注册到 Eureka Server 上面来
2. 提供注册表，注册表就是所有注册上来服务的一个列表，Eureka Client 在调用服务时，需要获取这个注册表，一般来说，这个注册表会缓存下来，如果缓存失效，则直接获取最新的注册表
3. 同步状态，Eureka Client 通过注册、心跳等机制，和 Eureka Server 同步当前客户端的状态

### 5.3.2 Eureka Client

Eureka Client 主要是用来简化每一个服务和 Eureka Server 之间的交互。Eureka Client 会自动拉取、更新以及缓存 Eureka Server 中的信息，这样，即使 Eureka Server 所有节点都宕机，Eureka Client 依然能够获取到想要调用服务的地址（但是地址可能不准确）。

#### 5.3.2.1 服务注册

服务提供者将自己注册到服务注册中心（Eureka Server），需要注意，所谓的服务提供者，只是一个业务上的划分，本质上他就是一个 Eureka Client。当 Eureka Client 向 Eureka Server 注册时，他需要提供自身的一些元数据信息，例如 IP 地址、端口、名称、运行状态等等。

#### 5.3.2.2 服务续约

Eureka Client 注册到 Eureka Server 上之后，事情没有结束，刚刚开始而已。注册成功后，默认情况下，Eureka Client 每隔 30 秒就要向 Eureka Server 发送一条心跳消息，来告诉 Eureka Server 我还在运行。如果 Eureka Server 连续 90 秒都没有收到 Eureka Client 的续约消息（连续三次没发送），它会认为 Eureka Client 已经掉线了，会将掉线的 Eureka Client 从当前的服务注册列表中剔除。

服务续约，有两个相关的属性（一般不建议修改）：

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

- eureka.instance.lease-renewal-interval-in-seconds 表示服务的续约时间，默认是 30 秒
- eureka.instance.lease-expiration-duration-in-seconds 服务失效时间，默认是 90 秒

### 5.3.2.3 服务下线

当 Eureka Client 下线时，它会主动发送一条消息，告诉 Eureka Server，我下线啦。

### 5.3.2.4 获取注册表信息

Eureka Client 从 Eureka Server 上获取服务的注册信息，并将其缓存在本地。本地客户端，在需要调用远程服务时，会从该信息中查找远程服务所对应的 IP 地址、端口等信息。Eureka Client 上缓存的服务注册信息会定期更新(30 秒)，如果 Eureka Server 返回的注册表信息与本地缓存的注册表信息不同的话，Eureka Client 会自动处理。

这里，也涉及到两个属性，一个是是否允许获取注册表信息：

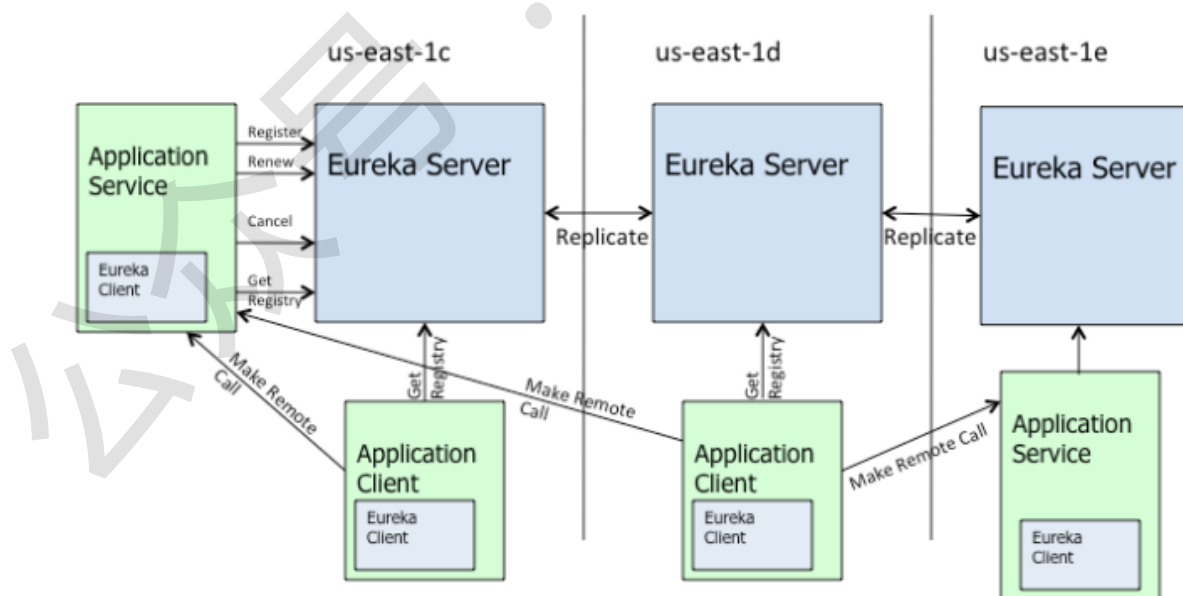
```
eureka.client.fetch-registry=true
```

Eureka Client 上缓存的服务注册信息，定期更新的时间间隔，默认 30 秒：

```
eureka.client.registry-fetch-interval-seconds=30
```

## 5.4 Eureka 集群原理

我们来看官方的一张 Eureka 集群架构图：



在这个集群架构中，Eureka Server 之间通过 Replicate 进行数据同步，不同的 Eureka Server 之间不区分主从节点，所有节点都是平等的。节点之间，通过置顶 serviceUrl 来互相注册，形成一个集群，进而提高节点的可用性。

在 Eureka Server 集群中，如果有某一个节点宕机，Eureka Client 会自动切换到新的 Eureka Server 上。每一个 Eureka Server 节点，都会互相同步数据。Eureka Server 的连接方式，可以是单线的，就是 A-->b-->C，此时，A 的数据也会和 C 之间互相同步。但是一般不建议这种写法，在我们配置 `serviceUrl` 时，可以指定多个注册地址，即 A 可以即注册到 B 上，也可以同时注册到 C 上。

Eureka 分区：

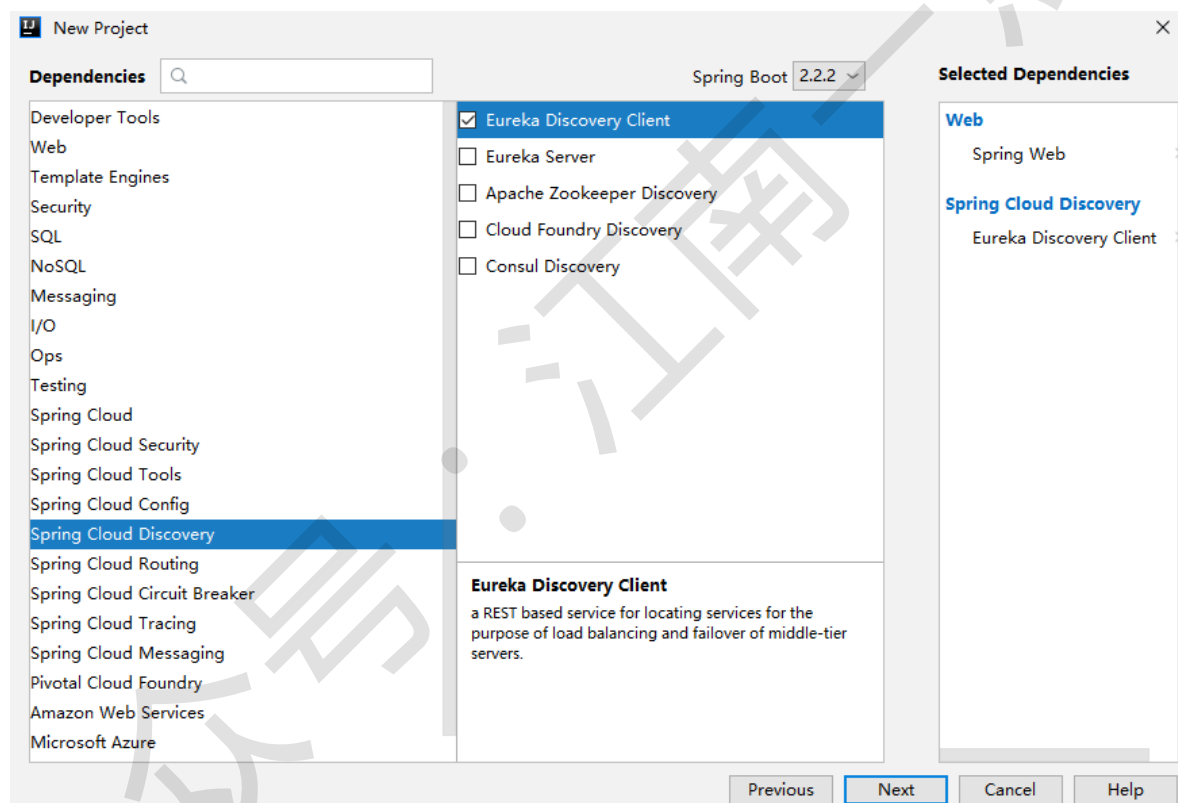
1. region：地理上的不同区域
2. zone：具体的机房

## 6. 服务注册与消费

### 6.1 服务注册

服务注册就是把一个微服务注册到 Eureka Server 上，这样，当其他服务需要调用该服务时，只需要从 Eureka Server 上查询该服务的信息即可。

这里我们创建一个 provider，作为我们的服务提供者，创建项目时，选择 Eureka Client 依赖，这样，当服务创建成功后，简单配置一下，就可以被注册到 Eureka Server 上了：



项目创建成功后，我们只需要在 `application.properties` 中配置一下项目的注册地址即可。注册地址的配置，和 Eureka Server 集群的配置很像。配置如下：

```
spring.application.name=provider
server.port=1113
eureka.client.service-url.defaultZone=http://localhost:1111/eureka
```

三行配置，分别表示当前服务的名称、端口号以及服务地址。

接下来，启动 Eureka Server，待服务注册中心启动成功后，再启动 provider。

两者都启动成功后，浏览器输入 <http://localhost:1111>，就可以查看 provider 的注册信息：

springEureka

HOME    LAST 1000 SINCE STARTUP

### System Status

Environment	test	Current time	2020-01-09T20:31:12 +0800
Data center	default	Uptime	00:04
		Lease expiration enabled	false
		Renews threshold	3
		Renews (last min)	0

### DS Replicas

localhost

### Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
PROVIDER	n/a (1)	(1)	UP (1) - DESKTOP-L25CJN4:provider:1113

### General Info

## 6.2 服务消费

### 6.2.1 基本用法

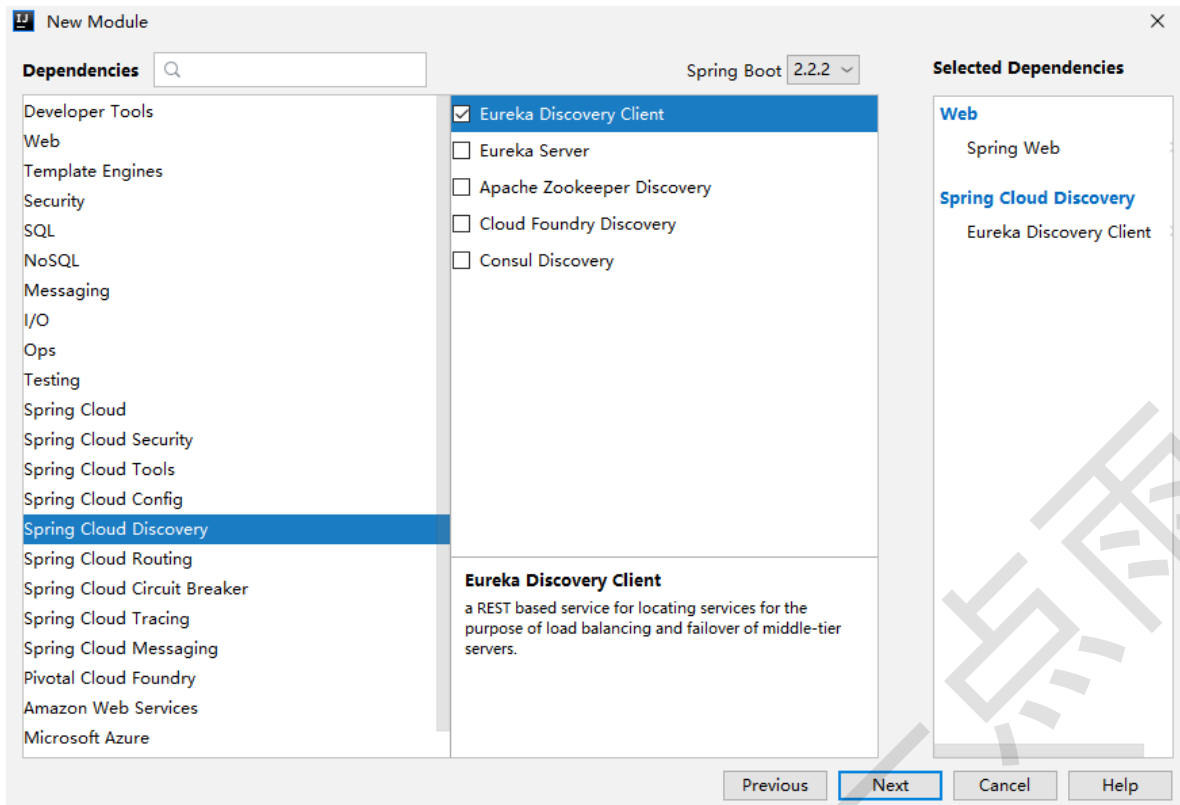
首先在 provider 中提供一个接口，然后创建一个新的 consumer 项目，消费这个接口。

在 provider 中，提供一个 hello 接口，如下：

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        return "hello javaboy";
    }
}
```

接下来，创建一个 consumer 项目，consumer 项目中，去消费 provider 提供的接口。consumer 要能够获取到 provider 这个接口的地址，他就需要去 Eureka Server 中查询，如果直接在 consumer 中写死 provider 地址，意味着这两个服务之间的耦合度就太高了，我们要降低耦合度。首先我们来看一个写死的调用。

创建一个 consumer 项目，添加 web 和 eureka client 依赖：



创建完成后，我们首先也在 application.properties 中配置一下注册信息：

```
spring.application.name=consumer
server.port=1115
eureka.client.service-url.defaultZone=http://localhost:1111/eureka
```

配置完成后，假设我们现在想在 consumer 中调用 provider 提供的服务，我们可以直接将调用写死，就是说，整个调用过程不会涉及到 Eureka Server。

```
@GetMapping("/hello1")
public String hello1() {
    HttpURLConnection con = null;
    try {
        URL url = new URL("http://localhost:1113/hello");
        con = (HttpURLConnection) url.openConnection();
        if (con.getResponseCode() == 200) {
            BufferedReader br = new BufferedReader(new
InputStreamReader(con.getInputStream()));
            String s = br.readLine();
            br.close();
            return s;
        }
    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return "error";
}
```

这是一段利用了 HttpURLConnection 来发起的请求，请求中 provider 的地址写死了，意味着 provider 和 consumer 高度绑定在一起，这个不符合微服务的思想。

要改造它，我们可以借助 Eureka Client 提供的 `DiscoveryClient` 工具，利用这个工具，我们可以根据服务名从 Eureka Server 上查询到一个服务的详细信息，改造后的代码如下：

```
@Autowired
DiscoveryClient discoveryClient;
@GetMapping("/hello2")
public String hello2() {
    List<ServiceInstance> list = discoveryClient.getInstances("provider");
    ServiceInstance instance = list.get(0);
    String host = instance.getHost();
    int port = instance.getPort();
    StringBuffer sb = new StringBuffer();
    sb.append("http://")
        .append(host)
        .append(":")
        .append(port)
        .append("/hello");
    HttpURLConnection con = null;
    try {
        URL url = new URL(sb.toString());
        con = (HttpURLConnection) url.openConnection();
        if (con.getResponseCode() == 200) {
            BufferedReader br = new BufferedReader(new
InputStreamReader(con.getInputStream()));
            String s = br.readLine();
            br.close();
            return s;
        }
    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return "error";
}
```

注意，`DiscoveryClient` 查询到的服务列表是一个集合，因为服务在部署的过程中，可能是集群化部署，集合中的每一项就是一个实例。

这里我们可以稍微展示一下集群化部署。

首先，修改 `provider` 中的 `hello` 接口：

```
@RestController
public class HelloController {
    @Value("${server.port}")
    Integer port;
    @GetMapping("/hello")
    public String hello() {
        return "hello javaboy:" + port;
    }
}
```

因为我一会会启动多个 `provider` 实例，多个 `provider` 实例的端口不同，为了区分调用时到底是哪一个 `provider` 提供的服务，这里在接口返回值中返回端口。

修改完成后，对 `provider` 进行打包。`provider` 打包成功之后，我们在命令行启动两个 `provider` 实例：

```
java -jar provider-0.0.1-SNAPSHOT.jar --server.port=1113
java -jar provider-0.0.1-SNAPSHOT.jar --server.port=1116
```

启动完成后，检查 Eureka Server 上，这两个 provider 是否成功注册上来。

注册成功后，在 consumer 中再去调用 provider，DiscoveryClient 集合中，获取到的就不是一个实例了，而是两个实例。这里我们可以手动实现一个负载均衡：

```
int count = 0;
@GetMapping("/hello3")
public String hello3() {
    List<ServiceInstance> list = discoveryClient.getInstances("provider");
    ServiceInstance instance = list.get((count++) % list.size());
    String host = instance.getHost();
    int port = instance.getPort();
    StringBuffer sb = new StringBuffer();
    sb.append("http://")
        .append(host)
        .append(":")
        .append(port)
        .append("/hello");
    HttpURLConnection con = null;
    try {
        URL url = new URL(sb.toString());
        con = (HttpURLConnection) url.openConnection();
        if (con.getResponseCode() == 200) {
            BufferedReader br = new BufferedReader(new
InputStreamReader(con.getInputStream()));
            String s = br.readLine();
            br.close();
            return s;
        }
    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return "error";
}
```

在从集合中，获取数据时，通过一个小小举动，就可以实现线性负载均衡。

## 6.2.2 升级改造

从两个方面进行改造：

1. Http 调用
2. 负载均衡

Http 调用，我们使用 Spring 提供的 RestTemplate 来实现。

首先，在当前服务中，提供一个 RestTemplate 的实例：



```

@SpringBootApplication
public class ConsumerApplication {

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

    @Bean
    RestTemplate restTemplateOne() {
        return new RestTemplate();
    }

}

```

然后，在 Http 调用时，不再使用 HttpURLConnection，而是直接使用 RestTemplate：

```

@Autowired
RestTemplate restTemplate;
@Autowired
DiscoveryClient discoveryClient;
@GetMapping("/hello2")
public String hello2() {
    List<ServiceInstance> list = discoveryClient.getInstances("provider");
    ServiceInstance instance = list.get(0);
    String host = instance.getHost();
    int port = instance.getPort();
    StringBuffer sb = new StringBuffer();
    sb.append("http://")
        .append(host)
        .append(":")
        .append(port)
        .append("/hello");
    String s = restTemplate.getForObject(sb.toString(), String.class);
    return s;
}

```

用 RestTemplate，一行代码就实现了 Http 调用。

接下来，使用 Ribbon 来快速实现负载均衡。

首先，我们需要给 RestTemplate 实例添加一个 @LoadBalanced 注解，开启负载均衡：

```

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

```

此时的 RestTemplate 就自动具备了负载均衡的功能。

此时的调用代码如下：

```

@Autowired
@Qualifier("restTemplate")
RestTemplate restTemplate;
@GetMapping("/hello3")
public String hello3() {
    return restTemplate.getForObject("http://provider/hello", String.class);
}

```

Java 中关于 Http 请求的工具实际上非常多，自带的 HttpURLConnection，古老的 HttpClient，后起之秀 OkHttp 等，除了这些之外，还有一个好用的工具--RestTemplate，这是 Spring 中就开始提供的 Http 请求工具，不过很多小伙伴们可能是因为 Spring Cloud 才听说它。今天我们就来聊一聊这个 RestTemplate。

## 6.3 RestTemplate

RestTemplate 是从 Spring3.0 开始支持的一个 Http 请求工具，这个请求工具和 Spring Boot 无关，更和 Spring Cloud 无关。RestTemplate 提供了常见的 REST 请求方法模板，例如 GET、POST、PUT、DELETE 请求以及一些通用的请求执行方法 exchange 和 execute 方法。

RestTemplate 本身实现了 RestOperations 接口，而在 RestOperations 接口中，定义了常见的 RESTful 操作，这些操作在 RestTemplate 中都得到了很好的实现。

### 6.3.1 GET

首先我们在 provider 中定义一个 hello2 接口：

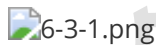
```

@GetMapping("/hello2")
public String hello2(String name) {
    return "hello " + name;
}

```

接下来，我们在 consumer 去访问这个接口，这个接口是一个 GET 请求，所以，访问方式，就是调用 RestTemplate 中的 GET 请求。

可以看到，在 RestTemplate 中，关于 GET 请求，一共有如下两大类方法：



这两大类方法实际上是重载的，唯一不同的，就是返回值类型。

getForObject 返回的是一个对象，这个对象就是服务端返回的具体值。getForEntity 返回的是一个 ResponseEntity，这个 ResponseEntity 中除了服务端返回的具体数据外，还保留了 Http 响应头的数

```

@GetMapping("/hello4")
public void hello4() {
    String s1 = restTemplate.getForObject("http://provider/hello2?name={1}",
    String.class, "javaboy");
    System.out.println(s1);
    ResponseEntity<String> responseEntity =
    restTemplate.getForEntity("http://provider/hello2", String.class, "javaboy");
    String body = responseEntity.getBody();
    System.out.println("body:"+body);
    HttpStatus statusCode = responseEntity.getStatusCode();
    System.out.println("HttpStatus:"+statusCode);
}

```

```

int statusCodeValue = responseEntity.getStatusCodeValue();
System.out.println("statusCodeValue:"+statusCodeValue);
HttpHeaders headers = responseEntity.getHeaders();
Set<String> keySet = headers.keySet();
System.out.println("-----header-----");
for (String s : keySet) {
    System.out.println(s+": "+headers.get(s));
}
}

```

这里大家可以看到，getForObject 直接拿到了服务的返回值，getForEntity 不仅仅拿到服务的返回值，还拿到 http 响应的状态码。然后，启动 Eureka Server、provider 以及 consumer，访问 consumer 中的 hello4 接口，既可以看到请求结果。

看清楚两者的区别之后，接下来看下两个各自的重载方法，getForObject 和 getForEntity 分别有三个重载方法，两者的三个重载方法基本都是一致的。所以，这里，我们主要看其中一种。三个重载方法，其实代表了三种不同的传参方式。

```

@GetMapping("/hello5")
public void hello5() throws UnsupportedEncodingException {
    String s1 = restTemplate.getForObject("http://provider/hello2?name={1}",
String.class, "javaboy");
    System.out.println(s1);
    Map<String, Object> map = new HashMap<>();
    map.put("name", "zhangsan");
    s1 = restTemplate.getForObject("http://provider/hello2?name={name}",
String.class, map);
    System.out.println(s1);
    String url = "http://provider/hello2?name=" + URLEncoder.encode("张三", "UTF-8");
    URI uri = URI.create(url);
    s1 = restTemplate.getForObject(uri, String.class);
    System.out.println(s1);
}

```

这就是我们说的三种不同的传参方式。

### 6.3.2 POST

首先在 provider 中提供两个 POST 接口，同时，因为 POST 请求可能需要传递 JSON，所以，这里我们创建一个普通的 Maven 项目作为 commons 模块，然后这个 commons 模块被 provider 和 consumer 共同引用，这样我们就可以方便的传递 JSON 了。

commons 模块创建成功后，首先在 commons 模块中添加 User 对象，然后该模块分别被 provider 和 consumer 引用。

然后，我们在 provider 中，提供和两个 POST 接口：

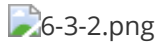
```

@PostMapping("/user1")
public User addUser1(User user) {
    return user;
}
@PostMapping("/user2")
public User addUser2(@RequestBody User user) {
    return user;
}

```

这里定义了两个 User 添加的方法，两个方法代表了两种不同的传参方式。第一种方法是以 key/value 形式来传参，第二种方法是以 JSON 形式来传参。

定义完成后，接下来，我们在 consumer 中调用这两个 POST 接口。



可以看到，这里的 post 和前面的 get 非常像，只是多出来了三个方法，就是 postForLocation，另外两个 postForObject 和 postForEntity 和前面 get 基本一致，所以这里我们主要来看 postForObject，看完之后，我们再来看这个额外的 postForLocation。

```
@GetMapping("/hello6")
public void hello6() {
    MultivalueMap<String, Object> map = new LinkedMultivalueMap<>();
    map.add("username", "javaboy");
    map.add("password", "123");
    map.add("id", 99);
    User user = restTemplate.postForObject("http://provider/user1", map,
    User.class);
    System.out.println(user);
    user.setId(98);
    user = restTemplate.postForObject("http://provider/user2", user,
    User.class);
    System.out.println(user);
}
```

post 参数到底是 key/value 形式还是 json 形式，主要看第二个参数，如果第二个参数是 MultivalueMap，则参数是以 key/value 形式来传递的，如果是一个普通对象，则参数是以 json 形式来传递的。

最后再看一下 postForLocation。有的时候，当我执行完一个 post 请求之后，立马要进行重定向，一个非常常见的场景就是注册，注册是一个 post 请求，注册完成之后，立马重定向到登录页面去登录。对于这种场景，我们就可以使用 postForLocation。

首先我们在 provider 上提供一个用户注册接口：

```
@Controller
public class RegisterController {
    @PostMapping("/register")
    public String register(User user) {
        return "redirect:http://provider/loginPage?username=" +
        user.getUsername();
    }

    @GetMapping("/loginPage")
    @ResponseBody
    public String loginPage(String username) {
        return "loginPage:" + username;
    }
}
```

注意，这里的 post 接口，响应一定是 302，否则 postForLocation 无效。

注意，重定向的地址，一定要写成绝对路径，不要写相对路径，否则在 consumer 中调用时会出问题

```

@GetMapping("/hello7")
public void hello7() {
    MultiValueMap<String, Object> map = new LinkedMultiValueMap<>();
    map.add("username", "javaboy");
    map.add("password", "123");
    map.add("id", 99);
    URI uri = restTemplate.postForLocation("http://provider/register", map);
    String s = restTemplate.getForObject(uri, String.class);
    System.out.println(s);
}

```

这就是 postForLocation，调用该方法返回的是一个 Uri，这个 Uri 就是重定向的地址（里边也包含了重定向的参数），拿到 Uri 之后，就可以直接发送新的请求了。

### 6.3.3 PUT

PUT 请求比较简单，重载的方法也比较少。

我们首先在 provider 中提供一个 PUT 接口：

```

@PutMapping("/user1")
public void updateUser1(User user) {
    System.out.println(user);
}
@PutMapping("/user2")
public void updateUser2(@RequestBody User user) {
    System.out.println(user);
}

```

注意，PUT 接口传参其实和 POST 很像，也接受两种类型的参数，key/value 形式以及 JSON 形式。

在 consumer 中，我们来调用该接口：

```

@GetMapping("/hello8")
public void hello8() {
    MultiValueMap<String, Object> map = new LinkedMultiValueMap<>();
    map.add("username", "javaboy");
    map.add("password", "123");
    map.add("id", 99);
    restTemplate.put("http://provider/user1", map);
    User user = new User();
    user.setId(98);
    user.setUsername("zhangsan");
    user.setPassword("456");
    restTemplate.put("http://provider/user1", user);
}

```

consumer 中的写法基本和 post 类似，也是两种方式，可以传递两种不同类型的参数。

### 6.3.4 DELETE

DELETE 也比较容易，我们有两种方式来传递参数，key/value 形式或者 PathVariable（参数放在路径中），首先我们在 provider 中定义两个 DELETE 方法：

```
@DeleteMapping("/user1")
public void deleteUser1(Integer id) {
    System.out.println(id);
}
@DeleteMapping("/user2/{id}")
public void deleteUser2(@PathVariable Integer id) {
    System.out.println(id);
}
```

然后在 consumer 中调用这两个删除的接口：

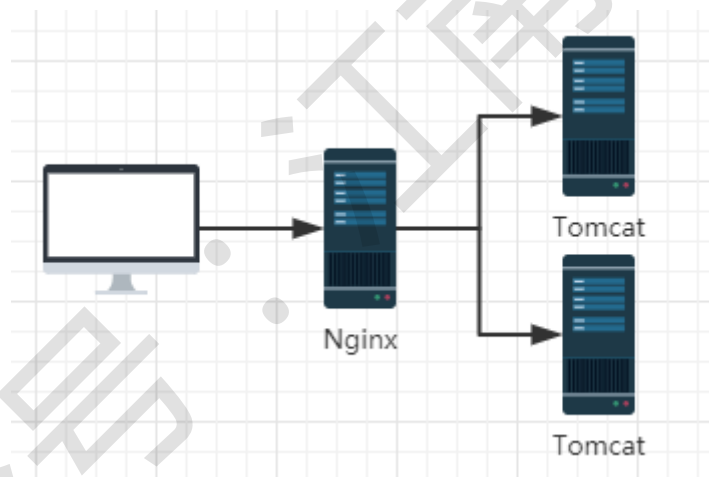
```
@GetMapping("/hello9")
public void hello9() {
    restTemplate.delete("http://provider/user1?id={1}", 99);
    restTemplate.delete("http://provider/user2/{1}", 99);
}
```

delete 中参数的传递，也支持 map，这块实际上和 get 是一样的。

## 6.4 客户端负载均衡

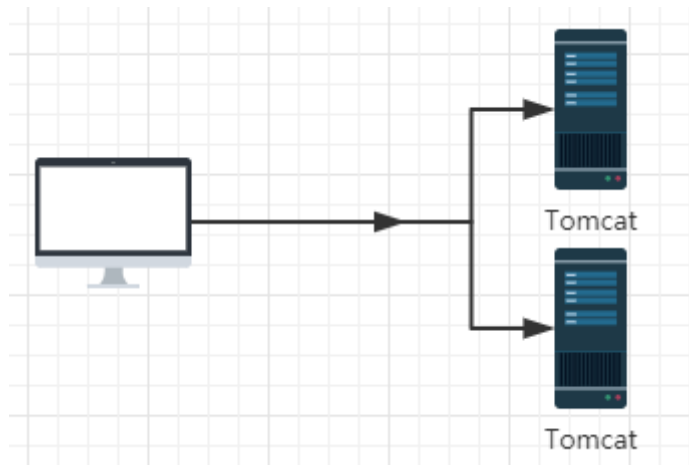
客户端负载均衡就是相对服务端负载均衡而言的。

服务端负载均衡，就是传统的 Nginx 的方式，用 Nginx 做负载均衡，我们称之为服务端负载均衡：



这种负载均衡，我们称之为服务端负载均衡，它的一个特点是，就是调用的客户端并不知道具体是哪一个 Server 提供的服务，它也不关心，反正请求发送给 Nginx，Nginx 再将请求转发给 Tomcat，客户端只需要记着 Nginx 的地址即可。

客户端负载均衡则是另外一种情形：



客户端负载均衡，就是调用的客户端本身是知道所有 Server 的详细信息的，当需要调用 Server 上的接口的时候，客户端从自身所维护的 Server 列表中，根据提前配置好的负载均衡策略，自己挑选一个 Server 来调用，此时，客户端知道它所调用的是哪一个 Server。

在 RestTemplate 中，要想使用负载均衡功能，只需要给 RestTemplate 实例上添加一个 @LoadBalanced 注解即可，此时，RestTemplate 就会自动具备负载均衡功能，这个负载均衡就是客户端负载均衡。

## 6.5 负载均衡原理

在 Spring Cloud 中，实现负载均衡非常容易，只需要添加 @LoadBalanced 注解即可。只要添加了该注解，一个原本普普通通做 Rest 请求的工具 RestTemplate 就会自动具备负载均衡功能，这个是怎么实现的呢？

整体上来说，这个功能的实现就是三个核心点：

1. 从 Eureka Client 本地缓存的服务注册信息中，选择一个可以调用的服务
2. 根据 1 中所选择的服务，重构请求 URL 地址
3. 将 1、2 步的功能嵌入到 RestTemplate 中

## 7. Consul

在 Spring Cloud 中，大部分组件都有备选方案，例如注册中心，除了常见 Eureka 之外，像 zookeeper 我们也可以直接使用在 Spring Cloud 中，还有另外一个比较重要的方案，就是 Consul。

Consul 是 HashiCorp 公司推出来的开源产品。主要提供了：服务发现、服务隔离、服务配置等功能。

相比于 Eureka 和 zookeeper，Consul 配置更加一站式，因为它内置了很多微服务常见的需求：服务发现与注册、分布式一致性协议实现、健康检查、键值对存储、多数据中心等，我们不再需要借助第三方组件来实现这些功能。

### 7.1 安装

不同于 Eureka，Consul 使用 Go 语言开发，所以，使用 Consul，我们需要先安装软件。

在 Linux 中，首先执行如下命令下载 Consul：

```
wget https://releases.hashicorp.com/consul/1.6.2/consul_1.6.2_linux_amd64.zip
```

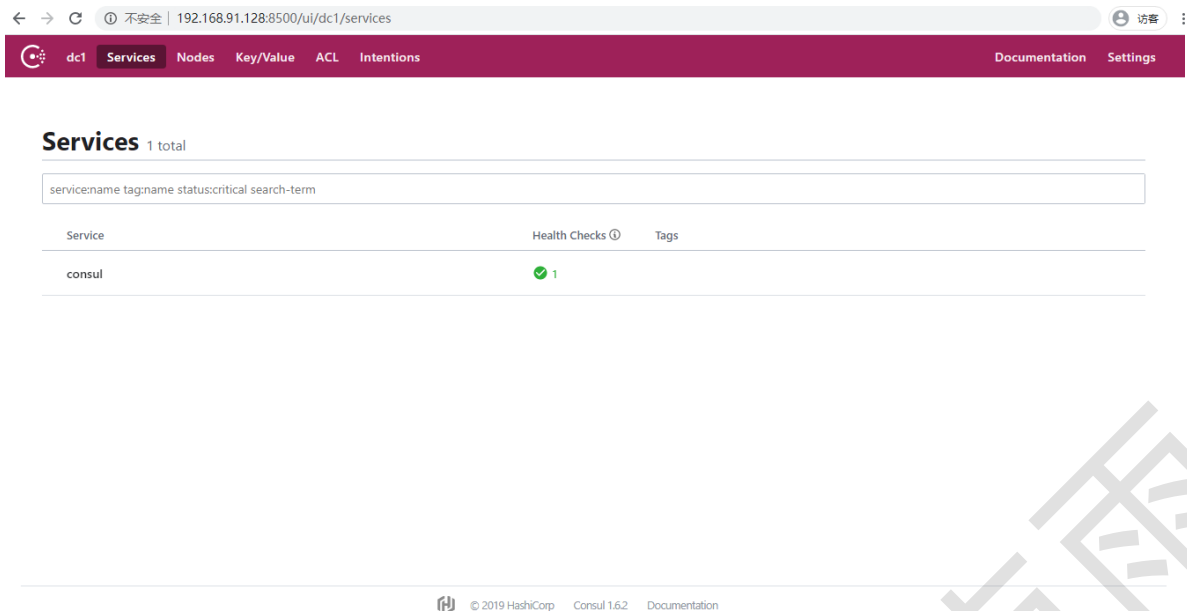
然后解压下载文件：

```
unzip consul_1.6.2_linux_amd64.zip
```

解压完成后，我们在当前目录下就可以看到 consul 文件，然后执行如下命令，启动 Consul：

```
./consul agent -dev -ui -node=consul-dev -client=192.168.91.128
```

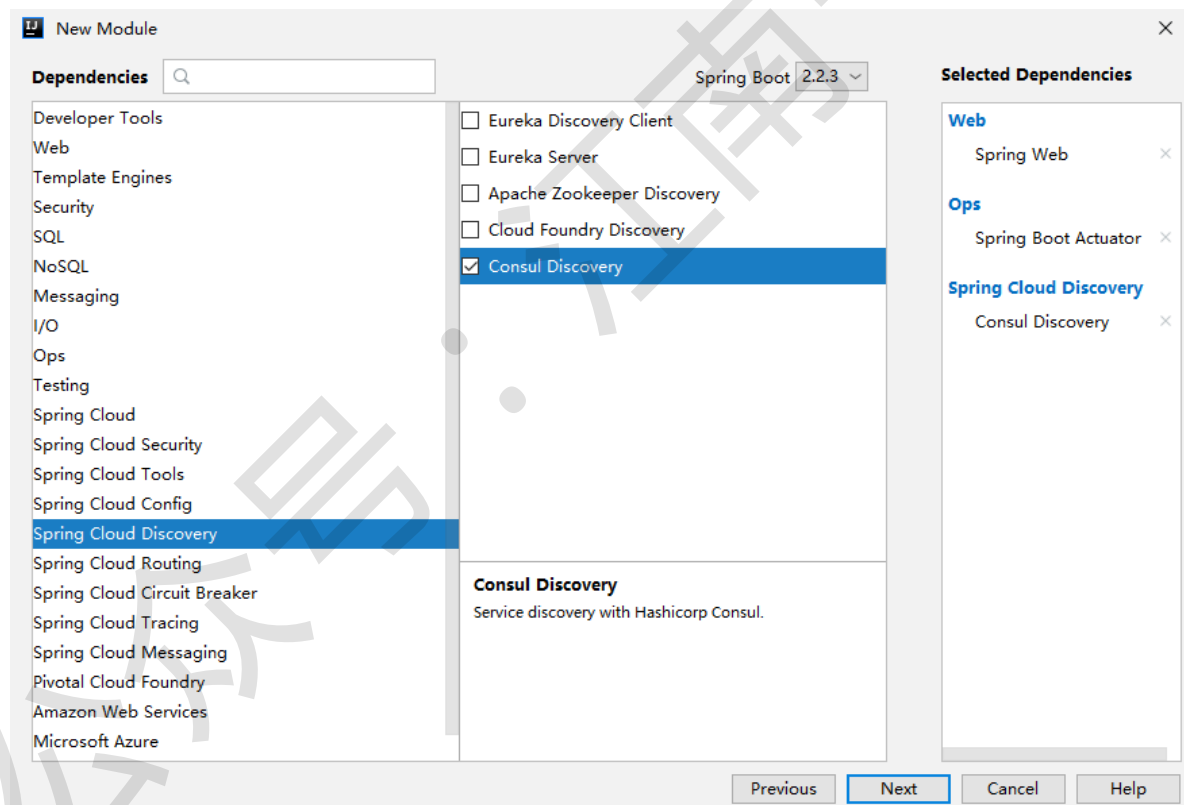
启动成功后，在物理机中，我们可以直接访问 Consul 的后台管理页面（注意，这个访问要确保 8500 端口可用，或者直接关闭防火墙）：



## 7.2 Consul 使用

简单看一个注册消费的案例。

首先我们来创建一个服务提供者。就是一个普通的 Spring Boot 项目，添加如下依赖：



项目创建成功后，添加如下配置：

```
spring.application.name=consul-provider
server.port=2000
# Consul 相关配置
spring.cloud.consul.host=192.168.91.128
spring.cloud.consul.port=8500
spring.cloud.consul.discovery.service-name=consul-provider
```

在项目启动类上开启服务发现的功能：



```

@SpringBootApplication
@EnableDiscoveryClient
public class ConsulProviderApplication {

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

}

```

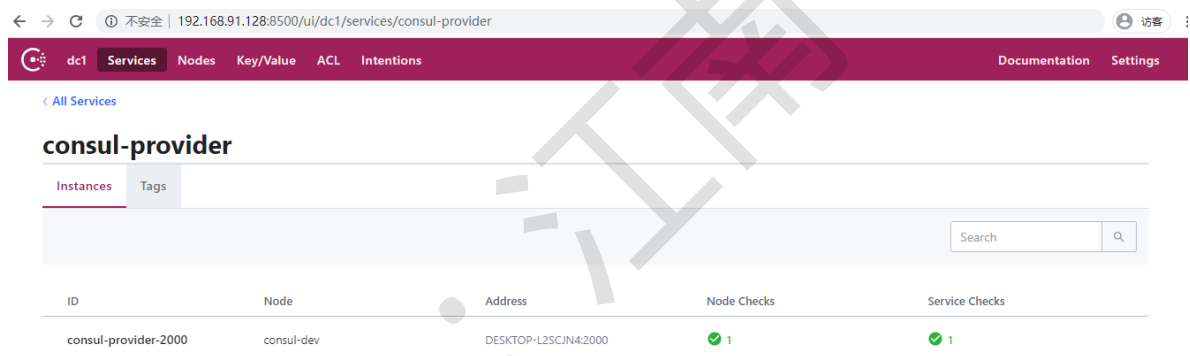
最后添加一个测试接口：

```

@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        return "hello";
    }
}

```

接下来就是启动项目，项目启动成功后，访问 consul 后台管理页面，看到如下信息，表示 consul 已经注册成功了。



## 7.3 Consul 集群注册

为了区分集群中的哪一个 provider 提供的服务，我们修改一下 consul 中的接口：

```

@RestController
public class HelloController {
    @Value("${server.port}")
    Integer port;

    @GetMapping("/hello")
    public String hello() {
        return "hello>>" + port;
    }
}

```

修改完成后，对项目进行打包。打包成功后，命令行执行如下两行命令，启动两个 provider 实例：

```

java -jar consul-provider-0.0.1-SNAPSHOT.jar --server.port=2000
java -jar consul-provider-0.0.1-SNAPSHOT.jar --server.port=2001

```

启动成功后，再去 consul 后台管理页面，就可以看到有两个实例了：

← → ↻ ① 不安全 | 192.168.91.128:8500/ui/dc1/services/consul-provider 访客

dc1 Services Nodes Key/Value ACL Intentions Documentation Settings

< All Services

consul-provider

Instances Tags

🔍

ID	Node	Address	Node Checks	Service Checks
consul-provider-2000	consul-dev	DESKTOP-L2SCJN4:2000	✔ 1	✔ 1
consul-provider-2001	consul-dev	DESKTOP-L2SCJN4:2001	✔ 1	✔ 1

## 7.4 消费

首先创建一个消费实例，创建方式和 provider 一致。

创建成功后，添加如下配置：

```
spring.application.name=consul-consumer
server.port=2002
spring.cloud.consul.host=192.168.91.128
spring.cloud.consul.port=8500
spring.cloud.consul.discovery.service-name=consul-consumer
```

开启服务发现，并添加 RestTemplate：

```
@SpringBootApplication
@EnableDiscoveryClient
public class ConsulConsumerApplication {

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

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

最后，提供一个服务调用的方法：

```
@RestController
public class HelloController {

    @Autowired
    LoadBalancerClient loadBalancerClient;

    @Autowired
    RestTemplate restTemplate;

    @GetMapping("/hello")
    public void hello() {
        ServiceInstance choose = loadBalancerClient.choose("consul-provider");
        System.out.println("服务地址: " + choose.getUri());
        System.out.println("服务名称: " + choose.getServiceId());
    }
}
```

```
String s = restTemplate.getForObject(choose.getUri() + "/hello",
String.class);
System.out.println(s);
}
}
```

这里，我们通过 `loadBalancerClient` 实例，可以获取要调用的 `ServiceInstance`。获取到调用地址之后，再用 `RestTemplate` 去调用。

然后，启动项目，浏览器输入 <http://localhost:2002/hello>，查看请求结果，这个请求自带负载均衡功能。

## 8. Hystrix

1. 基本介绍
2. 简单使用/容错/服务降级
3. 请求命令
4. 异常处理
5. 请求缓存
6. 请求合并

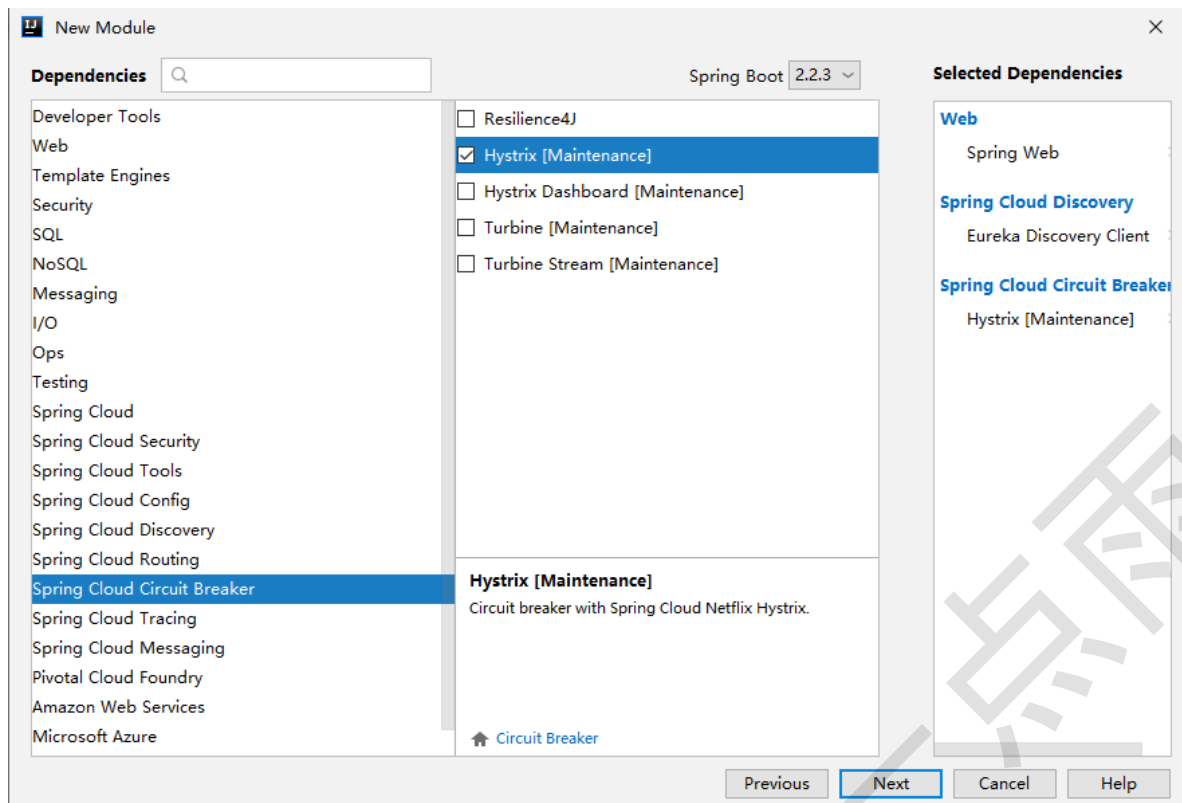
### 8.1 基本介绍

Hystrix 叫做断路器/熔断器。微服务系统中，整个系统出错的概率非常高，因为在微服务系统中，涉及到的模块太多了，每一个模块出错，都有可能整个服务出，当所有模块都稳定运行时，整个服务才算是稳定运行。

我们希望当整个系统中，某一个模块无法正常工作，能够通过我们提前配置的一些东西，来使得整个系统正常运行，即单个模块出问题，不影响整个系统。

### 8.2 基本用法

首先创建一个新的 SpringBoot 模块，然后添加依赖：



项目创建成功后，添加如下配置，将 Hystrix 注册到 Eureka 上：

```
spring.application.name=hystrix
server.port=3000
eureka.client.service-url.defaultZone=http://localhost:1111/eureka
```

然后，在项目启动类上添加如下注解，开启断路器，同时提供一个 RestTemplate 实例：

```
@SpringBootApplication
@EnableCircuitBreaker
public class HystrixApplication {

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

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

启动类上的注解，也可以使用 @SpringCloudApplication 代替：

```

@SpringBootApplication
public class HystrixApplication {

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

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

```

这样，Hystrix 的配置就算完成了。

接下来提供 Hystrix 的接口。

```

@Service
public class HelloService {
    @Autowired
    RestTemplate restTemplate;

    /**
     * 在这个方法中，我们将发起一个远程调用，去调用 provider 中提供的 /hello 接口
     *
     * 但是，这个调用可能会失败。
     *
     * 我们在这个方法上添加 @HystrixCommand 注解，配置 fallbackMethod 属性，这个属性表示
     该方法调用失败时的临时替代方法
     * @return
     */
    @HystrixCommand(fallbackMethod = "error")
    public String hello() {
        return restTemplate.getForObject("http://provider/hello", String.class);
    }

    /**
     * 注意，这个方法名字要和 fallbackMethod 一致
     * 方法返回值也要和对应的方法一致
     * @return
     */
    public String error() {
        return "error";
    }
}

@RestController
public class HelloController {
    @Autowired
    HelloService helloService;

    @GetMapping("/hello")
    public String hello() {
        return helloService.hello();
    }
}

```

## 8.3 请求命令

请求命令就是以继承类的方式来替代前面的注解方式。

我们来自定义一个 HelloCommand:

```
public class HelloCommand extends HystrixCommand<String> {

    RestTemplate restTemplate;

    public HelloCommand(Setter setter, RestTemplate restTemplate) {
        super(setter);
        this.restTemplate = restTemplate;
    }

    @Override
    protected String run() throws Exception {
        return restTemplate.getForObject("http://provider/hello", String.class);
    }
}
```

调用方法:

```
@GetMapping("/hello2")
public void hello2() {
    HelloCommand helloCommand = new
    HelloCommand(HystrixCommand.Setter.withGroupKey(HystrixCommandGroupKey.Factory.a
    sKey("javaboy")), restTemplate);
    String execute = helloCommand.execute(); // 直接执行
    System.out.println(execute);
    HelloCommand helloCommand2 = new
    HelloCommand(HystrixCommand.Setter.withGroupKey(HystrixCommandGroupKey.Factory.a
    sKey("javaboy")), restTemplate);
    try {
        Future<String> queue = helloCommand2.queue();
        String s = queue.get();
        System.out.println(s); // 先入队, 后执行
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}
```

注意:

1. 一个实例只能执行一次
2. 可以直接执行, 也可以先入队, 后执行

**号外: 通过注解实现请求异步调用**

首先, 定义如下方法, 返回 `Future<String>`:

```

@HystrixCommand(fallbackMethod = "error")
public Future<String> hello2() {
    return new AsyncResult<String>() {
        @Override
        public String invoke() {
            return restTemplate.getForObject("http://provider/hello",
String.class);
        }
    };
}

```

然后，调用该方法：

```

@GetMapping("/hello3")
public void hello3() {
    Future<String> hello2 = helloService.hello2();
    try {
        String s = hello2.get();
        System.out.println(s);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}

```

通过继承的方式使用 Hystrix，如何实现服务容错/降级？重写继承类的 getFallback 方法即可：

```

public class HelloCommand extends HystrixCommand<String> {

    RestTemplate restTemplate;

    public HelloCommand(Setter setter, RestTemplate restTemplate) {
        super(setter);
        this.restTemplate = restTemplate;
    }

    @Override
    protected String run() throws Exception {
        return restTemplate.getForObject("http://provider/hello", String.class);
    }

    /**
     * 这个方法就是请求失败的回调
     *
     * @return
     */
    @Override
    protected String getFallback() {
        return "error-extends";
    }
}

```

## 8.4 异常处理

就是当发起服务调用时，如果不是 provider 的原因导致请求调用失败，而是 consumer 中本身代码有问题导致的请求失败，即 consumer 中抛出了异常，这个时候，也会自动进行服务降级，只不过这个时候降级，我们还需要知道到底是哪里出异常了。

如下示例代码，如果 hello 方法中，执行时抛出异常，那么一样也会进行服务降级，进入到 error 方法中，在 error 方法中，我们可以获取到异常的详细信息：

```
@Service
public class HelloService {
    @Autowired
    RestTemplate restTemplate;

    /**
     * 在这个方法中，我们将发起一个远程调用，去调用 provider 中提供的 /hello 接口
     * <p>
     * 但是，这个调用可能会失败。
     * <p>
     * 我们在这个方法上添加 @HystrixCommand 注解，配置 fallbackMethod 属性，这个属性表示
     该方法调用失败时的临时替代方法
     *
     * @return
     */
    @HystrixCommand(fallbackMethod = "error")
    public String hello() {
        int i = 1 / 0;
        return restTemplate.getForObject("http://provider/hello", String.class);
    }

    /**
     * 注意，这个方法名字要和 fallbackMethod 一致
     * 方法返回值也要和对应的方法一致
     *
     * @return
     */
    public String error(Throwable t) {
        return "error:" + t.getMessage();
    }
}
```

这是注解的方式。也可以通过继承的方式：

```
public class HelloCommand extends HystrixCommand<String> {
    RestTemplate restTemplate;

    public HelloCommand(Setter setter, RestTemplate restTemplate) {
        super(setter);
        this.restTemplate = restTemplate;
    }

    @Override
    protected String run() throws Exception {
        int i = 1 / 0;
        return restTemplate.getForObject("http://provider/hello", String.class);
    }
}
```



```

/**
 * 这个方法就是请求失败的回调
 *
 * @return
 */
@Override
protected String getFallback() {
    return "error-extends:"+getExecutionException().getMessage();
}
}

```

如果是通过继承的方式来做 Hystrix，在 getFallback 方法中，我们可以通过 getExecutionException 方法来获取执行的异常信息。

另一种可能性（作为了解）。如果抛异常了，我们希望异常直接抛出，不要服务降级，那么只需要配置忽略某一个异常即可：

```

@HystrixCommand(fallbackMethod = "error", ignoreExceptions =
    ArithmeticException.class)
public String hello() {
    int i = 1 / 0;
    return restTemplate.getForObject("http://provider/hello", String.class);
}

```

这个配置表示当 hello 方法抛出 ArithmeticException 异常时，不要进行服务降级，直接将错误抛出。

## 8.5 请求缓存

请求缓存就是在 consumer 中调用同一个接口，如果参数相同，则可以使用之前缓存下来的数据。

首先修改 provider 中的 hello2 接口，一会用来检测缓存配置是否生效：

```

@GetMapping("/hello2")
public String hello2(String name) {
    System.out.println(new Date() + ">>>" + name);
    return "hello " + name;
}

```

然后，在 hystrix 的请求方法中，添加如下注解：

```

@HystrixCommand(fallbackMethod = "error2")
@CacheResult//这个注解表示该方法的请求结果会被缓存起来，默认情况下，缓存的 key 就是方法的参数，缓存的 value 就是方法的返回值。
public String hello3(String name) {
    return restTemplate.getForObject("http://provider/hello2?name={1}",
        String.class, name);
}

```

这个配置完成后，缓存并不会生效，一般来说，我们使用缓存，都有一个缓存生命周期这样一个概念。这里也一样，我们需要初始化 HystrixRequestContext，初始化完成后，缓存开始生效，HystrixRequestContext close 之后，缓存失效。

```

@GetMapping("/hello4")
public void hello4() {
    HystrixRequestContext ctx = HystrixRequestContext.initializeContext();
    String javaboy = helloService.hello3("javaboy");
    javaboy = helloService.hello3("javaboy");
    ctx.close();
}

```

在 ctx close 之前，缓存是有效的，close 之后，缓存就失效了。也就是说，访问一次 hello4 接口，provider 只会被调用一次（第二次使用的缓存），如果再次调用 hello4 接口，之前缓存的数据是失效的。

默认情况下，缓存的 key 就是所调用方法的参数，如果参数有多个，就是多个参数组合起来作为缓存的 key。

例如如下方法：

```

@HystrixCommand(fallbackMethod = "error2")
@CacheResult//这个注解表示该方法的请求结果会被缓存起来，默认情况下，缓存的 key 就是方法的参数，缓存的 value 就是方法的返回值。
public String hello3(String name,Integer age) {
    return restTemplate.getForObject("http://provider/hello2?name={1}",
    String.class, name);
}

```

此时缓存的 key 就是 name+age，但是，如果有多个参数，但是又只想使用其中一个作为缓存的 key，那么我们可以通过 @CacheKey 注解来解决。

```

@HystrixCommand(fallbackMethod = "error2")
@CacheResult//这个注解表示该方法的请求结果会被缓存起来，默认情况下，缓存的 key 就是方法的参数，缓存的 value 就是方法的返回值。
public String hello3(@CacheKey String name, Integer age) {
    return restTemplate.getForObject("http://provider/hello2?name={1}",
    String.class, name);
}

```

上面这个配置，虽然有两个参数，但是缓存时以 name 为准。也就是说，两次请求中，只要 name 一样，即使 age 不一样，第二次请求也可以使用第一次请求缓存的结果。

另外还有一个注解叫做 @CacheRemove()。在做数据缓存时，如果有一个数据删除的方法，我们一般除了删除数据库中的数据，还希望能够顺带删除缓存中的数据，这个时候 @CacheRemove() 就派上用场了。

@CacheRemove() 在使用时，必须指定 commandKey 属性，commandKey 其实就是缓存方法的名字，指定了 commandKey，@CacheRemove 才能找到数据缓存存在哪里了，进而才能成功删除掉数据。

例如如下方法定义缓存与删除缓存：

```

@HystrixCommand(fallbackMethod = "error2")
@CacheResult//这个注解表示该方法的请求结果会被缓存起来，默认情况下，缓存的 key 就是方法的参数，缓存的 value 就是方法的返回值。
public String hello3(String name) {
    return restTemplate.getForObject("http://provider/hello2?name={1}",
String.class, name);
}
@HystrixCommand
@CacheRemove(commandKey = "hello3")
public String deleteUserByName(String name) {
    return null;
}

```

再去调用：

```

@GetMapping("/hello4")
public void hello4() {
    HystrixRequestContext ctx = HystrixRequestContext.initializeContext();
    //第一请求完，数据已经缓存下来了
    String javaboy = helloService.hello3("javaboy");
    //删除数据，同时缓存中的数据也会被删除
    helloService.deleteUserByName("javaboy");
    //第二次请求时，虽然参数还是 javaboy，但是缓存数据已经没了，所以这一次，provider 还是会收到请求
    javaboy = helloService.hello3("javaboy");
    ctx.close();
}

```

如果是继承的方式使用 Hystrix，只需要重写 getCacheKey 方法即可：

```

public class HelloCommand extends HystrixCommand<String> {

    RestTemplate restTemplate;
    String name;

    public HelloCommand(Setter setter, RestTemplate restTemplate,String name) {
        super(setter);
        this.name = name;
        this.restTemplate = restTemplate;
    }

    @Override
    protected String run() throws Exception {
        return restTemplate.getForObject("http://provider/hello2?name={1}",
String.class, name);
    }

    @Override
    protected String getCacheKey() {
        return name;
    }

    /**
     * 这个方法就是请求失败的回调
     *
     * @return
     */
}

```

```

    */
    @Override
    protected String getFallback() {
        return "error-extends:"+getExecutionException().getMessage();
    }
}

```

调用时候，一定记得初始化 HystrixRequestContext:

```

@GetMapping("/hello2")
public void hello2() {
    HystrixRequestContext ctx = HystrixRequestContext.initializeContext();
    HelloCommand helloCommand = new
    HelloCommand(HystrixCommand.Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("javaboy")), restTemplate,"javaboy");
    String execute = helloCommand.execute();//直接执行
    System.out.println(execute);
    HelloCommand helloCommand2 = new
    HelloCommand(HystrixCommand.Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("javaboy")), restTemplate,"javaboy");
    try {
        Future<String> queue = helloCommand2.queue();
        String s = queue.get();
        System.out.println(s);//先入队，后执行
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
    ctx.close();
}

```

## 8.6 请求合并

如果 consumer 中，频繁的调用 provider 中的同一个接口，在调用时，只是参数不一样，那么这样情况下，我们就可以将多个请求合并成一个，这样可以有效提高请求发送的效率。

首先我们在 provider 中提供一个请求合并的接口：

```

@RestController
public class UserController {

    @GetMapping("/user/{ids}")//假设 consumer 传过来的多个 id 的格式是 1,2,3,4....
    public List<User> getUserByIds(@PathVariable String ids) {
        String[] split = ids.split(",");
        List<User> users = new ArrayList<>();
        for (String s : split) {
            User u = new User();
            u.setId(Integer.parseInt(s));
            users.add(u);
        }
        return users;
    }
}

```

这个接口既可以处理合并之后的请求，也可以处理单个请求（单个请求的话，List 集合中就只有一项数据。）

然后，在 Hystrix 中，定义 UserService：

```
@Service
public class UserService {
    @Autowired
    RestTemplate restTemplate;

    public List<User> getUsersByIds(List<Integer> ids) {
        User[] users = restTemplate.getForObject("http://provider/user/{1}",
        User[].class, StringUtils.join(ids, ","));
        return Arrays.asList(users);
    }
}
```

接下来定义 UserBatchCommand，相当于我们之前的 HelloCommand：

```
public class UserBatchCommand extends HystrixCommand<List<User>> {
    private List<Integer> ids;
    private UserService userService;

    public UserBatchCommand(List<Integer> ids, UserService userService) {
        super(HystrixCommand.Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("batchCmd")).andCommandKey(HystrixCommandKey.Factory.asKey("batchKey")));
        this.ids = ids;
        this.userService = userService;
    }

    @Override
    protected List<User> run() throws Exception {
        return userService.getUsersByIds(ids);
    }
}
```

最后，定义最关键的请求合并方法：

```
public class UserCollapseCommand extends HystrixCollapser<List<User>, User, Integer> {
    private UserService userService;
    private Integer id;

    public UserCollapseCommand(UserService userService, Integer id) {
        super(HystrixCollapser.Setter.withCollapserKey(HystrixCollapserKey.Factory.asKey("UserCollapseCommand")).andCollapserPropertiesDefaults(HystrixCollapserProperties.Setter().withTimerDelayInMilliseconds(200)));
        this.userService = userService;
        this.id = id;
    }

    /**
     * 请求参数
     */
}
```

```

    * @return
    */
    @Override
    public Integer getRequestArgument() {
        return id;
    }

    /**
     * 请求合并的方法
     *
     * @param collection
     * @return
     */
    @Override
    protected HystrixCommand<List<User>>
    createCommand(Collection<CollapsedRequest<User, Integer>> collection) {
        List<Integer> ids = new ArrayList<>(collection.size());
        for (CollapsedRequest<User, Integer> userIntegerCollapsedRequest :
collection) {
            ids.add(userIntegerCollapsedRequest.getArgument());
        }
        return new UserBatchCommand(ids, userService);
    }

    /**
     * 请求结果分发
     *
     * @param users
     * @param collection
     */
    @Override
    protected void mapResponseToRequests(List<User> users,
Collection<CollapsedRequest<User, Integer>> collection) {
        int count = 0;
        for (CollapsedRequest<User, Integer> request : collection) {
            request.setResponse(users.get(count++));
        }
    }
}

```

最后就是测试调用：

```

@GetMapping("/hello5")
public void hello5() throws ExecutionException, InterruptedException {
    HystrixRequestContext ctx = HystrixRequestContext.initializeContext();
    UserCollapseCommand cmd1 = new UserCollapseCommand(userService, 99);
    UserCollapseCommand cmd2 = new UserCollapseCommand(userService, 98);
    UserCollapseCommand cmd3 = new UserCollapseCommand(userService, 97);
    UserCollapseCommand cmd4 = new UserCollapseCommand(userService, 96);
    Future<User> q1 = cmd1.queue();
    Future<User> q2 = cmd2.queue();
    Future<User> q3 = cmd3.queue();
    Future<User> q4 = cmd4.queue();
    User u1 = q1.get();
    User u2 = q2.get();
    User u3 = q3.get();
    User u4 = q4.get();
}

```

```

        System.out.println(u1);
        System.out.println(u2);
        System.out.println(u3);
        System.out.println(u4);
        ctx.close();
    }
}

```

## 通过注解实现请求合并

```

@Service
public class UserService {
    @Autowired
    RestTemplate restTemplate;

    @HystrixCommand(batchMethod = "getUsersByIds", collapseProperties =
    {@HystrixProperty(name = "timerDelayInMilliseconds", value = "200")})
    public Future<User> getUserById(Integer id) {
        return null;
    }

    @HystrixCommand
    public List<User> getUsersByIds(List<Integer> ids) {
        User[] users = restTemplate.getForObject("http://provider/user/{1}",
        User[].class, StringUtils.join(ids, ","));
        return Arrays.asList(users);
    }
}

```

这里的核心是 @HystrixCommand 注解。在这个注解中，指定批处理的方法即可。

测试代码如下：

```

@GetMapping("/hello6")
public void hello6() throws ExecutionException, InterruptedException {
    HystrixRequestContext ctx = HystrixRequestContext.initializeContext();
    Future<User> q1 = userService.getUserById(99);
    Future<User> q2 = userService.getUserById(98);
    Future<User> q3 = userService.getUserById(97);
    User u1 = q1.get();
    User u2 = q2.get();
    User u3 = q3.get();
    System.out.println(u1);
    System.out.println(u2);
    System.out.println(u3);
    Thread.sleep(2000);
    Future<User> q4 = userService.getUserById(96);
    User u4 = q4.get();
    System.out.println(u4);
    ctx.close();
}

```

## 9. OpenFeign

### 9.1 OpenFeign

前面无论是基本调用，还是 Hystrix，我们实际上都是通过手动调用 RestTemplate 来实现远程调用的。使用 RestTemplate 存在一个问题：繁琐，每一个请求，参数不同，请求地址不同，返回数据类型不同，其他都是一样的，所以我们希望能够对请求进行简化。

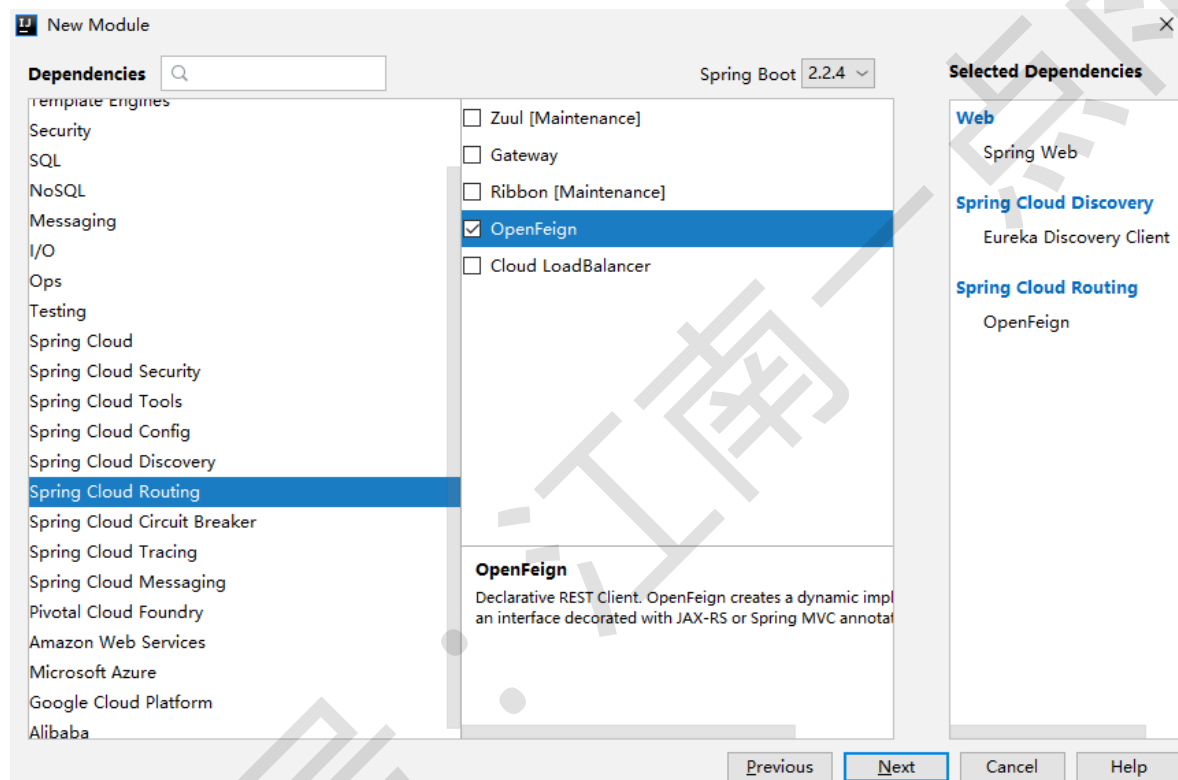
我们希望对请求进行简化，简化方案就是 OpenFeign。

一开始这个组件不叫这个名字，一开始就叫 Feign，Netflix Feign，但是 Netflix 中的组件现在已经停止开源工作，OpenFeign 是 Spring Cloud 团队在 Netflix Feign 的基础上开发出来的声明式服务调用组件。关于 OpenFeign 组件的 Issue: <https://github.com/OpenFeign/feign/issues/373>

### 9.1.1 HelloWorld

继续使用之前的 Provider。

新建一个 Spring Boot 模块，创建时，选择 OpenFeign 依赖，如下：



项目创建成功后，在 application.properties 中进行配置，使项目注册到 Eureka 上：

```
spring.application.name=openfeign
server.port=4000
eureka.client.service-url.defaultZone=http://localhost:1111/eureka
```

接下来在启动类上添加注解，开启 Feign 的支持：

```
@SpringBootApplication
@EnableFeignClients
public class OpenfeignApplication {

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

}
```

接下来，定义 HelloService 接口，去使用 OpenFeign：



```
@FeignClient("provider")
public interface HelloService {

    @GetMapping("/hello")
    String hello();//这里的方法名无所谓，随意取
}
```

最后调用 HelloController 中，调用 HelloService 进行测试：

```
@RestController
public class HelloController {
    @Autowired
    HelloService helloService;

    @GetMapping("/hello")
    public String hello() {
        return helloService.hello();
    }
}
```

接下来，启动 OpenFeign 项目，进行测试。

## 9.2 参数传递

和普通参数传递的区别：

1. 参数一定要绑定参数名。
2. 如果通过 header 来传递参数，一定记得中文要转码。

测试的服务端接口，继续使用 provider 提供的接口。

这里，我们主要在 openfeign 中添加调用接口即可：

```
@FeignClient("provider")
public interface HelloService {

    @GetMapping("/hello")
    String hello();//这里的方法名无所谓，随意取

    @GetMapping("/hello2")
    String hello2(@RequestParam("name") String name);

    @PostMapping("/user2")
    User addUser(@RequestBody User user);

    @DeleteMapping("/user2/{id}")
    void deleteUserById(@PathVariable("id") Integer id);

    @GetMapping("/user3")
    void getUserByName(@RequestHeader("name") String name);
}
```

**注意，凡是 key/value 形式的参数，一定要标记参数的名称。**

HelloController 中调用 HelloService：

```

@GetMapping("/hello")
public String hello() throws UnsupportedEncodingException {
    String s = helloService.hello2("江南一点雨");
    System.out.println(s);
    User user = new User();
    user.setId(1);
    user.setUsername("javaboy");
    user.setPassword("123");
    User u = helloService.addUser(user);
    System.out.println(u);
    helloService.deleteUserById(1);
    helloService.getUserByName(URLEncoder.encode("江南一点雨", "UTF-8"));
    return helloService.hello();
}

```

注意：

放在 header 中的中文参数，一定要编码之后传递。

## 9.3 继承特性

将 provider 和 openfeign 中公共的部分提取出来，一起使用。

我们新建一个 Module，叫做 hello-api，注意，由于这个模块要被其他模块所依赖，所以这个模块是一个 Maven 项目，但是由于这个模块要用到 SpringMVC 的东西，因此在创建成功后，给这个模块添加一个 web 依赖，导入 SpringMVC 需要的一套东西。

项目创建成功后，首先添加依赖：

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.2.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.javaboy</groupId>
    <artifactId>commons</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>

```

然后定义公共接口，就是 provider 和 openfeign 中公共的部分：

```

public interface IUserService {
    @GetMapping("/hello")
    String hello(); // 这里的方法名无所谓，随意取

    @GetMapping("/hello2")
    String hello2(@RequestParam("name") String name);

    @PostMapping("/user2")
    User addUser(@RequestBody User user);

    @DeleteMapping("/user2/{id}")
    void deleteUserById(@PathVariable("id") Integer id);

    @GetMapping("/user3")
    void getUserByName(@RequestHeader("name") String name);
}

```

```
}
```

定义完成后，接下来，在 provider 和 openfeign 中，分别引用该模块：

```
<dependency>
  <groupId>org.javaboy</groupId>
  <artifactId>hello-api</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>
```

添加成功之后，在 provider 中实现该接口：

```
@RestController
public class HelloController implements IUserService {
    @Value("${server.port}")
    Integer port;

    @Override
    public String hello() {
        return "hello javaboy:" + port;
    }

    @Override
    public String hello2(String name) {
        System.out.println(new Date() + ">>>" + name);
        return "hello " + name;
    }

    @PostMapping("/user1")
    public User addUser1(User user) {
        return user;
    }

    @Override
    public User addUser2(@RequestBody User user) {
        return user;
    }

    @PutMapping("/user1")
    public void updateUser1(User user) {
        System.out.println(user);
    }

    @PutMapping("/user2")
    public void updateUser2(@RequestBody User user) {
        System.out.println(user);
    }

    @DeleteMapping("/user1")
    public void deleteUser1(Integer id) {
        System.out.println(id);
    }

    @Override
    public void deleteUser2(@PathVariable Integer id) {
        System.out.println(id);
    }
}
```

```

    }

    @Override
    public void getUserByName(@RequestHeader String name) throws
    UnsupportedEncodingException {
        System.out.println(URLDecoder.decode(name, "UTF-8"));
    }
}

```

在 openfeign 中，定义接口继承自公共接口：

```

@FeignClient("provider")
public interface HelloService extends IUserService {
}

```

接下来，测试代码不变。

关于继承特性：

1. 使用继承特性，代码简洁明了不易出错。服务端和消费端的代码统一，一改俱改，不易出错。这是优点也是缺点，这样会提高服务端和消费端的耦合度。
2. 9.2 中所讲的参数传递，在使用了继承之后，依然不变，参数该怎么传还是怎么传。

## 9.4 日志

OpenFeign 中，我们可以通过配置日志，来查看整个请求的调用过程。日志级别一共分为四种：

1. NONE：不开启日志，默认就是这个
2. BASIC：记录请求方法、URL、响应状态码、执行时间
3. HEADERS：在 BASIC 的基础上，加载请求/响应头
4. FULL：在 HEADERS 基础上，再增加 body 以及请求元数据。

四种级别，可以通过 Bean 来配置：

```

@SpringBootApplication
@EnableFeignClients
public class OpenfeignApplication {

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

    @Bean
    Logger.Level loggerLevel() {
        return Logger.Level.FULL;
    }
}

```

最后，在 application.properties 中开启日志级别：

```
logging.level.org.javaboy.openfeign.HelloService=debug
```

重启 OpenFeign，进行测试。

## 9.5 数据压缩

```
# 开启请求的数据压缩
feign.compression.request.enabled=true
# 开启响应的数据压缩
feign.compression.response.enabled=true
# 压缩的数据类型
feign.compression.request.mime-types=text/html,application/json
# 压缩的数据下限, 2048 表示当要传输的数据大于 2048 时, 才会进行数据压缩
feign.compression.request.min-request-size=2048
```

## 9.6 +Hystrix

Hystrix 中的容错、服务降级等功能, 在 OpenFeign 中一样要使用。

首先定义服务降级的方法:

```
@Component
@RequestMapping("/javaboy")//防止请求地址重复
public class HelloServiceFallback implements HelloService {
    @Override
    public String hello() {
        return "error";
    }

    @Override
    public String hello2(String name) {
        return "error2";
    }

    @Override
    public User addUser2(User user) {
        return null;
    }

    @Override
    public void deleteUser2(Integer id) {

    }

    @Override
    public void getUserByName(String name) throws UnsupportedOperationException {

    }
}
```

然后, 在 HelloService 中配置这个服务降级类:

```
@FeignClient(value = "provider", fallback = HelloServiceFallback.class)
public interface HelloService extends IUserService {
}
```

最后, 在 application.properties 中开启 Hystrix。

```
feign.hystrix.enabled=true
```

也可以通过自定义 FallbackFactory 来实现请求降级:

```

@Component
public class HelloServiceFallbackFactory implements
FallbackFactory<HelloService> {
    @Override
    public HelloService create(Throwable throwable) {
        return new HelloService() {
            @Override
            public String hello() {
                return "error---";
            }

            @Override
            public String hello2(String name) {
                return "error2---";
            }

            @Override
            public User addUser2(User user) {
                return null;
            }

            @Override
            public void deleteUser2(Integer id) {

            }

            @Override
            public void getUserByName(String name) throws
UnsupportedEncodingException {

            }
        };
    }
}

```

HelloService 中进行配置:

```

@FeignClient(value = "provider", fallbackFactory =
HelloServiceFallbackFactory.class)
public interface HelloService extends IUserService {
}

```

## 10. Resilience4j

### 10.1 Resilience4j 简介

Resilience4j 是 Spring Cloud Greenwich 版推荐的容错解决方案, 相比 Hystrix, Resilience4j 专为 Java8 以及函数式编程而设计。

Resilience4j 主要提供了如下功能:

1. 断路器
2. 限流
3. 基于信号量的隔离
4. 缓存

- 5. 限时
- 6. 请求重试

## 10.2 基本用法

首先搭建一个简单的测试环境。

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
</dependency>
```

### 10.2.1 断路器

Resilience4j 提供了很多功能，不同的功能对应不同的依赖，可以按需添加。

使用断路器，则首先添加断路器的依赖：

```
<dependency>
  <groupId>io.github.resilience4j</groupId>
  <artifactId>resilience4j-circuitbreaker</artifactId>
  <version>0.13.2</version>
</dependency>
```

一个正常执行的例子：

```
@Test
public void test1() {
    //获取一个CircuitBreakerRegistry实例，可以调用ofDefaults获取一个
    //CircuitBreakerRegistry实例，也可以自定义属性。
    CircuitBreakerRegistry registry = CircuitBreakerRegistry.ofDefaults();
    CircuitBreakerConfig config = CircuitBreakerConfig.custom()
        //故障率阈值百分比，超过这个阈值，断路器就会打开
        .failureRateThreshold(50)
        //断路器保持打开的时间，在到达设置的时间之后，断路器会进入到 half open 状态
        .waitDurationInOpenState(Duration.ofMillis(1000))
        //当断路器处于half open 状态时，环形缓冲区的大小
        .ringBufferSizeInHalfOpenState(2)
        .ringBufferSizeInClosedState(2)
        .build();
    CircuitBreakerRegistry r1 = CircuitBreakerRegistry.of(config);
    CircuitBreaker cb1 = r1.circuitBreaker("javaboy");
    CircuitBreaker cb2 = r1.circuitBreaker("javaboy2", config);
    CheckedFunction0<String> supplier =
    CircuitBreaker.decorateCheckedSupplier(cb1, () -> "hello resilience4
    Try<String> result = Try.of(supplier)
        .map(v -> v + " hello world");
    System.out.println(result.isSuccess());
    System.out.println(result.get());
}
```

一个出异常的断路器：

```
@Test
```

```

public void test2() {
    CircuitBreakerConfig config = CircuitBreakerConfig.custom()
        //故障率阈值百分比, 超过这个阈值, 断路器就会打开
        .failureRateThreshold(50)
        //断路器保持打开的时间, 在到达设置的时间之后, 断路器会进入到 half open 状态
        .waitDurationInOpenState(Duration.ofMillis(1000))
        //当断路器处于half open 状态时, 环形缓冲区的大小
        .ringBufferSizeInClosedState(2)
        .build();

    CircuitBreakerRegistry r1 = CircuitBreakerRegistry.of(config);
    CircuitBreaker cb1 = r1.circuitBreaker("javaboy");
    System.out.println(cb1.getState()); //获取断路器的一个状态
    cb1.onError(0, new RuntimeException());
    System.out.println(cb1.getState()); //获取断路器的一个状态
    cb1.onError(0, new RuntimeException());
    System.out.println(cb1.getState()); //获取断路器的一个状态
    CheckedFunction0<String> supplier =
    CircuitBreaker.decorateCheckedSupplier(cb1, () -> "hello resilience4j");
    Try<String> result = Try.of(supplier)
        .map(v -> v + " hello world");
    System.out.println(result.isSuccess());
    System.out.println(result.get());
}

```

注意, 由于 ringBufferSizeInClosedState 的值为 2, 表示当有两条数据时才会去统计故障率, 所以, 下面的手动故障测试, 至少调用两次 onError, 断路器才会打开。

### 10.2.2 限流

RateLimiter 本身和前面的断路器很像。

首先添加依赖:

```

<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-ratelimiter</artifactId>
    <version>0.13.2</version>
</dependency>

```

限流测试:

```

@Test
public void test3() {
    RateLimiterConfig config = RateLimiterConfig.custom()
        .limitRefreshPeriod(Duration.ofMillis(1000))
        .limitForPeriod(4)
        .timeoutDuration(Duration.ofMillis(1000))
        .build();

    RateLimiter rateLimiter = RateLimiter.of("javaboy", config);
    CheckedRunnable checkedRunnable =
    RateLimiter.decorateCheckedRunnable(rateLimiter, () -> {
        System.out.println(new Date());
    });

    Try.run(checkedRunnable)
        .andThenTry(checkedRunnable)
        .andThenTry(checkedRunnable)
        .andThenTry(checkedRunnable)

```



```
        .onFailure(t -> System.out.println(t.getMessage()));  
    }  
}
```

### 10.2.3 请求重试

首先第一步还是加依赖:

```
<dependency>  
  <groupId>io.github.resilience4j</groupId>  
  <artifactId>resilience4j-retry</artifactId>  
  <version>0.13.2</version>  
</dependency>
```

案例:

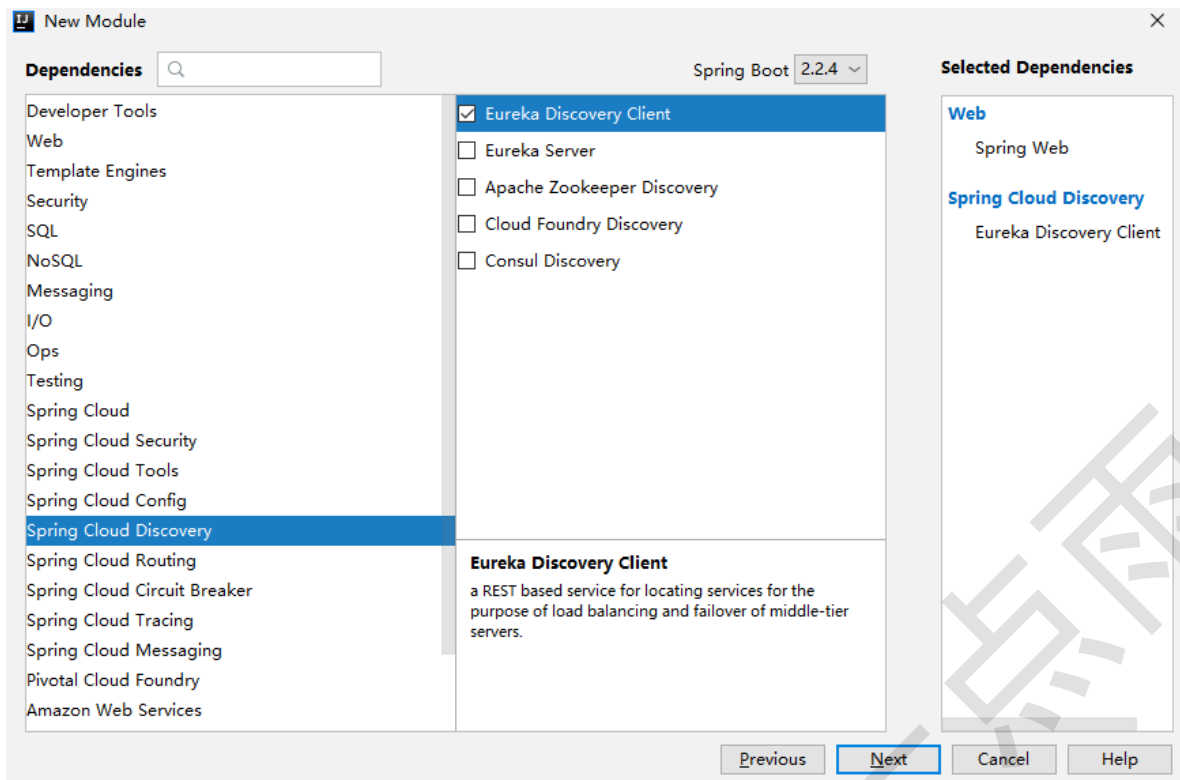
```
@Test  
public void test4() {  
    RetryConfig config = RetryConfig.custom()  
        //重试次数  
        .maxAttempts(2)  
        //重试间隔  
        .waitDuration(Duration.ofMillis(500))  
        //重试异常  
        .retryExceptions(RuntimeException.class)  
        .build();  
    Retry retry = Retry.of("javaboy", config);  
    Retry.decorateRunnable(retry, new Runnable() {  
        int count = 0;  
        //开启了重试功能之后, run 方法执行时, 如果抛出异常, 会自动触发重试功能  
        @Override  
        public void run() {  
            if (count++ < 3) {  
                throw new RuntimeException();  
            }  
        }  
    }).run();  
}
```

## 10.3 结合微服务

Retry、CircuitBreaker、RateLimiter

### 10.3.1 Retry

首先创建一个 Spring Boot 项目, 创建时, 添加 eureka-client 依赖, 使之能够注册到 eureka 上。



项目创建成功后，手动添加 Resilience4j 依赖：

```
<dependency>
  <groupId>io.github.resilience4j</groupId>
  <artifactId>resilience4j-spring-boot2</artifactId>
  <version>1.3.1</version>
  <exclusions>
    <exclusion>
      <groupId>io.github.resilience4j</groupId>
      <artifactId>resilience4j-circuitbreaker</artifactId>
    </exclusion>
    <exclusion>
      <groupId>io.github.resilience4j</groupId>
      <artifactId>resilience4j-ratelimiter</artifactId>
    </exclusion>
    <exclusion>
      <groupId>io.github.resilience4j</groupId>
      <artifactId>resilience4j-bulkhead</artifactId>
    </exclusion>
    <exclusion>
      <groupId>io.github.resilience4j</groupId>
      <artifactId>resilience4j-timelimiter</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

resilience4j-spring-boot2 中包含了 Resilience4j 的所有功能，但是没有配置的功能无法使用，需要将其从依赖中剔除掉。

接下来，在 application.yml 中配置 retry：

```
resilience4j:
  retry:
    retry-aspect-order: 399 # 表示Retry的优先级
    backends:
```

```

    retryA:
      maxRetryAttempts: 5 # 重试次数
      waitDuration: 500 # 重试等待时间
      exponentialBackoffMultiplier: 1.1 # 间隔乘数
      retryExceptions:
        - java.lang.RuntimeException
spring:
  application:
    name: resilience4j
server:
  port: 5000
eureka:
  client:
    service-url:
      defaultzone: http://localhost:1111/eureka

```

最后，创建测试 RestTemplate 和 HelloService：

```

@SpringBootApplication
public class Resilience4jApplication {

    public static void main(String[] args) {
        SpringApplication.run(Resilience4jApplication.class, args);
    }

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

    @Service
    @Retry(name = "retryA")//表示要使用的重试策略
    public class HelloService {
        @Autowired
        RestTemplate restTemplate;
        public String hello() {
            return restTemplate.getForObject("http://localhost:1113/hello",
String.class);
        }
    }

    @RestController
    public class HelloController {
        @Autowired
        HelloService helloService;
        @GetMapping("/hello")
        public String hello() {
            return helloService.hello();
        }
    }
}

```

### 10.3.2 CircuitBreaker

首先从依赖中删除排除 CircuitBreaker。

然后在 application.yml 中进行配置：

```

resilience4j:
  retry:
    retry-aspect-order: 399 # 表示Retry的优先级
    backends:
      retryA:
        maxRetryAttempts: 5 # 重试次数
        waitDuration: 500 # 重试等待时间
        exponentialBackoffMultiplier: 1.1 # 间隔乘数
        retryExceptions:
          - java.lang.RuntimeException
    circuitbreaker:
      instances:
        cbA:
          ringBufferSizeInClosedState: 5
          ringBufferSizeInHalfOpenState: 3
          waitInterval: 5000
          recordExceptions:
            - org.springframework.web.client.HttpServerErrorException
      circuit-breaker-aspect-order: 398

```

配置完成后，用 @CircuitBreaker 注解标记相关方法：

```

@Service
@CircuitBreaker(name = "cbA", fallbackMethod = "error")
public class HelloService {
    @Autowired
    RestTemplate restTemplate;

    public String hello() {
        return restTemplate.getForObject("http://localhost:1113/hello",
String.class);
    }

    public String error(Throwable t) {
        return "error";
    }
}

```

@CircuitBreaker 注解中的 name 属性用来指定 circuitbreaker 配置，fallbackMethod 属性用来指定服务降级的方法，需要注意的是，服务降级方法中，要添加异常参数。

### 10.3.3 RateLimiter

RateLimiter 作为限流工具，主要在服务端使用，用来保护服务端的接口。

首先在 provider 中添加 RateLimiter 依赖：

```

<dependency>
  <groupId>io.github.resilience4j</groupId>
  <artifactId>resilience4j-spring-boot2</artifactId>
  <version>1.2.0</version>
  <exclusions>
    <exclusion>
      <groupId>io.github.resilience4j</groupId>
      <artifactId>resilience4j-circuitbreaker</artifactId>
    </exclusion>
  </exclusions>
</dependency>

```

```

        <exclusion>
            <groupId>io.github.resilience4j</groupId>
            <artifactId>resilience4j-bulkhead</artifactId>
        </exclusion>
        <exclusion>
            <groupId>io.github.resilience4j</groupId>
            <artifactId>resilience4j-timelimiter</artifactId>
        </exclusion>
    </exclusions>
</dependency>

```

接下来，在 provider 的 application.properties 配置文件中，去配置 RateLimiter：

```

# 这里配置每秒钟处理一个请求
resilience4j.ratelimiter.limiters.r1A.limit-for-period=1
resilience4j.ratelimiter.limiters.r1A.limit-refresh-period=1s
resilience4j.ratelimiter.limiters.r1A.timeout-duration=1s

```

为了查看请求效果，在 provider 的 HelloController 中打印每一个请求的时间：

```

@Override
@RateLimiter(name = "r1A")
public String hello() {
    String s = "hello javaboy:" + port;
    System.out.println(new Date());
    return s;
}

```

这里通过 @RateLimiter 注解来标记该接口限流。

配置完成后，重启 provider。

然后，在客户端模拟多个请求，查看限流效果：

```

public String hello() {
    for (int i = 0; i < 5; i++) {
        restTemplate.getForObject("http://localhost:1113/hello", String.class);
    }
    return "success";
}

```

## 10.4 服务监控

微服务由于服务数量众多，所以出故障的概率很大，这种时候不能单纯的依靠人肉运维。

早期的 Spring Cloud 中，服务监控主要使用 Hystrix Dashboard，集群数据库监控使用 Turbine。

在 Greenwich 版本中，官方建议监控工具使用 Micrometer。

Micrometer：

1. 提供了度量指标，例如 timers、counters
2. 一揽子开箱即用的解决方案，例如缓存、类加载器、垃圾收集等等

新建一个 Spring Boot 项目，添加 Actuator 依赖。项目创建成功后，添加如下配置，开启所有端点：

```
management.endpoints.web.exposure.include=*
```

然后就可以在浏览器查看项目的各项运行数据，但是这些数据都是 JSON 格式。

Console Endpoints	
Beans Health Mappings	
Path	Method
/**	
/actuator [GET]	Actuator root web endpoint
/actuator/beans [GET]	Actuator web endpoint 'beans'
/actuator/caches [GET]	Actuator web endpoint 'caches'
/actuator/caches [DELETE]	Actuator web endpoint 'caches'
/actuator/caches/{cache} [GET]	Actuator web endpoint 'caches'
/actuator/caches/{cache} [DELETE]	Actuator web endpoint 'caches-cache'
/actuator/conditions [GET]	Actuator web endpoint 'conditions'
/actuator/configprops [GET]	Actuator web endpoint 'configprops'
/actuator/env [GET]	Actuator web endpoint 'env'
/actuator/env/{toMatch} [GET]	Actuator web endpoint 'env-toMatch'
/actuator/health [GET]	Actuator web endpoint 'health'
/actuator/health/** [GET]	Actuator web endpoint 'health-path'
/actuator/heapdump [GET]	Actuator web endpoint 'heapdump'
/actuator/info [GET]	Actuator web endpoint 'info'
/actuator/loggers [GET]	Actuator web endpoint 'loggers'
/actuator/loggers/{name} [POST]	Actuator web endpoint 'loggers-name'
/actuator/loggers/{name} [GET]	Actuator web endpoint 'loggers-name'

我们需要一个可视化工具来展示这些 JSON 数据。这里主要和大家介绍 Prometheus。

## 10.4.1 Prometheus

### 安装

```
wget
https://github.com/prometheus/prometheus/releases/download/v2.16.0/prometheus-2.16.0.linux-amd64.tar.gz
tar -zxvf prometheus-2.16.0.linux-amd64.tar.gz
```

解压完成后，配置一下数据路径和要监控的服务地址：

```
cd prometheus-2.16.0.linux-amd64/
vi prometheus.yml
```

修改 prometheus.yml 配置文件，主要改两个地方，一个是数据接口，另一个是服务地址：

```
# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
      - targets:
        # - alertmanager:9093

# Load rules once and periodically evaluate them according to the global 'evaluation_interval'.
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label 'job=<job_name>' to any timeseries scraped from this config.
  - job_name: 'prometheus'
    metrics_path: '/actuator/prometheus'
    scrape_interval: 5s

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ['192.168.91.1:8080']
```

监控的地址

每隔5秒抓取一次数据

要监控的服务地址

接下来，将 Prometheus 整合到 Spring Boot 项目中。

首先加依赖：

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

然后在 application.properties 配置中，添加 Prometheus 配置：

```
management.endpoints.web.exposure.include=*
management.endpoint.prometheus.enabled=true
management.metrics.export.prometheus.enabled=true
management.endpoint.metrics.enabled=true
```

接下来启动 Prometheus。

启动命令：

```
./prometheus --config.file=prometheus.yml
```

启动成功后，浏览器输入 <http://192.168.91.128:9090> 查看 Prometheus 数据信息。

Grafana: <https://grafana.com/grafana/download?platform=linux>

## 11. Zuul

### 11.1 服务网关

Zuul 和 Gateway

由于每一个微服务的地址都有可能发生变化，无法直接对外公布这些服务地址，基于安全以及高内聚低耦合等设计，我们有必要将内部系统和外部系统做一个切割。

一个专门用来处理外部请求的组件，就是服务网关。

- 权限问题统一处理
- 数据剪裁和聚合

- 简化客户端的调用
- 可以针对不同的客户端提供不同的网关支持

Spring Cloud 中，网关主要有两种实现方案：

- Zuul
- Spring Cloud Gateway

## 11.2 Zuul

Zuul 是 Netflix 公司提供的网关服务。

Zuul 的功能：

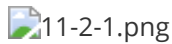
- 权限控制，可以做认证和授权
- 监控
- 动态路由
- 负载均衡
- 静态资源处理

Zuul 中的功能基本上都是基于过滤器来实现，它的过滤器有几种不同的类型：

- PRE
- ROUTING
- POST
- ERROR

### 11.2.1 HelloWorld

首先创建项目，添加 Zuul 依赖。



项目创建成功后，将 zuul 注册到 eureka 上：

```
spring.application.name=zuul
server.port=2020
eureka.client.service-url.defaultZone=http://localhost:1111/eureka
```

然后在启动类上开启网关代理：

```
@SpringBootApplication
@EnableZuulProxy //开启网关代理
public class ZuulApplication {

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

}
```

配置完成后，重启 Zuul，接下来，在浏览器中，通过 Zuul 的代理就可以访问到 provider 了。

<http://localhost:2020/provider/hello>

在这个访问地址中，provider 就是要访问的服务名称，/hello 则是要访问的服务接口。

这是一个简单例子，Zuul 中的路由规则也可以自己配置。



```
zuul.routes.javaboy-a.path=/javaboy-a/**
zuul.routes.javaboy-a.service-id=provider
```

上面这个配置，表示 /javaboy-a/\*\*，满足这个匹配规则的请求，将被转发到 provider 实例上。

上面两行配置，也可以进行简化：

```
zuul.routes.provider=/javaboy-a/**
```

### 11.2.2 请求过滤

对于来自客户端的请求，可以在 Zuul 中进行预处理，例如权限判断等。

定义一个简单的权限过滤器：

```
@Component
public class PermissFilter extends ZuulFilter {
    /**
     * 过滤器类型，权限判断一般是 pre
     * @return
     */
    @Override
    public String filterType() {
        return "pre";
    }

    /**
     * 过滤器优先级
     * @return
     */
    @Override
    public int filterOrder() {
        return 0;
    }

    /**
     * 是否过滤
     * @return
     */
    @Override
    public boolean shouldFilter() {
        return true;
    }

    /**
     * 核心的过滤逻辑写在这里
     * @return 这个方法虽然有返回值，但是这个返回值目前无所谓
     * @throws ZuulException
     */
    @Override
    public Object run() throws ZuulException {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest(); //获取当前请求
        String username = request.getParameter("username");
        String password = request.getParameter("password");
        if (!"javaboy".equals(username) || !"123".equals(password)) {
```

```

        //如果请求条件不满足的话，直接从这里给出响应
        ctx.setSendZuulResponse(false);
        ctx.setResponseStatusCode(401);
        ctx.addZuulResponseHeader("content-type","text/html;charset=utf-8");
        ctx.setResponseBody("非法访问");
    }
    return null;
}
}

```

重启 Zuul，接下来，发送请求必须带上 username 和 password 参数，否则请求不通过。

<http://localhost:2020/javaboy-a/hello?username=javaboy&password=123>

### 11.2.3 Zuul 中的其他配置

#### 匹配规则

例如有两个服务，一个叫 consumer，另一个叫 consumer-hello，在做路由规则设置时，假如出现了如下配置：

```

zuul.routes.consumer=/consumer/**
zuul.routes.consumer-hello=/consumer/hello/**

```

此时，如果访问一个地址：<http://localhost:2020/consumer/hello/123>，会出现冲突。实际上，这个地址是希望和 consumer-hello 这个服务匹配的，这个时候，只需要把配置文件改为 yml 格式就可以了。

#### 忽略路径

默认情况下，zuul 注册到 eureka 上之后，eureka 上的所有注册服务都会被自动代理。如果不想给某一个服务做代理，可以忽略该服务，配置如下：

```

zuul.ignored-services=provider

```

上面这个配置表示忽略 provider 服务，此时就不会自动代理 provider 服务了。

也可以忽略某一类地址：

```

zuul.ignored-patterns=/**/hello/**

```

这个表示请求路径中如果包含 hello，则不做代理。

#### 前缀

也可以给路由加前缀。

```

zuul.prefix=/javaboy

```

这样，以后所有的请求地址自动多了前缀，/javaboy

## 12. Gateway

### 12.1 简介

特点：

- 限流
- 路径重写
- 动态路由
- 集成 Spring Cloud DiscoveryClient
- 集成 Hystrix 断路器

和 Zuul 对比：

1. Zuul 是 Netflix 公司的开源产品，Spring Cloud Gateway 是 Spring 家族中的产品，可以和 Spring 家族中的其他组件更好的融合。
2. Zuul1 不支持长连接，例如 websocket。
3. Spring Cloud Gateway 支持限流。
4. Spring Cloud Gateway 基于 Netty 来开发，实现了异步和非阻塞，占用资源更小，性能强于 Zuul。

## 12.2 基本用法

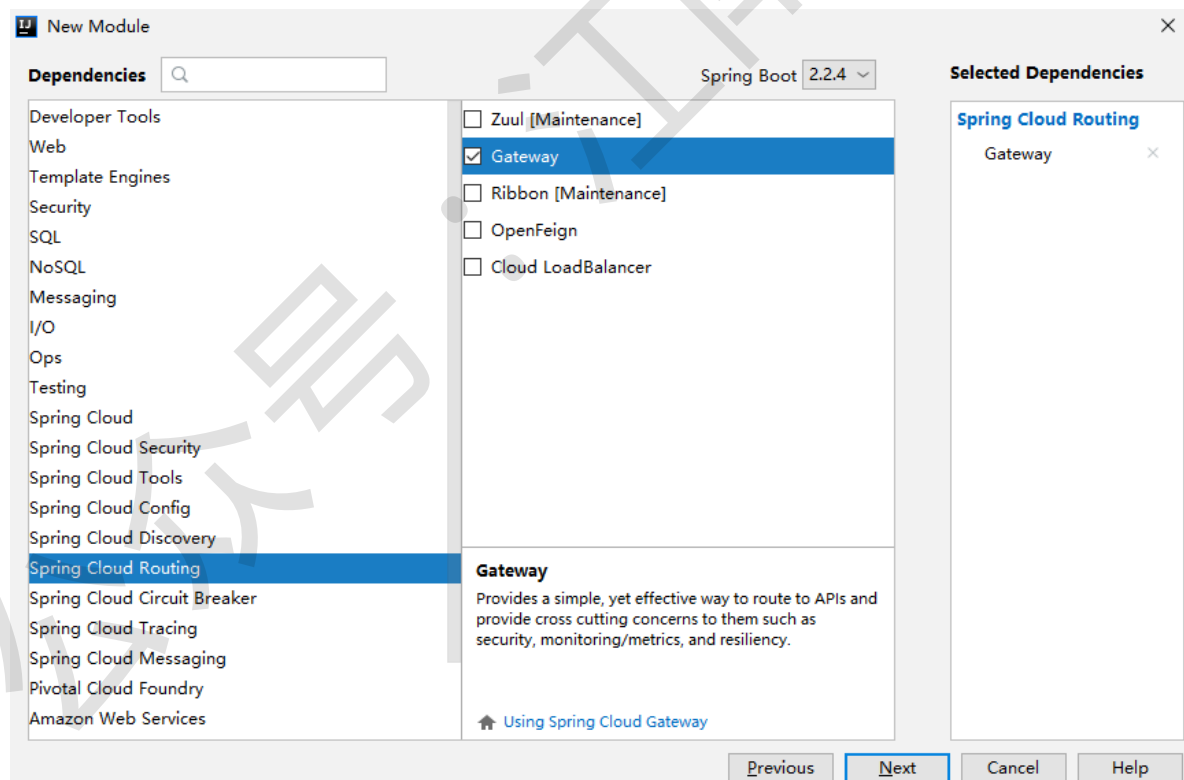
Spring Cloud Gateway 支持两种不同的用法：

- 编码式
- yml 配置

两种都来看下。

### 编码式

首先创建 Spring Boot 项目，添加 Spring Cloud Gateway 模块：



项目创建成功后，直接配置一个 RouteLocator 这样一个 Bean，就可以实现请求转发。

```

@Bean
RouteLocator routeLocator(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("javaboy_route", r ->
            r.path("/get").uri("http://httpbin.org"))
        .build();
}

```

这里只需要提供 RouteLocator 这个 Bean，就可以实现请求转发。配置完成后，重启项目，访问：<http://localhost:8080/get>

### properties 配置

```

spring.cloud.gateway.routes[0].id=javaboy_route
spring.cloud.gateway.routes[0].uri=http://httpbin.org
spring.cloud.gateway.routes[0].predicates[0]=Path=/get

```

### YML 配置

```

spring:
  cloud:
    gateway:
      routes:
        - id: javaboy_route
          uri: http://httpbin.org
          predicates:
            - Path=/get

```

## 12.2.1 服务化

首先给 Gateway 添加依赖，将之注册到 Eureka 上。

加依赖：

```

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

```

加配置：

```

spring:
  cloud:
    gateway:
      routes:
        - id: javaboy_route
          uri: http://httpbin.org
          predicates:
            - Path=/get
    application:
      name: gateway
  eureka:
    client:
      service-url:
        defaultZone: http://localhost:1111/eureka

```

配置路由转发：

```
spring:
  cloud:
    gateway:
      routes:
        - id: javaboy_route
          uri: http://httpbin.org
          predicates:
            - Path=/get
      discovery:
        locator:
          enabled: true # 开启自动代理
    application:
      name: gateway
  eureka:
    client:
      service-url:
        defaultzone: http://localhost:1111/eureka
  logging:
    level:
      org.springframework.cloud.gateway: debug
```

接下来，就可以通过 Gateway 访问到其他注册在 Eureka 上的服务了，访问方式和 Zuul 一样。

## 12.3 Predicate

通过时间匹配：

```
spring:
  cloud:
    gateway:
      routes:
        - id: javaboy_route
          uri: http://httpbin.org
          predicates:
            - After=2021-01-01T01:01:01+08:00[Asia/Shanghai]
```

表示，请求时间在 2021-01-01T01:01:01+08:00[Asia/Shanghai] 时间之后，才会被路由。

除了 After 之外，还有两个关键字：

- Before，表示在某个时间点之前进行请求转发
- Between，表示在两个时间点之间，两个时间点用，隔开

也可以通过请求方式匹配，就是请求方法：

```
spring:
  cloud:
    gateway:
      routes:
        - id: javaboy_route
          uri: http://httpbin.org
          predicates:
            - Method=GET
```

这个配置表示只给 GET 请求进行路由。

通过请求路径匹配：

```
spring:
  cloud:
    gateway:
      routes:
        - id: javaboy_route
          uri: http://www.javaboy.org
          predicates:
            - Path=/2019/0612/{segment}
```

表示路径满足 /2019/0612/ 这个规则，都会被进行转发，例如：

<http://www.javaboy.org/2019/0612/git-install.html>

<http://www.javaboy.org/2019/0612/git-basic.html>

通过参数进行匹配：

```
spring:
  cloud:
    gateway:
      routes:
        - id: javaboy_route
          uri: http://httpbin.org
          predicates:
            - Query=name
```

表示请求中一定要有 name 参数才会进行转发，否则不会进行转发。

也可以指定参数和参数的值。

例如参数的 key 为 name，value 必须要以 java 开始：

```
spring:
  cloud:
    gateway:
      routes:
        - id: javaboy_route
          uri: http://httpbin.org
          predicates:
            - Query=name,java.*
```

多种匹配方式也可以组合使用。

```
spring:
  cloud:
    gateway:
      routes:
        - id: javaboy_route
          uri: http://httpbin.org
          predicates:
            - Query=name,java.*
            - Method=GET
            - After=2021-01-01T01:01:01+08:00[Asia/Shanghai]
```

## 12.4 Filter

Spring Cloud Gateway 中的过滤器分为两大类：

- GatewayFilter
- GlobalFilter

AddRequestParameter 过滤器使用：

```
spring:
  cloud:
    gateway:
      routes:
        - id: javaboy_route
          uri: lb://provider
          filters:
            - AddRequestParameter=name,javaboy
          predicates:
            - Method=GET
```

这个过滤器就是在请求转发路由的时候，自动额外添加参数。

## 13. 分布式配置中心

### 13.1 基本用法

分布式配置中心解决方案：

国内：

- 360: QConf
- 淘宝: diamond
- 百度: disconf

国外：

- Apache Commons Configuration
- owner
- cfg4j

#### 13.1.1 简介

Spring Cloud Config 是一个分布式系统配置管理的解决方案，它包含了 Client 和 Server 。配置文件放在 Server 端，通过 接口的形式提供给 Client。

Spring Cloud Config 主要功能：

- 集中管理各个环境、各个微服务的配置文件
- 提供服务端和客户端支持
- 配置文件修改后，可以快速生效
- 配置文件通过 Git/SVn 进行管理，天然支持版本回退功能。
- 支持高并发查询、也支持多种开发语言。

公众号【江南一点雨】，底部菜单，有一个教程合集 -> Git教程

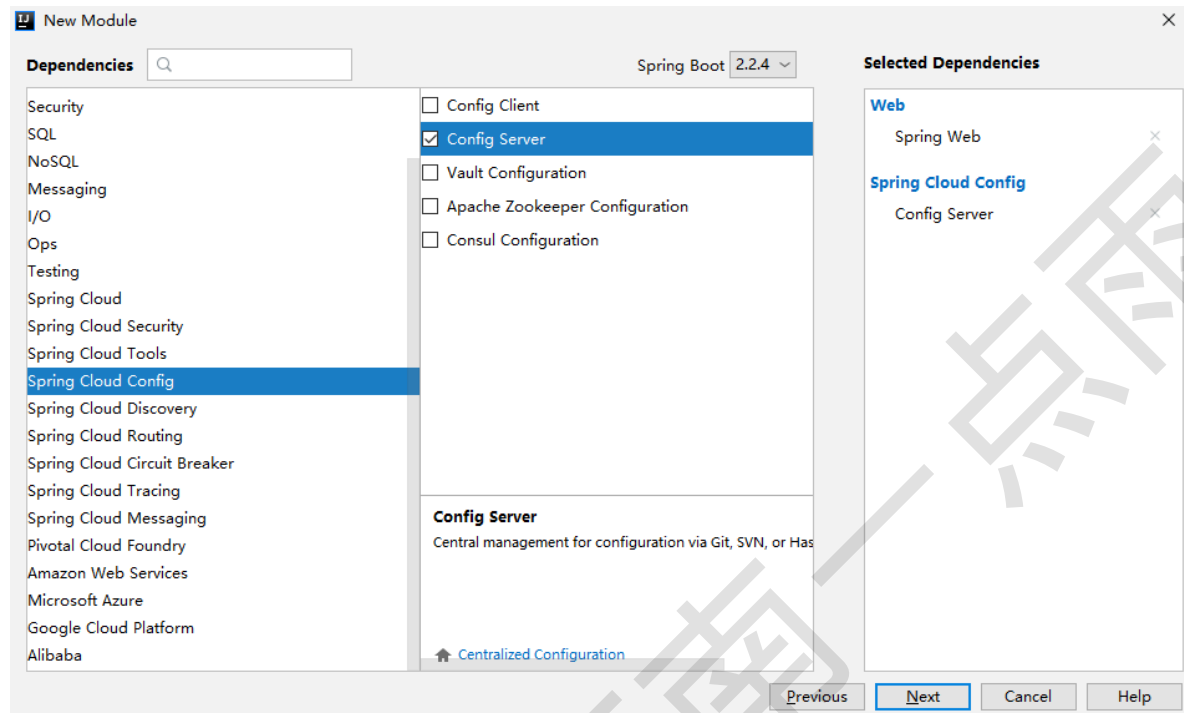
#### 13.1.2 准备工作

准备工作主要是给 GitHub 上提交数据。

本地准备好相应的配置文件，提交到 GitHub: <https://github.com/wongsung/configRepo>

### 13.1.2 ConfigServer

首先创建一个 ConfigServer 工程，创建时添加 ConfigServer 依赖：



项目创建成功后，项目启动类上添加注解，开启 config server 功能：

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {

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

}
```

然后在配置文件中配置仓库的基本信息：

```
spring.application.name=config-server
server.port=8081
# 配置文件仓库地址
spring.cloud.config.server.git.uri=https://github.com/wongsung/configRepo.git
# 仓库中，配置文件的目录
spring.cloud.config.server.git.search-paths=client1
# 仓库的用户名密码
spring.cloud.config.server.git.username=1510161612@qq.com
spring.cloud.config.server.git.password=
```

启动项目后，就可以访问配置文件了。访问地址: <http://localhost:8081/client1/prod/master>

实际上，访问地址有如下规则：



/ {application} / {profile} / [{label}]  
/ {application} - {profile}. yml  
/ {application} - {profile}. properties  
/ {label} / {application} - {profile}. yml  
/ {label} / {application} - {profile}. properties

applicaiton 表示配置文件名

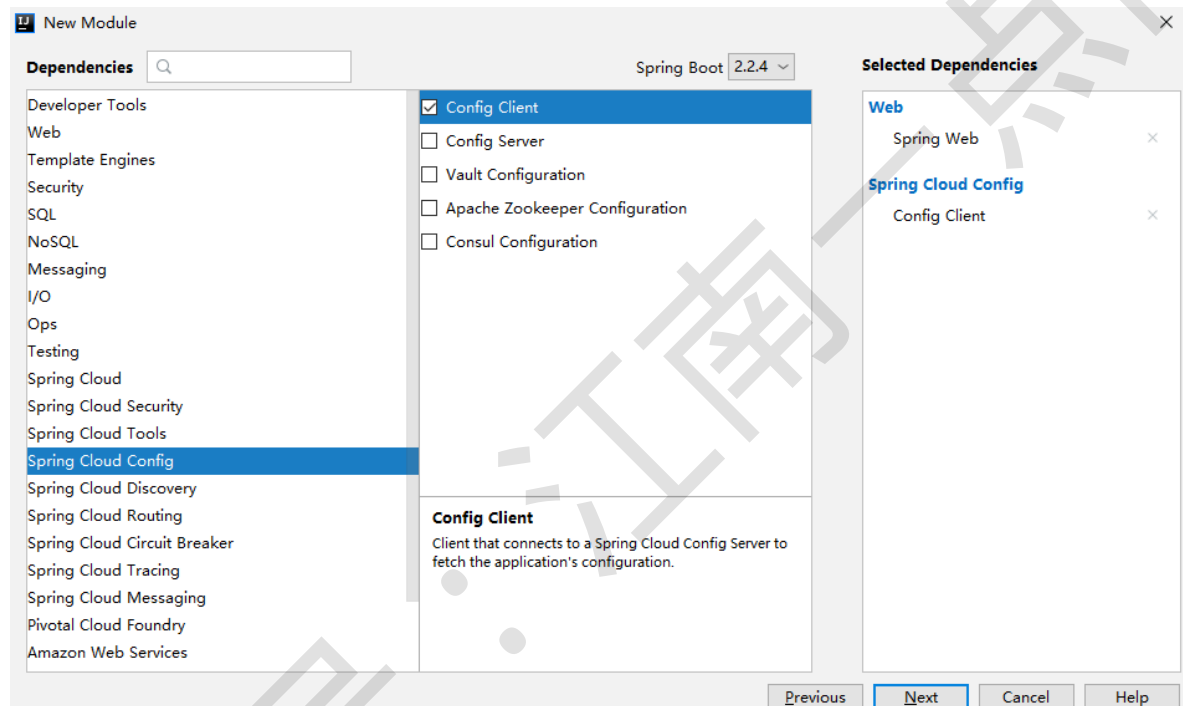
profile 表示配置文件 profile, 例如 test、dev、prod

label 表示git 分支, 参数可选, 默认就是master

接下来, 可以修改配置文件, 并且重新提交到 GitHub, 此时, 刷新 ConfigServer 接口, 就可以及时看到最新的配置内容。

### 13.1.3 ConfigClient

创建一个 Spring Boot 项目, 添加 ConfigClient 依赖:



项目创建成功后, resources 目录下, 添加 bootstrap.properties 配置, 内容如下:

```
# 下面三行配置, 分别对应 config-server 中的 {application}、{profile}以及{label}占位符
spring.application.name=client1
spring.cloud.config.profile=dev
spring.cloud.config.label=master
spring.cloud.config.uri=http://localhost:8080
server.port=8082
```

接下来创建一个 HelloController 进行测试:

```
@RestController
public class HelloController {
    @Value("${javaboy}")
    String javaboy;
    @GetMapping("/hello")
    public String hello() {
        return javaboy;
    }
}
```

### 13.1.4 配置

使用占位符灵活控制查询目录。

修改 config-server 配置文件：

```
spring.cloud.config.server.git.search-paths={application}
```

这里的 {application} 占位符，表示链接上来的 client1 的 spring.application.name 属性的值。

在 config-server 中，也可以用 {profile} 表示 client 的 spring.cloud.config.profile，也可以用 {label} 表示 client 的 spring.cloud.config.label

虽然在实际开发中，配置文件一般都是放在 Git 仓库中，但是，config-server 也支持将配置文件放在 classpath 下。

在 config-server 中添加如下配置：

```
# 表示让 config-server 从 classpath 下查找配置，而不是去 Git 仓库中查找
spring.profiles.active=native
```

也可以在 config-server 中，添加如下配置，表示指定配置文件的位置：

```
spring.cloud.config.server.native.search-locations=file:/E:/properties/
```

## 13.2 配置文件加解密

### 13.2.1 常见加密方案

- 不可逆加密
- 可逆加密

不可逆加密，就是理论上无法根据加密后的密文推算出明文。一般用在密码加密上，常见的算法如 MD5 消息摘要算法、SHA 安全散列算法。

可逆加密，看名字就知道可以根据加密后的密文推断出明文的加密方式，可逆加密一般又分为两种：

- 对称加密
- 非对称加密

对称加密指加密的密钥和解密的密钥是一样的。常见算法 des、3des、aes

非对称加密就是加密的密钥和解密的密钥不一样，加密的叫做公钥，可以告诉任何人，解密的叫做私钥，只有自己知道。常见算法 RSA。

### 13.2.2 对称加密

首先下载不限长度的 JCE: [http://download.oracle.com/otn-pub/java/jce/8/jce\\_policy-8.zip](http://download.oracle.com/otn-pub/java/jce/8/jce_policy-8.zip)

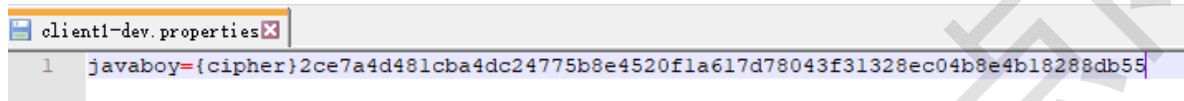
将下载的文件解压，解压出来的 jar 拷贝到 Java 安装目录中：C:\Program Files\Java\jdk-13.0.1\lib\security

然后，在 config-server 的 bootstrap.properties 配置文件中，添加如下内容配置密钥：

```
# 密钥
encrypt.key=javaboy
```

然后，启动 config-server，访问如下地址，查看密钥配置是否OK: <http://localhost:8081/encrypt/status>

然后，访问: <http://localhost:8081/encrypt>，注意这是一个 POST 请求，访问该地址，可以对一段明文进行加密。把加密后的明文存储到 Git 仓库中，存储时，要注意加一个 {cipher} 前缀。



### 13.2.3 非对称加密

非对称加密需要我们首先生成一个密钥对。

在命令行执行如下命令，生成 keystore：

```
keytool -genkeypair -alias config-server -keyalg RSA -keystore
D:\springcloud\config-server.keystore
```

命令执行完成后，拷贝生成的 keystore 文件到 config-server 的 resources 目录下。

然后在 config-server 的 bootstrap.properties 目录中，添加如下配置：

```
encrypt.key-store.location=config-server.keystore
encrypt.key-store.alias=config-server
encrypt.key-store.password=111111
encrypt.key-store.secret=111111
```

重启 config-server，测试方法与对称加密一致。

注意，在 pom.xml 的 build 节点中，添加如下配置，防止 keystore 文件被过滤掉。

```
<resources>
  <resource>
    <directory>src/main/resources</directory>
    <includes>
      <include>**/*.properties</include>
      <include>**/*.keystore</include>
    </includes>
  </resource>
</resources>
```

## 13.3 安全管理

防止用户直接通过访问 config-server 看到配置文件内容，我们可以用 spring security 来保护 config-server 接口。

首先在 config-server 中添加 spring security 依赖：

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

添加完依赖后，config-server 中的接口就自动被保护起来了。

默认自动生成的密码不好记，所以我们可以自己在 config-server 中，自己配置用户名密码。

在 config-server 的配置文件中，添加如下配置，固定用户名密码：

```
spring.security.user.name=javaboy
spring.security.user.password=123
```

然后，在 config-client 的 bootstrap.properties 配置文件中，添加如下配置：

```
spring.cloud.config.username=javaboy
spring.cloud.config.password=123
```

## 13.4 服务化

前面的配置都是直接在 config-client 中写死 config-server 的地址。

首先启动 Eureka。

然后，为了让 config-server 和 config-client 都能注册到 Eureka，给它俩添加如下依赖：

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

然后，在 application.properties 配置文件中配置注册信息。

```
eureka.client.service-url.defaultZone=http://localhost:1111/eureka
```

然后，修改 config-client 的配置文件，不再直接写死 config-server 的地址了。

```
# 下面三行配置，分别对应 config-server 中的 {application}、{profile}以及{label}占位符
spring.application.name=client1
spring.cloud.config.profile=dev
spring.cloud.config.label=master
#spring.cloud.config.uri=http://localhost:8081
# 开启通过 eureka 获取 config-server 的功能
spring.cloud.config.discovery.enabled=true
# 配置 config-server 服务名称
spring.cloud.config.discovery.service-id=config-server
server.port=8082

spring.cloud.config.username=javaboy
spring.cloud.config.password=123

eureka.client.service-url.defaultZone=http://localhost:1111/eureka
```

注意，加入 eureka client 之后，启动 config-server 可能会报错，此时，我们重新生成一个 jks 格式的密钥。

```
keytool -genkeypair -alias mytestkey -keyalg RSA -keypass 111111 -keystore  
D:\springcloud\config-service.jks -storepass 111111
```

生成之后，拷贝到 configserver 的 resources 目录下，同时修改 bootstrap.properties 配置。

```
# 密钥  
#encrypt.key=javaboy  
  
encrypt.key-store.location=classpath:config-service.jks  
encrypt.key-store.alias=mytestkey  
encrypt.key-store.password=111111  
encrypt.key-store.secret=111111  
  
spring.security.user.name=javaboy  
spring.security.user.password=123
```

同时也修改一个 pom.xml 中的过滤条件：

```
<resources>  
  <resource>  
    <directory>src/main/resources</directory>  
    <includes>  
      <include>**/*.properties</include>  
      <include>**/*.jks</include>  
    </includes>  
  </resource>  
</resources>
```

## 13.5 动态刷新

当配置文件发生变化之后，config-server 可以及时感知到变化，但是 config-client 不会及时感知到变化，默认情况下，config-client 只有重启才能加载到最新的配置文件。

首先给 config-client 添加如下依赖：

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

然后，添加配置，使 refresh 端点暴露出来：

```
management.endpoints.web.exposure.include=refresh
```

最后，再给 config-client 使用了配置文件的地方加上 @RefreshScope 注解，这样，当配置改变后，只需要调用 refresh 端点，config-client 中的配置就可以自动刷新。

```

@RestController
@RefreshScope
public class HelloController {
    @Value("${javaboy}")
    String javaboy;
    @GetMapping("/hello")
    public String hello() {
        return javaboy;
    }
}

```

重启 config-client, 以后, 只要配置文件发生变化, 发送 POST 请求, 调用 <http://localhost:8082/actor/refresh> 接口即可, 配置文件就会自动刷新。

## 13.6 请求失败重试

config-client 在调用 config-server 时, 一样也可能发生请求失败的问题, 这个时候, 我们可以配置一个请求重试的功能。

要给 config-client 添加重试功能, 只需要添加如下依赖即可:

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
</dependency>

```

然后, 修改配置文件, 开启失败快速响应。

```

# 开启失败快速响应
spring.cloud.config.fail-fast=true

```

然后, 注释掉配置文件的用户名密码, 重启 config-client, 此时加载配置文件失败, 就会自动重试。

也可以通过如下配置保证服务的可用性:

```

# 开启失败快速响应
spring.cloud.config.fail-fast=true
# 请求重试的初识间隔时间
spring.cloud.config.retry.initial-interval=1000
# 最大重试次数
spring.cloud.config.retry.max-attempts=6
# 重试时间间隔乘数
spring.cloud.config.retry.multiplier=1.1
# 最大间隔时间
spring.cloud.config.retry.max-interval=2000

```

## 14.Spring Cloud Bus

Spring Cloud Bus 通过轻量级的消息代理连接各个微服务，可以用来广播配置文件的更改，或者管理服务监控。

安装 RabbitMQ。

Docker 中 RabbitMQ 安装命令：

```
docker run -d --hostname my-rabbit --name some-rabbit -p 15672:15672 5672:5672
rabbitmq:3-management
```

首先给 config-server 和 config-client 分别加上 Spring Cloud Bus 依赖。

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

然后，给两个分别配置，使之都连接到 RabbitMQ 上：

```
spring.rabbitmq.host=192.168.91.128
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

同时，由于 configserver 将提供刷新接口，所以给 configserver 加上 actuator 依赖：

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

然后记得在 config-server 中，添加开启 bus-refresh 端点：

```
management.endpoints.web.exposure.include=bus-refresh
```

由于给 config-server 中的所有接口都添加了保护，所以刷新接口将无法直接访问，此时，可以通过修改 Security 配置，对端点的权限做出修改：

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .httpBasic()
            .and()
            .csrf().disable();
    }
}
```

在这段配置中，开启了 HttpBasic 登录，这样，在发送刷新请求时，就可以直接通过 HttpBasic 配置认证信息了。

最后分别启动 config-server 和 config-client，然后修改配置信息提交到 GitHub，刷新 config-client 接口，查看是否有变化。

然后，发送如下 POST 请求：<http://localhost:8081/actuator/bus-refresh>

这个 post 是针对 config-server 的，config-server 会把这个刷新的指令传到 rabbitmq，然后 rabbitmq 再把指令传给 各个 client。

## 逐个刷新

如果更新配置

公众号：江舞一点墨